Analysis of Bug Fixing Processes Using Program Slicing Metrics

Raula Gaikovina Kula, Kyohei Fushida, Shinji Kawaguchi, and Hajimu Iida

Graduate School of Information Science, Nara Institute of Science and Technology
Takayamacho 8916-5, Ikoma, Nara 630-0101, JAPAN
{raula-k,kyohei-f,kawaguti}@is.naist.jp, iida@itc.naist.jp

Abstract. This paper is a report of a feasibility study into an alternative assessment of software processes at the micro-level. Using the novel approach of applying program slicing metrics to identify bug characteristics, the research studied relationships between bug characteristics and their bug fixing processes. The results suggested that specific characteristics such as cyclomatic complexity may relate to how long it takes to fix a bug. Results serve as a proof of concept and a starting point for this proposed assessment methodology. Future refinement of the metrics and much larger sample data is needed. This research is an initial step in the development of assessment tools to assist with Software Process Improvement.

Keywords: Program Slicing, Software Bug, Bug Fixing Process, Software Process Improvement

1 Introduction

The quality and improvement of software processes are seen as vital to any software development project. Traditionally, software processes usually refer to the main phases in the software development life cycle. Improvement of these processes is a major activity for larger software organizations as benefits are seen in the cost and business value of improvement efforts, as well as the yearly improvement in productivity of development, early defect detection and maintenance, and faster time to market [1].

Capability Maturity Model Integration (CMMI) [2] is the most common form of rating software development organization's quality of performance. Other models such as International Standards (ISO 9000), [3] have been used for the quality of an organization's software development processes. There have been a number of studies on the issues relating to the implementation of these CMMI and ISO 9000 [4], [5], [6]. Much of the issues lie with the size of the organization as these models are expensive to assess and implement from small software organization [7]. These studies show that the processes assessments are sometimes tedious and usually not tailored for specific companies. Also additional studies show that higher management support, training, awareness, allocation of resources, staff involvement, experienced staff and defined software improvement process implementation methodology is critical for software process improvement [8].

There has been several related work into trying to make the models easier and better to use. Yoo el al [9] suggests a model that combines CMMI and ISO methods. Armburst el al [10] takes a different approach by treating software as manufacturing product lines, therefore making the processes systematic and generic.

In contrast to such frameworks macro-level process assessments and improvement, an alternative measure of software processes can be done through the inspection of process execution histories. This analysis however is performed at the developer's level, and measures the fine grained processes called 'micro processes'. One such research has been done by Morisaki el al [11]. This research analyzes the quality of micro processes by studying the process execution histories such as change logs. Quality of the process is measured by the execution sequence, occurrences of missing or invalid sequences. This paper is based on this research and seeks improvement of micro-level processes will contribute to Software Process Improvement.

Our motivation is to develop a quantitative assessment model of software process at the micro level. It is aimed towards improving the efficiency of the software developer's workload at the maintenance phase of bug fixing. Current models of process assessment are focused on a higher software life cycle level and need complicated assessment such as CMMI and ISO 9000, which, as mentioned earlier are expensive as they require highly trained assessors. The author's goal is to work towards the development of models that assess the quality of micro processes.

More specifically, a proof of concept towards a prediction model based on estimating the bug processes execution time is the perceived contribution of this research. Being able to estimate the bug fixing process will improve the development process as more developers can manage resources and time based on how long it will take to fix a bug.

It is envisioned this study may aid developers in the micro process of bug fixing. The concept is that aiding the developer with tools to better manage bugs, leads to better quality in software processes at this level, influencing the overall improvement of the software development process.

The paper's objective is to provide a proof of concept that how a bug fixing process is executed may be related to a certain characteristic of a bug. For instance, perhaps bugs that have a lengthy fixing process have certain similar characteristics as compared to bugs that are easily fixed in just a couple of days. Bug characteristic is defined as the properties of the fragments of code prone to the bug fix. Based on this concept, the following hypotheses are presented for testing:

- Hypothesis 1: Bugs with similar bug characteristics share similar bug fixing process executions
- Hypothesis 2: Bugs that do not follow the usual micro process occur due to some bug characteristic.

In this paper, Section 2 presents the methodology, describing the terms and definitions used in the research as well as the proposed approach. Section 3 then explains the experiment using the approach and tools to carry out the experiments. The section also presents the findings of the experiments. Section 4 is the discussion and analysis of the results. Also the validation of the research is discussed as well as

the application of the research is included. Finally, Section 5 outlines the conclusion and the future works.

2 Methodology

As mentioned in the research objectives, the research aims to find a relationship between a bug characteristic and the processes used to fix the bug. The approach taken was to first reconstruct a bug fixing process. Then using program slicing techniques identify bug prone fragments as well as define certain characteristics of a bug. Finally comparisons are performed to group and find relationship between bugs with similar bug fixing processes and similar bug characteristics.

2.1 Bug Fixing Process Definitions and Analysis

This research focuses on the day to day processes performed in the development of software. This paper assumes data repositories including following two locations in which bug information is stored daily.

1)Bug Tracking System

Typically a software development team uses a system to manage bugs in a medium to large scale project. The tracking system usually tracks progress of bug. This research utilizes the bug tracking systems to gather various properties of the bug.

2)Source Code Repository

The source code repository refers the system manages changes to source code in a software project. The two main system used is the Subversion (SVN) and Concurrent Versioning System (CVS). For this research, both SVN and CVS repositories were used to gather data on bug characteristics.

The generalized model of micro processes of bug tracking system used in the MPA model by Morisaki was used. The research showed that bug fixing comprises of these steps illustrated in Figure 1.

Step-1. *Bug Reported:* This is the start of the micro process. It signifies that there is a request to check and fix source code. Usually this event is signified when a new bug is created in the system.

Step-2. *Bug Assigned:* This is where the bug gets assigned to a developer. It is signified when the bug changes status indicating that the bug is being worked on. This event is signified by a change in status of the bug to 'assigned'.

Step-3. *Bug Fixed*: This stage is when code is being fixed. When code is being updated means that the bug fix is being applied to the source code, therefore captures the real time of when the bug is being fixed. This is signified by editing code in the source code repository and a reference to the bug is made in the change log or comments section of the changes.

Step-4. *Bug Closed*: This is the final stage of the bug fixing process. This is when the bug is acknowledged as fixed. This is signified when the bug status is changed to 'closed'.

These steps are generalized and is differs based on organization and project needs.

The quality of the micro process analysis is evaluated by how well the different phases are executed. Bugs were evaluated on whether they followed this model. Grouping of the bugs would be according to how well the model is followed. The groupings are identified as either correct sequences or incorrect sequences.

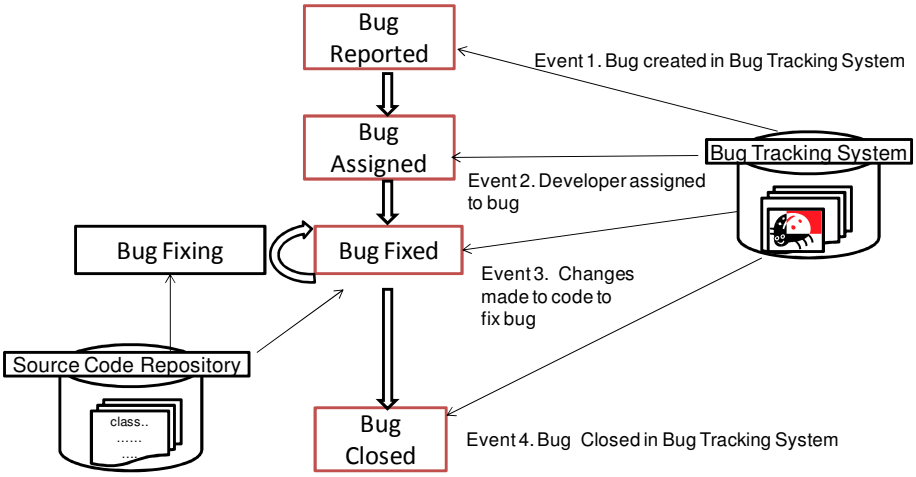

Fig.1. Bug Fixing Process Model

2.2 Program Slicing Metrics for Bug Characteristics

To assess the bug characteristics, program slicing metrics is used in this approach to describe bug characteristic metrics. Program slicing is a concept first described by Weiser [12] as a method to study the behavior of source code through the flow dependency and control dependency relationship among statements. This work however has been used in the field of Software Evolution [13].

Previous work has been done on using program slicing metrics to classify bugs. Pan el. al. [14] showed proved that program slicing metrics work as well as conventional bug classification methodologies.

Since this is the initial research on applying program slicing metrics to micro process analysis, two of the most basic code metrics, Lines of Code (LoC) and Cyclomatic Complexity (CC) [15] were employed to express characteristics of the program slices.

2.3 Experiment Approach

The analysis of the bug fixing process includes three steps: 1) micro process extraction and analysis and 2) program slice extraction and analysis, and 3) comparison and analysis of data.

1) Micro process extraction and analysis:

This step involves data mining and extraction of bug related attributes from both the source code repository and the bug tracking system. Table 1 and 2 shows the data that is extracted from the bug tracking system and the data extracted from source code repository, respectively.

Table 1. Data extracted from the bug tracking system

Attribute	Description
Bug ID	Used to identify the bug
Date Opened	When the bug was reported
Date Closed	When the bug was reported as fixed
Bug Priority	Understand bug's importance

Table 2. Data extracted from source code repository

Attribute	Description
Revision ID	Identification number used to track changes made to source code
Bug ID	References the changes made to a bug with bugID
Commit Date	Date when the bug fix was applied

2) The program slices extraction and analysis:

This step involves analysis of the code using the program slicing metrics. The methodology used was to first identify the file edited during the bug fix, then extract the associated files directly affected by bug fix using program slicing. Then the code metrics for the affected files were calculated. The following explains in detail how each metric was used to for bug classification.

- Bug LoC (Lines of Code): This metric is proposed to measure how much of the code is potentially affected by the bug. This is measured using the program slicing metric Lines of Code (LoC). The range of a bug is summarized in the equation below:

$$BugCC = \sum_{f \in affectedfiles} LoC(f) \quad (1)$$

where *affectedfiles* refers to the source code files that were affected when the bug was being fixed, and *LoC(f)* is lines of code of file *f*. The affected files include files that are dependent code related to the files edited during the bug fix.

- Bug CC (Cyclomatic Complexity): This metric is proposed to calculate the potential complexity of the code affected by the bug. This will be measured using the program slicing metric CC as explained in section 2.2. The severity of a bug is:

$$BugCC = \sum_{f \in affectedfiles} CC(f) \quad (2)$$

where *affectedfiles* refers to the source code files that were affected when the bug was being fixed, and *CC(f)* is cyclomatic complexity of file *f*. Since CC is calculated per function, Bug CC is sum of all the cc for each function in the affected files. The affected files include files that are dependent code related to the files edited during the bug fix

3) Comparison and Analysis of data

Using the MPA data extracted and the program slicing metrics extracted from the software project, the bugs are grouped according to similar program slicing metrics characteristics.

3 Experiment

We conducted an initial experiment using Open Source Software (OSS) repositories. The main reason for choosing OSS was that it is easily accessible as well as does not require special permission or software licenses to analyze datasets.

3.1 Experiment Tools

We used following tools selected for both the program slicing and the tool used to extract the MPA data:

Program Slicing Tool

Based on previous surveys [16] on the available program slicing tools, three program slicing tools [17], [18], [19] were tested and the most appropriate tool selected. In the end CodeSurfer [19] was chosen because it met the needs of the research as well as being used in Pan's use of program slicing metrics for bug classification. The main limitation of CodeSurfer is that only projects in the C/C++ programming languages can be analyzed.

Bug Extraction Tool (Extract the MPA Data)

Using the micro process model we designed a bug extraction tool that would help search and extract the needed data related to this research. Published tools such as Kenyon [20] are available, however because data to be extracted are specific to this study, it was justified to develop the tool in-house.

Our extraction tool which was developed in java, acted as a web spider searching the online data repositories, and then parsing extracted data. In order to make the tool more flexible and not become a constraint on the research, the tool was developed to search both SVN and CVS repositories. The extracted data is mentioned in section 2.3. All bugs were verified to be bugs and not change requests.

3.2 Test Subjects

Due to the limitations of the CodeSurfer slicing tool only being able to analyze projects based on the C programming language, projects were selected based on program slicing capability (C++ projects were selected.) and on quality of bug extraction (if there was a reference between the bug tracking system and the source code repository). Using Sourceforge.net as a source of the OSS projects, the following three software projects met the selection criteria:

- *Scintilla*(Up to Ver. 2.01)*:* Scintilla is an editing component for Win32 and GTK+. It comes bundled with SciTE, which is a Scintilla based text editor. It is used when editing and debugging source code.
- *WxWidgets*(Up to Rev. 62931)*:* WxWidgets is a C++ library that lets developers create applications for Windows, OS X, Linux and UNIX on 32-bit and 64-bit architectures, as well as several mobile phones platforms including Windows Mobile, iPhone SDK and embedded GTK+. Like Filezilla, WxWidgets used subversion and houses its bug tracking system outside Sourceforge.net.
- *Filezilla* (Up to Rev. 3295)*:* Filezilla is the open source File Transfer Protocol solution. The client component of the project was used for analysis. Filezilla used the subversion repository system and hosts its own bug tracking system.

The bugs used in the experiment were all that had complete data. This means all bugs in the projects that had its bug fixing process reconstructed and bug characteristics calculated were used. All bugs from project start date till the experiment date, December 2009 were analyzed. Figure 2 summarizes the collected data.

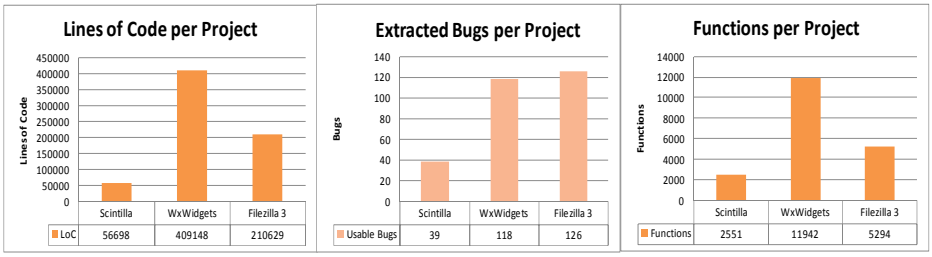

Fig.2. General Comparisons between Projects

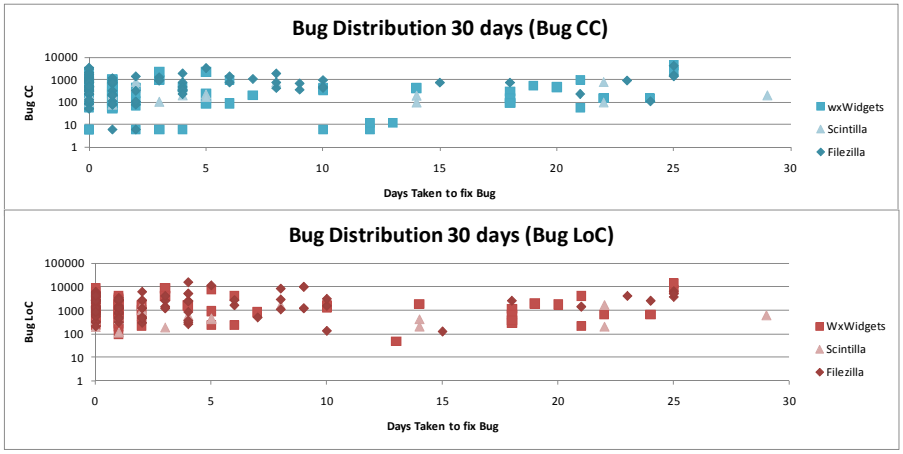

Fig.3. Bug Distribution for the first 30 days

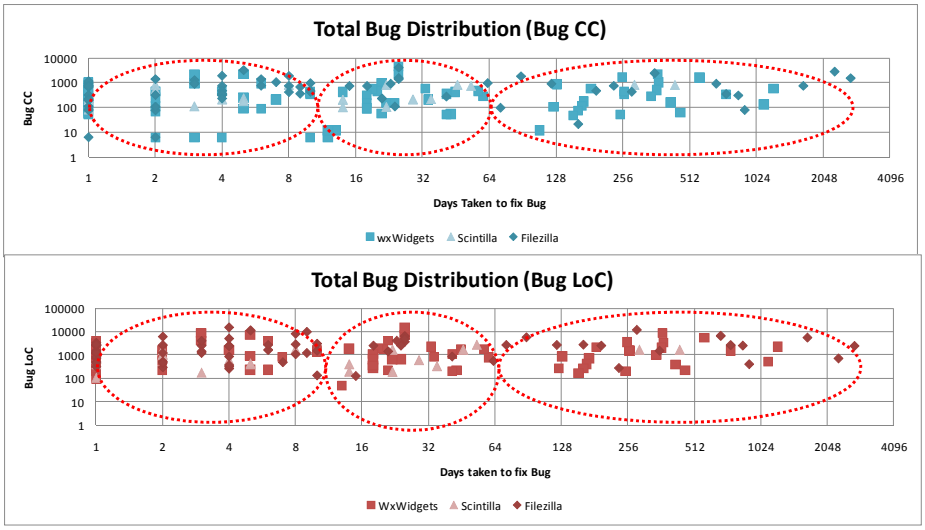

Fig.4. Total Bug Distribution on log-log graph

3.3 Findings

The general approach taken in analysis of the information was to do various comparisons of bug characteristics against their bug fixing process. Comparisons and groupings were: 1) Duration of bug fix and 2) Assessment of bug fixing process.

With these two groupings, bug characteristics were applied to the groups to compare and attempt to find some similarities.

1) For the duration of the bug fix, the proposed analysis method used the number of days it took to fix a bug. This was calculated as:

$$\textit{Days taken to fix a bug} = \textit{Date of Bug Fix} - \textit{Date of Bug Detected} \quad (3)$$

where bug fix is the date when the code changes are committed to the code and bug detected being the date when the bug was first reported. Bug Fix was used instead of 'Bug Close' status as it is more accurate representation of when the bug was addressed.

As seen from Figure 3 the distribution of bugs seem to differ from 10 days onwards, thus identified as a cluster. Since the distribution covers a very large range of data, logarithmic graphs were used to plot the rest of the data. Using visual identification shown in Figure 4, it was seen that around bugs that are older than 64 days have a wider distribution compared to the bugs fixed in less than 64 days. With this observation, the second group of bugs was proposed from 11 to 64 days and the remaining third group from 64 days onwards.

2) Assessment of bug fixing process

The following groupings were applied based on the MPA model:

- *Standard Sequence:* Bugs that have all the processes sequenced in correct order.
- *Incorrect Sequence*: This group of bugs has incorrect sequences of the bug process compared the proposed model. The term 'incorrect' refers to abnormal sequence and behavior with the bug fixing process.

Significantly, 71%-83% of each project bugs were being *Fixed and Closed on the same day*. Therefore it was added as a separate group. It was noticed that Filezilla and WxWidgets were missing the 'assign' step for the MPA model.

Figure 5 illustrates the comparison of bug fixing process groups against the duration of bug fix groups. For Scintilla, the incorrect sequences occur in Bugs fixed in 0-10 days only. Also bugs that were fixed and closed the same day occur in all three groups. Finally bugs that took more than 64 days were all closed on the same day as being fixed. In the case of WxWidgets, there is an even distribution of the different types of bug fixing process executions, however, bugs fixed and closed on the same day always is greater than the other groups. Finally with Filezilla, Figure 5 indicates that the incorrect sequence, although very small, occurs in bugs that took up to 10 days to fix and more than 64 days to fix. As with WxWidgets, the bugs that were fixed and closed in the same day occur in all bug groups, however, most are found in bugs that took up to 10 days to fix. As seen below, results do not to show a clear trend.

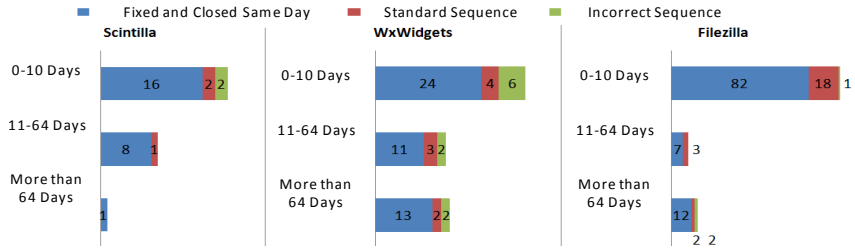

Fig.5. Bug fixing process compared with bug classifications

Program Slicing Metrics

Using the assessment of bug fixing processes groupings, comparisons are made against the bug characteristics.

- Bug CC Analysis: The graphs in Figure 6 illustrate the distribution of cyclomatic complexity of the bugs. The results indicate that the 11 to 64 days group has the largest distribution compared to the other groups. Also 11 to 64 days has higher bug CC compared as well. This suggests that bugs that are fixed in 11 to 64 days have a larger range of bug CC and are higher in complexity as compared to the other groupings. It is also interesting to note that bugs that took than 64 days to fix have low complexity and a lower distribution.

- Bug LoC Analysis: The box plots illustrated in Figure 7 show the Bug LoC for bugs within the bug classifications. The graphs clearly show that bugs that took 11-64 days to fix have the highest lines of code in all projects. Apart from the highest lines of code we cannot make other similar characteristics. It seems that there is no clear trend from the three projects. Scintilla and WxWidgets seem to indicate that the 11 to 64 days has a largest distribution of Bug CC, however Filezilla suggest that maybe all groups have a similar distribution. These results indicate that further analysis with a larger test group need to be done.

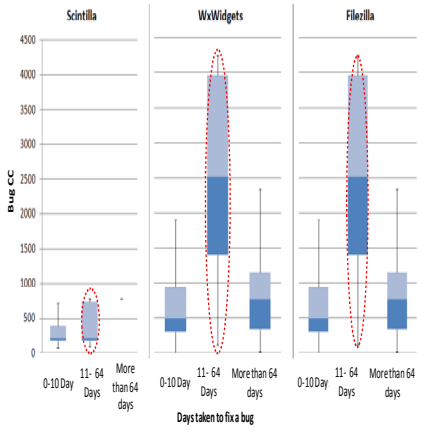

Fig.6. Bug CC Distribution using Bug Classification across Projects

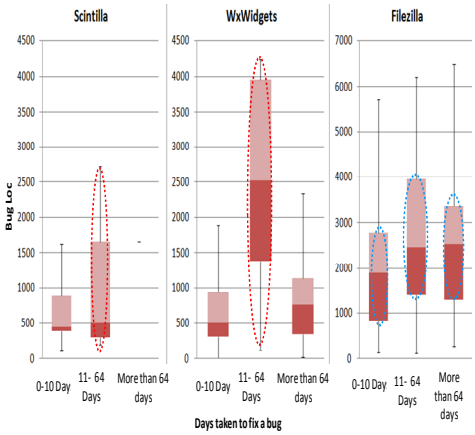

Fig.7. Lines of Code Distribution using Bug Classification across Projects

Total Data Analysis

Finally, we performed an analysis combining both the micro process execution and program slicing metrics. Results are as follows:

- Bug CC: The graphs in Figure 8 are the distribution of Bug CC metric of bugs grouped according to the bug fixing process per project. The graphs suggest that in each project, bugs with incorrect sequences have the highest distribution and maximum bug CC. The interpretation could be misleading as seen in the previous section, incorrect sequences account for only 2% to 15% of the sample

size. However shows that further analysis is needed and the direction is promising.

- Bug LoC: Figure 9 show the results of when the groupings of bugs according to execution process were measured using the Bug LoC metric. Similar to the results seen in Bug CC, Bug LoC also displays incorrect sequences as having the widest range and highest lines of code as compared to the other groups. Similar to Bug CC analysis with program slicing, this information could be misleading as the sample sizes for the incorrect sequences are extremely low.

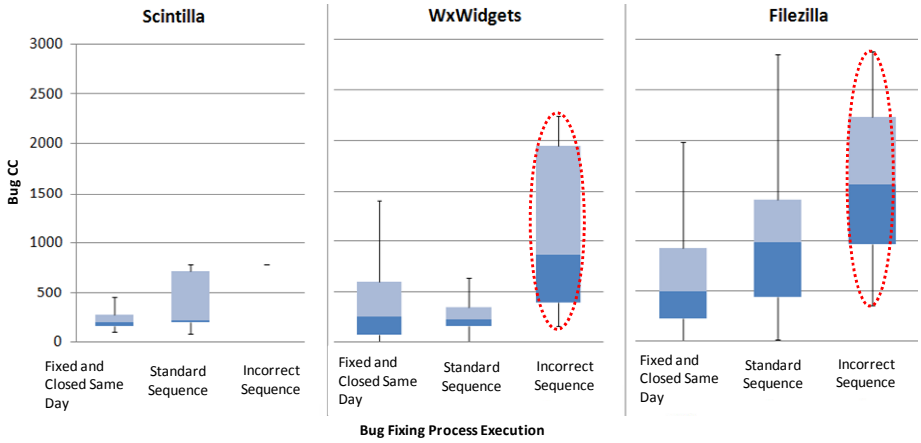

Fig.8. Bug CC and Process Classifications

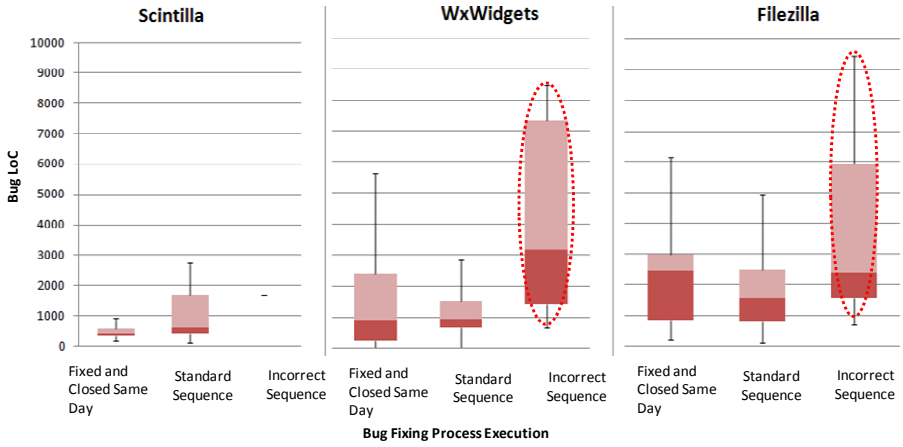

Fig.9. Bug LoC and Process Classification

4 Discussion

4.1 Threats to Validity

This paper shows that the bug classification has some many challenges and validation issues with getting the correct datasets for experiments. Firstly the extraction methods are messy due to the many variations in the way the data repositories are managed, for example CVS and SVN. Additionally many of the software projects have subtle differences that make data extraction difficult. Other research have also encountered these perils, making OSS analysis data validation questionable [21]. Also the analysis only covered bugs that the bug fixing process could be re-constructed as well as have generated bug characteristics. This constraint also contributed to the small data we had to use for the analysis.

Another threat would be the accuracy of the metrics. Currently the slices are at file level, which could be arguably too broad a scope. However we do plan to do calculations of Bug CC and Bug LoC at function level to get more accurate results. It is stressed that what was more important was the results and feasibility of this new approach.

Though all the difficulties, the research did find feasible projects and was able to successful experiment and draw results. Since this is a proof on concept, emphasis was placed more on the results and methodology rather than the validation of the approach. The total final analysis in Figure 8 and 9 show very interesting results, however due to the small amount of data statistical analysis could not be performed. It is envisioned that with more accurate metrics, statistical analysis will be applied.

4.2 Findings in Analysis

The approach taken proves that bugs could be grouped based on program slicing based metrics. Using our proposed approach results firstly showed that using the program slicing metrics, the bugs were grouped according to how long it takes to fix the bug. With visual analysis of the distribution, it was found that the bug distribution changed after 10 days and 64 days. One theory based on project management, 10 days is 2 man weeks therefore developers may be pushed by deadlines to fix the bugs before 10 days.

To better understand the bug classification, the distributions of program slicing metrics were analyzed according to the bug groupings. Findings in Figure 6 and 7 suggest that bugs fixed within 10 days show lower complexity. Bugs between 11 and 64 days show a wider range of complexity and bugs that took more than 64 days had a much lower complexity. Since most bugs lie in the 0-10 day group, it can be concluded that bugs with lower complexity are fixed in less days. However another conclusion could be related to project management, 10 days is 2 weeks so maybe a project manager may require a set of bugs to be completed by that deadline. In relation of lines of code per bug, there seem some trends but nothing concrete enough to propose, therefore further analysis and review of the lines of code metric need to be performed.

The second part of the research aimed at analyzing the micro process execution, and grouped using the bug classifications. Results firstly show not all projects follow the proposed model for bug fixing. Both Filezilla and WxWidgets omitted the 'assign to developer' activity in the bug fixing process. Possible reasons would be the limitations of the bug tracking tool or the assignment process is not part of that projects bug fixing process related to two of the three projects. Also this shows the real time-frame in which the bug was fixed.

Another interesting finding was the appearance of bugs that were closed on the same day that it was fixed. In all test subjects, results indicated that the majority of bugs had bug closed the same day they were fixed. It was found that bugs closed in 0-10 days were most prone to errors in the bug fixing process. All projects indicated more bugs with incorrect sequences occur in bugs fixed in ten days. Bugs older than 10 days have a lesser chance of having incorrect sequences.

Finally, the main part of the analysis was done combining both the micro process execution and program slicing metrics. Bugs were grouped according to their bug fixing process execution against both Bug CC and Bug LoC. These results suggest that bugs with incorrect sequence in their bug fixing processes show a wide distribution with very high cyclomatic complexity and lines of code as compared to bugs that follow the general bug fixing model and bugs that are closed in the same day as being fixed. This data however may not be reliable as the sample set for incorrect sequence is too small to have any statistical significance.

4.3 Testing Hypothesis

Putting together the bug classifications and the analysis of the micro process as a proof of concept, there is enough evidence to suggest that there is indeed a relationship between bug characteristics and its processes executed. For example for bugs taking up to 10 days to fixing generally have low complexity and lines of code show a tendency to be closed in the same day and have a higher likelihood to have errors in its process execution.

In response to the hypotheses, the findings support Hypothesis 1 as our research shows groupings of similar bugs having similar bug characteristics. For example, bugs with incorrect micro process execution show much higher Bug CC metric than the correct standard and fixed and closed in same day bugs. Hypothesis 2 is also supported by the evidence that higher Bug CC is more prone to incorrect sequences. The experiments therefore suggest a relationship between bug characteristics and how they were fixed. However, further research it needed to have more confidence in these hypotheses.

5 Conclusion and Future Works

As mentioned throughout the paper, this work is seen as proof of concept with the final aim of developing a prediction model for bug fixing process based on the bug's characteristics. Future work will be to refine the tools and metrics used. For this

research, only two program slicing metrics were introduced. As indicated by the results, extensive analysis is needed to make more concrete judgments.

Current results of the research act as a proof of concept for the use of program slicing in the inspection of the bug fixing processes in a software development project. It is envisioned that the research will help contribute to a better understanding and classification of bugs based on the nature of code. Currently the program slicing is performed according to file level code metrics. Future work will attempt to program slice at function level, giving precise metrics. A larger sample data that can give statistical analysis will be experimented to further validate the results.

If successful, the research can used to help create 'prediction models' for bugs. For example, based on a history of previous bugs and the trends of a specific project, a bug's fixing process could be estimated from its bug characteristics , thus assist developers handle the bug faster. This would then contribute to software improvement at this micro level.

The implementation could be a tool that is used during the bug detection and assignment stage of the bug fixing process. It would be able to analyze what part of the code are affected, and based on the classification, predict how long the bug would normally take to fix as well code based metrics such as the complexity of code.

In the bigger picture, this work contributes towards a predictive process model based on the program slicing metrics. As this is a novel approach, this study contributes as a starting point towards this goal.

Acknowledgements

This work is being conducted as a part of the StagE project, The Development of Next-Generation IT Infrastructure, supported by the Ministry of Education, Culture, Sports, Science and Technology.

References

1. Herbsleb, J., Carleton, A., Rozum, J., Siegel J., Zubrow, D.: "Benefits of CMM-Based Software Process Improvement: Initial Results", CMU/SEI-94-TR-13. Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University (1994)
2. Margaret, K. K., Kent, J. A., "Interpreting the CMMI: A Process Improvement Approach", Auerbach Publications, CSUE Body of Knowledge area: Software Quality Management, (2003)
3. Schmauch, C. H.:"ISO 9000 for Software Developers", 2nd. ASQ Quality Press, (1995)
4. Beecham, S., Hall, T., Rainer, A.: "Software process problems in twelve software companies: an empirical analysis", Empirical Software Engineering, v.8, (2003), 7-42
5. Baddoo, N., Hall, T.:"De-Motivators of software process improvement: an analysis of practitioner's views", Journal of Systems and Software, v.66, n.1, (2003) 23-33
6. Niazi, M., Babar, M. A.: "De-motivators for software process improvement: an analysis of Vietnamese practitioners' views", in: International Conference on Product Focused Software Process Improvement PROFES 2007, LNCS, v.4589, (2007) 118-131

7. Brodman, J.G., Johnson, D.L.: "What small businesses and small organizations say about the CMMI", In Proceedings of the 16th International Conference on Software Engineering, IEEE Computer Society, (1994)
8. Rainer, A., Hall, T.:"Key success factors for implementing software process improvement: a maturity-based analysis", Journal of Systems and Software v.62, n.2, (2002) 71-84
9. Yoo, C., Yoon, J., Lee, B., Lee, C., Lee, J., Hyun, S., Wu, C.: "A unified model for the implementation of both ISO 9001:2000 and CMMI by ISO-certified organizations", The Journal of Systems and Software 79, n.7, (2006) 954-961.
10. Armbrust, O., Katahira, M., Miyamoto, Y., Münch, J., Nakao, H., Campo, A. O. "Scoping software process lines", Software Process Improvement and Practice, v.14, n.3, May, (2009) 181-197
11. Morisaki, S., Iida, H.: "Fine-Grained Software Process Analysis to Ongoing Distributed Software Development", 1st Workshop on Measurement-based Cockpits for Distributed Software and Systems Engineering Projects (SOFTPIT 2007), Munich, Germany, Aug., (2007) 26-30
12. Weiser, M.: "Program slicing", In Proceedings of the 5th international Conference on Software Engineering (San Diego, California, United States). International Conference on Software Engineering. IEEE Press, Piscataway, NJ, Mar. 09 - 12, (1981) 439-449
13. Hall, T., Wernick, P.: "Program Slicing Metrics and Evolvability: an Initial Study ", Software Evolvability, IEEE International Workshop, (2005) 35-40
14. Pan, K., Kim, S., Whitehead, E. J. Jr.:"Bug Classification Using Program Slicing Metrics", Source Code Analysis and Manipulation, IEEE International Workshop, (2006) 31-42
15. Watson, A., McCabe, T.: "Structured testing: A testing methodology using the cyclomatic complexity metric", National Institute of Standards and Technology, Gaithersburg, MD, (NIST) Special Publication, (1996) 235-500
16. Xu, B., Qian, J., Zhang, X., Wu, Z.,Chen, L.: "A brief survey of program slicing", SIGSOFT Softw. Eng. Notes v.30, n.2, Mar., (2005) 1-36
17. Ranganath, V. P., Hatcliff, J.: "Slicing concurrent Java programs using Indus and Kaveri", International Journal on Software Tools for Technology Transfer (STTT), v.9 n.5, Oct., (2007) 489-504
18. Wang, T., Roychoudhury, A.: "Dynamic slicing on Java bytecode traces", ACM Trans. Program. Lang. Syst. v. 30, n.2, Mar., (2008) 1-49
19. Anderson, P., Zarins, M., "The CodeSurfer Software Understanding Platform", Proceedings of the 13th International Workshop on Program Comprehension, May 15-16, (2005)147-148
20. Bevan, J., Whitehead, E. J., Kim, S., Godfrey, M.: "Facilitating software evolution research with Kenyon", In Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT international Symposium on Foundations of Software Engineering (Lisbon, Portugal). ESEC/FSE-13. ACM, New York, Sept. 05-09, (2005) 177-186
21. Howison, J., Crowston, K.: "The perils and pitfalls of mining SourceForge", presented at Mining Software Repositories Workshop, International Conference on Software Engineering, Edinburgh, Scotland, May 25, (2004)