Quantitative Evaluation of Software Quality Metrics in Open-Source Projects

Henrike Barkmann Rüdiger Lincke Welf Löwe

Software Technology Group, School of Mathematics and Systems Engineering
Växjö University, 351 95 Växjö, Sweden
hbaex07@student.vxu.se, {rudiger.lincke | welf.lowe}@vxu.se

Abstract

The validation of software quality metrics lacks statistical significance. One reason for this is that the data collection requires quite some effort. To help solve this problem, we develop tools for metrics analysis of a large number of software projects (146 projects with ca. 70.000 classes and interfaces and over 11 million lines of code). Moreover, validation of software quality metrics should focus on relevant metrics, i.e., correlated metrics need not to be validated independently. Based on our statistical basis, we identify correlation between several metrics from well-known object-oriented metrics suites. Besides, we present early results of typical metrics values and possible thresholds.

1. Introduction

Quality assurance and management processes and practices, e.g., the ISO 9000 family [10] and CMMI [17], actively help to improve the quality of software and its development. They depend on qualitative and quantitative data for identifying problems and controlling the success of the applied measures. Metrics of software attributes provide such quantitative data, and various metrics suites have been suggested, e.g., [5, 6]. Software Quality Models like ISO 9126 [9] integrate different metrics.

Validation of software metrics shows that metrics actually allow conclusions on the quality of software [2, 4, 7, 11]. Most studies conclude that metrics are indeed a valid indicator for quality of software, defect detection, maintainability, etc. Most scientific studies conclude with *qualitative* correlations, i.e., high values seem to indicate problems for certain metrics, while low values are desirable [1].

On the other hand, there are metrics tools, cf. [15] for an overview, and practical recommendations, e.g., the NASA database of software quality metrics [16], that suggest *quantitative* thresholds of some metrics. These thresholds are based on experience but not validated scientifically.

Collecting metrics and quality for validation studies requires quite some effort. One of the biggest case studies we know of is described in the "FAMOOS Handbook of Reengineering" [1]. It involves 6 projects with about 5.1 million lines of code. Yet, quantitative values as part of an empirical study are presented for just three projects with a total of about 300 classes measured with 7 metrics.

Hence, metrics-based quality management processes usually define project-relative thresholds. They find relative outliers, i.e., system parts with metric values extremely large or small when related to the average values in the whole system. Moreover, they find negative tendencies over the development process, i.e., system parts with metric values increasing (decreasing) over time while small (high) values actually correlate with good quality. However, project-relative thresholds are critical unless the majority of system parts is in good quality.

Our ultimate goal is therefore to find and validate absolute thresholds for (at least some) quality metrics. In order to improve validation efficiency, this paper contributes with:

(i) Tools which complement existing software metrics tools to allow the automated analysis of a large number of (open-source) software projects. This allows to extract the metrics values for a quantitative evaluation.

(ii) Identification of dependent metrics, i.e., pairs of metrics that are statistically correlated. Thus, the focus of the validation is reduced to fewer independent metrics.

We apply our tools (i) to analyze 146 projects with 70.000 classes and interfaces and over 11 million lines of code, and use this as the statistical basis for our correlation analysis (ii). As a side effect, the metrics collected allow to provide early results of the typical distribution of metrics values and possible absolute metric thresholds.

The structure of the paper is as following. Section 2 introduces the tools and processes developed. Section 3 discusses the dependent metrics experiment, including the hypothesis, the experimental setup, and assessment of the hypothesis. In Section 4, we describe the problems experienced during our research. We conclude our paper in Section 5.

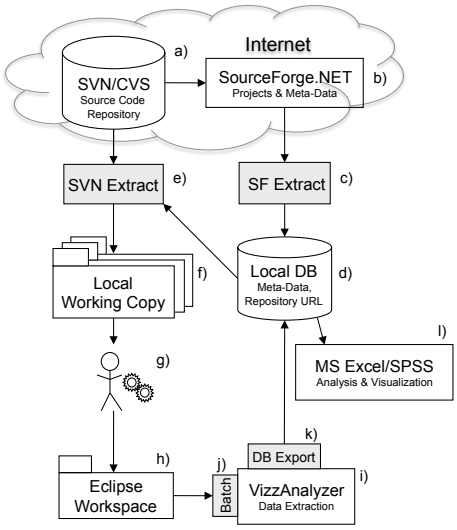

Figure 1. Tools and processes used.

2. Tools and processes

Figure 1 provides an overview of the tools and processes developed. It is built around an open-source project repository – SVN/CVS and SourceForge (a, b), an IDE for extracting basic information about the projects – Eclipse (h), a metrics tool for computing metrics values – VizzAnalyzer (i), a local database storing the data – MS Access (d), and tools for the statistical analysis – MS Excel and SPSS (l).

First, we extract the source code (e) and associated meta-data (c) from the SVN and CVS repositories (a) of SourceForge.NET[1] projects (b). The meta-data is visible as attributes on the project home page of the respective projects. Manually collecting this information for almost 150 projects is expensive. We therefore developed *SF Extract*, which can connect to SourceForge, read the list of all Java projects available, follow the links to each project home page, parse the home page (HTML) to extract the relevant properties, and store them in the database (d) for further processing. Since the meta-data contains also information about the url of the SVN or CVS repository containing the source code (b), we could develop *SVN Extract*, which downloads the source code of these projects based on the information in the database in an automated and efficient way. The files and folders of the individual projects in the repository are stored in local working copies (f).

To assure that the downloaded projects are complete and compile, we import them *manually* (g) into an Eclipse workspace (h) as Java projects. The Eclipse compiler notified us about build problems and compile errors. We then satisfy the missing dependencies by searching the proper libraries. In some cases, we can use the existing build documentation and scripts to find out what is needed, while in other cases, we can use the maven project description to download the missing dependencies. In some cases, we have to try our luck by searching the internet for the library in the correct version. So far, we could not find a way to automate this step in our process, so it takes the better part of our time. However, we can exclude this manual step in our process at the price of ignoring projects that do not compile directly.

The VizzAnalyzer[2] metrics tool (i) is fed with low-level information from the Eclipse project (syntax, cross references, etc.) and computes the metrics. Since the VizzAnalyzer is designed for interactive work and analyzes only a single project at a time, we had to extend it with an interface (j) allowing for batch-processing of a list of projects. Further, we implemented an export engine (k) to store the computed metrics in a database for later processing. For the actual statistical analysis of the metrics, we use MS Excel and SPSS[3] (l) both with access to the database (d).

3. Dependent metrics

This section documents the experiments we conduct in order to find statistically dependent and independent metrics. It also demonstrates the practical value of the tools developed. Furthermore, we present some intermediate results on metrics, e.g., histograms for selected metrics preceding future research on quantitative thresholds.

For all pairs of metrics (considered in this study) we aim at *invalidating* either of the following hypothesis:

H_0 The pair of metrics values is independent in all software systems (considered in the study).

H_1 The pair of metrics values is dependent, that is, showing a statistically significant correlation between the measured values in all software systems (considered in the study).

The objects of our study are metric values and their distributions calculated for every class in each of the software systems analyzed. The purpose is to test for independence of measured metrics values. We take a researchers perspective who tries to improve efficiency in (later) validations of software quality metrics. The main effects under study are metric values and their correlations. The set of tools, metrics, and software systems for analysis are selected randomly but determined by practical considerations.

3.1. Experimental setup

Our experiment is performed on standard equipment, i.e., one PC satisfying the minimum requirements for our tools, cf. Section 2, and a 10 MBit internet connection. All measures are performed and analyzed on this computer, on which the downloaded source code and the extracted data is stored as well.

[1] http://sourceforge.net – in future referred to as SourceForge.

[2] http://www.arisa.se

[3] http://www.spss.com

Practical constrains – Ideally, we would measure all known software metrics in all existing software systems, or at least randomly select the set of metrics and test systems. But, the selection is constrained by practical considerations. A detailed presentation of the selection of projects and metrics is provided further down in this section. Here we discuss the selection process and the practical constraints.

First, there are way to many software metrics described, and many of them are not well enough defined to implement them in an unambiguous way [13, 15]. Hence, we limit ourselves to the set of metrics which appear practically interesting to us and which are well-defined and implemented in the VizzAnalyzer. For our study, we focus on class metrics and skip metrics for packages and methods, since they are not needed for answering our research questions.

Second, further limitations apply to the software systems analyzed. We obviously cannot analyze all existing systems. Additionally, legal restrictions limit the number of suitable systems, and our software metrics tool is best tested on Java programs. Therefore, we restrict ourselves to open-source Java software that is legally available on SourceForge. We prefer programs with a high SourceForge-ranking, since we assume that these programs have a larger user base, and, hence relevance.

Finally, many projects do not compile after download (about 30%, cf. Section 4 for details), and can thus not be analyzed right away. Most of them compile after manual fixes. Still, some of them remain not analyzable and are therefore excluded.

Software project selection – We select 146 open-source Java projects randomly apart from the practical constraints. We have got a large variety of projects from different categories in the SourceForge classification. Since it is not possible to list all 146 projects included in the experiment, we provide a complete list of the projects together with some statistics (e.g., number of classes, lines of code, etc.) at the following url: http://w3.msi.vxu.se/~rle/research/quest2009/projects.html.

Software quality metric selection – The software metrics selected are a collection of the most popular metrics discussed in literature. They are taken from different well know metrics suites like *Chidamber & Kemerer* [5] – Coupling Between Objects (CBO), Depth of Inheritance Tree (DIT), Lack of Cohesion in Methods (LCOM), Number Of Children (NOC), Response For a Class (RFC), Weighted Method Count (WMC) using McCabe Cyclomatic Complexity as weight for the methods; *Li & Henry* [12] – Data Abstraction Coupling (DAC), Message Passing Coupling (MPC), Number Of local Methods (NOM), Number of Attributes and Methods (NAM/SIZE2); *Bieman & Kang* [3] – Tight Class Cohesion (TCC); *Hitz & Montazeri* [8] – Locality of Data (LD), Improvement of LCOM (ILCOM); and the metrics Lines Of Code (LOC), Lack Of Documentation (LOD), Length of class names (LEN). The exact definitions of these metrics are described in a compendium [14] and implemented in the VizzAnalyzer.

3.2. Assessment of the hypothesis

Here, we describe how we assess our hypothesis. We first discuss measurement and data collection, and then evaluation and analysis of the collected data.

Measurement and data collection is performed in three phases, two automated and one manual phase. Phase one focuses on the collection of the raw data. We use our *SF Extract tool* to download the meta-data of 146 SourceForge projects. The projects are sorted and selected according to their SourceForge rank and project category. Then, we use the *SVN Extract tool* to download these projects to local working copies. The data extraction is fast and reliable due to automation. In this phase, it should not be necessary to skip projects due to errors.

Phase two involves manual work. We have to prepare the downloaded projects for analysis with the VizzAnalyzer by importing them to an Eclipse workspace. During this process, we have to make sure that the projects compile without error to be able to analyze them in the next phase. This is not immediately the case for all projects downloaded; some have missing dependencies or other problems to be solved manually, cf. Section 4.

Phase three uses the extended VizzAnalyzer tool to analyze the projects in the workspace as a batch job. It is, again, automated. The exported data includes metrics for each class of every project, the name and the version of the project, some meta-data, and additional statistics.

Our tools perform quite well as some benchmarks illustrate: extracting meta-data (22 different properties) for 1000 projects takes SF Extract in average 30-110 minutes. The high variance seems to come from varying internet traffic and SourceForge load. Downloading 8 projects with a total of 157 MB takes SVN Extract ca. 167 seconds, which is almost the maximum we can get with our internet connection (0.97 MB/s). VizzAnalyzer requires ca. 109 minutes for a batch job (analysis and export) of 10 projects (58-7065, in total 13'223 files), on average about 0.5 seconds per file (0.04-10 sec.) and 11 minutes (0.13-12.8 min.) per project.

Evaluation and analysis is performed after the metric computations are written into the database for statistical analysis. Figure 2 shows histograms for the 16 metrics included in this study. Each histogram shows the distribution of classes over the different metric values, i.e., the number of classes (one grid line corresponds to 1000 classes on the y-axis) with a certain metrics value (which is metrics-dependent on the x-axis). The caption for each histogram contains the *minimum*, *maximum*, *average*, *median*, and *modus* values of a metric, and the *maximum number of classes* for the modus value over all projects.

We observe that basically none of the metrics (maybe

with the exception of LEN) are normally distributed. Most metrics are right-skewed (positive skewness) or have two peaks at both sides of the range (negative kurtosis). This has immediate implications for testing our hypothesis H_0, since we cannot use correlation tests expecting normally distributed data. Instead, we have to select a correlation test like Spearman's rank correlation coefficient [18] which is insensitive to non-normally distributed data.

The histogram also allows us to claim some hypotheses about absolute metrics thresholds. For metrics with almost all values in a very narrow value range, we can possibly identify some absolute thresholds beyond which a metric value is rather exotic. For instance, NOM values are close to 7 for the majority of classes, and higher values become rapidly exotic. This (peak) distribution may allow us to identify an absolute threshold value.

Using the Spearman's rank correlation coefficient, we get the correlations between the metrics as shown in Table 1. The coefficients of correlation show strong connections between 5 metric pairs with results greater or equal 0.90. Four of the six metrics involved (CBO, DAC, NAM, NOM, RFC and WMC) can be excluded, since they are redundant.

Based on these findings, we can reject our hypothesis H_0 for some pairs of metrics (indicated with bold-face correlation values in Table 1) and thus support our alternative hypothesis H_1.

3.3. Validity evaluation

To ensure validity of our study, we follow the design and methods recommended in [19], and, in the following, we discuss possible threats to validity.

Construct Validity ensures correct operational measures for the concepts being studied. We assure that there are no other varying factors than the software systems used, which may influence the outcome of the study. We select an appropriate set of metrics which have been discussed in literature, and for which we provide exact definitions. We assure that making the projects compilable has minimal impact.

Internal Validity ensures 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, since we do not try to explain causal relationships, but rather deal with an exploratory study. The possibility to interfere is limited in our setting. There are no human subjects which could be influenced, which could lead to different results depending on the date or person of the study. The influence on the projects is limited.

External Validity ensures the generalizability of our findings beyond the immediate case study. We include the most obvious open-source software projects available on the internet. These should represent a realistic fraction of the software developed and used in practice. We are aware that there is a much larger body of software in the commercial sector, including in-house software. Further, the limited selection of 146 software projects of the same programming language should still posses enough statistical power to generalize our conclusions – at least to other open-source Java projects. We randomize the project selection apart from the constraints discussed.

Reliability ensures 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 follow a straight-forward design, thus simplicity should support reliability. We document all important decisions and intermediate results as well as the procedures for the analysis. Yet, due to the space limitations of this report, we cannot include all information needed, but we link to external sources that contain additional details, where appropriate. We minimize our impact on the used artifacts and documented any modifications in protocols and log files. We describe the design of the experiments including the subsequent selection process.

4. Lessons learned

In this section, we provide an overview of the lessons we learned during the performed case study. We include more details and explanations for the problems we had, and point out the limitations of our approach for overcoming them. We further suggest possible improvements to our tools *and* to the way how source code repositories are currently organized to aid their automatic analysis.

While preparing the downloaded projects for analysis, the following problems occured (% projects affected):

Missing dependencies (28%): Required dependencies were not part of the repository. In many cases, we were able to identify them, since they were listed in the build instructions. In some cases it was not at all clear (and thus time consuming) which libraries in what version were needed, and we had to search the internet.

Syntax errors (7%): For some reason, the developers checked in code which does not compile. If the solution was obvious, we manually fixed the error. Otherwise, we commented out the line(s) to get compilable projects and tryed to affect the metrics as less as possible.

Ambiguous types (2%): Many classes import types using a wild-card, e.g., `import java.io.*`. Since they do not exactly specify which types are used, it could happen that, after modifying the original build configuration, e.g., selecting a different version of Java or a library, the compiler could not decide any longer which type to use. This could be fixed manually.

In less then 1% of the projects we had problems with multiple definitions of types, mismatches between directory and package structures, class-generators in the build pro-

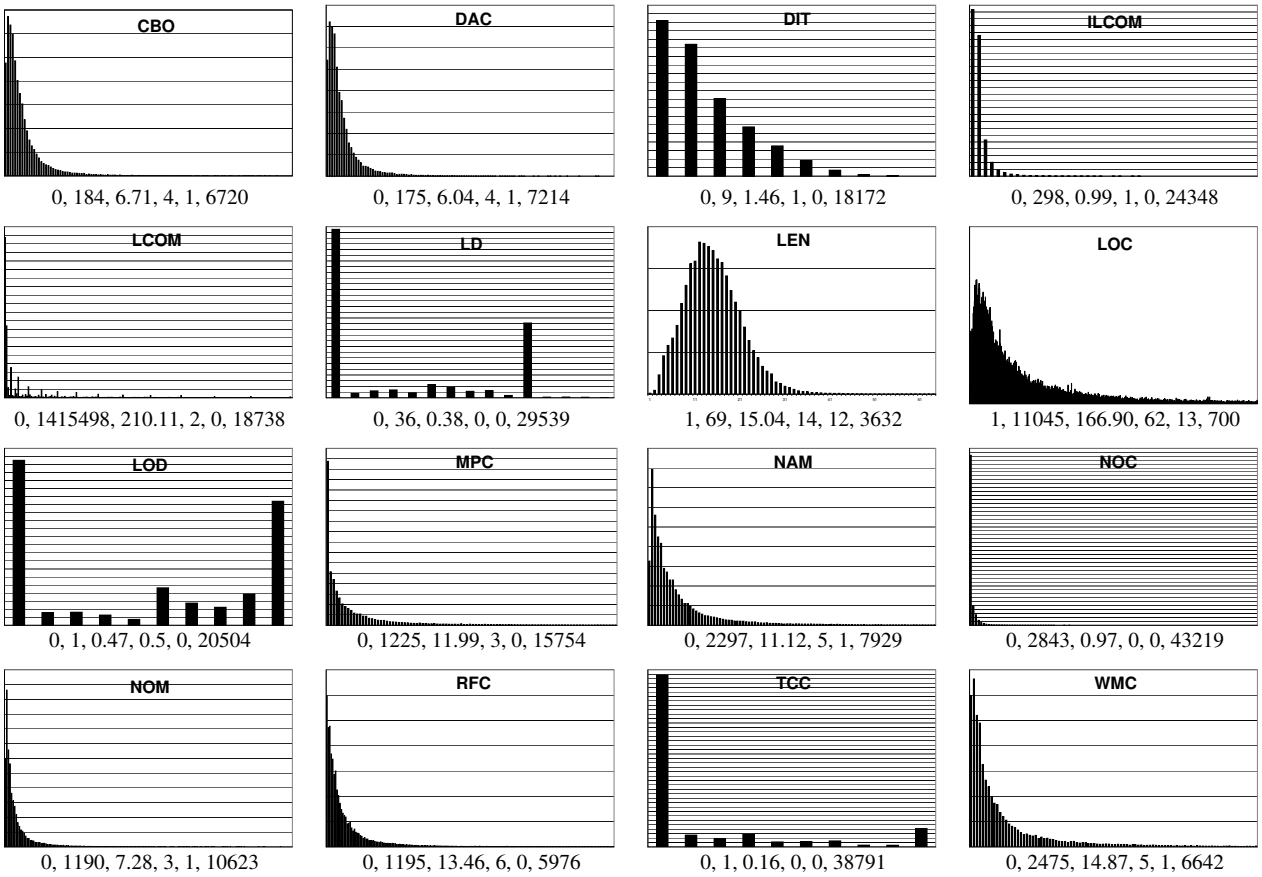

Figure 2. Histograms – statistical metrics values; min, max, average, median, modus, max number.

cess, and incomplete code. These problems could be fixed manually so that the projects compiled.

Altogether, 32% of the 146 projects included in our study needed manual fixes in order to enable analysis. Extending our tools with information about dependencies of libraries allowed us to complete the missing dependencies in an automated way.

Other issues will possibly never be solvable in an automated way. Therefore, we want to propose a number of suggestions which are both of scientific and practical benefit to the research and the open-source community. All suggestions are basically known guidelines involving common sense, but they are (too) often ignored because of convenience or lack of time.

Coherent organization of dependencies: Project repositories should contain complete projects, so that they are independent of other sources. Therefore, dependencies should be included in the version management. The goal is to check out and build directly.

Convention over configuration: Project repositories should have a clear layout and structure. This should be planned from the beginning, to allow good usability, extensibility and conformity. Other good principles include declarative execution and reuse of build logic.

Frequent tagging: This allows a gapless version history, and restoring previous states.

Testing: Similar to software, repositories and workspaces should be tested, in order to check for missing dependencies and other configuration and build problems.

Documentation: Similar to software, the build process should be documented.

We found that the projects based on Maven[4], a software project management tool, required less manual interaction to complete the dependencies than other projects.

5. Conclusion and future work

Validating quality metrics allows us to use efficient metrics-based software quality assurance and to guarantee their effectiveness. Large amounts of quantitative project data is needed in order to achieve high statistical significance. As we showed in related work, there have been quite some studies directed towards the validation of software quality metrics [2, 4, 7, 11]. However, all quantitative studies were limited to few smaller or one large systems.

[4] http://maven.apache.org/

	CBO	DAC	DIT	ILCOM	LCOM	LD	LEN	LOC	LOD	MPC	NAM	NOC	NOM	RFC	TCC	WMC
CBO	1															
DAC	**0.982**	1														
DIT	0.529	0.528	1													
ILCOM	0.397	0.414	0.391	1												
LCOM	0.539	0.551	0.405	0.478	1											
LD	0.315	0.334	0.430	0.794	0.449	1										
LEN	0.131	0.132	0.266	0.079	0.146	0.157	1									
LOC	0.581	0.600	0.142	0.477	0.580	0.325	-0.075	1								
LOD	0.092	0.072	0.245	0.307	0.276	0.372	0.095	-0.208	1							
MPC	0.830	0.813	0.534	0.576	0.597	0.505	0.210	0.667	0.276	1						
NAM	0.519	0.533	0.165	0.632	0.682	0.468	-0.045	0.837	0.096	0.627	1					
NOC	0.061	0.087	0.406	0.572	0.380	0.620	-0.046	-0.11	0.262	0.216	0.061	1				
NOM	0.563	0.580	0.238	0.598	0.799	0.480	0.002	0.791	0.137	0.650	**0.911**	0.144	1			
RFC	0.717	0.709	0.277	0.529	0.715	0.412	0.019	0.801	0.120	0.817	0.837	0.022	**0.907**	1		
TCC	0.336	0.355	0.543	0.781	0.468	0.803	0.206	0.269	0.447	0.510	0.411	0.841	0.455	0.367	1	
WMC	0.599	0.606	0.204	0.570	0.725	0.440	-0.01	0.844	0.112	0.712	0.880	0.054	**0.939**	**0.930**	0.405	1

Table 1. Correlation between metrics according to Spearman's rank correlation coefficient.

We described our attempts towards improving efficiency of validating metrics.

Our first contribution provides tool support for collecting large amounts of quantitative data on open-source software systems written in Java. We presented the tools and the collection process. We applied this process to collect metric data of almost 150 distinct projects with over 70.000 classes and over 11 million lines of code. This was possible by just one person in a few weeks of part time work.

Our second contribution reduces the number of metrics to validate. We could show correlation among the individual metrics, indicating that some of them seem to measure the same properties. This study also showed the practical applicability of our tools.

Finally, we describe metric values statistically, getting a first overview of the absolute value ranges for some well-known metrics suits. They deliver hypotheses for possible absolute thresholds, which in turn could be used to identify classes that are threats to quality. Experimental evaluations of the thresholds is future work.

We plan to extend our existing database with more projects. During this, we hope to be able to improve our process and tools to allow the fully automatic download and analysis of projects, needing no manual interaction (as far as possible). We hope to be able to find automatic solutions for missing dependencies and other problems identified.

The ultimate goal, however, is to experimentally validate Software Quality Models that are based on a number of independent, automatically assessable metrics by showing their correlations to well-accepted quality attributes, like costs of ownership, that are not automatically predictable.

References

[1] M. Bär, et al. The FAMOOS Object-Oriented Reengineering Handbook. http://www.iam.unibe.ch/~famoos/handbook/, Oct. 1999.

[2] V. R. Basili, et al. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Trans. Sw. Eng.*, 1996.

[3] J. M. Bieman and B. Kang. Cohesion and Reuse in an Object-Oriented System. In *SSR '95: Proc. of the 1995 Symp. on Softw. reusability*, New York, USA, 1995. ACM.

[4] K. E. Emam, et al. A validation of object-oriented metrics. ftp://ai.iit.nrc.ca/pub/iit-papers/NRC-43607.pdf, Oct. 1999.

[5] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object-Oriented Design. *IEEE Trans. on Softw. Eng.*, 1994.

[6] F. B. e Abreu. The MOOD metrics set. In *Proceedings ECOOP Workshop on Metrics*, 1995.

[7] R. Harrison, et al. An Investigation into the Applicability and Validity of Object-Oriented Design Metrics. *Empirical Software Engineering*, 3(3):255–273, 1998.

[8] M. Hitz and B. Montazeri. Measure Coupling and Cohesion in Object-Oriented Systems. In *Proc. of Int. Symp. on Applied Corporate Computing (ISAAC'95)*, Oct. 1995.

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

[10] ISO. ISO 9000:2005 "Quality management systems - Fundamentals and vocabulary", 2005.

[11] T. Khoshgoftaar, et al. Assessing uncertain predictions of software quality. In *Proc. 6th Int. SW Metrics Symp.*, 1999.

[12] W. Li and S. Henry. Object-Oriented Metrics that Predict Maintainability. *Journal of Systems and Software*, 1993.

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

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

[15] R. Lincke, J. Lundberg, and W. Löwe. Comparing software metrics tools. In *ISSTA '08: Proc. of the 2008 int. symp. on Software testing and analysis*, New York, USA, 2008. ACM.

[16] NASA. Recommended thresholds for satc code metrics – recommended thresholds for oo languages. http://satc.gsfc.nasa.gov/metrics/codemetrics/oo/thresholds/, Nov. 2008.

[17] SEI. Capability Maturity Model Integration (CMMI). http://www.sei.cmu.edu/cmmi/cmmi.html, 2006.

[18] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 100(3/4):441–471, 1987.

[19] R. K. Yin. *Case Study Research : Design and Methods (Applied Social Research Methods)*. SAGE Publ., Dec. 2002.