Expertise
and insight
for the future

Jani Knaappila

Measuring Structural Software Quality

Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master's Thesis

11 November 2020

Author Title Number of Pages Date	Jani Knaappila Measuring Structural Software Quality 42 pages 11 November 2020
Degree	Master of Engineering
Degree Programme	Information Technology
Instructor(s)	Ville Jääskeläinen, Principal Lecturer

The purpose of this thesis is to research a way to measure quality attributes of software and find metrics that can then be used to improve the quality of software.

Software quality can be understood in different ways by different people. Therefore, software quality is first described from a standards point of view. Two categories for software quality metrics are identified and further investigated. Product-related quality metrics are metrics that are measured from a static code, a technic, which is known as static code analysis. Another category is then the process related metrics. Process related metrics measure how people develop software. Multiple studies have shown that metrics measuring how software is developed, are more accurate of predicting faults than static code analysis.

Three metrics were chosen to be measured: Code ownership, Code Churn and Code Complexity. It is described in detail how those are calculated in practice. Further, an aggregate metric is introduced to give a single number as a quality indicator. The framework where the solution needs to operate is defined, including software source code structure, software repository environment and issue tracking software. The software solution is then defined and was developed to measure these metrics.

This proof of concept solution was then used to measure metrics from a set of software components in a software repository. Results were then analysed and explained. One software component was chosen to be more closely examined, and from there several files with the highest value for an aggregate metric were analysed even in more detail.

Finally, the solution was evaluated by comparing predictions against real issues. Results were inconclusive but provide similar results as previous studies. These results are encouraging. There are several threats to validity issues, but this can be engaged with when this proof of concept is further developed.

Keywords	software quality, code ownership, code churn, code complexity

Contents

Abstract

List of Abbreviations

1 Introduction 1

2 Software Quality 3

2.1 ISO/IEC 25010 3

2.2 Quality Attributes 5

2.3 Product-Related Metrics 5

2.4 Process-Related Metrics 11

2.5 Faults and Failures 16

3 Method and Material 17

3.1 Design Process 17

3.2 Quality Metrics 18

3.2.1 Code Ownership 18

3.2.2 Code Churn 19

3.2.3 Code Complexity 20

3.3 Aggregated Metric 20

3.4 Software Components 22

3.5 Software Repository and Restrictions 24

3.6 Evaluation Method 25

4 Solution Development 26

4.1 Requirements 26

4.2 Architecture 26

4.2.1 High-level Overview 27

4.2.2 Measurement Application Design 28

4.3 The Reasoning for the Design Decisions 32

5 Results and Analysis 33

5.1 Quality Metric Results 33

5.2 Detail Analysis of Component B 36

6 Evaluation and Conclusions 39

6.1 Threats to Validity 40

6.2 Conclusions 41

References

List of Abbreviations

CSV	Comma Separated Values
IoT	Internet of Things
SQuaRE	The series of standards ISO/IEC 25000, also known as SQuaRE (Systems and Software Quality Requirements and Evaluation)
SW	Software
UML	Unified Modelling Language
VCS	Version Control System

1 Introduction

Quality of software generally has three aspects: functional quality, process quality, and structural quality [1]. The user of the software experiences the functional quality of software, how well the software satisfies the user needs. Process quality is the quality of the software development process, how well the software team follows processes and can deliver their promises. Furthermore, structural quality is the degree of how well the software is designed, and this is experienced first-hand by the development team.

Process quality is measured by process or quality management, was the software delivered on time with the functionality committed to, and how it can be improved next time. Functional quality is measured by how well the software does what it is required to do; measuring this is the responsibility of software quality assurance. Structural quality, on the other hand, is not easy to measure, and it is sometimes even totally neglected, despite it being subject of study since the 1970s.

The goal of this thesis is to improve the software quality at Silicon Labs. Silicon Labs is a semiconductor company, which provides integrated circuits and software for the Internet of Things (IoT) devices. The quality of IoT devices is paramount as they are standalone devices that need to operate for years without interrupts. Thus, many resources have been put on improving quality. Functional quality is improved by increasing functional testing and process quality is improved by following standards and improving processes. Structural quality can be divided into two parts: Quality of software product and quality of the software development process. The quality of software products is improved by applying existing measurement tools, and to detect possible quality issues from software code, and then correct those. However, these tools do not capture the software development process quality; thus, there is a need for a tool to measure how well was the software designed.

The research question aims to capture that need:

> How to measure the structural quality attributes of software components in such a manner that it can be used to improve the quality?

The outcome of this thesis is a proof of concept solution, showing how software development process quality could be improved by measuring specific metrics. To limit the scope of the thesis, the solution is using generally available off-the-shelf components

Metropolia
University of Applied Sciences

and methods. Although the literature shows that there exist measurement metrics using advanced methods such as AI, ML and Fuzzy logic, these were excluded from this proof of concept. However, the proof of concept can be later to be extended to include these methods.

There exist similar kind of studies, where the aim was to build a tool to measure software quality [2] [3] [4]. Nevertheless, those concentrate only on software product quality.

This thesis is divided into several sections. First, the current state of the art is examined, related studies are listed, and quality attributes which they have introduced are explained. Difference between product and process-related metrics is explained.

Next chapter describes a set of quality attributes that are chosen to be measure, and how they are measured. This chapter also describes the environment where this solution needs to operate.

Solution development chapter provides the requirements and how those requirements are fulfilled in architectural design. This chapter also explains the technical implementation of measured the metrics and how data is accessed and provided for others.

The solution is then used to analyse the Silicon Labs software repository. Several components are selected for investigation. Results are shown and discussed in detail, and what can be seen from the results. One component is selected to be investigated even further to find out if there is something to be improved.

The last chapter discusses what has been found out, how they are validated, and what are threats to validity. Moreover, what improvement and future work are planned for this proof of concept solution.

2 Software Quality

Software Quality is a concept, which is understood differently by different people [1]. It is sometimes understood that software without defects or bugs is of good quality. Counting number of bugs is quite an oversimplification, as software quality is more than that, as will be explained further in this chapter.

David Chappell defines three different quality aspects for software; they are functional quality, structural quality, and process quality [1]. He defines these that functional quality tells how well the software meets the defined requirements of the system, and structural quality means how well the software code is structured. Moreover, process quality means how well the software development team develops the software, for example, staying in allocated time and budget.

These functional and structural qualities can also be defined as functional and non-functional qualities. Keshavarz et al. define a functional requirement as fundamental actions that must take place and non-functional requirements as qualitative requirements of the system [5]. Thus, functional and non-functional qualities then fulfil these requirements.

Len Bass and Paul Clements argue against separating system qualities to functional and non-functional categories [6, p. 66]. They give an example of engine control; how can functional quality be correctly implemented if the non-functional quality of timing behaviour is not also considered.

2.1 ISO/IEC 25010

ISO/IEC 25010 “Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models” is a standard defining the quality models for computer systems and software products. Its definition for software quality is the following: “the degree to which a software product satisfies stated and implied needs when used under specified conditions”. It also defines two quality models: “quality in use” and “product quality”- models. Product quality model is relevant for the topic of this thesis and is displayed in Figure 1. [7]

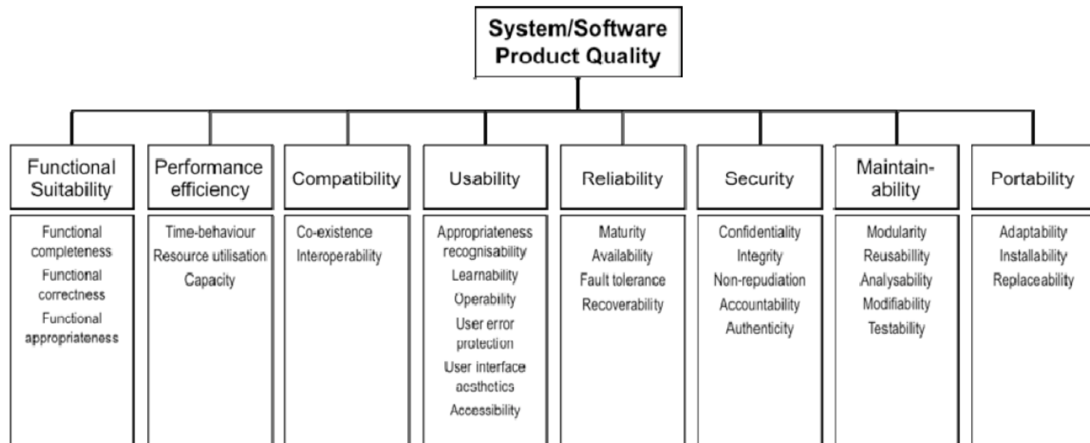

Figure 1. ISO/IEC 25010 Product Quality Model. [4]

The product quality is categorised into eight characteristics. The first characteristic "Functional Suitability" defines the degree to which the product or system provides functions that meet stated and implied needs. Performance efficiency characteristic covers the resource usage of the software. Compatibility characteristic is about the interoperability of the system with other systems. Usability characteristic is the machine-human interface aspect of quality and general usability of the system. Reliability characteristic defines how the system performs specified functions under specific conditions for a specific time. Security characteristic covers data protection and access to data. Maintainability characteristic is about modifiability of the system. And portability characteristic defines the efficiency with which a system can be moved from one environment to another. [7]

Functional testing of the system aims to cover all these quality characteristics of the system. It is relatively easy to automate the testing of functional completeness of the system; does it do what it is required to do. Also, performance efficiency is easy to automate; Does the system do what is required to do in the time provided. But there are quality characteristics of the system that are not easy to automate and thus easily skipped in the testing of the system. For example, maintainability characteristic of the system has a human component and cannot be automated.

The maintainability characteristic is defined by the ISO/IEC 25010 as "degree of effectiveness and efficiency with which a product or system can be modified by the intended

maintainers”. Maintainer in this context means the developers who are developing software and adapting it to changing requirements.

2.2 Quality Attributes

The quality attribute is defined by ISO/IEC 25010 as “*Inherent property or characteristic of an entity that can be distinguished quantitatively or qualitatively by human or automated means*” [7].

Like mentioned before, some of the quality characteristics or attributes are not easy to measure automatically. These two different kinds of quality attributes can be categorised either as product-related or process-related attributes. Graves et al. define product-related as something that is taken as a snapshot of the software and process as something that is a change in the software [8]. Henderson-Sellers further discusses this categorisation in the book “Object-Oriented Metrics: Measures of Complexity” [9].

A product-related metric is measured from source code itself. Different methods for measurement for product related metrics and history of measurement is described in the next section.

Process related metrics are then measured from changes in the source code. These metrics can be understood to measure how source code was implemented. Process related metrics are the most important metrics for this thesis and are further discussed in a later chapter.

2.3 Product-Related Metrics

One way to find metrics for software quality is analysing the source code of the software. This method is called a static code analysis, and it uses another software to analyse the software source code. Static code analysis has a long history, and the first static analysing tools were used to indicate the correctness of software source code against the required standard [10].

Cyclomatic Complexity

Metrics to measure software quality attributes is continually being researched [11]. One of the first ones was Cyclomatic Complexity developed by McCabe in 1976 [12]. Cyclomatic complexity, as the name suggests, gives metrics about the complexity of the source code or how many individual paths there are through the software.

A simple example of cyclomatic complexity is shown in Figure 2. This example shows how a single branch in software creates two distinct paths in the software. One path is taken when the comparison is true and another one when it is false, giving, in this case, a cyclomatic complexity of 2.

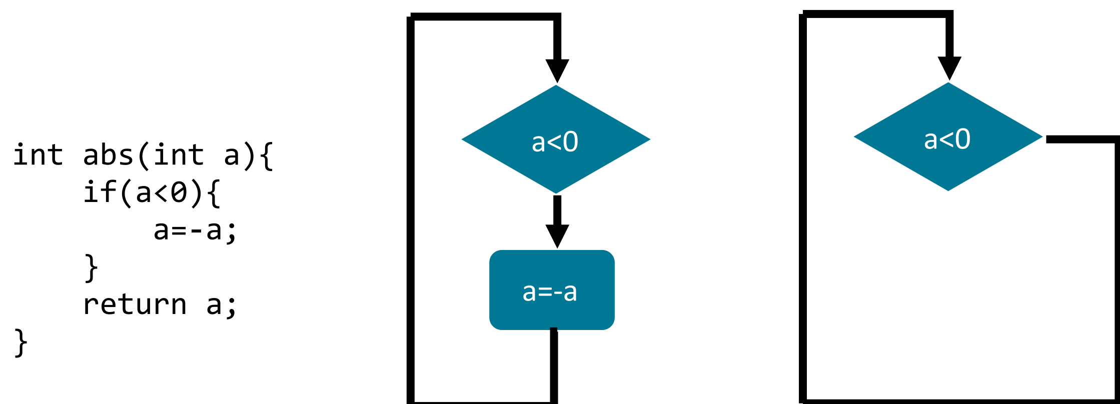

Figure 2. A simple function where there are two distinct paths through the function, giving cyclomatic complexity of 2

Watson et al. claim that if cyclomatic complexity metric is too high, the software will be more prone to errors, harder to understand, harder to test and harder to modify [13, p. 15]. This is disputed by Oram et al., they argue that lines of codes are linearly related to cyclomatic complexity and questions, even the validity of measuring it. Instead, they suggest that a simple number of lines of codes should be considered [14, p. 140].

However, cyclomatic complexity continues to be a popular metric and is well understood by stakeholders in the software development organization [15].

Halstead Complexity Metric

Halstead complexity metric [16] measures the computational complexity of the software by counting the number of operators and operands in the software. Figure 3 shows the same example as before, with lists of unique operators and operands.

```
int abs(int a){
 if(a<0){
 a=-a;
 }
 return a;
}
```

Unique operators:

```
abs, int, if, (), {}, <, -, =,;
```

Unique operands:

```
a, 0
```

Figure 3. Same simple function with operators and operands listed

From this example number of unique and total operators and operands are calculated. These are then used as parameters for Halstead complexity metrics. These parameters and metrics are shown in Table 1. Halstead metrics contain different kind of metrics from program length to even the time required to program. Halstead complexity metric is the first metric that tries to predict the number of faults in the software.

Table 1. Halstead complexity metrics example

$\eta_1 = 8$	Number of unique operators
$\eta_2 = 2$	Number of unique operands
$N_1 = 12$	Total number of operators
$N_2 = 5$	Total number of operands
$\eta = \eta_1 + \eta_2 = 10$	Program vocabulary
$N = N_1 + N_2 = 17$	Program Length
$\widehat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 = 26$	Calculated estimated program length
$V = N_1 + N_2 \approx 56.47$	Volume
$D = \frac{\eta_1}{2} \frac{N_2}{\eta_2} = 10$	Difficulty
$V = DV \approx 564.73$	Effort
$T = \frac{E}{18} \approx 31.37s$	The time required to program (s)
$T = \frac{E^{\frac{2}{3}}}{3000} \approx 0.02$	Number of delivered bugs

Information Flow Metric

The software can also be measured by looking at how information flows through the software. This information flow metric was developed by Henry and Kafura [17]. It measures the fan-in and fan-out of the functions. Fan-in is defined as how many are calling the function under measurement and fan-out then how many other functions are called from the function under measurement; this is shown in Figure 4.

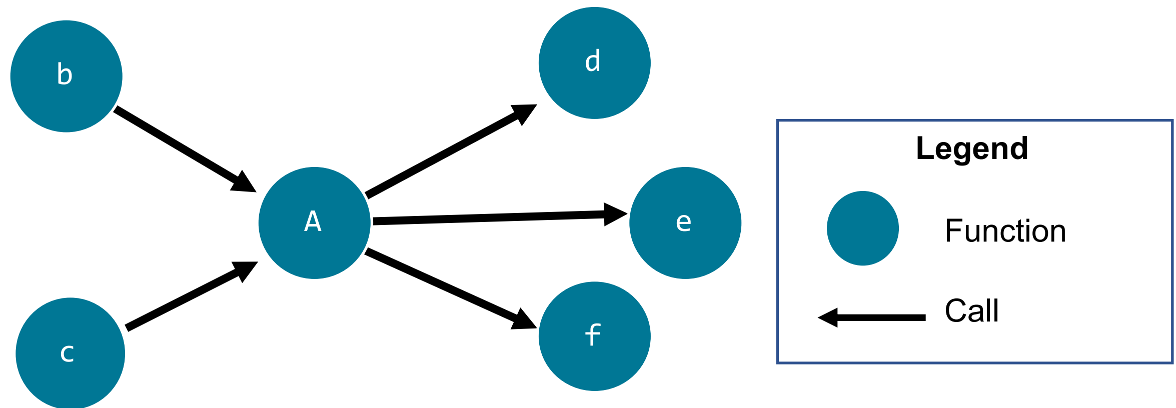

Figure 4. Call graph for A. Giving fan-in of two and fan-out of three for A

Henry and Kafura then defined a complexity value for a procedure, shown in Formula (1). This is calculated from previously mentioned fan-in and fan-out metrics.

$$Complexity\ Value\ for\ Procedure = Length * (fan\text{-}in * fan\text{-}out)^2 \qquad (1)$$

They mention that fan-in and fan-out are weighted based on the belief that complexity is nonlinear. And the power of two takes into account the programmer interaction needed, as explained by Brooks [18]. They argue that high complexity value shows stress point in software as it has virtually multiple dependencies to other systems.

Function Point Metric

Albrecht et al. introduced a function point metric [19] in 1983, shown in Formula (2). It considers five parameters of the software: inputs, outputs, logic files, interface files and inquiries generated by the software. This aims to capture the size of the software from a business perspective.

$$
\begin{aligned}
&Function\ Points = \\
&4 * Number\ of\ inputs + \\
&5 * Number\ of\ outputs + \\
&4 * Number\ of\ inquiries + \\
&7 * Number\ of\ Interface\ files + \\
&10 * Number\ of\ logical\ files
\end{aligned} \quad (2)
$$

They claim that these metrics are useful as these can be measured at an early stage of the development process, and they are easier to understand by the user or customer than previous metrics. It is also explained that the function points metric measures the work needed and can then be utilised during project estimations.

Maintainability Index

There is now many different kind of metrics, but to understand the general maintainability of the system Oman and Hagemeister presented software maintainability as a hierarchy [20], where source code is just one part of the maintainability of the software. Their hierarchy is shown in Figure 5.

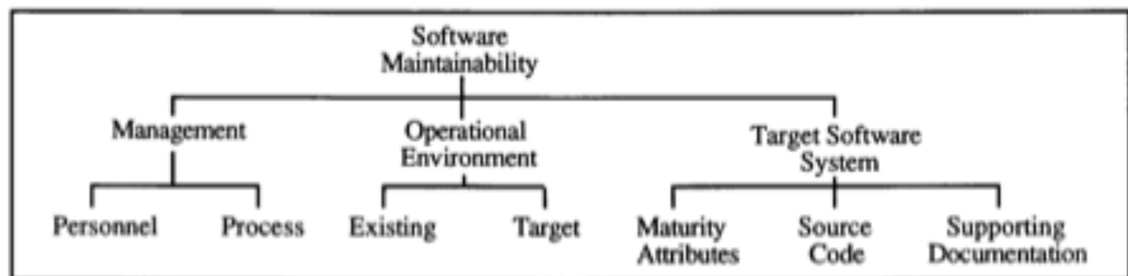

Figure 5. A Software Maintainability Hierarchy [20].

They also developed a derived metric, which uses different metrics to calculate a compound metric. It uses previously mentioned metrics such as cyclomatic complexity and Halstead complexity to give a single number for maintainability of the software, the maintainability index.

Khoshgoftaar et al. takes this even further and used principal component analysis and discriminant analysis to predict fault-prone software modules [21]. They did not try to

predict failures; instead, they classified software modules into two categories, is a module fault-prone or not. This then helps management to allocate resources on modules that cause most of the problems during testing and maintenance.

They later even applied artificial intelligence to predict failures in software [22]. Results of this study show that neural network models are better in prediction than regression models, and they recommend considering the neural networks for fault prediction. However, they also say that neural networks may not work in all situations or environments.

Using static code analysis to find metrics for software quality attributes is a well-known industry standard, and there exist products to do that. One such product is SonarQube [23], it measures several metrics, including cyclomatic complexity and Halstead complexity.

Product-related metrics can also be used to improve the functional testing of the software. Cyclomatic complexity can help software unit testing. Watson and McCabe state “Specifically, the minimum number of tests required to satisfy the structured testing criterion is exactly the cyclomatic complexity” [13, p. 33].

2.4 Process-Related Metrics

In the previous chapter, the mentioned metrics analyse the characteristics of the source code to measure quality attributes of the software. However, software is more than just executable code. Oman and Hagemeister also used values such as the number of comments in the source code, formatting of the code and even quality of supporting documentation [20]. Etzkorn et al. analysed identifier naming (e.g. variables and functions) as a predictor for faults [24]. Binstock has mentioned how the size of the codebase of unit tests compared to the size of project codebase can be used as a metric [25]. These are characteristics of the software that do not affect the execution of software. Nevertheless, they do affect the people developing software.

These people-related influences on software were already observed by Conway in 1968 [26]. Conway demonstrated that there is a very close relationship between the software architecture and the organisation designing it. They explained that if there are two or more software modules, then the designers of these modules need to discuss and agree

on the interfaces between them (Figure 6). Thus, organisational structure or communication structure will show up in the design of the system, as shown in Figure 7.

> Organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations.
>
> - Melvin E. Conway [26]

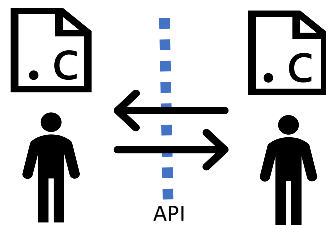

Figure 6. Designers of modules need to agree on how modules communicate with each other.

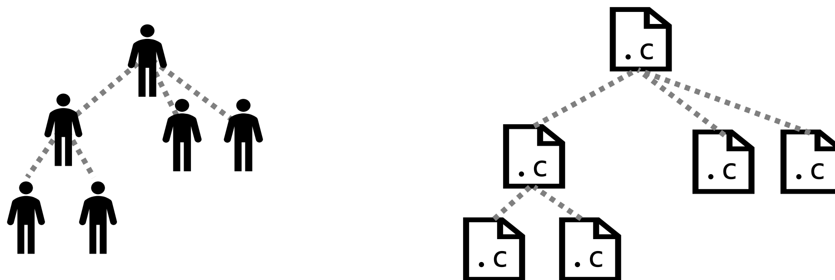

Figure 7. The organisation will naturally build a design which is a copy of its organisation structure

Conway also discussed why large systems disintegrate. They mention how designing is a different kind of work compared to manual labour. Conventional management solves problems by assigning more people to projects. However, on design work, more people cause an exponential amount of communication paths, which directly affects the architecture of the system.

Brooks in his book “The Mythical Man-month” [18] then extends Conway’s work. He contends that conceptual integrity is the most crucial consideration in the system design. He states that it is better to have one set of design ideas and rather omit features, than to

have many, maybe well designed, but uncoordinated ideas. Brooks also proposes designing software as “surgical teams”, where one person is doing the majority of design, and other team members are then doing supportive work.

Khoshgoftaar et al. had already noticed that changes in the software affect quality [22]. The research was done to understand better what the effects of software changes into quality are. They commented about the nonlinearity relation between changes and complexity metrics:

> Given the nonlinear nature of neural networks, this suggests there is some form of nonlinearity describing the relationship between software complexity metrics and gross change.
>
> - Khoshgoftaar et al. [22]

Gall et al. used version control data to detect which software components are coupled together [27]. They discussed the problems associated with product-related metrics that it only reveals syntactic dependencies between modules.

> Such measures do not reveal all dependencies (e.g. dynamic relations). In fact, some dependencies are not written down either in documentation or in the code. The software engineer just “knows” that to make a change of a certain type, he or she has to change a certain set of modules.
>
> - Gall et al. [27]

Their paper describes how they used software release history to detect software coupling. If two or more components are changed together, then they are logically coupled together. This can be then used to detect structural shortcomings in software and can be used to predict failures.

A similar study was done by Zimmerman et al. [28]. They used data mining to analyse change history. From changes in source code, they tried to predict the further changes required, find couplings not shown by static analysis and to prevent errors due to incomplete changes. The conclusion was that this method is better on stable software, but on new software, it is not very accurate, which is not surprising, as it would have to predict new functions.

Graves et al. found that numbers of lines of code or other code related metrics are not helpful predicting future faults once the amount of changes into the code is taken into account [8]. They built several models, which used change management data of software to predict new faults in software. Their best model, called the weighted time damp model used change-data but also weighted it based on the age of change.

They explained that in large and long-lived software systems, the process related metric is more useful than product-related metric. Especially the number of changes in the software. They noticed that complexity metrics to be not any more useful predictors than lines of code: “We found that nearly all of the complexity measures were virtually perfectly predictable from lines of code” – Graves et al. [8]. Their correlation matrix is shown in Table 2.

Table 2. Correlations of complexity metrics

	1	2	3	4	5	6	7	8	9	10	11	12
1 Lines Of Code	1	.97	.88	.88	.91	.99	.98	.92	.97	.85	.72	.35
2 McCabe V(G)1	.97	1	.88	.90	.88	.95	.95	.89	.93	.86	.76	.29
3 Functions	.88	.88	1	.82	.89	.85	.84	.91	.84	.76	.65	.29
4 Breaks	.88	.90	.82	1	.83	.86	.85	.85	.85	.78	.67	.27
5 Unique Operators	.91	.88	.89	.83	1	.89	.87	1.00	.94	.65	.47	.48
6 Total Operands	.99	.95	.85	.86	.89	1	1.00	.90	.98	.85	.72	.31
7 Program Volume	.98	.95	.84	.85	.87	1.00	1	.88	.97	.87	.74	.28
8 Expected Length	.92	.89	.91	.85	1.00	.90	.88	1	.94	.69	.53	.42
9 Variable Count	.97	.93	.84	.85	.94	.98	.97	.94	1	.77	.60	.38
10 MaxSpan	.85	.86	.76	.78	.65	.85	.87	.69	.77	1	.92	-.10
11 MeanSpan	.72	.76	.65	.67	.47	.72	.74	.53	.60	.92	1	-.25
12 Prog Level	.35	.29	.29	.27	.48	.31	.28	.42	.38	-.10	-.25	1

Bell et al. has also stated that the importance of changes in predictions are so high that even simple changed/not-changed variable can predict failures [29].

Schröter et al. found that the highest correlation with pre-release failures of software was the number of changes from the previous version among the measured metrics [30].

Nevertheless, they also commented that it is not surprising as those changes also contained the fix for the failure. On the other hand, the post-release failures had almost no correlation with process metrics.

In these studies, it has been found that changes in the codebase and other developer activities affect the quality of software. A possible explanation for this is provided by Nagappan et al. when they demonstrated empirically that organisational structure affects the quality of software [31]. They used organisational structure and compared it to changes in software to predict faults in software. They found that organisational structure is a better predictor for faults in software than any other metric; their results are listed in Table 3.

Table 3. Nagappan et al. model accuracy [31]

Model	Precision	Recall
Organisational Structure	86.2%	84.0%
Code Churn	78.6%	79.9%
Code Complexity	79.3%	66.0%
Dependencies	74.4%	69.9%
Code Coverage	83.8%	54.4%
Pre-Release Bugs	73.8%	62.9%

Conway had already 40 years earlier explained how organizational structure affects the architecture of the system [26]. Nagappan et al. now demonstrated that if there is a mismatch between the organisational structure and the software structure it affects the quality of software.

Several studies then examined how other social aspects affect the quality of the software. Meneely et al. performed an empirical analysis on the Linux code base and found a correlation between the number of developers and the quality of code [32]. Bird et al. examined the relationship between ownership and software quality [33]. They found that high levels of ownership are correlated with fewer defects. Madeyski and Jureczko made an empirical study about the process metrics, to find out which metrics improve defect prediction metrics [34].

Tornhill discusses in his book “Software Design X-Rays” [35] heavily about process metrics. They explain why ownership of code is necessary and how a lack of ownership causes diffusion of responsibility. Diffusion of responsibility is a phenomenon where a person is less likely to take responsibility when others are present. More developers there are, the less individual developer takes responsibility. Tornhill is also the author of CodeScene [36] application which measures some of the process metrics.

2.5 Faults and Failures

Faults and failures are mentioned in previously mentioned studies [8] [21] [22] [29] as something that is being predicted by quality metrics. However, these studies do not clearly define what is meant by their usage of faults or failures. This confusion is also mentioned by Hatton [37], and he defines fault as some inconsistency in code that may cause failure.

Basili et al. studied the cause of faults, and they found that half of the faults are caused by requirements being incorrect or misinterpreted, or functional specification being incorrect or misinterpreted [38].

All companies with a software development process need to track their failures. These failures are reported by a user of the software and may contain failures that are caused by something else than pure software fault either directly or indirectly, although for the user it might manifest the same way. For example, error in documentation may be reported as a software fault because the software does not match the documentation. Alternatively, like the previously mentioned misinterpreted specification causes the software not to match the expectations.

Schröter et al. described how version control data can be correlated with the bug database [30]. The number of bugs in this database can then be used as a target variable when predicting faults. However, this requires strict software development processes, where each developer would record the corresponding fault into the change.

3 Method and Material

This chapter explains the design process used to create the solution. The solution is a computer program capable of analysing software code and giving results for a set of quality metrics. Quality metrics being measured are listed and explained in further sections. Furthermore, the software environment used in the company and mandated restrictions are described. Finally, the evaluation method of the solution is described.

3.1 Design Process

The design process of this thesis follows the design science research process model [39]. The design process is presented in Figure 8.

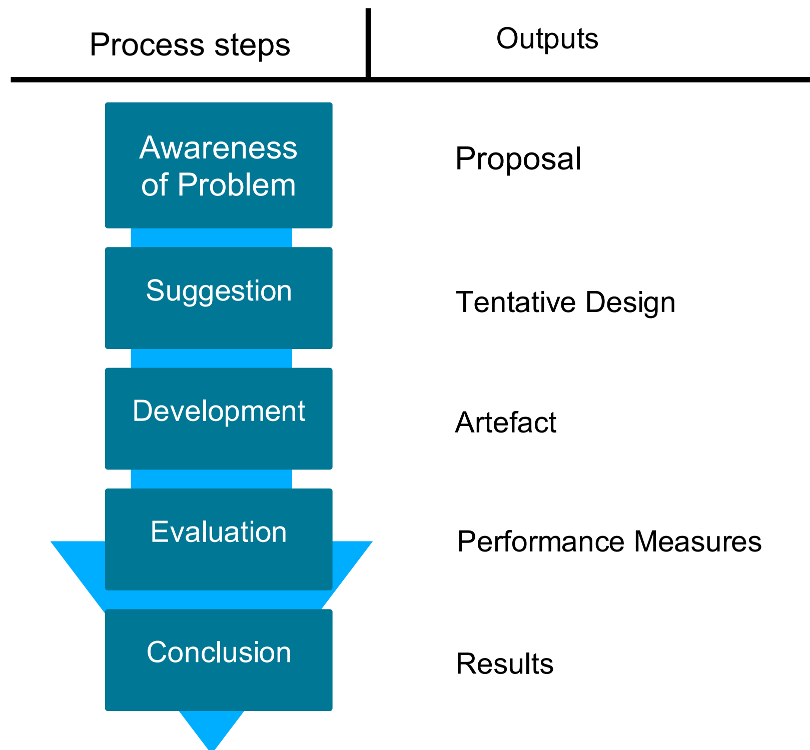

Figure 8. The Design Process.

Awareness of the problem came from interviewing of people at Silicon Labs, these were Director of software development, design managers and lead software architect. During these interviews, it came clear what is the problem, and it became a proposal for this thesis.

To tackle this problem a suggestion step was started, which created a tentative design. This tentative design was demonstrated, and it was accepted that this is a promising way to solve the problem.

The tentative design was further specified during the development phase. During this phase, a proof of concept was built. This created the artefact.

Evaluation of the artefact was done during the evaluation phase. The evaluation was done by observing the actual metrics of software development and comparing them against actual failures found in the software. The conclusion of the thesis discusses the results and further development possibilities.

3.2 Quality Metrics

The initial set of quality metrics are intentionally kept small. Three top quality metrics from Table 3 are selected to be measured; these three metrics are quality metrics which are easy to measure but still provide useful information. The system is designed to be extensible; thus, more quality metrics can be added later. Following quality metrics are selected for measurement for this proof of concept:

3.2.1 Code Ownership

Code ownership is the degree of how much a single developer contributes to the software. There are different ways to define it. Bird et al. calls it a Proportion of Ownership and defines it as a number of commits done by a developer compared to the total number of commits [33], as shown in Formula (3).

$$Proportion\ of\ Ownership = \frac{nc(a_i)}{NC} \tag{3}$$

$nc(a_i)$ is the number of commits from a developer a_i

NC is the total number of commits

This defines ownership per each developer, to get a single value, these values need to be aggregated. D’Ambros et al. introduced a fractal value, which summarizes the contribution of each developer compared to all contributions [40]. This is shown in Formula (4).

$$Fractal\ Value = 1 - \sum_{a_i \in A} \left(\frac{nc(a_i)}{NC} \right)^2 \quad (4)$$

A is the set of developers

$nc(a_i)$ is the number of commits from the developer a_i

NC is the total number of commits

This Fractal value is zero if a single developer has written all software, and it gets closer to one more there are developers working on software. This value tells how diffused software development is.

3.2.2 Code Churn

Code Churn is the number of changes in codebase over time. The term was introduced by Nagappan et al. in their article “Use of Relative Code Churn Measures to Predict System Defect Density” [41], but similar metric was already used previously by Graves et al. in their article “Predicting fault incidence using software change history” [8]. Code Churn is calculated as the relation of added and changed files compared to total files, as shown in Formula (5).

$$Code\ Churn = \frac{LOC_{churned}}{LOC_{total}} \quad (5)$$

$$LOC_{churned} = LOC_{inserted} + LOC_{removed}$$

LOC_{total} is the total number of lines of code

LOC_{total} is measured from full source listing. However, if Code Churn is measured from the changes in software, then it is questionable that should Code Churn also include code that was made before the change. Instead, Code Churn is normalised over the change, as shown in Formula (6).

$$Code\ Churn = \frac{LOC_{churned} - \mu}{\sigma} \quad (6)$$

$$LOC_{churned} = LOC_{inserted} + LOC_{removed}$$

μ is the mean of the $LOC_{churned}$

σ is the standard deviation of the $LOC_{churned}$

3.2.3 Code Complexity

Code Complexity can be understood in different ways, as discussed in section 2.3. However, in the context of this work, it means precisely the comprehension complexity or how hard it is the developer to understand the code. Oram et al. argue that syntactic complexity metrics do not provide enough information to capture the effort needed to comprehend the code or at least not more than simple lines of code can provide [14]. The same conclusion was done by Graves et al. [8]

This reasoning allows comprehension complexity to be measured simply from lines of code. As this is measured over a change, only those lines that are added to the software are counted. This is then also normalised as shown in Formula (7).

$$Code\ Complexity = \frac{(LOC_{added}) - \mu}{\sigma} \quad (7)$$

$$LOC_{added} = LOC_{inserted} - LOC_{removed}$$

μ is the mean of the (LOC_{added})

σ is the standard deviation of the (LOC_{added})

3.3 Aggregated Metric

Aggregated Metric or derived metric is a single metric that is calculated from other metrics. The purpose is to map other metrics to an easily understandable value, that can tell some truth from the software. This has been done multiple times before, as explained in chapter 2.3.

On some previous works, these metrics were handled as they were correlated with each other. Halstead used simply a sum of metrics [16] and Oman et al. used weighted arithmetic mean [20]. Khoshgoftaar et al. mentioned non-correlation between metrics [21], but these were then handled as distinct metrics and not aggregated.

In this work, it was found out that these metrics do not correlate with each other. This is discussed in section 5.1. This means that calculating the aggregated metric is not a simple linear problem.

If each metric alone indicates a possible lack of quality from a certain point of view, the aggregated metric could then be defined to be a value where all metrics indicate something at the same time, as shown in Formula (8). This metric can then be used to sort data.

$$Aggregated\ Metric = Fractal\ Value\ AND\ Code\ Churn\ AND\ Code\ Complexity \quad (8)$$

Nonlinearity suggests that machine learning algorithms could be applied to solve this. This was done previously by Madeyski et al. They built a model using machine learning and used it to predict faults in software [42]. Machine learning algorithms require a feedback signal [43], and this is problematic as there is no such information readily available in this case. Another option is to think of this problem as solvable by probability theory, and there exist tools to solve nonlinear problems in probability theory [44].

A more straightforward solution is found in another branch of mathematics called fuzzy mathematics, which includes fuzzy logic. According to Ross, fuzzy systems have a high potential to be applied to a complex system [45]. The fuzzy model has been used before by Singh et al. [46] to build a model to predict software maintainability. However, in this prototype there is an elementary rule as was shown in Formula (8), thus fuzzy logic, in this case, can be applied directly.

In fuzzy logic, variables vary in range from 0 to 1. 0 meaning not probable and closer to one variable goes more probably true it is. Fractal Value metric fits this requisite, but Code Churn and Code Complexity metric need to be clamped to this range. Instead of using a simple clipping, a sigmoid function will be applied to get a smoother curve. The hyperbolic tangent function has suitable properties for smoothing values. It translates any input value into range -1 to 1, as can be seen in Figure 9.

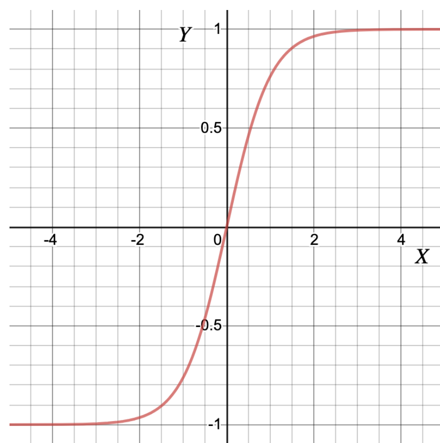

Figure 9. A plot of hyperbolic tangent

AND operation in fuzzy logic is defined to be minimum of two values. Inserting this into Formula (8), it becomes Formula (9) and then it is easy to calculate.

$$Aggregated\ Metric = min(Fractal\ Value, Code\ Churn, Code\ Complexity) \qquad (9)$$

3.4 Software Components

As mentioned before, Silicon Labs embedded software consists of components. These software components contain source code, libraries and other files necessary for using the functionality of the component.

These components can depend on other components, which means that including one component into the software project may also cause the inclusion of other components. This is illustrated in Figure 10. Component A depends on component B and C, and component C also depends on component D. Thus, including component A into the project, will also include components B, C and D.

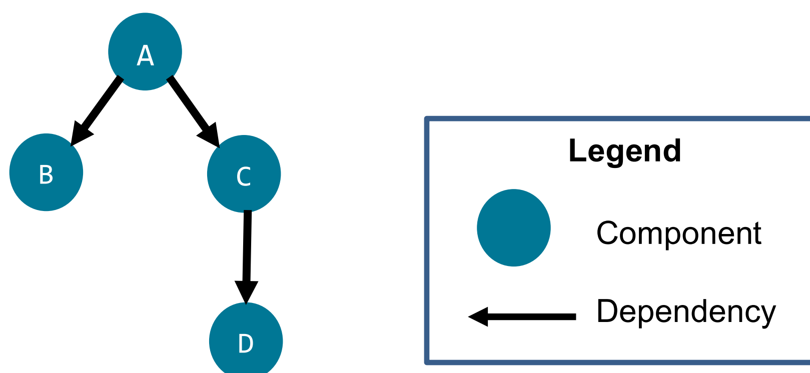

Figure 10. Example of component dependencies.

Each component is defined by its YAML-file. These files describe the requirements components have toward other components. In addition, software repository has a single metadata file, which lists the locations of all these component metadata files in the repository. Using this repository metadata file, it is then possible to iterate through all these component metadata files and construct the entire hierarchy of software components in the software repository.

Silicon Labs software development organisation is defined in such a way that there is a clear division of responsibilities and work between different teams. Each team is responsible for its own set of components. Components are generally never developed by multiple teams.

Silicon Labs is a multinational company, and development teams are located around the globe. They have different cultural backgrounds, and the projects they are working are different. Thus, the code they produce is not necessarily comparable with each other. So even if there is a dependency between components, this dependency is not taken into account in this work. This work concentrates measuring individual teams, thus metrics from components are analysed individually, and they should not be merged or analysed against each other.

It is vital to not mix-up software component and software module terms between each other. These terms have had different meaning depending on who defines them. Also, their meaning has historically been changed over the years [47, pp. 29-31]. On this thesis software module is defined as a unit of implementation with the specified interface. This

follows the definition by Parnas [48], where the module definition is based on information hiding.

A component is then a logical structure which provides functionality to end product by providing a composition of software modules.

3.5 Software Repository and Restrictions

All software source code and metafiles are stored in a version control system (VCS). VCS stores current and previous states of software source code. Additionally, it tracks changes on all files and has records of the date and author of changes. VCS used at Silicon Labs and in this work is the Git version control system.

Every component is available in a single Git repository called super. In this Git repository, each change is recorded as a commit. This commit also has information about the previous commit, thus commits creates a chain of changes. Every commit in this chain uniquely represents the full state of the repository at that point of time. Commit also records metadata such as author and date of the change and optional developer written description of the purpose of a commit, see Figure 11. Each commit can then be used to recreate the state of the software at that point in time. To track changes over time requires tracking changes from all commits during that period.

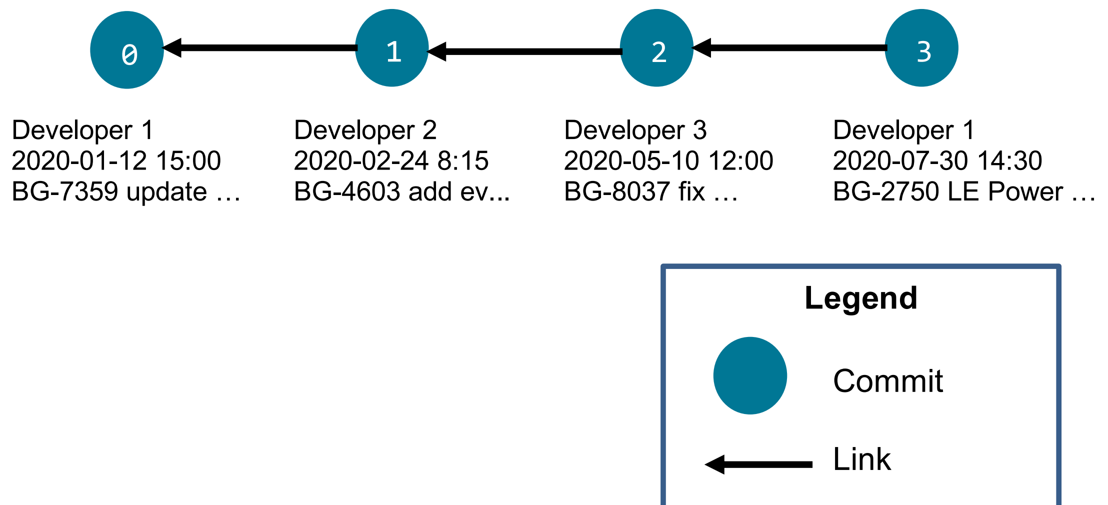

Figure 11. Each commit links to the previous commit and contains delta from it. They also contain metadata about the commit itself.

Metropolia
University of Applied Sciences

Issues related to software development are recorded in Jira software. Each issue is categorised to be a bug, feature, task, or some other category. For this work, we are only interested in bugs in the software project. There is no direct coupling with the Git version control system and Jira issue tracking software. Instead, they are coupled by adding Jira issue tag into Git commit description, which then identifies the Jira issue this commit is related to. Example of how these issue numbers are recorded can be seen in Figure 10.

Although all embedded software is written in plain C, all the internal development tools must be written in Python programming language. This is mandated by a companywide coding standard.

3.6 Evaluation Method

As is with previous studies, the results of metrics are validated by comparing them against recorded faults of the software.

This is achieved by analysing metrics for each file in a component against faults detected in a file. Git commits naturally record which files are changed. This enables measuring and calculating metrics per file trivial. Git commits also record a description of the commit. If that description is appropriately filled, then it has a recording of which Jira issue it references, this can then be used to measure how many bugs a file has.

4 Solution Development

The solution is developed as a proof of concept to understand better the problem statement and what is needed to measure the quality attributes.

The programming language used is Python 3.0.

4.1 Requirements

Requirements for the solution is divided roughly between two categories, the first category is requirements needed to implement the needed functionality and the second category is the non-functionalities to improve the quality of software.

The needed functionality is to measure a set of quality metrics from changes in software components over time. There are three quality attributes selected to be measured: Code Ownership, Code Churn, and Code Complexity. The source code to be measured is stored in a software repository. Results must then be made available into a separate database. Records in result database must contain measured quality metric, what was used to compute it and when it was computed.

Apart from the functionalities mentioned in the previous section is the ability to extend the solution. As this solution is a proof of concept, it is expected to be modified later and continued to be worked on. Related to the easiness to modify and extend is the ability to quickly start to work on the solution and apply it into use. To put this into a formal context and how these are defined in ISO 25010 [7]: The explicitly stated non-functionalities are the maintainability characteristic and its sub characteristics modularity and modifiability. Another one is portability characteristic’s sub-characteristic installability.

4.2 Architecture

The solution is designed to be modular to enable easy modifiability and extensibility. It also leverages existing commonly available solutions and software to make it easy to install and use. In the next sections, the architecture of a solution is explained in detail. Architecture description follows the layout described by Clements et al. in their book “Documenting Software Architectures” [47].

First, a high-level overview of the solution is provided. This puts software in a general context. Next, each module of a system is explained, including the measurement application. Measurement application is explained in detail and how it is structured. Following

that each component of the measurement application is explained, and their design decisions justified.

4.2.1 High-level Overview

The solution consists of several software modules and databases. Modules, databases, dataflow and how different actors are related to the solution is illustrated in Figure 12. The aim is to have simple dataflow through the system and for each software module to have a clear purpose.

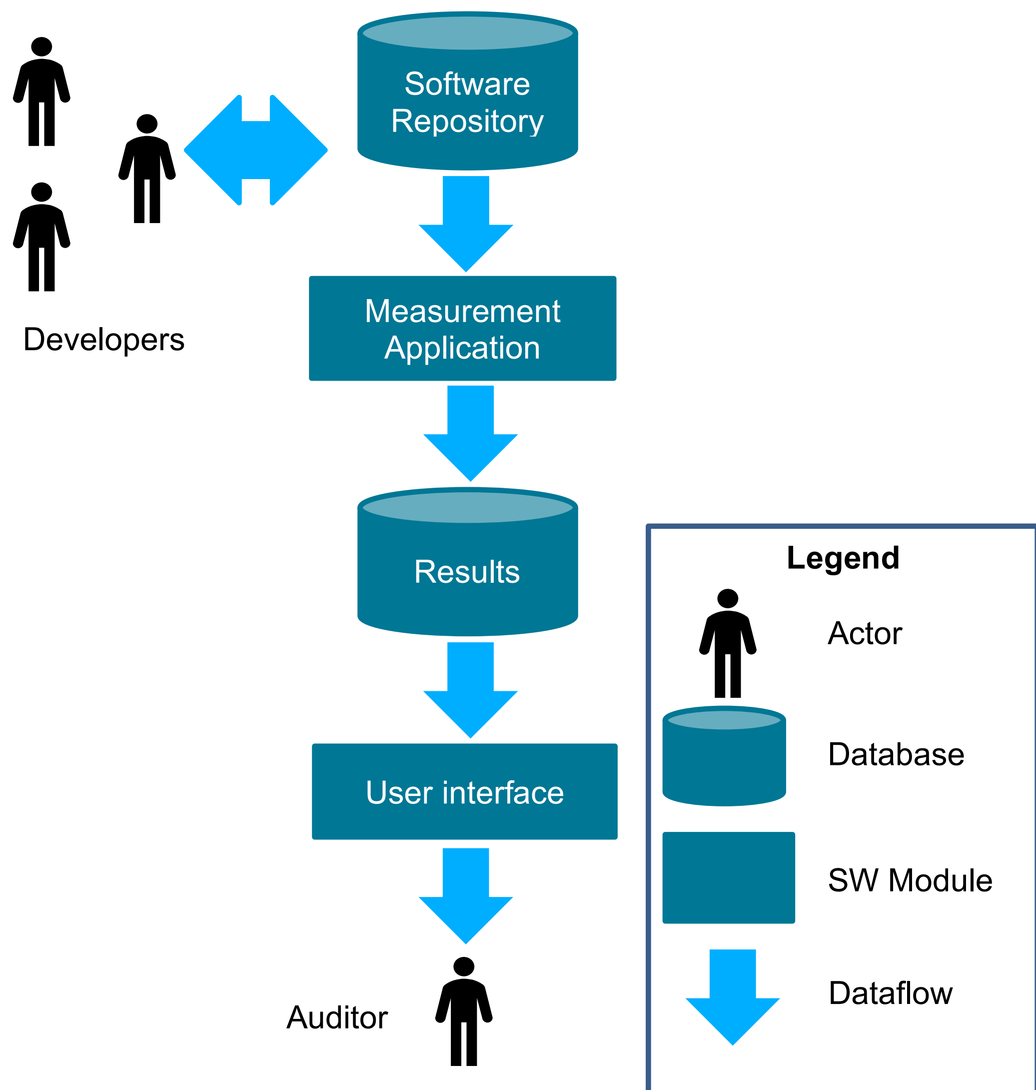

Figure 12. Primary view of the solution

Developers are actors who access the software repository while they develop software. Software repository stores not only the software code but also metadata as explained

before in chapter 3.5. Measurement Application SW module then access this software repository and retrieves metadata about the software and calculates software metrics based on the metadata. Software metric results are then stored on a different database.

An auditor is an actor who is interested in software quality metrics. They use some user interface to access the results database and get quality metrics about the software under construction.

Software repository used with this solution is the Git version control system.

4.2.2 Measurement Application Design

Measurement application is designed to be modular, meaning that there is no tight coupling between different software modules. This is done to enable better modifiability. The structure of the application is illustrated in Figure 13.

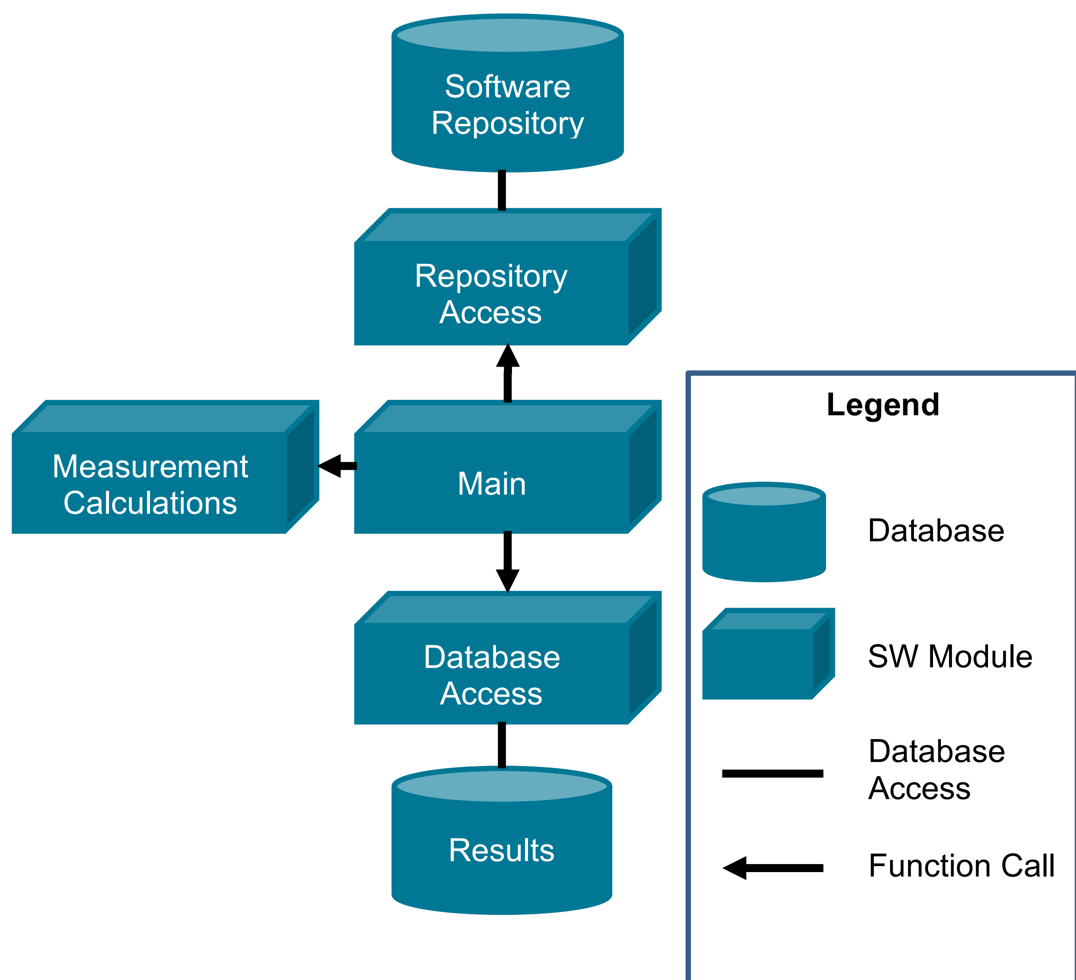

Figure 13. Module view of measurement application

Application external interface is only through database accesses, one database is for input and another one for output.

The application consists of four software modules, each having a specific task. These modules are analysed further below.

Repository Access

The repository access module is responsible for accessing the software repository. It first builds a structural view of the whole system. It is done by reading the file containing a list of all components in the system. These component metafiles are then read, which in turn defines all the modules and files for the component. With this information, the structural view of the system can be built, the UML diagram for this is shown in Figure 14.

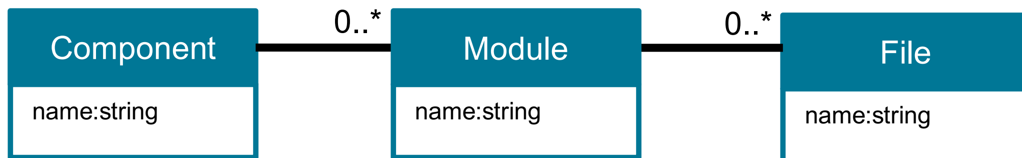

Figure 14. The data model for system structure

Repository access module uses external Git program to analyse files. Git is a command-line application and also provides results of the analysis in text form. The results are then parsed to get them into a valid form for further processing. Following command is used to execute external Git command:

```
Git log --numstat --format=%ae,%at --since=2019-01-01
 --until=2019-12-31 example.c
```

This command asks Git to provide a commit log for file `example.c`. The `-numstat` defines Git to output added and removed files in a format which is easier to parse. `--format=%ae,%at` configures Git to also outputs author email and timestamp of a change. Finally `--since` and `--until` defines the time period where the commits are needed.

Following is an example of what Git would output:

```
38 10 example.c
```

```
A@silabs.com,1574332766

25 9 example.c
B@silabs.com,1574252587
```

In this example, there are two commits from authors A and B. Author B has added 25 lines and removed 9. Author A has added 38 lines and removed 10. This kind of log is straightforward to parse.

This Git command is run on every source file in every module in every component. From these logs for each file, the Repository Access module extends the data model by adding information about authors into each file. Updated UML diagram is shown in Figure 15.

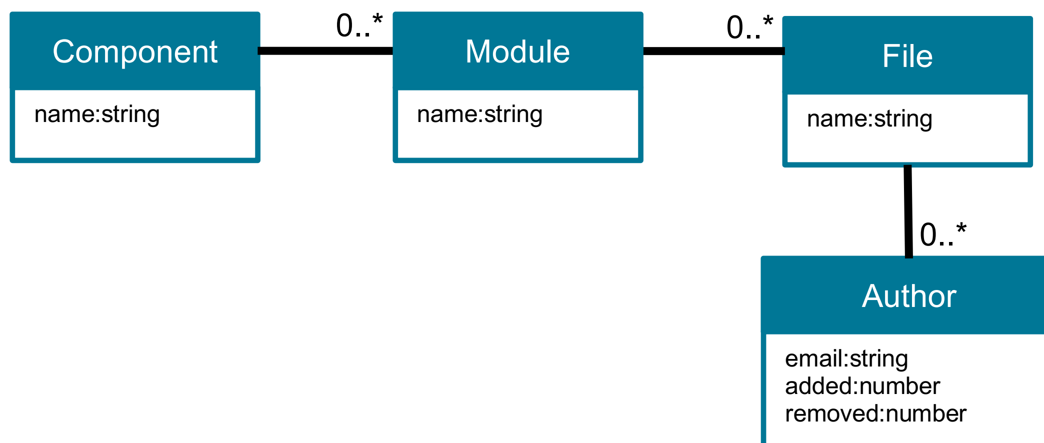

Figure 15. The data model for files in the repository. Attributes are not a full list, but only given as an example.

Essentially this data structure is a tree-like structure, which simplifies data processing. Tree-like structure enables efficient parallel execution on calculations.

During this phase the data is cleansed. Any data that has proportionally massive changes in any of the values are removed. These are usually either auto-generated code or some other anomaly in the data.

Metropolia
University of Applied Sciences

Measurement Calculations

Measurement calculations are done for every file and then propagated upwards for every module and every component. Measurements for each quality attribute are computed as explained previously in paragraph 3.2.

This then further extends the original data model with measurement attributes for components, modules and files.

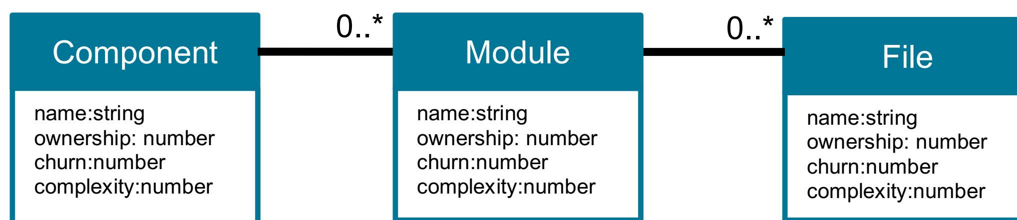

Figure 16. Data model after calculations

Data model after calculations is shown in Figure 16. The model does not contain any data about authors anymore.

Database Access

Database access then pushes this measurement data into the database. In this proof of concept, a CSV (Comma-separated values) file is used as a database. CSV file is a simple format where each line is a single entry. Data model after calculations is in normalized form and data needs to be denormalised for CSV.

Denormalisation in this case means removing the hierarchy from the data model. Each entry for each file it also records the module and component, which the file is part of. Denormalised data model is shown in Table 4.

Table 4. Denormalised data

Component	Module	File	Ownership	Churn	Complexity
…	…	…	…	…	…
…	…	…	…	…	…

This CSV file can then be easily imported into any application and to be visualized.

Metropolia
University of Applied Sciences

4.3 The Reasoning for the Design Decisions

Reasons for each design decision is based on the requirements. Solution's architecture is modular, and each module can be individually replaced without affecting the rest of the system.

Repository access module uses external Git application instead of accessing Git database using Python library. The reason is to have it easier to be installed without dependency on other Python libraries.

Measurement calculations module calculates measurement for only three attributes, but this can be easily extended to cover more metrics as there are no dependencies with other modules.

Similarly, the database access module is very primitive implementation, but there is no reason why it could not be changed access to some other database. The current implementation is the bare minimum for this proof of concept.

5 Results and Analysis

Results were achieved by measuring files from three different components. These components were selected based on the following criteria: Large and mature enough that there is enough data to process. Enough of developers working on a component to provide large enough sample size. Distinct enough that there is no dependency between them. Three components were selected: B, Z and W. These components are different communication protocol stacks.

The metrics for files in these components were then measured in three different periods. 2018, 2019 and full history. These then give nine sets of data.

5.1 Quality Metric Results

Pearson correlation coefficients for components B, Z and W were measured during several periods; results are listed in Table 5. From these results, it can be seen that there is no linear correlation between metrics, except in some cases between Code Churn and Code Complexity. Reason for this can be seen from Formula (6) and Formula (7). They are calculated from the same base value, differing only by how much code is deleted. Thus, a high correlation between them means that substantially more code is being added than deleted.

Table 5. Pearson correlation coefficients for metrics from components B, Z and W

Component	Period	Diffusion - Complexity	Diffusion – Churn	Churn – Complexity
B	**2018**	0.09	0.04	0.7
	2019	0.15	0.11	0.54
	Full	0.04	0.03	0.53
Z	**2018**	0.33	0.10	0.47
	2019	0.10	-0.01	0.72
	Full	0.16	0.20	0.82
W	**2018**	0.22	0.04	0.46
	2019	0.16	0.17	0.63
	Full	0.08	0.11	0.79

Pearson rank correlations for metrics were calculated similarly; results are shown in Table 6. These show similar kind of results as with Pearson correlation. Thus, there is no linear relation, nor they are monotonously related.

Table 6. Spearman rank correlations for metrics from components B, Z and W

Component	Period	Diffusion - Complexity	Diffusion – Churn	Churn – Complexity
B	**2018**	0.02	-0.13	0.60
	2019	0.05	-0.02	0.50
	Full	-0.01	-0.07	0.62
Z	**2018**	0.24	-0.35	-0.09
	2019	-0.09	-0.28	0.45
	Full	0.00	-0.04	0.78
W	**2018**	0.19	-0.35	0.30
	2019	0.18	-0.10	0.32
	Full	0.04	0.10	0.79

Another way to look at these metrics is to observe the scatter plots. Component B scatter plots for the full period is shown in Figure 17. It can be observed that there is no meaningful relation between diffusion and churn. But complexity and churn show some relation as mentioned before.

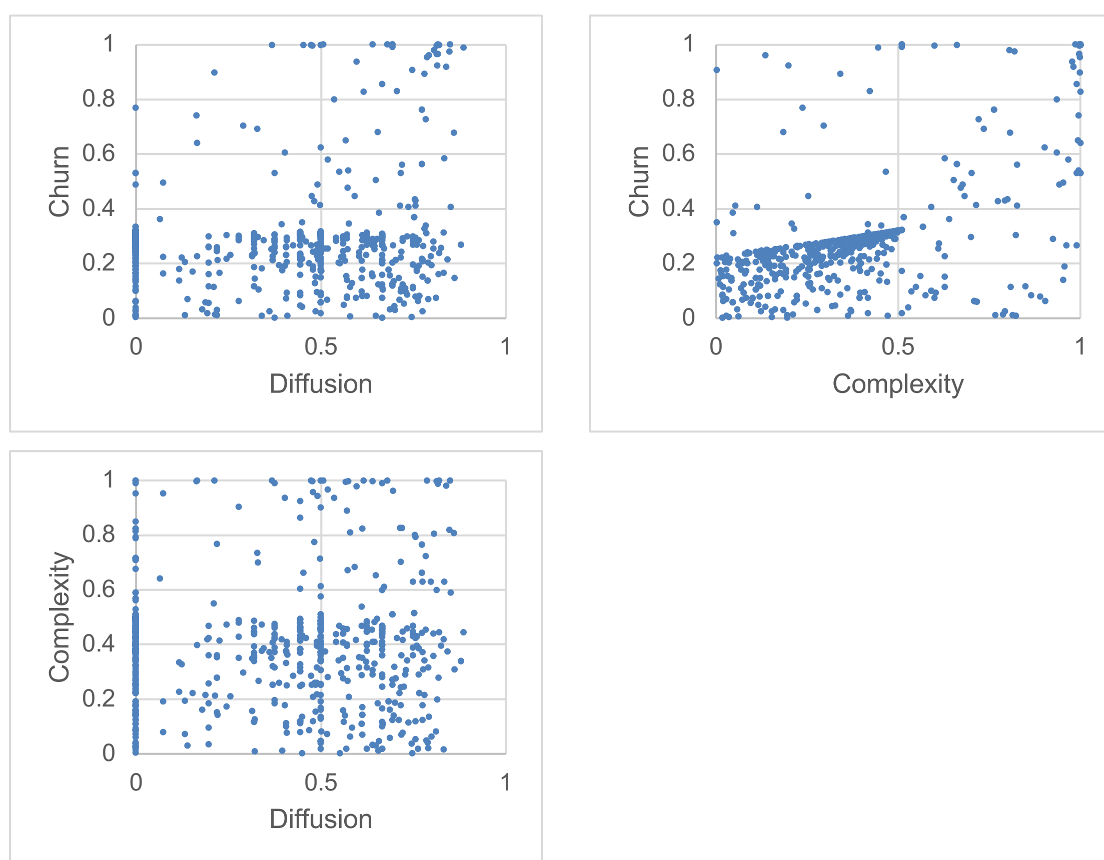

Figure 17. Scatter plots for metrics from Component B over full development time.

A similar observation can be seen from scatter plots for component Z, as seen in Figure 18. There is again a clear correlation between complexity and churn. Another observation for component Z is that the codebase is heavily diffused, this tells that lot of developers has been developing different parts of the component, and there is no clear ownership.

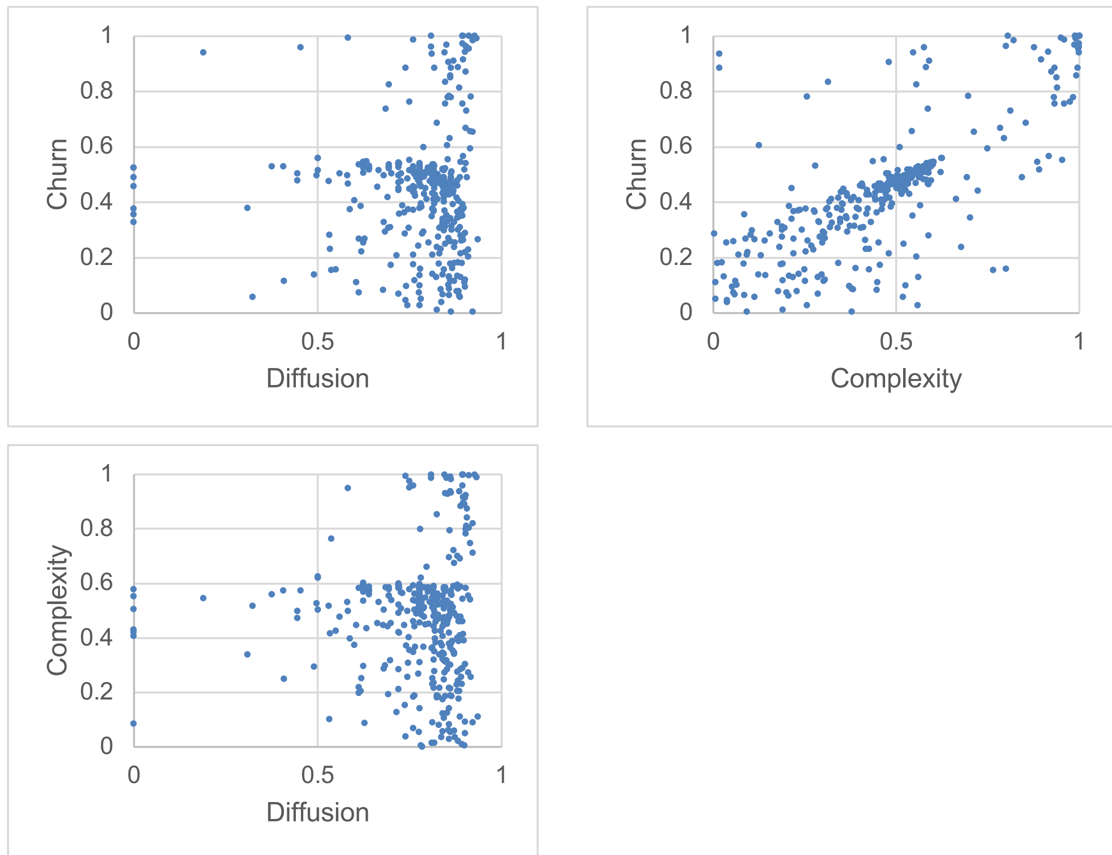

Figure 18. Scatter plots for metrics from Component Z over full development time

Scatter plots for component W is shown in Figure 19. These results are closer to component B than Z. Component W has several files where diffusion is 0, meaning that the file is developed by a single person. This kind of behaviour is also present in component B. Again, it is visible how churn and complexity are correlated.

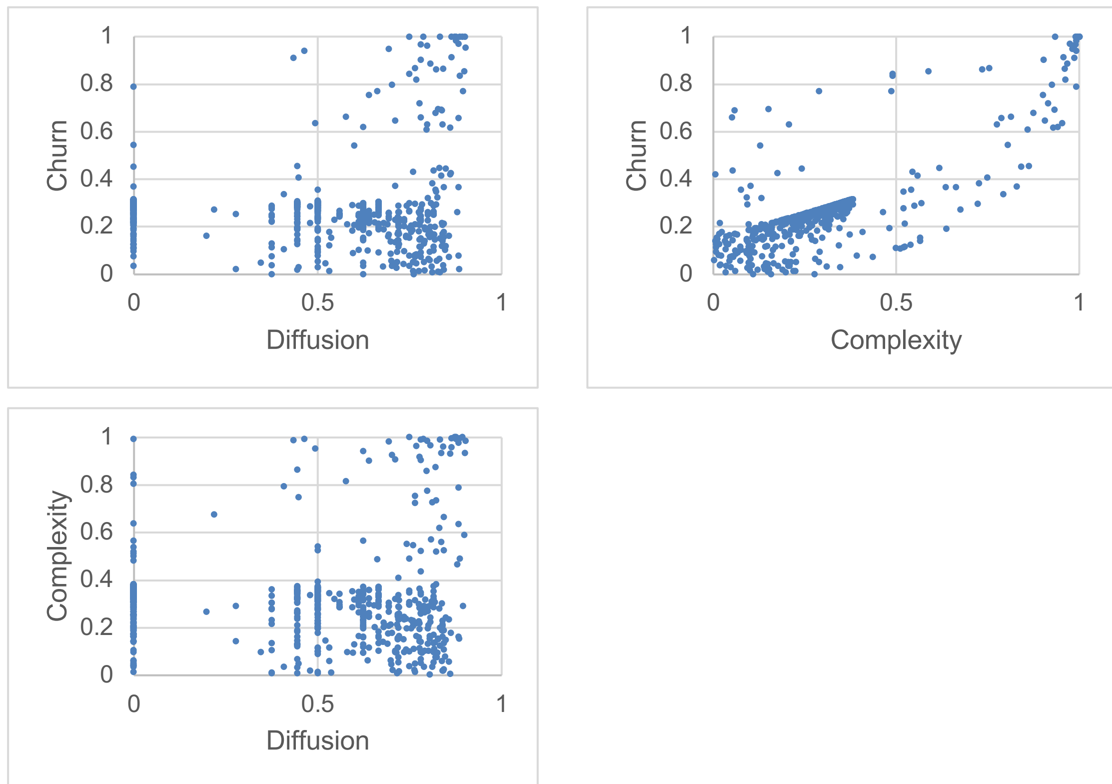

Figure 19. Scatter plots for metrics from Component W over full development time

These results, especially scatter plots show a difference between these components. Even when they are all protocol stacks, with over ten years of development and multiple developers they show different results. They are developed by different teams, which have different cultural backgrounds. This may be one of the reasons that there is a difference between the components.

5.2 Detail Analysis of Component B

Component B has 728 files, developed by 23 people over the past ten years. Metrics for all files over the full history of component B is shown in Figure 20, the files on the x-axis are sorted based on aggregate value. Closer to one the aggregate value goes, the more probable it is that there is something that needs closer inspection.

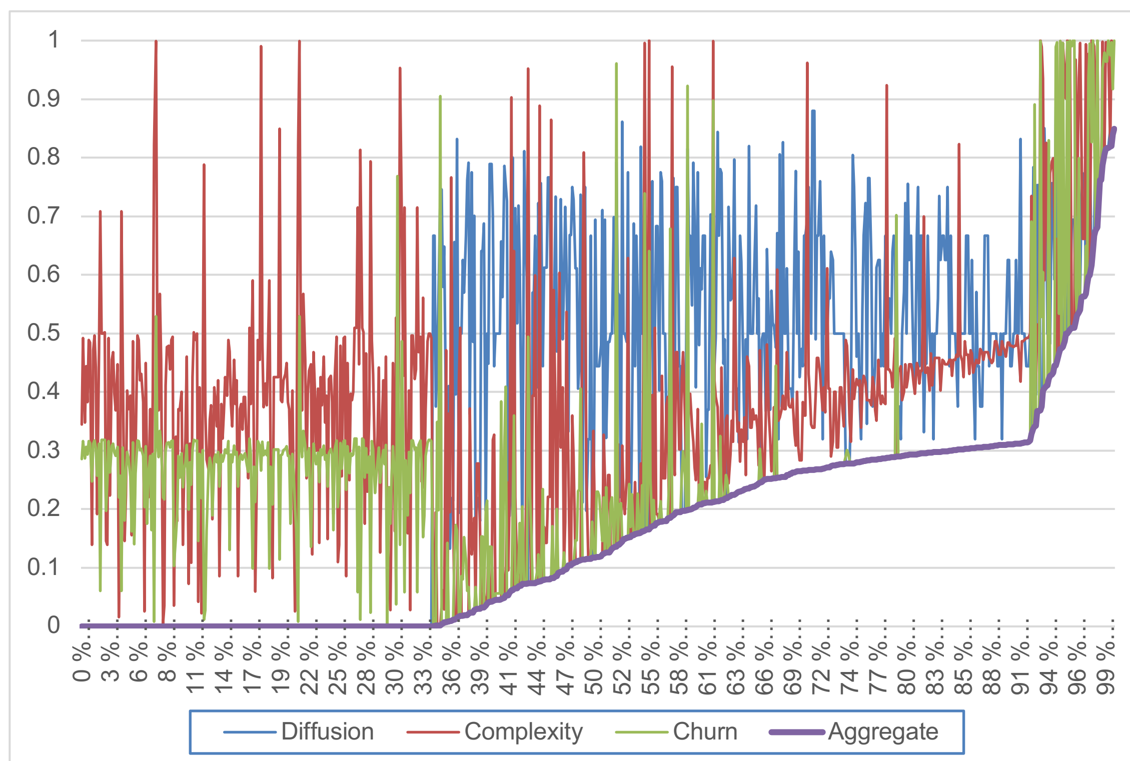

Figure 20. Metrics for each file in component B during the past ten years.

Ten files with highest aggregate values are listed in Table 7. The table also lists the category where each file belongs to. API category is the application programming interface, containing functions providing access to the component. Core category is the category containing the main functionality of the component. And Utility category is supporting functions, like memory management.

Table 7. Files with the highest aggregation values in Component B

File	**Category**	**Developers**	**Diffu-sion**	**Complexity**	**Churn**	**Aggregate**
1	API	17	0.85	0.99	0.99	0.85
2	Core	11	0.83	0.98	0.92	0.83
3	Core	11	0.82	0.99	0.99	0.82
4	Core	23	0.85	0.82	0.97	0.82
5	Core	20	0.82	0.98	0.99	0.82
6	Utility	16	0.82	0.99	0.96	0.82
7	API	15	0.81	0.81	0.98	0.81
8	Core	12	0.79	0.99	0.95	0.79
9	Utility	13	0.77	0.76	0.76	0.76
10	Core	10	0.78	0.72	0.73	0.73

These files are considered to be the most critical files for the quality of the software. They have the highest diffusion as a lot of developers have been working on them. They have the highest churn, as a lot of lines of code has been added and removed. Moreover, they have the highest complexity as a lot of code has been added to them. All indicators point to the fact they are considered problematic, making them have the highest aggregate value.

After discussing these results with design managers for component B, there is some reasoning why these files are not as bad as they look like. One metric which is not measured is code coverage. Core files have been extensively unit tested and have good code coverage. But, files in the API category have not been extensively unit tested. Instead, they are tested during part of functionality testing by the quality assurance team. Code coverage should then also be possible to be obtained for these files.

Utility files provide generic functionality. Due to this nature, they are depended on by multiple other files. This also causes a lot of changes on these files. There is also an evident lack of ownership for these files, and because of this, they are not heavily unit tested.

6 Evaluation and Conclusions

The quality metrics are used to predict the lack of quality in software. The lack of quality further manifests itself as a shortcoming of expectations in the functionality of the software. These shortcomings are reported as bugs and recorded in the issue tracking software.

To evaluate the solution, the results from quality metrics can be compared with the found bugs. This is achieved by using issue tracking software and version control software to calculate how many bugs each source file in a component has. As each commit has a recording of which issue it is related, and each issue can be classified to be a bug. Then it follows that each commit can also be classified to be a bug related commit, or more precisely a bug fix. If a commit is not a bug fix, then it is a new feature introduced into the software. Example of this shown in Table 8, these are the same files as in Table 7.

Table 8. Files with bugs and features added

File	Bugs	Features	Diffusion	Complexity	Churn	Aggregate
1	4	21	0.85	0.99	0.99	0.85
2	13	3	0.83	0.98	0.92	0.83
3	16	1	0.82	0.99	0.99	0.82
4	20	25	0.85	0.82	0.97	0.82
5	19	23	0.82	0.98	0.99	0.82
6	3	1	0.82	0.99	0.96	0.82
7	20	25	0.81	0.81	0.98	0.81
8	6	10	0.79	0.99	0.95	0.79
9	0	1	0.77	0.76	0.76	0.76
10	10	15	0.78	0.72	0.73	0.73

This information of bugs for each file can now be correlated with measured metrics. Two methods are used. Pearson correlation coefficient measures the linear relationship between variables, and Spearman's rank correlation coefficient assesses the monotonic relationship between variables. These relationships are shown in Table 9.

Table 9. Pearson and Spearman correlations for bugs

	Features	Diffusion	Complexity	Churn	Aggregate
Pearson	0.77	0.29	0.29	0.39	0.35
Spearman	0.51	0.40	0.013	0.07	0.25

From these results, it can be seen that the highest correlation with bugs is the new features; this is true for both linear and monotonic correlation. This is quite obvious as new features create new source code that then may contain bugs.

Churn is the best metric for linear correlation with bugs. This has already been found out previously as was explained in section 2.4, but in this case, it is not much better than complexity or diffusion. For monotonic correlation, the best predictor is diffusion. Interestingly complexity or churn is not significantly related at all.

The aggregate metric is the second-best metric in both correlation cases. This is understandable as it also takes into account other metrics, which then averages results.

6.1 Threats to Validity

There are several threats to validity. This is a proof of concept work, and these threats need to be taken into account when improving on this work.

It is understood that different teams develop differently, and this is already taken into account, as was explained in section 3.4. However, it may still be that development inside one team is still subdivided into distinct sub-teams with different cultural and historical backgrounds. And this may affect the way they develop code.

Source code used in this case is not necessarily only code used for the end application. Some of it is source code used for applications during development, such as test applications or software development tools. Any work done on those does not reflect directly to the outside of the development team. This means none of it will show up in issue tracking software. This would then skew the validation of the data. Even if failures on those applications do not directly affect the quality of the software provided for customers, it would be good to improve the quality of those applications too.

A related issue with source code is that some of the code may be auto-generated, and this will then heavily skew the results. It is bad practice to add auto-generated code into a version control system, but in practice, it happens. An effort was made to remove some of the prominent cases, for example, where a single commit contained thousands of removed and added lines.

Commits into Git repository depend on the way developer works. Some people like to make all changes into one commit, and some people make multiple small commits related to a single issue. Some people document meticulously what the commit contains, and some then put just one-line comment. And even if the company's coding guideline requires that each commit must have the issue tagged, only about 15% of all commits had that added. This is one of the primary reasons for the errors when doing a correlation with metrics with the actual bugs as not all the commit recorded if they were bugfixes or anything else at all.

This proof of concept does not take time information into account. Newer changes have the same effect as later ones. There have been some attempts before to take into account, Graves et al. [8] dampened older changes, making their effect on prediction less influential.

6.2 Conclusions

The purpose of this thesis was to gain an understanding of how software quality could be improved by measuring it. During this work, several things were found out about software quality.

Software quality means how well software fulfils the requirements. In practice, meaning that does the software does what it is supposed to do. Not only functionally, but how well it fulfils the expectations. Functionality can be tested easily, but fulfilling the expectations is much harder as it is a more subjective topic.

During the literary review, it was found that half of the bugs are caused by incorrect requirements or misinterpreted specification. It implies that software cannot be made perfect just by improving software development. Also, the process affecting the development needs to improve for providing better requirements toward software development.

It is economically more feasible to detect bugs early; thus, it is attractive to measure source code and detect failures from source code, instead of compiling and running the software and find them later in development. Static code analysis tools are used to measure source code and other metrics related to the product itself. These metrics are called product-related metrics. Another category is then process-related metrics; these metrics

measures how source code was developed. This, in practice, means how software was changed over time.

There is evidence showing that process-related metrics are more important in predicting failures in software than product-related metrics. Indeed, it is also established that a simple number of lines is as good in predicting faults in software as any other more complicated static code analysis metric.

The most important metric predicting software quality is how well organization structure matches the software structure. Measuring this is not easy, but it can be taken into consideration when planning organization structure. In fact, this is one of the aspects affecting the planning of teams at Silicon Labs. That teams interfaces match the software component interfaces.

Proof of concept solution used git change history to measure quality metrics from the software. It proves that Git can be used to harvest information from software development. Although it also means that git commits need to be tagged with proper issues. There is now ongoing discussion to mandate this, and then force the inclusion of tags into Git commits by automatically validating them.

Proof of concept was applied to the software repository; results are inconclusive but encouraging. One interesting finding was that bugs seem to be caused by features. Although this sound obvious but it could be used to direct software quality assurance resources to verify new features, instead of verifying old features.

This proof of concept solution is promising and could be further improved by taking into account the threats to validity.

References

[1] D. Chappell, “THE THREE ASPECTS OF SOFTWARE QUALITY: FUNCTIONAL, STRUCTURAL, AND PROCESS,” 2018. [Online]. [Accessed 19 10 2018].

[2] M. Immonen, Tieto Software Product Quality Analysis system. Master’s thesis., Degree Programme in Information Technology. Tampere University of Applied Sciences., 2009.

[3] T. Moisio, Further Development of Tieto Software Product Quality Analysis System. Master's thesis, Degree Programme in Information Technology. Tampere University of Applied Sciences., 2012.

[4] J. Viljanen, Measuring software maintainability. Master's Thesis, Degree Programme in Computer Science and Engineering. Aalto University, 2015.

[5] G. Keshavarz, N. Modiri and M. Pedram, “Metric for Early Measurement of Software Complexity,” *International journal on computer science and engineering,* vol. 3, no. 6, pp. 2482-2490, 2011.

[6] L. Bass, P. Clements and R. Kazman, Software Architecture in Practice, 3rd ed., Addison-Wesley, 2013.

[7] ISO/IEC 25010: Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models, Geneva, Switzerland: International Organization for Standardization, 2011.

[8] T. L. Graves, A. F. Karr, J. S. Marron and H. Siy, “Predicting fault incidence using software change history,” *IEEE Transactions on Software Engineering,* vol. 26, no. 7, pp. 653-661, 2000.

[9] B. Henderson-Sellers, Object-Oriented Metrics: Measures of Complexity, Prentice Hall, 1995.

[10] B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsbarrow, N. J. Ward and D. W. R. Marsh, “Industrial perspective on static analysis,” *Software Engineering Journal,* pp. 69-75, 1995.

[11] S. Montagud, S. Abrahão and E. Insfran, “A systematic review of quality attributes and measures for software product lines,” *Software Quality Journa,* vol. 20, no. 3-4, pp. 425-486, 2012.

[12] T. J. McCabe, “A Complexity Measure,” *IEEE Transactions on Software Engineering,* pp. 308-320, 1976.

[13] A. H. Watson and T. J. McCabe, “Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric,” Computer Systems Laboratory National Institute of Standards and Technolog, Gaithersburg, 1996.

[14] A. Oram and G. Wilson, Making Software: What Really Works, and Why We Believe It, O’Reilly Media Inc, 2011.

[15] C. Ebert and J. Cain, “Cyclomatic Complexity,” *IEEE Software,* vol. 33, pp. 27-29, 2016.

[16] M. H. Halstead, Elements of Software Science, Amsterdam: Elsevier, 1977.

[17] S. Henry and D. Kafura, “Software Structure Metrics Based on Information Flow,” *IEEE Transactions on Software Engineerin,* Vols. SE-7, no. 5, pp. 510-518, 1982.

[18] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering,* ADDISON-WESLEY PUBLISHING COMPANY, 1975.

[19] A. J. Albrecht and J. E. Gaffney, “Software Function, Source Lines of Code, And Development Effort Prediction: A Software Science Validation,” *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,* Vols. SE-9, no. 6, pp. 639-647, 1983.

[20] P. Oman and J. Hagemeister, “Metrics for assessing a software system's maintainability,” *Proceedings Conference on Software Maintenance,* pp. 337-344, 1992.

[21] T. M. Khoshgoftaar, E. B. Allen, R. Halstead and G. P. Trio, “Detection of Fault-Prone Software Modules During a Spiral Life Cycle,” *Proceedings of International Conference on Software Maintenance,* pp. 69-76, 1996.

[22] T. M. Khoshgoftaar and R. M. Szabo, “Improving Code Churn Predictions During the System Test and Maintenance Phases,” *International Conference on Software Maintenance,* pp. 58-67, 1996.

[23] “SonarQube,” [Online]. Available: https://www.sonarqube.org/. [Accessed 8 January 2020].

[24] L. H. Etzkorn, S. Gholston and W. E. Hughes, “A Semantic Entropy Metric,” *Journal of Software Maintenance,* vol. 14, no. 4, pp. 293-310, 2002.

[25] A. Binstock, “Deciding on Metrics,” *Software Development Times,* no. 171, p. 37, 2007.

[26] M. E. Conway, “How do committees invent?,” *Datamation magazine,* pp. 28-31, 1968.

[27] H. Gall, K. Hajek and M. Jazayeri, “Detection of Logical Coupling Based on Product Release History,” *International Conference on Software Maintenance,* pp. 190-198, 1998.

[28] T. Zimmermann, P. Weißgerber, S. Diehl and A. Zeller, “Mining Version Histories to Guide Software Changes,” in *In Proceedings of the 26th International Conference on Software Engineering (ICSE ’04). IEEE Computer Society*, USA, 2004.

[29] R. M. Bell, T. J. Ostrand and E. J. Weyuker, “Does Measuring Code Change Improve Fault Prediction?,” in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering, PROMISE 2011*, Alberta, Canada, 2011.

[30] A. Schröter, T. Zimmermann, R. Premraj and A. Zeller, “If your bug database could talk...,” in *IN PROCEEDINGS OF THE 5TH INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING, VOLUME II: SHORT PAPERS AND POSTERS*, 2006.

[31] N. Nagappan, B. Murphy and V. Basili, “The influence of organizational structure on software quality,” *ACM/IEEE International Conference on Software Engineering,* vol. 30, pp. 521-530, 2008.

[32] A. Meneely and L. Williams, “Secure Open Source Collaboration: An Empirical Study of Linus' Law,” in *Proceedings of the 16th ACM conference on Computer and communications security*, New York, USA, 2009.

[33] C. Bird, N. Nagappan, B. Murphy, H. Gall and P. Devanbu, “Don’t Touch My Code! Examining the Effects of Ownership on Software Quality,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering*, Szeged, Hungary, 2011.

[34] L. Madeyski and M. Jureczko, “Which process metrics can significantly improve defect prediction models? An empirical study,” *Software Quality Journal,* vol. 23, pp. 393-422, 2015.

[35] A. Tornhill, Software Design X-Rays, Pragmatic Bookshelf, 2018.

[36] Empear, “CodeScene,” [Online]. Available: https://codescene.io/. [Accessed 20 3 2020].

[37] L. Hatton, “Reexamining the Fault Density– Component Size Connection,” *IEEE Software,* vol. 14, no. 2, pp. 89-97, 1997.

[38] R. V. Basili and T. B. Perricone, “Software errors and complexity: an empirical investigation,” *ACM,* vol. 27, no. 1, pp. 42-52, 1984.

[39] V. Vaishnavi, W. Kuechler and S. Petter, “Design Science Research in Information Systems,” 2017. [Online]. Available: http://www.desrist.org/design-research-in-information-systems/.

[40] M. D'Ambros, M. Lanza and H. Gall, “Fractal Figures: Visualizing Development Effort for CVS Entities,” *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis,* pp. 1-6, 2005.

[41] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *27th International Conference on Software Engineering, 2005. ICSE*, Saint Louis, MO, USA, 2995.

[42] M. Jurezcko and L. Madeyski, “Which process metrics can significantly improve defect prediction models? An empirical study,” *Software Quality Journal,* vol. 23, p. 393–422, 2015.

[43] F. Chollet, Deep Learning with Python, Manning Publications, 2017.

[44] G. Taraldsen, “Nonlinear probability. A theory with incompatible stochastic variables,” arXiv, 2017.

[45] T. J. Ross, Fuzzy Logic with Engineering Applications, 4th Edition, Wiley, 2016.

[46] Y. Singh, P. K. Bhatia and O. Sangwan, “Predicting software maintenance using fuzzy model,” *SIGSOFT Softw. Eng. Notes,* vol. 34, no. 4, pp. 1-6, 2009.

[47] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord and J. Stafford, Documenting Software Architectures, Addison-Wesley Professional, 2010.

[48] D. Parnas, “Information Distribution Aspects of Design Methodology,” *IFIPS Congress,* pp. 339-334, 1971.