Discovering Maintainability Changes in Large Software Systems*

Arthur Molnar
Babeş-Bolyai University
Cluj-Napoca, Romania
arthur@cs.ubbcluj.ro

Simona Motogna
Babeş-Bolyai University
Cluj-Napoca, Romania
motogna@cs.ubbcluj.ro

ABSTRACT

In this paper we propose an approach to automatically discover meaningful changes to maintainability of applications developed using object oriented programming languages. Our approach consists of an algorithm that employs the values of several class-level software metrics that can be easily obtained using open source software. Based on these values, a score that illustrates the maintainability change between two versions of the system is calculated. We present relevant related work, together with the state of research regarding the link between software metrics and maintainability for object oriented systems. In order to validate the approach, we undertake a case study that covers the entire development history of the jEdit open source text editor. We consider 41 version pairs that are assessed for changes to maintainability. First, a manual tool assisted examination of the source code was performed, followed by calculating the Maintainability Index for each application version. In the last step, we apply the proposed approach and compare the findings with those of the manual examination as well as those obtained using the Maintainability Index. In the final section, we present the identified issues and propose future work to further fine tune the approach.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; *Software version control*;

KEYWORDS

maintainability; maintainability index; object oriented metrics; case study

ACM Reference format:
Arthur Molnar and Simona Motogna. 2017. Discovering Maintainability Changes in Large Software Systems. In *Proceedings of IWSM Mensura, Gothenburg, Sweden, 2017,* 6 pages.
https://doi.org/10.475/123_4

*Produces the permission block, and copyright information

IWSM/Mensura '17, October 25–27, 2017, Gothenburg, Sweden

ACM ISBN 978-1-4503-4853-9/17/10…$15.00
https://doi.org/10.1145/3143434.3143447

1 INTRODUCTION

Software development evolved dramatically over the past decades, enabling the creation of increasingly complex applications. This in turn generated continued interest related to measuring and ensuring software quality. Large software systems are developed incrementally by teams over a long period of time. Methodologies such as Agile encourage dividing development activities over smaller time frames, with emphasis placed on always having working software at hand. In practice, this results in many application versions that allow shipping new functionalities to customers quickly. However, experience shows that maintenance of these systems becomes very expensive, with costs estimated at 50 % of total life cycle costs [28].

In order to address this issue, a clear understanding of maintainability, together with ways of measuring and controlling it are required. The IEEE guides [5] define maintainability as *"the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment"*. Starting from the definition above, we consider a maintainability change any modification of the source code that impacts how easily it can be fixed, improved or otherwise changed. As "you cannot control that which you cannot measure" [18], it becomes obvious that software metrics have value in assessing key software characteristics. Therefore, our approach employs mature and well-studied object-oriented metrics that offer numerical information about source code aspects such as hierarchical program structure, complexity, as well as information regarding dependency, in the form of cohesion and coupling.

The current study proposes a new approach based on collecting object oriented metric values that provide information regarding several aspects of source code, and observing the change in their values for consecutive versions of the target application. The reason for focusing on value differences is that most tools provide different thresholds when interpreting metric values. We find this to be a continued subject of differing scientific opinion [23].

The idea of using software metrics to produce an assessment of system maintainability is not new. An illustrative example is the Maintainability Index (MI) that is used as a baseline in our study. The MI has both its users [3, 8] as well as detractors, as many practitioners and researchers consider it to be of little relevance, or even obsolete [19], [27].

2 PRELIMINARIES

2.1 Software Metrics

Many software metrics have been proposed over the years, and their role in understanding and improving software quality is becoming more important nowadays, especially for large applications.

Some of the most widely used metrics simply count source code lines, functions, classes and modules in the program. Their simplicity and straightforwardness allowed them to remain relevant, or be easily adopted for new programming paradigms and languages. In our case study, we harness these metrics during manual source code examination to quickly assess which areas of the application underwent most significant change.

The appearance of object oriented programming languages lead to the definition of specialized metrics, that capture aspects related to object orientation. The Chidamber & Kemerer (CK) metrics suite [17] is one of the most popular, and is implemented in most of the available metric tools [6, 13]. Object oriented metrics are classified as describing four internal characteristics of the object oriented paradigm: inheritance, structural complexity, coupling and cohesion [23].

The study takes into consideration a subset of the CK metrics suite, from which one metric is selected for each of the four internal characteristics presented above. The selection ensures that relevant characteristics of the system are captured. Selected metrics are at class level in order to facilitate fine grained analysis. The selected metrics are:

- *Depth of Inheritance Tree* (DIT) [17] defined as the length of the longest inheritance path between a given class and the root of the tree; DIT is a structural metric.
- *Weighted Methods per Class* (WMC) [17] defined as the sum of the cyclomatic complexities of all methods of a given class.
- *Coupling Between Objects* (CBO) [17] representing the number of other classes that are coupled to the given class. Namely, two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.
- *Lack of Cohesion in Methods* (LCOM4) [20] defined by the difference between the number of method pairs using at least one common instance variables and the number of method pairs that do not use any common variables.

The well established metric for maintainability is Maintainability Index (MI), introduced in 1992 [26] as a measure of how easy it is to maintain a program. Several metric and code inspection tools can compute the MI, including JHawk [3], SonarQube [11], the Metrics .NET library[9]. The Maintainability Index (MI), first introduced in [26], was defined as:

$MI = 171 - 5.2 * ln(aveV) - 0.23 * ave(G) - 16.2 * ln(aveLOC)$, where:

- aveV = average Halstead volume
- ave(G) = average cyclomatic complexity
- aveLOC = average number of source code lines.

The MI depends on the size of the program, expressed as number of lines and two complexity indicators. The Halstead volume refers to the number of operands and operators, and the cyclomatic complexity refers to the number of possible code paths. Other aspects critical for object oriented programs, such as structure, cohesion and coupling are not taken into consideration.

There are also studies that criticize the MI [19, 29]. The main problems regarding it are summarized as follows: first, tools that compute the MI use different formulas, which, in consequence, implies that they introduce different thresholds; even more, the ranges are not well defined. For example, the formula used in Visual Studio resettles the MI into the [0,100] range [8]. The thresholds used show bad maintainability for scores between 0 and 9, moderate for scores between 10 and 19, and good maintainability for scores above 20 [8]. Second, programming languages have evolved significantly since the 90's, and their constructs and relations between them have changed. Finally, by computing the MI at package or application level, the coarse granularity might not offer adequate solutions for identifying the code that contributes most to low maintainability.

2.2 Related work

The relation between object oriented metrics and maintainability has presented a lot of interest in research studies. Several studies have illustrated this relation and took into consideration the CK, as well as other object oriented metrics [21], [23], [19]. Even if [21] proved that "the prediction of maintenance effort metric is possible", no such formula has been proposed based on object oriented metrics. [22] proposes a comprehensive compendium of the relation between OO metrics and ISO 9126 quality factors, including maintainability. It offers valuable information that was used in the presented study, but only at descriptive level.

The main difference between these results and the proposed approach is that we do not focus on threshold values, but on the modification of the metric values over time. We believe that observation of the changes in metric values and their cumulative impact on maintainability offer useful information when dealing with multiple versions of large-scale applications.

3 DISCOVERING MAINTAINABILITY CHANGES

3.1 Approach

This paper proposes a new methodology for detecting meaningful changes to object-oriented software systems. The idea is based on recent work that explores the relation between metrics and desirable properties of software systems on one hand [15, 22, 24], and the availability of software tools to calculate advanced object-oriented metrics at class, package and system level [6, 11, 13], on the other.

Much of the existing work on the topic was undertaken during the domination of the procedural paradigm, where counting the number of modules, functions and their cyclomatic complexity could provide a good estimation of the system's complexity and soundness. However, as illustrated by [12], out of the 5 most popular programming languages today, four (Java, C++, Python and C#) allow building large-scale systems that leverage advanced object-oriented concepts. In this case, we find that existing results do not take into consideration aspects specific to the development of object-oriented systems, such as the depth of inheritance hierarchies, use of encapsulation, or the level of coupling and cohesion between system modules.

The main objective of our work is to detect meaningful change between any two given versions of a software system, and provide a clear and simple representation of this change in numeric format. To achieve this, our methodology employs the object oriented metrics presented in the previous section. The selection of representative metrics was guided by the ARISA's Software Compendium [22], which provides the influence metrics have on the maintainability

Algorithm 1 Maintainability Change

Require: sv1, sv2; {software versions}
ms; {metric set: DIT, WMC, CBO and LCOM}
{m(c,sv) = value of metric m for class c in version sv }
If (c ∈ sv2) and (c ∉ sv1)
then m(c,sv1) = 0, ∀ m ∈ ms

```
mc = 0
for (metric m ∈ ms) do
  for (class c ∈ sv1 ∪ sv2) do
    if m(c, sv1) < m(c, sv2) then
      mc = mc - 1
    end if
    if m(c, sv1) > m(c, sv2) then
      mc = mc + 1
    end if
  end for
end for
return mc
```

of software. Metric selection includes the following metrics that operate on class level: DIT, WMC, CBO and LCOM. Each of them represents one of the size & structure, complexity, cohesion and coupling characteristics of software. Selected metrics are presented as being highly related with software's maintainability characteristic [22]. Of course, there also exist others metrics that influence maintainability. As object-oriented systems are perceived as a collection of objects passing messages between them, rather than a collection of functions we chose not to include counting metrics such as code line, method or class count. Other non included metrics are related with metrics that have been included. In this category sit the TCC [15], that is related with LCOM as well as cyclomatic complexity, which is included in the WMC value. Other metric values are ambivalent regarding maintainability, such as the number of children. These can illustrate both positive as well as negative effects and are difficult to ascertain automatically. In the same category we place metrics for code comments and documentation, as they are only able to provide information regarding the presence and size of these elements, not their quality [27].

In contrast to the MI and other such endeavors, our approach is geared towards assessing the change in maintainability that occurs during software development. To this purpose, we developed an algorithm that when provided with values for the selected metrics for two versions of a software system, will provide the maintainability change between them, in the form of an integer number. A negative result implies decreased maintainability, with a positive result implying an increase. When running the algorithm for the same software version, the result will be 0. Algorithm 1 provides the pseudo code implementation of our approach.

The algorithm works by comparing the values for the provided metrics for all classes that appear in at least one of the software versions considered. For classes that do not appear in one of the versions, a default value of 0 is considered. For each instance where a metric value increases, the maintainability change score is decreased by 1. The opposite is done when metric values decrease. The effect is that the score is decreased for every $(metric, class)$ pair where the metric score has increased, and increased when the opposite happens. The end result is the maintainability change score, to which all metrics account with the same weight.

3.2 The Case Study

We evaluate our approach using the jEdit text editor [7]. jEdit is open source under the GPL, developed entirely using the Java programming language and straightforward to set up both in regards with its development environment as well as binary code. jEdit was also a popular system under test for previous research focused on application testing [14, 30] and program comprehension [25]. Furthermore, jEdit enjoys a large user base, having over 91k downloads over the first half of 2017, and over 5.5 million downloads over its 17 year lifetime. Most importantly, the project has a full development log on SourceForge, allowing access to all existing versions.

For the purposes of our case study, all versions released by the developers are employed. This amounts to a total of 42 versions, released between January 2000 and March 2017. All versions are available for download on the project website [1]. We must note that some of the versions carry the *"pre"* moniker, indicative of interim versions, or the *"final"* moniker, which indicate the final release for a given version. However, all released versions are included in our case study. Given that the proposed approach does not rely on information outside of source code, we will not examine possible relations between version numbers and changes to maintainability. Version numbers are used only to uniquely identify each version.

Using only released versions ensured that the application was complete and included all required modules. As the objective was to study an approach for maintainability change, we decided to carry out the assessment as a longitudinal study that targets several versions of a complex application. It is important to note that as in the case of many open-source projects, development work on jEdit was not consistent over the 17 years considered.

To prepare the case study, each version was compiled and checked for code and functionality completeness. Each version's source code was examined and many were found that included third party library code, such as GNU regex [4], Beanshell [2] and ASM [10]. Library code was factored out to ensure metric results are not skewed. DIT, WMC and LCOM values were recorded using the Metrics2 Eclipse plugin [13], while CBO was retrieved using IntelliJ's Metrics Reloaded [6]. In the general case, using several tools to extract metric values can be problematic, given that differences in tool implementation can lead to inconsistency between obtained values. The proposed approach avoids this issue by only comparing values that belong to the same metric, which were always extracted using the same tool.

3.3 Evaluation

The evaluation is undertaken in three steps. In each step, jEdit's evolution is evaluated using consecutive version pairs, of which there are a total of 41. The analysis is detailed in the following sections. The first step was to manually inspect the source code of the targeted versions and identify those versions that brought the most significant changes to the software.

[1] https://sourceforge.net/projects/jedit/files/jedit/

In order to achieve this, we use a combination of manual source code inspection and the output of established software metrics, including the number of packages, classes, number of lines of code and cyclomatic complexity. The next step consists of comparing our source code inspection with information gained by calculating the MI. The objective is to discover how significant changes to software are discoverable using the MI. In the last step, we apply our proposed algorithm and compare the obtained results with those of the manual code inspection and those obtained using the MI. The entire raw data set, together with the source code used to process it are available on our website [2].

3.3.1 Source code inspection. We started by studying how jEdit source code evolved during development. The first version publicly released is 2.3pre2 and consists of 322 classes that comprise over 22k lines of code, organized in 10 packages. These numbers steadily increase throughout the application's development, as illustrated in Figure 1. The latest considered version is 5.4.0 and consists of 93k lines of code, organized into 957 classes and 29 packages.

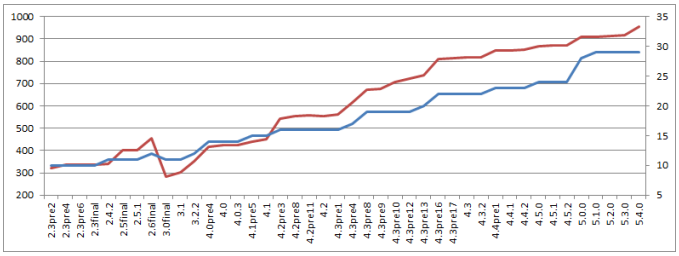

Figure 1: Package (blue, scale on right) and class (red, scale on left) numbers for application versions

A first observation is that the application core was stabilized before the first publicly released version. This is illustrated by the fact that all 10 packages from version 2.3pre2 are found in all subsequent versions, up to and including 5.4.0. Furthermore, many of the application's core classes suffered changes, but were present in the same core package across many application versions. However, as it was expected, most of the core classes increased in complexity, by the addition of methods and code, illustrated by the increasing trend in the values of the recorded metrics.

The results of the initial examination are presented in Table 1. From the 41 version pairs, 11 pairs were selected that were identified as containing the most significant changes. Given that the size of the data set is over 26k classes, several metrics were extracted for each application version to help the analysis. While the present paper only details the changes we identified as important, the entire data set is available on our website [1].

3.3.2 The Maintainability Index. In this section we analyze the value and change in the system's MI for the studied application versions. For each version, the Maintainability Index is calculated using the formula from the previous section of this paper, with one change. Instead of counting source code lines, we employ the number of statements, as this appears to be a better indicator for languages such as Java [3]. The additional term considering source code comments is also not considered, as the quality of comments cannot be assessed automatically. The MI was calculated using our own implementation, that employs values extracted using the Metrics Reloaded plugin [6].

Our analysis is backed by the manual examination presented in the previous section. Figure 2 illustrates that the system's MI remains within the [67,78] range across all versions. Related literature [16] as well as tools [8] would assess the system as having good maintainability throughout its development history. However, as Figure 2 illustrates, we identify two points where the MI values indicate large changes: version 2.4.2, and version 3.0final, where the MI value increases, and significantly decreases, respectively. Analyzing the source code changes presented in Table 1 reveals that manual examination also identified the changes. In the case of version 2.4.2, removing complex code was expected to increase the reported MI. However, we observe that the largest change to the MI was a decrease of 9 points that was actually driven by the removal of 153 application classes in version 3.0final. As shown in Table 2, the removal of a large number of classes consisting of low-complexity code resulted in increased average Halstead volume and average number of statements, which actually lowered the system's MI.

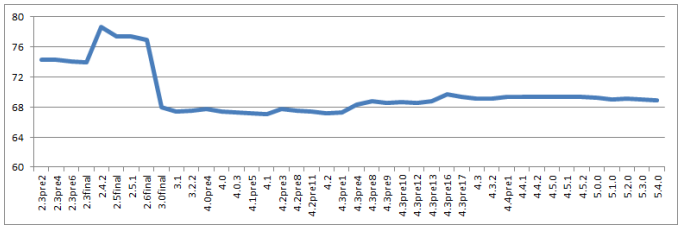

Figure 2: Maintainability Index for jEdit versions

We believe this to be a suitable example regarding the limitations of the MI in assessing changes to complex software systems. Examining the versions shows that jEdit 3.0 has 171 fewer classes and 3000 fewer lines of code than 2.6, while the rest of the system's architecture did not undergo significant changes. While we believe that it is difficult to assess exactly how much system maintainability changes when eliminating a large number of classes having similar function, it is hard to argue that this results in a marked increase in maintainability. Furthermore, it can be observed that none of the other 9 instances of major changes that were presented in Table 1 lead to significant modifications of the MI.

In fact, all versions after 3.0 had a MI in the [67, 70] range, with all components of the metric showing rather small variation. In more detail, the Halstead volume stabilized in the [1650, 1890] range, the average cyclomatic complexity in the [2.46, 2.73] range, and the average number of statements remained between 47 and 52. Between these versions, the code base grew from 282 classes and 27k lines of code to the 957 classes and 93k lines of code in the latest version. This, coupled with the data presented in the previous section illustrates the intrinsic limitations of the MI. When used as a global indicator, it cannot provide finer grained detail at package or module level, especially when many changes in the target application were focused on particular packages.

[2] https://bitbucket.org/guiresearch/tools

Table 1: Summary of meaningful source code changes identified using tool-assisted manual examination

Version	**Significant Changes**
2.4.2	21 classes containing parser code for various programming languages having high statement number and cyclomatic complexity removed from *"org.gjt.sp.jedit.syntax"* package, replaced with a set of XML files and a unified implementation. 25 classes were removed in this version, and 44 others added.
2.5final	Over 70 new classes that provide new features, including GUI improvements, several editor panes as well as support for FTP and VFS. Most of the major changes are concentrated in a small number of source files which include many classes and have high statement count and cyclomatic complexity.
2.6final	Added package *"org.gjt.sp.jedit.browser"* that includes complex code for VFS browsing, together with required event handlers and I/O support. From a source code perspective, there are several new source files containing over 100 new classes and code having high complexity. Most of the newly-introduced classes are not public, so the newer version only adds 44 new source files.
3.0final	Package *"org.gjt.sp.jedit.actions"*, which contained 153 event handler classes with low statement count and cyclomatic complexity was deleted.
3.2.2	Package *"org.gjt.sp.jedit.pluginmgr"* adds a new plugin manager. New GUI components added to relevant packages. 64 classes were added, with relatively little complex code removed.
4.0pre4	Code with high cyclomatic complexity added for text area management. The *"org.gjt.sp.jedit.buffer"* package was added, containing code for text buffer events and the application's document model. Some of the code for text area management was removed. As intuitively expected, this remains a complex area of jEdit, with several of the largest classes in terms of line count and cyclomatic complexity.
4.2pre3	The *"org.gjt.sp.jedit.menu"* was added, together with changes to the plugin management, text area and GUI components. These changes resulted in 133 new classes, while 43 classes were removed, either due to refactoring or replaced by newer implementation.
4.3pre4	Included large changes to application buffers and GUI components in the form of 75 newly added classes.
4.3pre8	New code added to the *"org.gjt.sp.jedit.textarea"* and *"org.gjt.sp.jedit.util"* packages, together with the introduction of two new packages. Little code was deleted to offset the newly added or reworked functionalities.
4.3pre16	Two new packages were added, together with new code in the *"org.gjt.sp.jedit.textarea"* and *"org.gjt.sp.jedit.gui"* packages.
5.0.0	Four new packages were added, but they do not contain complex code. However, 385 of the 415 source code files in the previous application version were updated, bringing changes throughout the application source code. Like in the previous version, very little code was removed.

Table 2: Maintainability Index between versions 2.6 and 3.0

Version	**MI**	V	G	LOC
2.6	**76.93**	1045	2.49	34.44
3.0	**67.95**	1647	2.48	51.81

In addition, we observe that since most of the application's core packages and modules that were present in the 2.3pre2 version also appeared in most, if not all subsequent versions indicates that the developers managed to establish a sensible architecture, on which many additional features could be introduced.

3.3.3 Assessing Maintainability Change. For the second part of our evaluation, we apply algorithm 1 described in the previous section to the 41 consecutive version pairs of jEdit. The results are illustrated in Figure 3. What interests us most is to ascertain whether the proposed algorithm provides a finer grained result regarding changes in software when compared with both the manual examination as well as the MI.

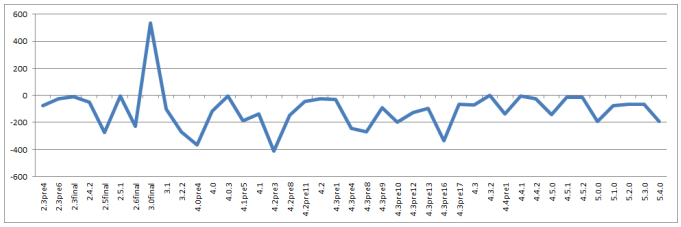

Figure 3: Maintainability change between consecutive jEdit version pairs

Figure 3 shows the maintainability change between the considered version pairs. Application versions are shown on the X axis, while the maintainability change is shown on the Y axis. As such, the first data point shows the changes between version 2.3pre2 and 2.3pre4, and so on. The last data point illustrates the maintainability change between versions 5.3.0 and 5.4.0. A value of 0 indicates no change, while positive and negative values indicate increased and decreased maintainability, respectively. Larger values are indicative of bigger changes in both directions.

The first observation is that for all but one version, the algorithm shows a decrease in maintainability. Comparing the maintainability change scores obtained using our algorithm with the changes identified by manual source code examination, we observe a high degree of overlap. More precisely, out of the 11 significant changes identified in Table 1, only the changes in version 2.4.2 are missed by our algorithm. In this case, the advantage of removing highly complex code was balanced by adding numerous classes containing simple code. Given that assessing software maintainability in the case of complex source code changes remains an open problem, providing an objective comparison between MI and our approach can only have an empirical basis. However, we believe that in this case, the improvement indicated by the MI is not one that can actually be leveraged. The changes to jEdit version 3.0final also present interest, as the MI and our approach indicate the opposite. While the MI records its largest decrease among the available versions,

our approach indicates the largest increase in maintainability. As discussed in the corresponding section, the MI drops due to increasing average values produced by removing a large number of classes containing low complexity code. In opposition to the MI, our approach correctly identifies that maintainability is improved when removing roughly 30% of the total number of classes.

When compared to the MI, the proposed algorithm produces more evident changes between the versions. This is due to the fact that the approach employs neither coefficients nor functions such as the MI, which makes changes to maintainability appear more pronounced. Also, we observed that continued development on jEdit, that resulted in a large increase of the code base was captured more accurately using our approach. Starting with application version 3.0final, both the MI and our approach showed limited changes to the jEdit code base. This is also in agreement with the manual source code examination, and indicates that the application has matured. The latest version that we evaluated to include major changes was 5.0.0, in which code was updated across most modules, but very few modules were added.

3.3.4 Threats to Validity. The first threat to the validity of our study comes from the limited size of the sample data. While we could not find related work that provides a similar longitudinal view, we are aware that to validate our approach, thorough evaluation is required targeting different application types and development platforms. Another issue concerns the objective evaluation of maintainability change. As such, evaluating both the MI and our approach includes the subjective dimension of the manual source code examination. Furthermore, while selection of metrics was to ensure that key software characteristics are captured, further extended study is required to ascertain the role and importance of each metric. This is also one of the key grievances for the MI, where many are doubtful of its accuracy [27, 29].

4 CONCLUSIONS AND FUTURE WORK

The paper proposes a method to discover and assess meaningful changes affecting the maintainability of large scale object oriented software systems. Our approach is based on the four internal characteristics of object orientation: inheritance, structural complexity, coupling and cohesion [23]. We select a representative metric for each characteristic, and evaluate the result using the entire development history of the jEdit text editor.

Tool-assisted manual examination found that as expected, jEdit's 17 year development history includes several major changes, as many application features were added, reworked and refactored. We identified the most important such changes, using software tools to double-check the assessment and presented them in Table 1. Calculating MI revealed both its strength as well as weakness. While for many version pairs it remained in agreement with our assessment, its largest score variation was counter to our examination. The index showed a marked decrease in maintainability, where our assessment showed there was none. The proposed approach proved to be more sensitive to source code changes than the Maintainability Index. Furthermore, we found it more accurate when many classes were added or removed from the system.

Our evaluation also illustrated where to focus future work. We believe that the inclusion of additional metrics, as well as taking into account the magnitude of change between versions will bring results closer to those obtained using manual source code evaluation. Further more, we aim to include other applications as case studies, together with undertaking more extensive manual source code examination, given its inherent degree of subjectivity. This will allow us to better ascertain the magnitude of changes as well as a finer grained identification of modules in need of refactoring.

REFERENCES

[1] Artifacts supporting the paper. https://bitbucket.org/guiresearch/tools.
[2] Beanshell library. http://www.beanshell.org/.
[3] Discussion on measuring the maintanability index. http://www.virtualmachinery.com/sidebar4.htm.
[4] GNU Regex library. https://www.gnu.org/software/regex/.
[5] IEEE standard glossary of software engineering terminology. http://standards.ieee.org/findstds/standard/610.12-1990.html.
[6] Intellij Metrics Reloaded. https://plugins.jetbrains.com/plugin/93-metricsreloaded. Online; 15- Oct- 2016.
[7] jEdit text editor. http://jedit.org/7.
[8] Maintainability Index in Microsoft Visual Studio. https://blogs.msdn.microsoft.com/zainnab/2011/05/26/code-metrics-maintainability-index/.
[9] .NET Metrics library. https://github.com/etishor/Metrics.NET. Online; 15- Oct- 2016.
[10] OW2 ASM library. http://asm.ow2.org/.
[11] Sonar. https://www.sonarqube.org. Online; 15- Oct- 2016.
[12] TIOBE index for june 2017. https://www.tiobe.com/tiobe-index/.
[13] Eclipse Metrics2 plugin, "metrics 1.3.6". http://metrics.sourceforge.net., 2016.
[14] Arlt, S., Banerjee, I., Bertolini, C., Memon, A. M., and Schäf, M. Grey-box GUI testing: Efficient generation of event sequences. *CoRR abs/1205.4928* (2012).
[15] Bieman, J., and Kang, B. Cohesion and Reuse in an Object-Oriented System. *ACM Symposium on Software Reusability* (1995).
[16] Bray, M. C4 software technology reference guide, 1997, carnegie mellon university.
[17] Chidamber, S. R., and Kemerer, C. F. A metrics suite for object-oriented design. *IEEE Trans. Soft Ware Eng. 20*, 6 (1994), 476–493.
[18] DeMarco, T. *Controlling Software Projects: Management, Measurement, and Estimates.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1986.
[19] Heitlager, I., Kuipers, T., and Visser, J. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 6th International Conference on the Quality of Information and Communications Technology, QUATIC 2007, Lisbon, Portugal, September 12-14, 2007, Proceedings* (2007), pp. 30–39.
[20] Henderson-Sellers, B. *Object-oriented Metrics: Measures of Complexity.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
[21] Li, W., and Henry, S. Maintenance metrics for the object oriented paradigm. *IEEE Proc. First International Software Metrics Symp* (1993), 52–60.
[22] Lincke, R., and Lowe, W. Compendium of software quality standards and metrics. http://www.arisa.se/compendium/quality-metrics-compendium.html, 2013. Online; accessed 7 March 2017.
[23] Marinescu, R. *Measurement and Quality in Object Oriented Design.* PhD thesis, Faculty of Automatics and Computer Science, University of Timisoara, 2002.
[24] Martin, R. OO Design Quality Metrics - An Analysis of Dependencies. In *Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics* (1994), OOPSLA'94.
[25] Molnar, A. JETracer - A framework for java GUI event tracing. *CoRR abs/1702.08008* (2017).
[26] Oman, P., and Hagemeister, J. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992* (Nov 1992), pp. 337–344.
[27] van Deursen, A. Think twice before using the maintainability index. https://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index/.
[28] Vliet, H. v. *Software Engineering: Principles and Practice*, 3rd ed. Wiley Publishing, 2008.
[29] Welker, K. D. The software maintainability index revisited. http://web.archive.org/web/20021120101304/http://www.stsc.hill.af.mil/crosstalk/2001/08/welker.html.
[30] Yuan, X., and Memon, A. M. Generating event sequence-based test cases using gui runtime state feedback. *IEEE Transactions on Software Engineering 36*, 1 (Jan 2010), 81–95.