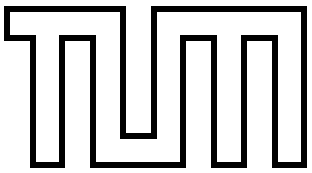


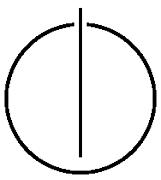
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Seminararbeit

Maintainability of Java Methods: A Quantitative Approach

Andreas Heckl



Abstract

Software maintenance accounts for a large part of the total cost of a system. Many tools exist to help ensure maintainability by identifying quality defects. However, only few focus on metrics in their presentation and provide information at method level. Since those few tools are not as helpful to a software maintainer as state-of-the-art static analysis tools, we aim at closing the gap by providing a tool that assists in prioritizing the methods of a system from a maintainability perspective. We achieve this by highlighting values, detecting multivariate outliers and including the number of quality defects found by state-of-the-art static analysis tools.

Contents

List of Figures

1 Introduction

The concept of software quality goes back many decades. Barry Boehm began working on this as early as the 1970s [BBL76]. Since then many standards and norms have been created, some parallel to each other, others replacing the former, like the ISO/IEC 25010 [ISO11] standard, which replaced the ISO/IEC 9126 standard [(IS01]. What all these models have in common is that they divide software quality into several criteria, but remain abstract in the description of these criteria. For example, the ISO/IEC 25010 standard includes the following eight characteristics in its software product quality model:

- Functional Suitability
- Performance Efficiency
- Compatibility
- Usability
- Reliability
- Security
- Maintainability
- Portability

Although in this standard each of the characteristics listed is subdivided again, these descriptions also remain abstract, e.g. Maintainability is subdivided into Modularity, Reusability, Analyzability, Modifiability and Testability. However, the standard does not tell us how we can measure the fulfillment of these criteria in practice. In our opinion, one possible reason for this is the fact that software is used in many different application domains, which require different emphases and can have special characteristics. For example, medical software products are inherently different from database systems, therefore it is not possible to say what reliability concretely means for both. Thus, we do not expect future software standards to be so concrete that they tell us how we can measure their quality characteristics in practice. Therefore we chose to investigate how an abstract concept in a standard, namely maintainability, can be measured in practice. For this thesis, maintainability as the quality aspect of our interest was chosen mainly for two reasons: Fist, we think that it holds true for most software systems that they have to be understood, changed or improved over time and therefore in our opinion maintainability is among the quality aspects that are important for the vast majority of systems. Second, it has been shown that the cost of maintaining a software system represents a significant portion of the total costs that arise during a system's life cycle [Pig96],[BBL76] and even increases over time [CM78].

Specifically, we will focus on the maintainability of Java methods. We chose method level analysis because, as we will see in Chapter 2, there are already a large number of tools for analysis and evaluation at other levels, such as class, package or system level, but this choice is massively limited as soon as one becomes interested in evaluation at method level.

Goal In the course of research only a few tools were found, which provide method level metrics for Java systems. In Chapter 2.2 we will see that these tools do not tell us which method should be inspected with high priority from a maintenance perspective. Hence we introduce a tool that assists a software maintainer by finding out which methods to prioritize.

In the course of this thesis we will approach the question of how to assess the maintainability of a Java method from two directions: On the one hand, we will look at existing tools and examine their strengths and limitations (Chapter 2). On the other hand, we will discuss what would be useful to measure in terms of evaluating the maintainability of a method, regardless of whether tools for these measurements already exist or not and name some requirements an ideal hypothetical tool should fulfill (Chapter 3). In this context, in Chapter 4 we will also present our self developed tool that aims at extending the functionalities of the existing method level tools towards the hypothetical tool from Chapter 3.

2 Existing Static Analysis Tools

Static analysis, in contrast to dynamic analysis, does not require the code to be executed. In our study, we chose to focus on static analysis tools for two reasons: First, in our opinion they are easier to use since we do not need to think of possible inputs. Second, research shows that these tools seem to be underused even though they assist developers in finding defects [JSMHB13].

During our research we have found that a variety of static analysis tools for the Java language exist. Since we are particularly interested in approaching maintainability of Java methods quantitatively, we divide the inspected tools into two groups: Those that do provide metrics on a method level and those that do not. From now an we will call the first group *metrics-oriented* and the second one *findings-oriented*, where we refer to a *finding* as a potential instance of a quality defect that was found by the respective tool.

In these tools, findings have certain properties, such as an associated category or the exact location in the code that is affected. For instance, an analysis tool could list a finding if it finds a variable declaration where the name of the variable starts with an uppercase letter. The associated category could then be *naming* and the location would be the package, class name and line in which this declaration occurred.

In our study, most analyzed tools are findings-oriented. Therefore, we will first dive into these to understand how the process of software maintenance appears be approached most widely in practice. We will then discuss the metrics-oriented tools and work out the differences.

2.1 Findings-Oriented Static Analysis Tools

Examples of these tools are Teamscale[1], Squore[2] and Sonarqube[3]. All these tools offer a modern web interface with many different views of the system under investigation, such as dashboards, lists of findings or metrics. We refer to these views as *perspectives* in the rest of this thesis and will now go briefly through some of those that we found similar among multiple tools we investigated.

Dashboard Perspective Figure 2.1 shows an example of a dashboard in Teamscale. In the left half of the figure we see the four metrics clone coverage, lines of code, method length and nesting depth displayed. Let us look into the method length metric in the bottom left corner of the figure. Teamscale refers to this as an *assessment metric*, which means it does not only show the raw numbers but also indicates a quality status. This is achieved by coloring the pie chart into green, yellow and red where a green assessment means a better quality status than yellow and yellow means better than red. In this particular case, the green portion indicates that 54% of the code is part of methods whose length is considered to be all right, whereas the red portion indicates that around 24% of the code is part of methods that are considered to be to long. The thresholds for the coloring can be customized by the user. However, this assessment metric can only be obtained down to the class level. It is not possible to gain information about a single method. This yields for all metrics in Teamscale.

In the right half of Figure 2.1 we can see a bar chart that shows how many findings were found in the respective categories. In this example, Teamscale categorized its findings into *code anomalies, code duplication, documentation, naming* and *structure*. This leads us to the next perspective, which is the findings perspective.

Findings perspective This perspective shows a list of findings with their attributes such as a message, a location and a type. It also allows filtering, e.g. by the categories we saw above (Figure 2.1) in the bar chart. A click on an entry in the list shows the respective piece of source code.

1 https://www.cqse.eu/en/teamscale/overview/
2 https://www.vector.com/de/de/produkte/produkte-a-z/software/squore/#c147445
3 https://www.sonarqube.org/

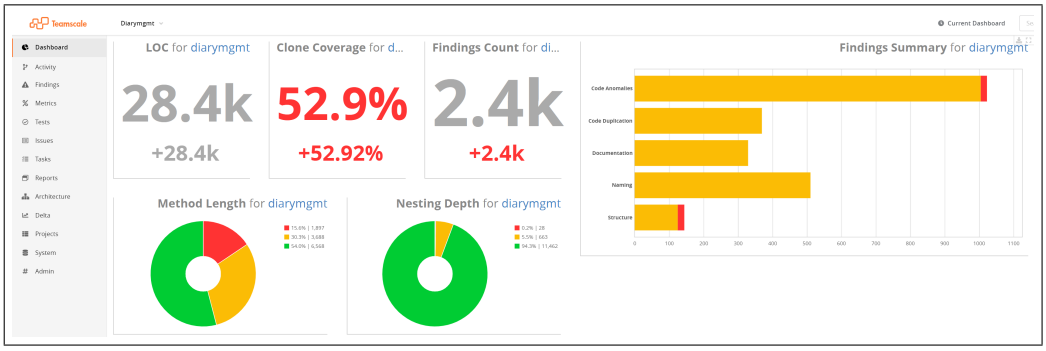


Figure 2.1: Extract from a dashboard in Teamscale

We assume that in practice this is the perspective where most of the work is done when one is assigned with the task of maintaining the software, since it allows one to filter those classes or packages that are in your own area of responsibility and remove the quality defects that were found or mark them as a false positive. One can then continue with the next list entry and therefore step by step improve the quality of the system by fixing the defects.

Measures Perspective The last perspective we want to talk about here is the measures perspective. This is also a concept that we came across on multiple static analysis tools. We want to show this with the example of Sonarqube. Figure 2.2 shows the subcategory maintainability within the measures perspective in Sonarqube. It provides information about the maintainability in several dimensions:

- Data points represent Java classes.
- The x and y-axis correspond to lines of code and technical debt, respectively. Technical debt in this context is the estimated time effort to fix the quality defects found in the respective class
- The size of a data point corresponds to the number of code smells in the respective class. In Sonarqube, a code smell is a specific category of finding, among others such as security vulnerability and bug.
- The color of a data point indicates a quality assessment, similar to that we have discussed for Teamscale's method length assessment. In this case, the colors stand for maintainability ratings from A (intensive green) to E (intensive red).

This form of visualization allows one to easily identify outlier classes, e.g. in Figure 2.2 the three yellow to red data points at the top left area have only few code smells and lines of code, yet they have a bad maintainability rating and a high technical debt. Just like in Teamscale, the lowest level Sonarqube allows us in this perspective is in case of Java systems the class level.

Let us now discuss the strengths and limitations of findings-oriented tools.

Strengths We found that the findings-oriented tools provide a good overview of the quality status of the system through their dashboards and also allow for the efficient correction of defects in the code, as they provide lists of findings that show a software maintainer where in the code a potential defect lies. Thus, in our opinion they are well suited for both managers and maintainers. Multiple filtering options such as filtering by paths or categories of findings allow a structured approach to maintenance. Special functions such as the test gap analysis in Teamscale also facilitate efficient maintenance of the system.

Limitations These tools do not provide us with answers to the question of the quantitative evaluation of individual Java methods. The provided metrics go down to the class level and are on the one hand not comprehensive, on the other hand they are hardly evaluated qualitatively.

In the following section, some tools are presented that provide metrics on the method level.

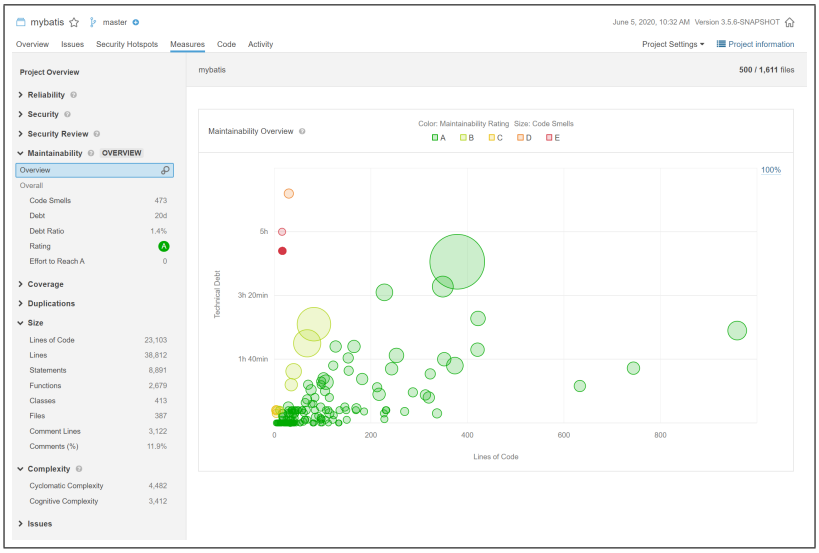


Figure 2.2: Measures Perspective in Sonarqube

2.2 Metrics-Oriented Static Analysis Tools

In the course of the research for this thesis, we found five tools that provide metrics on method level for the Java language:

- JavaMetrics by *Semantic Designs*[4]
- Understand by *scitools*[5]
- JHawk by *Virtual Machinery*[6]
- Sourcemeter by *Frontendart*[7]
- Klocwork by *RogueWave*[8]

Let us now briefly go through each.

JavaMetrics tool by Semantic Designs This tool is only available for Java 1.5 and Windows 7. On an E-mail request, whether also Windows 10 and newer versions of Java would be supported, we got no answer. This leads us to the conclusion that it is currently not maintained. However, on the homepage we found an example output[9]. A section of this can be viewed in Figure 2.3. The output format is an XML file in which the metrics that belong to a method are written in separate XML tags each. A total of 19 metrics is calculated per method. However, from the homepage we learn that this tool only outputs the data and does not have any mechanism for quality assessment or prioritizing methods for maintenance.

Sourcemeter by Frontendart First of all we want to state that we encountered a problem with the download: Attempting a download resulted in an http error. We observed that this problem lasted for several days but now has been resolved.
Sourcemeter can be run from the command line and produces output files on different levels such as class- or package level. The relevant output file for us is a csv file on method level. The rows of this file correspond to the methods and the columns to the metrics. A complete reference of metrics can be found in the documentation[10]. We notice that this tool also only provides raw data. However, Sourcemeter is also available as a plugin for Sonarqube. This plugin offers assessment of metrics by coloring the respective values. An example can be seen in Figure 2.4. Again, the rows correspond to the methods and the columns to the metrics. A positive assessment is indicated by

4 http://www.semdesigns.com/Products/Metrics/JavaMetrics.html
5 https://scitools.com/static-analysis-tool/
6 http://www.virtualmachinery.com/jhawkprod.htm
7 https://www.sourcemeter.com/
8 https://www.perforce.com/products/klocwork
9 http://www.semdesigns.com/Products/Metrics/JavaMetrics.xml
10 https://www.sourcemeter.com/resources/java/

```
    <SEIMaintainabilityIndex>125.91</SEIMaintainabilityIndex>
  </MethodMetrics>
▼ <MethodMetrics>
    <MethodName>addAccessibleSelection</MethodName>
    <LineNumber>3206</LineNumber>
    <JavaCodeLines>10</JavaCodeLines>
    <CommentLines>9</CommentLines>
    <CyclomaticComplexity>3</CyclomaticComplexity>
    <Conditionals>2</Conditionals>
    <DecisionDensity>0.30</DecisionDensity>
    <MaxLoopDepth>0</MaxLoopDepth>
    <MaxConditionalDepth>2</MaxConditionalDepth>
    <LineNumber>3208</LineNumber>
    <HalsteadUniqueOperators>14</HalsteadUniqueOperators>
    <HalsteadUniqueOperands>14</HalsteadUniqueOperands>
    <HalsteadOperatorOccurrence>33</HalsteadOperatorOccurrence>
    <HalsteadOperandOccurrence>22</HalsteadOperandOccurrence>
    <HalsteadProgramLength>55</HalsteadProgramLength>
    <HalsteadProgramVocabulary>28</HalsteadProgramVocabulary>
    <HalsteadProgramVolume>264.40</HalsteadProgramVolume>
    <HalsteadProgramDifficulty>11.00</HalsteadProgramDifficulty>
    <HalsteadProgramEffort>2908.45</HalsteadProgramEffort>
    <HalsteadBugPrediction>0.09</HalsteadBugPrediction>
    <SEIMaintainabilityIndex>153.28</SEIMaintainabilityIndex>
  </MethodMetrics>
▼ <MethodMetrics>
    <MethodName>clearAccessibleSelection</MethodName>
    <LineNumber>3239</LineNumber>
```

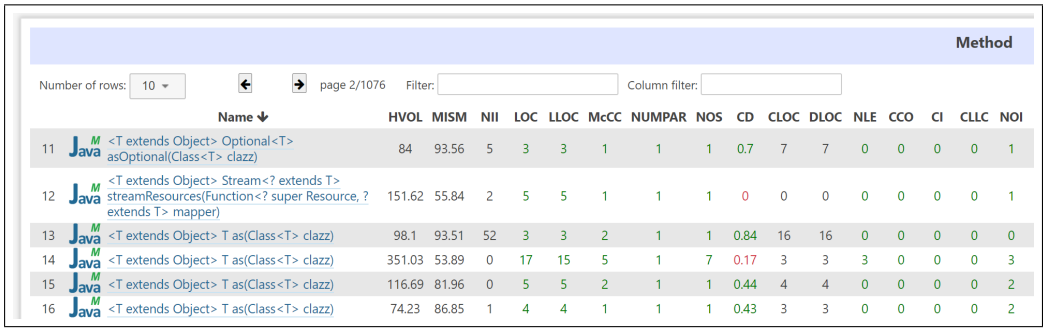
Figure 2.3: Sample output of the JavaMetrics tool by Semantic Designs

a green value, a negative by a red value. For example, the method in row number 14 in Figure 2.4 has a value of 5 for the metric *McCC (Cyclocmatic Complexity*, see Chapter 2.3 for discussion). The green color of this value indicates a positive assessment. In contrast to this, the value 0.17 for the metric *CD (Comment Density: ratio of the comment lines of method to the total lines of the method)* for the same method has a negative assessment which is indicated by the red color. However not every metric has a quality assessment: If the values of a column are black, no quality assessment is made. One practical feature is that a click on the classname in the first column opens up a new window which shows the sourcecode of the method. We find this plugin useful, yet we observe some limitations:

1. The plugin only uses a subset of the metrics of the desktop version.
2. The threshold values for the quality assessment are hard set. The plugin does not allow setting relative thresholds, e.g. it is not possible to highlight the 25 percent of methods which have the most Lines of Code.
3. There is no mechanism for multivariate analysis, i.e. the plugin only allows assessment when a single metric is considered at a time, but it does not allow to take multiple metrics into account and perform an assessment on these.

Understand by Scitools Understand is a modern analysis tool, which offers mainly visualizations of source code, but also metrics on method level. These metrics can be found in the online documentation under *Program Unit Metrics Report*[11]. However, these are only nine metrics, of which five are covered in the exact same or a similar form by the other tools we investigated. We therefore decided to neglect this tool in the following.

11 https://scitools.com/support/metrics-reports/

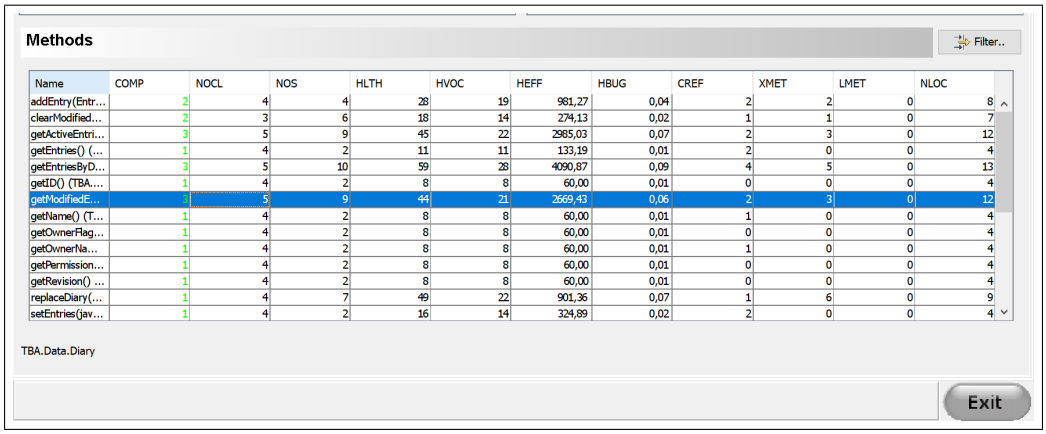


	Name	HVOL	MISM	NII	LOC	LLOC	McCC	NUMPAR	NOS	CD	CLOC	DLOC	NLE	CCO	CI	CLLC	NOI
11	<T extends Object> Optional<T> asOptional(Class<T> clazz)	84	93.56	5	3	3	1	1	1	0.7	7	7	0	0	0	0	1
12	<T extends Object> Stream<? extends T> streamResources(Function<? super Resource, ? extends T> mapper)	151.62	55.84	2	5	5	1	1	1	0	0	0	0	0	0	0	1
13	<T extends Object> T as(Class<T> clazz)	98.1	93.51	52	3	3	2	1	1	0.84	16	16	0	0	0	0	0
14	<T extends Object> T as(Class<T> clazz)	351.03	53.89	0	17	15	5	1	7	0.17	3	3	3	0	0	0	3
15	<T extends Object> T as(Class<T> clazz)	116.69	81.96	0	5	5	2	1	1	0.44	4	4	0	0	0	0	2
16	<T extends Object> T as(Class<T> clazz)	74.23	86.85	1	4	4	1	1	1	0.43	3	3	0	0	0	0	2

Figure 2.4: Methods perspective of the Sourcemeter plugin for Sonarqube

Klocwork by RogueWave From the homepage and an online live demo in which we were able to take part, it seems that Klocwork is a modern static analysis tool which also provides metrics on method level. An overview of the provided metrics can be found in the online documentation [12]. However, we were not able to get a free trial license, therefore we cannot discuss this tool. Yet, from our experience during the online live demo, we guess that the points 2 and 3 of the above mentioned shortcomings of the Sourcemeter plugin for Sonarqube also yield for Klocwork.

JHawk by VirtualMachinery An overview of the metrics provided at method level by JHawk can be found on the homepage of the product[13]. On the one hand, JHawk allows exporting raw data at system-, package-, class- and method level as XML, HTML or CSV files. On the other hand, it also allows for analysis in the application itself: Consider Figure 2.5, which shows a table that is similar structured like the table we saw in the Sourcemeter plugin for Sonarqube. Each row again corresponds to a method and each column to a metric. By double clicking column headers one is able to sort the column values. Colors are used to indicate the assessment of the metrics: green, yellow and red correspond to safe, warning and danger. The absolute thresholds for these levels can be set for every metric in a separate file that is included in the download package. Depending on which version of JHawk is purchased, it is also possible to define filters by the 3 mentioned levels and define own metrics.



Name	COMP	NOCL	NOS	HLTH	HVOC	HEFF	HBUG	CREF	XMET	LMET	NLOC
addEntry(Entr...	2	4	4	28	19	981,27	0,04	2	2	0	8
clearModified...	2	3	6	18	14	274,13	0,02	1	1	0	7
getActiveEntri...	3	5	9	45	22	2985,03	0,07	2	3	0	12
getEntries() (...	1	4	2	11	11	133,19	0,01	2	0	0	4
getEntriesByD...	3	5	10	59	28	4090,87	0,09	4	5	0	13
getID() (TBA....	1	4	2	8	8	60,00	0,01	0	0	0	4
getModifiedE...	3	5	9	44	21	2669,43	0,06	2	3	0	12
getName() (T...	1	4	2	8	8	60,00	0,01	1	0	0	4
getOwnerFlag...	1	4	2	8	8	60,00	0,01	0	0	0	4
getOwnerNa...	1	4	2	8	8	60,00	0,01	1	0	0	4
getPermission...	1	4	2	8	8	60,00	0,01	0	0	0	4
getRevision() ...	1	4	2	8	8	60,00	0,01	0	0	0	4
replaceDiary(...	1	4	7	49	22	901,36	0,07	1	6	0	9
setEntries(jav...	1	4	2	16	14	324,89	0,02	2	0	0	4

Figure 2.5: Method level analysis tab in JHawk

We observed the following shortcomings:

1. The points 2 and 3 of the above mentioned shortcomings of the Sourcemeter plugin for Sonarqube also yield for JHawk.
2. In our opinion, the usability of JHawk can not hold up to modern static analysis tools like the ones we discussed in Chapter 2.1. For example, it takes at least six mousclicks to create a filter in the methods tab shown in Figure 2.5. Furthermore, when filters are created, one is not able to see which one is applied currently and which one not. Another example is that

12 https://docs.roguewave.com/en/klocwork/current/functionandmethodlevelmetrics
13 http://www.virtualmachinery.com/Jhawkmetricslist.htm

threshold values for metrics have to be set in a separate file, which needs to be opened with a text editor. We would have preferred to be able to set this directly in the application.

It is important to not only understand the functionality and structure of the mentioned tools, but also the contents. Hence, in the following section we will briefly introduce some metrics that we have often encountered during our research.

2.3 Metrics on Method Level

Lines of Code (LOC) We encountered this metric in each of the five tools examined. It can also be found in the previously considered static analysis tools like Teamscale and Sonarqube, however not for single methods. We found that some tools categorize the lines of a method and then provide different metrics based on this categorization. For example, the tool *Understand* lists the following metrics, which all provide information about certain types of lines of a method (the metrics and definitions are taken from Understand's documentation[14]).

- Lines: Total number of lines in the function.
- Comment: Number of comment lines in the function.
- Blank: Number of blank lines in the function.
- Code: Number of lines in the function that contain any code.
- Lines-exe: Lines of code in the function that contain no declaration.
- Lines-decl: Lines of code in the function that contain a declaration or part of a declaration.

We further found that even though the investigated tools calculate the same metric, they use different names, e.g. the category of lines which are neither comment nor blank lines, but contain only Java code, was found under the name *Logical Lines of Code* in Sourcemeter[15], *Lines of Java Code* in JavaMetrics Tool[16], *Code* in Understand[14] and *Non-comment, non-blank lines of code in the function or method* in Klocwork[17]. Another approach to LOC is made by the tool JHawk, in which only lines that contain a Java statement are taken into account and lines that only contain brackets are excluded[18]. Thus, our experience coincides with the findings of Ngyuen et al. [NDRTB07] that a standardized definition and calculation of this metric has not yet been put into practice.

Maximum Nesting Depth Similar to LOC, this is a measure that comes in variations, mainly depending on which Java constructs are taken into account. For example, Klocwork provides the metric *Maximum level of control nesting*, which takes *if, switch, for, while, and do-while* statements into account[17], while Sourcemeter calculates the metric *Nesting Level* by also considering *try, catch and finally* statements[15]. A different approach is taken by the JavaMetrics tool, which shows the maximum loop depth separately[16].

Clone Metrics Among the five tools we investigated, Sourcemeter is the only one to provide cloning metrics at a method level. Nevertheless, we want to mention this family metrics at this point, since from our experience code cloning plays an important role in state-of-the-art static analysis tools. It has also been shown in literature that in the process of software maintenance inconsistent changes to duplicated code lead to faults and increase costs [JDHW09]. Sourcemeter provides a total of 8 clone metrics at method level. A detailed list and description can be found in the documentary [15]. It is notable that these metrics are exactly the same which are provided at class and package level, since in our opinion measuring cloning metrics at method level can be misleading for one reason: It is possible that the threshold for checking a piece of code for clones is greater than the length of the method. In order to illustrate this, let us consider a hypothetical clone detection tool and a hypothetical class with two methods with eight lines of Java code each, where the methods are exactly the same except for their names. If the threshold for the detection of clones in the tool is 15 lines of Java code, then the tool would detect a clone at class level, since

14 https://scitools.com/support/metrics-reports/
15 https://www.sourcemeter.com/resources/java/
16 http://www.semdesigns.com/Products/Metrics/JavaMetrics.txt
17 https://docs.roguewave.com/en/klocwork/current/functionandmethodlevelmetrics
18 http://www.virtualmachinery.com/sidebar1.htm

the length of the two methods together exceeds 15, but not at method level. Tools like Teamscale allow the user to configure this threshold, however, Sourcemeter does not. We are also not able to find a threshold in the documentary. Therefore in our opinion the analysis of cloning metrics at method level should be considered with caution.

Halstead Measures Maurice Halstead introduced several metrics in 1977 [Hal77]. These are Program Vocabulary, Program Length, Calculated Program Length, Volume, Difficulty, Effort, Time Required to Program and Number of Delivered Bugs. The respective definitions can be derived from the original publication[Hal77]. All these measures are based on the number of operands and operators. Vocabulary, Length and Difficulty are directly calculated from the number of operands and operators, whereas the other measures take the three ones just mentioned into account, e.g. Effort = Difficulty * Volume. From the original publication it is not exactly clear which constructs of the Java language, which first appeared 18 years after Halstead introduced the measures, correspond to operands and operators. Therefore is is possible that tools calculate the measures differently for the same method.
Curtis et al found that, in the software maintenance activity of modification, these measures correlate with the accuracy of the modification and the time to completion [CSM+79]. However, there is also criticism of the Halstead measures, e.g. Zuse states that the Volume has unclear properties for small progams and calls it the most misunderstood Halstead measure [Zus05].

Cyclomatic Complexity Cyclomatic Complexity is also known as the McCabe Metric and was developed in 1976 by Thomas J. McCabe [McC76]. In contrast to the previously considered metrics, this metric attempts to define complexity in a single number and then use it as a foundation for testing. It is based on the control flow graph of the respective method. The exact calculation can be obtained from the original publication [McC76]. McCabe himself stated that 10 seems to be a reasonable upper bound for his metric and advised programmers to restructure their code of they exceeded this bound. This metric is also a target of criticism. Shepperd states that it is of "very limited utility" and questions the theoretical and empirical fundament of the Cyclomatic Complexity [She88].

Maintainability Index The Maintainability Index was introduced by Oman and Hagemeister in 1992 [OH92]. Similar to McCabe's Cyclomatic Complexity, the Maintainability Index aims at capturing the maintainability of a piece of software in one single value. To achieve this, they first created a hierarchical structure of maintainability attributes and then combined these into one value. The exact categorization and calculations can be found in the original publication [OH92], however, simplifying we can say that it is a combination of lines of code, comments, Halstead Volume and Cyclomatic Complexity [CALO94]. Oman and Hagemeister propose to use this metric to evaluate the relative maintainability of a software system. Some critical arguments are that the Maintainability Index is limited to syntactic analysis and that its goal of evaluating maintainability in a way which leaves out context is questionable [BDP06].

2.4 Summary: Limitations and Shortcomings of Existing Tools

In summary, we can say that there are useful static analysis tools, but they do not provide information at the method level. We find the inspected tools that do provide information at the method level to have significant shortcomings such as too few metrics, outdated for current versions of Java or only providing raw data. Thus, in the next chapter we discuss possible requirements for an ideal static analysis tool.

3 A Hypothetic Maintainability Analysis Tool

In this chapter, we will derive requirements that are needed to perform effective static analysis at method level. Let us therefore first address some points of criticism of metric-oriented static analysis. Besides our finding that existing metrics-oriented tools suffer from shortcomings, metrics-oriented approaches to maintainability in general are target of criticism: Broy et al. state that metrics based approaches are limited to syntactic characteristics only and can not take semantic aspects like appropriate data structures and documentation into account [BDP06]. They conclude that pure metrics contribute insufficiently to proper quality assessment. Other approaches to software quality have been introduced, e.g. activity based ones [DWP+07] [SOP+18]. However, in this chapter we do not want to limit ourselves to existing metrics-oriented approaches but rather work out how an ideal metrics-oriented tool could look like.

Let us recall the overall goal: we want to improve the maintainability of a system by analyzing at the method level. If we want to achieve this using a quantitative approach, we have to ask ourselves what some high level requirements for such a tool can be:

Definite Ranking The tool should rank the methods by their *inspection worthiness*, i.e. it should show us which methods affect the system's maintainability the most and should therefore be prioritized in the maintenance task.

Simplicity Ideally one single metric would be sufficient to facilitate the above described ranking. There already have been several approaches to this like the maintainability index discussed in Chapter 2.3. As we saw, these approaches are not free of criticism and we therefore highly doubt that there will ever be one single metric to satisfy this requirement.

Quality Assessment The tool should not only provide metrics or raw data, but also assess the quality of each metrics or the whole method in order to indicate whether a value is critical or not. We have seen examples of this in Chapter 2.1 with Teamscale's method length assessment.

Customizablity Missing customization options and hard to find configuration options prevent developers from using static analysis tools [JSMHB13]. Therefore the tool should offer such options and make them usable in a comfortable way.

Semantic Aspects The tool should not be limited to syntactical analysis but also be able to assess semantic aspects such as meaningful variable names and comments. To our knowledge, this is not possible today, however there may be steps towards automating such analyses in the future through the use of artificial intelligence.

Concrete Instructions The tool should not only show and assess the metrics, but also give concrete instructions what to improve at which position in the method. For example, Johnson et al. found that developers find it helpful to not only be shown a quality defect, but also be hinted at how to fix it [JSMHB13].

Certainly these requirements are not complete and can be extended, but in our opinion they provide a good guideline for method level maintainability analysis. Therefore, in the next chapter we will present a tool we developed which extends the existing metrics-oriented tools we found towards the requirements we just mentioned. We do not claim to have implemented a single requirement perfectly, nor to have dealt with all of them, yet we think that this tool can make a contribution to assisting in maintenance at method level.

4 A Tool for Maintenance on Method Level

With the development of our own analysis tool, we aim at bridging the gap between the existing tools analyzed in Chapter 2 and the ideal hypothetical tool discussed in Chapter 3. The purpose of this tool is to help a software maintainer decide which method to prioritize. We focused on 4 aspects during the design:

- highlighting values relatively instead of defining absolute thresholds
- application of multivariate analysis in addition to univariate analysis
- integrating findings from existing static analysis tools
- Flexibility: the user should be able to display only those metrics that he considers important

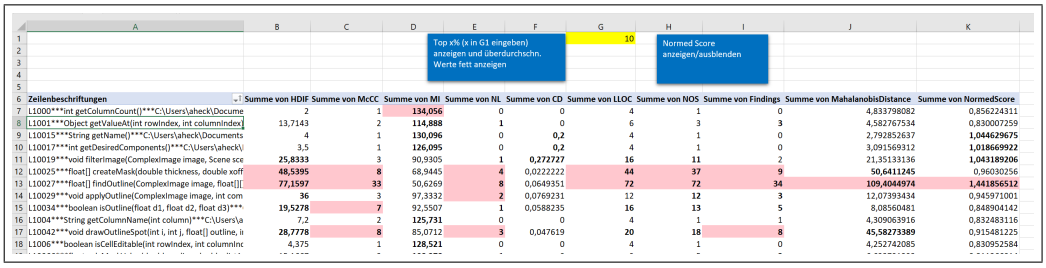
We will now present the functionality of the tool.

4.1 Functionality

Overview Our tool consists of two main components. One is a Python script which takes two inputs: First, a csv file that contains metrics on method level, e.g. from a tool like JHawk or Sourcemeter. Second, a csv file that contains findings, e.g. from a tool like Teamscale or Sonarqube. The script then cleans the data, adds additional information and outputs the data as a xlsx file, which is the default data format of Microsoft Excel. The second component is an Excel pivot table, which uses the data generated by the Python script as a basis. It allows filtering, sorting and highlighting. A software maintainer can use this pivot table to find out which methods to prioritize. Let us now explain the functionality in more detail.

Python script The Python script can be started from the command line. It asks the user for input, notably which metric file and which findings file to process. The other user inputs regard the structure of these files. We included these steps, since we want our tool to be able to process metric files from different sources and cannot assume that different tools follow the same structure regarding their outputs. The script then cleans up the data, scales every metric value between zero and one and stores this information in additional columns. It also calculates the Mahalanobis Distance [Mah36] for every method, which allows one to detect multivariate outliers. From the multitude of existing multivariate statistical methods we have chosen the Mahalanobis Distance, since it is able to deal with many variables. For example, Sourcemeter calculates over 30 metrics on method level. This amount of variables is not well suited when using other outlier detection approaches such as the k-nearest neighbors algorithm [CH67].
From the previously specified findings file, the script now maps the number of findings to each method from the metrics file and stores this information in a new column. Finally, all this data is exported to an xlsx file, which serves as the database for the pivot table.

Pivot table The pivot table is where the actual analysis takes place. Figure 4.1 shows a sample configuration of the table. Rows correspond to methods and columns to metrics. In the leftmost column we see the method ID that was built with the Python script. The three rightmost columns were calculated by the Python script and show the number of findings, Mahalanobis Distance and the sum of the scaled values of all the metrics that are currently shown (this we call *NormedScore*). The other metrics are a subset of those that were included in the metrics csv input file which was read in by the Python script. In contrast to the quality indications we have seen in Chapter 2.2, we highlight values by relative thresholds: in every column, the values that are above average are printed bold, an the largest x percent of values are red, where x can be defined by the user and changed at any time with only one cell input and one mouse click. All the build in Excel pivot table functionalities such as filtering, sorting, defining own formulas and adding or deleting columns via drag and drop are also available.

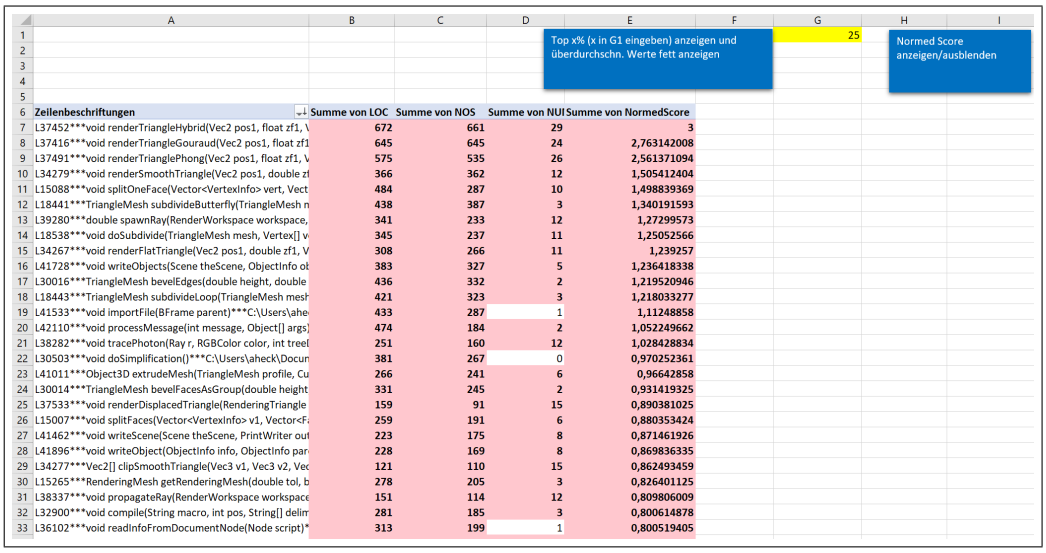


	A	B	C	D	E	F	G	H	I	J	K
1					Top x% (x in G1 eingeben) anzeigen und überdurchschn. Werte fett anzeigen		10	Normed Score anzeigen/ausblenden			
2											
3											
4											
5											
6	Zeilenbeschriftungen	Summe von HDIF	Summe von McCC	Summe von MI	Summe von NL	Summe von CD	Summe von LLOC	Summe von NOS	Summe von Findings	Summe von MahalanobisDistance	Summe von NormedScore
7	L1000***int getColumnCount()***C:\Users\aheck\Docume	2	1	**134,056**	0	0	4	1	0	4,833798082	0,856224311
8	L1001***Object getValueAt(int rowIndex, int columnIndex	13,7143	2	**114,888**	1	0	6	3	3	4,582767534	0,830007259
9	L10015***String getName()***C:\Users\aheck\Documents	4	1	**130,096**	0	**0,2**	4	1	0	2,792852637	**1,044629675**
10	L10017***int getDesiredComponents()***C:\Users\aheck\	3,5	1	**126,095**	0	**0,2**	4	1	0	3,091569312	**1,018669922**
11	L10019***void filterImage(ComplexImage image, Scene sce	**25,8333**	3	90,9305	**1**	**0,272727**	**16**	**11**	2	21,35133136	**1,043189206**
12	L10025***float[] createMask(double thickness, double xoff	**48,5395**	**8**	68,9445	**4**	0,0222222	**44**	**37**	**9**	**50,6411245**	0,96030256
13	L10027***float[] findOutline(ComplexImage image, float[][	**77,1597**	**33**	50,6269	**8**	0,0649351	**72**	**72**	**34**	**109,4044974**	**1,441856512**
14	L10029***void applyOutline(ComplexImage image, int com	**36**	3	97,3332	**2**	0,0769231	12	**12**	**3**	12,07393434	0,945971001
15	L10034***boolean isOutline(float d1, float d2, float d3)***	**19,5278**	**7**	92,5507	**1**	0,0588235	**16**	**13**	**5**	8,08560481	0,848904142
16	L1004***String getColumnName(int column)***C:\Users\a	7,2	2	**125,731**	0	0	4	1	1	4,309063916	0,832483116
17	L1042***void drawOutlineSpot(int i, int j, float[] outline, i	**28,7778**	**8**	85,0712	**3**	0,047619	**20**	**18**	**8**	**45,58273389**	0,915481225
18	L1006***boolean isCellEditable(int rowIndex, int columnIn	4,375	1	**128,521**	0	0	4	1	0	4,252742085	0,830952584

Figure 4.1: Sample configuration of the pivot table

An exemplary scenario in practice could then look as follows: A software maintainer sorts the table by the Mahalanobis Distance to display the largest values at the top. Since the Mahalanobis Distance takes all metrics from the original metrics csv file into account, the methods listed at the top tend to have extreme values in several metrics. If one assumes that the methods at the top are a bigger threat to the maintainability of the system than the methods at the bottom, and therefore need to be refactored with a higher probability, a software maintainer can prioritize these methods.

Another approach would be to sort after the *NormedScore*. In contrast to the Mahalanobis Distance, the NormedScore only takes the metrics into account which are displayed currently. It is calculated by the sum of the scaled metric values and therefore weights all displayed metrics equally. One possible scenario where you want to use this could be the following: If one is interested in the three metrics Lines of Code, Number of Statements and Number of Parameters, one could display only these metrics and then click on the right blue button at the top to display the NormedScore and sort after it. Figure 4.2 shows this configuration with an example project.



	A	B	C	D	E	F	G	H	I
1					Top x% (x in G1 eingeben) anzeigen und überdurchschn. Werte fett anzeigen		25	Normed Score anzeigen/ausblenden	
2									
3									
4									
5									
6	Zeilenbeschriftungen	Summe von LOC	Summe von NOS	Summe von NUI	Summe von NormedScore				
7	L37452***void renderTriangleHybrid(Vec2 pos1, float zf1, V	**672**	**661**	**29**	**3**				
8	L37416***void renderTriangleGouraud(Vec2 pos1, float zf1	**645**	**645**	**24**	**2,763142008**				
9	L37491***void renderTrianglePhong(Vec2 pos1, float zf1, V	**575**	**535**	**26**	**2,561371094**				
10	L34279***void renderSmoothTriangle(Vec2 pos1, double zf	**366**	**362**	**12**	**1,505412404**				
11	L15088***void splitOneFace(Vector<VertexInfo> vert, Vect	**484**	**287**	**10**	**1,498839369**				
12	L18441***TriangleMesh subdivideButterfly(TriangleMesh n	**438**	**387**	**3**	**1,340191593**				
13	L39280***double spawnRay(RenderWorkspace workspace,	**341**	**233**	**12**	**1,27299573**				
14	L18538***void doSubdivide(TriangleMesh mesh, Vertex[] v	**345**	**237**	**11**	**1,25052566**				
15	L34267***void renderFlatTriangle(Vec2 pos1, double zf1, V	**308**	**266**	**11**	**1,239257**				
16	L41728***void writeObjects(Scene theScene, ObjectInfo ob	**383**	**327**	**5**	**1,236418338**				
17	L30016***TriangleMesh bevelEdges(double height, double	**436**	**332**	**2**	**1,219520946**				
18	L18443***TriangleMesh subdivideLoop(TriangleMesh mesh	**421**	**323**	**3**	**1,218033277**				
19	L41533***void importFile(BFrame parent)***C:\Users\ahe	**433**	**287**	1	**1,11248858**				
20	L42110***void processMessage(int message, Object[] args)	**474**	**184**	**2**	**1,052249662**				
21	L38282***void tracePhoton(Ray r, RGBColor color, int treel	**251**	**160**	**12**	**1,028428834**				
22	L30503***void doSimplification()***C:\Users\aheck\Docur	**381**	**267**	0	**0,970252361**				
23	L41011***Object3D extrudeMesh(TriangleMesh profile, Cu	**266**	**241**	**6**	**0,96642858**				
24	L30014***TriangleMesh bevelFacesAsGroup(double height	**331**	**245**	**2**	**0,931419325**				
25	L37533***void renderDisplacedTriangle(RenderingTriangle	**159**	**91**	**15**	**0,890381025**				
26	L15007***void splitFaces(Vector<VertexInfo> v1, Vector<F	**259**	**191**	**6**	**0,880353424**				
27	L41462***void writeScene(Scene theScene, PrintWriter out	**223**	**175**	**8**	**0,871461926**				
28	L41896***void writeObject(ObjectInfo info, ObjectInfo par	**228**	**169**	**8**	**0,869836335**				
29	L34277***Vec2[] clipSmoothTriangle(Vec3 v1, Vec3 v2, Vec	**121**	**110**	**15**	**0,862493459**				
30	L15265***RenderingMesh getRenderingMesh(double tol, b	**278**	**205**	**3**	**0,826401125**				
31	L38337***void propagateRay(RenderWorkspace workspace	**151**	**114**	**12**	**0,809806009**				
32	L32900***void compile(String macro, int pos, String[] delim	**281**	**185**	**3**	**0,800614878**				
33	L36102***void readInfoFromDocumentNode(Node script)*	**313**	**199**	1	**0,800519405**				

Figure 4.2: The table is sorted by the metric *normedScore*.

In this case, the relative threshold value for the red coloring was set to 25 percent (corresponds to the upper quartile). We can now see that most of the methods that have the highest NormedScore also are within the upper quartile of the other 3 metrics. One could therefore decide to prioritize the methods at the top for maintenance.

4.2 Technical Background

Let us now briefly discuss some technical aspects which are interesting in our opinion.

Automation As stated, our tool consists of two components, the Python script and the pivot table. These are wrapped in a batch script, which first starts the Python script and then opens

up the xlsm (Excel's file format when macros are used) file which contains the pivot table. Every time the xlsm file is opened, the pivot table automatically refreshes its database (the output files of the Python script). This way we achieve a high degree of automation.

Power Query Another feature worth mentioning is Microsoft Power Query for Excel, which is a build in Excel feature since the 2016 versions. On the one hand, this allows us to use multiple output files from the Python script (e.g. from different software projects) as a database for the pivot table, as long as they are stored in the same folder. This means that in one table, methods from multiple projects can be analyzed. On the other hand, Power Query allows to store the data from the xlsx files in the background (i.e. the data is not stored in a separate sheet in the file that contains the pivot table). This allows automatically refreshing the pivot table when the containing xlsm file is opened.

Programming For cleaning and processing the data in the Python script, we use common Python modules such as pandas, scipy and numpy. In the xlsm file, we use two VBA macros to highlight specific values and calculate the NormedScore. The user can run these macros by clicking blue buttons above the pivot table. Examples of these can be seen in Figure 4.1 and 4.2.

5 Discussion

Let us first discuss the strengths and limitations of our tool before we present an idea for further work.

Strengths

a) Our tool offers an easy to use interface that is already familiar to users with basic knowledge of Microsoft Excel pivot tables. The numerous filter and sorting options exceed those of the existing tools we found at the method level. Advanced Excel users with knowledge in Visual Basic For Applications can write their own macros, e.g. to format the data according to their needs.

b) The tool also offers a high degree of flexibility as it allows the user to choose which metrics to display and which not. This selection can be changed at any time.

c) To our knowledge, this is the first static analysis tool at method level which uses a multivariate measure to take all the metrics that are available into account. This allows one to find out which methods are probably more critical to maintainability than others.

d) Furthermore, through the use of Power Query we achieve a high degree of automation and are able to analyze data from multiple projects.

Limitations

a) Compared to the presented Sourcemeter plugin for Sonarqube in Chapter 2.2 it is not possible to display the source code of the corresponding class by clicking on the method name. There is no direct connection between the source code and the pivot table. This limitation could be overcome when the concepts of our tool are directly integrated into a state-of-the-art static analysis tools. This will be explained in more detail under *Future Work*.

b) Mapping the number of findings to methods is currently limited to csv files generated by Teamscale. It is also conceivable to enable other tool's outputs as data sources. This requires additional effort in programming the Python script.

c) Furthermore, the tool cannot display which finds belong to a method, only the amount of findings. Thus, it is not possible to give a software maintainer a concrete hint which defect was found and how to fix it. In the next paragraph we will see how this limitation could be addressed in the future.

Future Work In the future, it is conceivable that the approach that this tool takes could be integrated into a modern, findings-oriented static analysis tool such as Teamscale, e.g. via a plugin. This could be done by creating a method perspective that is similar to the basic structure of our tool. The prerequisite for this is that the surrounding tool calculates metrics on the method level. If this were not the case, our tool would have to be extended to calculate the metrics itself instead of importing csv files from another tool.

On the one hand, this integration should keep the strengths that our tool offers currently, such as numerous sorting and filtering options, highlighting based on relative thresholds and a multivariate outlier measure. On the other hand, the previously mentioned limitations could be overcome. Here is an example regarding limitation a):

In Teamscale and Sonarqube it is possible to display the part of the source code in which a finding is located by simply clicking on the finding. Therefore, a similar mechanism for a method perspective is conceivable, in which a click on the method displays the corresponding source code, just as it is already possible today in Sourcemeter's plugin for Sonarqube.

Limitation b) would become obsolete, since there is no more need to process findings from other static analysis tools when the plugin is already integrated in a findings-oriented static analysis tools.

Limitation c) could also be overcome when a mapping technique similar to the one we used was implemented and information about the findings, such as the description or date, which are already existing in today's static analysis tools, would then be added.

The main advantage of an direct integration would be the combination of the findings-oriented and the metrics-oriented approaches. A typical scenario in practice could look as follows: A software maintainer uses the methods perspective to identify a concrete method which could be highly critical to maintainability based on its metric values. He then would be able to display which findings belong to this method and remove these defects in the source code.

This means that one would no longer have to maintain the system finding by finding, but rather method by method and then inspect the findings within a method. In our opinion, this approach could lead to more efficient maintenance for two reasons: First, it allows one to prioritize critical methods. The second reason is based on our opinion that a single method is easier to understand than the whole corresponding class. In today's state-of-the-art findings-oriented tools, it is only possible to filter findings down to the class level. If a plugin version of our tool now changes this and makes the method level available, a maintainer can work more efficient when fixing defects method by method compared to class by class. Let us give an hypothetical scenario in which this could be the case: Imagine a maintainer M wants to fix all findings of a specific class. There are three entries in the list of findings of the class: The first and third finding belong to method A, and the second belongs to method B. Assume that fixing each finding requires M to completely understand the corresponding method. If M now follows the method based approach, he would first try to understand method A, fix the first and the third finding and then move on to method B and the second finding. If he would not follow the method based approach, he would first try to understand method A and fix the first finding, then try to understand method B and the second finding and then finally go back to method A and fix the third finding. However, it is possible that in the meantime, he is not completely aware of the functionality of method A anymore and thus has to spend additional time to understand it once more.

6 Conclusion

The goal of this study is to identify Java methods which should be prioritized for software maintenance. Our quantitative approach eliminates weaknesses of existing tools, which only offer setting absolute thresholds for metrics, lack analysis of multiple metrics at once and do not take quality defects found by state-of-the-art static analysis tools into account. The tool we developed features a Python script for cleaning and preparing quantitative data which then serves as an input for an easy to use and flexible Microsoft Excel tool. With Power Query, it uses a modern data importing technology and offers all the build-in pivot table analysis functions such as sorting, filtering and choosing the metrics to display. Additionally, the tool allows highlighting metric values by relative thresholds, detecting multivariate outliers and integrates the number of findings from static analysis tools. Through the use of these features it is possible to find out which methods have particularly conspicuous metric values compared to other methods, be it for a single metric or multiple metrics or once. Assuming that especially these methods could be critical for the maintainability of the system and therefore probably need to be refactored, they can be inspected with a high priority.

This is a part of our early work to bridge the gap between those tools that focus their presentation on metrics and findings, respectively. In the future it is conceivable to integrate the concepts of our tool directly into modern static analysis tools via a plugin. Since these tools typically provide a view of the source code files, an integrated plugin version of our tool could allow to not only show the number of findings for each method, but also the finding's message and the respective piece of code, where changes should be made. This way one could combine the insights of both the metrics-oriented and the findings-oriented approaches. Furthermore it would allow a structured approach to the process of maintenance method by method.

Bibliography

[BBL76] Barry W Boehm, John R Brown, and Mlity Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605, 1976.

[BDP06] Manfred Broy, Florian Deissenboeck, and Markus Pizka. Demystifying maintainability. In *Proceedings of the 2006 International Workshop on Software Quality*, WoSQ 2006, pages 21–26, New York, NY, USA, 2006. Association for Computing Machinery.

[CALO94] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[CH67] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[CM78] Joseph P. Cavano and James A. McCall. A framework for the measurement of software quality. In *Proceedings of the Software Quality Assurance Workshop on Functional and Performance Issues*, pages 133–139, New York, NY, USA, 1978. Association for Computing Machinery.

[CSM+79] Bill Curtis, Sylvia Sheppard, Phil Milliman, M.A. Borst, and Tom Love. Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics. *Software Engineering, IEEE Transactions on*, SE-5:96– 104, 04 1979.

[DWP+07] Florian Deissenboeck, Stefan Wagner, Markus Pizka, Stefan Teuchert, and J-F Girard. An activity-based quality model for maintainability. In *2007 IEEE International Conference on Software Maintenance*, pages 184–193. IEEE, 2007.

[Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA, 1977.

[(IS01] International Standard Organization (ISO). International standard iso/iec 9126, information technology - product quality - part1: Quality model, 2001.

[ISO11] ISO/IEC. Iso/iec 25010: Systems and software engineering-systems and software quality requirements and evaluation (square)-system and software quality models. *Technical Report*, 2011.

[JDHW09] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering*, pages 485–495. IEEE, 2009.

[JSMHB13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[Mah36] Prasanta Chandra Mahalanobis. On the generalized distance in statistics. National Institute of Science of India, 1936.

[McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[NDRTB07] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A sloc counting standard. In *Cocomo ii forum*, volume 2007, pages 1–16. Citeseer, 2007.

[OH92] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–344, 1992.

[Pig96] Thomas M Pigoski. *Practical software maintenance: best practices for managing your software investment*. Wiley Publishing, 1996.

[She88] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

[SOP+18] Markus Schnappinger, Mohd Hafeez Osman, Alexander Pretschner, Markus Pizka, and Arnaud Fietzke. Software quality assessment in practice. In Markku Oivo, Daniel Méndez, and Audris Mockus, editors, *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–6, New York, NY, USA, 10112018. ACM.

[Zus05] Horst Zuse. *Resolving the mysteries of the Halstead measures*. Technische Universität Berlin, Fakultät IV-Elektrotechnik und Informatik, 2005.