MASTER'S THESIS 2021

Identification of Technical Debt in Code using Software Metrics

Erica Schillström, Dan Wahlin

ISSN 1650-2884
LU-CS-EX: 2021-22

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2021-22

Identification of Technical Debt in Code using Software Metrics

Identifiering av teknisk skuld i kod med mjukvarumätvärden

Erica Schillström, Dan Wahlin

Identification of Technical Debt in Code using Software Metrics

Erica Schillström
er8884sc-s@student.lu.se

Dan Wahlin
da8647wa-s@student.lu.se

June 21, 2021

Master's thesis work carried out at IKEA IT AB.

Supervisors: Sandra Åström, sandra.astrom1@ingka.ikea.com
Martin Höst, martin.host@cs.lth.se

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

Abstract

The metaphor Technical Debt describes the consequences of taking shortcuts in the software development process for short-term benefit, at the expense of higher maintenance in the future. Every large software system contains Technical Debt in some way or another, the difficult question is to know when, where, and how to repay the debt.

To answer this we conducted three main steps. A literature study on how to identify, measure, and manage Technical Debt (i), interviews (ii), and finding key software metrics based on the Goal-Question-Metric paradigm to build a Technical Debt Model with a visualisation dashboard (iii). This was used for a case study at an organisation where files from two projects were analysed and the model was evaluated by comparing the results with opinions from developers who created the system.

The final model used 8 metrics as input and had a total success rate of 80% when comparing the model's ranking with mutual opinion of the developers.

The dashboard was seen to be a useful tool for discussion and a quick way to identify areas in the code with Technical Debt. It was also discovered that measuring Technical Debt with only software metrics is a hard to impossible task, and many sources of information must be used in conjunction. To circumvent this, an implementation model for the organisation using several information channels together with the dashboard was created.

Keywords: Software Development, Technical Debt, Software Metrics

Acknowledgements

Firstly, we would like to thank IKEA Retail for the opportunity to complete our master's thesis with them. Specifically, a big thank you to Sandra Åström for helping us throughout the project and always making us feel welcome, and to Magnus Pettersson for valuable feedback and encouragement.

Secondly, we would like to send a big thank you to the rest of the team for their participation in the interviews and the evaluation.

Finally, we want to thank Martin Höst for being a supportive supervisor and giving us great advice with his expertise in the area.

Contents

1 Introduction **11**
1.1 Problem Statement . . . 11
1.2 Research Questions . . . 11
1.3 Contribution . . . 12
1.4 Distribution of Work . . . 12
1.5 Outline . . . 12

2 Background and Related Work **13**
2.1 Theory . . . 13
2.1.1 Technical Debt . . . 13
2.1.2 Software Metrics . . . 15
2.1.3 Agile Software Development . . . 17
2.2 Technical Debt Management . . . 18
2.2.1 Working with Technical Debt . . . 18
2.2.2 Best Practises to Reduce Technical Debt . . . 24
2.3 Case Company . . . 28

3 Method **29**
3.1 Literature Study . . . 30
3.2 Interviews . . . 31
3.2.1 Informative Interview . . . 31
3.2.2 In-depth Technical Debt Interviews . . . 32
3.3 Identification of Key Metrics . . . 32
3.4 Data Collection . . . 33
3.4.1 Product Metrics . . . 33
3.4.2 Process Metrics . . . 33
3.5 Technical Debt Model . . . 33
3.5.1 Weighing the Parameters . . . 34
3.5.2 Adjusting for File Size . . . 34
3.5.3 Normalisation . . . 34

3.5.4 Technical Debt Index . . . 35
3.6 Evaluation of Model . . . 35
3.7 Implementation of Visualisation . . . 37

4 Result 39
4.1 Interviews . . . 39
4.1.1 Types of Technical Debt in the Organisation . . . 39
4.1.2 Reasons for Acquiring Technical Debt . . . 40
4.1.3 Best Practises Already Used by the Organisation . . . 40
4.1.4 Improvement Suggestions by the Organisation . . . 41
4.2 Identification of Key Metrics . . . 42
4.2.1 Goal-Question-Metric . . . 42
4.2.2 Desired Metrics . . . 43
4.2.3 Actual Metrics . . . 43
4.2.4 Code Smells Approximation . . . 44
4.3 Data Collection . . . 45
4.4 Technical Debt Model . . . 45
4.5 Evaluation of Model . . . 47
4.5.1 Validation of Identified Files . . . 47
4.6 Technical Debt Visualisation . . . 50
4.7 Implementation in the Development Process . . . 53
4.7.1 Technical Debt List . . . 53
4.7.2 Training of Developers . . . 55

5 Discussion 57
5.1 Interviews . . . 57
5.2 Selection of Metrics . . . 57
5.3 Technical Debt Model . . . 58
5.3.1 Model Composition . . . 58
5.3.2 Model Precision . . . 59
5.3.3 Model Shortcomings . . . 60
5.4 Evaluation of Method . . . 61
5.5 Technical Debt Visualisation . . . 61
5.6 Implementation in the Development Process . . . 62
5.7 Threats to Validity . . . 62

6 Conclusion 65

7 Future Work 67

References 69

Appendix A Interview Questions 75

Appendix B In-depth Interview Questions 77

Appendix C Results from Goal-Question-Metric 79

Appendix D Metric Definitions **83**

Abbreviations

API Application Programming Interface. 33, 41, 43, 48, 60

ATD Architectural Technical Debt. 27, 65

CSV Comma Separated Values. 33, 34, 37, 45

GQM Goal Question Metric. 17, 32, 42, 43, 61

JS JavaScript. 33, 40, 42, 43

LOC Lines of Code. 20, 25, 34, 37, 45–47, 50–52, 58, 61

SM Software Metrics. 12, 13, 15, 16, 19–23, 31, 41, 42, 53, 59, 61, 65, 67

TD Technical Debt. 11–15, 18, 19, 21–36, 39–41, 43, 45–47, 50–53, 55, 57–63, 65, 67

TDM Technical Debt Management. 14, 15, 18, 24, 28, 32, 42

TS TypeScript. 33, 40, 42, 43

Chapter 1
Introduction

Technical Debt (TD) is a metaphor used to describe the consequences of taking a shortcut in the software development process. The shortcut gives the team benefits in the short term with faster releases to the customer and decreased time-to-market (Yli-Huumo et al., 2016). However, debt does not come without a cost, and the long-term consequence of taking too many shortcuts and never paying them back can eventually lead to software bankruptcy.

When developing software and delivering features at a high pace, there is the risk of creating TD. TD is usually caused by the need of cutting corners and making somewhat easier or limited solutions with poor architecture, automation, and quality assurance. If the TD grows too big, there is a high risk that a team has to spend most of the time maintaining and fixing issues, instead of being able to build new functionality.

1.1 Problem Statement

In order to enhance the way of working and address the TD, there is an urge to have proper monitoring, both of the code itself but also how the teams are delivering the software. That is, there is a need to measure and visualise the TD, to make appropriate decisions on how to improve and balance between continuous improvements and developing new features.

1.2 Research Questions

The overall vision of the master's thesis was to identify and understand how a large software organisation can succeed when working with TD and develop a strategy to know when, where, and how the TD should be repaid. The goal was therefore to investigate best practices around working with TD and finding a way to measure and visualise TD in code. The thesis shall also include implementation of these measurements in a team. Four research questions related to these steps were formed:

- **RQ1.** What are the best practises to manage and visualise TD according to literature?
- **RQ2.** What software metrics can be used in this type of organisation to measure TD?
- **RQ3.** How can these software metrics and best practices be implemented in the development process of a large software organisation?
- **RQ4.** How well does the measurements mirror the perceived TD in a project at the case company?

1.3 Contribution

This thesis contributes with knowledge and experience from applying Software Metrics (SM) to measure, visualise and prospectively reduce TD at an organisation of this type. The thesis also contributes with concrete examples of metrics and how they can be used to find which files in a project contain the most TD. Finally, this thesis contributes with examples of how the data collection for these metrics can be implemented and further introduced in an organisation of this type.

1.4 Distribution of Work

The work has been distributed equally between the two authors. Some parts of the thesis have been conducted in pair and some has been divided due to time limits and the fact that a PC was required, which we only had one of. Dan Wahlin has therefore been responsible for the implementation of measurements and visualisations, while Erica Schillström had responsibility for data collection and data analysis.

1.5 Outline

The report begins with a background and related work description in Chapter 2, where the term TD is presented together with topics related to it and best practices on working with TD in an organisation. The chapter further describes work done in the software metrics subgroup of TD identification as well as work on visualising TD.

Chapter 3 presents the method used in the thesis including how the literature study and interviews were conducted as well as how the TD model was formed, visualised and evaluated. The results from the interviews, the TD model, visualisation and evaluation are presented in Chapter 4. The method, the results and the threats to validity are discussed in Chapter 5. Finally, the thesis will be wrapped up with conclusions about the results in Chapter 6 and suggested future work in Chapter 7.

Chapter 2
Background and Related Work

The metaphor TD is a complex concept that can not be described with only one example, nor can it be measured in only one way. Furthermore, it is a concept that not everyone is familiar with and the questions regarding TD and how to manage it are many. This chapter provides an overview of the theory with a deeper description of TD, best practices to consider when working with TD and relevant information about the case company, needed to fully understand the background of the thesis.

2.1 Theory

This section serves the purpose of providing a theoretical background to the reader about TD, SM and agile software development. Each concept description includes a background, examples, and benefits of adapting the concepts in the software development process.

2.1.1 Technical Debt

The term TD was first coined by Cunningham (1992):

> "Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt."

Time saved from releasing a feature with lower quality earlier will be lost when the maintenance is greatly slowed down due to, for example, undocumented and nonreusable code. However, it is difficult to prioritise long-term benefits that have no immediate customer value, making it a common problem.

Since Cunningham coined the TD term it has been further researched, especially in recent years (Yli-Huumo et al., 2016; Seaman and Guo, 2011). Cunningham had more focus on the

code but nowadays the general consensus is that TD can appear in all parts of the software development (Tom et al., 2013). Some examples of TD types are:

- **Code Debt** such as duplicated code.
- **Design and Architectural Debt** such as needless dependencies.
- **Knowledge Distribution Debt** such as lost developer knowledge.
- **Documentation Debt** such as outdated documentation.
- **Testing Debt** such as lack of test coverage or excessive manual testing.

Li et al. (2015) performed a systematic mapping study on TD and Technical Debt Management (TDM) by collecting studies and creating a classification and thematic analysis on these studies. The main goal was to get a broad understanding of the concept of TD and an overview of the current state of TDM. The systematic mapping study contains 94 different studies about TD and TDM. Eight TDM activities were identified, presented in Table 2.1. The study showed that these TDM activities received different levels of attention and that *TD identification*, *TD measurement* and *TD repayment* have each been mentioned in more than 50 per cent of the 94 selected studies.

Table 2.1: TDM activities. * Activity is mentioned in more than 50 per cent of the studies

TDM activity
TD identification *
TD measurement *
TD prioritisation
TD prevention
TD monitoring
TD repayment *
TD representation/documentation
TD communication

Table 2.2: Non-TD types

Non-TD
Defects
Unimplemented features or functionalities
Lack of supporting processes
Unfinished tasks in the development process
Trivial code quality issues
Low external quality

As a way of defining the metaphor TD, six things that should be regarded as non-TD have been identified (Li et al., 2015) and are listed in Table 2.2. There are several studies identifying

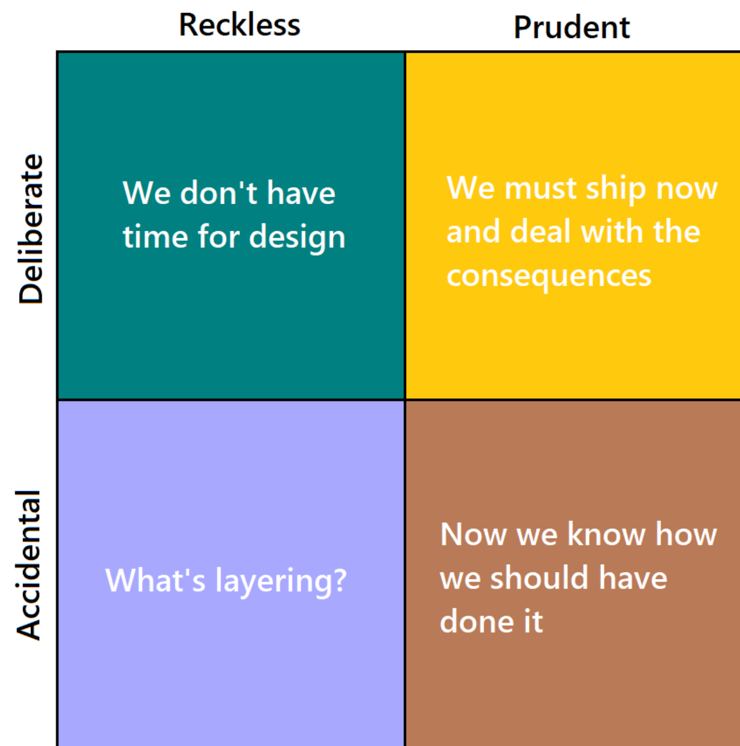


Figure 2.1: The Technical Debt Quadrant

defects as a type of TD and some as non-TD. According to Li et al. (2015), there is a need for more empirical studies with evidence on the TDM process along with more dedicated tools for managing TD.

The acquirement of TD is mostly a factor of time constraints, i.e. a deadline needs to be met (Cunningham, 1992), but it can also be acquired as a factor of other reasons or unconsciously. Another aspect of the acquirement is if it is done recklessly or if there is some afterthought to it. The Technical Debt Quadrant in Figure 2.1 is commonly used to exemplify these reasoning (Fowler, 2018).

On the other hand, not acquiring any TD is also substandard as it slows down the development and requires an enormous workload to achieve since everything has to be done perfectly from the start (Kniberg, 2013). Deliberately acquired TD that later can be repaid will speed up the software development while also keeping it maintainable. In other words, working with TD is about managing the balance between acquiring and repaying it.

2.1.2 Software Metrics

SM are measurements of software characteristics which has become an essential part of good software engineering (Fenton and Bieman, 2014). The measurements can be made during any part of the software development process and are used to get a better insight into the software. This includes, for example, measuring quality or performance and visualising the results to help understanding, or as a basis to make decisions. In software, metrics are classified into three categories based on entities and wanted attributes. These categories are process metrics,

product metrics, and resource metrics.

Process Measurements

Process metrics describe the characteristics of methods, techniques, and tools used in the software development process. There are many metrics that can be measured in agile software development to visualise how the process is evolving. These metrics are widely used in the industry (Altvater, 2017). Some examples are presented in Table 2.3:

Table 2.3: Process attributes and metrics

Attribute	Metric
Speed	Lead Time Cycle Time Sprint Burndown
Customer Satisfaction	Number of Bugs Reported Customer Rating

Product Measurements

Product metrics describe characteristics of a product. There are large amounts of SM (Fenton and Bieman, 2014; Lanza and Marinescu, 2007) that can be used to measure different attributes of the product. Some examples are presented in Table 2.4.

Table 2.4: Product attributes and metrics

Attribute	Metric
Size	Kilo Lines of Code Number of Attributes Number of Methods
Complexity	Cyclomatic Complexity Depth of Inheritance Tree Lack of Cohesion of Methods
Quality	Number of Defects Code Duplication

Resource Measurements

Resource metrics describe the characteristics of the entities required by a process activity and help us to understand and control the process. There are many important metrics related to different activities of software development. Some examples are presented in Table 2.5.

Table 2.5: Resource attributes and metrics

Attribute	Metric
Teams	Team Size Productivity Communication Level
Software	Reliability Usability Size
Personnel	Price Experience

Determine What to Measure

A single metric value is not very useful for visualising or basing decisions on, nor is it possible to measure and analyse everything due to time and money restrictions. One way to determine what to measure is to use the Goal Question Metric (GQM) paradigm (Fenton and Bieman, 2014). If the project has clearly defined goals, it is possible to know the state of the project and its processes. To use GQM, one or more major goals for the project are formulated. These goals are used to formulate questions to determine if the goal has been achieved or not. Finally, it is decided what metrics need to be measured to answer these questions. An example of GQM is shown in Table 2.6.

Table 2.6: Example of GQM

GQM	
Goal	Analyse the software quality
Question	How much unwanted behaviour is in the code?
Metrics	Defects per Kilo Line of Code & Amount of Code Duplication.

Another way to have proper monitoring is to both use several metrics in different parts of the software development process and to combine related metrics to get a better perspective or normalise the metric regarding, for example, size.

2.1.3 Agile Software Development

Agile software development is a framework with four core values:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

Further, Scrum is one type of agile software development framework that values cross-functional teams to rapidly deliver features while also being adaptive to change (Alliance,

2017). The Scrum framework adapts the core values by using teams that work in short cycles, so-called sprints, that usually are between 1 to 4 weeks long where a backlog (a list) of tasks are to be completed in this time. At the end of each sprint, an evaluation is made, called a retrospective, where lessons learned and the sprint result is discussed. These short cycles allow quick changes to requirements and adaptions to evolving markets with fast feature releases.

A Scrum team generally contains a Scrum Master, Product Owner, and a smaller team of developers, testers, quality assurance, and more. There are however no set rules on how a Scrum team should be organised, meaning that many variations exist.

Technical Debt in Agile Teams

The agile software development way of working introduces a lot of TD (Bavani, 2012). Teams working in an agile development process are delivering business value frequently while constantly adapting to changes during the process and still following the schedule . While trying to maintain this steady work tempo, teams will most likely introduce TD and the software becomes much harder to maintain. This is a result when less focus is put on creating readable and reusable code and documentation (Alliance, 2017) as it brings no direct customer value.

Eisenberg (2012) presents that awareness and management of TD are especially crucial for projects employing agile methods. This is motivated due to the characteristics of agile development with the expectations for continual refactoring and frequent change with continuous integration. All parts of an agile team need to learn and experience the various types of TD and TDM. For the agile team to understand and manage TD better, they need to not only be aware but also aligned to successfully deliver business value. A team is aware when the same understanding of the meaning of TD is shared among the team members and they are aware of it. On the other hand, a team is unaware when the team members do not share the same understanding of the meaning of TD nor are they aware of it. Agile teams are aligned when the same understanding on how to coordinate and manage TD is shared among the team members and the reduction of TD brings business value to the stakeholders. A team is not aligned when the team members are unable to jointly coordinate and manage TD and become reactive when it comes to paying off TD and may eventually disappoint in delivering business value (Bavani, 2012).

2.2 Technical Debt Management

This section is a result of the literature study done as a part of the thesis and serves the purpose of answering RQ1 about best practices to manage and visualise TD. The section is presented in this chapter to give the reader all information needed to grasp the future parts of the thesis.

2.2.1 Working with Technical Debt

The main steps to reduce, visualise and work with TD found in the literature study are identification, measuring, and managing. These steps will be examined and different approaches to working with them are presented in this section.

Identification Methods

The first step in being able to manage and visualise TD is to decide what methods to use to identify it. As stated in Section 2.1.1, TD can be created in all parts of the software development and can be identified in different ways depending on where it emerges. The TD term is vaguely defined, making identification of it a difficult task, and it is often the most costly step in reducing TD (Guo et al., 2016). Different individuals will identify different TD, and so will different tools (Zazworka et al., 2013). The performed literature study resulted in defining the more useful and essential identification methods as:

- Code smells
- Agile metrics
- Version control and commit evaluation
- Developer knowledge
- Backtracking system defects

Different techniques should be used in conjunction to form a more complete picture of the extension of TD in the software development (Zazworka et al., 2014). Combining both metrics and human evaluation will also increase the understanding since there are TD that tools can't efficiently measure in contrary to humans, and vice versa. An example of TD that would be easier for a human to spot than a tool is the quality, not amount, of documentation (Seaman and Guo, 2011).

Code smells, introduced by Fowler (2018), is one way to identify TD in the code and code architecture since it can be measured with SM directly in the source code and is closely linked with hard to maintain code. Some of the most common and quality-related code smells, defined by Lanza and Marinescu (2007) are:

- "God Class": A class that centralises the intelligence of the system and performs too much work on its own. This reduces both the maintainability and readability of the system.
- "Data Class": A class containing data that has no own complex functionality itself but rather holds data for other classes to do work on. The class is a sign of related data not being kept together nor in the right class.
- "Brain Method": Just like how the God Class unevenly centralises a system into a single class the brain method contains most of a class's functionality. It is often very long and complex making it hard to maintain and understand.
- "Significant Duplication": With significant duplication of the code finding errors is much harder, you can no longer say "Class X does this so the error should be there". Significant Duplication also clutters the code making it less readable and unnecessary time is spent writing code that already exists.
- "Shotgun Surgery": An effect where changing a single operation leads to having to do small changes to a lot of operations and classes. It is the result of undesirable coupling where large amounts of other classes and methods are calling a certain operation.

Own combinations of metrics and their thresholds to identify different code smells can be defined since the code smells and the threshold values will vary from project to project, but also between different coding languages. There has been a lot of work done in this area to establish different definitions, two examples of metrics (described in Appendix D) and thresholds to define a God Class are:

- Figure 2.2 shows a definition using the metrics Access to Foreign Data (ATFD), Weighted Methods for Class (WMC) and Tight Class Cohesion (TCC) (Lanza and Marinescu, 2007).

- Figure 2.3 shows a definition using the metrics Lines of Code (LOC), Weighted Methods for Class (WMC), Coupling Between Objects (CBO) and Lack of Cohesion in Methods (LCOM) (Mori et al., 2018).

The first definition, by Lanza and Marinescu (2007), uses easy to grasp thresholds such as *one third* and *few* but also the relative threshold *very high*, which was found by computing WMC for several projects and selecting the upper limit. The second definition, by Mori et al. (2018), solely uses relative values, *high*, for the different parameters that also were acquired by analysing the SM for different projects. The high thresholds are varying for each SM.

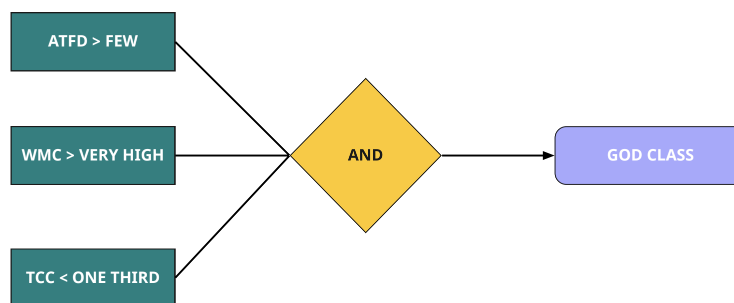


Figure 2.2: Definition of a god class using three software metric parameters.

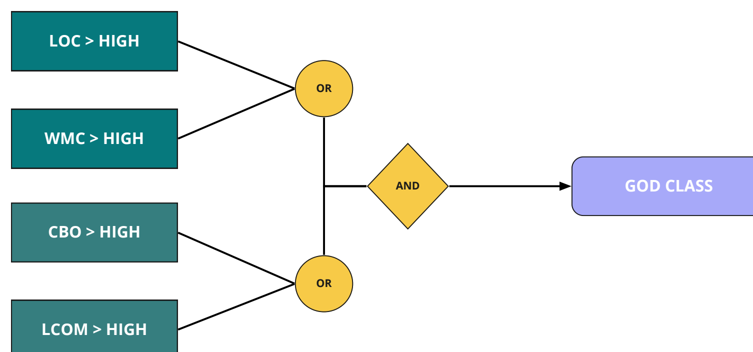


Figure 2.3: Definition of a god class using four software metric parameters.

TD can also be identified by analysing the agile development process, as the characteristic high speed of the agile development decreases, this can be an indication of growing TD causing problems. In an article interviewing development leads about the best metrics for measuring software development productivity (Altvater, 2017) the following agile metrics, among others, were named:

- Lead Time
- Time Spent on a Subtask
- Sprint Burndown
- Cycle Time
- Standup Meeting Length
- Production Issue Fix Rate

These metrics can be used to identify the existence of TD in general as it can be the reason for the decreased productivity, but also specify the location of the TD depending on the metric.

Version control and commits can also be analysed to identify TD. In some cases, using SM to find code smells in very large systems is time-consuming and inefficient (Tornhill, 2019). There is also the risk that a lot of the code smells are found in areas of the code that are hardly or never used or changed, making the refactoring of this code needless. Instead, commit frequency can be analysed since methods and classes with high commit frequencies are being changed more often which could be because they are too centralised or complex, meaning that they contain a lot of TD.

Another study analysing commits, by Alfayez et al. (2018) discovered that developers with frequent commits to a project introduced less TD than those who committed less frequently. It was also seen that a developer with more seniority to the system introduced less TD than a developer new to the system. Finally, it was seen that the longer the interval between commits the more TD was introduced for each commit. These results can be used as a way of identifying or preventing TD, allowing developers fulfilling these criteria to work on less critical areas of the system.

The use of developer knowledge of the code is also a resource that can be used for TD identification. Developers are aware of some of the TD they introduce, and often leave comments on code they have just written containing TD to remind themselves to go back and fix it later. This TD can be identified by giving the developers time each sprint to go back and find it, and especially use their insight to find the most system-critical TD. Yli-Huumo et al. (2016) found that a few of the teams they interviewed had a 20 per cent of development time for improving internal quality-rule in action.

If using the developers who wrote the system is not possible, Natural Language Processing can be used to search for terms in source code comments associated with bad code to identify this self-admitted TD (da Silva Maldonado et al., 2017).

Backtracking system defects is also a good indicator of TD (Seaman and Guo, 2011), it directly relates to faults in the code and backtracking to the defect origin can give hints of minor faults or the presence of widespread TD. Defects can easily be measured with SM both from customer feedback and failed requirement tests. The number of reported or found defects per sprint can be measured to picture what direction the project is moving.

Measuring

The second step in working with TD is to track the TD over one or several projects in a time context. For example, the number of released features per sprint, amount of God Classes, and number of defects can be measured at different times in the project. This would allow trends to be seen, if the amount of TD is quickly increasing different managing actions can be taken. Consistently measuring the different metrics is an important part of taking care of TD in a project. Measuring once and taking action and not verifying the result of these actions will not give any feedback about the managing procedure. It will neither give hints about how the project is moving.

Jabangwe et al. (2015) conducted a systematic literature review to find which SM were most closely related to maintainability and reliability in the literature. The study found that size, complexity, cohesion, and coupling metrics were the best to assess quality for object-oriented systems.

As a way of estimating the severity of different TD items that can appear in the development many studies recommend using the principal and interest analogy from the finance sector (Guo et al., 2016; Martini et al., 2016; Seaman and Guo, 2011). Principal in the TD domain represents the cost and effort needed to remove the TD item right now, while the interest represents the extra cost and effort that is being accumulated over time as a direct reason of not removing the TD, for example, larger maintenance costs.

The interest can further be divided into two parts: rate and probability. The rate represents the severity and the amount of interest if it is not removed by looking at the negative aspects. While probability represents the risk of the interest becoming a problem in the future. There are some TD that will exist but never cause problems. An example of such TD is the lack of documentation of a system, if the developers working on the system know its ins and outs, no documentation will be needed as long as they stay on the project for its entire life cycle. However, if one or all of the developers quit for some reason before this, the lack of documentation will instantly be a large problem for the developers taking over the project. An example of a TD item description including these parameters is presented in Table 2.7.

Table 2.7: An example of a TD item description

TD item	Description
TD Type:	Lack of test coverage
Location:	Method *example_method* in module *Example*.
Description:	The method *example_method* from last sprint has not been tested thoroughly
Principal:	Medium (medium level effort to unit test the method)
Interest rate:	High (the risk of bugs and possible extra effort needed will be costly)
Interest probability:	High (this is an important method that the system heavily relies on)

The list of TD items can be used for measuring the total principal and interest for the whole project. By doing these measurements during a fixed time, the data can show if the TD in the project is increasing or decreasing.

Several studies suggested using this type of TD list as a visualisation of the current TD. On the other hand, Eisenberg (2012) suggested basing the TD visualisation on estimating the cost of fixing each debt that was found through static and dynamic code analysis. These costs should be summarised into a total dollar value that could be analysed.

Martini et al. (2016) found, by interviewing 226 developers from 15 different software organisations, that only 28 per cent of the respondents were individually tracking some kind of TD by, for example, writing up TD items in an Excel spreadsheet or use some kind of code analysis tool to find bad code. 7.2 per cent had their organisation's management recognise the importance of TD and had a budget (around 10-30 per cent) directed for manual tracking and repaying of debt. None of them had measuring tools fully integrated into the development process estimating TD and especially TD interest and principal to visualise the consequences of repaying, or not repaying TD. None of them neither had the measuring institutionalised in the entire organisation. According to the study, these were important steps that the organisation should strive to achieve to improve their TD measurements.

Yli-Huumo et al. (2016) interviewed 8 different development teams from the same organisation and found that if they were measuring TD, it was done by either measuring the amount of TD items on the backlog or by using the code analysis tool SonarQube[1] to measure and compare different metrics. This type of tool uses a method called SQALE for calculations (Letouzey, 2012). The method was intended to objectively measure and manage the quality of the delivered source code. It was also designed to be generic and applicable with any coding language. The method is based on the fundamental principles of the measurement theory as well as the representation condition. With SQALE, measuring the quality of a product means measuring its amount of TD. The method uses a remediation index to measure the cost of actions correcting non-compliance in the product which is similar to measuring the distance between its current state and its quality target for each component of the code.

In a research article by Eisenberg (2012), it is suggested to measure higher-level SM such as the number of God Classes in the system after each sprint and lower-level SM such as duplicated code or other violations in the code that is to be committed at each contribution time. This is due to large system evaluations being very time consuming, which would slow down the development.

There are several tools for measuring TD through, for example, identifying code smells with metrics, finding violations of code conventions or analysing version control commits. A few of the tools also offer estimations on the principal cost to refactor the found TD and the interest of not removing it. Some of them also allow their metrics to be defined and measured. The earlier mentioned open-source tool SonarQube is the most used in the industry followed, in no particular order, by SonarGraph[2], CAST[3] and NDepend[4] (Avgeriou et al., 2020; Bogner et al., 2018).

When selecting a tool it is important to be sure of what is expected from the tool. A few questions to help this selection are (Avgeriou et al., 2020):

- Does it measure what we want it to measure?
- Can own metric definitions be defined?
- How is principal and interest defined?
- Which coding languages are compatible?
- How easy is it implemented into the development process?

[1] `www.sonarqube.org`
[2] `www.hello2morrow.com/products/sonargraph`
[3] `www.castsoftware.com`
[4] `www.ndepend.com`

Managing

The final step once the TD has been identified and measured, is to take decisions on when and if to remove it, which there are several approaches to. Most of them require some kind of approximation of cost or time required to fix a specific TD item. These approximations are hard to make without previous experience, what item requires more effort? Reducing code duplication from 30 per cent to 10 per cent or increasing the documentation coverage from 50 per cent to 60 per cent, and what are their monetary costs? Consequently, there is quite a steep learning curve when introducing TDM, it will however become easier as knowledge and experience are built.

A few of the TDM approaches found with the literature study are closely related to strategies found in finance. The first approach, introduced by Guo and Seaman (2011), is the portfolio approach that uses the portfolio management theory from the finance domain to create a portfolio containing TD items. The principal, expected interest rate and interest standard deviation for each item are estimated. The portfolio's return is calculated to decide which TD items should be resolved first. The second approach is the highest interest first method, where the TD item that is estimated to have the biggest negative impact on the project in the future is repaid first (Seaman and Guo, 2011). The earlier the high interest item is removed the shorter time its high interest is accumulated, and the repayment will be less expensive. The third approach includes calculating a break-even point (Seaman and Guo, 2011) between, for example, the value of newly implemented functions versus the negative value of higher maintenance. When the positive value of the induced TD no longer outweighs the negative value it must be repaid.

Decisions can also be made directly from software metric values (Eisenberg, 2012). One approach is to first select relevant metrics for the organisation, and for each metric create three thresholds: Green - desirable, Yellow - average, Red - bad. The values for these thresholds must be found empirically for each company, this can be done by determining the desired green metric values from projects that are deemed successful and have few maintenance issues. The opposite can be done to set the values for the yellow and red thresholds. False positives received from the measuring tool(s) can also be used to adjust the thresholds to increase their efficiency. The current debt, and especially the cost to move into the desired threshold can be calculated by estimating the cost for the different metrics and their changes, for example, the cost of lack of automated test coverage or, cost of improving test coverage.

When managing it is also easy to forget, and hard to put a monetary value on the human aspect of TD when making decisions (Dietrich, 2016). Deciding to repay some TD will increase the morale of the developers, no longer having to manufacture endless explanations why implementing new features take three times the estimated time and vice versa. This has to be taken into account when deciding to repay or not to repay some TD.

2.2.2 Best Practises to Reduce Technical Debt

Within software development, we have identified six areas that need to be covered in TDM and in each of these areas, we present approaches to reduce and prevent TD in a project. The findings are based on best practices from the literature study and are intended to be used as a complement to the three steps of working with TD.

Code

In order to write code to reduce TD, some afterthought must be applied while writing it. Who is going to read the code? The complexity must be adjusted to fit the readers level (Cline, 2018). A coding convention should be followed and effort should be put to follow the idioms of the language the code is being written in. For instance, Google uses language-specific style guides that all engineers are to follow which has proven to be successful (Morgenthaler et al., 2012).

Several philosophies have their theories regarding the maximum level of if-statement indentation, maximum LOC per method etc. for each coding language. They all have their strengths and weaknesses and it is hard to say which philosophy is the best. The important thing is to select one and try to adhere to it (there will always be situations where exceptions must be made), consistency is key for maintainability.

The value of constants, enumerations and describing parameter names and order to ease the code's readability should also not be underestimated (Cline, 2018). Methods and classes should not be too long to improve maintainability and to reduce code smells. It will also increase the codes testability and especially the automation of the testing.

As mentioned in Section 2.2.1, Significant Duplication is one very common Code Smell. Zhang et al. (2011) found that systems containing duplicated code are much more fault-prone compared to systems containing other code smells. This is supported by Fontana et al. (2012) who also draw the conclusion that duplicated code is one of the worst Code Smell and the removal of it will improve software quality greatly. Therefore attention should be directed at generalising the code by removing duplication and instead, for example, create generic methods.

Refactoring

One way to reduce TD and make a system more maintainable is to refactor the code, although there are some guidelines when going about this. Refactoring is both expensive and time-consuming, and it is therefore important to identify the parts of the code that needs to be refactored the most (Fontana et al., 2012). This identification could be done by, for example, analysing which parts of the code is the most complex, is being used the most, or has the highest commit frequency (Tornhill, 2019). Refactoring a module that is currently working and will not receive further changes or affect other modules is essentially a waste of resources (Guo et al., 2016). However, having a very high or low commit frequency is not always seen as negative or positive. The decisions should be based on a combination of different metrics.

It is also suggested to prioritise refactoring of duplicated code since it is a big indicator of low maintainability (Fontana et al., 2012). Refactoring duplicated code leads to a less complex and more cohesive system which improves its usability .

One suggested way of working when developing to prevent the need for large refactoring of the entire system is to work in smaller increments (Kniberg, 2013) :

1. Write tests to confirm all requirements.
2. Implement the feature so that the tests passes without too much regard to quality.
3. Directly refactor the newly written code while using the tests as confirmation that the feature still works.

Another suggested way of working is to incrementally refactor code to avoid large costly code refactoring (Jeffries, 2014):

1. Implement features as usual.
2. When coming across chunks of *bad* code, instead of avoiding it like before, refactor each bad chunk that the new feature passes through.
3. Eventually paths of *good* code that can be used will form, and the previous big refactoring will no longer be necessary.

Documentation & Comments

Lacking or nonexistent documentation of the software is one common type of TD that is widely recognised by software developers (Tom et al., 2013). Having someone unfamiliar with the system make changes to it will require a lot of effort just to know where to begin. Sufficient system documentation will reduce this required effort. It is also important to update the documentation when extensive changes to the system are made. A survey sent out to practitioners in the industry found that the most reported symptom of low maintainability was outdated documentation (Bogner et al., 2018).

Another way to reduce the documentation TD is through value-adding code comments. Comments should not be used to describe what the chunk of code *is* doing but rather what it *should* be doing and why the current approach to tackle the problem was taken (Cline, 2018). This will give relevant information to the person trying to understand or fix the current code, while not being redundant.

Writing what the code should be doing will work as a fail-safe for the maintainer trying to find errors. The maintainer can quickly realise if the code is implemented according to what it is supposed to be doing. Explaining why the current approach was taken will also save time when, for example, fixing errors since alternative solutions that the original developer already tried, that did not work for some reasons, can be quickly ruled out. Comments that do not add value will only make the reading of the code slower and have a reverse effect.

Testing

Testing is a great way to discover effects of TD such as defects. However, there is also a great deal of TD that can be created during testing (Wiklund et al., 2012). Automated testing is one way to save time and resources, it allows testing to be done more often since the developers do not have to manually test all of their produced code. The saved time can be used on implementing new features or writing more maintainable code. Automating tests is however not the easiest task and can create a lot of TD if not done carefully.

Wiklund et al. (2012) investigated TD in test automation and found that software design principles are often forgotten when working with test automation systems. These forgotten principles include things like systematisation and documentation which would allow the automation to easier be implemented in new projects or understood by new users. They also found that using the same automation tool for different products would give many benefits such as re-usability and better knowledge transfer between different products. The study however also recognised that putting restrictions on tool usage could harm the organisation

since each tool has its limits and will have difficulties adapting to some products required tests.

Kasurinen et al. (2009) found in their study on testing problems in practice that taking test design into account early in the project, even at product planning would improve the testing process. With a defined system architecture, and especially a standardised architecture following an organisational model, the test automation process becomes easier enabling greater test coverage and less implementation time required.

In general, the testing should be held to the same standards as normal software development, cheating with standardisation, maintainability and quality of the testing will introduce equal TD as when doing the same with code. Doing this will require a larger upfront effort but it will also be worth it in the long run.

Architecture

When shortcuts are made in the architecture of a project, the TD is called Architectural Technical Debt (ATD). ATD regards taking a shortcut while designing the architecture of a software system (Verdecchia et al., 2018). A poor design decision affects the software structure and the interaction between objects.

Many code smells are a direct result of poor software architecture, for example, God Classes and Data Classes. This type of ATD can be reduced by having this knowledge so that when implementing new features it is done sustainable and generically.

2.3 Case Company

The case company this master's thesis is written for is IKEA Retail, where the investigated teams are working in an agile way using the Scrum development process. Each team has a product owner and a responsible engineering manager. The teams are working on several projects and consist of a varying amount of engineers whose assignments include front-end and back-end development, testing and deciding on software architecture. Requirements for new features for the products are mainly determined by the different national markets that use the product. Decisions regarding priorities between implementing new software features and refactoring old code takes place in dialogue with the product owner and within the team itself. These discussions are conducted daily. The team is currently not working with TDM but are aware that their products contain TD in different forms.

The software architecture is created by the team itself through discussions in the beginning and throughout the project when new features are added to the code. The software architecture is intended to be consistent in all projects. The team is working with an open mindset without any hierarchy. Close teamwork gives the team ability to work with a lot of freedom and to utilise known and possessed techniques. In addition to this, the team uses naming conventions to be consistent in writing code as well as code review for each commit. The current code is difficult for an outsider to understand and be familiarised with due to its complexity. In terms of testing, the team uses automated tests regularly and manual tests when preparing for software deployment.

The product investigated in the thesis is a web application whose main functions are creating a membership account and finding existing memberships. The web application is intended to be used on screens located inside the department stores. It is released to actual customers and no other system is dependent on the product's functionality. However, the product depends on other systems' and data platforms' functionality. This means that the team needs close communication with other teams.

Chapter 3
Method

In this chapter, the methodology used to answer the four research questions and complete the goals of the thesis will be described. This includes a description of each step and the motivation of the chosen method. The initial approach includes six steps to be conducted in the order shown in Figure 3.1. The steps are further described in the following list:

- A **literature study** of technical debt, software metrics and best practices, including research on how to tackle technical debt in large software organisations.
- **Interviews** with stakeholders about where and why technical debt appear in their development.
- **Identification** of key software metrics to measure and visualise TD in code.
- **Proposing data collection** from the company's system, needed to elevate their software delivery and operational performance awareness.
- **Implementation** of visualisations and applying our **Technical Debt Model** in a project.
- **Evaluation** of the software metrics regarding how well the measurements mirror the perceived TD in a project from relevant stakeholders in the organisation.

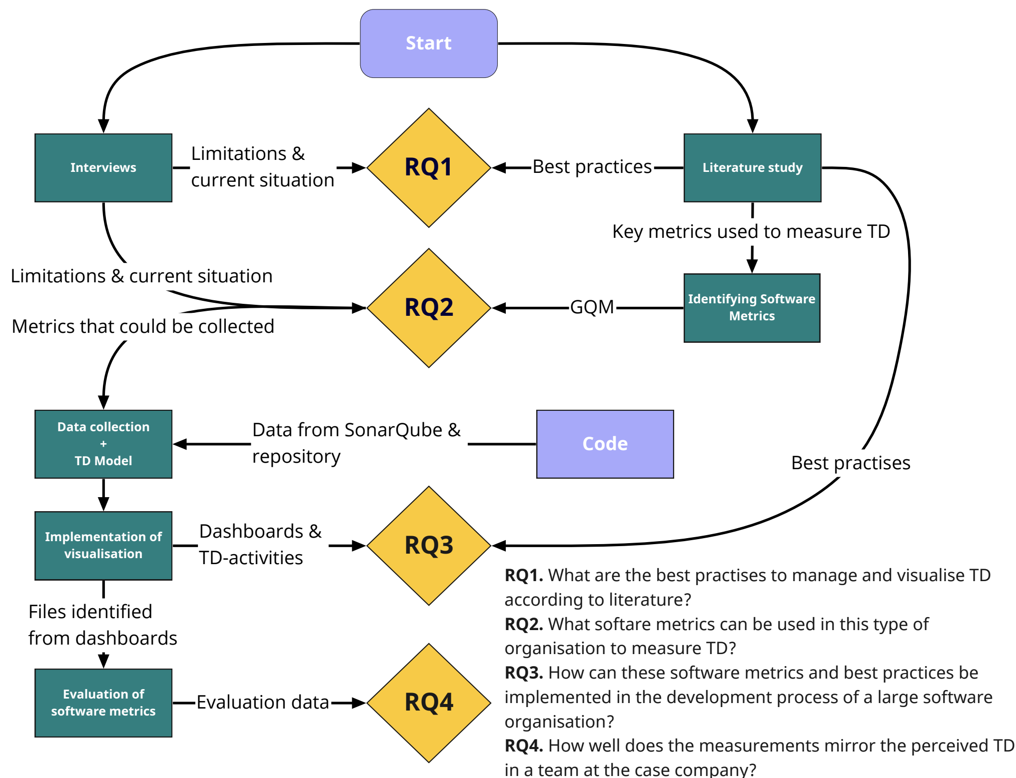


Figure 3.1: Overview of the approach to solve the research questions

3.1 Literature Study

To build knowledge about the rather vague and wide term Technical Debt, it was decided to conduct a literature study. The study provides credible knowledge from a lot of various sources in a reasonable amount of time. The literature study intended to explore several subjects regarding TD to get enough knowledge to answer the research questions and to continue with the following steps. These subjects were:

- Defining TD
- Different types of TD and its attributes
- Reasons for the existence of TD
- Identification methods for TD
- Measuring TD
- Managing TD
- Actions to avoid creation of TD
- Visualising TD with software metrics

During the literature study, the focus was to read peer-reviewed literature and relevant course literature. The literature was mainly found in the online databases Google Scholar and Lund University's LUB-search. We used a combination of keywords to find relevant and broad information. The keywords were "Technical Debt" on its own and together with "Manage", "Measure", "Software Metrics" or "Identify". When interesting and relevant papers were found, information about the paper was added to a document. When the document contained a list of about 50 sources, the literature study was paused and the sources in the document were sorted into categories to confirm that there was enough information about each category. Thereafter, each paper was read in the following order: Title, Abstract, Introduction and Conclusion. If the article was considered relevant enough, the full paper was read and later discussed between the researchers. Some snowball sampling was also done when other relevant articles were referenced in the articles from the original list. These papers were summarised in another document to highlight their key findings. The less relevant papers were also summarised, but with less depth, in case they were to become more relevant further down the line of the project.

Roughly 60 papers of various length from both journals, conferences as well as a few books made it into the document and were analysed during the literature study. Finally, around 50 of these were read thoroughly and about 35 made it into the report.

3.2 Interviews

In parallel with the literature study two types of interviews were performed, one informative interview and three in-depth interviews. The informative interview was done to get information about the case company and get an overview of how the team currently was working. The three in-depth interviews were held with two software developers and one Dev-Ops[1] engineer to get a general view of the team's TD knowledge, where they run into the most TD problems and how they are currently dealing with it.

3.2.1 Informative Interview

This interview was held during week 4 of the thesis together with our two supervisors and a consultant from their team who is working with SM. The interview gave us valuable information about how the teams are working within the organisation and to identify potential interviewees for the in-depth interviews.

The interview was discussion-focused and had 16 planned questions, although some were skipped if the answers had already been given during previous questions. One researcher was assigned as interviewer and the second one took notes and verified that all questions were answered. The interview questions can be found in Appendix A. The interview was recorded to give the researchers more time to focus on the interview and asking supplementary questions instead of only taking notes. The interview was originally held in Swedish and the answers were translated into English during the transcription. The duration of the interview including discussion about the in-depth interview was about one hour.

The transcript of the recorded interview was done shortly after the interview. Most of

[1] https://theagileadmin.com/what-is-devops/

the gathered information was used in Section 2.3, but it was also used as a guideline when creating the questionnaire for the in-depth interviews.

3.2.2 In-depth Technical Debt Interviews

After the first interview, we decided on having three in-depth interviews. These were carried out as a part of providing information about where and why TD existed in the team's products, and to answer RQ1 and RQ2. Two of the interviews were with software developers and one with a Dev-Ops engineer. The three candidates were contacted after a discussion with our supervisor since the candidates had shown interest in the thesis project. The three candidates that agreed to participate were from two different teams. The interview was performed similarly to the informative interview except that the focus was on receiving answers to the questions without a discussion. There were 19 prepared questions, although some additional questions were asked when the answers were not fully covering the questions. The duration of the interviews was about an hour each including an introduction and a discussion. All in-depth interviews were held in English and recorded with the consent of the interviewee. The questions used for the interviews can be found in Appendix B.

The interviews were transcribed shortly after they were held. To sort out and categorise valuable information it was decided to perform a coding of the transcription. This resulted in five different categories that were most relevant and can be seen below. Each sentence was read and analysed between the two researchers before it was placed in one of these categories.

- Why TD appears
- Different types of TD
- Improvement suggestions
- How is the TD found
- Why is the TD not refactored
- Miscellaneous

The few sentences placed in miscellaneous were seen as irrelevant for the thesis and was ignored. When the coding of the three transcriptions was done, each category was summarised and the results discussed with our supervisors. The key findings are presented in Section 4.1.

3.3 Identification of Key Metrics

The GQM paradigm that was presented in Section 2.1.2, was used to identify what metrics to gather. The approach was made with the insights from the literature study together with the goal of measuring TD as a basis. Instead of only using one goal for GQM, we defined one main goal and three sub-goals. The main goal was identified from the thesis' problem statement in Section 1.1 and the three sub-goals covers three different areas of TDM.

Some code smells mentioned in Section 2.1.1 were later on approximated using other metric combinations than those defined in Chapter 2 due to limitations in the actual gathering of metrics.

3.4 Data Collection

The data collection was conducted in different ways depending on the software metric category. One of the greater limitations was that the analysed projects were private, hence no tools with only Open-Source licenses could be used. The software also had to be secure and able to process JavaScript (JS) and TypeScript (TS) and preferably free, which narrowed the tool scope further. Since the organisation already had a licence for SonarQube and it fulfilled the previously specified requirements, this software was selected. The product that was analysed consisted of two projects, a front-end project and a back-end project.

3.4.1 Product Metrics

The two projects were manually downloaded from Github at 12 different points in time, from April 2020 to March 2021. The multiple downloads were to be able to see how the project size had evolved from month to month. They were scanned one by one with SonarQube's SonarScanner which saved the measurement data locally on the PC. The data could be viewed by connecting to localhost. SonarQube also has a Web Application Programming Interface (API) that could be used to collect the scanned data depending on different query parameters. The API was used with a Python program written by us to collect total project metrics for all measurement points, and file-level metrics for the most recent one, which was March 2021. This data was stored in two Comma Separated Values (CSV) files, which would later be used for the visualisation.

Since SonarQube did not measure coupling, a Python script to measure this metric was written. The script browsed through all files in the projects that could couple to other files (JS and TS files). The import rows of each file were parsed and matched to existing files counting the number of occurrences. These values were added to the file level CSV file.

3.4.2 Process Metrics

In the process metrics category total commits per file in the previously specified time frame were collected using GitHub's REST API. Repository and file paths were specified respectively. The file paths were retrieved from the SonarQube API using the previous program. The commit data per file was added to the file level CSV file.

Two agile metrics, tasks completed and tasks started, were collected by searching in the team's issue tracking software with a search query of “name contains *project name*” and “task completed”. The results containing a lot of different information were exported to Excel and further processed to count tasks started and completed during the specified time frame.

3.5 Technical Debt Model

To measure the TD in each file and project some kind of model had to be created. The model had to be able to take in many parameters and allow comparison between files of very different sizes and complexities. It also needed to be easily adjusted so that more parameters

could be included in the future, and the importance of each parameter could be changed depending on the type of project being analysed.

Finally, each parameter was analysed for independence with a correlation test using the Pearson method. This test was done on the file data stored in the CSV files using a Python module.

3.5.1 Weighing the Parameters

The parameters used as input in the model were weighed in three ways:

- Based on the understanding of the severity of different code smells and metrics gathered in the literature study. Number of occurrences of each metric and code smell in the literature, the authors opinions and the results from their work. For example, Fontana et al. (2012) drawing the conclusion that duplicated code is the worst code smell or god classes being mentioned in the majority of the studies.
- Based on how the metrics are related to different code smells. For example, what combination of metrics would describe a God Class or Significant Duplication closest?
- Adjusting for false positives. Metric values and importance vary greatly between different projects and sectors, and the preliminary weighing had too much weight on some metrics outputting very large TD values for sector-specific files. For example, files containing market-specific text were first heavily favoured.

If a larger value of the parameter was seen to increase or decrease TD was also analysed, some metrics were seen as positive. For example, a larger comment value was seen as positive since explaining comments increase the maintainability. However, some verification was also done to see that the positive metrics did not go too far.

3.5.2 Adjusting for File Size

Some of the parameters were divided by the file LOC size. This step was done to be able to compare files of differing sizes. Each parameter was evaluated on the basis if a larger file *most likely* would lead to a larger value on that parameter. The parameters that were not divided by LOC were perceived to have a low correlation to file size.

3.5.3 Normalisation

After adjusting for file size, the parameters were normalised between 0 and 1. This was made to both more easily compare files with each other and to have the parameter values ranging in the same interval. Before the normalisation, for example, the parameters divided by LOC had very low values compared to the parameters that is not divided.

The parameters were first normalised per project since it was found from the literature study that metrics could differ very much from project to project. An example of this normalisation can be seen in Table 3.2. However, after receiving feedback that the case company would like to be able to compare several of their projects in the future. The projects were

also normalised based on both projects, with the possibility of adding more projects to the normalisation. An example of this normalisation can be seen in Table 3.1

Table 3.1: Example of a normalisation based on both projects. There is only one max (1) and min (0) value per metric.

Project	**File**	Metric 1	Metric 2	...	Metric N
Front-end	File-fe-1.ts	1	0.200	...	0
...	...	...	...	...	...
Front-end	File-fe-n.ts	0.450	0.280	...	0.500
Back-end	File-be-1.ts	0	0	...	0.500
...	...	...	...	...	...
Back-end	File-be-n.ts	0.333	1	...	1

Table 3.2: Example of a normalisation based on the project itself. Each project has its own max (1) and min (0) value per metric.

Project	**File**	Metric 1	Metric 2	...	Metric N
Front-end	File-fe-1.ts	1	1	...	0.278
...	...	...	...	...	...
Front-end	File-fe-50.ts	0.333	0	...	0.900
...	...	...	...	...	...
Front-end	File-fe-n.ts	0	0.723	...	1
Back-end	File-be-1.ts	0.235	0	...	0.221
...	...	...	...	...	...
Back-end	File-be-50.ts	0.556	1	...	0
...	...	...	...	...	...
Back-end	File-be-n.ts	0.921	0.322	...	1

3.5.4 Technical Debt Index

To realise the possibility of comparing TD for different projects further, beyond the visualisation, a technical debt index was calculated. This index was deemed good enough showing the average TD per file in the project. This approach would allow projects of different sizes to be compared.

3.6 Evaluation of Model

To evaluate the accuracy of the model, feedback from the organisation was used. This feedback involved different stakeholders from the organisation that had a valid amount of insight into the code itself and also how the team delivers the software. By having this insight it was possible for the stakeholders to approximately estimate the amount of TD at different points in the project and different places in the codebase.

The chosen evaluation method consisted of two steps where the stakeholders used their system knowledge to identify the files with the most TD and rank these files further using the “Hundred Dollar Test” (Cumulative voting).

The Hundred Dollar Test is a way of scoring using imaginary money instead of points (Straker, 2021). The idea of spending money tend to grab more attention to the scoring and the money are allocated more carefully. The test is done in three steps:

1. Assuming that you have $100 to spend on different options

2. Allocate your $100 across the different options based on the criteria

3. Review your decision and look at how the money is spent so that the money is not spread too evenly.

Before the first step of the evaluation, the participants received a presentation file with the information found in the literature study regarding TD and its best practices. They were sent an email containing a link to a repository holding 15 files from two of the projects they work on (front-end and back-end). These 15 files had been selected by the researcher’s identification method. Five of them were considered to have “low”, five “moderate” and five “high” amounts of TD according to the researcher’s method and understanding of TD. The files were also selected to make sure that each category had files of similar sizes.

The email also contained the following instructions:

> “In the linked repository, there are 15 files from your projects. You are to select the five ones that you feel contain the most Technical Debt based on your intuition.
>
> Technical Debt is defined as the debt that is accumulated when shortcuts are taken in the development process to, for example, reduce time-to-market. If this debt is not repaid the risks of future problems with maintainability, release speed etc. is greatly increased. Two examples of Technical Debt are the lack of testing and writing unmaintainable code, to meet an upcoming deadline.”

Once each participant had selected the five files they perceived to contain the most TD, they were summoned to a meeting. In this meeting, all participants were to discuss their selections. If everyone selected the same five files, the participants would move on to the Hundred Dollar Test. If there were some differences in the selections the participants would have to discuss their decisions and mutually come to an agreement which five files that were considered to contain the most TD and would be used in the test.

With the five selected files, the participants would mutually conduct the test on the files. Their selection criteria would be based on how much effort it would take to remove the TD in the code.

Finally, the first file selection data from every participant as well as the data from the test would be collected. If the selections from the developers were similar to the selection made through the researchers’ identification method, the model would be considered valid.

3.7 Implementation of Visualisation

The data was visualised using Plotly Dash which is a tool for the programming languages Python, R and Julia, that can create interactive web pages and include several different visualisation possibilities. The CSV files including the collected data were read in a Python program that was using the Dash tool. Some files were filtered by setting a minimum LOC size and Cyclomatic Complexity. This was done to filter outliers or files with no complexity that would cluster the visualisation. The visualisations of the data were done using different plots and text boxes.

Chapter 4

Result

The following chapter presents results from the interviews, outcome and evaluation of the TD model, implementation, and visualisation of software metrics to identify TD in the current projects. The results from the literature study are presented in Section 2.1 as they are a part of the theory. The remaining parts of the results are presented in the same order as the activities were presented in chapter 3.

4.1 Interviews

In order to get a better understanding of the current situation in the team and what challenges they face in the aspect of TD, three interviews were conducted. The main findings from the interviews are divided into four sections and provide information about TD in the projects, precautions that have been taken and ideas for further improvements.

4.1.1 Types of Technical Debt in the Organisation

During the three interviews, a deeper understanding of the most common types of TD that the teams were struggling with was built. These types are listed in Table 4.1 and are all in line with the types that were presented in Section 2.1.1.

Table 4.1: TD Types Identified at the case company

Debt category	Type
Knowledge Distribution debt	Junior developer sub-par coding Lack of knowledge and expertise Dependency on one key person
Code debt	Duplicate code and similar functions Features added in market specific code instead of templates Too many lines in one method
Testing debt	Too little unit testing Test redundancy Lack of test coverage
Design and Architectural debt	Design loopholes Low quality architecture
Documentation debt	Not enough code-explaining comments to some extent Missing project documentation

4.1.2 Reasons for Acquiring Technical Debt

The developers had several ideas of the main reasons for the acquirement of these debts. Firstly, it was due to a combination of lack of knowledge and expertise which resulted in shortages in both architecture, testing and code. When the developers implemented new features without fully understanding the requirements it resulted in taking shortcuts in the future to meet the requirements.

Secondly, it was acquired both willingly and unwillingly due to time pressure from the markets and deadlines within the team. Quick development of new features to catch customers were implemented with less perfection. The developers skipped steps that were considered less important in order to start coding and deploy earlier. The focus was more on deploying and delivering the applications than on quality. The implementation of more tests to increase test coverage was identified as a solution to an improved development process and product. However, this was not prioritised by the product owner at the beginning of the project. The architecture was identified as a critical part and creating a good architecture from the start lead to much fewer problems in the future.

4.1.3 Best Practises Already Used by the Organisation

The interviews also resulted in the realisation that the interviewees' teams, to some extent, were already using some of the best practice procedures found in the literature study. The teams had a positive culture regarding TD and other problems, where raising issues was encouraged. They were also doing a few things on code level: Attempting to stick to standard JS and TS notation and formatting with the help of different tools and IDEs. Code reviews were also done on each pull request where one or two developers would need to approve it. Finally, they were commenting code of lesser quality as a reminder to return and fix it.

Some automation was also used in the teams for both testing, integration and deployment as well as measurements of some SM to get an overview of the health of the product.

4.1.4 Improvement Suggestions by the Organisation

Some further plans had not yet been put in motion to reduce TD in their projects. For example by containerising their application, remaking the system architecture to improve it or standardising the way of calling external API's. One interviewee also noted the importance of increased communication in the organisation and said:

"At the moment it is like everyone is doing their own thing. Every team is working independently without talking to other teams, the wheel is getting reinvented over and over again. Sharing knowledge will save time and money and make it more secure."

Another improvement suggestion was to introduce a "perfect prototype", with all possible TD preventing methods and analysing it to learn key lessons about TD based development.

4.2 Identification of Key Metrics

The scope of possible code metrics was heavily limited due to the organisation developing in TS and JS. There are several available free code analysis tools for languages such as Java and C++ but much fewer for JS and TS. Most of the analysis tools for JS and TS also report on stylistic errors, possible null pointers etc. but do not provide any SM. The few who does are behind a paywall unless the project is Open source software. The analysed projects are private and a few companies provide tools for retrieving SM in JS and TS for payment, but do not publicly describe which metrics they retrieve.

4.2.1 Goal-Question-Metric

The main goal of the GQM paradigm was summarised into the following sentence: *Get a better insight and have a proper monitoring, both of the code itself but also how the team delivers the software*. Based on this main goal three new goals were derived, which covers different areas of TDM; *Identification*, *Measuring* and *Managing*. Nine questions were generated to determine if the goals were met, and to answer these questions a large amount of metrics were identified. The metrics were divided into the three metric categories; process, product and resource metrics. The results from the GQM paradigm can be seen in Figure 4.1.

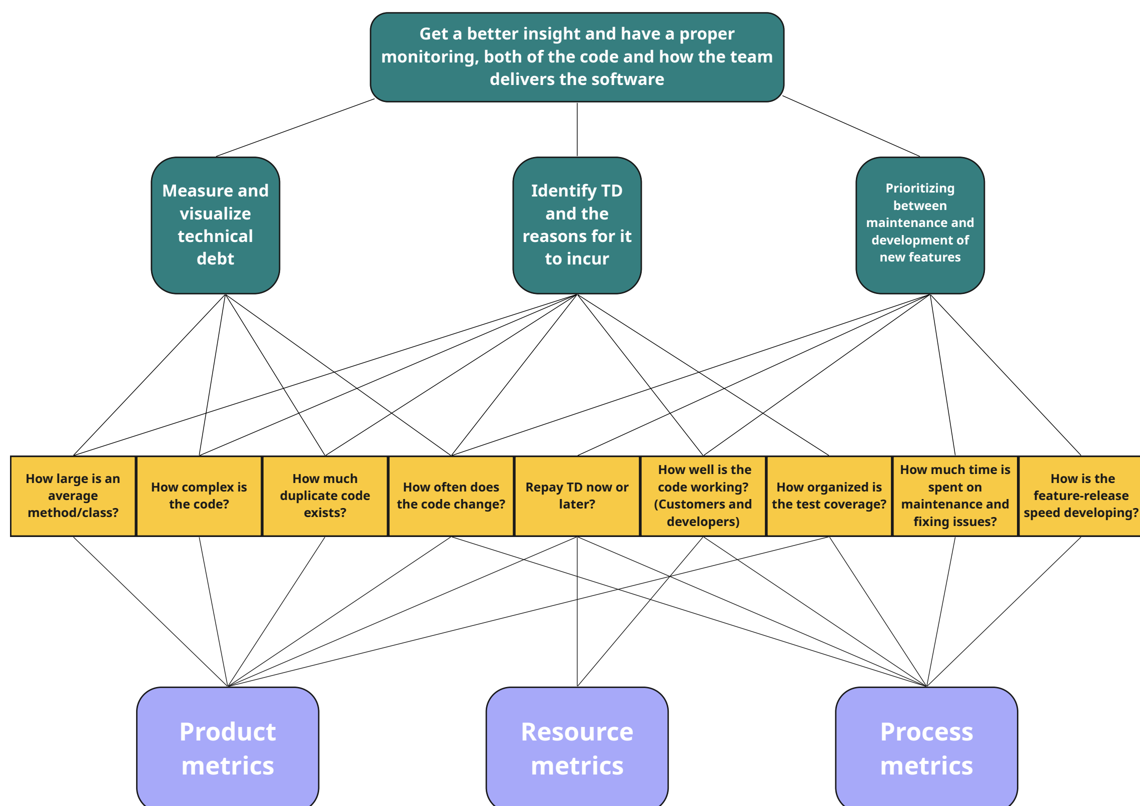


Figure 4.1: Goal Question Metrics

4.2.2 Desired Metrics

The metrics found in the literature study that were desired to be collected and included in the TD model as well as being deemed the most relevant by the researchers based on the GQM paradigm can be found in Appendix C.

4.2.3 Actual Metrics

Not all desired metrics were able to be retrieved due to the reasons stated in Section 4.2. The actual list of obtained metrics is presented in the following sections.

Product Metrics

SonarQube had a free version for private projects that could process JS and TS files and provide some metrics. The drawback however was that the free version could not be coupled with Git, so all projects, at different points in time, had to be manually downloaded and scanned. From SonarQube the following metrics from the desired list as well as a new metric that was deemed relevant could be collected:

- Cyclomatic Complexity
- Cognitive Complexity (new)
- Lines of Code
- Number of Functions
- Number of Duplicated Blocks
- Number of Comment Lines

Where Cognitive Complexity is a metric of how hard it is to understand the code's control flow compared to Cyclomatic Complexity which is calculated based on the number of paths through the code (S.A, 2021). Finally, one code metric had to be measured with a self-written script to identify how closely connected a file is with other files:

- File Coupling

Process Metrics

To obtain commit metrics Github's API was used for the projects and with the help of the issue tracking software, task data could be collected. The following metrics were obtained:

- Number of Commits per File
- Number of Started Tasks per Month
- Number of Completed Tasks per Month

4.2.4 Code Smells Approximation

Code smells were approximated in the files using the collected metrics and this approximation can be seen in Table 4.2.

Table 4.2: Code smells approximation for the files using the collected metrics.

Code Smell	Metric(s)
God Class	Lines of Code, Cyclomatic Complexity, Cognitive Complexity, Commits
Brain Method	Lines of Code, Cyclomatic Complexity, Cognitive Complexity, Number of Functions, Commits
Significant Duplication	Duplicated Blocks, Lines of Code
Shotgun Surgery	Coupling

4.3 Data Collection

Metrics for hundreds of files with thousands of LOC in the front-end project were collected. The same was done for the smaller back-end project. Retrieving metrics for a given amount of projects takes around 10 minutes and once it has been done, the data being stored in CSV files can instantly be visualised in the future. Based on our experience, it is recommended to analyse 1 or 2 projects at a time.

The collected data was analysed using Pearson's correlation analysis. This was done in order to see how independent the metrics were compared to each other, so that all of them had a reason to be in the coming model. A positive values close to 1 shows that there is a strong positive correlation between the two metrics. A negative value close to -1 shows that there is a strong negative correlation between the two metrics. Values close to 0 show low or no correlation between the metrics. The results for the front-end project can be seen in Table 4.3 and for the back-end in Table 4.4.

Table 4.3: Pearson's correlation test for the front-end files

	Cyclomatic Complexity	Cognitive Complexity	Lines of Code	Comment Lines	Duplicated Blocks	Functions	Commits	Coupling
Cyclomatic Complexity	1.000	0.695	0.316	0.156	0.159	0.786	0.216	0.076
Cognitive Complexity	0.695	1.000	0.204	0.217	0.157	0.384	0.269	0.023
Lines of Code	0.316	0.204	1.000	0.026	0.151	0.300	0.117	0.019
Comment Lines	0.156	0.217	0.026	1.000	0.024	0.113	0.124	-0.019
Duplicated Blocks	0.159	0.157	0.151	0.024	1.000	0.169	0.182	-0.125
Functions	0.786	0.384	0.300	0.113	0.169	1.000	0.123	0.154
Commits	0.216	0.269	0.117	0.124	0.182	0.123	1.000	0.154
Coupling	0.076	0.023	0.019	-0.019	-0.125	0.019	0.154	1.000

Table 4.4: Pearson's correlation test for the back-end files

	Cyclomatic Complexity	Cognitive Complexity	Lines of Code	Comment Lines	Duplicated Blocks	Functions	Commits	Coupling
Cyclomatic Complexity	1.000	0.966	0.732	0.635	0.588	0.342	0.402	-0.039
Cognitive Complexity	0.966	1.000	0.691	0.660	0.581	0.124	0.465	-0.041
Lines of Code	0.732	0.691	1.000	0.518	0.392	0.488	0.373	0.047
Comment Lines	0.635	0.660	0.518	1.000	0.500	0.063	0.298	0.064
Duplicated Blocks	0.588	0.581	0.392	0.500	1.000	0.031	0.113	-0.118
Functions	0.342	0.124	0.488	0.063	0.031	1.000	-0.103	-0.082
Commits	0.402	0.465	0.373	0.298	0.113	-0.103	1.000	0.276
Coupling	-0.039	-0.041	0.047	0.064	-0.118	-0.042	0.276	1.000

4.4 Technical Debt Model

TD was first calculated on file level using the TD model equation in 4.1, but before calculating the TD value with the collected metrics, they had to be processed. First, some of the process metrics were divided by the file's LOC. The values of all metrics were normalised in two ways, either with respect to both projects or separately. The metrics normalised values ranged between zero and one.

The normalised data was put in equation 4.1 where each metric was weighted after how important the researchers had perceived each metric based on the literature study and the interviews as well as adjusting for false positives. The metrics were also analysed to see if they positively or negatively affected the TD in the file. How each metric was weighted, effected TD and if it was divided by the file's LOC can be seen in Table 4.5, and for this case study, eight metrics were collected hence m has the value eight. TD_i is the calculated TD for file i where W_k is the weight for metric number k and M_k is the value for metric number k for this file, i.

$$TD_i = \sum_{k=1}^{m} W_k M_k \tag{4.1}$$

To be able to compare different projects, a Technical Debt Index (TDI) was calculated. This index showed how much TD on average was located in each file and used the calculated TD values using the normalised data based on both projects. The TDI formula can be seen in 4.2, where n is the amount of files in the project and TD_i is the calculated TD for file i in the project.

$$TDI = \frac{\sum_{i=1}^{n} TD_i}{n} \tag{4.2}$$

Table 4.5: Data processing for TD calculation

Metric (M)	Weight (W, %)	Divided by LOC?	Effect on TD
Cyclomatic Complexity (CC)	12.5	yes	increase
Cognitive Complexity (CoC)	17.5	yes	increase
Lines of Code (LOC)	15	no	increase
Comment Lines (COM)	−7.5	yes	decrease
Duplicated Blocks (DUP)	12.5	yes	increase
Functions (FNC)	−7.5	yes	decrease
Commits (CMT)	15	no	increase
Coupling (CPL)	12.5	no	increase

4.5 Evaluation of Model

The evaluation of the software metric based TD model was done with the method presented in Section 3.6. The 15 files were selected using the TD model and LOC as there should be files of similar size in each category. The results from the two steps of the evaluation are presented below. Once the evaluation was done the developers' selection criteria used during the evaluation was retrieved, presented in Table 4.6.

Table 4.6: Individual developer selection criteria for the first step of the evaluation.

	File selection criteria
Developer 1	Number of developers working on the file? How large is the file? How many commits has the file received? How much code is copy pasted? Gut feeling
Developer 2	Is the file rushed and sloppy? How important is the file?
Developer 3	Gut feeling Knowledge built from being a developer

4.5.1 Validation of Identified Files

The results from the first part of the evaluation is presented in Figure 4.7 for the front-end files and Figure 4.8 for the back-end files. Each file has been given a new, anonymous name based on LOC and if it belongs to the front-end or back-end in this thesis. The original names were used for the developers during the evaluation. The files are presented with a rounded LOC and its estimated amount of TD from the model calculation. The cells containing the developers' motivation (if any) has been given the colour grey if that file was voted on, otherwise white. Each file's total number of votes is presented at the end of each LOC section.

The results from the second part of the evaluation are presented in Figure 4.9. The five files that the developers chose for the Hundred Dollar test and the respective amount of given dollar is presented together with the calculated amount of TD using the two normalisation techniques. The actual rank the files should have been placed in according to the model is also shown in the table.

Table 4.7: Results from developer feedback for the front-end files.

Amount of TD	Low	Medium	High
LOC: 500-700	**LargeFile1-fe.tsx (LOC: 500, TD: 5.41)**	**LargeFile2-fe.ts (LOC: 700, TD: 14.56)**	**LargeFile3-fe.tsx (LOC: 500, TD: 22.60)**
Developer 1	Contains content that we receive from the markets. This is not a file we make changes to unless the markets have made a change.		Large file with a lot of logic and many developers have been involved. The file has also been rewritten and there is a V2.
Developer 2	LargeFile1-fe.tsx was not picked as there's no real tech debt in this file.	LargeFile2-fe.ts was not picked as there's no real or extremly small tech debt in this file.	LargeFile3.tsx was picked due to its implementation first time around was a rushed MVP. It's since had 2 revisions, making it slightly better each time, but due to the project getting major revamps its still left in some need of rework.
Developer 3	A copy of this content is not something we should maintain - it should exist in some centralised repo.	Redux boilerplate, could probably be reduced but no big deal.	
Number of votes	1	0	2
LOC: 200-400	**MediumFile1-fe.ts (LOC: 200, TD: 8.16)**	**MediumFile2-fe.ts (LOC: 300, TD: 10.53)**	**MediumFile3-fe.tsx (LOC: 400, TD: 16.60)**
Developer 1		MediumFile2-fe contains parsed data that we retrieve from an API. We then use the data to retrieve translations and the like and thus no file that we tinker with much as developers.	Relatively large file that contains a lot of logic.
Developer 2	MediumFile1-fe.ts was not picked since it is working as it should currently. There are 2 markets that have been out-commented due to testing, but its a an extremly small issue to correct.	MediumFile2-fe.ts was not picked as there's no real tech debt in this file.	MediumFile3-fe.tsx was picked & is a large remnant from the project version. Originally it was catered for several markets as a way to alter & show your profile. The code was messy and the UX hasted. This got broken down into several new views & pages in the new version, instead of 1 large file trying to do it all. Theres only 1 market left using this page/view today and once they are rolled out with the new project version it'll be removed compleatly.
Developer 3		This is configuration generated by a script.	Too much code in one place.
Number of votes	0	0	3
LOC: 100-200	**SmallFile1-fe.ts (LOC: 200, TD: 7.18)**	**SmallFile2-fe.tsx (LOC: 200, TD: 10.48)**	**SmallFile3-fe.tsx (LOC: 100, TD: 19.99)**
Developer 1	Contains all countries, languages and area codes. A file that is pretty straight forward I would say.	Many developers have contributed to this file even though it is relatively small.	Many developers have contributed to this file and I guess there are many commits as every time a new market is added there is a commit.
Developer 2	SmallFile1-fe.ts was not picked as there's no real or extremly small tech debt in this file.	SmallFile2-fe.tsx was picked & is a remnant from the old project version and subject to be deleted as soon as all markets have been migrated to the new project version.	SmallFile3-fe.tsx was not picked as there's no real or extremly small tech debt in this file.
Developer 3	Just some data that should very rarely change.		Hard coded market rules should not be placed in this button component.
Number of votes	0	1	2

Table 4.8: Results from developer feedback for the back-end files.

Amount of TD	Low	Medium	High
LOC: 100-200	**MediumFile1-be.ts (LOC: 200, TD: 2.59)**	**MediumFile2-be.ts (LOC: 100, TD: 7.14)**	**MediumFile3-be.ts (LOC: 100, TD: 17.70)**
Developer 1	Contains only a number of different country and language codes.	Not many developers contributed to the code. Well structured code.	
Developer 2	MediumFile1-be.ts was not picked as there's no real or extremly small tech debt in this file.	MediumFile2-be.ts was picked due to it being a vital part of the application, which is used often and could easily be broken down for a slimmer implementation.	MediumFile3-be.ts was picked. Its not an essential file to the project but theres a lot of repetitive code in there, which could easily be re-written to be smarter.
Developer 3	Bad file name, it does actually not do any validation. It is overhead caused by TypeScript and can probably be solved in some better way.		Looks much like repeated code.
Number of votes	1	1	3
LOC: 0-100	**SmallFile1-be.ts (LOC: 50, TD: 3.47)**	**SmallFile2-be.ts (LOC: 50, TD: 5.58)**	**SmallFile3-be.ts (LOC: 50, TD: 17.67)**
Developer 1	Many developers who contributed to the code.	One developer who made the file, easy to read code, not many lines.	Relative static code that does not change if nothing changes in the system we send the payload to.
Developer 2	SmallFile1-be.ts was not picked as there's no real or extremly small tech debt in this file.	SmallFile2-be.ts was not picked as there's no real or extremly small tech debt in this file.	SmallFile3-be.ts was not picked. It is working as intended and theres no 'real tech debt' in this file, theres 2 rows that could be removed but in comparison to other files its a tiny job.
Developer 3			This is just mapping from one data structure to another
Number of votes	1	0	0

Table 4.9: Results from the Hundred Dollar Test.

File	MediumFile3-fe.tsx	LargeFile3-fe.tsx	SmallFile3-fe.tsx	MediumFile3-be.ts	SmallFile1-be.ts
Given Dollars	45	30	15	7	3
Comments	* Barely used by markets. * Unnecessarily complex. * Very large and important for project. * Some functionality already broken out.	* Already replaced with V2. * Even new one can be improved and rewritten.	* Contain market-specific logic that should be in other places. * Low value for project to fix.	* A lot of copy paste. * Should be rewritten	* ts-ignore * Decoder token without verifying signature
Project normalisation made using both projects					
Calculated TD	16.60	22.60	19.99	17.70	3.47
Calculated Rank	4	1	2	3	5
Project normalisation made separately					
Calculated TD	21.02	27.43	22.89	15.82	6.03
Calculated Rank	3	1	2	4	5

4.6 Technical Debt Visualisation

The dashboard created for visualisation showing the Open-Source project *Excalidraw*[1] will be presented in Figures 4.2 to 4.7 going from the top to the bottom of the dashboard. This project was selected in order to keep the privacy of the case company's source code and because of its similarities in structure and programming languages. The graphs in the dashboard are fully interactive meaning you can zoom, hover over data points to see more information and click on data points to hide and show them while the plot resizes to best fit the current data points. It is also easy to set parameters limiting min or max file size and complexity and to remove files that are not relevant for the visualisation. For this visualisation the minimum file size is set to 25 LOC.

Figure 4.2 shows the top view of the dashboard where the project which should have its metrics visualised is selected. Below the drop-down menu, a text box shows the current projects TD index and compares its ranking with other projects that have been scanned.

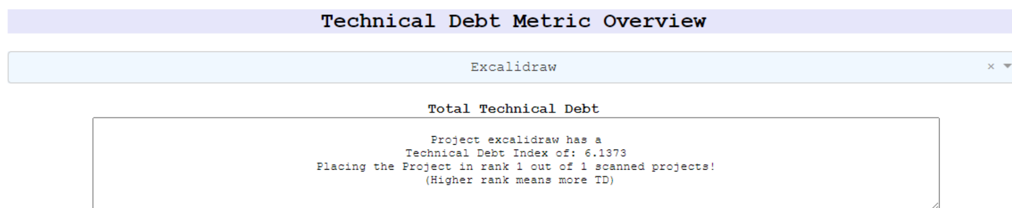


Figure 4.2: Top view of the dashboard.

Figure 4.3 shows file-level metrics. The y-axis shows the number of commits per file in the last 12 months while the x-axis shows cyclomatic complexity per LOC. The circle size depends on the LOC of the file while the colour ranges from red to green depending on the number of functions per LOC in the file. This graph showcasing the more important metrics can be used to find outlier files as a first step in deciding where to start a refactoring.

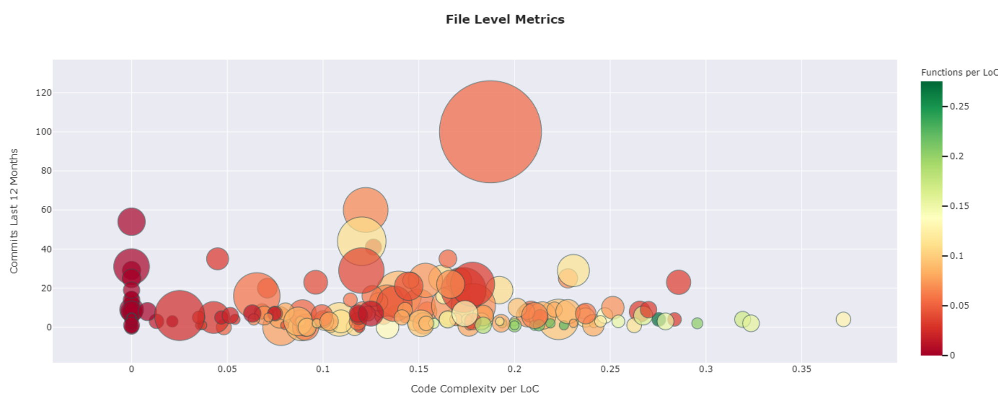


Figure 4.3: File level metrics visualising commits, complexity, size and functions per line of code for each file in the project.

[1] `https://github.com/excalidraw/excalidraw`

Figure 4.4 shows an example of total project metrics to get a better overview of how the project has been evolving. The graph includes both the codebase's size and tasks started and completed in the last 12 months. These are simulated values as tasks completed and started could not be collected for Excalidraw. Finally, the cyclomatic complexity per LOC can also be toggled on.

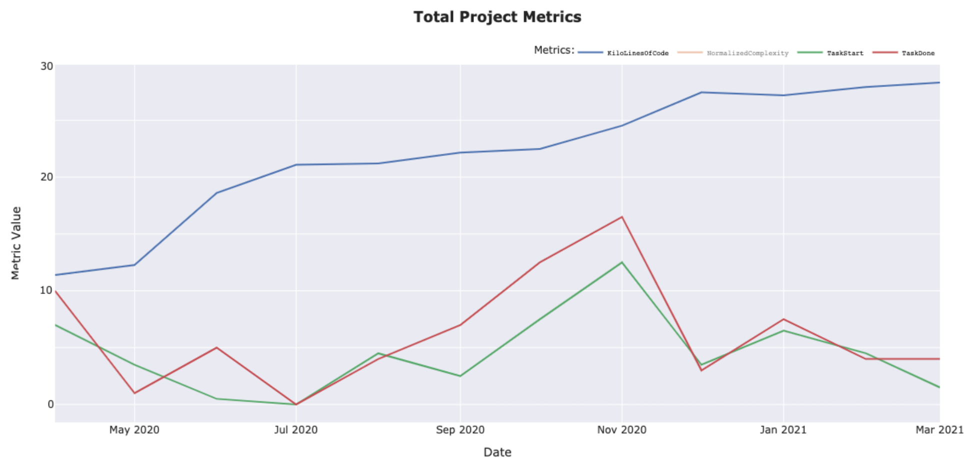


Figure 4.4: Total project overview showing how the project size, tasks started and tasks completed has evolved the last 12 months.

Figure 4.5 shows a treemap graph of the files in the project and their structure, where the file rectangle size depends on the file LOC and the rectangle colour ranges from green to red depending on the amount of calculated TD in the file. This graph is used to quickly show what areas of the system that has the most TD.

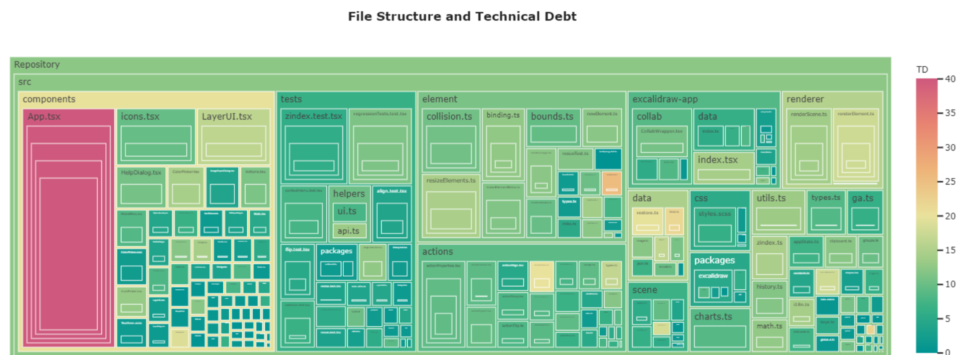


Figure 4.5: Tree graph showing the project structure and which files contain the most TD.

Figure 4.6 shows the calculated TD per file where the files are sorted depending on the calculated TD. They also range in color depending on the file LOC. This graph can be used to easily parse through the files with the most TD working your way down. It also shows how the TD is divided in the project; exponentially, linearly etc.

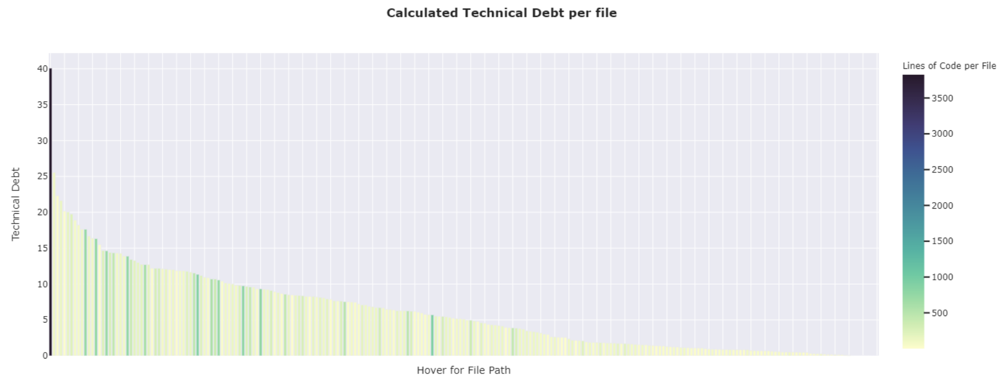


Figure 4.6: Bar chart showing the files ranging from left to right depending on calculated TD.

Figure 4.7 shows spider diagrams of the five files with the most TD in order, from left to right with the file names displayed above. The spider diagrams do not include the weighing used to calculate TD and instead show the normalised values. This graph can be used to see what metrics made the files rank in the top five and to simplify potential refactoring.

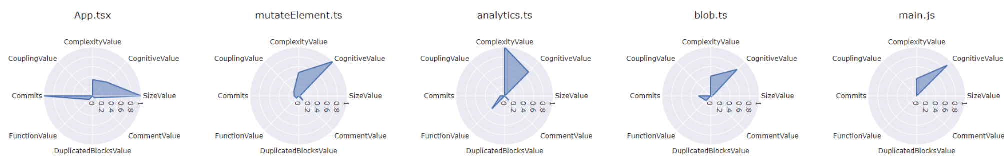


Figure 4.7: Spider charts of the five files containing the most TD.

4.7 Implementation in the Development Process

As a way of combining the findings from the literature study and the results from the TD visualisation, we have proposed an approach for the organisation to implement in order to manage TD. The implementation approach includes several activities and the full flow chart of these are presented in Figure 4.8. The approach includes three parts: measuring, visualising, and information sharing. The first two parts includes measurements of SM and visualisation using dashboards, and the third part includes a TD list, knowledge sharing within the team, defect data about the product and other relevant information.

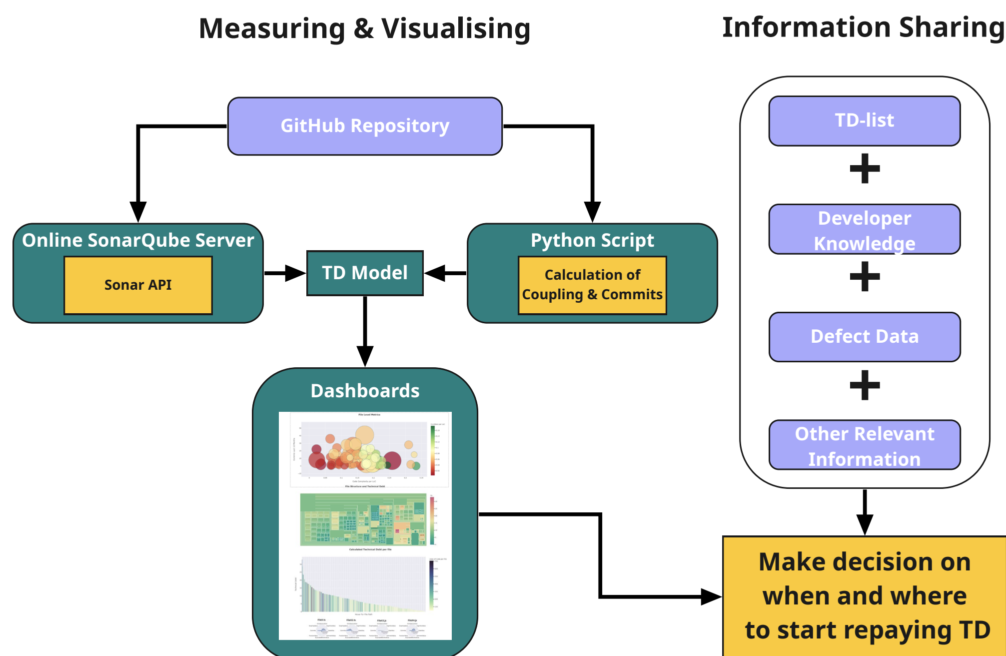


Figure 4.8: Implementation of TD in a team

4.7.1 Technical Debt List

The first activity is the tracking of TD items using a list where the proposed approach is through the already used issue tracking software. As presented in Section 2.2.1 and Table 2.7, each TD item will be provided with a description and ratings of principal, interest rate, and interest probability. The principal and interest rate are measured in person-hours and interest probability is measured in per cent. The TD list has been tested in the team, see Figure 4.9 and 4.10, by introducing it in the teams issue tracking software. When the developer finds a TD item, it should be added as a sub-task inside the task called *Technical Debt* and provided with a correct description as seen in the red box in Figure 4.10. For each sub-task, historical data and developer experience is proposed to be used as the basis to estimate principal and interests. The list of all sub-tasks is inside the red box in Figure 4.9 along with guidelines in the green box for how to add a new TD item.

Technical debt

Edit | Comment | Assign | More | Backlog | In Review | Workflow

Details

Type: Story
Priority: Low
Component/s:
Labels: None
Accept. criterias: EMPTY
Definition of Done: EMPTY
Epic Link:

Status: BACKLOG (View Workflow)
Resolution: Unresolved

Description

Please provide this information for each created TD item!

Description: Short description of the TD item

Principal: Estimation of how much time (weeks-days-hours) it would take to remove the debt

Interest Rate: Estimation of the extra time that has to be paid over time if the TD item is not repaid (weeks-days-hours).

Interest Probability: The risk of the interest actually becoming a problem in the future (percent %).

Attachments

Sub-Tasks

1. Create unit testing for component X — BACKLOG

Figure 4.9: The full TD list in the issue tracking software with one item added (red box), and guidelines for how to add a new TD item (green box)

Create unit testing for component X

Edit | Comment | Assign | More | Backlog | In Review | Workflow

Details

Type: Sub-task
Priority: Medium
Component/s:
Labels: None
Accept. criterias: EMPTY
Status: BACKLOG (View Workflow)
Resolution: Unresolved

Description

Description: Component X in File Y needs to be unit tested

Principal: 3h

Interest rate: 2h

Interest probability: 40%

Figure 4.10: Example of a TD item in the issue tracking software

4.7.2 Training of Developers

As a part of the implementation, we have conducted several presentations to stakeholders from different teams at the case company. The purpose of the presentations was to share acquired knowledge about TD and as a part of making the teams more aware and aligned and to show how each team could use and take advantage of the thesis work. The presentations were held digitally and lasted for one hour each. The presentations included both theoretical information about TD and findings from the literature study as well as information about the proposed implementation of working with TD and visualising it. The presentations were appreciated by the employees and there was an interest from several teams to implement the measurements and visualisations on more projects.

Chapter 5
Discussion

In this Chapter, we discuss the method, the results, and threats to validity, both regarding their outcomes and their possible weaknesses. The results that are discussed include the interviews, the work done at the case company, the TD model & visualisations, and finally the outcome of the evaluation.

5.1 Interviews

The interviews showed that there was a lot of TD in the case company that was very similar to the types discussed in the literature study. The reasons for acquiring this TD were also very similar to the ones discussed in the literature study. Reasons such as lack of knowledge, time pressure from markets, and prioritising different things other than TD prevention or TD reduction, were lifted by the interviewees. Due to how similar the TD issues were at the case company compared to the theory it both showed that the literature has come a long way defining TD and that the case company was very suited for a master's thesis like this.

The case company also had several best practices already in use as well as a few improvement suggestions. This both reduced what we had to recommend and limited the thesis scope to some extent. No suggestions or metric measuring needed to be done regarding code reviews, having a positive attitude towards TD and sticking to the standard coding language notation. Some of the improvement suggestions were also added to the recommendations left to the case company for further TD improvement after the thesis.

5.2 Selection of Metrics

The selected metrics differed quite a bit from the desired ones, due to the reasons previously explained in Section 4.2. This led to us having to estimate code smells using more general metrics compared to the more detailed definitions found in the literature study. However,

we still perceive that the selected metrics capture most code characteristics that relate to TD such as complexity, duplication, size, understandability, and importance. Unfortunately, the lowest level the metrics could measure at was on file level, it would have been better to be able to identify specific methods in files that contained more TD compared to the others. This was however not possible using SonarQube, and most files were of the layout that they contained one method that was much larger than the others. This meant that once the file had been identified it was rather simple to locate which area of the file the deeper analysis and refactoring should begin. This was also due to the nature of the projects which had a structure where each file, in general, had a single task which meant that it did not need to be further divided.

Pearson's correlation test varied between the projects where strong correlations between metrics could be found in the back-end project which could not be found in the front-end project. This could be explained by the lower sample size for the back-end that had less files and LOC compared to the front-end. It could also be explained by the front-end and the back-end projects having different tasks such as the back-end being more logic-heavy and having more short functions.

Beyond this, the metrics seemed to be rather independent based on the correlation analysis, specifically for the front-end project. Most metrics had a correlation factor below 0.3, with the exception of number of Functions, Cyclomatic Complexity and Cognitive Complexity. This could be explained by SonarQube's implementation of Cyclomatic Complexity where it increased by one for each function. We still found the number of Functions to be a relevant metric even though this strong correlation existed since it was an important metric to evaluate the code smell named Brain Method.

Although Cognitive Complexity and Cyclomatic Complexity had a very strong correlation factor (0.69) we decided to include both metrics in the model. This was because Cognitive Complexity is a variation of Cyclomatic Complexity that aims to remove some weaknesses from the former. A simple, easy-to-read switch case could have a Cyclomatic Complexity of eight where it has a Cognitive Complexity of one. We decided to use both complexity metrics even though they had a strong correlation in both projects due to Cyclomatic Complexity's proven experience, and not to have simple switch cases to skew our model.

So although there was strong correlation between some metrics, especially on the back-end project, all metrics are still believed to have their place in the model and contribute with something the others metrics can not capture.

5.3 Technical Debt Model

In this section, we discuss the model composition, as well as the precision of the model based on the developers' evaluation. Finally, we discuss the shortcomings of the model.

5.3.1 Model Composition

The decision to make the model produce a single value was not certain at the start of the model design. The literature study had shown that TD is a broad term and it is not easy to describe it with a single value. Despite this, we needed some way to rank the files, and

eventually, it was decided that a single value was the best option for this. The single value is, however, produced from a combination of several different metrics. The visualisations in the form of the spider diagrams and the file-level metric plot also help in showing the broader picture.

The weighing of the parameters went relatively smooth. It was not too much work checking if the files were ranked in an accurate order. The first weighing based on an estimation of each metrics' appearance in the literature study was good, but for these projects that weighing had too much impact on the size and number of commits. The analysed projects often received many smaller commits adding a few lines of code to include a new market or had long text files. These were files that did not need individual refactoring but rather showed that they could be solved differently. After reducing the size and commit weights and making some other minor modifications to the other metrics the model was finished.

5.3.2 Model Precision

The model precision was off for the back-end project where only three out of six individual votes were placed on files the model predicted as having *high* TD. Especially, *SmallFile3-be.ts* received no votes although the model had ranked it as having high TD, mostly because it had a lot of duplicated code. However, this was intended since it was mapping from one data structure to another. It is worth noting that the file with the highest calculated TD in the back-end project was not included in the evaluation since there were no other files of similar size.

Worth noting is also that the two low TD files that were chosen by the developers were chosen for reasons that our model did not take into account. The reasons were *"bad file name"* and *"many developers who contributed to the code"*. Our model did not analyse names, and although the model had commits included, it did not take into account who did these commits. The number of different developers is a metric that would have been interesting to measure, but questions following this measurement such as *"Is a certain developer linked to more TD?"*, is something we wanted to avoid in this thesis for ethical reasons.

The model performed better for the front-end project where seven out of nine individual votes were placed in the predicted files. One of the votes that was placed in our *low* category hinted at a larger problem: *"A copy of this content is not something we should maintain - it should exist in some centralised repo[sitory]"*, something that is hard to measure with SM.

Another interesting aspect is that a few of the files found to be containing high amounts of TD were old files that had been rewritten in new versions (V2). They had been rewritten due to *"having messy code"* and being a *"rushed MVP [Minimum Viable Product]"* which both are indicators of TD. However, some other files also had received new versions which the model did not rank as high.

The second step of the evaluation where the final five files were agreed upon turned out better. Four out of these files had been identified as having high amounts of TD from the model, although the dollars were spent a bit different compared to our calculated TD. The participants were asked why the back-end project received fewer dollars whereas they responded that the *"the technical debt in the back-end project would require less work to remove"*, even though the metrics showed similar TD values as for the front-end project. This further indicates what has been discussed before, that it is hard to compare different projects and sectors, perhaps the back-end and the front-end projects should be analysed independently.

This is something that the normalisation based on the project themselves indicate, where the TD values better follow the developers opinions.

When looking at the ordering for the files with high TD it can be seen that they were not placed in the correct order according to the model. They did however have very similar calculated values meaning that there is not that much separating them, hinting that the model lacks some finer precision. Worth noting is that when looking at the motivation for the ranking of *LargeFile3-fe.tsx* which has the highest calculated TD, the developers indicated that it had already been refactored. *"LargeFile3-fe.tsx received fewer dollars since a new version already had been created"*. Perhaps it would have been placed first if the new version did not exist and we could argue that it already was placed first since it had been refactored into a new version, compared to i.e. *MediumFile3-fe.tsx* which has not been refactored.

The data from the evaluation shows that the model has somewhat good precision. A model like this, or improved with further metrics, can be a good way to automatically find files in larger projects that can be further analysed. As seen in the evaluation, there are several aspects identified by developers that the model can not evaluate. Therefore, for best results, it should not be used as a single source for decision making. Using it in combination with a TD-list and stakeholder feedback is strongly recommended. The versatility of the model, such as allowing more metrics to be added, weights to be changed, and allowing different ways to normalise, increases its usability in other projects.

5.3.3 Model Shortcomings

The model had quite a few shortcomings. Firstly, identifying larger problems such as that content should be moved to an API, file names being incorrect, lack of unit testing or that the architecture is sub-par is not possible with the model. The question is if it was possible to gather any metrics desired, would some combination of these be enough to answer these higher-level questions? Or will always some human intervention be necessary?

It would also have been preferred if the model could identify specific parts of the files where the TD was concentrated. This shortcoming is much due to the limitations of the analysed projects and code analysis software. SonarQube did not measure metrics on a lower level than file level. Identifying specific methods or classes is possible using other tools that have lower-level metrics and is something that is recommended for further improvement.

Finally, the question of how applicable the model is for web development projects needs to be discussed. Much of the metric theory the model is based on is built from papers analysing more back-end heavy Java projects. Web development often has a more predefined structure with components that almost always are used. It often has much loose coupling between files and new components are added frequently which could mean that the commit metric is not as important for web development. There could also be other metrics that have a much larger impact on web development that was not discussed at all in the literature that was studied. To try and negate this shortcoming we had several discussions during the key metric selection phase with the developers about what metrics they thought had the most impact.

5.4 Evaluation of Method

The method steps were based on goals proposed by the case company and as we wanted to make sure that the thesis fulfilled all the goals, it was decided to follow this approach. The actual method for each step came as a combination of logic; this was the logical way to approach the problem, and from looking at the literature and seeing what steps were necessary when working with SM, which is, for example, where the GQM method came in. All steps were necessary in our opinion, although they could have been carried out in different ways.

The research questions were used as guidelines for choosing each step. A literature study and interviews with stakeholders was considered the best option for answering RQ1 as it provided a broad understanding of the subject both in relation to other companies' working methods and the case company's current situation. For this step, we did not consider other approaches since it was recognised as necessary steps to start the thesis work with. The interviews and literature study also gave a good foundation for RQ2 together with the GQM method, which was recognised in the literature.

The final steps of the method, measuring and visualising, was a combination of the result from the literature study, available resources and our knowledge in the area. It was realised during our research that our visualisations could not compare to other visualisation tools available on the market. It was however decided that we wanted to create dashboards for the data we retrieved and use it when introducing the implementations. If the team and case company is satisfied with the visualisations, they can decide to go further with the measurements or start paying for a subscription of a tool.

5.5 Technical Debt Visualisation

The dashboard used for visualisation received a positive response from stakeholders at the case company. The ability to both get an overview of the entire project and see metrics for the specific files was elevated by the stakeholders. That it also contained easy to grasp colours from green (good) to red (bad) showing important metrics and the calculated TD values allowed for an easy learning curve. The possibility to also see what characteristics placed the top five files first with the spider diagrams was appreciated. Although there was some relevant feedback that the spider diagrams did not include the weighing for each parameter, which could be a bit misleading.

The visualisation also included the total project metrics LOC and tasks started and completed per month. At first, we wanted to include some of the other code metrics used on file level in this graph, however, after doing a correlation test using the Pearson method it was seen that on project level all of those metrics strongly correlated to size.

We had an aspiration to have more agile metrics beyond the number of tasks completed and started, to have a better overview of how the development speed was progressing. This was however difficult to collect in an easy manner that would be recommended to the case company in the future. We could only export the tasks tagged for the project to Excel where it further was manually processed and summarised. This meant that some tasks fell under the radar if they were not tagged correctly. One of the reasons for this was that several teams had their tasks in the same backlog and not all of them were tagged to the corresponding project.

They neither had any tags that corresponded to the front-end or back-end project leading to us having to combine the tasks for both of them and show the same for both projects.

Other than that, the dashboard was positively received and a lot of interest was shown by different stakeholders wanting to analyse their projects and have their TD visualised. Which was done by us delivering both the processed data and visualisation tool to them and creating a manual on how to use the software for scanning of metrics and visualising them which was given to the case company.

The dashboard could have included the TD list that was implemented in the issue tracking software instead, see Figure 4.9. However, due to how the teams currently were logging their work daily whereas our dashboard was suggested to be updated once a month, it would be too much of a hassle to have to log it in there. Although, a list implementation in the dashboard reading from the TD list in the issue tracking software, could be a possibility.

5.6 Implementation in the Development Process

Throughout the thesis work, the several presentations of theory, as well as demos of our work and the dashboard, has received positive feedback. There is however a problem with presenting large amounts of information and recommendations about a subject that the listeners do not know too much about. It might feel overwhelming and there is no real knowledge of where to even begin. To try to make sure that the work in this thesis is used, two precautions were made. The first precaution was the creation of a manual on how to use the software to both gather and visualise metrics as well as simplifying the steps into a smoother process, and showing some teams how to use it.

The second precaution was to write a recommendation list where the most critical best practices and background theory was listed. The list also included steps on how to implement it into the development process.

We believe that these precautions provide a boost to the initialisation of working with TD. Simple key steps are easier to both implement and make decisions of, and the easy to grasp feedback from the dashboard also makes it easy to work with. Hopefully by having made these precautions some of our recommendations are implemented in the development process and are further improved as experience is built.

5.7 Threats to Validity

The largest threats to the validity of this thesis can be found in the evaluation stage. Before the evaluation, the developers that were to rank the TD raised the issue that they were not sure how to define TD since it is a very broad term. To help their understanding they were sent a presentation made by us containing the information found in Chapter 2.1. This could be interpreted as skewing the results to some extent. However, our model did not contain all parts of the presentation, there were no instructions on how to rank TD and in the end, the developers' stated selection criteria differed from this presentation.

The low sample size of the evaluation and how it was conducted can also be seen as a threat. Instead of us selecting 15 files, a more correct result would probably come from having

all developers who had worked on the project go through each file together and finding the ones with most and least TD. This was, however, not possible due to the large amount of time it would take, as well as that not all developers who had worked on the projects were available. Several projects should also have been evaluated in a similar way to better capture the preciseness of the model, this was also not possible due to time constraints. However, to increase the validity the developers received a lot of time and space to both select their files and discuss their selections and reasoning amongst them.

The fact that the TD model was mainly based on theory from papers having studied TD and maintainability for Java applications compared to the studied web development projects in this thesis is also something to factor. Although the difference in project type and programming language, the basics regarding TD and maintainability should not differ too much between them. The metrics were also selected to cover the most crucial and general characteristics of code that exist in all types of project to decrease this threat. This means that the model and way of working in the thesis will work for varying projects and organisations.

Chapter 6
Conclusion

Throughout this master's thesis we have aimed to get a deeper understanding of both the theory of TD, with a literature study, and practical examples of it in real software development, with the help of interviews. One key finding from this was that TD is a broad term where the definition, to some extent, differs depending on the author. The interviews showed that many developers have insights in TD issues and best practices without being that familiar with the actual term and that these insights differed a lot from developer to developer. It will require a lot of effort to start working with TD, but as experience is built it will become easier.

Another key finding was that measuring and identifying TD in software development is not an easy task since different types of debts are identified in different ways and are more or less complex. For example, it is much harder and complex to identify and refactor ATD for a system compared to doing the same for a copy pasted method in a system module.

In this thesis, we have attempted to measure and visualise TD with the help of SM found to be related to TD and maintainability. A TD model has also been created based on these metrics. Even though we could not collect all metrics that were desired due to the limitations of the thesis work, the results still hint that the current model and dashboard can be used to both identify TD and direct refactoring work. It was also found that the model mirrors some of the TD identification made by developers who created the project and have a lot of insight. Although both the results and the literature study showed that SM are not enough on their own to measure TD, due to their complex and broad nature.

It has also been shown that a dashboard visualising TD is both possible and works as a good basis for discussion that helps the evaluation of implementation decisions and the describing of how different developers define maintainability.

Finally, how some parts of the organisation currently work with TD has also been studied and different recommendations regarding best practices and ways of working from the literature has been given from these.

Chapter 7
Future Work

To get a more complete picture of measuring TD with SM it would be interesting to see similar work using further or other metrics. Either by writing own software, purchasing software or analysing a project in a different programming language with more software options to collect these metrics. Using SM on a larger scale of projects to get a better result of their correctness and ability to compare different projects, especially in web development is also an interesting topic. It would also be interesting to try using metrics on method-level to better identify code smells.

In the times of machine learning, setting up a linear regression model using metrics and using stakeholders with knowledge of the project as feedback for the model is something that could be further analysed. This would probably lead to a more precise model with improved identification results but would need more time and effort from the developers.

The difference in efficiency to remove TD with a perfect SM management method compared to none is also something that has not been researched too much. Following "perfect" protocol and having metrics and visualisations for every aspect of the development process such as code metrics, architectural metric, deployment metrics and using these in combination with a TD list, developer knowledge etc. Then comparing how much more efficiently TD could be removed from a project using all these tools with not using any assistance.

Finally, the dashboard could be improved by both containing a TD-list and using the estimated principal and interest for each item in the total project debt estimation.

References

R. Alfayez, P. Behnamghader, K. Srisopha, and B. Boehm. An exploratory study on the influence of developers in technical debt. In *Proceedings of the 2018 international conference on technical debt*, pages 1–10, 2018.

S. Alliance. Overview: What is Scrum? `https://www.scrumalliance.org/about-scrum/overview`, 2017. Accessed: 2021-01-26.

A. Altvater. Development Leaders Reveal the Best Metrics for Measuring Software Development Productivity. `https://stackify.com/measuring-software-development-productivity/`, 2017. Accessed: 2021-02-04.

P. C. Avgeriou, D. Taibi, A. Ampatzoglou, F. A. Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, N. Moschou, I. Pigazzini, et al. An overview and comparison of technical debt measurement tools. *IEEE Software*, 2020. doi: 10.1109/MS.2020.3024958.

R. Bavani. Distributed agile, agile testing, and technical debt. *IEEE software*, 29(6):28–33, 2012.

J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. Limiting technical debt with maintainability assurance: an industry survey on used techniques and differences with service- and microservice-based systems. In *Proceedings of the 2018 International Conference on Technical Debt*, pages 125–133, 2018.

B. Cline. 5 TIPS TO WRITE MORE MAINTAINABLE CODE. `https://www.brcline.com/blog/5-tips-write-maintainable-code#:~:text=Maintainable%20code%20is%20basically%20the,the%20change%20could%20break%20something`, 2018. Accessed: 2021-02-06.

W. Cunningham. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1992.

E. da Silva Maldonado, E. Shihab, and N. Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, 2017.

E. Dietrich. The Human Cost of Tech Debt. https://www.infragistics.com/community/blogs/b/erikdietrich/posts/the-human-cost-of-tech-debt, 2016. Accessed: 2021-02-06.

R. J. Eisenberg. A threshold based approach to technical debt. *ACM SIGSOFT Software Engineering Notes*, 37(2):1–6, 2012.

N. Fenton and J. Bieman. *Software metrics: a rigorous and practical approach*. CRC press, 2014.

F. A. Fontana, V. Ferme, and S. Spinelli. Investigating the impact of code smells debt on quality code evaluation. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pages 15–22. IEEE, 2012.

M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

Y. Guo and C. Seaman. A portfolio approach to technical debt management. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 31–34, 2011.

Y. Guo, C. Seaman, and F. Q. da Silva. Costs and obstacles encountered in technical debt management–A case study. *Journal of Systems and Software*, 2016.

R. Jabangwe, J. Börstler, D. Šmite, and C. Wohlin. Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review. *Empirical Software Engineering*, 20(3):640–693, 2015.

R. Jeffries. Refactoring – Not on the backlog! https://ronjeffries.com/xprog/articles/refactoring-not-on-the-backlog/, 2014. Accessed: 2021-02-08.

J. Kasurinen, O. Taipale, and K. Smolander. Analysis of Problems in Testing Practices. In *2009 16th Asia-Pacific Software Engineering Conference*, pages 309–315, 2009. doi: 10.1109/APSEC.2009.17.

H. Kniberg. Good and Bad Technical Debt (and how TDD helps). https://blog.crisp.se/2013/10/11/henrikkniberg/good-and-bad-technical-debt, 2013. Accessed: 2021-02-08.

M. Lanza and R. Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

J.-L. Letouzey. The SQALE method for evaluating Technical Debt. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pages 31–36, 2012. doi: 10.1109/MTD.2012.6225997.

Z. Li, P. Avgerioua, and P. Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

A. Martini, T. Besker, and J. Bosch. The introduction of technical debt tracking in large companies. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 161–168. IEEE, 2016.

J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali. Searching for build debt: Experiences managing technical debt at Google. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pages 1–6. IEEE, 2012.

A. Mori, G. Vale, M. Viggiato, J. Oliveira, E. Figueiredo, E. Cirilo, P. Jamshidi, and C. Kastner. Evaluating domain-specific metric thresholds: an empirical study. In *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 41–50. IEEE, 2018.

S. S.A. Metric Definitions. `https://docs.sonarqube.org/latest/user-guide/metric-definitions/`, 2021. Accessed: 2021-04-06.

C. Seaman and Y. Guo. Chapter 2 - Measuring and Monitoring Technical Debt. In *Advances in Computers*, volume 82 of *Advances in Computers*, pages 25–46. Elsevier, 2011. URL `https://www.sciencedirect.com/science/article/pii/B9780123855121000025`.

D. Straker. The Hundred Dollar Test. `http://creatingminds.org/tools/hundred_dollar`, 2021. Accessed: 2021-03-15.

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

A. Tornhill. GOTO 2019 * Prioritizing Technical Debt as if Time and Money Matters * Adam Tornhill. `https://www.youtube.com/watch?v=fl4aZ2KXBsQ&ab_channel=GOTOConferences`, December 2019.

R. Verdecchia, I. Malavolta, and P. Lago. Architectural technical debt identification: The research landscape. In *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 11–20. IEEE, 2018.

K. Wiklund, S. Eldh, D. Sundmark, and K. Lundqvist. Technical debt in test automation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 887–892. IEEE, 2012.

J. Yli-Huumo, A. Maglyas, and K. Smolander. How do software development teams manage technical debt?–An empirical study. *Journal of Systems and Software*, 120:195–218, 2016.

N. Zazworka, R. O. Spínola, A. Vetro', F. Shull, and C. Seaman. A case study on effectively identifying technical debt. *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, pages 42–47, 2013.

N. Zazworka, C. Izurieta, S. Wong, Y. Cai, C. Seaman, F. Shull, et al. Comparing four approaches for technical debt identification. *Software Quality Journal*, 22(3):403–426, 2014.

M. Zhang, N. Baddoo, P. Wernick, and T. Hall. Prioritising refactoring using code bad smells. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 458–464. IEEE, 2011.

Appendices

Appendix A
Interview Questions

1. What do you do within CE?
2. Can you describe how the idea for the thesis came about?
3. How is the organization built? Especially around your team?
4. Can you describe your team/s? (history, size, roles)
5. Can you describe the product the teams are working with?
6. How do you work with technical debt today?
7. Who decides what the team should do regarding the implementation of new features or refactoring to remove technical debt for example?
8. How does the architecture work for your software, do you have any models or rules that are followed?
9. Does it happen that the software deviates from the architecture for different reasons?
10. What are the effects of TD?
11. How do you prioritize the repayment of different types of technical debts?
12. How is technical debt found/identified?
13. How much time is spent on refactoring?
14. Do you perform any measurements of technical debt?
15. What are the most important aspects of TD in your opinion, and what measures do you think are best for measuring these?

16. Do you have any other thoughts / ideas about TD and about the future work around TD?

Appendix B
In-depth Interview Questions

1. What team are you in and how big is it?
2. What is your role and responsibilities in your team?
3. Could you give a quick history of the team and product you are working on?
4. Describe your knowledge about technical debt?
5. Tell us about situations where the team have taken shortcuts and the effects of taking these shortcuts?
6. How does the team make decisions about technical debt?
7. Do you feel comfortable raising TD issues to the team?
8. How do you know if technical debt exists? Can you be more specific about where it comes up?
9. How is technical debt considered when writing code?
10. What kind of technical debt/issues do you most often find in the code/documentation?
11. What are the team's code refactoring procedures?
12. When you run into code that you find hard to maintain, which are primary reasons for this?
13. Could you describe the main steps in testing your code before going into production? Are there any standards?
14. Describe how the implementation of a new feature is conducted?
15. How much does the team use software metrics?

16. Are you currently measuring any metrics?
17. What software metrics do you believe have the most impact on visualizing TD or seeing code quality
18. Do you have any ideas on how the company should take care of managing, finding, reducing and paying shortcuts?
19. Do you have something to raise about the subject that we have not asked about?

Appendix C
Results from Goal-Question-Metric

Goals

1. Get a better insight and have a proper monitoring, both of the code itself but also how the team delivers the software.
   (a) Identify TD and the reasons for it to incur
   (b) Measure and visualize technical debt
   (c) Prioritizing between improvements and developing new features

Questions

1. How complex is the code?
2. How often does the code change?
3. How well is the code working? Both for customers but also for developers
4. How often are new releases made?
5. How much time is spent on maintenance and fixing issues?
6. Repay TD now or later?
7. How much duplicate code exists?
8. How large is an average method/class?
9. How often is documentation updated?
10. How organized is the test coverage?
11. How is the feature-release speed developing?

Metrics

1. Product Metrics
   (a) God Class
      i. Weighted Method Count (WMC)
      ii. Tight Class Cohesion (TCC)
      iii. Access to Foreign Data (ATFD)
   (b) Data Class
      i. Weight of Class (WOC)
      ii. Weighted Method Class (WMC)
      iii. Number of Attributes (NOA) - (NOAP) / (NOAM)
   (c) Brain Method
      i. Lines of Code (LOC)
      ii. McCabe's Cyclomatic Complexity
      iii. Maximum Nesting Level
      iv. Number of Accessed variables (NOAV)
   (d) Significant duplication
      i. Size of Exact Clone (SEC)
      ii. Size of Duplication Chain (SDC)
      iii. Line Bias (LB)
   (e) Intensive coupling
      i. Max Nesting
      ii. Coupling Intensity (CINT)
      iii. Coupling Dispersion (CDISP)
   (f) Shotgun surgery
      i. Changing Methods (CM)
      ii. Changing Classes (CC)
   (g) Number of defects
   (h) Number of defects reported by users

2. Process metrics
    (a) Developer reported TD-items
    (b) Number of released features
    (c) Number of completed tasks
    (d) Average time spent on a subtask
    (e) Cycle Time
    (f) Lead Time
    (g) Number of defects found in testing
    (h) Commit interval time
    (i) Number of commits
    (j) Number of commits per method
3. Resource Metrics
    (a) Developer reported TD-items

Appendix D
Metric Definitions

The three first metrics definition are from Lanza and Marinescu (2007) while the last two are from Mori et al. (2018).

- Access to Foreign Data (ATFD): Counts how many attributes from other classes that are directly accessed from the measured class.
- Weighted Method for Class (WMC): The sum of the cyclomatic complexity of all methods in a class divided by the number of methods.
- Tight Class Cohesion (TCC): The relative number of method pairs in a class that access at least one common attribute that is in that class.
- Coupling Between Objects (CBO): The number of classes that are coupled to a class by it calling methods or accessing attributes of the other classes.
- Lack of Cohesion in Methods (LCOM): Divides the pairs of methods in a class that do not access its attributes by the pairs of methods that access common attributes.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2021-06-10

EXAMENSARBETE Identification of Technical Debt in Code using Software Metrics
STUDENTER Erica Schillström, Dan Wahlin
HANDLEDARE Martin Höst (LTH)
EXAMINATOR Ulf Asklund (LTH)

Hur ska en organisation hantera och mäta teknisk skuld?

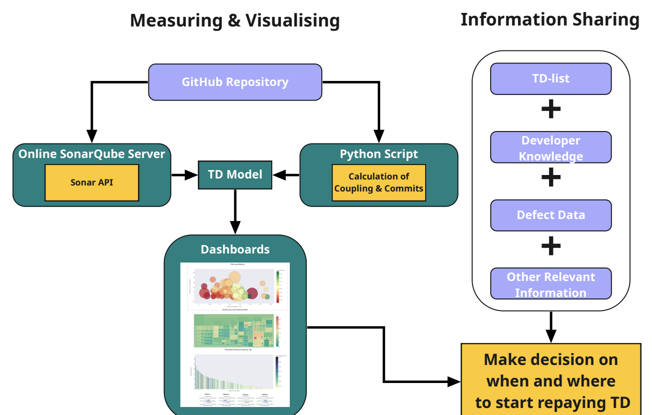
POPULÄRVETENSKAPLIG SAMMANFATTNING **Erica Schillström, Dan Wahlin**

Teknisk skuld förekommer i alla former av mjukvaruutveckling och om skulden ej behandlas rätt kan konsekvenserna bli stora. Detta arbete undersöker dels hur en organisation bör jobba med teknisk skuld och dels hur teknisk skuld i kod kan identifieras och visualiseras med en modell av mjukvarumätvärden.

Teknisk skuld är en metafor som används för att beskriva konsekvenserna av att ta en genväg i utvecklingsprocessen av ett mjukvarusystem. Genvägen ger ett team fördel i form av snabbare leveranser till kund, men en skuld kommer aldrig utan en kostnad och att ta för många genvägar utan att betala tillbaka dem kan leda till långsiktiga konsekvenser. Denna skuld ökar även risken att ett team tvingas spendera mer tid på att underhålla och åtgärda problem än att kunna bygga nya funktioner. Det finns således ett behov att hantera och mäta teknisk skuld för att fatta relevanta beslut kring refaktoriseringar och utveckling av mjukvaran.

I vårt examensarbete har vi byggt en modell med hjälp av åtta mjukvarumätvärden för att uppskatta mängden teknisk skuld i en fil. Denna modell kan användas för att identifiera och rangordna filer i ett mjuvarusystem som bör ses över för refaktorisering. Dessa mjukvarumätvärden valdes utifrån en litteraturstudie där samtliga steg av hanteringen av teknisk skuld undersöktes. Mätvärdena uppskattar egenskaper som komplexitet, storlek, viktighet, beroende samt underhållbarhet i koden. Modellen byggdes upp genom att vikta och normalisera de olika mätvärdena baserat på litteraturstudien.

Datan från modellen visualiserades i en dashboard som både visade individuella mått, samt ett uträknat teknisk skuld-värde. Dashboarden visade sig vara ett bra verktyg för att skapa diskussion samt snabbt få en överblick över projektet och hitta områden att analysera djupare. Arbetet resulterade även i fyra slutsatser gällande



organisationers arbete med teknisk skuld: De bör logga förekomster av teknisk skuld samt uppskatta dess allvarlighet, avlägga bestämd tid för att identifiera och refaktorisera teknisk skuld, uppmuntra till diskussion och utbilda utvecklarna i ämnet samt utföra mätningar av viktiga mått i koden. Rek. arbetssätt kan ses i figuren ovan.