BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

SOURCE CODE METRICS FOR QUALITY IN JAVA

KVALITATIVNÍ METRIKY ZDROJOVÉHO KÓDU V JAZYCE JAVA

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **VLADYSLAV SHERSTOBITOV**
AUTOR PRÁCE

SUPERVISOR **Ing. ZBYNĚK KŘIVKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Department of Information Systems (DIFS) Academic year 2019/2020

Bachelor's Thesis Specification

22417

Student:	**Sherstobitov Vladyslav**
Programme:	Information Technology
Title:	**Source Code Metrics for Quality in Java**
Category:	Compiler Construction

Assignment:

1. Study a chosen library for Java source code analysis (e.g. Spoon) and study the possibilities to access several open source projects implemented in Java programming language using existing tools and APIs (e.g. GitHub).
2. Study various code metrics that influence the project quality with focus on the quality and usability of the entire software project such as cyclomatic complexity, maintainability index, and density of comments.
3. According to the instructions of the supervisor/consultant, prepare the set of testing projects in Java (approx. several dozens or hundreds) to assess their quality.
4. According to the instructions of the supervisor/consultant, design a tool to analyze these projects with focus on the evaluation of useful metrics and automatic assessment of the code quality in the Java software projects.
5. Implement the designed tool.
6. Test the tool using the prepared set of testing projects, evaluate and sum up the results including the proposal of some future improvements.

Recommended literature:

- Norman Fenton, James Bieman. Software Metrics. *A Rigorous and Practical Approach*, Third Edition. CRC Press, 2014.
- Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, Lionel Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. In Software: Practice and Experience, Wiley-Blackwell, 2015. Doi: 10.1002/spe.2346.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Křivka Zbyněk, Ing., Ph.D.**
Consultant:	Tišnovský Pavel, Ing., Ph.D., RedHatCZ
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	November 1, 2019
Submission deadline:	July 31, 2020
Approval date:	October 16, 2019

Abstract

In order to measure and improve code quality, there needs to be a system in place consisting of quantitative metrics and their analysis. With software quality metrics that represent how well source code is written, source code can be evaluated. Based on this evaluation, code may meet or not meet the criteria set, which may be used for many purposes.

The current research shows the program developed based on identified code qualities and related metrics, methods used in creating the program, and test results based on open-source projects.

Abstrakt

Pro kontrolu a zlepšovaní kvality zdrojového kódu musí být zaveden systém, který se skládá z kvantitativních metrik a jejich analýzy. S použitím metrik kvality softwaru, které reprezentují, jak dobře je zdrojový kód napsán, lze hodnotit zdrojový kód. Na základě tohoto hodnocení může kód splňovat nebo nesplňovat stanovená kritéria, která mohou být použita pro mnohá účely.

Daný výzkum prezentuje program vyvinutý na základě identifikovaných vlastností kódu a souvisejících metrik, metod používaných při tvorbě programu a výsledků testů založených na projektech s otevřeným zdrojovým kódem.

Keywords

Software quality, Java Spoon, quality of code, metrics, software metrics.

Klíčová slova

Kvalita software, Java Spoon, kvalita zdrojového kódu, metriky, softwarové metriky.

Reference

SHERSTOBITOV, Vladyslav. *Source Code Metrics for Quality in Java.* Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Zbyněk Křivka, Ph.D.

Rozšířený abstrakt

Pro kontrolu a zlepšovaní kvality kódu musí být zaveden systém, který se skládá z kvantitavních metrik a jejich analýzy. Pro vybudování takového systému, byly stanovené konkrétní metriky zdrojového kódu pomoci kterých je možné identifikovat, jak dobře je zdrojový kód napsán. Taky program musí umožnit stanovit, pokud požadovaná kritéria pro dany program byli splněné. Výsledný program musí být schopný vyhodnotit kvalitu zdrojového kódu.

Na základě ISO standardu v práci byly identifikované a popsané klíčové vlastnosti, které popisují dobře vypracovaný kód. Na základě těchto vlastností bylo vybráno a důkladně popsáno pět následujících metrik zdrojového kódu: počet řádků kódu, hustota komentářů, cyklomatická složitost, index udržovatelnosti a Halsteadův index.

Knihovna Java Spoon umožnuje generování abstraktních syntaktických stromů. Pomoci AST a regulárních výrazů, použitých na vyhodnocení uzlů stromu, daný program vypočítává výsledky pro každou metodu, udává průměrné hodnoty pro třídy a také pro celý projekt. Poté jsou výsledky porovnávány s výsledky z jiných projektu. Po porovnaní hodnot metrik uživatel dostane percentil kvality kódu pro cyklomatickou složitost, index udržovatelnosti a hustotu komentářů, a na základě toho může zjistit, jak moc kvalitní kód, v porovnaní s ostatními projekty má.

Vytvořeny program i práce se zaměřují na jazyk Java. Pro získání zdrojových textů projektů kvůli otestovaní výsledného programu bylo třeba získat přistup ke 100 projektů s otevřeným zdrojovém kódem. Všechny testovací projekty byli získaný pomoci GitHub.

Také je nutné prozkoumat a vzít v úvahu různé kódové metriky, které ovlivňují kvalitu projektu se zaměřením především na kvalitu a jednoduchou použitelnost celého softwarového projektu. Pro danou práci bylo vybráno pět kódových metrik, viz výše.

Byl navržen nástroj pro analýzu těchto projektů se zaměřením na ohodnocení užitečných metrik a automatické vyhodnocení kvality kódu v softwarových projektech napsaných v jazyce Java.

Pomocí připravené sady testovacích projektů byl otestován navržený nástroj včetně návrhu některých možných budoucích vylepšení. Poté byly ohodnoceny a shrnuté výsledky. Byla provedena analýza pro zjištění, která metrika má největší vliv na výslednou kvalitu programu.

Source Code Metrics for Quality in Java

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Zbyněk Křivka, Ph.D. The supplementary information was provided by Ing. Tišnovský Pavel, Ph.D. All the relevant information sources, which were used during the preparation of this thesis, are properly cited and included in the list of references.

. . . .
Vladyslav Sherstobitov
July 31, 2020

Acknowledgements

I would like to thank my supervisor Ing. Zbyněk Křivka, Ph.D. for his guidance and valuable feedback. Also, my sincere thanks go to Ing. Tišnovský Pavel, Ph.D. for his useful advice.

Contents

Chapter 1

Introduction

With the increasing importance of software in everyday tasks and the focus of developers on software quality, it is essential to track quality during all stages of development.

The goal of the project is to define metrics crucial to software quality, how they are applied in code evaluation, and how this data can be used to improve code quality for a better end product. As well, to provide a program capable of measuring code quality by the means of metrics defined by renowned research based on standards.

The task is to achieve an easy to implement and use command-line application for evaluation of source code written in Java language, and for it to have an output of quantitative metrics such as Lines of Code, Cyclomatic Complexity, Halstead Metrics, Comment Density, and most importantly Maintainability Index. The project focuses on the importance of maintainability as a software quality and Maintainability Index as the means of translating quantitative data into a qualitative measure.

This project explains how with the help of the Java Spoon library, abstract syntax trees, and regular expressions, the code calculates results for each method, gives an average values for classes, and then the outcomes are compared between programs for analysis to identify the outliers.

1.1 Document Structure

There are 6 chapters in this thesis.

Chapter 2 describes the importance of software quality metrics. This chapter provides information about quantitative and qualitative metrics and how they are connected. It covers how qualitative evaluation could be obtained based on the quantitative metrics received from the static analysis of the code. Section 2.2 introduces ISO standards for the evaluation of software quality. It is followed by Sections 2.3 and 2.4 where it is described why maintainability is the main focus during software development and how it can save a lot of resources during each phase of the product development.

Chapter 3 contains information about every metric that was used in this project. Each section is started with a brief theoretical overview of the metric followed by subsections about calculations, applications, and implementations. In the final Section 3.5 recommended values for metrics are introduced.

In Chapter 4 Implementation of the CLI application for static analysis of the source code is described. It covers a basic overview of the architecture of the application and describes how with the help of Java Spoon Library, abstract syntax tree, and usage of regular expression a detailed report about the metrics of the project is received. Also, possible improvements were defined at the end of Section 4.5.

Chapter 5 covers the testing process for the applications. Also, it describes how tested projects were accessed. Section 5.5 evaluates and sums up received results including a visual overview in the form of graphs.

Chapter 6 concludes how the main goal of this research was achieved.

Chapter 2

Software Metrics

2.1 Software Quality Metrics

Quality metrics are an important part of effective quality management in every process and plan. Simply put, a code quality metric is a measure that allows translating software features into quantitative data, giving an ability to measure performance, and with an ability to measure performance the software development can become more effective [6].

Many aspects can be evaluated only by the users by interacting with the software or its prototype, some can involve different hardware, and most can be identified by the analysis of the source code. This means that software quality measurement methods can be broken down into two major groups: quantitative and qualitative.

Quantitative methods use numbers. They measure countable objects represented by numbers and are necessary for statistical analysis, progression, and comparison with similar products. The biggest advantage of quantitative data is that it can be prepared automatically, measured at any stage of software product completion, and their ease of integration. Importantly, methods need to have specified targets or a model for comparison in order to be effective [3].

Qualitative methods use objects differently than numbers. Data can be acquired by the means of interviews, surveys, observations. This complicates the process of implementation, may make the data achieved less objective and difficult to process, however, the outcome may give a richer and more informative result. Qualitative methods in most cases require a complete product or its prototype and may assist with future improvements of measurement structures used [3].

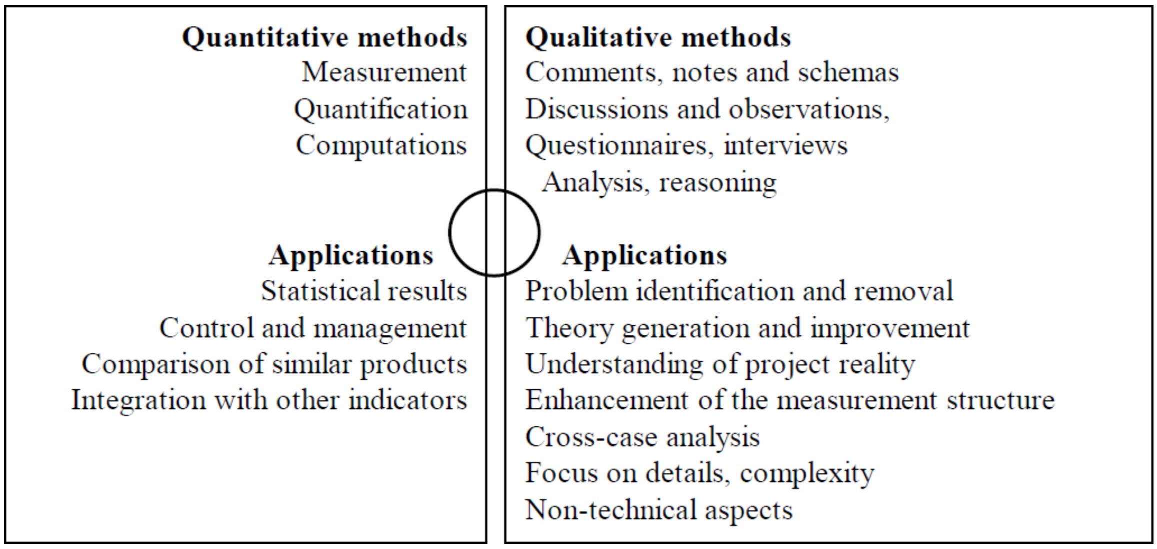
Quantitative methods
Measurement
Quantification
Computations
Applications
Statistical results
Control and management
Comparison of similar products
Integration with other indicators
Qualitative methods
Comments, notes and schemas
Discussions and observations,
Questionnaires, interviews
Analysis, reasoning
Applications
Problem identification and removal
Theory generation and improvement
Understanding of project reality
Enhancement of the measurement structure
Cross-case analysis
Focus on details, complexity
Non-technical aspects

Figure 2.1: Comparison of quantitative and qualitative methods in aspects of techniques and applications. The Figure is taken from [3].

Overall, this means that both measurements complement each other and allow for a greater final product, but during the development well defined quantitative metrics will have the following benefits:

1. Quantitative metrics can be tracked during the whole cycle showing weaknesses and dangers without the need to wait until the end of the development to see the results.
2. Productivity depends on the time spent on tasks and which tasks in particular. In order to track productivity and make improvements, quantitative metrics are needed to prioritize the tasks for the team.
3. Quantitative metrics can be used as a means of communication. Keeping metrics in the target will mean that there are no issues with the project if there is a deviation from target progression that can be reported to the management by using established metrics improving awareness and workflow.

2.2 Software Quality Models

In order to identify what metrics are necessary to improve and measure software quality it is necessary to first identify what characteristics a finished software product needs to have.

Such characteristics were defined in ISO 9126 — Software Product Evaluation — Quality Characteristics and Guidelines for their Use [8] which was published 1991 due to increase of computer usage in many application areas, and its correct operation is critical to business success and human safety. It was divided into four parts:

- quality model
- external metrics
- internal metrics

- quality in use metrics.

And the quality model itself consists of 6 key characteristics: Maintainability, Efficiency, Portability, Reliability, Functionality, Usability, see Figure 2.2.

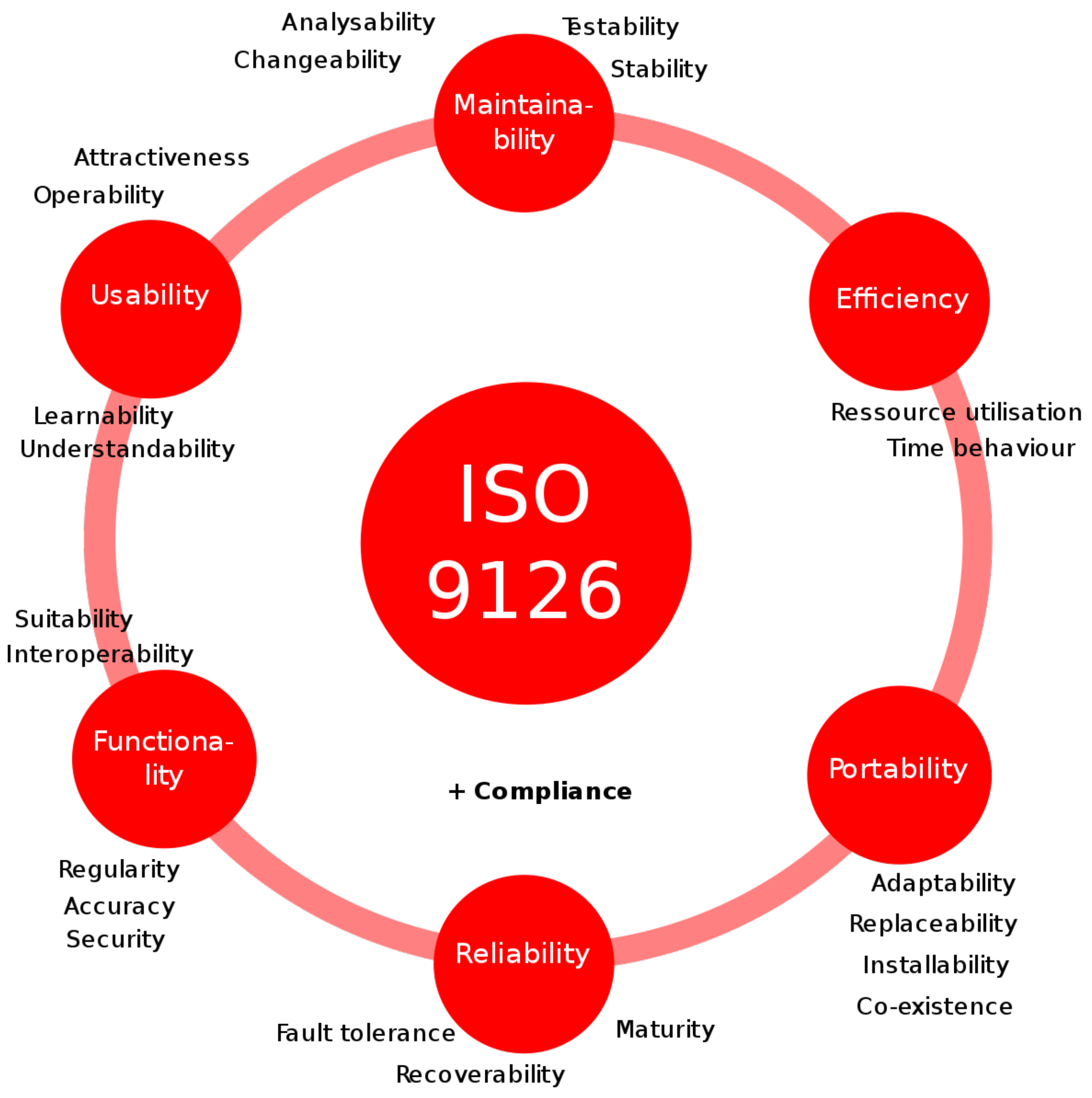

Figure 2.2: ISO 9126. The Figure is taken from [4].

With the growth of the software sector, on March 1, 2011, ISO/IEC 9126 was replaced by ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models [9]. According to it, software quality is the most important factor in software development as it defines customer satisfaction and determines the success of the software project. To better suit modern requirements a set of adjustments has been made with the main two being new characteristics, see Figure 2.3.

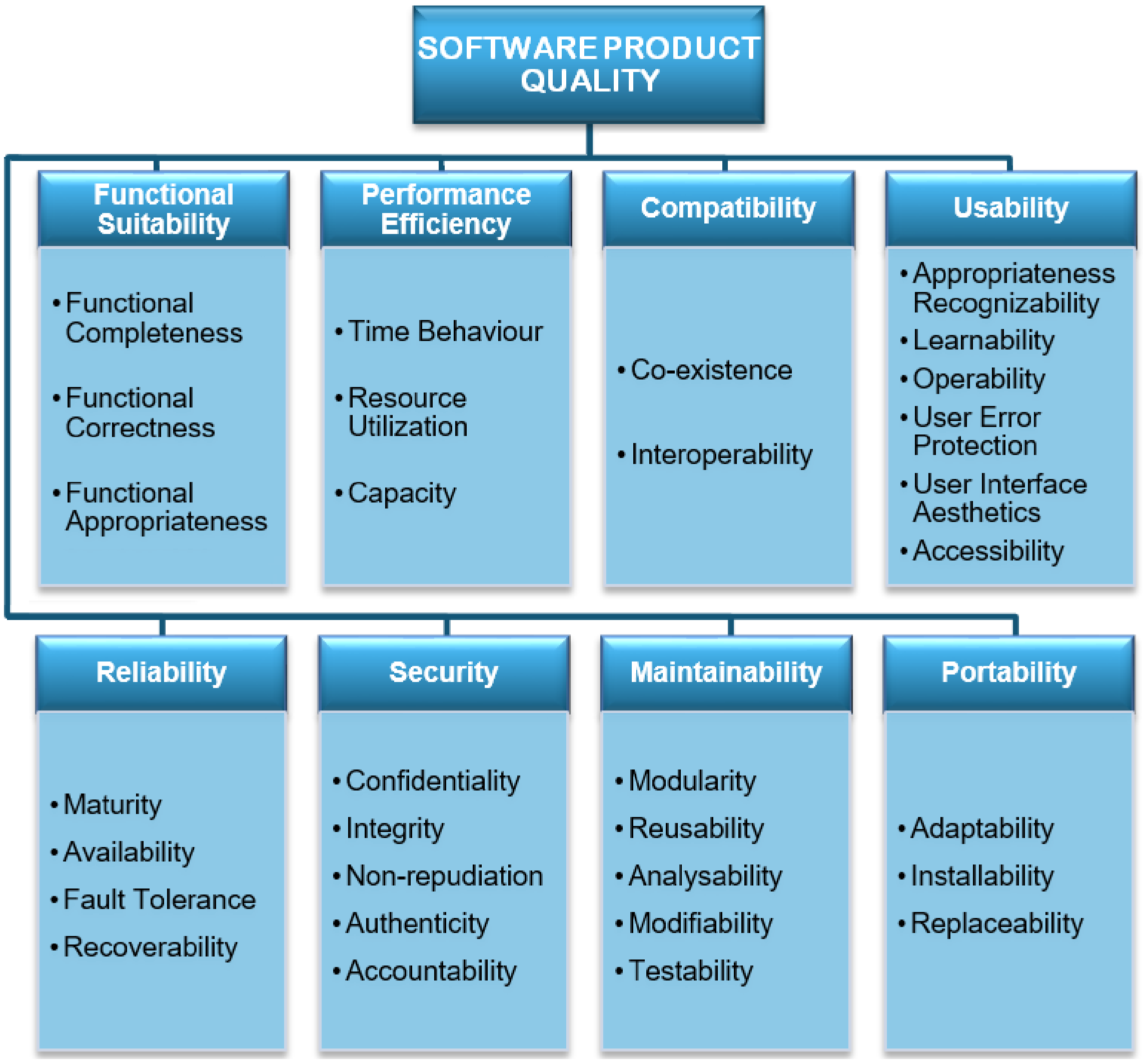

Figure 2.3: ISO 25010. The Figure is taken from https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

Previously it was defined that the goal is to achieve a set of quantitative metrics in order to measure software quality during development, and in this sense the key quality is Maintainability with its sub–characteristics.

2.3 Software Maintainability

Software maintainability represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements [9].

Software maintainability differs from hardware maintainability by the fact that it does not "wear out" and ages in other ways. If the environment in which software functions stayed the same, it could function indefinitely and in most cases without any degradation, and in the state, it was originally designed. However, we are not isolated from changes in software and hardware surrounding said software products, and in most cases, software

Corrective maintenance	**Perfective maintenance**
The process of receiving reports of errors, diagnosing the problem, and fixing it.	The process of receiving suggestions and requests for enhancements or modifications, evaluating their effects, and implementing them.
Adaptive maintenance	**Preventative maintenance**
The process of assessing the effects of "environmental changes" on a software system, and then modifying the system to cope with those changes.	The process of planning code reorganizations, implementing them, and testing to ensure that they have no adverse impact.

Table 2.1: Main factors of software maintenance.

systems are modified after they have been delivered to include new features required in the future. Taking this into consideration, four major factors in software maintenance [13] can be identified in Table 2.1.

2.4 Reasons to Track Maintainability

An increasing rate of changes in technologies helps provide a better software product at the end, but it requires existing code to be adjusted to fit modern requirements. The cost of software maintenance is rising dramatically and it has been estimated in [11] that nowadays software maintenance accounts for more than 90% of the total cost of software, whereas it was around 50% a couple of decades ago [5]. Knowing how maintainable software is will have a financial and quality impact on the final product.

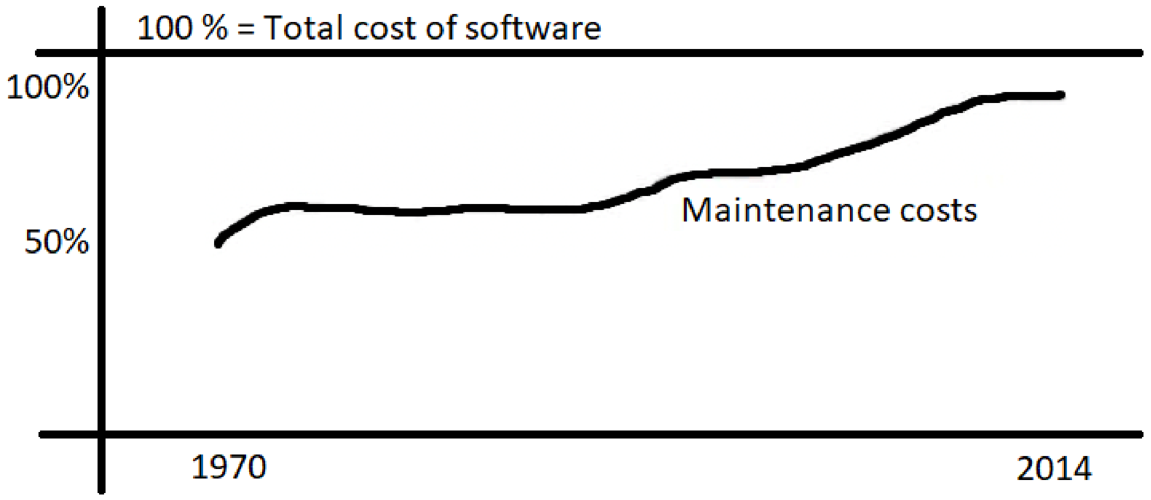

Figure 2.4: Development of Software maintenance costs as a percentage of the total cost. The Figure is taken from http://asq.org/public/wqm/how-to-save-on-software-maintenance-costs.pdf

Financial Impact

From the financial aspect, low maintainability will affect the product by an increased rate of software rot.

Software rot is a state of software when it becomes more difficult to change than required [12]. It does not mean that complicated and hard to update code is rotten, but more than changes will imply additional spending in one of the three key areas:

1. **Time** – an inability to provide changes before the deadline.
2. **Resources** – more team members, more equipment needed to implement the changes.
3. **Quality** – if the deadline must be met and no additional resources are available quality will suffer.

In order to define software rot more accurately, a couple of factors need to be noted [7]:

1. Difficulty and resource requirements do not identify software rot but may be caused by the nature of the code and requested features.
2. If software functions, it does not mean the absence of software rot. The code may still function appropriately but implementing changes may be harder than expected.
3. Rotten code does not mean that software loses profitability.

Reasons of Software Rot

One of the biggest reasons for software rot is a concept of technical debt. It is a metaphor by Ward Cunningham meaning when making a decision, an easier path is taken instead of a more practical approach, and from this point, an implied cost for an additional rework grows [18]. If this debt is overlooked, with every subsequent release, it will lead to code becoming unmaintainable, and additional resources will need to be used to rewrite whole parts.

Instances of technical debt:

- Inconclusive or insufficient requirement definition. This may help to start on a project earlier, however, will require redesigning later.
- Deadline pressure when quicker solutions are given priority.
- Existence of tightly functioning components where subjects are not modular.
- Lack of documentation.
- A parallel development in different branches.
- Delayed refactoring – even if an issue is identified it may be addressed later than required, further increasing the debt.
- Lack of knowledge and behavioral issues (lack of leadership or ownership).

Overall, technical debt is not always a bad thing. It is a rational decision to go with a faster solution to meet the deadline and address the shortcut taken after the release, however, it is important to address the shortcut on time. In case even a few known things are not changed on time, it would be increasingly hard to estimate how much resources will be required later for a fix, and prolonged avoidance will be a major reason for software rot.

Quality Impact

Another reason to monitor software maintainability is its quality of being an enabler of other code quality characteristics. It is best seen when maintainability sub-characteristics from ISO 25010 are reviewed [9]:

- **Modularity** – degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
- **Reusability** – degree to which an asset can be used in more than one system, or in building other assets.
- **Analyzability** – degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
- **Modifiability** – degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
- **Testability** – degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

Having high maintainability will allow for easier improvements in other 6 software characteristics whether it is to make an improvement in a software feature, optimize resource utilization, or implement a range of requests from the end-user.

2.5 Conclusion

Having highly maintainable code and tracking its maintainability throughout the lifecycle will increase overall software quality. It is possible by maintainability having an impact on code quality characteristics defined in ISO 25010. At the same time, from the definition of technical debt and software rot – the main reasons for decreases in maintainability are changes brought into the code incorrectly. This means that software rot can be slowed down by the implementation of proactive measures.

Maintainability sub-characteristics can be used to create a system of metrics to assist with maintainability measurement, see Figure 2.5.

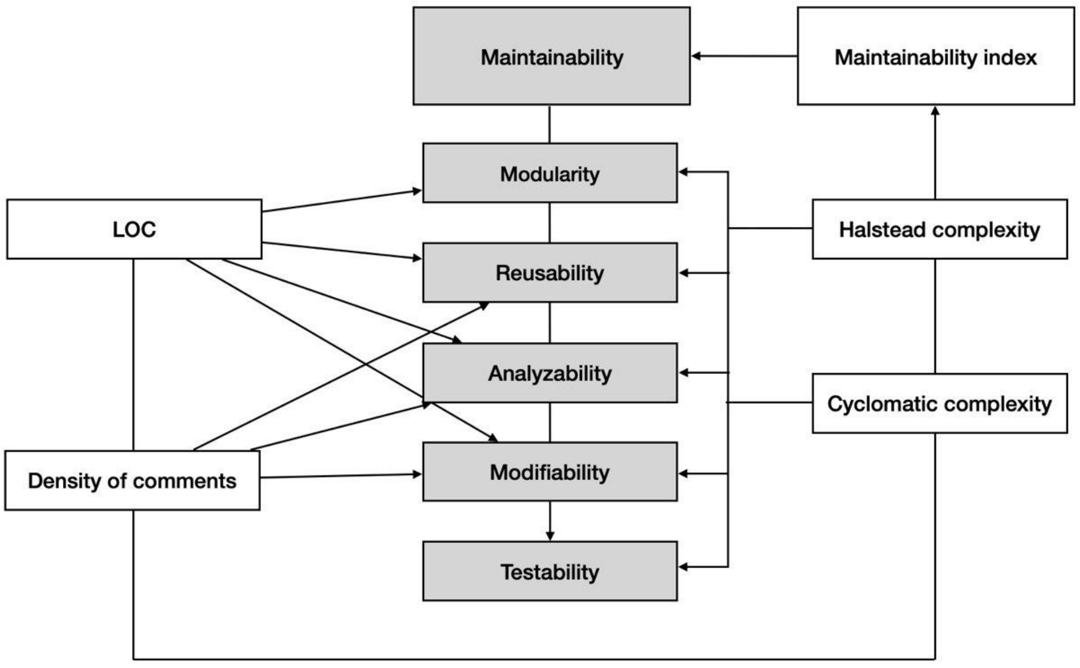

Figure 2.5: Relation between Software Maintainability and Maintainability Index.

Using Cyclomatic Complexity, Halstead complexity measures, Density of Comments we can calculate Maintainability Index proposed by Oman and Hagemeister. All metrics are quantitative and satisfy requirements set previously in this chapter. As well, they cover sub-characteristics of maintainability defined in ISO 25010 [9] according to research provided in Chapter 3. Lastly, they can be used in order to predict and track software quality by reviewing software source code as it is shown in Chapter 4.

Overall the most important principles about metrics in software quality are:

- They can measure performance, lead the team in the correct direction, and track progress.
- Based on the results, the management can implement actions and assign them correct priority based on deviations of metrics during the development process.

- Metrics are no bounds and are here not to limit developers, there are instances where a correct decision would be not to follow the most rational way according to raw numbers.

Chapter 3

Source Code Metrics

After defining the required characteristics and metrics that will allow us to monitor progression, it's essential to find a correct approach to calculation. Further calculation methods will be used in the application.

3.1 Lines of Code

Lines of Code is one of the most traditional metrics to quantify the complexity of software, as well as it is one of the simplest to understand and count [15]. Not only Lines of Code is a useful metric itself, but a vital component in the calculation of many others. There are four types of LOC:

1. **LOCphy** – a sum of physical lines of code.
2. **LOCbl** – a sum of blank lines.
3. **LOCpro** – a sum of program lines.
4. **LOCcom** – a sum of comment lines.

The best way to use LOC metrics is to set targets for LOCpro per method and class in order to maintain appropriate complexity (and in return maintainability, readability, and repairability), and as a result to justify or set requirements of higher than average LOCcom.

Physical (LOCpro) and logical (LLOC) are two major measures of Lines of Code metric. Physical lines of code is much easier to calculate but is sensitive to the formatting style of the developer. Logical lines of code while being harder to calculate are independent of style and formatting, and are more objective in code length and complexity [15].

```
for (int i = 0; i < 10; i++) System.out.println("Index is: " + i); /* How many
    lines of code is this? */
```

This example consists of 1 LLOC and 2 LOCpro. Comment lines are not included in the calculation, used only for the illustration.

```
/* How many lines of code is this? */
for (int i = 0; i < 10; i++) {
   System.out.println("Index is: " + i);
}
```

This example consists of a total 3 LLOC and 2 LLOpro. Comment lines are not included in the calculation, used only for the illustration.

3.1.1 Calculation

For further evaluation only logical lines of code and logical lines of commented code are important. Not all lines are calculated. Some of the lines do not complicate the overall logic, so must be excluded.

Lines of code that are excluded from the calculation:

1. **File docs (license doc)** – mostly contain information about the license, author, and data of creating this file.
2. **Empty lines** – depends only on the style of the programmer. Sometimes used for code separation or modularity.
3. **Import lines** – could be hundreds and mostly does not affect code complexity.
4. **Package lines**.
5. **Lines with empty comment line or start or end of block comment** – does not contain any information and could be replaced with one line inline comment.

```
#line LLOC LLOCcom

#1    (0)   (0)     /**
#2    (0)   (0)      * license information
#3    (0)   (0)      * author: some name
#4    (0)   (0)      */
#5    (0)   (0)     package some.package.example
#6    (0)   (0)
#7    (0)   (0)     import example.of.some.imp
#8    (0)   (0)
#9    (0)   (0)     /**
#10   (0)   (0)      *
#11   (1)   (1)      * javadoc example
#12   (0)   (0)      */
#13   (1)   (0)     public class Main {
#14   (1)   (0)       public static void main(String[] args) {
#15   (1)   (1)         // example of separate inline comment
#16   (0)   (0)
#17   (1)   (0)         if (args.length != 1) {
#18   (1)   (0)           System.err.println("Bad");
#19   (2)   (1)           System.exit(1); // example of inline comment
#20   (1)   (0)         }
#21   (1)   (0)       }
#22   (1)   (0)     }
```

This example consists of total 11 LLOC and 3 LLOcom.

3.2 Cyclomatic Complexity

Cyclomatic Complexity is a metric that was developed by Thomas J. McCabe, Sr. in 1976. Simply put, it shows the number of linearly independent paths in the section of code [19], meaning that if code has no conditionals or decision points, it's complexity will be equal to one, if there is a single `if` statement, there would be two possible outcomes and the complexity would be equal to 2, and, similarly, an `if` statement with two conditions (same as `switch`) would increase complexity by 2.

3.2.1 Calculation

The first way to calculate McCabe Cyclomatic Complexity (MCC) [19] is to use a formula:

$$MCC = E - N + 2P \tag{3.1}$$

Where
P = number of connected components
E = number of edges of the graph
N = number of nodes of the graph
However, MCC can be used for modules (subroutines, methods, etc.), and in this case P will always be equal to 1, simplifying the formula to:

$$MCC = E - N + 2 \tag{3.2}$$

The second way would be to use a control flow graph of a module, where the program begins executing at the node (1), then going to a conditional statement with one condition at the node (2), then going to a conditional statement with two conditions at node (6), and then exiting at the node (11), see Figure 3.1.

```
(1)  while (i < 100) {
(2)      if (arr[i] % 2 == 0) {
(3)          parity = 0;
         } else {
(4)          parity = 1;
(5)      }
(6)      switch (parity) {
             case 0:
(7)              System.out.println("a[" + i + "] is even");
             case 1:
(8)              System.out.println("a[" + i + "] is odd");
             default:
(9)              System.out.println("Unexpected error");
          }
(10)      i++;
      }
(11) int a = 10;
```

To calculate MCC from a graph, we need to count the number of edges and nodes, giving an MCC of 5 for the example above.

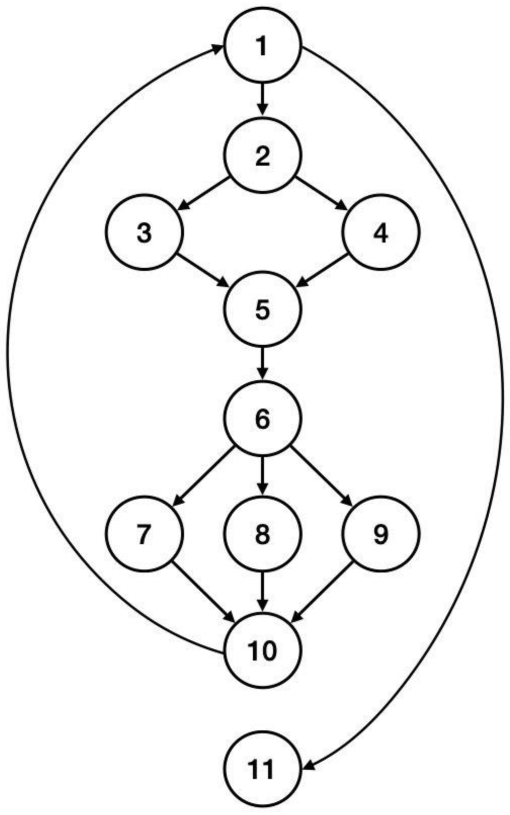

Figure 3.1: Cyclomatic Complexity example to the previous source code.

And using the formula mentioned previously:
MCC = 14 - 11 + 2 = 5

3.2.2 Application

One of the original applications of the metric by McCabe is to limit the complexity of the code for modules. The recommended use was to identify modules with the complexity **higher than 10** and subsequently to split them into smaller modules in order to keep the structure and readability. **Limits over 15** have been used successfully as well according to McCabe, however, should be reserved for projects that would have operational advantages over typical projects such as experienced staff, formal design, a modern programming language, structured programming, code walkthroughs, and a comprehensive test plan. Such projects would require additional testing for more complex modules [19].

Prevent Defects

There are multiple [17], [21] research conclusions that prove there is a correlation between the size of the program (measured in lines of code) with the frequency of defects, however, they are more inconclusive than correlations between MCC and defects. This does not necessarily imply that by always keeping MCC in check, the number of failures will be minimized, but by controlling Cyclomatic Complexity and limiting code complexity as follows should improve code and comment quality overall.

Importance of Cyclomatic Complexity in Unit Testing

Unit testing is a software testing method defined by individual units of code are tested to determine whether they are fit for use [10]. The goal of the unit test is to make sure that each segment of code satisfies the required conditions.

One of the biggest advantages of unit testing is its ability to be used in the early stages of development, as a result of improving the structure of the code, functionality, and identifying errors in the first cycles of development. In this case, MCC could be used to determine the number of test cases required to achieve thorough test coverage for a module.

In order to make sure that software is well functioning, tests should cover as many branches as possible. The more branches are covered, the fewer errors will be unnoted. This is the situation in which CC will assist since it identifies the lowest number of branches in a module, and by managing, CC in code number of tests necessary can be brought down to a minimum:

1. MCC is an upper bound for test cases required to achieve full branch coverage (to make sure that all branches have been executed at least once).

2. MCC is a lower bound to achieve full path coverage, since every with every additional IF statement number of paths will grow by a factor of two, and at the same time MCC will grow by 1. As for path and branch coverage, the following statement will always be true: the number of branches $\leq$ MCC $\leq$ number of paths.

As for management, unit testing oriented development will make time spent on developing more efficient as per points said above in addition to the outcome, that such code will be more maintainable and reusable.

3.2.3 Implementation

In this project, CC is calculated with the help of abstract syntax tree up until methods, and after, a number of keywords are calculated according to a predefined regular expression table.

The easiest way to calculate the Cyclomatic Complexity for Java programming language is to count a number of occurrences of those symbols: `if, while, for, case, catch, &&, ||, ?`

3.3 Halstead Complexity

Halstead metrics have been introduced in 1977 by Maurice Howard Halstead [1]. His goal was to identify measurable properties of software, and relations between them. Metrics are experimented and used extensively since that time and are one of the oldest measures of program complexity.

3.3.1 Calculation

Halstead's metrics are based on a system, where the source code is interpreted as a sequence of tokens and each token is classified (either as an operator or an operand. According to this, four quantities are calculated and the rest of the metrics are derived from them:

N_1 – number of operators

N_2 – number of operands

n_1 – number of unique operators
n_2 – number of unique operands

Halstead Length totals the number of operators and operands, and in case the number of statements is small and Halstead Volume is high, it would indicate, that the individual statements are quite complex.

$$N = N_1 + N_2 \tag{3.3}$$

On the other side, in case there are many different variables or a small number of variables are used repeatedly, it would indicate complexity among statements and measured by the **Halstead Vocabulary**.

$$n = n_1 + n_2 \tag{3.4}$$

The **Halstead Bugs** attempts to estimate the number of bugs that there are liable to be in a particular piece of code.

$$B = \frac{E^{\frac{2}{3}}}{3000} \text{ or } \frac{V}{3000} \tag{3.5}$$

To calculate the approximate difficulty of writing and maintaining the code, the **Halstead Difficulty** uses a formula to assess the complexity base on the numbers of unique operators and operands.

$$D = \frac{n_1}{2} * \frac{N_2}{n_2} \tag{3.6}$$

The **Halstead Effort** attempts to estimate the amount of work that it would take to recode a particular method.

$$E = D * V \tag{3.7}$$

The **Program Level** is the inverse of the Halstead Difficulty and would mean, that a low-level program is more likely to contain errors, than a high-level program.

$$L = \frac{1}{D} \tag{3.8}$$

Time to Implement would be proportional to the effort and according to Halstead, it should be divided by 18 to receive approximate time in seconds.

$$T = \frac{E}{18} \text{ (sec)} \tag{3.9}$$

The **Halstead Volume** uses program length and vocabulary size to give a measure of the amount of code written.

$$V = N * \log_2 n \tag{3.10}$$

3.3.2 Application

The Halstead metrics are applicable for the written code, however, one of the best uses would be during the development to follow complexity trends. This is done so, since maintainability should be a concern during the early stages and technical debt in further ones, and a significant complexity increase may be a sign of a high-risk module.

There are two weaknesses that need to be highlighted about Halstead metrics. The first is, that the most appealing metrics to follow would be Difficulty, Number of Bugs, and Time to Implement, and the issue is, that they are strongly based on assumptions and correlate mainly with Volume. As the size of the program grows, so does the number of bugs, difficulty, and time to implement. With the data calculated, it may not always be usable even with other metrics coupled, except for statistical analysis.

The second one is, that Halstead metrics measure lexical and textual complexity per calculations above, rather than structural complexity exemplified by Cyclomatic Complexity, and it was one of the reasons for heavy criticism. To address both issues, not all metrics are used, they are coupled together with Cyclomatic Complexity and most importantly, they show the best results as a strong component to Maintainability Index.

3.3.3 Implementation

Halstead metrics are calculated with the help of the abstract syntax tree and for the methods, regular expressions are used for calculation. regular expressions are used for identifying operands and operators.

3.4 Density of Comments

Density of Comments is a metric representing the percentage of comment lines in a given part of the source code, which is calculated by dividing the number of commented lines by a total number of lines of code [2]. The metric can be used as a quality indicator to see how much code is commented, and assume code maintainability, hence the survival and longevity.

Main Uses of Comments

1. **Planning and reviewing.** Comments can be used to outline the initial intention. In this case, not code itself should be described, but what it's meant to do. This can be used to compare needs with results, as well as introducing yourself to the code that was previously worked in (to understand the logic behind).

2. **Code description.** Even though comments can be used to explain written code, it is considered a bad practice for the reason, that if code needs explaining, most likely it's too complex, so it needs re-writing. The best use of this scenario would be during bug fixing, or when there is a need to explain why a particular block of code does not meet best practices.

3. **Algorithmic description.** Sometimes a comment may contain a solution to a specific problem, so they may require an explanation of the methodology. Even though it may sound more like an explanation of the code, it would be considered more of an explanation of the reason behind, making it easier to understand the logic and motivation.

4. **Resource inclusion.** In some occasions not only text can be used as comments, but it may contain diagrams, graphs, or even copyright notices.
5. **Metadata.** Metadata is often stored in a comment. Many maintainers put guidelines for feedback on any improvements, they may make. Other metadata may include the name of the original creator, the date of the first version, the name of the maintainer, names of people who edited the program so far, the URL of documentation, license information, etc.
6. **Debugging.** It is considered common practice to comment out a code snippet in order to identify the source of an error. By commenting out parts and running code an error can be isolated and corrected.
7. **Automatic documentation generation.** Sometimes it is appropriate to store in the comments metadata and documentation. In case functions and classes are commented properly, and annotations are kept, the documenting process may be simplified, and data in it will keep relevance with updates.
8. **Directive uses.** Normal comment characters in some cases are co-opted in order to create a particular directive for an interpreter or editor.
9. **Stress relief.** On some occasions, programmers will add comments to relieve stress by commenting on working conditions, tools used, or the quality of the code itself.

Comment Recommendations

With the definition of Comment density metric and uses of comment, requirements can be set in order to improve maintainability, repairability, and readability of code.

The first would be to have at least one comment per procedure with a clear description, as well, it is recommended to describe each parameter and a return value of a function(ranges of values to be expected and range returned). The procedure comment should follow the procedure declaration line. In case the procedure is long or complex it is advised to include comments inside the body.

Including recommended uses and positions it has been identified, that 10% of code should be commented. In some cases code can be self-descriptive, and thus commented less, but, if heavy use of comments is required, it may be a warning sign for code complexity and lack of maintainability.

As for readability, 10% as well is when clarity seems to peak according to research, since lack of comments may hide logic and abundance of unclear comments will have a negative effect too.

3.4.1 Application

In the program, Comment Density calculates the relation of comments (incl: JavaDoc, block comments and inline comments; excluding blank lines, imports and packages) to the logical lines of code.

3.5 Maintainability Index

Maintainability Index (MI) is a single-value indicator for how maintainable the source code is, proposed by Oman and Hagemeister, allowing to control and improve, how easy it will be to support, remove bugs, and implement new features [20].

3.5.1 Calculation

To calculate MI, values of the average Halstead Volume per module, the Cyclomatic Complexity, the number of Lines of Code, and the comment ratio of the system are needed [6]. In order to increase MI, Cyclomatic Complexity should be reduced, as well as the Volume (V), and the total number of Lines of Code (LOC). Overall this means that MI keeps all the qualities of metrics in it and serves as a good and simple summary of the overall state of the code.

MI is calculated with the help of multiple metrics, but it shouldn't be the only way of measure of how good a program is. Instead, it may be an effective tool in a summary with other metrics and may be used as a way to track program improvement in progression with each subsequent release, as well as in decision making to change existing code or to rewrite it. In addition to this part will be focused on reducing technical debt with the help of MI.

$$\begin{aligned} &171 - 5.2 * \ln(avgHV) - 0.23 * avgCC(g') - \\ &- 16.2 * \ln(avgLOC) + 50 * \sin\sqrt{(2.4 * perCM)} \end{aligned} \tag{3.11}$$

Where
HV = Halstead Volume
CC = Cyclomatic Complexity
LOC = Lines of Code
$perCM$ = % Comment Lines

3.5.2 Implementation

With the help of MI, negative changes will identify the growth of technical debt, from that point separate modules can be reviewed, and changes can be made to keep code readable and reliable. Standard benchmarks can be used, which are as follows:

LOCpro

- per function min – **4**, maximum – **40**.
- per file minimum – **4**, maximum – **400**.

There are two negative outcomes for Comment Density values:

1. Density of Comments being above the recommended level would mean that some functions are over commented (most likely due to lexical or logical complexity issues) and as a result, will negatively affect readability and maintainability.
2. If Density of Comments is below target, it would mean that complex code is under commented and issues may arise when new developers need to look at it, or when the developer returns to his own code but cannot understand the logic behind it as well negatively affecting readability and maintainability in terms of repairability.

Cyclomatic Complexity

- **1 - 10** – good Cyclomatic Complexity. Structured and well-written code that is easy to test.
- **10 - 20** – acceptable Cyclomatic Complexity. Fairly complex code that could be a challenge to test. Depending on what you are doing these sorts of values are still acceptable if they are done for a good reason.
- **20 - 35** – bad Cyclomatic Complexity. The code is complex and is hard to test. This code should be refactored, broke down into smaller methods, or using some design patterns.
- **35 >** extremely bad Cyclomatic Complexity. very complex code, that is not at all testable and almost impossible to maintain or extend. Should be fully refactored or rewritten.

If the maximum value for MCC is **greater than 15**, then in most cases code should be restructured to achieve better test coverage and readability.

Density of Comments

Recommended average 10%, in case of higher LOCpro and Cyclomatic Complexity may be increased.

Maintainability Index

- **140 - 171** – good maintainability. The code as a whole can be considered maintainable and reliable.
- **100 - 140** – moderate maintainability. Majority of the code meets the requirements, however, there is either a part of code that may be edited for improvement or it contains a reasonable explanation.
- **40 - 100** – poor maintainability. Most likely there are major issues with code complexity in combination with poor commenting, and as a result, will not be re-usable.
- **< 40** – extremely poor maintainability. Every single module needs to be rewritten, impossible to re-use.

3.5.3 Application

As the goal of MI is to identify the impact of changes in software on its maintainability the best application would be to track MI over a period of time. Calculating the slope between two major releases provides information about possible instances of software rot in specific modules.

Once the affected module is identified, tracing back to the previous point shows what changes, in particular, have affected the results, allowing to address possible issues in a timely manner.

If an application has a high LOC value, MI may as well be a great indicator for further analysis with a heat map in order to review the values of other metrics. This helps isolate code that is most impactful for further focus and research.

Overall, simple changes can be implemented on a regular base to save a lot of effort in the long term:

- Check for two types of modules: ones with low MI, and ones with trending degradation.
- Group modules by functionality and check how often changes are made in those groups (new features are implemented, modifications or updates are made), in order to assist the responsible team with data to further isolate and resolve the issue.
- In case data is inconclusive, cross-checks should be done focusing on metrics such as Cyclomatic Complexity.
- Review the amount of code written to implement new functionality. With time, code reviews should be performed in order to improve optimization. Having redundant code is one of the biggest reasons for technical debt increase.
- After any optimization changes are performed, tests should be run again to make sure MI has improved.

3.5.4 Summary

MI is calculated with the help of multiple metrics [20], but it shouldn't be the only way of measure of how good a program is. Instead, it may be an effective tool in a summary with other metrics and may be used as a way to track program improvement in progression with each subsequent release.

The best way to use MI is to manage technical debt in order to improve the overall quality of the software product and to cut down maintenance-related costs.

Chapter 4

Implementation

The main goal of the application is to provide an easy to use tool for a detailed report generation. The report is to contain data on relevant metrics that have been defined and described in Chapters 2 and 3.

The Java programming language was chosen for the application development with Maven being used for the application's building process, as well as an external Java Spoon Library for analysis and transformations of Java Source Code described in Chapter 4.3.

This is a command-line application. As an input, the user is only required to provide a project directory, and as an output, the user will get a detailed report. The report contains basic info about the project and metrics for each method, class, and average values for the whole project. For a better understanding of the values and to receive a qualitative evaluation, which shows a comparison in the form of percentile with other open-source projects, which will be described in more detail in Chapter 5. Based on the output the user should be able to identify critical points in the code and areas of opportunity for improvement to prioritize tasks for higher code quality.

Output analysis may be automated with basic UNIX command-line utilities. Overall, it is recommended to trend results over some time in order to track project quality and to avoid drops in the metrics by means of Continuous Integration or Continuous Delivery.

The program is purposefully written without a graphical user interface (GUI) for simplified automatization to receive results for every build after every release. Since the focus group are Java developers – using, automating, and personalizing a CLI program will not be an issue. Source code should be used as the input, and for simplified usage and configuration it is expected for code to be built. Unit test files should be ignored for optimized performance and results, otherwise, all `.java` files will be analyzed. Lines of Code and Density of Comments metrics should be calculated before Java Spoon usage since Java Spoon transforms source code into an abstract syntax tree and metrics cannot be calculated at this point. After successful LOC and Density of Comments calculation, the previously mentioned Java Spoon library should be used to receive AST as an output.

4.1 Use of the Program

The minimum requirements for launching the application are Java 8 and one external library – Java Spoon. Also, Maven will be needed for adding the required dependencies. The project could be launched via CLI with only one parameter — a destination to the tested directory.

On the output, the user will get the `statistics.txt` file which contains all the metrics for every single method, class, and average values for the whole project. An example of the output can be found in appendix B.

4.2 Metrics Calculation

Every metric is calculated separately, so for each metric AST is reviewed. Even though such an approach affects the performance of the program, it allows for the program to be modular, and new metrics can be added much more easily. A medium-sized project is analyzed within a few seconds and with a big sized project, its calculation may take up to a couple of minutes. It is expected for the program to be used with every release so no negative impact on productivity is expected, however, the benefit is with improved Continuous Integration or Continuous Delivery.

1. The Density of Comments is calculated for each class and the whole project. For this metric LLOC is used as mentioned in Section 3.4.
2. CC is calculated by node calculation within the AST. In the final result CC for every method is noted, as well as the CC sum for every class (as a sum of CC for methods within a class), and an average result for all classes. Getters, setters, and overwritten methods from `Object` class are excluded from the calculation since these methods can give false good results in metrics. In the final results total, CC is noted, however, results cannot be compared as a percentile of other open-source projects, since it correlates with project size, so it is more appropriate to review the percentile of average CC per method.
3. Halstead Index is calculated with usage of regex from `java.util` library since Java Spoon library cannot identify operands and operators, so instead regular expressions are used. Within the Hasted Index, multiple metrics are calculated, and even though some can be debatable, the main one is Halstead Volume since it is used for Maintainability Index calculation.
4. For the Maintainability Index, previously mentioned metrics are used with defined formula from Section 3.5.

4.3 Java Spoon library

Java Spoon enables Java developers to perform three operations [16] with the source code in a simple manner:

- **Analyze** – provide software engineers with the primitives to write their own analyses.
- **Transform** – enables to modify any part of the code.
- **Generate** – creates the possibility to generate code based on templates.

Its greatest advantage is that it allows performing listed operations without deep knowledge of parsing.

The Main Features of Java Spoon

1. Provides a Java metamodel for representing Java ASTs easy to read and transform.
2. API to transform and generate Java source code.
3. The use of generic typing for static checking of the analyses and transformations.
4. The native and seamless integration and processing of Java annotations.
5. A pure Java statically-checked templating engine.

Overview of Spoon

When a Java program is given as an input, it is parsed with a compiler to produce a first AST. Then, nodes are created and deleted in order to simplify the initial AST and to create a model that is easier and more intuitive to manipulate. This complete-time (CT) model is an instance of a Spoon metamodel. In case the user makes a transformation, the processing, and the templating, engine takes these transformations as an input and applies them to the Java model. In the end, the Spoon model contains all the original source code in the tree nodes.

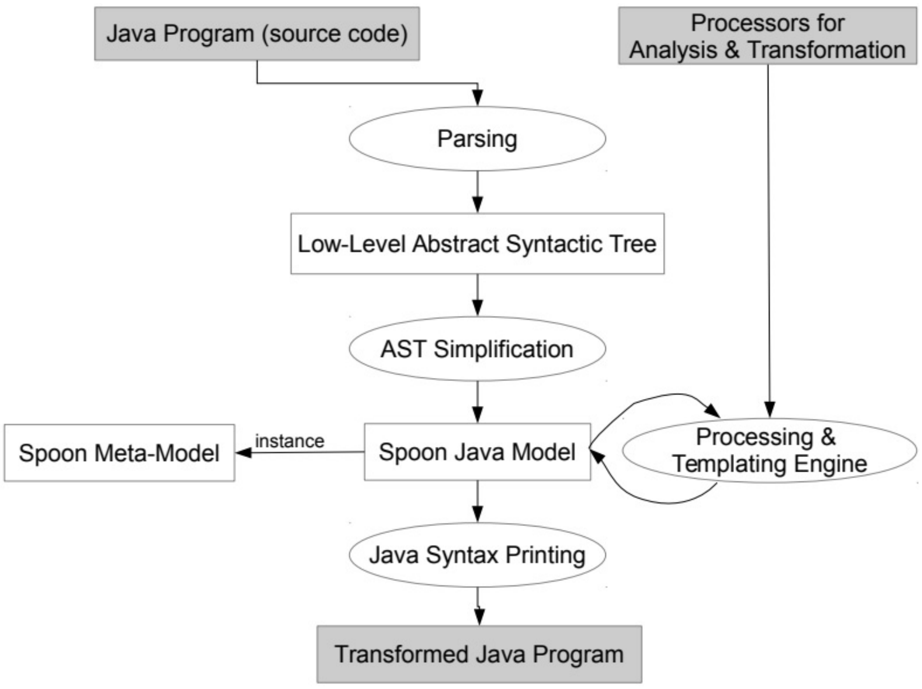
Java Program (source code)
Processors for Analysis & Transformation
Parsing
Low-Level Abstract Syntactic Tree
AST Simplification
Spoon Meta-Model
instance
Spoon Java Model
Processing & Templating Engine
Java Syntax Printing
Transformed Java Program

Figure 4.1: Java Spoon metamodel.

The Spoon Metamodel of Java

A programming language can have different metamodels. An AST is an instance of a metamodel. Each metamodel, and each AST, will be more or less appropriate depending on the kind of task. The Spoon metamodel was designed to be easily understandable by a normal java developer. The Spoon metamodel allows for the creation of its own analyses and transformations.

The Spoon metamodel consists of three parts:

- The structural part – contains the declarations of the program elements, such as interface, class, variable, method, annotation, and enum declarations. CTElement is a parent element and the rest inherits from it. In the image, all elements are prefixed by “CT” which stands for “compile-time”.

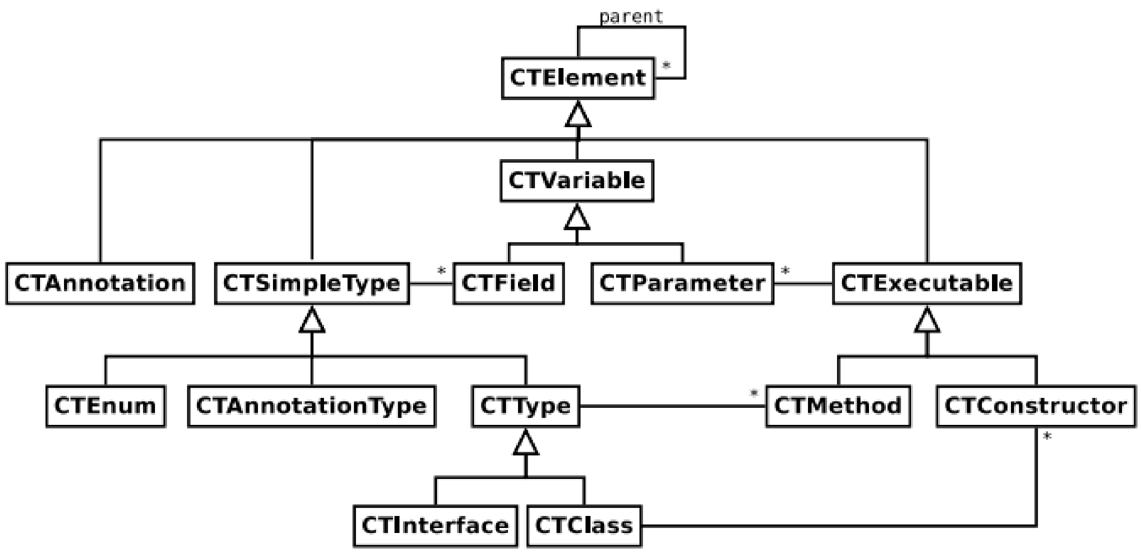

Figure 4.2: Structure of the element.

- The code part – contains the executable Java code, such as the one found in method bodies. Since Java is a complex language, the code metamodel figure does not contain all classes. There are two main types of code elements: statements and expressions.
- The reference part – models the references to program elements (for instance a reference to a type). It expresses the fact, that that program references elements that are not necessarily reified into the metamodel. References are used by metamodel elements to reference elements weakly. Weak references make it more flexible to construct and modify a program model.

4.4 Abstract Syntax Tree

An abstract syntax tree (AST) is a way of representing the syntax of a programming language as a hierarchical tree-like structure [14]. In comparison to Concrete Syntax Trees (another type of a syntax tree used in order to get an exact representation of the code), abstract syntax trees represent code at an abstract level disregarding unimportant elements such as grammar symbols.

4.4.1 Application

ASTs are used to represent the structure of the program code. It may serve as a representation of the program in the intermediate stages.

Properties of the AST:

- Does not contain inessential punctuation,
- Stores extra information to the program (e.g. position of elements),
- Can be edited, enhanced.

In the AST operators serve as nods and operands are leaves

4.5 Possible Improvements

The code has been constructed by means of modular programming. The main advantages that come with this approach are that the code has lower average complexity, makes it easier to be tested, and most importantly, allows for reusability and simplified adjustments.

In the future, we see that the program can be improved by expanding the metrics calculated, adding a graphical user interface, and using more than 100 projects as a base for calculating percentile value. Also the possibility of adding a config file to control which metrics are calculated, since unnecessary calculations or complications of analysis may affect time consumption.

Chapter 5

Testing

This chapter describes the process of testing and received results analysis. Testing was performed on 100 projects differing in size and characteristics. All tested projects were open-source and were acquired from GitHub, described in Chapter 5.3. The testing evaluation described in Chapter 5.2.

All the data from testing could be found in `statistics.ods` file.

5.1 Tested Projects

The main requirement for a project to be chosen for testing was relevance and relative public success, meaning it needed to have more than 100 stars on GitHub or more than 500 followers.

All the projects were acquired by the simple bash script with a `git clone` operation on the list of the needed project on the input. In the program described in the implementation Chapter 4, a hundred open source-projects were tested to build a base for future comparison.

5.2 Evaluation of Results

Overall results for almost all the projects are quite high, that is can be explained because those projects are popular and many developers working on those projects.

Based on the statistics acquired from the data base user can get a percentile for all key metrics, such as Cyclomatic Complexity, Density of Comments, and most important — Maintainability index. This percentile gives a qualitative evaluation for a project in comparison to the analyzed projects.

The program has an opportunity to adjust the test base for better project evaluation. This allows comparing either more projects or projects that have more similar characteristics. For getting the best results, the test base should be done on the set of similar by size and area projects.

Based on testing and analysis, reviewing metrics and percentile immediately shows possible areas of improvement. Percentile was defined for simplifying the output for the user, so he does not need to study the whole report. User can instantly see not only how good his or her project is, but to see what the main weakness of the project is. As an example, if the user gets percentile for Density of Comments around 70, percentile for MI around 80, and percentile for Cyclomatic Complexity around 40, the user can clearly see that the main focus should be on breaking the methods to smaller methods or simplifying the logic.

5.3 GitHub

GitHub is a web-based version control system developed and started by Chris Wanstrath, P. J. Hyett, Tom Preston-Werner, and Scott Chacon in 2008 using Ruby on Rails. GitHub is mainly used for computer code, supports source code management and version control functionality of Git while adding its own features[1]. After one year of being online, by February 2009, GitHub had accumulated 46000 repositories, and in November 2018 it had hit 100 million repositories with 31 million developers.

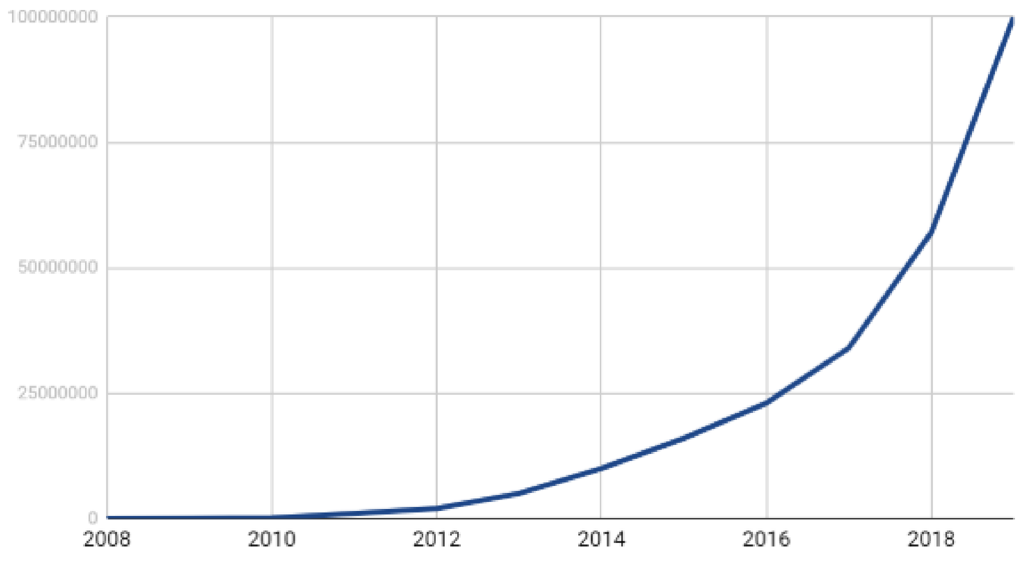

Figure 5.1: Github growth

Features

Projects on GitHub can be accessed using Git command-line interface supporting all standard Git commands. Multiple clients have been created by GitHub and third-party developers that integrate with the platform. There are two types of repositories: public and private. Being a registered or non-registered user defines what information can be accessed and changed.

In addition to code hosting, editing, and tracking, the site provides social networking like functionality for the ease of collaboration and identifying positive trends and changes.

Features defined by the GitHub site:

1. Formattable documentation (README files).

[1]http://github.com

2. Wikis.
3. Issue tracking supporting feature request, the fulfillment of milestones.
4. Pull requests.
5. Commits history.
6. GitHub pages and subscriptions.
7. Code hosting.

5.4 Limitations

During the testing process, there have been found 2 limitations related to the Java Spoon library:

1. Java Spoon could not create an abstract syntax tree if there is presented at least one `.java` file with syntax or semantic errors. It is recommended to build the project before testing.

   Error message: `spoon.compiler.ModelBuildingException: The type BaseIdent is already defined for Cgnnc project`

2. Sometimes Java Spoon has a problem with creating AST for larger projects with more than 50 000 Lines of Code. It has a problem with inner (nested) classes which are accessed from the different packages. Workaround for this issue would be to temporarily delete problem classes or to test the whole project by smaller parts.

   Error message: `spoon.SpoonException: Cannot compute access path to type: Cls$CustomException in context of type: Cls`

5.5 Summary

After testing all acquired projects with the program, statistics show that project size (LOC) and Density of Comments play a major role in code maintainability and as quality as a result. Average Cyclomatic Complexity has almost no impact on the final result for Maintainability Index, so should be used only for identifying methods that crossed the defined limit of CC on this project. Halstead Index gives us the information about the length of the program, and many theoretical metrics, such as Time to Implement, Number of Delivered Bugs, Effort, and Difficulty. The author could not identify any advantages or areas where Halstead metrics could be useful.

Chapter 6

Conclusion

Quantitative and qualitative software metrics play a major role in all software development stages. This is why one of the goals was to identify the key characteristics of good code, and thus the software metrics to control those characteristics. An important part of the thesis was to find the relation between qualitative and quantitative metrics and the possibility to evaluate the quality of the code based on quantitative data.

The main goal of the thesis was to implement a command-line application capable of evaluating the source code metrics. This goal was achieved successfully. The implemented application allows users to analyze java source code projects and receive the calculated metrics in the form of a report. For the program to function we have defined ways of calculating the metrics and implementing those calculations in the code. The key approaches to calculation were building the abstract syntax trees and the usage of regular expressions. The program developed had met the requirements of being adjustable and easy in usage since it was developed with a modular programming approach. It has been tested on a hundred of the open-source projects, thus creating a base for comparison.

The evaluation of the tested projects shows us that the most important metric for Maintainability Index is Lines of Code and Density of Comments. Cyclomatic Complexity plays a minor role in the calculation of Maintainability Index, so it's better to evaluate this metric independently from other metrics. Volume from Halstead Measures also plays a minor role in the calculation of Maintainability Index. Halstead Index as an independent metric does not bring any useful info, so should be used with a reserve.

Future work includes expanding the number of metrics used for identifying the software code quality. Also, a graphical user interface could be useful for some developers for having better visualization. Another improvement would be to use a wider base of the tested projects for future comparison.

Bibliography

[1] *Measurement of Halstead Metrics with Testwell CMT++ and CMTJava*. Accessed 2020-05-27.
Retrieved from: https://www.verifysoft.com/en_halstead_metrics.html

[2] Arafat, O.; Riehle, D.: *The Comment Density of Open Source Software Code.* Accessed 2020-07-03. 195-198 pp.. in Companion to Proceedings of the 31st International Conference on Software Engineering (ICSE 2009). IEEE Press, 2009.
Retrieved from: http://128.223.4.25/events/icse2009/images/postPosters/The%20Comment%20Density%20of%20Open%20Source%20Software%20Code.pdf

[3] Bobkowska, A.: *Quantitative and qualitative methods in process improvement and product quality assessment.* January 2001.

[4] Chhabra, P. S.: *ISO 9126 Quality Model.* January 2018. Accessed 2020-07-30.
Retrieved from:
http://thesuccessin5.blogspot.com/2018/01/iso-9126-quality-model.html

[5] Coen, J. B.; Vogt, H. H.: *How to save on software maintenance costs.* November 2014. Accessed 2020-06-4.
Retrieved from:
http://asq.org/public/wqm/how-to-save-on-software-maintenance-costs.pdf

[6] Coleman, D.; Ash, D.: *Using Metrics to Evaluate Software System Maintainability.* August 1994: pp. 44–49.

[7] Eick, S. G.; Graves, L.; Karr, A. F.: *Does Code Decay? Assessing theEvidence from Change Management Data.* January 2001. Accessed 2020-06-07.
Retrieved from: https://pdfs.semanticscholar.org/ecd9/43f52274c575f8018fb0fad8bd45b645e03d.pdf

[8] ISO: *ISO/IEC TR 9126-4:2004 Software engineering — Product quality — Part 4: Quality in use metrics.* 2004. Accessed 2020-01-28.
Retrieved from: https://www.iso.org/standard/39752.html

[9] ISO: *ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.* 2011. Accessed 2020-01-28.
Retrieved from:
https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[10] Kolawa, A.; Huizinga, D.: *Automated Defect Prevention: Best Practices in Software Management.* 2007. ISBN ISBN 978-0-470-04212-0. 75 pp.

[11] Koskinen, O. J.: *Software Maintenance Costs*. Accessed 2020-01-26.
Retrieved from: http://users.jyu.fi/~koskinen/smcosts.htm

[12] Matt, B.: *Signs Your Software is Rotting*. June 2020. Accessed 2020-07-13.
Retrieved from:
https://codurance.com/2020/06/09/signs-your-software-is-rotting/

[13] McCormack, J.; Conway, D.: *CSE2305 - Object-Oriented Software Engineering*.
Accessed 2020-05-14.
Retrieved from:
http://users.monash.edu/~jonmc/CSE2305/Topics/13.25.SWEng4/html/text.html

[14] Newcomb, P.: *Abstract Syntax Tree Metamodel Standard*. 2005. Accessed 2020-02-21.
Retrieved from: https://www.omg.org/news/meetings/workshops/ADM_2005_Proceedings_FINAL/T-3_Newcomb.pdf

[15] Nguyen, V.; Deeds-Rubina, S.; Tan, T.; et al.: *A SLOC Counting Standard*. 2007.
Accessed 2020-02-16.
Retrieved from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.550.8181&rep=rep1&type=pdf

[16] Pawlak, R.; Monperrus, M.; Petitprez, N.; et al.: *Spoon: A Library for Implementing Analyses and Transformations of Java Source Code*. September 2015. Accessed 2019-12-20.
Retrieved from: https://hal.inria.fr/hal-01078532/document

[17] R. Subramanyam and M. S. Krishnan: *Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects*. April 2003: pp. 297–310.

[18] Ward, C.: *Debt Metaphor*. February 2009. Accessed 2020-05-22.
Retrieved from: https://www.youtube.com/watch?v=pqeJFYwnkjE

[19] Watson, A. H.; McCabe, T. J.: *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. 1996. Accessed 2020-04-20.
Retrieved from: https://csse.usc.edu/TECHRPTS/2007/usc-csse-2007-737/usc-csse-2007-737.pdf

[20] Welker, K. D.: *The Software Maintainability Index Revisited*. August 2001, Accessed 2020-02-17.
Retrieved from: http://web.archive.org/web/20021120101304/http://www.stsc.hill.af.mil/crosstalk/2001/08/welker.html

[21] Yu, P.; Systa, T.; Muller, H.: *Predicting faultproneness using OO metrics. An industrial case study*. 2002: pp. 99–107.

Appendix A

Contents of the Attached CD

Following directories and files can be found on the CD:

- Directory `src` – a directory containing source code files and Java Spoon `.jar`
- File `README.md` – a file describing the project and instructions of running the project.
- File `tested_projects.txt` – a file containing list of the tested projects.
- File `download_github_projects.sh` – a script to download all the tested projects from GitHub.
- Directory `latex` – a directory containing LaTeXsource files.
- File `output_metrics.ods` – a file containing statistics from the testing.
- File xshers00.pdf – an electronic version of this thesis.

Appendix B

Example of the output `statistics.txt`

Statistics for the whole project:
Number of classes: 11
Number of all classes (including all nested classes): 15
Total number of methods: 56 (getters, setters and overwritten methods from `Object` class are excluded)
Logical lines of code: 924
Average cyclomatic complexity: 2.16
Maintainability index: 45.75
Density of comments: 8.55%
Halstead index:
Volume: 15492.27
Difficulty: 697.69
Effort: 401406.89
Program length: 3072
Time to implement: 22300.38 s
Number of delivered bugs: 4.94

Percentiles of the key metrics:
Cyclomatic complexity: 49
Maintainability Index: 10
Density of comment: 32

Classes:
`com.github.gcacace.signaturepad.MainActivity:`
Number of methods: 8
Logical lines of code: 133
Commented lines of code: 8
Density of comments: 6.02%
Average cyclomatic complexity: 2.12
Maintainability index: 61.08
Halstead index:
Volume: 2834.46

Difficulty: 117.36
Effort: 52982.44
Program length: 546
Time to implement: 2943.47 s
Number of delivered bugs: 0.82
methods:
`com.signaturepad.MainActivity.addJpgSignatureToGallery(Bitmap signature):`
Cyclomatic Complexity: 2
Halstead index:
Volume: 256.76
Difficulty: 13.6
Effort: 3491.99
Vocabulary: 27
Program Length: 54
Time to implement: 194 s
Number of delivered bugs: 0.08
Program level: 0.07
...