Electronic Communications of the EASST
Volume 60 (2013)

Proceedings of the
Seventh International Workshop on
Software Quality and Maintainability
-
Bridging the gap between end user expectations,
vendors' business prospects,
and software engineers' requirements on the ground
(SQM 2013)

A Drill-Down Approach for Measuring Maintainability at Source Code Element Level

Péter Hegedűs, Tibor Bakota, Gergely Ladányi, Csaba Faragó, and Rudolf Ferenc

21 pages

Guest Editors: Eric Bouwers, Yijun Yu
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A Drill-Down Approach for Measuring Maintainability at Source Code Element Level

Péter Hegedűs[1], Tibor Bakota[2], Gergely Ladányi[3], Csaba Faragó[4], and Rudolf Ferenc[5]

[1] hpeter@inf.u-szeged.hu
[2] bakotat@inf.u-szeged.hu
[3] lgergely@inf.u-szeged.hu
[4] farago@inf.u-szeged.hu
[5] ferenc@inf.u-szeged.hu
University of Szeged
Department of Software Engineering
Árpád tér 2. H-6720 Szeged, Hungary

Abstract: Measuring source code maintainability has always been a challenge for software engineers. To address this problem, a number of metrics-based quality models have been proposed by researchers. Besides expressing source code maintainability in terms of numerical values, these models are also expected to provide explicable results, i.e. to give a detailed list of source code fragments that should be improved in order to reach higher overall quality.

In this paper, we propose a general method for drilling down to the root causes of a quality rating. According to our approach, a *relative maintainability index* can be calculated for each source code element for which metrics are calculated (e.g. methods, classes). The index value expresses the source code element's contribution to the overall quality rating.

We empirically validated the method on the jEdit open source tool, by comparing the results with the opinions of software engineering students. The case study shows that there is a high, 0.68 Spearman's correlation, which suggests that relative maintainability indices assigned by our method express the subjective feelings of humans fairly well.

Keywords: Relative maintainability index, Method level maintainability, Metrics-based quality model, ISO/IEC 9126

1 Introduction

Aggregating a measure for maintainability has always been a challenge in software engineering. The ISO/IEC 9126 standard [ISO01] defines six high-level product quality characteristics that are widely accepted both by industrial experts and academic researchers. These characteristics are: functionality, reliability, usability, efficiency, maintainability and portability. Maintainability is probably the most attractive, noticeable and evaluated quality characteristic of all. The importance of maintainability lies in its very obvious and direct connection with the costs of altering the behavior of the software package. Although the standard provides a definition for maintainability, it does not provide a straightforward way of quantifying it. Many researchers exploited this vague definition and it has led to a number of practical quality models [MPKS00, BD02, Azu96, BHK+11]. The non-existence of formal definitions and the subjectiveness of the notion are the major reasons for it being difficult to express maintainability in numerical terms.

Besides expressing source code maintainability in terms of numerical values, these models are also expected to provide explicable results, i.e. to give a detailed list of source code fragments that should be improved by the programmers in order to reach higher overall quality. Current approaches usually just enumerate the most complex methods, most coupled classes or other source code elements that bear with external values for some source code metric. Unfortunately, this is not enough; constellations of the metrics should also be taken into consideration. For example, a source code method with a moderate McCabe's complexity [McC76] value might be more important from a maintenance point of view than another method with a higher complexity, provided that the first one has several copies and contains coding problems as well. It follows that a more sophisticated approach is required for measuring the influence of individual source code elements on the overall maintainability of a system.

In this paper, we propose a general method for drilling down to the root causes of problems with the maintainability of a software system. According to our approach, a *relative maintainability index* is calculated for each source code element, which measures the extent to which the overall maintainability of the system is being influenced by it. For measuring the maintainability of a software system, we employed our probabilistic source code maintainability model [BHK+11].

We empirically validated the approach on the jEdit open source tool, by comparing the results with the opinions of software engineering students. The case study shows that there is a high, 0.68 Spearman's correlation at $p < 0.001$ significance level, which suggests that relative maintainability indices assigned by our method express the subjective feelings of humans fairly well.

In the next section, we summarize some of the studies related to ours. Then, in Section 3 we present a detailed description of our approach. Next, in Section 4 we present a case study, which evaluates the usefulness of the approach. In Section 5 we collect some threats to the validity of our approach. Finally, in Section 6 we round off with the conclusions and the lessons learned.

2 Related Work

Measuring software quality has always been a central issue in software engineering. Starting from as early as the late 70's a large number of theoretical quality models have been introduced like McCall's model [MRW77], Boehm's model [BBK+78], Dromey's model [Dro95], the FURPS model [Gra92], etc. Chung et al. emphasize the importance of defining software quality aspects (i.e. non-functional requirements) already in the requirements establishing phase [CLC09].

The release of the ISO/IEC 9126 [ISO01] and ISO/IEC 25000 [ISO05] quality standards has created a new and very important direction of software quality measurement by defining a unified set of properties of software quality. The majority of researches deal with determining these quality properties for the system as a whole. However, only a few papers study the quality on finer levels (e.g. methods or classes). The aim of the current work is to present a novel approach and algorithm for calculating quality attributes on this fine level. But first, we introduce the related results and techniques for determining the ISO/IEC 9126 quality characteristics on the level of source code elements.

Machine learning is a widely used technique to approximate subjective human opinions based on different predictors. It is a very powerful tool; usually the built-up models are close to the optimal solution. In this case the test and the learning instances usually come from the same dataset with the same distribution.

Bagheri and Gasevic [BG11] used this technique to approximate the maintainability property of software product line feature models. They studied the correlation between the low-level code metrics and the high-level maintainability subcharacteristics evaluated by graduate students. They also applied different machine learning models to predict the subjective opinions.

In our previous work [HBI$^+$11] we used machine learning algorithms to predict the maintainability subcharacteristics of Java methods. The subjective opinions were collected from 36 experts and the effectiveness of the models was measured by cross-validation. Based on different source code metrics the J48 decision tree algorithm was able to classify the changeability characteristic of the methods into 3 classes (bad, average, good) with 0.76 precision.

The bottom-up methods on the other hand do not use subjective opinions about the characteristics, the validation dataset is independent from the model.

Heitlager et al. [HKV07] described a bottom-up approach developed by the Software Improvement Group (SIG) [SIG] for code analysis focused on software maintainability. They use threshold values to split the basic metric values into five categories from poor to excellent. The evaluation in their approach means summing the values for each attribute (having the values between -2 and +2) and then aggregating the values for higher properties.

Alves et al. [AYV10] presented a method to determine the threshold values more precisely based on a benchmark repository [CV08] holding the analysis results of other systems. The model has a calibration phase [BSV10] for tuning the threshold values of the quality model in such a way that for each lowest level quality attribute they get a desired symmetrical distribution. They used the $\langle 5,30,30,30,5 \rangle$ percentage-wise distributions over 5 levels of quality.

The SIG model originally used binary relations between system properties and characteristics, but Correia et al. [CKV09] prepared a survey to elicit weights to the model. They concluded that using weights does not improve the model because of the variance in developers' opinion. Furthermore, as an interpretation of the ranking, an evaluator can point out the really weak points of the examined system regarding the different quality attributes.

Our approach has the same phases as the SIG model, but the phases themselves are very different. The most significant difference is that our approach presented in this paper is created for measuring the maintainability of source code elements (e.g. classes or methods) and not the system as a whole. Our purpose was to determine the critical elements of a system which cause the largest decrease in overall maintainability, providing technical guidelines for the developers.

To convert the low-level source code metrics into quality indices we also used a benchmark with a large amount of system evaluations, but we applied it in a different way. During the calibration, instead of calculating threshold values we approximate a normal distribution function called benchmark characteristic (see Section 3), which is used to determine the goodness of the system with respect to a certain metric. The distribution used in the SIG model is also very close to a normal distribution but we had to use a continuous scale since the impact of a source code element on the overall quality of the system could be very small.

The SQUALE model presented by Mordal-Manet et al. [MBD+09] introduces the so called practices to connect the ISO/IEC 9126 characteristics with metrics. A practice in a source code element expresses a low-level rule and the reparation cost of violating this rule. The reparation cost of a source code element is calculated by the sum of the reparation costs of its rule violations. The quality of a source code element can be measured by the average cost per line. After calculating this raw value they convert it to a goodness value (A, B, C, D, E) using thresholds. Rule violations have an important role in our approach too, but besides the number of serious and medium rule violations in a source code element we consider other source code metrics as well. Our algorithm does not measure reparation costs, but the extent to which the overall maintainability of the system is being influenced by a source code element. On the other hand, we also showed that maintainability has a serious effect on development costs [BHL+12].

3 Approach

In this section we present an approach that is an extension of our earlier research achievement concerning software quality models [BHK+11]. First, we briefly introduce our probabilistic software quality model, which can be used for measuring source code maintainability at system level. In our approach, the so-called *benchmark characteristics* play an important role, therefore, a separate subsection is devoted to this issue. Finally, we present the basic idea of how the relative maintainability index for individual source code elements can be calculated.

3.1 The Columbus Quality Model

Our probabilistic software quality model [BHK+11] is based on the quality characteristics defined by the ISO/IEC 9126 [ISO01] standard. In our approach, the relations between quality attributes and characteristics at different levels are represented by an acyclic directed graph, called the *attribute dependency graph (ADG)*. The nodes at the lowest level (i.e. without incoming edges) are called *sensor nodes*, while the others are called *aggregate nodes*. Figure 1 shows an instance of the applied ADG. The description of the different quality attributes can be found in Table 1.

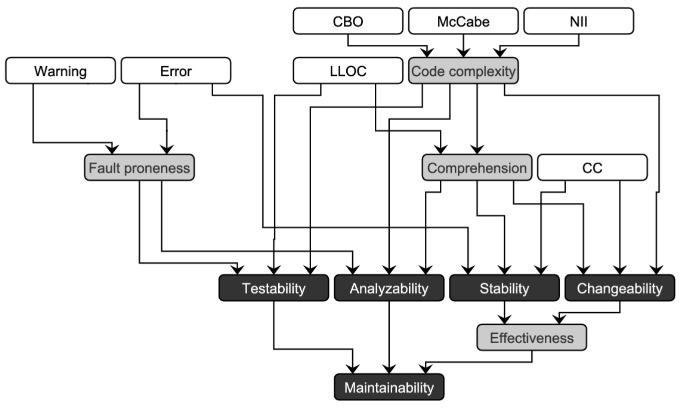


Figure 1: Java attribute dependency graph

Table 1: The quality properties of our model

Sensor nodes	
McCabe	McCabe cyclomatic complexity [McC76] defined for the methods of the system.
CBO	Coupling between object classes, which is defined for the classes of the system.
NII	Number of incoming invocations (method calls), defined for the methods of the system.
LLOC	Logical lines of code of the methods.
Error	Number of serious PMD [PMD] coding rule violations, computed for the methods of the system.[1]
Warning	Number of suspicious PMD coding rule violations, computed for the methods of the system.[1]
CC	Clone coverage [BYM+98]. The percentage of copied and pasted source code parts, computed for the methods of the system.
Aggregated nodes defined by us	
Code complexity	Represents the overall complexity (internal and external) of a source code element.
Comprehension	Expresses how easy it is to understand the source code.
Fault proneness	Represents the possibility of having a faulty code segment.
Effectiveness	Measures how effectively the source code can be changed. The source can be changed effectively if it is easy to change and changes will likely not have unexpected side-effects.
Aggregated nodes defined by the ISO/IEC 9126	
Analyzability	The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified.
Changeability	The capability of the software product to enable a specified modification to be implemented, where implementation includes coding, designing and documenting changes.
Stability	The capability of the software product to avoid unexpected effects from modifications of the software.
Testability	The capability of the software product to enable modified software to be validated.
Maintainability	The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.

The sensor nodes in our approach represent source code metrics that can be readily obtained from the source code. In the case of a software system, each source code metric can be regarded as a random variable that can take real values with particular probability values. For two different software systems, let $h_1(t)$ and $h_2(t)$ be the probability density functions corresponding to the same metric. Now, the *relative goodness value* (from the perspective of the particular metric) of one system with respect to the other, is defined as

$$\mathcal{D}(h_1,h_2) = \int_{-\infty}^{\infty} (h_1(t) - h_2(t))\,\omega(t)\,dt,$$

where $\omega(t)$ is the weight function that determines the notion of goodness, i.e. where on the horizontal axis the differences matter more. Figure 2 helps us understand the meaning of the formula: it computes the non-symmetrical signed area between the two functions weighted by the function $\omega(t)$.

[1] The full list of applied PMD rules is available online: http://www.inf.u-szeged.hu/~hpeter/SQM2013/PMD.xls

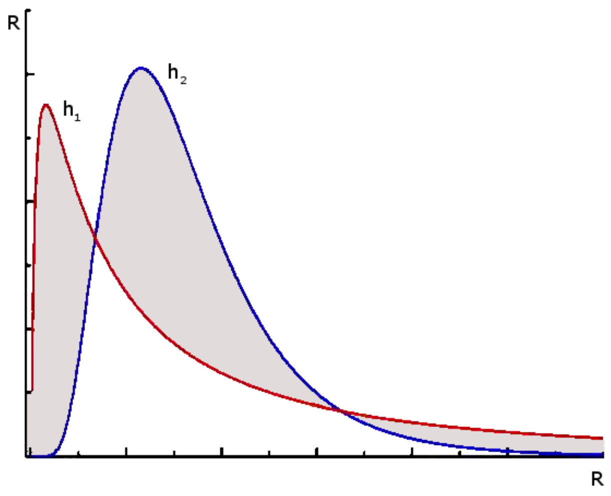


Figure 2: Comparison of probability density functions

For a fixed probability density function h, $\mathscr{D}(h,\_)$ is a random variable, which is independent of any other particular system. We will call it the *absolute goodness* of the system (from the perspective of the metric that corresponds to h). The empirical distribution of the absolute goodness can be approximated by substituting a number of samples for its second parameter, i.e. by making use of a repository of source code metrics of other software systems. We used the repository introduced earlier [BHK+11], containing the metric results of 100 Java systems. The probability density function of the absolute goodness is called the *goodness function*. The expected value of the absolute goodness will be called the *goodness value*. Following the path described above, the goodness functions for the sensor nodes can be easily computed.

For the edges of the *ADG*, a survey was prepared, where the IT experts and researchers who filled it were asked to assign weights to the edges, based on how they felt about the importance of the dependency. They were asked to assign scalars to incoming edges of each aggregate node, such that the sum is equal to one. Consequently, a multi-dimensional random variable $\vec{Y}_v = \left(Y_v^1, Y_v^2, \ldots, Y_v^n\right)$ will correspond to each aggregate node v. We define the aggregated goodness function for the node v in the following way:

$$g_v(t) = \int\limits_{\substack{t=\vec{q}\vec{r} \\ \vec{q}=(q_1,\ldots,q_n)\in\Delta^{n-1} \\ \vec{r}=(r_1,\ldots,r_n)\in C^n}} \vec{f}_{\vec{Y}_v}(\vec{q})\, g_1(r_1)\ldots g_n(r_n)\, d\vec{r}d\vec{q}, \tag{1}$$

where $\vec{f}_{\vec{Y}_v}(\vec{q})$ is the probability density function of $\vec{Y}_v$, $g_1, g_2, \ldots g_n$ are the goodness functions corresponding to the incoming nodes, Δ^{n-1} is the $(n-1)$-standard simplex in $\Re^n$ and C^n is the standard unit n-cube in $\Re^n$.

Although the formula may look frightening at first glance, it is just a generalization of how aggregation is performed in the classic approaches. Classically, a linear combination of goodness values and weights is taken, and it is assigned to the aggregate node. When dealing with probabilities, one needs to take every possible combination of goodness values and weights, and also the probabilities of their outcome into account. Now, we are able to compute goodness functions for each aggregate node; in particular the goodness function corresponding to the *Maintainability* node as well.

3.2 Benchmark Characteristics

After a benchmark containing several systems is available and a particular model is defined, the goodness values for each software in the benchmark can be calculated. In this way – for a particular node in the ADG – several goodness values can be obtained, which can actually be considered as samples of a random variable. According to the *Central Limit Theorem* for independent (not necessarily identically distributed) random variables, this data tends to a normal distribution which is independent of the benchmark histograms. This is naturally a theoretical result, and it states that having large number of systems in our benchmark, the constructed goodness functions are (almost) independent of the particular systems in the benchmark which is also supported by our empirical observations.

As each system is compared to every other system twice (with opposite signs), the expected value of the distribution is necessarily zero. The distribution functions obtained in this way are called the *benchmark characteristics*. Figure 3 shows an example of a benchmark characteristic for the Maintainability node.

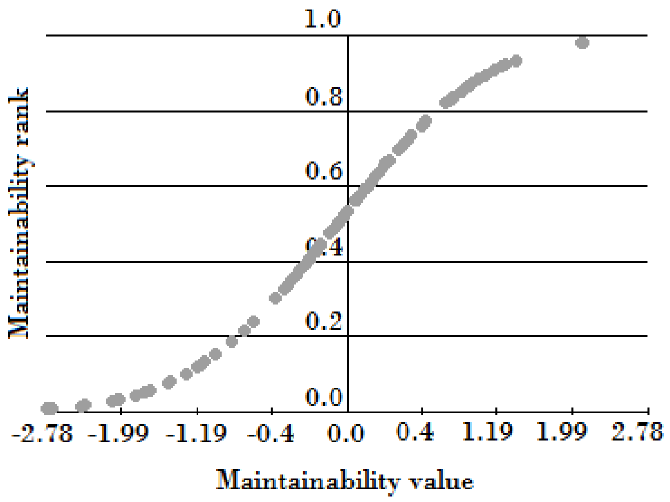


Figure 3: Benchmark characteristic for Maintainability

The characteristic functions map the $(-\infty, \infty)$ interval to the $(0,1)$ interval. Therefore, we use these functions to transform the goodness values of a particular software system being evaluated to the $(0,1)$ interval. In this way, a common scale is obtained for the goodness values of different systems; i.e. they become comparable to each other. The normalized goodness value is basically the proportion of the systems within the benchmark whose goodness values are smaller than the evaluated one.

3.3 The Drill-Down Approach

The above approach is used to obtain a system-level measure for source code maintainability. Our aim is to drill down to lower levels in the source code and to get a similar measure for the building blocks of the code base (e.g. classes or methods). For this, we define the *relative maintainability index* for the source code elements, which measures the extent to which they affect the system level goodness values. The basic idea is to calculate the system level goodness values, leaving out the source code elements one by one. After a particular source code element is left out, the system level goodness values will change slightly for each node in the ADG. The difference between the original goodness value computed for the system, and the goodness value computed without the particular source code element, will be called the *relative maintainability index* of the source code element itself. The relative maintainability index is a small number that is either positive when it improves the overall rating or negative when it decreases the system level maintainability. The absolute value of the index measures the extent of the influence to the overall system level maintainability. In addition, a relative index can be computed for each node of the ADG, meaning that source code elements can affect various quality aspects in different ways and to different extents.

Calculating the system-level maintainability is computationally very expensive. To obtain the relative indices, it is enough to compute only the goodness values for each node in the ADG; one does not need to construct the goodness functions. Luckily, computing the goodness values without knowing the goodness functions is feasible. It can be shown that calculating goodness functions and taking their averages is equivalent to using only the goodness values throughout the aggregation.

In the following, we will assume, that $\omega(t)$ is equal to t for each sensor node, which means that e.g. twice as high metric value means twice as bad code. While this linear function might not be appropriate for every metric, it is a very reasonable weight function considering the metrics used by the quality model. However, the presented approach is independent of the particular weight function used, and the formalization can be easily extended to different weight functions. Next, we will provide a step-by-step description of the approach for a particular source code element.

1. For each sensor node n, the goodness value of the system without the source code element e can be calculated via the following formula:

   $$g_{rel}^{e,n} = \frac{Kg_{abs}^{n} + m}{K-1} - \frac{1}{N}\sum_{j=1}^{N}\frac{M_j}{K-1}$$

   where g_{abs}^{n} is the original goodness value computed for the system, m is the metric value of the source code element corresponding to the sensor node, K is the number of source code elements in the system for which the sensor node is considered, N is the number of the systems in the benchmark, and $M_j\,(j = 1,\dots,N)$ are the averages of the metrics for the systems in the benchmark.

2. The goodness value obtained in this way is transformed to the $(0,1)$ interval by using the characteristic function of the sensor node n. For simplicity reasons, we assume that

from now on $g_{rel}^{e,n}$ stands for the transformed goodness value and it will be referred to as goodness value as well.

3. Due to the linearity of the expected value of a random variable, it can be shown that Formula 1 simplifies to a linear combination, provided that only the expected value needs to be computed. Therefore, the goodness value of an aggregate node n can be computed in the following way:

$$g_{rel}^{e,n} = \sum_i g_{rel}^i E\left(Y_v^i\right)$$

where $g_{rel}^i\,(i = 1,\dots)$ are the transformed goodness values of the nodes that are on the other sides of the incoming edges and $E\left(Y_v^i\right)$ is the expected value of the votes on the i^{th} incoming edge. Please note that since $\sum_i E\left(Y_v^i\right) = 1$, and $\forall i, g_{rel}^i \in (0,1)$, the value of $g_{rel}^{e,n}$ will always fall into the $(0,1)$ interval, i.e. no transformation is needed at this point.

4. The relative maintainability index for the source code element e and for a particular ADG node n is defined as

$$g_{idx}^{e,n} = g_{abs}^n - g_{rel}^{e,n}$$

The relative maintainability index measures the effect of the particular source code element on the system level maintainability computed by the probabilistic model. It is important to notice that this measure determines an ordering among the source code elements of the system, i.e. they become comparable to each other. And what is more, the system level maintainability being an absolute measure of maintainability, the relative index values become absolute measures of all the source code elements in the benchmark. In other words, computing all the relative indices for each software system in the benchmark will give rise to an absolute ordering among them.

Moreover, it can be shown that it is not possible to have only positive or negative indices (with the special case when every index is zero). Therefore we can always point out those methods that decrease maintainability (i.e. there will always be room for improvement).

4 Empirical Validation

We evaluated the approach on the jEdit v4.3.2 open source text editor tool (http://www.jedit.org/), by considering a large number of its methods in the source code.[2] The basic properties of jEdit's source code and the selected methods are shown in Table 2. These and all other source code metrics were calculated by our *Columbus Code Analyzer tool* [FBTG02]. A considerable number of students were asked to rate the maintainability and lower level quality aspects of 191 different methods. Here, we reused the data of our earlier empirical case study [HLSF12], where over 200 students manually evaluated the different ISO/IEC 9126 quality attributes of the methods in the jEdit tool. Even though we conducted a preliminary survey with IT experts, the amount of collected data was insufficient, therefore we chose to use the student evaluation for validation purposes. For the empirical validation, the averages of students' votes were taken and they were

[2] https://jedit.svn.sourceforge.net/svnroot/jedit/jEdit/tags/jedit-4-3-2

compared to the series of numerical values (i.e. relative maintainability indices) computed by the approach. (A repeated experiment using the median of the votes yielded very similar results.)

We calculated the Spearman's rank correlation coefficient for the two series of numbers. This coefficient can take its values from the range $[-1,1]$. The closer this value is to 1, the higher is the similarity between the manual rankings and the automatically calculated values.

Table 2: Basic metrics of the source code of jEdit and the evaluated methods

Metrics	**Value**
Logical Lines of Code (LLOC)	93744
Number of Methods (NM)	7096
Number of Classes (NCL)	881
Number of Packages (NPKG)	49
Number of evaluated methods	191
Average number of LLOC for the evaluated methods	26.41
Average number of McCC complexity for the evaluated methods	5.52

4.1 Manual Evaluation

The students who took part in the evaluation were third year undergraduate students and completed a number of programming courses. Some of them already had some industrial experience as well. Each of the students were asked to rank the quality attributes of 10 different methods of jEdit v4.3.2 subjectively. Altogether 191 methods have been distributed among them. For the evaluation, a web-based graphical user interface was constructed and deployed, which provided the source code fragment under evaluation together with the questionnaire about the quality properties of the methods.

The methods for the manual evaluation were selected randomly. We assigned the methods to students in such a way that each method was always supposed to be evaluated by at least ten persons. The students were asked to rate the subcharacteristics of maintainability defined by the ISO/IEC 9126 standard on a scale of zero to ten; zero means the worst, while ten is the best. These attributes were the following: *analyzability*, *changeability*, *testability*, *stability*) and a new quality attribute – *comprehensibility* – defined by us earlier [HBI+11]. More details regarding the data collection process and the used web application can be found in our previous work [HBI+11].

Table 3 contains some basic statistics of the collected student votes. The first column shows the average standard deviation values of student votes for the different quality attributes. The values vary between 1.8 and 2.2 which indicates that the students had an acceptable level of inter-rater agreement in general. Thus, we decided not to filter out any methods. The next two columns show the maximum and minimum of the standard deviations. The maximums are approximately two times higher than the averages, but the minimum values are very close to zero. For analyzability, changeability and maintainability attributes the minimum is exactly zero, meaning that there was at least one method that got exactly the same rating from each student.

Table 3: Statistics of the student votes

Quality attribute	**Avg. std.dev.**	**Max. std.dev.**	**Min. std.dev.**
Analyzability	1.87	3.93	0.00
Comprehensibility	1.89	4.44	0.44
Stability	2.22	4.31	0.53
Testability	2.04	3.82	0.32
Changeability	2.01	3.62	0.00
Maintainability	1.97	3.93	0.00

Table 4: Spearman's correlations among the source code metrics and students' opinions

	Comprehen-sibility		**Analyzability**		**Changeability**		**Stability**		**Testability**		**Maintain-ability**	
Metric	**R**	**p-val.**	**R**	**p-val.**	**R**	**p-val.**	**R**	**p-val.**	**R**	**p-val.**	**R**	**p-val.**
CC	0.02	0.39	0.03	0.37	0.01	0.46	0.04	0.30	0.02	0.41	0.04	0.29
LLOC	**-0.63**	0.00	**-0.68**	0.00	**-0.57**	0.00	**-0.55**	0.00	**-0.63**	0.00	**-0.72**	0.00
McCC	**-0.46**	0.00	**-0.48**	0.00	**-0.41**	0.00	**-0.49**	0.00	**-0.45**	0.00	**-0.53**	0.00
NII	-0.03	0.33	-0.01	0.42	-0.02	0.40	-0.09	0.10	-0.02	0.38	-0.03	0.34
Error	**-0.17**	0.01	**-0.15**	0.02	-0.07	0.15	-0.10	0.08	-0.08	0.13	**-0.19**	0.00
Warning	**-0.21**	0.00	**-0.25**	0.00	**-0.22**	0.00	**-0.21**	0.00	**-0.23**	0.00	**-0.19**	0.00

Table 4 shows the Spearman's correlation coefficients for the different metric values used in our quality model (see Figure1) and the average votes of the students for the high-level quality attributes (the CBO metric is not listed because it is a metric defined for classes and not for methods). Based on the data, a number of observations can be made:

- All of the significant Spearman's correlation coefficients (R values) are negative. This means that the greater the metric values are, the worse are the different quality properties. This is in line with our expectations, as lower metrical values are usually desirable, while higher values may suggest implementation or design related flaws.

- The quality attributes correlate mostly with the logical lines of code (LLOC) and the McCabe's cyclomatic complexity (McCC) metrics. The reason for this might be that these are very intuitive and straightforward metrics that can be observed locally, by looking only at the code of the method's body.

- As an analogy to the previous observation, being hard to interpret the metrics locally, the clone coverage (CC) and the number of incoming invocations (NII) metrics have no connection with the students' votes (i.e. the p-values are high).

- The number of rule violations also shows a noticeable correlation with the quality ratings. Surprisingly, the suspicious rule violations show a higher correlation than the really serious ones. The reason for this might be that either the students considered different violations as serious or the fact that the number of the most serious rule violations was low (five times lower than suspicious violations), which may have biased the analysis.

4.2 Model-Based Evaluation

We calculated the quality attributes for all the methods of jEdit by using the implementation of the algorithm presented in Section 3. The relative maintainability indices are typically small positive or negative numbers. The negative values indicate negative impact on the maintainability of the system while positive indices imply positive effect. As we are mainly interested in the order of the methods based on their impact on the maintainability, we assigned an integer rank to every method by simply sorting them according to their relative maintainability index in a decreasing order. The method having the largest positive impact on the maintainability gets the best rank (number 1) while the worst method gets the worst rank (which equals to the number of methods in the system). Therefore, the most critical elements will be at the end of the relative maintainability based order (i.e. larger rank means worse maintainability).

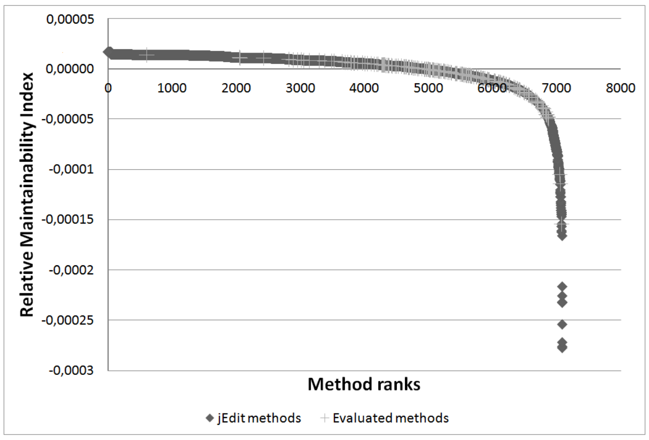


Figure 4: The relative maintainability indices and corresponding ranks

Figure 4 depicts the relative maintainability indices of the methods of jEdit and their corresponding ranks. It can be seen that there are more methods that increase the overall maintainability than those which decrease it. However, methods having positive impact only slightly improve the overall maintainability, while there are about 500 methods that have a significantly larger negative impact on the maintainability. In principle, these are the most critical methods that should be improved first, to achieve a better system level maintainability.

Figure 5 shows the density function of the computed relative maintainability indices. In accordance with the previous observations, we can see that there are more methods that increase the maintainability, however, their maintainability index values are close to zero. This means that they have only a small positive impact. The skewed left side denotes that there is a smaller number of methods decreasing the maintainability but they have a significantly larger negative effect (their index is farther out from zero). This is in line with the well-known Pareto principle [San92]: about 20% of the source code elements have negative maintainability indices but around 80% of the total maintainability degradation is caused by them.

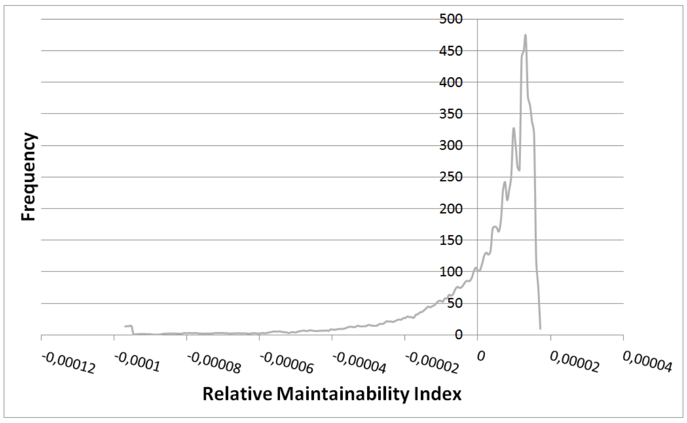


Figure 5: The density function of the relative maintainability indices

For evaluating the effectiveness of our approach, we took the calculated quality attributes for each of the manually evaluated methods. Our conjecture was that the model-based assessment of quality rankings will not differ much from the manually obtained values. If this proves to be true, a list of the most critical methods could always be generated automatically, which would correlate well with the opinion's of the developers.

Since most complexity metrics used in the model are not necessarily independent, we may not assume that the probabilistic density functions follow normal distributions even if the observations seem to support it. Moreover, we are interested in the similarity between the rankings produced by our algorithm and the rankings obtained with the manual evaluation, so we performed a Spearman's rank correlation analysis instead of Pearson's correlation.

The Spearman's correlation coefficient is defined as the Pearson's correlation coefficient between the ranked variables. Identical values (rank ties or value duplicates) are assigned a rank equal to the average of their positions in the ascending order of the values.

First, we analyzed the relationship between the quality ranking calculated by our algorithm and the rankings assigned by the students. The Spearman's correlation coefficients and the corresponding p-values can be seen in Table 5. In statistical hypothesis testing, the p-value is the probability of obtaining a test statistic that is at least as extreme as the one that was actually observed, assuming that the null hypothesis is true. In our case, the null hypothesis will be that there is no relation between the rankings calculated by the algorithm and the one obtained by manual evaluations.

As can be seen, the R values of the correlation analysis are relatively high for each quality attribute. All the values are significant at the level of 0.001. This means that there is a significant relationship between the automatically obtained rankings and the one derived from the students' evaluations. The best correlation was found between the data series of the Maintainability characteristic. According to the results we can automatically identify those critical source code elements that decrease the system's maintainability the most. These are the source code elements at the end of the ranked list (having the worst relative maintainability indices). The list of critical elements is crucial in order to improve the quality of a system, or at least to decrease the rate of its erosion.

Table 5: Spearman's correlation values among the relative maintainability indices and the manual evaluations

Quality attribute	Correlation with students' opinions (R value)	p-value
Analyzability	0.64	<0.001
Comprehensibility	0.62	<0.001
Changeability	0.49	<0.001
Stability	0.49	<0.001
Testability	0.61	<0.001
Maintainability	**0.68**	**<0.001**

Table 6: The largest differences between the automatic and manual rankings

Method name	Students' ranking	Model ranking	Rank diff.	CC	LLOC	McCabe	NII	Err.	Warn.
invokeSuperclassMethodImpl	177	57	120	0	17	2	1	0	0
fireContentInserted	29	139	110	1	18	3	2	0	1
fireEndUndo	13	121	108	1	15	3	0	0	1
move	168	66	102	0	16	3	0	0	0
fireTransactionComplete	42	142	100	1	16	3	5	0	1
read	113	16	97	0	10	2	0	0	0

Evaluation of the largest deviations in the manual and model based rankings. Although the results are promising in their current form, we went towards tracing down the differences between the manually and automatically obtained rankings. We collected and manually examined several methods that had the largest differences in their rankings. Table 6 lists the assessed methods, their rankings and their metrical values (except for CBO which is not defined for methods). In the following, we will provide a more detailed explanation regarding the differences, considering the methods one-by-one:

- *org.gjt.sp.jedit.bsh.ClassGeneratorImpl.invokeSuperclassMethodImpl(BshClassManager, Object, String, Object[])*

  This method attempts to find and invoke a particular method in the superclass of another class. While our algorithm found this method rather easy to maintain, the students gave it a very low ranking. Despite the fact that syntactically this program fragment is very simple it uses Java reflection which is by its nature difficult to read and comprehend by humans.

- *org.gjt.sp.jedit.buffer.JEditBuffer.fireContentInserted(int, int, int, int)*

  This method is a fairly simple function at a first glance, which fires an event to each listener of an object. On the other hand, from the maintainability point of view, changing the method might be risky as it has four clones (copy&paste). All the fire events are duplications of each other. The code contains also a medium rule violation; a catch block that catches all *Throwable* objects. It can hide problems like runtime exceptions or errors. Nevertheless, human evaluators tend to give more significance to local properties, like the lines of code or McCabe's complexity, because it is hard to explore the whole environment of the code fragment.

- *org.gjt.sp.jedit.Buffer.fireEndUndo()*

  This is exactly the same type of a fire method than the previous one. Therefore, the same reasoning holds in this case as well.

- *org.gjt.sp.jedit.browser.VFSBrowser.move(String)*

  This method is responsible for moving a toolbox. It is short and has low complexity, therefore our algorithm ranked it as a well-maintainable code. However, human evaluators found it hard to maintain. The code is indeed a little bit hard to read, because of the unusual indentation (every expression goes to new line) of the logical operators which might be the cause of the low human ranking.

- *org.gjt.sp.jedit.buffer.JEditBuffer.fireTransactionComplete()*

  This is another method responsible for firing events, just like the earlier ones. Therefore the same reasoning holds in this case as well.

- *org.gjt.sp.jedit.bsh.CommandLineReader.read(char[], int, int)*

  This method reads a number of characters and puts them into an array. The implementation itself is short and clear, not complex at all, therefore our algorithm ranked it as well-maintainable. However, the comment above the method starts with the following statement: “This is a degenerate implementation”. Human evaluators probably read the comment and marked the method as hard to maintain.

The manual assessment shed light to the fact that human evaluators tend to take into consideration a wider range of source code properties than the presented quality model. These properties are e.g. code formatting, semantic meaning of the comments or special language constructs like Java reflection. The automatic quality assessment should be extended with measurements of these properties to achieve more precise results.

On the other hand, the automatic quality assessment may take advantage from the fact that it is able to take the whole environment of a method into account for maintainability prediction. We found that human evaluators had difficulties in discovering non-local properties like clone fragments or incoming method invocations. Another issue was that while the algorithm was considering all the methods at the same time, the human evaluators assigned their ranks based only on the methods they evaluated. For example, while the best method gets a maximal score from the model, the evaluators may not recognize it as the best one, as they have not seen all the others.

Table 7: Methods with the worst and best maintainability indices

Method name	**Students' ranking**	**Model ranking**	**Rank diff.**	**CC**	**LLOC**	**McCabe**	**NII**	**Err.**	**Warn.**
processKeyEvent	152	190	38	0	171	39	1	1	2
checkDependencies	187	189	2	0	163	25	3	0	1
doSuffix	190	182	8	0	50	14	1	0	2
getState	1	1	0	0	3	1	0	0	0
UtilTargetError	13	2	11	0	3	1	0	0	0
getDefaultProperty	2	5	3	0	4	1	1	0	0

Evaluation of the methods with the worst and best maintainability indices. Besides analyzing methods that had the largest differences in their rankings it is also important to evaluate the methods that got the worst or best rankings both from students and the model. It is interesting to check what kind of properties these methods have. The first three rows of Table 7 show the methods with worst maintainability indices while the last three rows enumerate the easiest to maintain methods together with their source code metrics.

Methods with the worst maintainability are as follows:

- *org.gjt.sp.jedit.browser.VFSDirectoryEntryTable.processKeyEvent(KeyEvent)*

  This is a very large and complex event handler method with many branches and deep control structure nesting. It also contains one serious and two suspicious coding rule violations. Our model ranked it as the worst method and the students gave it a very low ranking also. However, from the students point of view it is far not the hardest to maintain code. The reason of this might be that the large complexity is mostly caused by long but well structured *switch* statements. The students might perceived lower complexity than the McCabe's cyclomatic complexity metric would suggest.

- *org.gjt.sp.jedit.PluginJAR.checkDependencies()*

  This method checks the dependencies among different jEdit plugins. It is similarly complex than the previous method, however, its high complexity is caused almost entirely by nested *if...else* statements. The method is also very long and deeply nested *try...catch* blocks are also very common. The model ranked it as second while the students as fourth worst maintainable method accordingly.

- *org.gjt.sp.jedit.bsh.BSHPrimarySuffix.doSuffix(Object, boolean, CallStack, Interpreter*

  The method performs some kind of suffix operation. It is algorithmically complex with many type casts, self-defined parameter types and outgoing invocations. Despite the fact that the method is not drastically long and fairly well commented the students found it the hardest to maintain code part. The model ranked it also very low as the 8th worst method.

The best maintainable methods are the following:

- *org.gjt.sp.jedit.msg.PropertiesChanging.getState()*

  This method is a simple *getter* that returns the value of a private data member of type *State* which is an internal enum. Being one line long it is not surprising that both students and the model ranked it as the easiest to maintain method.

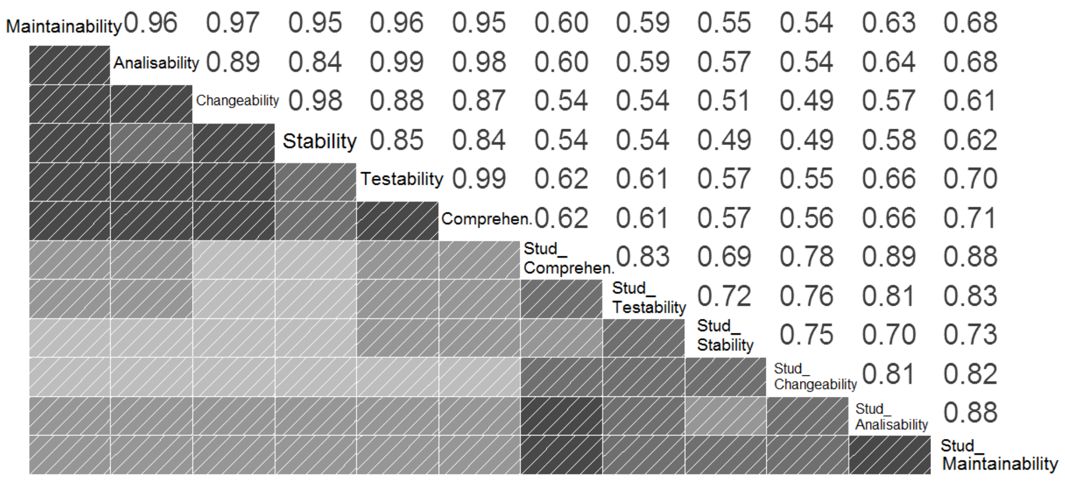
- *org.gjt.sp.jedit.bsh.UtilTargetError.UtilTargetError(Throwable)*

  This is the constructor of an error class in jEdit. It has only one line which is a call to the two parameter version of the constructor *UtilTargetError(String, Throwable)* with *null* as the first parameter. Its metric values are identical to that of the previous method and the model ranked it as the second best maintainable method accordingly. Although the students found the method to be well maintainable also, they ranked it only 13th. They might have taken into account the fact that changing this one line may cause changes in the called constructor also (i.e. they considered the NOI (number of outgoing invocations) metric). Probably it would be worthwhile to extend the model also with the NOI metric.

- *org.gjt.sp.jedit.buffer.JEditBuffer.getDefaultProperty(String)*

  This is again a very simple method. It returns a *null* value regardless of the parameter it gets. The students ranked it as the second best maintainable method while the model ranked it as fifth. The slight difference might be caused by the fact that the model took into account that the method has an incoming invocation which might be affected by a change in the *getDefaultProperty* method.

Correlation analysis of the model based and student assigned quality attributes. We also analyzed the correlations between the lower level quality attributes calculated by the algorithm and those assigned by the students. Figure 6 shows the diagram of the correlations. The names of the quality attributes are shown in the diagonal. The quality attributes starting with the *Stud_* prefix refer to the ones resulting from the manual evaluation; the rest of the attributes are computed by the model. In the upper right triangle of the diagram, the Spearman's correlation values of the different attributes can be seen. On the side, a graphical view of the correlation coefficients is presented, where the darker shade means a higher correlation. All the correlation values are significant at the 0.01 level.



Maintainability	0.96	0.97	0.95	0.96	0.95	0.60	0.59	0.55	0.54	0.63	0.68
	Analisability	0.89	0.84	0.99	0.98	0.60	0.59	0.57	0.54	0.64	0.68
		Changeability	0.98	0.88	0.87	0.54	0.54	0.51	0.49	0.57	0.61
			Stability	0.85	0.84	0.54	0.54	0.49	0.49	0.58	0.62
				Testability	0.99	0.62	0.61	0.57	0.55	0.66	0.70
					Comprehen.	0.62	0.61	0.57	0.56	0.66	0.71
						Stud_ Comprehen.	0.83	0.69	0.78	0.89	0.88
							Stud_ Testability	0.72	0.76	0.81	0.83
								Stud_ Stability	0.75	0.70	0.73
									Stud_ Changeability	0.81	0.82
										Stud_ Analisability	0.88
											Stud_ Maintainability

Figure 6: Correlations of the calculated and the manually assigned quality attributes

Based on the diagram, the following notable observations can be made:

- The dark triangle in the upper left corner shows that there is a very high correlation among the quality attributes calculated by our model. This is not surprising, since the model is hierarchical and the higher level attributes depend on the lower level ones.
- Similarly, the lower right corner shows a quite high correlation among the quality attributes evaluated by the students. However, the correlation coefficients are smaller than the coefficients among the attributes of the model. This suggests that students evaluated the

different quality properties somewhat independently from each other, not following the ISO/IEC 9126 standard's hierarchy strictly.

- Interestingly, the maintainability property evaluated by the students shows a slightly higher correlation with the algorithm-based approximation of comprehensibility and testability than with the maintainability value of the model. The reason might be that the comprehensibility and testability of the code are more exact concepts than the others, like stability, analyzability, etc. While comprehensibility is easier for the students to understand or evaluate, it makes them prone to equate maintainability and comprehensibility.

- It is promising that the model-based maintainability attribute shows the highest correlation with the maintainability property among the manually assessed ones. It means that our model interprets the quality on a broader scale than students do, i.e. it takes more factors into consideration. This might be because the students do not take into account all the hard-to-get-concepts of the ISO/IEC 9126 standard. According to our previous observation, the students tend to care the most about comprehensibility. This is in line with the fact that students prefer locally interpretable properties, like the lines of code or McCabe's complexity.

5 Threats to Validity, Limitations

Our validation is built on an experiment performed by students. This is a threat to the validity of the presented work due to the possible lack of their expertise and theoretical background. To compensate this drawback we selected third year undergraduate students who completed preliminary studies on software engineering and maintenance. Additionally, at least ten evaluations have been collected for each method to neutralize the effect of the possibly higher deviation in their votes.

For efficiency reasons we did not construct the goodness functions (probabilistic distributions, see Section 3) but computed only the goodness values for each node to obtain the relative indices. This might seem a loss of information, however, to get an ordering among the methods we should have derived the goodness values from the goodness functions anyway. Therefore, calculating only the goodness values is enough for our purposes.

The presented approach described in Section 3 assumes that the $\omega(t)$ weight function is equal to t for each sensor node (e.g. twice as high metric value means twice as bad code). This is not necessarily applicable for all the metrics (e.g. the very low and very high values of the Depth of Inheritance Tree metric is considered to be bad while there is an optimal range of good values). Our approach could be generalized to handle arbitrary weight functions, however, it would result a much more complex model. We decided to keep our approach simple and easy to understand as for most of the metrics (especially the ones included in our model) the applied weight function is reasonable.

The results might not be generalizable as we examined only 191 methods of one system. It required a huge amount of manual effort; performing more studies would require an even larger investment. This is actually, why the automatic maintainability analysis of source code elements is important. Despite the relatively small amount of empirical data, we consider the presented results as an important step towards the validation of our model.

6 Conclusion and Future Work

In this paper we presented the continuation of our previous work [BHK+11] on creating a model suitable for measuring maintainability of software systems according to the ISO/IEC 9126 standard [ISO01] based on its source code. Here, we further developed this model to be able to measure also the maintainability of individual source code elements (e.g. classes, methods). This allows the ranking of source code elements in such a way that the most critical elements can be listed, which enables the system maintainers to spend their resources optimally and achieve maximum improvement of the source code with minimum investment. We validated the approach by comparing the model-based maintainability ranking with the manual ranking of 191 Java methods of the jEdit open source text editor tool. The main results of this evaluation are:

- The manual maintainability evaluation of the methods performed by more than 200 students showed a high, 0.68 Spearman's correlation at $p < 0.001$ significance level with the model-based evaluation.
- Some of the differently ranked methods were manually revised and it turned out that humans take into consideration a wider range of local properties, while the model is able to explore the global environment more effectively.
- The Spearman's correlation of ISO/IEC 9126 attributes (model vs. manual) is high in general. This is especially true in the case of the maintainability property (as already mentioned).

As a future work we would like to perform more case studies to be able to generalize our findings. The current evaluation took only methods into account. We would like also to carry out a case study on manually evaluating the maintainability of classes and correlating the results with our drill-down approach.

Acknowledgements: This research was supported by the Hungarian national grants GOP-1.1.1-11-2011-0038 and GOP-1.1.1-11-2011-0006 and the TÁMOP 4.2.4. A/2-11-1-2012-0001 European grant.

References

[AYV10] T. L. Alves, C. Ypma, J. Visser. Deriving Metric Thresholds from Benchmark Data. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010)*. 2010.

[Azu96] M. Azuma. Software Products Evaluation System: Quality Models, Metrics and Processes – International Standards and Japanese Practice. *Information and Software Technology* 38(3):145–154, 1996.

[BBK+78] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. Macleod, M. J. Merrit. *Characteristics of Software Quality. Vol. 1.* TRW series of software technology. Elsevier, Amsterdam, Lausanne, New York, 1978.

[BD02] J. Bansiya, C. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering* 28:4–17, 2002.

[BG11] E. Bagheri, D. Gasevic. Assessing the Maintainability of Software Product Line Feature Models using Structural Metrics. In *Software Quality Journal 19(3):579-612*. Springer, 2011.

[BHK$^+$11] T. Bakota, P. Hegedűs, P. Körtvélyesi, R. Ferenc, T. Gyimóthy. A Probabilistic Software Quality Model. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)*. Pp. 368–377. IEEE Computer Society, Williamsburg, VA, USA, 2011.

[BHL$^+$12] T. Bakota, P. Hegedűs, G. Ladányi, P. Körtvélyesi, R. Ferenc, T. Gyimóthy. A Cost Model Based on Software Maintainability. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012)*. IEEE Computer Society, Riva del Garda, Italy, 2012.

[BSV10] R. Baggen, K. Schill, J. Visser. Standardized Code Quality Benchmarking for Improving Software Maintainability. In *Proceedings of the Fourth International Workshop on Software Quality and Maintainability (SQM 2010)*. 2010.

[BYM$^+$98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the 14th International Conference on Software Maintenance (ICSM'98)*. Pp. 368–377. IEEE Computer Society, 1998.

[CKV09] J. P. Correia, Y. Kanellopoulos, J. Visser. A Survey-based Study of the Mapping of System Properties to ISO/IEC 9126 Maintainability Characteristics. *IEEE International Conference on Software Maintenance (ICSM 2009)*, pp. 61–70, 2009.

[CLC09] L. Chung, P. Leite, J. Cesar. Conceptual Modeling: Foundations and Applications. In Borgida et al. (eds.). Chapter On Non-Functional Requirements in Software Engineering, pp. 363–379. Springer-Verlag, Berlin, Heidelberg, 2009.

[CV08] J. P. Correia, J. Visser. Benchmarking Technical Quality of Software Products. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE 2008)*. Pp. 297–300. IEEE Computer Society, Washington, DC, USA, 2008.

[Dro95] R. G. Dromey. A Model for Software Product Quality. *IEEE Transactions on Software Engineering* 21(2):146–162, 1995.

[FBTG02] R. Ferenc, Á. Beszédes, M. Tarkiainen, T. Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*. Pp. 172–181. IEEE Computer Society, Oct. 2002.

[Gra92] R. B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[HBI$^+$11] P. Hegedűs, T. Bakota, L. Illés, G. Ladányi, R. Ferenc, T. Gyimóthy. Source Code Metrics and Maintainability: a Case Study. In *Proceedings of the 2011 International Conference on Advanced Software Engineering And Its Applications (ASEA 2011)*. Pp. 272–284. Springer-Verlag CCIS, Dec. 2011.

[HKV07] I. Heitlager, T. Kuipers, J. Visser. A Practical Model for Measuring Maintainability. *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, pp. 30–39, 2007.

[HLSF12] P. Hegedűs, G. Ladányi, I. Siket, R. Ferenc. Towards Building Method Level Maintainability Models Based on Expert Evaluations. In *Proceedings of the 2012 International Conference on Advanced Software Engineering And Its Applications (ASEA 2012), accepted, to appear*. Springer-Verlag CCIS, 2012.

[ISO01] ISO/IEC. *ISO/IEC 9126. Software Engineering – Product quality*. ISO/IEC, 2001.

[ISO05] ISO/IEC. *ISO/IEC 25000:2005. Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*. ISO/IEC, 2005.

[MBD$^+$09] K. Mordal-Manet, F. Balmas, S. Denier, S. Ducasse, H. Wertz, J. Laval, F. Bellingard, P. Vaillergues. The Squale Model – A Practice-based Industrial Quality Model. In *Proceedings of the 25rd International Conference on Software Maintenance (ICSM 2009)*. Pp. 531–534. IEEE Computer Society, 2009.

[McC76] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering* 2:308–320, July 1976.

[MPKS00] S. Muthanna, K. Ponnambalam, K. Kontogiannis, B. Stacey. A Maintainability Model for Industrial Software Systems Using Design Level Metrics. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*. Pp. 248–256. IEEE Computer Society, Washington, DC, USA, 2000.

[MRW77] J. McCall, P. Richards, G. Walters. *Factors in Software Quality: Vol. 1: Concepts and Definitions of Software Quality*. Final Technical Report. General Electric, 1977.

[PMD] The PMD Homepage.
`http://pmd.sourceforge.net/`.

[San92] R. Sanders. The Pareto Principle: Its Use and Abuse. *Journal of Product & Brand Management* 1(2):37–40, 1992.

[SIG] Software Improvement Group.
`http://www.sig.eu/en/`.