Comparing software metrics tools

Conference Paper · January 2008

DOI: 10.1145/1390630.1390648 · Source: DBLP

3 authors:

Rüdiger Lincke
Linnaeus University
15 PUBLICATIONS **396** CITATIONS
SEE PROFILE

Jonas Lundberg
Linnaeus University
44 PUBLICATIONS **429** CITATIONS
SEE PROFILE

Welf Löwe
Linnaeus University
169 PUBLICATIONS **1,447** CITATIONS
SEE PROFILE

Comparing Software Metrics Tools

Rüdiger Lincke, Jonas Lundberg and Welf Löwe
Software Technology Group
School of Mathematics and Systems Engineering
Växjö University, Sweden
{rudiger.lincke|jonas.lundberg|welf.lowe}@vxu.se

ABSTRACT

This paper shows that existing software metric tools interpret and implement the definitions of object-oriented software metrics differently. This delivers tool-dependent metrics results and has even implications on the results of analyses based on these metrics results. In short, the metrics-based assessment of a software system and measures taken to improve its design differ considerably from tool to tool. To support our case, we conducted an experiment with a number of commercial and free metrics tools. We calculated metrics values using the same set of standard metrics for three software systems of different sizes. Measurements show that, for the same software system and metrics, the metrics values are tool depended. We also defined a (simple) software quality model for "maintainability" based on the metrics selected. It defines a ranking of the classes that are most critical wrt. maintainability. Measurements show that even the ranking of classes in a software system is metrics tool dependent.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Product metrics*

General Terms

Measurement, Reliability, Verification

1. INTRODUCTION

Accurate measurement is a prerequisite for all engineering disciplines, and software engineering is not an exception. For decades seek engineers and researchers to express features of software with numbers in order to facilitate software quality assessment. A large body of software quality metrics have been developed, and numerous tools exist to collect metrics from program representations. This large variety of tools allows a user to select the tool best suited, e.g., depending on its handling, tool support, or price. However, this assumes that all metrics tools compute / interpret / implement the same metrics in the same way.

For this paper, we assume that a *software metric* (or metric in short) is a mathematical definition mapping the entities of a software system to numeric *metrics values*. Furthermore, we understand a *software metrics tool* as a program which implements a set of software metrics definitions. It allows to assess a software system according to the metrics by extracting the required entities from the software and providing the corresponding metrics values. The Factor-Criteria-Metric approach suggested by McCall [27] applied to software leads to the notion of a *software quality model*. It combines software metrics values in a well-defined way to aggregated numerical values in order to aid quality analysis and assessment. A suitable software quality model is provided by the ISO 9126 [16, 17].

The goal of this paper is to answer the following two questions: First, do the metrics values of given software system and metrics definitions depend on the metrics tool used to compute them? Second, does the interpretation of metrics values of a given software system as induced by a quality model depend on the metrics tool?

Pursuing the above questions with an experiment might appear as overkill at first glance. However, the implications of a rejection of these hypotheses are of both practical and scientific interest: From a practical point of view, engineers and managers should be aware that they make metrics-*tool*-dependent decisions, which does not necessarily follow the reasoning and intention of those who defined the metrics. For instance, the focus on maintenance of critical classes – where "critical" is defined with a metrics-based assessment – would be relative to the metrics tool used. What would be the "right" decision then? Scientifically, the validation of (the relevance of) certain metrics is still an open issue. Controlled experiments involving metrics-based and manual assessment of a variety of real-world software systems are costly. However, such a validation would then not support/reject the validity of a software metrics set but rather the validity of the software metrics tool used. Thus, the results of such an experiment cannot be compared or generalized; the costs of such experiments could not be justified. Besides, assessing the above questions in detail also follows the arguments given in [31], which suggests that there should be more experimentation in computer science research.

The remainder of this paper is structured as follows: Section 2 describes an industrial project which led to the questions raised in this paper. It supports the practical relevance of the conducted experiments. Section 3 discusses

ISSTA'08, July 20–24, 2008, Seattle, Washington, USA.

some practical issues and sharpens the research questions. Section 4 presents the setup of our experiments. Sections 5 and 6 describe the experimental results and our interpretations for the two main questions respectively. In Section 7, we discuss threats to the validity of our study. Finally, in Section 8, we conclude our findings and discuss future work.

2. BACKGROUND

Eurocontrol develops, together with its partners, a high level design of an integrated Air Traffic Management (ATM) system across all ECAC States.[1] It will supersede the current collection of individual national systems [9]. The system architecture, called Overall ATM/CNS Target Architecture (OATA), is a UML specification.

As external consultants, we supported the structural assessment of the architecture using a metrics-based approach using our software metrics tool VizzAnalyzer. A second, internal assessment team used NTools, another software metrics tool. This redundant assessment provided the chance to uncover and avoid errors in the assessment method and tools. The pilot validation focused only on a subsystem of the complete architecture, which consisted of 8 modules and 70 classes. We jointly defined the set of metrics which quantify the architecture quality, the subset of the UML specification (basically class and sequence diagrams) as well as a quality model for maintainability and usability.

The definitions of the metrics we first selected refer to standard literature. During implementation, we found that the metrics definitions are ambiguous, too ambiguous to be implemented in a straight-forward way, and even too ambiguous to (always) be interpreted the same way by other participants in the assessment team. We therefore jointly created a *Metrics Definition Document* defining the metrics and their variants that should be used, the relevant software entities, attributes and relations – actually we defined a UML and project specific meta-model – and the exact scope of the analysis.

Among other things, we learned some lessons related to software metrics. Several issues with the metrics *definitions* exist: Unclear and inexact definitions of metrics open up the possibility for different interpretations and implementations. Different variants of the same metric are not distinguished by name, which makes it difficult to refer to a particular variant. Well known metrics from literature are used with slight deviations or interpreted differently than suggested originally, which partially changes the meaning of the metric. Consequently, deviations of metrics implementations in the metrics tools exist and, hence, metrics values are not comparable. More specifically, even though our VizzAnalyzer and NTools refer to the same informal metrics definitions, the results are not comparable. Despite creating a Metrics Definition Document for fixing variants in the metrics definition, it still did not solve the problem since it did not formally map the UML language to the meta-model, and the metrics definitions still used natural language and semi-formal approaches.

Most issues could be solved with the next iteration of the assessment. The UML to meta-model mapping was included, and the metrics definitions were improved. However, this required quite an effort (unrelated to the client's analysis questions but rather to the analysis method). Hence, two questions attracted our attention even after the project:

Q1 Do different software metric tools in general calculate different metrics values for the same metrics and the same input?

Q2 If yes, does this matter? More specifically, are these differences irrelevant measurement inaccuracies, or do they lead to different conclusions?

3. HYPOTHESES & PRACTICAL ISSUES

We want to know if the differences we observed between VizzAnalyzer and NTools in the context of the Eurocontrol project are just coincidental, or if they can also be observed in other contexts and with other software metrics tools. However, we want our approach to be both conservative wrt. the scientific method and practically relevant.

The set of tools, metrics, and test systems is determined by practical considerations. A detailed discussion of the final selection is provided in Section 4. But beforehand, we discuss the selection process. Ideally, we would install all metrics tools available, measure a random selection of software systems, and compare the results for all known metrics. But reality implies yet a number of practical limitations.

First, we cannot measure each metric with each tool, since the selection of implemented metrics differs from tool to tool. Hence, maximizing the set of metrics would reduce the set of comparable tools and vice versa. We need to compromise and select the metrics which appear practically interesting *to us*. The metrics we focus our experiment on include mainly object-oriented metrics as described in the metrics suites of, e.g., Chidamber & Kemerer, Li and Henry, et al. [5, 24].

Second, the availability of the metrics tools is limited, and we cannot know of all available tools. We found the tools after a thorough search on the internet, using the standard search engines and straight-forward search terms. We also followed references from related work. Legal restrictions, the programming languages the tools can analyze, the metrics they are capable to calculate, the size of the systems they can be applied on, and the data export functions pose further restrictions on the selection of tools. As a consequence, we selected only tools available without legal restrictions and which were meaningful to compare, i.e., those tools can analyze the same systems with the same metrics.

Third, further limitations apply to the software systems analyzed. We obviously cannot measure all available systems; there are simply too many. Also, legal restrictions limit the number of suitable systems. Most metrics tools need the source code and, therefore, we restricted ourselves to open source software as available on SourceForge.NET[2]. Additionally, the available software metrics tools limited the programming languages virtually to either Java or C/C++.

Finally, we cannot compare all metrics values of all classes of all systems to a "gold standard" deciding on the correctness of the values. Such a "gold standard" simply does not exist and it is impossible to compute it since the original metrics definitions are too imprecise. Thus, we restrict ourselves to test whether or not there are tool dependent differences of the metrics values. Considering the limitations, we scientifically assess our research question Q1 by *invalidating* the following hypothesis:

[1] European Civil Aviation Conference; an intergovernmental organization with more than 40 European states.

[2] http://sourceforge.net

H1 Different software metrics tools calculate the same metrics values for the same metrics and input system.

Given that H1 can be rejected, i.e., there exist differences in the measured values for different metrics tools, we aim to find out whether or not these differences really make a difference when using the metrics values for further interpretations and analyses, further referred to as *client analyses*. For designing an experiment, it could, however, be helpful to know of possible client analyses within software engineering. Bär et al. describe in the FAMOOS Object-Oriented Reengineering Handbook [3] several common (re-)engineering tasks as well as techniques supporting them. They address a number of goals and problems ranging from unbundling tasks, fixing performance issues, porting to other platforms and design extractions, to solving particular architectural problems, to general code cleanup. For details, refer to [3].

We have to deal with practical limitations, which include that we cannot investigate all client analyses. We decide therefore to select a (hopefully) representative and plausible client analysis for our investigation. Assuming the results can be transferred (practically) to other client analyses.

One client analysis of software metrics suggested in the FAMOOS Handbook supports controlling and focussing reengineering tasks by pin-pointing critical key classes of a system. This is actually the approach used in the Eurocontrol/OATA project discussed before. We design a similar client analysis for assessing Q2. The analysis assesses the maintainability of a system and identifies its least maintainable classes. The intention is to improve the general maintainability of the system by focussing on these classes first. The actual improvement is not part of our scope. This client analysis abstracts from the actual metrics values using a software quality model for maintainability. Moreover, the absolute value of maintainability of each class is even further abstracted to a rank of this class, i.e., we abstract the absolute scale metrics of maintainability to an ordinal scale. More specifically, we look at the list of the top 5–10 ranked classes according to their need for maintainability. We will compare these class lists – as suggested – based on the metrics values of the different metrics tools. Considering these limitations, we scientifically assess our research question Q2 by *invalidating* the following hypothesis:

H2 For client analyses based on the same metrics, different software metrics tools always deliver the same results for the input system.

From a practical point of view, the object of our study is the ordered set of least maintainable classes in the test systems. The purpose of the study is to investigate whether or not tool-dependent differences in the metrics values persist after abstraction through a software quality model, which can lead to different decisions (sets of classes to focus on). The perspective is from the point of view of software engineers using metrics tools for performing measures in a software system in order to find the least maintainable classes in a software system. The main effects studied are the differences between the sets of least maintainable classes calculated by each tool for a test system. The set of tools, metrics, software quality model and test systems will be determined by practical considerations. The context of the study is the same as for question Q1 above.

4. EXPERIMENTAL SETUP

Our experiments were performed on the available working equipment, i.e., a standard PC satisfying the minimum requirements for all software metrics tools. All measurements were performed on this computer and the extracted data were stored for further processing.

4.1 Software Metrics Tool Selection

For finding a set of suitable software metrics tools, we conducted a free search on the internet. Our first criteria was that the tools calculate any form of software metrics. We collected about 46 different tools which we could safely identify as software metrics tools. For each tool, we recorded: name, manufacturer, link to home page, license type, availability, (programming) languages supported, operating system/environment, supported metrics.

After a pre-analysis of the collected information, we decided to limit the set of tools according to analyzable languages, metrics calculated, and availability/license type. We found that the majority of metrics tools available can derive metrics for Java programs, others analyze C/C++, UML, or other programming languages. In order to compare as many tools as possible, we chose to analyze Java programs. Furthermore, about half of the tools are rather simple "code counting tools". They basically calculate variants of the Lines of Code (LOC) metric. The other half calculates (in addition to LOC) more sophisticated software metrics, as they have been described and discussed in literature, including metrics suites like the Halsted Metrics, Chidamber & Kemerer, Li and Henry, etc. [5, 10, 24] Moreover, not all tools are freeware, or commercial versions do not provide suitable evaluation licenses. Thus, our refined criteria focus on: *language*: Java (source- or byte-code), *metrics*: well-known object-oriented metrics on class level, *license type*: freely available or evaluation licenses.

Applying these new criteria left us with 21 commercial and non-commercial tools on which we took a closer look. Investigating the legal status of the tools, we found that some of them are limited to analyze just a few files at a time, or we can simple not get hands on these programs. Our final selection left us with the following 10 software metrics tools[3]:

Analyst4j is based on the Eclipse platform and available as a stand-alone Rich Client Application or as an Eclipse IDE plug-in. It features search, metrics, analyzing quality, and report generation for Java programs[4].

CCCC is an open source command-line tool. It analyzes C++ and Java files and generates reports on various metrics, including Lines Of Code and metrics proposed by Chidamber & Kemerer and Henry & Kafura [5].

Chidamber & Kemerer Java Metrics is an open source command-line tool. It calculates the C&K object-oriented metrics by processing the byte-code of compiled Java files[6].

Dependency Finder is open source. It is a suite of tools for analyzing compiled Java code. Its core is a dependency

[3]Tools considered but not selected because of the final criteria: CMTJava [32], Resource Standard Metrics [28], CodePro AnalytiX [15], Java Source Code Metrics [30], JDepend [6], JHawk [33], jMetra [14], JMetric [18], Krakatau Metrics [29], RefactorIT [2], and SonarJ [11].

[4]http://www.codeswat.com

[5]http://sourceforge.net/projects/cccc

[6]http://www.spinellis.gr/sw/ckjm

analysis application that extracts dependency graphs and mines them for useful information. This application comes as a command-line tool, a Swing-based application, a web application, and a set of Ant tasks[7].

Eclipse Metrics Plug-in 1.3.6 by Frank Sauer is an open source metrics calculation and dependency analyzer plugin for the Eclipse IDE. It measures various metrics and detects cycles in package and type dependencies[8].

Eclipse Metrics Plug-in 3.4 by Lance Walton is open source. It calculates various metrics during build cycles and warns, via the Problems View, of metrics 'range violations'[9].

OOMeter is an experimental software metrics tool developed by Alghamdi et al.It accepts Java/C# source code and UML models in XMI and calculates various metrics [1].

Semmle is an Eclipse plug-in. It provides an SQL like querying language for object-oriented code, which allows to search for bugs, measure code metrics, etc.[10].

Understand for Java is a reverse engineering, code exploration and metrics tool for Java source code[11].

VizzAnalyzer is a quality analysis tool. It reads software code and other design specifications as well as documentation and performs a number of quality analyses[12].

4.2 Metrics Selection

The metrics we selected are basically the "least common denominator", the largest common subset of the metrics assessable by all selected software metrics tools.

We created a list of all metrics which can be calculated by any of the tools considered. It turned out that the total number of different metrics (different by name) is almost 200. After carefully reading the metrics descriptions, we found that these different names seem to describe 47 different metrics. Matching them was not always straight forward and in some cases it is nothing but a qualified guess. Those 47 metrics work on different program entities, e.g., method, class, package, program, etc.

We considered only metrics as comparable when we were certain that the same concepts were meant. Further, we selected "class" metrics only, since this is the natural unit of object-oriented software systems and most metrics have been defined and calculated on class level. This left 17 object-oriented metrics which (i) we could rather securely assign to the same concept, (ii) are known and defined in literature, and (iii) work on class level. Of these metrics, we selected 9 which most of the 10 remaining software metric tools can calculate. The tools and metrics are shown in Table 1. The crosses "x" marks that a metrics can be calculated by the corresponding metric tool. It follows a brief description of the metrics finally selected:

CBO (Coupling Between Object classes) is the number of classes to which a class is coupled [5].

DIT (Depth of Inheritance Tree) is the maximum inheritance path from the class to the root class [5].

LCOM-CK (Lack of Cohesion of Methods) (as originally proposed by Chidamber & Kemerer) describes the lack of cohesion among the methods of a class [5].

[7]http://depfind.sourceforge.net
[8]http://sourceforge.net/projects/metrics
[9]http://eclipse-metrics.sourceforge.net
[10]http://semmle.com
[11]http://www.scitools.com
[12]http://www.arisa.se

Tools	Metrics								
Name	CBO	DIT	LCOM-CK	LCOM-HS	NOC	NOM	RFC	TCC	WMC
Analyst4j	x	x	x		x	x	x		x
CCCC	x	x			x	x			
Chidamber & Kemmerers Java Metrics	x	x	x		x	x	x		
Dependency Finder		x			x	x			
Eclipse Metrics Plugin 1.3.6		x		x	x	x			x
Eclipse Metrics 3.4			x	x					x
OOMeter	x	x	x		x			x	
Semmle		x	x	x	x	x	x		
Understand for Java	x	x	x		x	x			
VizzAnalyzer	x	x	x		x	x	x	x	x

Figure 1: Tools and metrics used in evaluation

LCOM-HS (Lack of Cohesion of Methods) (as proposed by Henderson-Sellers) describes the lack of cohesion among the methods of a class [12].

LOC (Lines Of Code) counts the lines of code of a class [13].

NOC (Number Of Children) is the number of immediate subclasses subordinated to a class in the class hierarchy [5].

NOM (Number Of Methods) is the methods in a class [12].

RFC (Response For a Class) is the set of methods that can potentially be executed in response to a message received by an object of the class [5].

WMC (Weighted Methods per Class) (using Cyclomatic Complexity [34] as method weight) is the sum of weights for the methods of a class [5].

Providing an unambiguous definition of these metrics goes beyond the scope of this paper. For details about the metric definitions, please refer to the original sources of the metrics. These often do not go far beyond the description given above, making it difficult to infer the complexity of the metrics and what it takes to compute them. This situation is part of the problem we try to illuminate. We discuss the unambiguity of metrics, consequences and possible solutions in [25] and provide unambiguous metrics definitions in [26].

4.3 Software Systems Selection

With the selection of software metrics tools, we limited ourselves to test systems written in Java (source and byte code). *SourceForge.NET* provides a large variety of open source software projects. Over 30.000 are written in Java and it is possible to search directly for Java programs of all kinds. Thus, we downloaded about 100 software projects which we selected more or less randomly. We tried to get a large variety of projects from different categories in the SourceForge classification. We preferred programs with a high ranking according to SourceForge, since we assumed that these programs have a larger user base, hence relevance.

We chose to analyze projects in different size categories. Because of the limited licenses of some commercial tools, we quite arbitrarily selected sizes of about 5, 50 and 500 source files. From the samples we downloaded, we randomly selected the final sample of three Java programs, one for each of our size categories. We do not expect that the actual program size affects the results of our study, but we prefer to work on diverse samples. The programs selected are:

Jaim implements the AOL IM TOC protocol as a Java library. The primary goal of JAIM is to simplify writing AOL bots in Java, but it could also be used to create Java based AOL clients. It consists of 46 source files. Java 1.5.[13]

[13]http://sourceforge.net/projects/jaimlib

jTcGUI is a Linux tool for managing TrueCrypt volumes. It has 5 source files. Java 1.6.[14]
ProGuard is a free Java class file shrinker, optimizer, and obfuscator. It removes unused classes, fields, methods, and attributes. It then optimizes the byte-code and renames the remaining classes, fields, and methods using short meaningless names. It consists of 465 source files. Java 1.5.[15]

4.4 Selected Client Analysis

We reuse some of the metrics selected to define a client analysis answering question Q2. We apply a software quality model for abstracting from the single metrics values to a maintainability value, which can be used to rank the classes in a software system according to their maintainability. As basis for the software quality model we use *Maintainability* as one of the six factors defined in ISO 9126 [16, 17]. We use four of its five criteria: *Analyzability*, *Changeability*, *Stability*, and *Testability*, and omit Compliance.

In order to be able to use the software quality model with all tools, we can only include metrics which are calculated by all tools. We should also have as many metrics as possible: we should have at least one coupling, one cohesion, one size, and one inheritance metric included to address the biggest areas of quality-influencing properties, as already suggested by Bär et al. in [3]. We further involve as many tools as possible. Maximizing the number of tools and metrics involved, we came to include 4 tools and 5 metrics. The tools are: Analyst4j, C&K Java Metrics, VizzAnalyzer, and Understand for Java. The metrics involved are: CBO, a coupling metric, LCOM-CK, a cohesion metric, NOM, a (interface) size metric, and DIT and NOC, inheritance metrics.

The composition of the quality model should not have a large influence on the results, as long as it is the same for each tool and project. The relations and weighting of metrics to criteria (Figure 2) can be seen arbitrarily.

Maintainability																				
1					1					1					1					
Analyzability					Changeability					Stability					Testability					
2	2	2	1	2	2	2	2	2	2	2	1	2	1	1	2	2	2	1	2
CBO	DIT	LCOM-CK	NOC	NOM	CBO	DIT	LCOM-CK	NOC	NOM	CBO	DIT	LCOM-CK	NOC	NOM	CBO	DIT	LCOM-CK	NOC	NOM

Figure 2: ISO 9126 based software quality model

The table can be interpreted in the following way: The factor Maintainability (first row) is described by its four criteria: Analyzability, Changeability, Stability and Testability to equal parts (weight 1, second row). The individual criteria (third row) are depending on the assigned metrics (last row) according to the specified weights (weight 1 or 2, fourth row). The mapping from the metrics values to the factors is by the percentage of classes being outliers according to the metrics values. Being a outlier means that the value is within the highest/lowest 15% of the value range defined by all classes in the system (self referencing model). Thus, the metrics values are aggregated and abstracted by the factors and the applied weights to the maintainability criteria, which describe the percentage of classes being outliers in the system, thus having bad maintainability. The value range is from 0.0 to 1.0 (0-100%), meaning that 0.0 is the best possible maintainability, since there are no outliers (metric values exceeding the given threshold relative to the other classes in the system), and 1.0 being the worst possible maintainability, since all metrics values for a class exceed their thresholds.

For example, if a class A has a value for CBO which is within the upper 15% (85%-100%) of the CBO values for all other classes in the system, and the other 4 metrics are not exceeding their thresholds, this class would have an Analyzability of 2/9, Changeability of 2/10, Stability of 2/7, and Testability of 2/9. This would result in a maintainability of 23.3% ((2/9+2/10+2/7+2/9)/4).

5. ASSESSMENT OF Q1/H1

5.1 Measurement and Data Collection

For collecting the data, we installed all 10 software metrics tools following the provided instructions. There were no particular dependencies or side effects to consider. Some tools provide a graphical user interface, some are stand-alone tools or plug-ins to an integrated development environment, others were command-line tools. For each of the tools being plug-ins to the Eclipse IDE we chose to create a fresh installation of the latest Eclipse IDE (3.3.1.1) to avoid confusing the different tools in the same Eclipse installation.

The test software systems were stored in a designated area so that all tools were applied on the same source code. In order to avoid unwanted modifications by the analyzing programs or measurement errors because of inconsistent code, we set the source code files to read-only, and we made sure that the software compiled without errors.

Once the tools were installed and the test software systems ready, we applied each tool to each system. We used the tool specific export features to generate intermediate files containing the raw analysis data. In most cases, the exported information contained the analyzed entities id plus the calculated attributes, which were the name and path to the class, and the corresponding metrics values. In most cases, the exported information could not be adjusted by configuring the metrics tools, so we had to filter the information prior to data analysis. Some tools also exported summaries about the metric values and other analysis results which we ignored. Most tools generated an HTML or XML report, others presented the results in tables, which could be copied or dumped into comma separated files. We imported the generated reports into MS Excel 2002. We stored the results for each test system in a separate Excel workbook, and the results for each tool in a separate Excel sheet. All this required mainly manual work.

All the tables containing the (raw) data have the same layout. The header specified the properties stored in each column *Class* and *Metrics*. *Class* stores the name of the class for which metrics have been calculated. We removed package information because it is not important, since there are no classes with the same name, and we could match the classes unambiguously to the sources. *Metrics* contains the metrics values calculated for the class as described in the previous section (CBO, DIT, LCOM-CK, LCOM-HS, LOC, NOC, NOM, TCC, WMC).

[14]http://sourceforge.net/projects/jtcgui
[15]http://sourceforge.net/projects/proguard

Tool	Data	CBO	DIT	LCOM-CK	LCOM-HS	LOC	NOC	NOM	RFC	WMC
Analyst4j	Max of Value	32.000	3.000	0.997			1.000	25.000	155.000	42.000
	Min of Value	4.000	1.000	0.800			0.000	6.000	12.000	10.000
	Average of Value	17.000	2.000	0.895			0.200	12.800	73.600	24.000
C&K Java Metrics	Max of Value	25.000	6.000	664.000			1.000	37.000	118.000	
	Min of Value	0.000	0.000	1.000			0.000	6.000	14.000	
	Average of Value	8.000	3.000	165.800			0.200	15.800	55.200	
CCCC	Max of Value	13.000	2.000				1.000	25.000		
	Min of Value	4.000	0.000				0.000	0.000		
	Average of Value	8.200	1.000				0.200	9.200		
Dependency Finder	Max of Value		2.000			231.000	1.000	45.000		
	Min of Value		1.000			30.000	0.000	6.000		
	Average of Value		1.200			108.200	0.200	19.600		
Eclipse Metrics Plugin 1.3.6	Max of Value		7.000		0.917		1.000	25.000		38.000
	Min of Value		1.000		0.000		0.000	6.000		9.000
	Average of Value		4.400		0.658		0.200	12.600		22.000
Eclipse Metrics Plugin 3.4	Max of Value			10.000	0.960					38.000
	Min of Value			0.000	0.000					9.000
	Average of Value			2.000	0.648					22.200
Semmle	Max of Value		4.000	323.000	0.980	314.000	1.000	25.000	109.000	
	Min of Value		1.000	7.000	0.839	50.000	0.000	6.000	8.000	
	Average of Value		2.600	97.000	0.903	150.400	0.200	12.600	57.400	
Understand For Java	Max of Value	32.000	7.000	96.000		407.000	1.000	25.000		
	Min of Value	5.000	1.000	73.000		59.000	0.000	6.000		
	Average of Value	**17.600**	4.400	82.200		200.600	0.200	12.800		
VizzAnalyzer	Max of Value	4.000	1.000	274.000		410.000	1.000	24.000	28.000	34.000
	Min of Value	0.000	0.000	4.000		64.000	0.000	5.000	6.000	8.000
	Average of Value	**1.000**	0.200	86.400		204.800	0.200	11.800	15.600	19.600

Figure 3: Differences between metrics tools for project jTcGUI

Tool	Data	CBO	DIT	LCOM-CK	LCOM-HS	LOC	NOC	NOM	RFC	WMC
Analyst4j	Max of Value	53.000	3.000	1.000			12.000	44.000	162.000	107.000
	Min of Value	0.000	1.000	0.000			0.000	2.000	2.000	2.000
	Average of Value	3.457	1.630	0.502			0.674	5.652	14.478	10.022
C&K Java Metrics	Max of Value	42.000	3.000	795.000			12.000	47.000	170.000	
	Min of Value	0.000	0.000	0.000			0.000	2.000	3.000	
	Average of Value	2.826	0.478	23.565			0.674	6.087	16.391	
CCCC	Max of Value	19.000	2.000				12.000	44.000		
	Min of Value	1.000	0.000				0.000	2.000		
	Average of Value	4.043	0.870				0.674	5.652		
Dependency Finder	Max of Value		2.000			341.000	12.000	47.000		
	Min of Value		1.000			5.000	0.000	2.000		
	Average of Value		1.674			28.130	0.674	6.609		
Eclipse Metrics Plugin 1.3.6	Max of Value		4.000		0.916		12.000	44.000		101.000
	Min of Value		1.000		0.000		0.000	0.000		2.000
	Average of Value		1.913		0.172		0.674	5.457		9.478
Eclipse Metrics Plugin 3.4	Max of Value			237.000	0.910					123.000
	Min of Value			0.000	0.000					1.000
	Average of Value			9.310	0.260					9.804
OOMeter	Max of Value	0.500	2.000	853.000			12.000			
	Min of Value	0.000	1.000	1.000			0.000			
	Average of Value	0.180	1.705	30.727			0.705			
Semmle	Max of Value		4.000	728.000	0.927	585.000	12.000	43.000	177.000	
	Min of Value		1.000	0.000	0.000	6.000	0.000	0.000	0.000	
	Average of Value		1.913	21.196	0.519	46.804	0.674	5.000	11.130	
Understand For Java	Max of Value	62.000	4.000	92.000		874.000	12.000	44.000		
	Min of Value	0.000	1.000	0.000		11.000	0.000	2.000		
	Average of Value	4.500	1.913	39.435		68.587	0.674	5.652		
VizzAnalyzer	Max of Value	39.000	1.000	536.000		882.000	12.000	42.000	91.000	81.000
	Min of Value	0.000	0.000	0.000		16.000	0.000	0.000	0.000	0.000
	Average of Value	2.630	0.674	16.370		73.239	0.674	4.478	7.587	7.152

Figure 4: Differences between metrics tools for project Jaim

5.2 Evaluation

Looking at some of the individual metrics values per class, it is easily visible that there are differences in how the tools calculate these values. For getting a better overview, we created pivot tables showing the average, minimum and maximum values per test system and metrics tool. If all tools would deliver the same values, we would get the same values. Looking at Figure 3, Figure 4, and Figure 5, we can recognize that there are significant differences for some metrics between some of the tools in all test systems.

Looking at *jTcGUI* (Figure 3), we see, that the average of the 5 classes of the system for the metric *CBO* varies between 1.0 as the lowest value (VizzAnalyzer) and 17.6 as the highest value (Understand for Java). This can be observed in a similar manner in the other two software systems. Thus, the tools calculate different values for these metrics. On the other hand, looking at the *NOC* metrics, we observe that all tools calculate the same values for the classes in this project. This can also be observe in *Jaim* (Figure 4), but not in *ProGuard* (Figure 5) where we observed some differences. *C&K Java Metrics* and *Dependency Finder* average to 0.440, *CCCC* to 1.495, *Eclipse Metrics Plug-in 1.3.6* to 0.480, *Semmle*, *Understand for Java*, *VizzAnalyzer* to 1.489.

Our explanation for the differences between the results for the CBO and the NOC metrics is that the CBO metrics is much more complex in its description, and therefore it is easier to implement variants of one and the same metric, which leads to different results. The NOC metrics is pretty straight forward to describe and to implement, thus the results are much more similar. Yet, this does not explain the differences in the *ProGuard* project.

Summarizing, we can reject our hypotheses H1 and our research questions Q1 should therefore be answered with: *Yes, there are differences between the metrics measured by different tools given the same input.*

Tool	Data	CBO	DIT	LCOM-CK	LCOM-HS	LOC	NOC	NOM	RFC	WMC
Analyst4j	Max of Value	76.000	4.000	2.000			121.000	116.000	458.000	412.000
	Min of Value	0.000	1.000	0.000			0.000	0.000	0.000	0.000
	Average of Value	6.780	1.482	0.461			**0.478**	9.036	25.036	19.928
C & K Java Metrics	Max of Value	89.000	7.000	6786.000			121.000	117.000	217.000	
	Min of Value	0.000	1.000	0.000			0.000	0.000	0.000	
	Average of Value	8.712	1.618	70.648			**0.440**	9.088	20.975	
CCCC	Max of Value	241.000	3.000				121.000	116.000		
	Min of Value	0.000	0.000				0.000	0.000		
	Average of Value	14.745	1.016				**1.495**	8.634		
Dependency Finder	Max of Value		4.000			1648.000	121.000	117.000		
	Min of Value		1.000			1.000	0.000	0.000		
	Average of Value		1.499			56.959	0.440	9.401		
Eclipse Metrics Plugin 1.3.6	Max of Value		7.000		1.023		121.000	116.000		395.000
	Min of Value		1.000		0.000		0.000	0.000		0.000
	Average of Value		1.674		0.190		**0.480**	8.500		18.587
Eclipse Metrics Plugin 3.4	Max of Value			346.000	1.000					359.000
	Min of Value			0.000	0.000					0.000
	Average of Value			4.172	0.203					18.065
Semmle	Max of Value		8.000	2702.000	1.000	2121.000	121.000	116.000	298.000	
	Min of Value		1.000	0.000	0.000	1.000	0.000	0.000	0.000	
	Average of Value		2.143	71.570	0.417	92.352	**1.489**	7.898	16.941	
Understand For Java	Max of Value	85.000	7.000	100.000		2836.000	121.000	116.000		
	Min of Value	0.000	0.000	0.000		1.000	0.000	0.000		
	Average of Value	9.405	1.546	34.875		143.624	**1.489**	8.143		
VizzAnalyzer	Max of Value	87.000	3.000	5009.000		2844.000	121.000	116.000	168.000	235.000
	Min of Value	0.000	0.000	0.000		1.000	0.000	0.000	0.000	0.000
	Average of Value	6.654	0.951	52.773		149.881	**1.489**	7.726	15.693	12.759

Figure 5: Differences between metrics tools for project ProGuard

5.3 Analysis

As shown in the previous section, there are a number of obvious differences among the results of the metrics tools. It would be interesting to understand why there are differences, i.e., what are the most likely interpretations of the metrics tool developers that lead to the different results (assuming that all results are intentional – not due to bugs in the tools). Therefore, we try to explain some of the differences found. For this purpose, we picked the class *TableModel* from the *jTcGUI* project. This class is small enough to manually apply the metrics definitions and variants thereof. We ignored TCC and LCOM-HS because they were only calculated by 2 respectively 3 tools. For the remaining 7 metrics and for each metrics tool, we give the metrics values (in parentheses) and provide our explanation.

Coupling metrics (CBO,RFC) calculate the coupling between classes. Decisive factors are the entities and relations in the scope and their types, e.g., class, method, constructor, call, access, etc. *Analyst4j* calculates for CBO 4 and RFC 12. These values can be explained by API classes being part of the scope. These are all imported classes, excluding classes from java.lang (String and Object). Constructors count as methods, and all relations count (including method and constructor invocations). *Understand for Java* and *CCCC* calculate CBO 5 and 8, resp. It appears to be the same as for *Analyst4j*, but they seem to include both `String` and `Object` as referenced classes. Additionally, *CCCC* also seems to include primitive types `int` and `long`. *C&K Java Metrics* calculates CBO 1 and RFC 14. This value can be explained if the API classes are *not* in the scope. This means that only the coupling to source class `TrueCrypt` is considered. On the other hand, for a RFC of 14, the API classes as well as the default constructor, which is present in the byte code analyzed, need to be included. *Semmle* calculates RFC 8. This value can be explained if the API is *not* in scope, and if the constructor is also counted as a method. *VizzAnalyzer* calculates CBO 1 and RFC 6, meaning that the API is *not* in scope, and the constructor does not count as a method.

Cohesion metrics (LCOM) calculate the internal cohesion of classes. Decisive factors are the entities and relations within the class and their types, e.g., method, constructor, field, invokes, accesses,etc. *Analyst4j*, *C&K Java Metrics*, *Eclipse Metric Plug-in 3.4*, and *Understand for Java* calculate LCOM-CK 0.8, 1.0, 0, and 73, resp. We cannot explain how these values are calculated. *Understand for Java* calculates some kind of percentage. *Semmle* calculates LCOM-CK 7. This does not match our interpretation of the metric definition provided by the tool vendor, and we cannot explain how this value is calculated. *VizzAnalyzer* calculates LCOM-CK 4. This value can be explained if the API is *not* in scope; and LCOM is calculated as number of method pairs not sharing fields minus number of method pairs sharing fields considering unordered method pairs.

Inheritance metrics (DIT) quantify the inheritance hierarchy of classes. Decisive factors are the entities and relations in the scope and their types, e.g., class, interface, implements, extends, etc. *Analyst4j*, *C&K Java Metrics*, *Eclipse Metrics Plug-in 1.3.6*, *Semmle*, and *Understand for Java* calculate DIT 2. These values can be explained if the API classes (Object and AbstractTableModel) are in scope, starting counting at 0 at Object and calculating DIT 2 for TableModel, which is source code. *CCCC* and *Dependency Finder* calculate DIT 1. These values can be explained if the API classes are *not* in scope, starting counting with 1 (TableModel, DIT 1). *VizzAnalyzer* calculates DIT 0. This value can be explained if the API classes are *not* in scope, starting counting with 0 (TableModel, DIT 0).

Size and Complexity metrics (LOC,NOM,WMC) quantify structural and textual elements of classes. Decisive factors are the entities and relations in the scope and their types, e.g., source code, class, method, loops and conditions, contains relations, etc. The compilation unit implementing the class TableModel has 76 lines. *Dependency Finder* calculates LOC 30. This can be explained if it counts only lines with statements, i.e., field declarations, and method bodies, from the beginning of the class declaration (line 18) to the end of the class declaration (closing }, line 76), excluding method declarations or any closing }. *Semmle* calculates LOC 50. This can be explained if it counts non-empty lines from the beginning of the class declaration (line 18) to the end of the class declaration (closing }). *Understand for Java* calculates LOC 59, meaning it counts all lines from line 18 to 76. *VizzAnalyzer* calculates LOC 64 and thus counts from

line 13 (class comment) to line 76, i.e., the full class declaration plus class comments.

Analyst4j, *C&K Java Metrics*, *CCCC*, *Dependency Finder*, *Eclipse Metrics Plug-in 1.3.6*, *Semmle*, *Understand for Java* all calculate NOM 6. The values can be explained if all methods and constructors are counted. *VizzAnalyzer* calculates NOM 5, thus it counts all methods excluding constructors.

Analyst4j calculates WMC 17. We cannot explain it, but we assume it includes constructors and might count each *if* and *else*. *VizzAnalyzer*, *Eclipse Metrics Plug-in 3.4* and *Eclipse Metrics Plug-in 1.3.6* calculate WMC 13, 15 and 14, resp. These values can be explained when they include constructor (not VizzAnalyer) and count 1 for every method, if, do, for, while, and switch. *Eclipse Metrics Plug-in 3.4* might count, in addition, the *default* statements.

Although we cannot exclude bugs in the tools, we recognized two main reasons for differences in the calculated values: First, the tools operate on different scopes, that is, some consider only the source code, others include the surrounding libraries or APIs. Second, there are differences in how metrics definitions are interpreted, e.g., some tools count constructors as methods, others do not; some start counting with 1, others with 0; some express values as percentage, others as absolute values, etc.

6. ASSESSMENT OF Q2/H2

In Section 5.2, we answered our first research question with *yes*. We now proceed with answering research question Q2: *are the observed differences really a problem?*

6.1 Measuring and Data Collection

Obviously, we can reuse the data collected by the metrics tools and the metrics and systems from stage one of our case study as input to our client analysis (see Section 4.4). We just add new columns for the factors and criteria of the software quality model and sort according to maintainability. If several classes receive the same value, we sort using the CBO and LCOM-CK values as the second and third sorting criteria. For jTcGUI, we select all 5 classes for comparison, for Jaim and ProGuard, we select the "top 10" classes.

6.2 Evaluation and Analysis

The "top 10 (5)" classes identified by the different tools in each project show tool dependent differences. Figures 6, 7, and 8 present the results as tables. Since there is no correct ranking or "gold standard", we compared each tool with all other tools. Once more, there is no "right or wrong", we just observe differences in the rankings due to the different input metrics values computed by the different metrics tools.

Figure 6, 7, and 8 describe the "top 5 or 10" classes for jTcGUI, Jaim and ProGuard as selected/ranked, based on the metrics data collected by each tool. *Rank* describes the order of the classes as described in the previous section. I.e., Rank 1 has the lowest maintainability (highest maintainability, CBO, and LCOM-CK value), Rank 2 the second lowest, and so on. The *Code* substitutes the class names with letters a-z for easier reference. The names of the classes are presented next to the substitution code. The first row is labeled with the tool name and sort reference.

Looking at Figure 6, we can recognize some small variations in the ranking for jTcGUI. Tool A and D get the same result. Tool B and C get the same result, which is slightly different from the ranking proposed by Tools A and D.

Tools	Code 1	Code 2	*Distance*	Disjunct (%)
A-B	**abcde**	**acbed**	**2**	**0%**
A-C	**abcde**	**acbed**	**2**	**0%**
A-D	abcde	abcde	0	0%
B-A	**acbed**	**abcde**	**2**	**0%**
B-C	acbed	acbed	0	0%
B-D	**acbed**	**abcde**	**2**	**0%**
C-A	**acbed**	**abcde**	**2**	**0%**
C-B	acbed	acbed	0	0%
C-D	**acbed**	**abcde**	**2**	**0%**
D-A	abcde	abcde	0	0%
D-B	**abcde**	**acbed**	**2**	**0%**
D-C	**abcde**	**acbed**	**2**	**0%**

Figure 9: Distance between rankings, jTcGUI

To further analyze this observation, we use the "Code" for each class to form a string describing the ranking of the classes. Thus, "abcde" corresponds to the ranking "Gui, TrueCryptGui, Password, TrueCrypt, and TableModel". In the context of the client analysis, this means that one should start refactoring the class with the lowest maintainability, which is "Gui", then "TrueCryptGui", etc. Using these substitution strings, we can easily compare them and describe their difference as numeric values, i.e., as edit distance and disjunct sets. We selected the Damerau-Levenshtein Distance [4, 7, 23] for expressing the edit distance between two strings, thus quantifying differences in the ranking over the same classes. A value of 0 means the strings are identical, a value larger than 0 describes the number of operations necessary to transform one string into another, and thus the difference the two provided strings in our case the order of the given classes. The higher the value, the more different are the calculated rankings. The maximum edit distance is the length of the strings in our cases 5 or 10, meaning that compared sets of classes have almost nothing in common regarding contained classes or order. We also measure how disjunct the provided rankings are as the percentage of classes which the two rankings do not have in common. More formally, c is the number of classes which are in both sets being compared (ranking 1 and ranking 2), and n is the number of classes which they can have possibly in common. $Disjunct = (1 - (c/n)) \times 100\%$.

Figures 9, 10, and 11 provide an overview of the differences between the rankings provided by the four tools per project. For jTcGUI (Figure 9), we observe just small differences in the ranking of the classes. The biggest differences (Damerau-Levenshtein Distance of 2) are between the tools having a distance value of 2. The disjunct set is always 0%, since all classes of the system are considered.

Tools	Code 1	Code 2	*Distance*	Disjunct (%)
A-B	abcdefghij	abklmnofpq	8	70%
A-C	abcdefghij	arsjituvbw	9	60%
A-D	abcdefghij	abdcghijtv	5	20%
B-A	abklmnofpq	abcdefghij	8	70%
B-C	**abklmnofpq**	**arsjituvbw**	**9**	**80%**
B-D	abklmnofpq	abdcghijtv	8	80%
C-A	arsjituvbw	abcdefghij	9	60%
C-B	**arsjituvbw**	**abklmnofpq**	**9**	**80%**
C-D	arsjituvbw	abdcghijtv	9	40%
D-A	abdcghijtv	abcdefghij	5	20%
D-B	abdcghijtv	abklmnofpq	8	80%
D-C	abdcghijtv	arsjituvbw	9	40%

Figure 10: Distance between rankings, Jaim

For Jaim (Figure 10), we observe much bigger differences in the rankings of the classes. The biggest differences are between the tools having a distance value of 9 and a disjunct set of 80%. Since the system has 46 classes of which

Tool A - Analyst4j		
Rank	Code	Classes
1	a	Gui
2	b	TrueCryptGui
3	c	Password
4	d	TrueCrypt
5	e	TableModel

Tool B - VizzAnalyzer		
Rank	Code	Classes
1	a	Gui
2	c	Password
3	b	TrueCryptGui
4	e	TableModel
5	d	TrueCrypt

Tool C - C&K Java Metrics		
Rank	Code	Classes
1	a	Gui
2	c	Password
3	b	TrueCryptGui
4	e	TableModel
5	d	TrueCrypt

Tool D - Understand for Java		
Rank	Code	Classes
1	a	Gui
2	b	TrueCryptGui
3	c	Password
4	d	TrueCrypt
5	e	TableModel

Figure 6: Ranking of jTcGUI classes according maintainability per tool

Tool A - Analyst4j		
Rank	Code	Classes
1	a	JaimConnection
2	b	ConfigTocResponse
3	c	FLAPFrame
4	d	BuddyUpdateTocResponse
5	e	Utils
6	f	TocSignonCommand
7	g	JaimStateException
8	h	JaimTimeoutException
9	i	TocResponse
10	j	TocCommand

Tool B - VizzAnalyzer		
Rank	Code	Classes
1	a	JaimConnection
2	b	ConfigTocResponse
3	k	TocSetConfigCommand
4	l	GenericTocResponse
5	m	IMTocResponse
6	n	NickTocResponse
7	o	SignOnTocResponse
8	f	TocSignonCommand
9	p	TocAddBuddyCommand
10	q	TocAddDenyCommand

Tool C - C&K Java Metrics		
Rank	Code	Classes
1	a	JaimConnection
2	r	FLAPFrameException
3	s	JaimException
4	j	TocCommand
5	i	TocResponse
6	t	JaimTest
7	u	FLAPFrameFactory
8	v	ReceiverThread
9	b	ConfigTocResponse
10	w	DeliveryThread

Tool D - Understand for Java		
Rank	Code	Classes
1	a	JaimConnection
2	b	ConfigTocResponse
3	d	BuddyUpdateTocResponse
4	c	FLAPFrame
5	g	JaimStateException
6	h	JaimTimeoutException
7	i	TocResponse
8	j	TocCommand
9	t	JaimTest
10	v	ReceiverThread

Figure 7: Ranking of Jaim classes according maintainability per tool

we include 10 in our "top 10", it is possible that not only the order changes, but that other classes are considered in comparison to other tools. Recognizable is that all metrics tools elect the same least maintainable class, *JaimConnection*.

For ProGuard (Figure 11), we again observe differences in the rankings of the classes. The biggest differences are between the tools having a distance value of 10 and a disjunct set of 70%. Since the system has 486 classes of which we include 10 in our "top 10", it is possible that not only the order changes, but that other classes are considered in comparison to other tools. Notable is that three of the four metrics tools select the same least maintainable class, *SimplifiedVisitor*. *Understand for Java* ranks it second.

Tools	Code 1	Code 2	*Distance*	Disjunct (%)
A-B	abcdefghij	abdiklmjno	8	50%
A-C	abcdefghij	abedpqrstc	7	50%
A-D	abcdefghij	caeuvwbdqr	10	50%
B-A	abdiklmjno	abcdefghij	8	50%
B-C	abdiklmjno	abedpqrstc	8	70%
B-D	**abdiklmjno**	**caeuvwbdqr**	**10**	**70%**
C-A	abedpqrstc	abcdefghij	7	50%
C-B	abedpqrstc	abdiklmjno	8	70%
C-D	abedpqrstc	caeuvwbdqr	9	30%
D-A	caeuvwbdqr	abcdefghij	10	50%
D-B	**caeuvwbdqr**	**abdiklmjno**	**10**	**70%**
D-C	caeuvwbdqr	abedpqrstc	9	30%

Figure 11: Distance between rankings, ProGuard

Précising, we found differences in the order and composition of classes elected to be least maintainable for all four tools in all three projects. The differences between the tool pairs varied, but especially in the larger projects are they significant. Regarding our fictive task, the software engineers and managers would have been presented with different sets of classes to focus their efforts on. We can only speculate about the consequences of such tool-dependent decisions.

Summarizing, we can reject our hypotheses H2 and our research questions Q2 should therefore be answered with: *Yes, it does matter and might lead to different conclusions.*

7. VALIDITY EVALUATION

We have followed the design and methods recommended by Robert Yin [35]. For supporting the validity, we now discuss possible threats to:

Construct Validity is about establishing correct operational measures for the concepts being studied. To ensure construct validity, we assured that there are no other varying factors than the software metrics tools, which influence the outcome of the study. We selected an appropriate set of metrics and brought only those metrics into relation where we had a high confidence that other experienced software engineers or researchers would come to the same conclusion, given that metrics expressing the same concept might have different names. We assured that we ran the metrics tools on identical source code. Further, we assumed that the limited selection of three software projects of the same programming language posses still enough statistical power to generalize our conclusions. We randomized the test system selection.

Internal Validity is about establishing a causal relationship, whereby certain conditions are shown to lead to certain other conditions, as distinguished from spurious relationships. We believe that there are no threats to internal validity, because we did not try to explain causal relationships, but rather dealt with an exploratory study. The possibility for interfering was limited in our setting. There were no human subjects which could have been influenced, which could have led to different results depending on the time or person of the study. The influence on the provided test systems and the investigated software metrics tools was limited. The variation points like data extraction and analysis allowed only for very small room for changes.

External Validity deals with the problem of knowing if our findings are generalizable beyond the immediate case study. We included the most obvious software metrics tools available on the internet. These should represent a good deal of tools used in practice. We are aware that there is likely a much larger body of tools, and many companies might have developed their own tools. It was necessary to greatly reduce the number of tools and metrics considered in order to obtain results that could allow for reasonable comparisons. Four tools and five metrics applied to three different systems is frankly spoken not very representative for the space of possibilities. Yet, we think the selection and problems uncovered are representative enough to indicate a general problem, which should stimulate additional research including tests of statistical significance. The same holds for the selection of software projects measured. We see no reason why other projects should allow for different conclusions than the three systems we analyzed, and the programming language should have no impact. The selected metrics could include a potential threat. As we have seen in Section 5, some metrics, like NOC, tend to be rather stable over the used tools. We only investigated object-oriented metrics. Other metrics, like the Halstead metrics [10] implemented by some of the tools, might behave differently. Yet, object-

Tool A - Analyst4j		
Rank	Code	Classes
1	a	SimplifiedVisitor
2	b	ProgramClassReader
3	c	ProGuardGUI
4	d	Optimizer
5	e	ClassPrinter
6	f	ConstantValueAttribute
7	g	SourceDirAttribute
8	h	SourceFileAttribute
9	i	SpecificDoubleValue
10	j	ProgramMethod

Tool B - VizzAnalyzer		
Rank	Code	Classes
1	a	SimplifiedVisitor
2	b	ProgramClassReader
3	d	Optimizer
4	i	SpecificDoubleValue
5	k	SpecificIntegerValue
6	l	SpecificLongValue
7	m	ProgramField
8	j	ProgramMethod
9	n	SpecificFloatValue
10	o	LibraryField

Tool C - C&K Java Metrics		
Rank	Code	Classes
1	a	SimplifiedVisitor
2	b	ProgramClassReader
3	e	ClassPrinter
4	d	Optimizer
5	p	KeepSpecificationsPanel
6	q	ConstantPoolRemapper
7	r	UsageMarker
8	s	Obfuscator
9	t	Utf8UsageMarker
10	c	ProGuardGUI

Tool D - Understand for Java		
Rank	Code	Classes
1	c	ProGuardGUI
2	a	SimplifiedVisitor
3	e	ClassPrinter
4	u	FilterDialog
5	v	ClassPathPanel
6	w	MemberSpecificationsPanel
7	b	ProgramClassReader
8	d	Optimizer
9	q	ConstantPoolRemapper
10	r	UsageMarker

Figure 8: Ranking of ProGuard classes according maintainability per tool

oriented metrics are among the most important metrics in use nowadays. The imaginary task and the software quality model used for abstracting the metrics values could be irrelevant in practice. We spent quite some thought on defining our fictive task, and considering the experiences we had, e.g., with Eurocontrol, and the reengineering tasks described by Bär et al in the FAMOOS Handbook of Re-engineering [3], we consider it as quite relevant. The way we applied software quality models is nothing new, it has been described in one or another form in literature [21, 19, 22, 8, 20].

Reliability assures that the operations of a study – such as the data collection procedures – can be repeated yielding the same results. The reliability of a case study is important. It shall allow a later investigator to come to the same findings and conclusions when following the same procedure. We followed a straight forward design, thus simplicity should support reliability. We documented all important decisions and intermediate results, like the tool selection, the mapping from the tool specific metrics names to our conceptual metrics names, as well as the procedures for the analysis. We minimized our impact on the used artifacts and documented any modifications. We described the design of the experiments including the subsequent selection process.

8. CONCLUSION AND FUTURE WORK

Software engineering practitioners – architects, developers, managers – must be able to rely on scientific results. Especially research results on software quality engineering and metrics should be reliable. They are used during forward-engineering, to take early measures if parts of a system deviate from the given quality specifications, or during maintenance, to predict effort for maintenance activities and to identify parts of a system needing attention.

In order to provide these reliable scientific results, quite some research has been conducted in the area of software metrics. Some of the metrics have been discussed and reasoned about for years, but only few metrics have even been validated experimentally to have correlations with certain software qualities, e.g., maintainability [24]. Refer to [25] for an overview of software quality metrics and quality models.

Moreover, software engineering practitioners should be able to rely on the tools implementing these metrics, to support them in quality assessment and assurance tasks, to allow to quantify software quality, and to deliver the information needed as input for their decision making and engineering processes. Nowadays a large body of software metrics tools exists. But these are not the tools which have been used to evaluate the software metrics. In order to rest on the scientific discussions and validations, i.e., to safely apply the results and to use them in practice, it would be necessary that all metrics tools implement the suggested metrics the way they have been validated.

Yet, we showed that metrics tools deliver different results given the same input and, hence, at least some tools do not implement the metrics as intended. Thus, we collected output for a set of nine metrics calculated by ten different metric tools on the same three software systems. We found that, at least for these investigated software metrics, tool-dependent differences exist. Still, for certain metrics, the tools delivered similar results. For rather simple metrics, like the Number of Children (NOC), most tools computed the same or very similar results. For other metrics, e.g., the Coupling Between object Classes (CBO) or Lack of Cohesion of Methods (LCOM), the results showed a much bigger variation. Overall, we can conclude that most tools provided different results for the same metrics on the same input.

In an attempt to explain our observations, we carefully analyzed the differences for selected classes and found (in most cases) reasonable explanations. Variations in the results were often related to different scopes that metrics were applied to and differences in mapping the extracted programming language constructs to a meta-model used in measurement. E.g., the tools in- or excluded library classes or inherited features in their measurements. Hence, it could be concluded that metrics definitions should include exact scope and language mapping definitions.

Minor differences in the metrics values would not be a problem if the interpretation of the values led to the same conclusions, i.e., if software engineering practitioners would be advised to act in a similar way. Since interpretation is an abstraction, this could still be possible. Actually, our assumption was that the differences observed in metrics values would be irrelevant after this abstraction.

To confirm our assumption, we defined a client analysis, which abstracted from the metrics values using a software quality model. The resulting maintainability values were interpreted to create a ranking among the measured classes. Software engineers could have been advised to attend to these classes according to their order. We found that even after abstraction, the two larger projects showed considerable differences in the suggested ordering of classes. The lists of the top 10 ranked classed differed up to 80% for some tool pairs and the same software systems.

Our final conclusions are that, from a *practical point of view*, software engineers need to be aware that the metrics results are tool dependent, and that these differences change the advice the results imply. Especially, metrics based results cannot be compared when using different metrics tools. From a *scientific point of view*, validations of software metrics turn out to be even more difficult. Since metrics results are strongly dependent on the implementing tools, a validation only supports the applicability of some metrics as implemented by a certain tool. More effort would be needed in specifying the metrics and the measurement process to make the results comparable and generalizable. Regarding future

work, more case studies should repeat our study for additional metrics, e.g., Halstead metrics [10], and for further programming languages. Moreover, a larger base of software systems should be measured to increase the practical relevance of our results. Additionally, an in-depth study should seek to explain the differences in the measurement results, possibly describing the metrics variants implemented by the different tools. Further more, with the insights gained, metrics definition should be revised.

Finally, we or other researchers should revise our experimental hypotheses, which have been stated very narrowly. We expected that all the tools provide the same metrics values and same results for client analyses, so that they can be literally interpreted in such a way that they do not require tests of statistical significance. Restating the hypotheses to require such tests, in order to get a better sense of how bad the numbers for the different tools really are, is additional future work supporting the generalization of our results.

9. ACKNOWLEDGMENTS

We would like to thank the following companies and individuals for kindly supplying us with evaluation licenses for the tools provided by their companies: CodeSWAT Support for *Analyst4j*. Oliver Wihler, Aqris Software AS, for *RefactorIT*, even though the tool could not be made available in time. Rob Stuart, Customer Support M Squared Technologies, for *Resource Standard Metrics Tool (Java)*. Olavi Poutannen, Testwell Ltd, for *CMTJava*. ARiSA AB for the *VizzAnalyzer tool*. We also thank our colleague Tobias Gutzmann for reviewing our paper.

10. REFERENCES

[1] J. Alghamdi, R. Rufai, and S. Khan. Oometer: A software quality assurance tool. *Software Maintenance and Reengineering, 2005. CSMR 2005. 9th European Conference on*, pages 190–191, 21-23 March 2005.

[2] Aqris software. http://www.aqris.com/.

[3] H. Bär, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, M. Przybilski, T. Richner, M. Rieger, C. Riva, A. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, and J. Weisbrod. The FAMOOS Object-Oriented Reengineering Handbook, Oct. 1999.

[4] G. V. Bard. Spelling-error tolerant, order-independent pass-phrases via the damerau-levenshtein string-edit distance metric. In *ACSW '07: Proc. of the 5th Australasian symposium on ACSW frontiers*, pages 117–124, Darlinghurst, Australia, 2007. ACS, Inc.

[5] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[6] Clarkware consulting inc. http://www.clarkware.com/.

[7] F. Damerau. A technique for computer detection and correction of spelling errors. *Comm. of the ACM*, 1964.

[8] R. G. Dromey. Cornering the Chimera. *IEEE Softw.*, 13(1):33–43, 1996.

[9] EUROCONTROL. Overall Target Architecture Activity (OATA). http://www.eurocontrol.be/oca/public/standard_page/overall_arch.html, Jan 2007.

[10] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[11] hello2morrow. http://www.hello2morrow.com/.

[12] B. Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[13] W. S. Humphrey. *Introduction to the personal software process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[14] hypercision inc. ,http://hypercision.com/.

[15] instantiations inc. http://www.instantiations.com/.

[16] ISO. ISO/IEC 9126-1 "Software engineering - Product Quality - Part 1: Quality model", 2001.

[17] ISO. ISO/IEC 9126-3 "Software engineering - Product Quality - Part 3: Internal metrics", 2003.

[18] Andrew cain. http://www.it.swin.edu.au/projects/jmetric/products/jmetric/default.htm.

[19] E.-A. Karlsson, editor. *Software Reuse: A Holistic Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[20] N. Kececi and A. Abran. Analysing, Measuring and Assessing Software Quality In a Logic Based Graphical Model, 2001. QUALITA 2001, Annecy, France, 2001, pp. 48-55.

[21] B. Laguë and A. April. Mapping of Datrix(TM) Software Metrics Set to ISO 9126 Maintainability Sub-Characteristics, October 1996. SES '96, Forum on Software Eng. Standards Issues, Montreal, Canada.

[22] Y. Lee and K. H. Chang. Reusability and Maintainability Metrics for Object-Oriented Software. In *ACM-SE 38: Proc. of the 38th annual on Southeast regional conference*, pages 88–94, 2000.

[23] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 1966.

[24] W. Li and S. Henry. Maintenance Metrics for the Object Oriented Paradigm. In *IEEE Proc. of the 1st Int. Sw. Metrics Symposium*, pages 52–60, May 1993.

[25] R. Lincke. *Validation of a Standard- and Metric-Based Software Quality Model – Creating the Prerequisites for Experimentation*. Licentiate thesis, MSI, Växjö University, Sweden, Apr 2007.

[26] R. Lincke and W. Löwe. Compendium of Software Quality Standards and Metrics. http://www.arisa.se/compendium/, 2005.

[27] J. A. McCall, P. G. Richards, and G. F. Walters. Factors in Software Quality. Technical Report Vol. I, NTIS Springfield, VA, 1977. NTIS AD/A-049 014.

[28] M squared technologies. http://www.msquaredtechnologies.com/.

[29] Power software. http://www.powersoftware.com/.

[30] Semantic designs inc. http://www.semdesigns.com/.

[31] W. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, May 1998.

[32] Verifysoft technology. http://www.verifysoft.com/.

[33] Virtual machinery. http://www.virtualmachinery.com/.

[34] A. H. Watson and T. J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. *NIST Special Pub. 500-235*, 1996.

[35] R. K. Yin. *Case Study Research : Design and Methods (Applied Social Research Methods)*. SAGE Publications, December 2002.