Experimental Study of Quantitative Analysis of Maintenance Effort using Program Slicing-based Metrics

Raula Gaikovina Kula∗, Kyohei Fushida†, Norihiro Yoshida∗ and Hajimu Iida∗

NAIST, JAPAN ∗
Research and Development Headquarters, NTT DATA, JAPAN †
raula-k@is.naist.jp, fushidak@nttdata.co.jp, yoshida@is.naist.jp, iida@itec.naist.jp

Abstract—During the software development lifecycle, studies have shown that over 75% of project costs originate from the maintenance phase. Analysis of the processes within the maintenance phase could prove beneficial since most maintenance activities revolve around source code. Accurate estimations of the maintenance effort spent on code changes would enable cost effective management of resources.

In this research, we investigate a quantitative approach to express maintenance effort, for which a set of program-sliced metrics is proposed. Using the time to resolve an issue as a measure of maintenance effort, we evaluated our proposed metrics against the basic code-based metrics Lines of Code and McCabe’s Cyclomatic Complexity. To eliminate outside factors, we performed an experimental case study on a set of pre-defined maintenance activities. Results suggest that program slicing metrics have the strongest correlation with maintenance effort, exhibiting a moderate degree of correlation with maintenance effort. In contrast, Lines of Code has a weak correlation with maintenance effort. This study contributes to our ongoing research into the analysis of maintenance processes.

Keywords-Micro Process; Software Metrics; Program Slicing; Software Maintenance;

I. INTRODUCTION

Studies have shown that the maintenance phase of a software development lifecycle consumes a substantial amount of time and effort as compared to the other development phases [1], [2]. Conventionally, to reduce these costs, most organisations employ Software Process Improvement (SPI) activities. However, more traditional process assessment models such as the CMMI [3] are generic and tend to cover all phases rather than focusing on a particular phase. To address this shortfall, in this paper, we primarily focus on the analysis specifically at the maintenance phase.

The purpose of this research is to investigate a quantitative approach of the analysis of maintenance processes. Other models such as the ISO/IEC 15504 Software Process Improvement and Capability Determination (SPICE) assessment model [4], does address some of the prior mentioned shortfalls of CMMI. SPICE metric results are based on ratings (process attribute rating and attribute indicators ratings). However, we propose a much quantitative approach, which is more fine-grained than SPICE. To address the high costs and generic features of CMMI, work similar to Pino [5] suggest processes and models that focus on SPI for Very Small Entities (VSE - smaller organisations with less than 25 employees). These models are effective, however, as mentioned by Colla [6], much like SPICE, the focus is often placed at qualitative or methodological factors rather than quantitative.

Much of the motivation and foundation of this work is built on our previous work [7], where a quantitative approach to study the fine-grained processes of bug fixing was investigated. Using three open source projects, we identified a quantitative relationship between the time to fix a bug with our proposed metrics. Most open source projects record maintenance effort coarsely as the number of days, which we found can be affected by many outside factors such as priority, developer’s workload and the maintenance work-flow (process) complexity. We eliminated these factors using a controlled environment setting.

In this paper, we defined these fine-grained processes as micro processes. By mining software repositories, sufficient data can be extracted to reconstruct the micro processes related to maintenance activities. We can then exploit these micro processes to express maintenance effort, which is defined as the time and resources spent towards the resolution of issues relating to defects, undesirable behaviour, enhancements, as well as the general maintenance of the software. The changes made to the source code in relation to the issue are referred to as code changes.

According to one of Lehman’s laws of evolution [8], we assume that higher maintenance effort is most likely caused by complex code. We believe that the analysis of source code properties and how they correlate to the maintenance effort during the resolution of an issue could show this quantitative. To test this theory, we aim to answer the following research questions:

- **RQ1.** Is there a quantitative relationship between maintenance effort and source code properties?

  Assuming that there is an evident relationship from RQ1, we then constructed the next research question:

- **RQ2.** Can we define a set of code-based metrics from this relationship?

Since there is a lack of reliable metrics to accurately measure maintenance [9], [10], we based and compared our
The purpose of this research is to investigate an alternative quantitative approach of the analysis of maintenance processes. We utilize program slicing for the measure of change impact analysis. Therefore, related work covers a range of fields, with each having a slightly different motivation, objective and approach. Due to space limitations we have only briefly covered each field.

There has been similar work on maintenance effort that studied the effort spent to fix bugs [12]. The authors of this work estimated the effort to fix an issue based on prior related issues, applying time as an indicator for effort. Also, the work by Kim [13] refers to time to fix bugs as an important factor.

Many of the metrics widely used in the field of program slicing are related to the evolution of code [16], [17]. Among those, many are cohesion and coupling based approaches. Similar research has used program slicing metrics to classify bugs using these metrics [18], [19]. Instead of the standard slicing metrics, we propose to enhance the basic code based metrics.

Program Slicing is well known in the field of change impact analysis. Gallager [21] illustrated its usefulness as it assists program comprehension, more specifically guiding developers to determine which code components are not related to a software change. Similar to this, Differential Symbolic Execution (DSE) characterizes the effects of a set of program changes in terms of behavioral program differences [22]. There also has been research to predict if a software change is clean or buggy [23]. Canfora applies program slicing also to indexing changes [24]. Li’s work [25] is more related to software processes which use used change impact analysis for requirements traceability. We also use program slicing techniques, however, the main difference in our work is that we assess the processes, specifically at the maintenance phase.

III. APPROACH OF THE STUDY

In this study, we decided to use a controlled environment to cover the different types of issues. There are several advantages: (1) Using a set of pre-determined issues allows to expose the different types of typical code changes. (2) In addition, we can pre-determined the complexity of the code change. (3) We can measure accurately the duration taken to resolve the issue. Normally, the duration to resolve an issue can be influenced by several factors such as the bug fixing process or work-flow, workload of the developer and in some instances the priority of the issue. This study specifically ignores all these factors, so that the maintenance effort is solely based on the time given to resolve.

First, we introduce the proposed metrics used to measure the source code properties of the code changes. Later in the section, we introduce the experiment with the four issues that were used in the experiment and the feedback questionnaire.

A. Proposed Metrics

To evaluate the program behavioural properties of the affected portions of the code, we wanted to measure both complexity and size of code changes. We chose to based
our proposed metrics on the most fundamental of code-based metrics [26]. McCabe’s Cyclomatic Complexity (CC) is a well-known software metric that indicates complexity [27]. To measure the size, instead of Lines of Code (LoC), which is another well-known software metric, we found it appropriate to also measure size by counting the number of functions, referred to as Function Count (FC). FC is another common metric that is used in the ISO/IEC 20968 and ISO/IEC 14143 for measurement of software size [28], [29]. To measure the size, instead of Lines of Code (LoC), we found it appropriate to also measure size by counting the number of functions, referred to as Function Count (FC). FC is another common metric that is used in the ISO/IEC 20968 and ISO/IEC 14143 for measurement of software size [28], [29]. Since we are using FC, we propose to program slice at the function level for the most accurate results.

To measure the change impact of the code changes, we introduce three metrics based on the number of functions affected by the code change in revision $r$.

- **EditedFC** (FC) The number of functions edited during code change, i.e., $|F_{E,r}|$. 

  **Rationale**: This is a non-program slicing metric for comparison.

- **BackwardSliceFC** (BSFC) The number of functions in $S_{B,r}$, i.e., $|S_{B,r}|$. 

  **Rationale**: This metric computes the number of functions that depend on $F_{E,r}$.

- **ForwardSliceFC** (FSFC) The number of functions in $S_{F,r}$, i.e., $|S_{F,r}|$. 

  **Rationale**: This metric computes the number of functions depending on functions in $F_{E,r}$.

To generate our metrics, the software analysis tool by GrammaTech called CodeSurfer [30], arguably the most sophisticated and widely used tool for interprocedural slicing, was used [18], [31], [32]. Customized scripts within CodeSurfer were used to calculate the CC and FC within the slices.

### B. Pilot Experiment - AlignMe

Software maintenance and evolution is inevitable, with constant changes to code. According to Yu [33], there are three main software maintenance activities: (a) Perfective - the addition of a new functionality, (b) Corrective - fixing of faults or bugs to the software and (c) Adaptive - new file formats or refactoring code. In this study, we only focused...
Table I
ALIGNMe FUNCTION DESCRIPTIONS

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td>runs the program, displays menu to console and calls up the Align instance</td>
</tr>
<tr>
<td>Align::doIt()</td>
<td>calls up Strategy::format() function</td>
</tr>
<tr>
<td>Align::setStrategy(int)</td>
<td>based in input creates either an instance of RightStrategy or LeftStrategy</td>
</tr>
<tr>
<td>Strategy::Strategy()</td>
<td>constructor for strategy class</td>
</tr>
<tr>
<td>Strategy::format()</td>
<td>reads the text from quote.txt and calls justify(char*) for either RightStrategy or LeftStrategy</td>
</tr>
<tr>
<td>Strategy::justify(char*)</td>
<td>virtual function is implemented by either RightStrategy or LeftStrategy</td>
</tr>
<tr>
<td>RightStrategy::RightStrategy()</td>
<td>constructor</td>
</tr>
<tr>
<td>RightStrategy::justify(char*)</td>
<td>implements the specific right alignments for the text</td>
</tr>
<tr>
<td>LeftStrategy::LeftStrategy()</td>
<td>constructor</td>
</tr>
<tr>
<td>LeftStrategy::justify(char*)</td>
<td>implements the specific left alignments for the text</td>
</tr>
</tbody>
</table>

Figure 2. AlignMe Class Diagram. Note that main() is used to instantiate the Align class

Figure 3. Screenshot of ideal solution for Issue 1. This should be the output with the center option is selected

on the first two activities due to two reasons. Firstly, due to time limitations of the experiment. Secondly, because of the difficulty in designing adaptive maintenance issue. We designed four issues: Issues 1 and 2 were perfective activities, while issues 3 and 4 addressed some corrective activities.

Other challenges encountered involved defining a set of issues were that they had to be solved within a time-frame by novice users to both the program and the programming language. Taking these factors into account, we developed the testing program AlignMe. For easier code understanding and appending new enhancements to the program, we incorporated software design patterns into our program.

AlignMe is a console-based application written in C++. It was developed and run in Visual Studio 2008 Environment on the Windows 7 platform. The main function of AlignMe is to display text from a stored file (quote.txt) in either a selectable Left or Right alignment. AlignMe has a total of 78 LoC with 4 classes consisting of 10 functions. Table I shows a summary of the functions and their descriptions. The class designs in Fig. 2, illustrate how the strategy design pattern is implemented. The strategy design pattern is commonly used to improve the readability and maintainability of source code. As seen class Strategy is the parent of classes RightStrategy and LeftStrategy.

The following are the maintenance activity issues designed for the experiment:

- **Issue 1 - Create a center alignment** At the current state, program offers the two options of either right or left alignment. Participants are asked to modify the program to add a center alignment. Fig. 3 illustrates the expected output. This is a perfective activity, which the ideal solution involves extension of the strategy design template to include a CenterStrategy class from the parent Strategy class, with appropriate changes in the justify() implementation. This issue was designed to have the largest size in terms of LoC.

- **Issue 2 - Allow user input of layout width** At this state, the program has a fixed width for the layout of the text. Issue 2 enables the user to customize the layout width. This is a perfective activity that is designed to generate high FC and CC metrics by including modifications to many functions. Fig. 4 shows the desired output. According to our proposed metrics, this issue should take high maintenance effort as has the highest FC and CC metrics.
Figure 4. Screenshot of implementations of the ideal solution for Issue 2. a) shows a width of 45 while b) has a width of 20.

- **Issue 3 - Allow subsequent alignment input** At this state, the program only allows a single alignment before exiting automatically. Issue 3 requires participants to allow the user to run several alignments in one session before choosing to exit. The issue is a bug because even though returning to the menu after alignment, it immediately exits if any key is pressed. Fig. 5 shows an example of the desired output, where the alignment is run three times. The ideal solution would be adding a for-loop to the main function, so the program continues to return to the main menu until the exit option is selected. This issue was designed to be a simple fix that only affects the `main()` function.

- **Issue 4 - Buffer overrun error** Since issue 2 allows user input of the layout, this creates another bug since the program cannot handle layout widths greater than 60. Issue 4 requires modifications to allow widths up to 80. In this case, the solution requires the user to manipulate the array size of the variables storing the text. This has to be done in classes RightStrategy, LeftStrategy and CenterStrategy. It is a simple fix, however, it requires the developer’s comprehension of the program.

**Analysis and Evaluation**

Using the proposed metrics, we then performed a quantitative analysis of the issues previously mentioned. Using our proposed metrics we evaluated each of the maintenance activities.

Taking a closer look at LoC from Fig. 6, we can see that issue 1 has the most modified lines, while issue 3 has the lowest. In contrast, according to the program slicing metrics shown in Fig. 7, issue 2 has the highest CC and FC values for both slicing and non-program slicing metrics.
The metrics are contradicting, because while the CC and FC metrics suggest issue 2 as candidates for high maintenance effort, LoC metrics, on the other hand suggests issue 1.

IV. RESULTS

Participants

A total of eight test subjects participated in the experiment. All participants had a strong programming background and were currently graduate students from the Information Science department. In fact, three of the participants had previously been programming instructors, while the rest had worked on small to medium software projects. Four of the test subjects preferred C++ while the other four listed java as the preferred programming language.

Experiment Environment

During the experiment, each participant was given a 10 minute pre-experiment tutorial, explaining the technical aspects of the program. After this, an additional 5 minutes was then allocated for questions and/or free time to get familiar with the program before the maintenance activities. Each task was introduced in sequence for all participants along with screenshots of the desired output as seen in Fig. 3, 4 and 5. Each task is marked as resolved only after testing for the desired output. Only then, the next task was introduced. After all the tasks were completed, a simple questionnaire was filled out by all participants to collect comments, preferred programming language and difficulty of each issue. For all participants, the same isolated experiment environment was used.

Evaluation

For each participant, the average time for completion of the whole experiment was within 2 hours, with each issue ranging from 10-60 minutes for completion. Fig. 8 shows the distribution of the time taken to solve each issue. Fig. 9 (a) shows the individual ranking of each issue with a score from 1-5, while Fig. 9 (b) shows when they are ranked in the order of difficulty. These graphs clearly suggest that issue 2 was the hardest issue to implement. In reference to issue 2, one subject commented that the "C++ syntax used to pass arguments across functions was the problem". Other responses noted that unfamiliarity with the programming language lead to difficulties with issue 2.

In regard to the rest of the issues, participants identified them as typical maintenance activities. We assume that the difficulty of issue 1 was greatly aided by the use of the strategy design pattern as pointed out by a participant. Also, it was interesting to note over 75% of the participants had limited knowledge on design patterns, but found it very useful. Issues 3 and 4 were noted to be typical code changes. Issue 3 was seen as the easiest as it only involved minor modifications within a single function. Issue 4 took slightly more time as modifications had to be made in several locations, but not significant.

V. DISCUSSION

Our research investigates a quantitative approach to measure maintenance effort using a set of proposed metrics. In our previous work, we studied real world projects, with results suggesting false positives of maintenance effort, which take more time to be resolved due to the following factors: (1) low priority (back-burner), (2) human resource assignments or even a (3) delay feedback in the bug fixing work-flow

Analysis of the issues in Fig. 6, 7, 8 and 9 suggest that issue 1 is most likely candidate for high maintenance. However, our proposed metrics instead suggested issue 2 and was proven by our experiment results.

A controlled environment was used in which other factors such as the bug fixing process, developer workload, and priority were nullified. Table II proves our proposed metrics outperform the conventional metrics. This is significant as in real life the correlation may not be as strong. The correlation could serve as guidelines to quantitatively assess
maintenance effort and processes, outperforming traditional measures such as LoC. The usage of these metrics can be applied from two viewpoints.

From a code-based perspective, identification of potentially high maintenance effort code portions could be candidates for source code maintenance activities such as refactoring, code reviews and code inspections. These activities would improve source code quality and maintainability. Correlation between complexity and maintenance effort, these activities have potential to reduce the maintenance efforts. In addition, the affected code could be marked as complex, special attention is taken during code modifications.

From a project team standpoint, proper assignment of the most proficient resources to be made to these complex code changes. Teams can manage and assess their current workflow or maintenance processes.

**Revisiting Research Questions**

We address our research questions below:

**-RQ1.** Is there a quantitative relationship between maintenance effort and source code properties? To evaluate our results, we used the spearman’s rank correlation as the data is non-parametric. Since familiarity with C++ was perceived to be an influential factor, we classified subjects for participants that were comfortable with C++ (C++ developer) and those not (other developers). Table II suggests moderate correlations for each set of metrics.

**-RQ2.** Can we define a set of code-based metrics from this relationship? As mentioned in RQ1, our experimental results suggest our proposed metrics have a moderate to strong degree of correlation with maintenance effort.

**-RQ3.** How do the metrics compare in degree of relationship? Seen from the Table. II we see a positive association between the non-slicing and slicing based metrics.

**Threats to Validity**

The main threat is whether our experiment is a true representation of the real world. A program comprising of 78 lines of code is not representative of real world programs, however, we believe that we fundamentally represented basic maintenance tasks that could be sufficient for a typical programmer could attempt without prior knowledge of the program. Also in this study, the program slices were manageable, which may not be the case in the real world. Program slicing is complex, especially with large systems and this issue is being considered for future work. Another threat was the assessment of participant’s experience and skill. Keeping this in mind, the issues were designed so that they could be solved within a reasonable time-frame. We designed the program so that it would be easily understood by typical programmers.

Internal threats where the accuracy of the data collected. The consistent trend of the performance of the metrics across the three groupings of correlations gives us confidence in our results. Another internal threat would be our program slicing tool. However, as mentioned earlier in the paper, CodeSurfer is one of the most widely used tool for program slicing.

**VI. Conclusion and Future Work**

The purpose of this research is to investigate an alternative quantitative approach of the analysis of maintenance effort. In this study, we quantitatively expressed this established phenomena, using several common code-based metrics as well as our proposed program slicing-based metrics. Although the results are promising, there are still outstanding issues for future work, including the following:

- More application to the real world. For future work, we would like to apply our approach to industrial projects.
- Explore other factors of maintenance effort. Evaluate the impact of factors such as the bug fixing process, developer workload, and issue priority. Also, explore differences between the non-slicing and slicing based metrics.
- Explore the change impact to handle large-scale systems. Further investigate strategies to manage large slicing. Heap slicing is a possibility.
- Assessment model for the assessment of the maintainability of software. Other aspects such as process effort, human and infrastructure management for the achievement of process objectives will be investigated.

Although, the study is in its early stages of validating the generalization of our approach, the final goal would be to create a process assessment framework and prediction model.

**Acknowledgment**

Thanks to Ana Erika Camargo Cruz for her contributions and the participants for their time and effort. This research is supported by JSPS, Grant-in-Aid for Scientific Research (No.22500027).
REFERENCES


