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 2019

Abstract

In order to control and improve code quality there needs to be a system in place consisting of quantitative metrics and their analysis. With software quality metrics that best 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 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

Aby bylo možné kontrolovat a zlepšovat kvalitu kódu, musí být zaveden systém, který se skládá z kvantitavních metrik a jejich analýzy. S použitím metrik kvality softwaru, které nejlépe 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 mnohé účely.

Daný výzkum prezentuje program vyvinutý na základě identifikovaných vlastností kódu a souvisejících metrik, metod používaných při tvorbě programu a výsledků testů založených na open-source projektech.

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, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Zbyněk Křivka, Ph.D.

Rozšířený abstrakt

Aby bylo možné kontrolovat a zlepšovat kvalitu kódu, musí být zaveden systém, který se skládá z kvantitavních metrik a jejich analýzy. Pro vybudování takového systému, byly stanovené cíle, pomocí kterých je možné identifikovat metriky zdrojového kódu, které co nejlépe vystihují, jak dobře je zdrojový kód napsán, který nesplňuje požadovaná kritéria a vytvořit program schopný vyhodnotit kvalitu zdrojového kódu, který je založen na stanovených kritériích.

Existuje hodně řešení s podobnou funkčností, nicméně v této práci s Java Spoon budou porovnány metody, které jsou vybrané autorem. Klíčový rozdíl je v tom, že Java Spoon je používán pro analýzu, transformaci a generování kódu, avšak tento projekt bude použit výhradně pro analýzu zdrojového kódu, aby se vyhnulo komplikacím a udržovala se jednoduché použití.

V práci byly identifikované 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 index Halstedu (Halstead Index).

S pomocí abstraktních syntaktických stromů a regulárních výrazů daný program vypočítává výsledky pro každou metodu, udává průměrné hodnoty pro třídy a také pro celý projekt. Poté výsledky jsou porovnávány mezi určenými programy pro analýzu za účelem zjištění percentilu metrik.

Hlavními požadavky programu snadné použití, snadná rozšiřitelnost kódu a pochopení výsledků, a nakonec snadné nastavení.

Je zapotřebí prostudovat vybranou knihovnu pro analýzu zdrojového kódu a také prostudovat možnost přístupu k několika open source projektům, které jsou implementované v programovacím jazyce Java s využitím existujících nástrojů a rozhraní API (např. 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.

Kromě toho, byl připraven soubor zahrnující sto testovacích projektů v Javě pro ohodnocení jejich kvality.

Dále 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. Poté byly ohodnoceny a shrnuty výsledky včetně návrhu některých možných budoucích vylepšení.

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 Křívka, Ph.D. The supplementary information was provided by Ing. Tišnovský Pavel, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Vladyslav Sherstobitov
May 16, 2019

Acknowledgements

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

Contents

Chapter 1

Introduction

The goal of the research is to develop a program capable of distinguishing good quality source code of Java programs based on the objective identifiers. The task is for this program not only to provide a readable output, but for it to be easy to use.

Prior to describing the program and its test outcomes, the qualities that describe good code are identified. Based on the qualities, the five following source code metrics are chosen and described: Lines of Code, Density of Comments, Cyclomatic Complexity, Maintainability Index, Halstead Measures.

Further this project is compared with Java Spoon, and then it is explained how with the help of Abstract Syntax Trees and regular expressions, the code calculates results for each method, gives average values for classes, and then the outcomes are compared between programs for analysis in order to identify the outliers.

Chapter 2

Software metrics

Quality metrics are an important part of the effective quality management in every process and plan. Simply put, code quality metric is a measure that allows to translate 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 [2].

Reasons to use quality metrics:

1. With metrics measurement implemented and targets set there is no need to wait till the end of the development to see the results, metrics can be tracked during the whole cycle showing weaknesses and dangers.

2. Productivity depends on time spent on tasks and on which tasks in particular. In order to track productivity and make improvements, metrics are needed to prioritise the tasks for the team.

3. Metrics as well can be used as means of communications. Keeping metrics in target will mean that there are no issues with the project, if there is a deviation from targets, faulty modules can be identified and an appropriate team contacted, and overall progression can be reported to the management by using established metrics improving awareness and workflow.

In order to improve and control software quality it is necessary to first identify what is needed from the code:

1. **Efficiency**. This characteristic is directly related to the performance of the running software, and is highly important to the consumer. Efficient code will use minimum resources and run fast.

2. **Reliability**. The second key component to quality for the consumer would be reliability of the software. In this case by reliability we mean an ability to perform consistently and without errors every time the code runs.

3. **Maintainability**. Having a highly maintainable code would mean to future-proof it. There are many instances when changes are needed: to fix bugs, to modify existing features, or to implement something new. Software that is easy to understand will make it easy to find needed parts, make changes and make sure no new bugs have been introduced, sequentially meaning that it makes it easier and takes less time to maintain.

4. **Readability**. There will always be circumstances when someone new has to look at the code, so adding an ability to understand the code easier, quicker, and clearer will ensure that everyone will be able to figure out what part is responsible for what, where changes need to be made, and that no part will have to be rewritten.

5. **Code Reuse**. As a result, if four goals are kept in check, the best outcome would be a code reuse. The definition is, that existing software, or software knowledge, can be used in order to build the new software increasing productivity, decreasing error number, and improving software quality overall.

As a result, if four goals are kept in check, the best outcome would be a code reuse. The definition is, that existing software, or software knowledge, can be used in order to build new software increasing productivity, decreasing error number, and improving software quality overall. And per characteristics, the following metrics have been chosen (as can be seen on the graph below): Cyclomatic Complexity, Halstead complexity measures, Density of Comments, and Maintainability index.

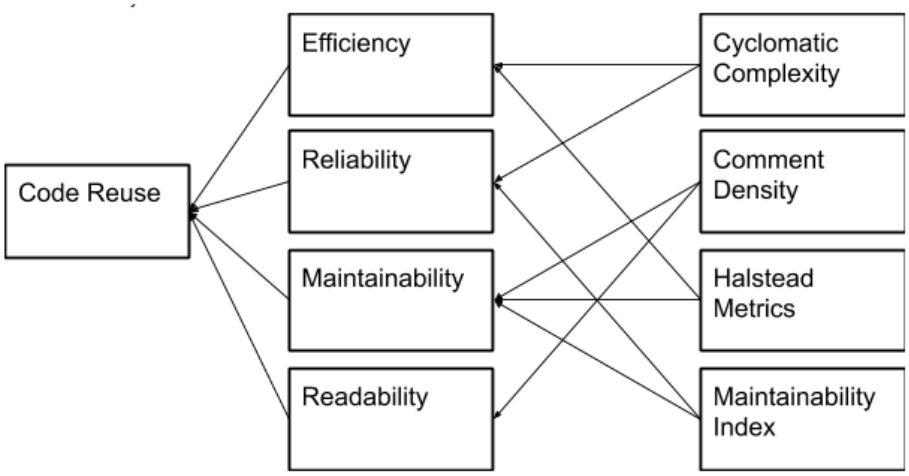

Figure 2.1: Relations between code characteristics and quality metrics.

2.1 Lines of code

Lines of code is one of the most traditional metrics to quantify complexity of software, as well as it is one of the simplest to understand and count [7]. Not only lines of code is a useful metric itself, but a vital component in calculation of many others. There are four types of LOC:

1. **LOCphy** - sum of physical lines of code.

2. **LOCbl** - sum of blank lines.

3. **LOCpro** - sum of program lines.

4. **LOCcom** - 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 and logical are two major measures of lines of code metric. Physical lines of code are much easier to calculate, but are sensitive to formatting style of the developer. Logical lines of code while being harder to calculate are independent from style and formatting, and are more objective in code length and complexity.

```
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 LOCphy and 2 LLOC.

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

This example consists of 3 LOCphy and 2 LLOC.

2.2 Cyclomatic complexity

Cyclomatic complexity is a metric that was developed by Thomas J. McCabe, Sr. in 1976. In the simplest way it shows the number of linearly independent paths in the section of code [4], 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.

2.2.1 Calculation

The first way to calculate McCabe cyclomatic complexity (MCC) is to use a formula

$$MCC = E - N + 2P$$

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:

$$MCC = E - N + 2$$

The second way would be to use a control flow graph of a module, where the program begins executing at the green nod, then going to a conditional statement with one condition, then going to a conditional statement with two conditions, and then exiting at the red nod.

In order to calculate MCC from a graph, we need to count the number of closed figures in a graph, in addition to the outer figure marked by an orange arrow, giving an MCC of 4 for the example above.

And using the formula mentioned previously:

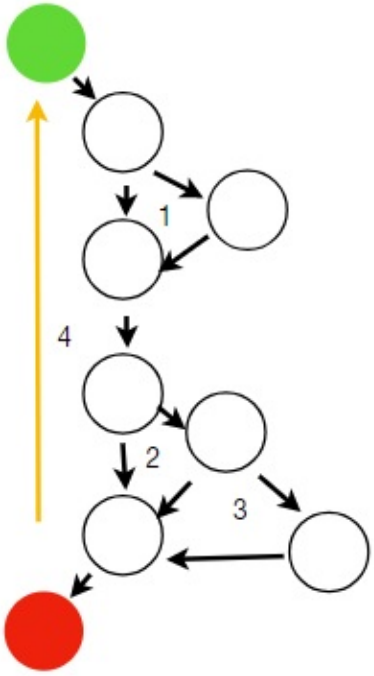

Figure 2.2: Cyclomatic complexity example.

$MCC = 11 - 9 + 2$
$MCC = 4$

2.2.2 Application

Identify and limit the complexity of the code

It was one of the original applications of the metric by McCabe to limit 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. With usage and development, it has been determined, that complexity rating of 10 is not always optimal and complexity of up to 15 has been permitted in particular cases, however, in order to keep the structure and readability, it was recommended to comment on the reason why the limit was exceeded.

Prevent defects

There are multiple researches proving there is correlation between size of 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, number of failures will be minimized, but by controlling cyclomatic complexity and limiting code complexity as follows, should improve code and comment quality overall.

Determine the number of test cases required

Another way to use MCC would be to determine the number of test cases required to achieve thorough test coverage for a module.

The value of MCC gives two pieces of information:

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: number of branches =< MCC =< number of paths.

Importance of CC in unit testing

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

One of the biggest advantages of unit testing is its ability to be implemented in the early stages of development, as a result improving structure of the code, functionality, and identifying errors in the first cycles of development.

In order to make sure that software is well functioning, tests should cover as many branches as possible. The more branches are covered, less 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 minimum.

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

2.2.3 Implementation

In this project, CC is calculated with the help of Abstract Syntax Tree up until methods, and after, a number of keywords is calculated according to predefined regular expression table.

The easiest way to calculate the cyclomatic complexity is to count a number of occurence of those symbols: *if, while, for, case, catch, &&, //, ?.*

2.3 Halstead complexity

Halstead metrics have been introduced in 1977 by Maurice Howard Halstead. 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.

2.3.1 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

$$N = N_1 + N_2$$

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.

$$n = n_1 + n - 2 - \text{ vocabulary size}(n_1 + n_2)$$

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.

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

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

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

In order to calculate 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.

$$E = D * V$$

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

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

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.

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

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

$$V = N * \log_2 n$$

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

2.3.2 Application

The Halstead metrics are applicable for the written code, however, one of the best uses would be during the development in order 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. So, as the size of the program grow, so does 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 best results as a strong component to Maintainability Index.

2.3.3 Implementation

Same as with CC, Halstead metrics are calculated with the help of Abstract Syntax tree up until methods, and for the methods regular expressions are used for calculation.

2.4 Density of comments

Density of comments is a metric representing the percentage of comment lines in a given part of the source code, that is calculated by dividing a number of commented lines by a total number of lines of code. 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 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, however, 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 explanation of the code, it would be considered more of 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 maintainer 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, licenses information, etc.

6. **Debugging.** It is considered a 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, 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 to for an interpreter or editor.

9. **Stress relief.** On some occasions programmers will add comments to relieve stress by commenting about working conditions, tools used, or the quality of the code itself.

Comment recommendations

With 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 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). 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 some researches, since lack of comments may hide logic and abundance of unclear comments will have a negative effect too.

2.4.1 Application

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

2.5 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. 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 [3]. In order to increase MI, Cyclomatic Complexity should be reduced, as well as the Volume (V), and a total number of lines of code. Overall this means that MI keeps all the qualities of metrics in it and serves as a good and simple summary of overall state of the code.

2.5.1 Implementation

MI is calculated with 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.

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

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

Technical debt is a concept, that 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. If this debt is overlooked, wit every subsequent release, it will lead to code becoming unmaintainable, and additional resources will need to be used in order to rewrite whole parts.

Reasons for technical debt

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

Consequences of technical debt

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, what is important, to address the shortcut on time. In case even few known things are not changed on time, it would be very hard to estimate how much resources will be required

later for a fix, and due to this it may become a major reason for missing the next deadline. From this we come to conclusion, that the amount of technical debt needs to be in check.

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

<0 - extremely poor maintainability

0-65 - poor maintainability

65-85 - moderate maintainability

85< - good maintainability

Trending MI over a period of time will help identify the rate of software degradation, and in this case with the help of the index it is possible to track back to the initial major release, when trend became negative. Calculating the differences between point releases will help locating the reason, and possibly specific module that needs to be addressed. Based on this information tasks can be prioritized in order to prevent further degradation.

The best way to manage technical debt is to create a set of proactive measures. Simple changes can be implemented on regular bases to save a lot of effort in a long term:

1. Check for two types of modules: ones with low MI, and ones with trending degradation.

2. 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 responsible team with data to further isolate and resolve the issue.

3. In case data is inconclusive, cross checks should be done focusing on metrics such as Cyclomatic Complexity.

4. 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.

5. After any optimization changes are performed, tests should be ran again to make sure MI has improved.

2.5.2 Application

Since all variables have been defined previously, the only thing left is to calculate the index with the help of formula.

2.6 Suggested targets for metrics

LOCpro

- per function min - 4, max - 40
- per file min - 4, max - 400

Cyclomatic Complexity

1 - 10	Structured and well written code that is easy to test.
10 - 20	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	Complex code that is hard to test. This code should be refactored, broke down into smaller methods, or using some design patterns.
>35	Very complex code, that is not at all testable and almost impossible to maintain or extend. Should be fully refactored or rewritten.

Density of comments

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

Maintainability Index

<0	extremely poor maintainability
0 - 65	poor maintainability
65 - 85	moderate maintainability
85<	good maintainability

Meaning for minimum and maximum targets

Maintainability index

Maintainability above 85

Code as a whole can be considered maintainable and reliable.

Maintainability 65-85

Majority of the code meats the requirements, however, there is either a part of code that may be edited for improvement, or it contains a reasonable explanation.

Maintainability below 65

Most likely there are major issues with code complexity in combination with poor commenting, and as a result will not be re-usable.

Cyclomatic Complexity

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

Density of comments

There are two negative outcomes for Comment Density values:

- Density of Comments being above 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.
- If Density of Comments is below target, it would mean that compex 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.

Conclusion

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

1. They can measure performance, lead the team in correct direction, and track progress
2. Based on results management can implement actions and assign them correct priority based on deviations of metrics in progression
3. Metrics are not bounds and are here not to limit developers, there are instances where correct decision would be not to follow the most rational way according to raw numbers

Chapter 3

Implementation

3.1 Java Spoon library

Java Spoon enables Java developers to perform three operations [6] with the source code in a simple manner:

- Analyze.
- Transform.
- Generate.

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

The main feature of 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 (The way Spoon functions)

When a Java program is given as an input, it is parsed with a compiler in order 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 templating engine takes these transformation as an input and applies them to the Java model. At the end, the Spoon model is translated back to source with a pretty printer.

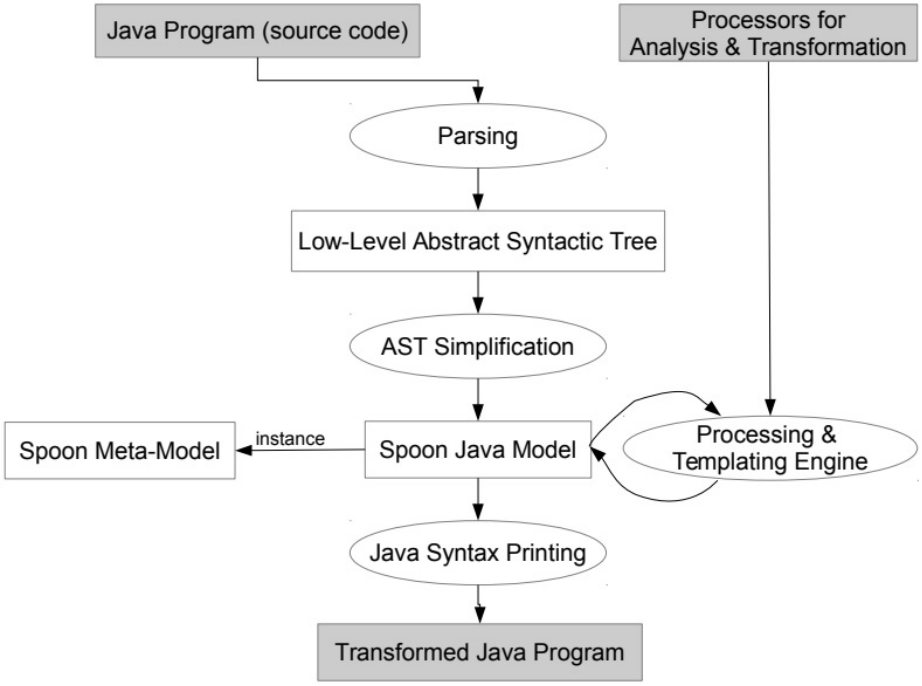

Figure 3.1: Java Spoon metamodel.

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 with a goal to be easily understandable by a normal java developer. The Spoon metamodel allows for creation of 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".
- 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 in a weak way. Weak references make it more flexible to construct and modify a program model.

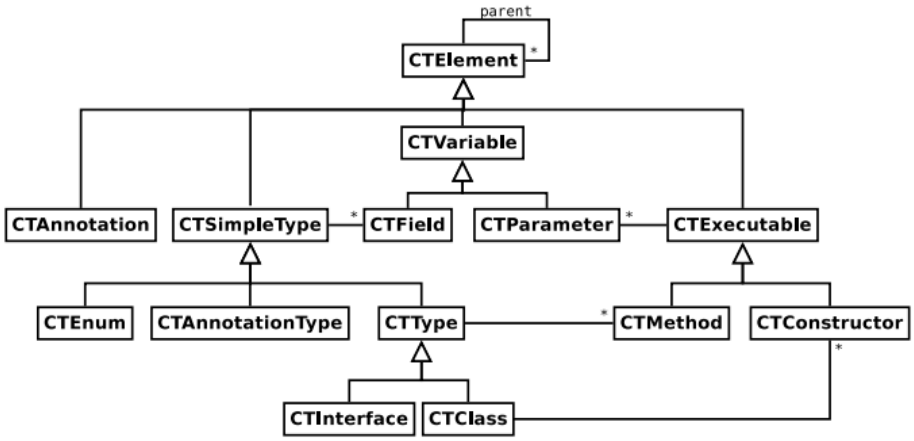

Figure 3.2: Structure of the element.

3.2 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 [5]. 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.

3.2.1 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

3.3 Possible improvements

The code has been constructed by the means of modular programming. Main advantages that come with this approach are that 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 metrics calculated, adding a Graphical User Interface, using more then 100 project as a base for calculating percentile value.

Chapter 4

Testing

A hundred of projects was tested using the five main characteristics which were predefined before testing. Afterwards, the percentile based on each of these characteristics was calculated. Thus the testing was carried out with the using of these features and had defined the percentile.

4.1 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.

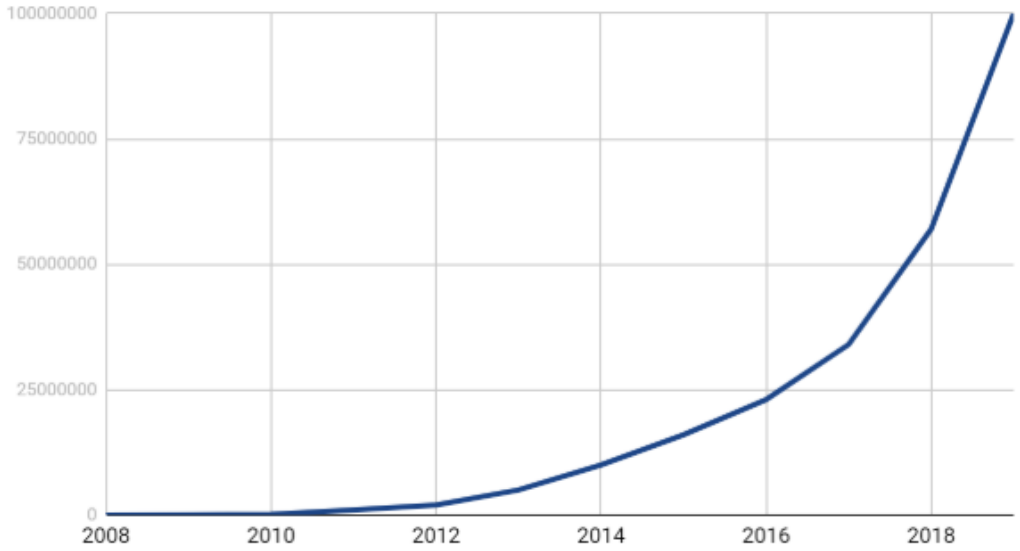

Figure 4.1: Github growth

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

4.2 Tested projects

A hundred of projects was tested using the five main characteristics which were predefined before testing. Afterwards, the percentile based on each of these characteristics was calculated. Thus the testing was carried out with the using of these features and had defined the percentile.

Chapter 5

Conclusion

Quantitative software metrics play a major role in all software development stages. This is why the goal was to identify the key characteristics of a good code, and thus the software metrics to control those characteristics, and to build a program capable of evaluating the source code based on the metrics.

In order for the program to function we have defined ways of calculating the metrics and to implement those calculations in the code. The key approaches to calculation were building the Abstract Syntax Trees and a usage of the 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.

Bibliography

[1] GitHub.
Retrieved from: https://github.com/

[2] Al-Qutaish, R.: Quality Models in Software Engineering Literature: An Analytical and Comparative Study. *Journal of American Science*. vol. 6. 11 2010: pp. 166–175.

[3] Coleman, D.; Ash, D.: Using Metrics to Evaluate Software System Maintainability. 8 1994: pp. 44–49.

[4] McCabe, A. H. W. J.: Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.
Retrieved from: https://csse.usc.edu/TECHRPTS/2007/usc-csse-2007-737/usc-csse-2007-737.pdf

[5] Newcomb, P.: Abstract Syntax Tree Metamodel Standard.
Retrieved from: https://www.omg.org/news/meetings/workshops/ADM_2005_Proceedings_FINAL/T-3_Newcomb.pdf

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

[7] Vu Nguyen, T. T., Sophia Deeds-Rubin; Boehm, B.: A SLOC Counting Standard.
Retrieved from: https://csse.usc.edu/TECHRPTS/2007/usc-csse-2007-737/usc-csse-2007-737.pdf

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2018/2019

Bachelor's Thesis Specification

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 http://www.fit.vutbr.cz/info/szz/

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, 2018
Submission deadline: May 15, 2019
Approval date: May 9, 2019