NAIST-IS-MT0851204

Master's Thesis

Using Program Slicing Metrics for the Analysis of Bug Fixing Processes

Raula Gaikovina Kula

February 4, 2010

Department of Information Systems
Graduate School of Information Science
Nara Institute of Science and Technology

A Master's Thesis
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
MASTER of ENGINEERING

Raula Gaikovina Kula

Using Program Slicing Metrics for the Analysis of Bug Fixing Processes

Raula Gaikovina Kula

Abstract

Software Process Improvement is increasingly becoming a major activity for most software development organizations due to the benefits seen in the cost and business value. Software process assessments, however, can be tedious, complicated and costly. This research explores alternative assessments by analysis of fine grain processes of bug fixing with their bug characteristics.

Using the novel approach of applying program slicing metrics to understand bugs, the research studied the relationships between bug characteristics and their bug fixing processes. The results suggested specific characteristics such as cyclomatic complexity may relate to how long it takes to fix a bug. Results in the study serve as a proof of concept to the feasibility of this proposed assessment. Future refinement of the metrics and much larger sample data is needed.

This research is seen as the initial step in the development of assessment tools to assists with Software Process Improvement. It opens possibilities for assessment tools for software processes.

Keywords:

Micro Process Analysis, Program Slicing, Software Bug, Software Bug Fixing

.*Master's Thesis, Department of Information Systems, Graduate School of Information

---

Science, Nara Institute of Science and Technology, NAIST-IS-MT0851204, February 4rd, 2010.

Table of Contents

List of Figures and Tables

1. Introduction

1.1 Overview – Software Process and Improvement

As software engineers, 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 as shown in the Figure below.

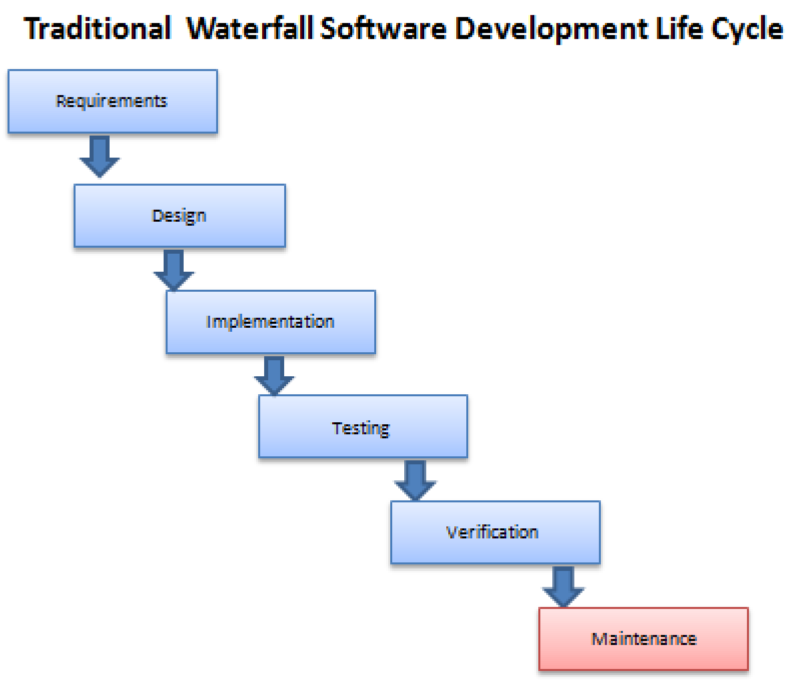


Figure 1 Software Development Life Cycle

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].

1.2 Related Work

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 [9] suggests a model that combines CMMI and ISO methods. Amburst [10] takes a different approach by treating software as manufacturing product lines, therefore making the processes systematic and generic.

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’. Such a process is the process to fix a bug within the maintenance level of the software development life cycle as shown in Figure 1.

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.

1.3 Motivation

The author's motivation is to present alternative methods of software process assessment, focused to help efficiency of the workload of developers. Current models of process assessment are at the software life cycle level and require complicated assessment such as CMMI and ISO 9000 as discussed in the related works. These assessments are also expensive as they require highly trained assessors. The author's ambition is to work towards models that improve and assess quality of micro processes.

More specifically, the research looks to further Morisaki's model of micro processes analysis in terms of the bug fixing processes. A proof of concept towards a prediction model based on estimating the bug processes execution time will be the contribution of this thesis towards improvement of software processes at a micro level. 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 by aiding the developer with tools to better manage bugs, it may lead to better quality in software processes at this level, as well as influencing the overall software development processes.

1.4 Research Objective

As an overview, the main objective of the research is to provide a proof of concept that there is indeed a relationship between a bugs attributes and how its related processes are executed. For example, a bug that has a lengthy process has certain characteristics as compared to a bug that is easily fixed in just a couple of days. Based on this objective, the following hypotheses are presented for testing:

Hypothesis 1: There similar bug program slicing metrics that we are able to classify a bug that has similar micro process executions

Hypothesis 2: Are incorrect micro process execution occur due to a classification of a bug?

1.5 Thesis Layout

The thesis is organized in the following parts. Chapter 2 presents the methodology, which includes the terms and definitions used in the research as well as the proposed approach. Chapter 3 then explains the experiment using the approach and tools to carry out the experiments. Chapter 4 presents the results of the experiments. Chapter 5 is a discussion and analysis of the results. Also the validation of the research is discussed as well as the application of the research. Finally, chapter 6 outlines the future work followed by chapter 7 with the conclusion.

2. Methodology

2.1 Overview

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. Using a novel approach, the research attempts to first classify bugs according to the characteristics of the code. Program slicing techniques have been proposed to identify certain characteristics of bugs.

The chapter is organized as follows. Section 2.2 outlines the definitions for this research. Section 2.3 and 2.4 discuss the micro process model as well as the program slicing metrics employed for this thesis. Finally section 2.5 describes the approach of the research.

2.2 Bug Fixing Process Definitions

This section covers the definitions related to the bug fixing process. This section introduces where bug related information is stored and what bug fixing model was used for this research.

2.2.1 Data Repositories

This research focuses on the day to day processes performed in the development of software. The next sections describe the two locations in which bug information is stored daily.

2.2.1.1 Bug Tracking System

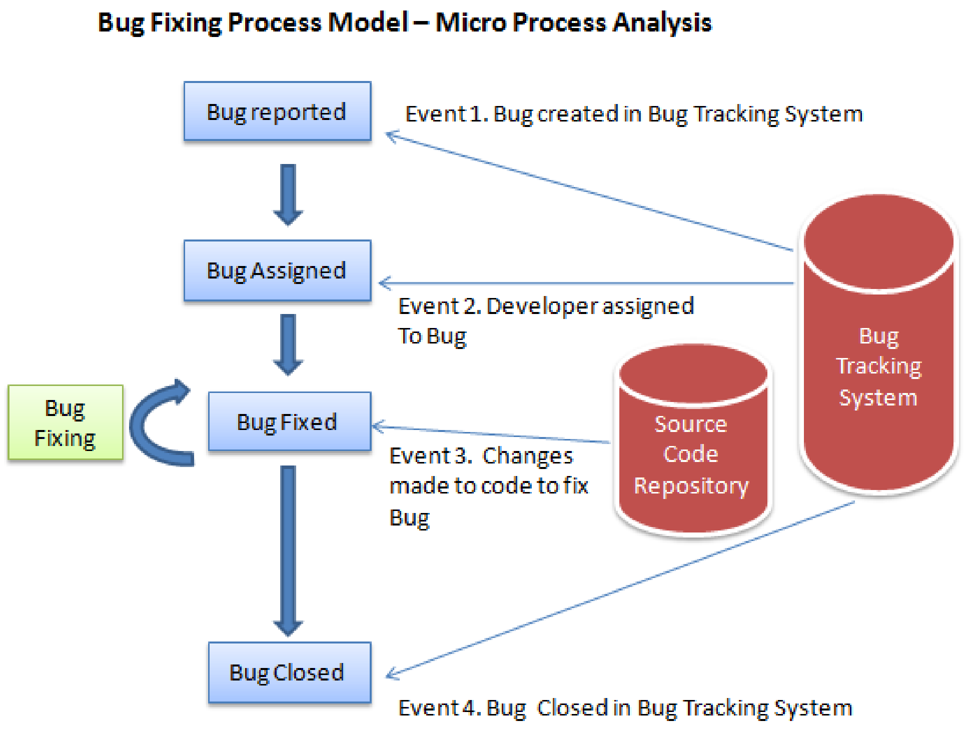
Typically a software development team uses a system to manage bugs in a medium to large scale project. The tracking system usually tracks progress of bug. This research utilizes the bug tracking systems to gather various properties of the bug.

2.2.1.2 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.

2.3 Micro Process Analysis (MPA)

For this research the generalized model of micro processes of bug tracking system proposed Morisaki. The research shows that bug fixing comprises of these steps illustrated below (Figure 2).



Bug Fixing Process Model – Micro Process Analysis
Bug reported
Event 1. Bug created in Bug Tracking System
Bug Assigned
Event 2. Developer assigned To Bug
Bug Fixing
Bug Fixed
Event 3. Changes made to code to fix Bug
Source Code Repository
Bug Tracking System
Bug Closed
Event 4. Bug Closed in Bug Tracking System

Figure 2 Bug Fixing Process Model

2.3.1.1 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.

2.3.1.2 Bug Assigned to Developer

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'.

2.3.1.3 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.

2.3.1.4 Closed Bug

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'.

The quality of the micro process analysis is evaluated by how well the different phases are executed. Incorrect sequences and omissions are used to indicate the low quality of micro processes of a bug fixing in a software project.

2.4 Program Slicing Metrics

To assess the bug characteristics, the novel approach taken was to attempt classification of bugs according to similar characteristics in code. Program slicing metrics were 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 basic program slicing metrics. The analysis is done at the function level with both metrics. The metrics chosen are described below. The concept was that lines of code measure how much of the code is affected by the bug and cyclomatic complexity measures complexity of the portion of the code affected by the code.

2.4.1 Lines of Code

The term refers to what extent of the program is affected by the code. Lines of Code (LoC) is used to define the total lines of code potentially affected by the bug. It is the sum of the lines of code within each method in each file. The LoC does not include the comment of blank lines in code.

2.4.2 Cyclomatic Complexity

Cyclomatic Complexity (CC) refers the complexity of the particular portion of code. It is calculated by using the dependency relationships of the code. As defined in "*Structured Testing: A Testing Methodology Using the Cyclomatic Complexity*" [15], CC it is represented as:

$$CC = E - N + 2$$

where E = edges, and N represents the number of nodes of the dependencies. A node is illustrated in the graph below (Figure 3).

The graph below illustrates a visual example of nodes and edges. Nodes and edges refer to how functions as called or call other functions in a program. As seen the dependency graph, the cyclomatic complexity for the function *'canonify'* more nodes and edges, means more relationships and 'complexity' of the function. This graph was generated by the program slicing tool (Codesurfer) used in this research.

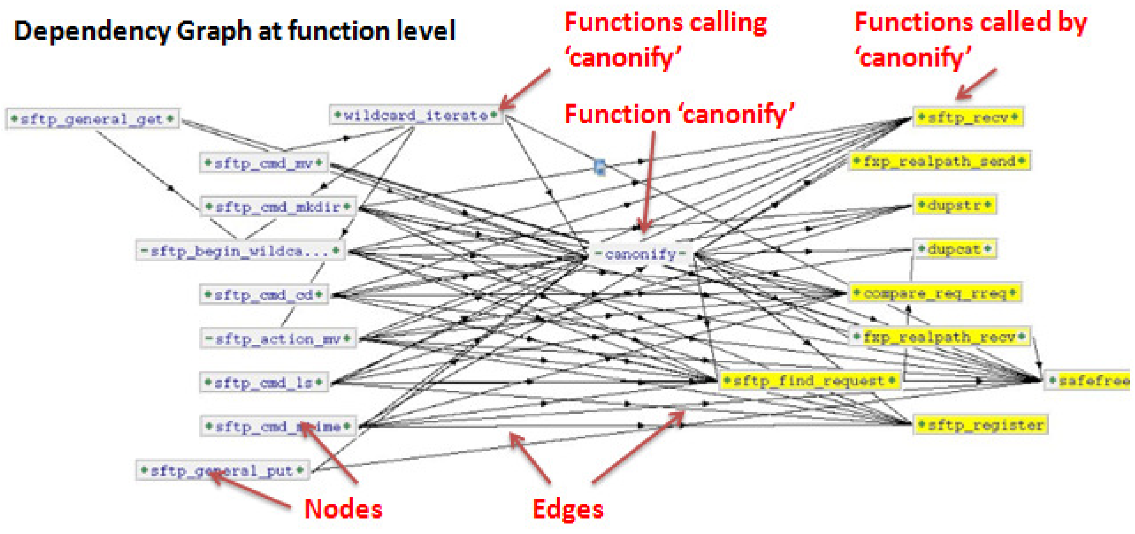


Figure 3 Program Slicing Cyclomatic Complexity Metric: Dependency Graph

2.5 Research Approach

This section discusses the approach taken for this research (Figure 4) to test the hypothesis. The next sections show each step of the approach and the definition of each attribute extracted.

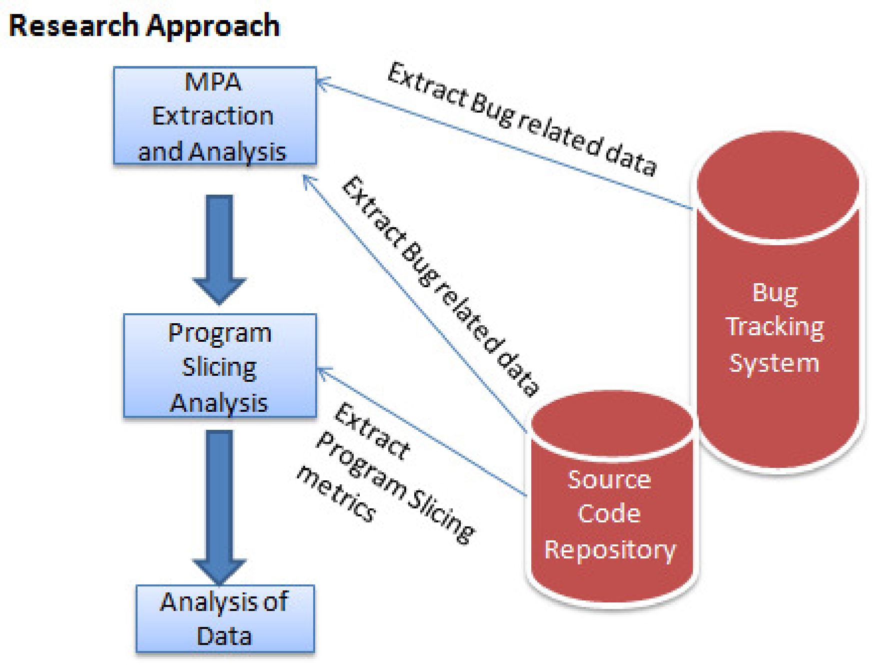


Figure 4 Research Approach

2.5.1 MPA Extraction and Analysis

For this research, the test data chosen was to be from Open Source Software. The main reason is that it is easily accessible as well as does not require special permission or software licenses to analyze datasets. Also there is a large range of software projects available.

The MPA extraction and analysis step involves data mining and extraction of bug related attributes from both the source code repository and the bug tracking system. The next two sections describes what data is extracted in order for the analysis

2.5.1.1 Bug Tracking System

Table 5 shows the data that is extracted from the bug tracking system.

Attribute	Description
Bug ID	This is used to identify the bug
Date Opened	This is when the bug was reported
Date Closed	This is when the bug was reported as fixed.
Bug Priority	This could be used to help understand its perceived importance.

Table 5 Data extracted from the bug tracking system

2.5.1.2 Source Code Repository

Table 6 is a description of the data that is extracted in the source code repository

Attribute	Description
Revision ID	Identification number used to track changes made to source code.
Bug ID	This is the reference to what changes where related to a bug.
Commit Date	This is the date when the code change was performed.

Table 6 Data extracted from source code repository

2.5.2 Program Slicing Analysis

This step involves analysis of the code using the program slicing metrics. Using the two metrics explained in section 2.4 the following explains in detail how each metric was used to for bug classification.

2.5.2.1 Bug LoC (Lines of Code)

This metric is proposed to measure how much of the code is potentially affected by the bug. This will be measured using the program slicing metric

‘Lines of Code (LoC)’ as defined in section 2.2.1. The range of a bug is summarized in the equation below:

$$BugLoC = \sum_{f \in editedfiles} LoC(f)$$

where editedfiles refers to the source code files that were changed when the bug was being fixed, and LoC(f) is lines of code of file *f*. Since there is more than one file related to a bug, the Bug LoC is a sum of the files lines of codes.

2.5.2.2 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 2.4.2. The severity of a bug is:

$$BugCC = \sum_{f \in editedfiles} \sum_{g \in \mathrm{func}(f)} CC(g)$$

where editedfiles refers to the source code files that were changed when the bug was being fixed, LoC(f) is lines of code of file *f*, func(*f*) is a set of functions defined in file *f*, and CC(g) is cyclomatic complexity of function *g*.Since the CC is calculate or the functions, sum of the CC of all functions in all files affected by the bug.

2.5.3 Analysis of Data

Using the MPA data extracted and the program slicing metrics extracted from the software project, the bugs will be grouped according to similar program slicing metrics characteristics. For this research, since the research is focused on the quality of micro processes involved in bug fixing, the analysis involves identifying how program slicing affects the execution of micro processes of a bug.

3. Experiment

3.1 Overview

This chapter describes the experiments taken in the implementation of the approach for this research. Section 3.2 presents the tools and techniques used for the experiments. Section 3.3 outlines the test subjects for the experiment.

3.2 Analysis Tools

This subsection describes the tools that were used for the experiments. It describes in detail the tool selected for both the program slicing and the tool used to extract the MPA data.

3.2.1 Program Slicing Tool

The sections below describe how the program slicing tool was selected and its limitations of its use.

3.2.1.1 Tool Selection

Based on previous surveys [16] on the available program slicing tools, three program slicing tools were tested. Each tool was installed, tested and the most appropriate tool selected. Below is the summary of the tools. (Table 7)

Tool	Operating System	Target Language	Program Slicing metrics
Codesurfer	**Linux**	**C, C++**	**Yes**
Indus Kaveri Java Slicer	Windows	Java	No
JSlice	Linux	Java	No

Table 7 List of Program Slicing Tools Evaluated

Although Java based programming projects were preferred, during the evaluation of the Indus Kaveri Slicer [17], java code needed to be translated to bytecode in order for program slicing to occur. Also both Jslice [18] and Indus Kaveri Slicer were noTable to easily provide the needed program slicing metrics for this research. In the end Codesurfer[1][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 mentioned in the section 2.4.

3.2.1.2 Program Slicing Limitations

The main limitation of Codesurfer is that the projects to be analyzed can only be developed in the C, C++ programming language.

3.2.2 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, it was developed to search both SVN and CVS repositories. This would not become a constraint on the research.

3.2.2.1 SVN Extraction

Below are the detailed process used to extract data from the SVN data repositories. Figure 8 below illustrated the searching algorithm to extract the data.

[1] Previously known as the Wisconsin Program Slicing Project

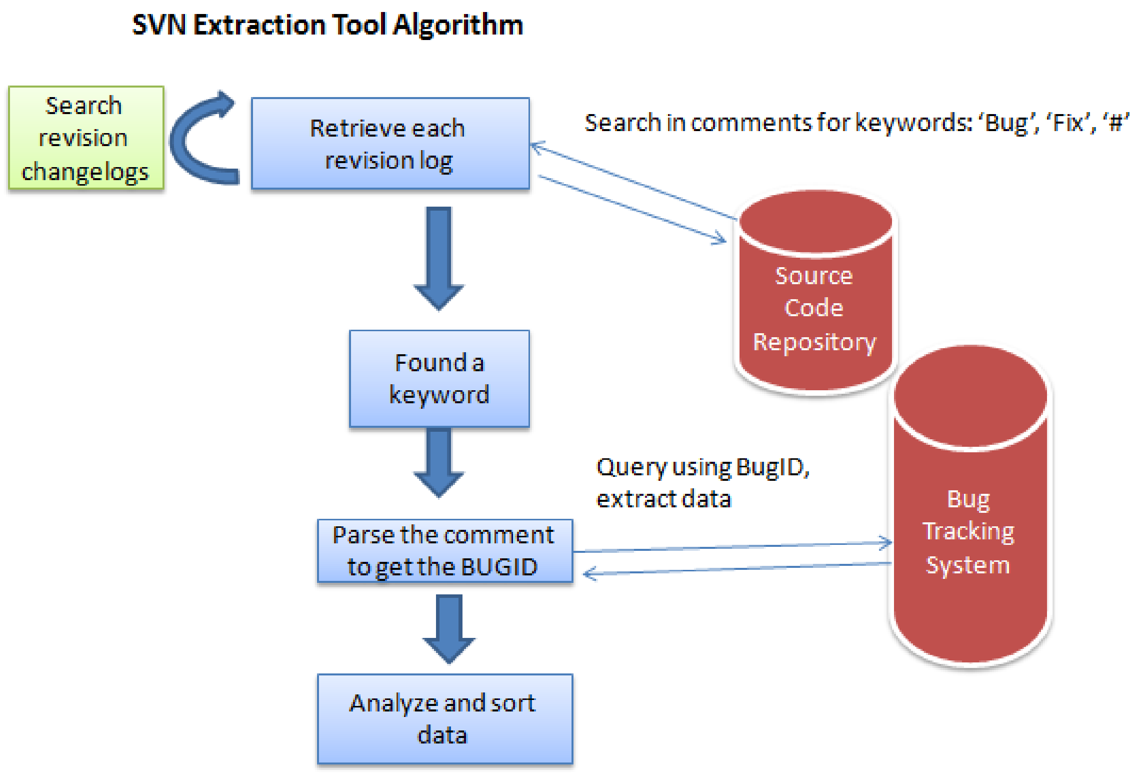

SVN Extraction Tool Algorithm
Search revision changelogs
Retrieve each revision log
Search in comments for keywords: 'Bug', 'Fix', '#'
Source Code Repository
Found a keyword
Query using BugID, extract data
Bug Tracking System
Parse the comment to get the BUGID
Analyze and sort data

Figure 8 Algorithm for Extraction of Bug Data from SVN Repository

Step 1 - Retrieve each revision, then search the comments/ change log for following keywords: 'bug', 'fix', '#'.

Step 2 - If it detects a bug, it then searches for a bug id. Once detected it also extracts the necessary bug related data from the source code repository.

Step 3- Finally using the bug id, we extract bug details from related bug tracking system.

Step 4- Analyze and sort data. This step involves the storing of the data so that it can be easily queried for information. Please refer to section 3.1.2.5 for more information.

The following Table presents the bug related data extracted by the tool.

Data Repository	Bug Attribute	Description
Source Code Repository	Revision ID	Revision Identification number
	Date of Bug Fix	When the files related to the bug fix was edited.
	Bug ID	Bug Identification number
Bug Tracking System	Bug ID	Bug Identification number
	Create Date	Date when bug was created
	Assigned Date	Date when developer was assigned to bug
	Closed Date	Date when the bug was fixed
	Priority	Priority level of the bug

Table 9 Bug Related Data extracted by the Extraction Tool

3.2.2.2 CVS Extraction

Figure 10 shows the detailed process used to extract data from the CVS repositories. Do note that in the CVS extraction, each file change logs as analyzed as opposed to SVN revisions in 3.1.2.1

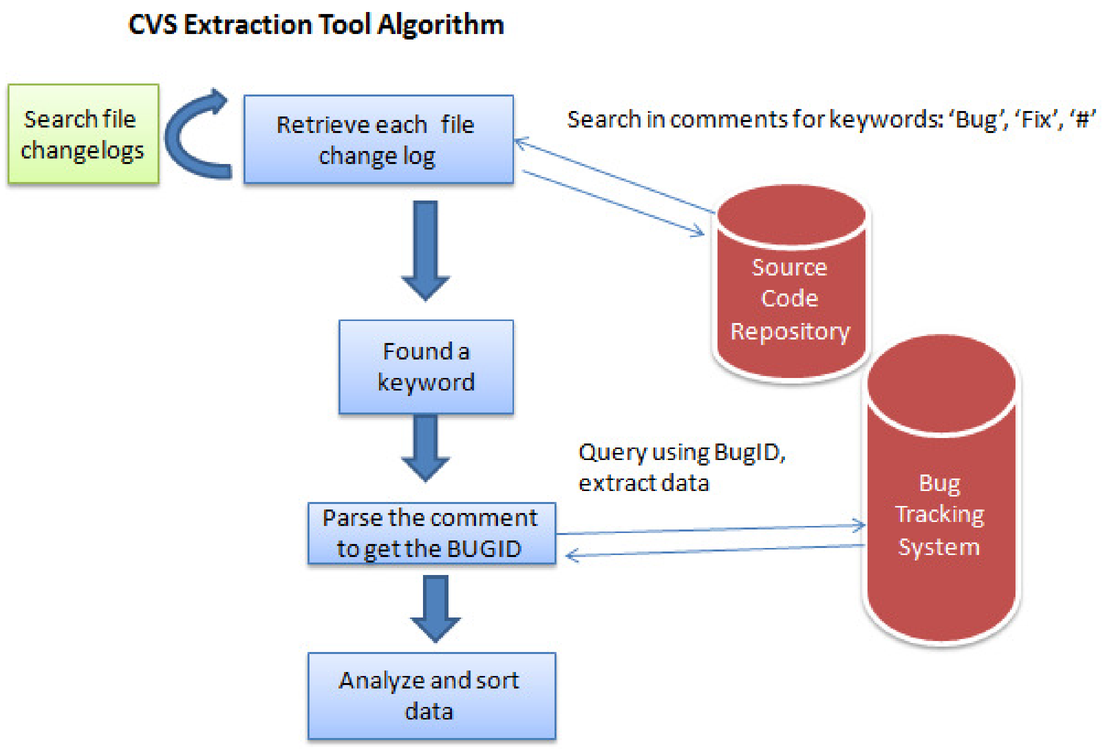

CVS Extraction Tool Algorithm
Search file changelogs
Retrieve each file change log
Search in comments for keywords: 'Bug', 'Fix', '#'
Source Code Repository
Found a keyword
Query using BugID, extract data
Parse the comment to get the BUGID
Bug Tracking System
Analyze and sort data

Figure 10 Algorithm for Extraction of Bug Data from CVS Repository

3.2.2.3 Tool Implementation

The tool was implemented using the java programming language. The java class *java.io.InputStreamReader, java.io.net* as well as *java.String were* implemented in a web spider and parsing tool.

3.2.2.4 Analysis of Data

To help with the analysis of the data, Microsoft access 2007 was used to query the data areas and then Microsoft excel 2007 graphs to display the results of the data.

3.3 Test Subjects

This subsection covers the test subjects that were used for the experiment. The selection criteria and brief introduction of the projects are given.

3.3.1 Selection Criteria

Due to the limitations of Codesurfer only able to analyze projects based on the C programming language. Projects were selected based on the following criteria:

1. *Program Slicing Capability* – Due the program slicing tool limitations, C++ projects were selected.

2. *Quality of Bug Extraction* – Another criterion was if there was a reference between the bug tracking system and the source code repository. Many software programs where not suitable for this experiment as their source code repository did not reference the related Bug Tracking System.

   The tool was designed to handle different online bug tracking systems and source code repositories. As long the needed data was being able to be retrieved, the tool could be modified to extract the data. This includes customized parsing specific for the project.

3.3.2 Test Projects

Using Sourceforge.net as a source of the Open Source Software projects C programming language based software projects were chosen based randomly, the three software projects met the selection criteria discussed in the previous section (3.3.2). The next subsection gives a brief introduction to each project.

3.3.2.1 Scintilla[2]

Scintilla is an editing component for Win32 and GTK+. It comes bundled

[2] http://www.scintilla.org

with SciTE, which is a Scintilla based text editor. It is used when editing and debugging source code.

3.3.2.2 wxWidgets[3]

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.

3.3.2.3 Filezilla[4]

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.

Finally, Table 11 below summarizes the type of data repository and micro process model of each test project. As shown below, all project source code are hosted independently. Scintilla uses the Sourceforge Bug Tracker as its bug tracking system. WxWidgets and Filezilla both use the Trac issue tracking system[5] to help manage their bugs. Table 12 show other particulars of the project such as number of developers, revision changes and how long the projects have been active.

[3] http://www.wxwidgets.org
[4] http://filezilla-project.org
[5] http://trac.edgewall.org/

Software Project	Date Accessed	Source Code Repository (hosted)	Source Code Repository Type	Online Bug Tracking System (hosted)
Scintilla	2009/12/20 (Up to Ver. 2.01)	Independently hosted[6]	CVS	Sourceforge[7]
WxWidgets	2009/12/18 (Up to Rev 62931)	Independently hosted[8]	SVN	Trac Open Source Project[9]
Filezilla	2009/11/25 (Up to Rev. 3295)	Independently hosted[10]	SVN	Trac Open Source Project[11]

Table 11 List of Test Subjects Data Repository Types and Systems

Software Project	Developers	Revision changes	Project Status
Scintilla	26 Developers	78 (Ver.2.02)	1999/03/14 – 2010/01/25 (11 years)
WxWidgets	34 Developers	63267(Ver.2.9.0)	1998/05/20 – 2010-/01/25 (12 years)
Filezilla	3 Developers	3611 (Ver. 3.3.1)	2004/03/8 – 2010/01/25 (5 years)

Table 12 Current Status of Test Projects (2010/01/26)

6 http://scintilla.cvs.sourceforge.net/scintilla/

7 http://sourceforge.net/tracker/?group_id=2439&atid=102439

8 http://svn.wxwidgets.org/svn/wx/wxWidgets/
9 http://trac.wxwidgets.org/
10 http://svn.filezilla-project.org/filezilla/
11 http://trac.filezilla-project.org/

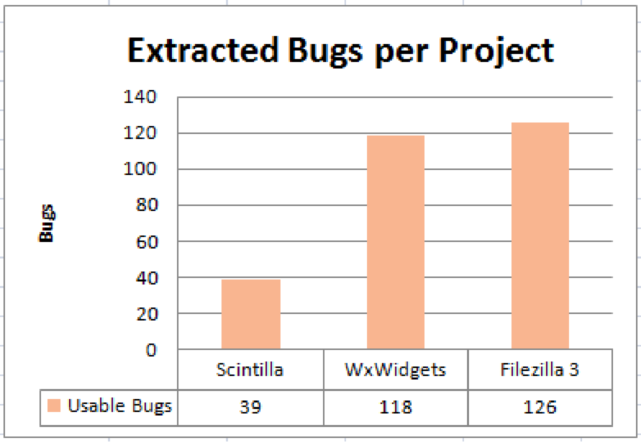
4. Experiment Findings

4.1 Overview

This chapter shows the experiment results for this project. The first sub section illustrates the general summarized data. Following this the analysis and classification of bugs is explained. Then based on these groupings, program slicing metrics and micro process are analyzed for each project

4.2 General Characteristics

This section gives an overview of each project as compared to the rest of the test group. Figure 13 below illustrates the differences in lines per code, extracted bugs and functions per project.



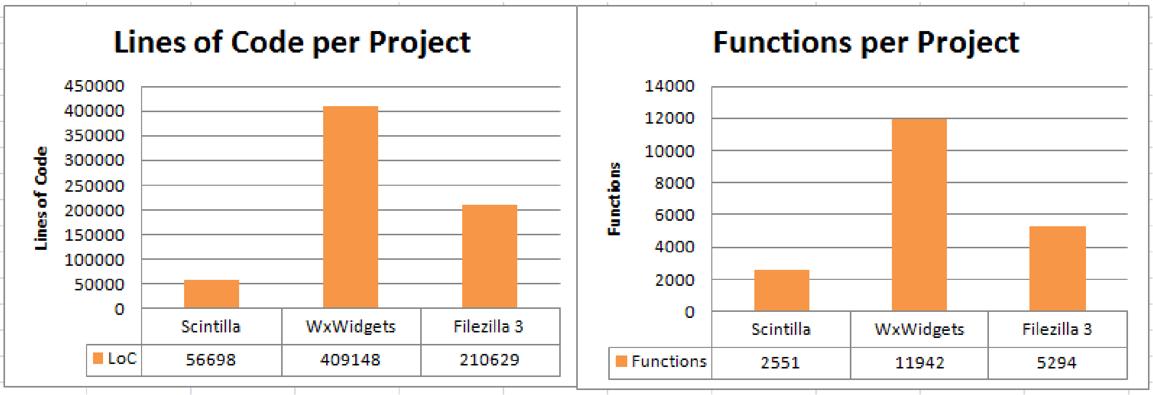


Figure 13 General Comparisons between Test Projects

Overall the results suggest that wxWidgets is the largest software project as compared to Scintilla and Filezilla since having more lines of code and functions. Scintilla is the smallest of the three projects, having the lowest lines of code and function. In regards to this research, both wxWidgets had similar number of extracted bugs where Scintilla has less.

4.3 Bug Classification

For the classification of the bugs, a proposed analysis method was using how long it takes to fix a bug. The number of days taken to fix a bug is calculated as:

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

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

4.3.1 Bug Distribution

The bug distribution was used to group the bugs. As seen in Figures 14, the first 30 days was plotted against the program slicing metrics. It was noticed that the bugs from 0 to 10 days were evenly distributed. From 10 days onwards, the distance between the bugs seems to increase. So we identify these bugs are the one group from 0-10 days.

Since the distribution covers a very large range of data, logarithmic graphs were used to plot the data. The Figure 15 shows the distribution of all bugs for this research. It can be 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 then the third group from 64 days onwards.

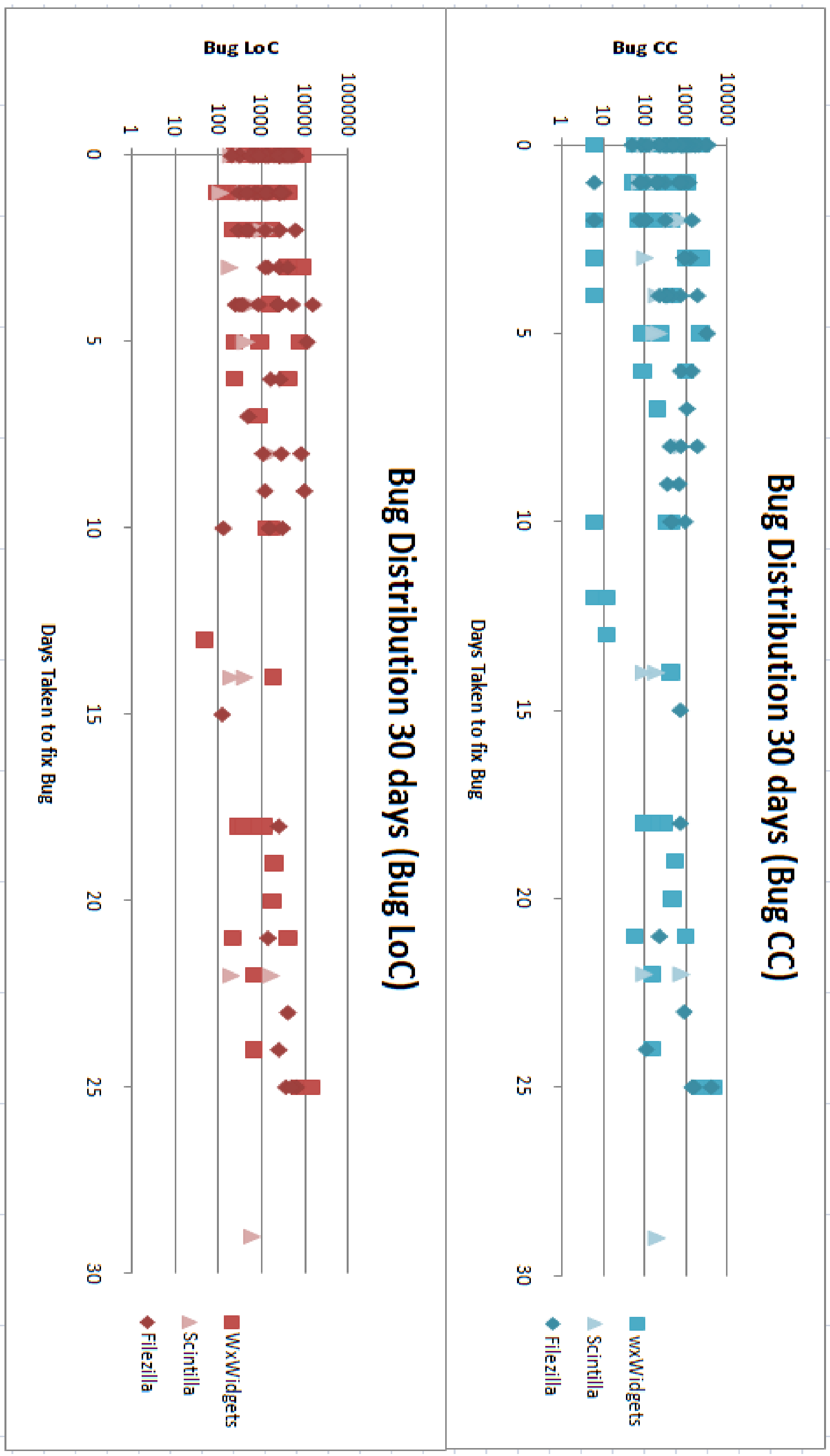


Figure 14 Bug Distribution over a 30 Day period

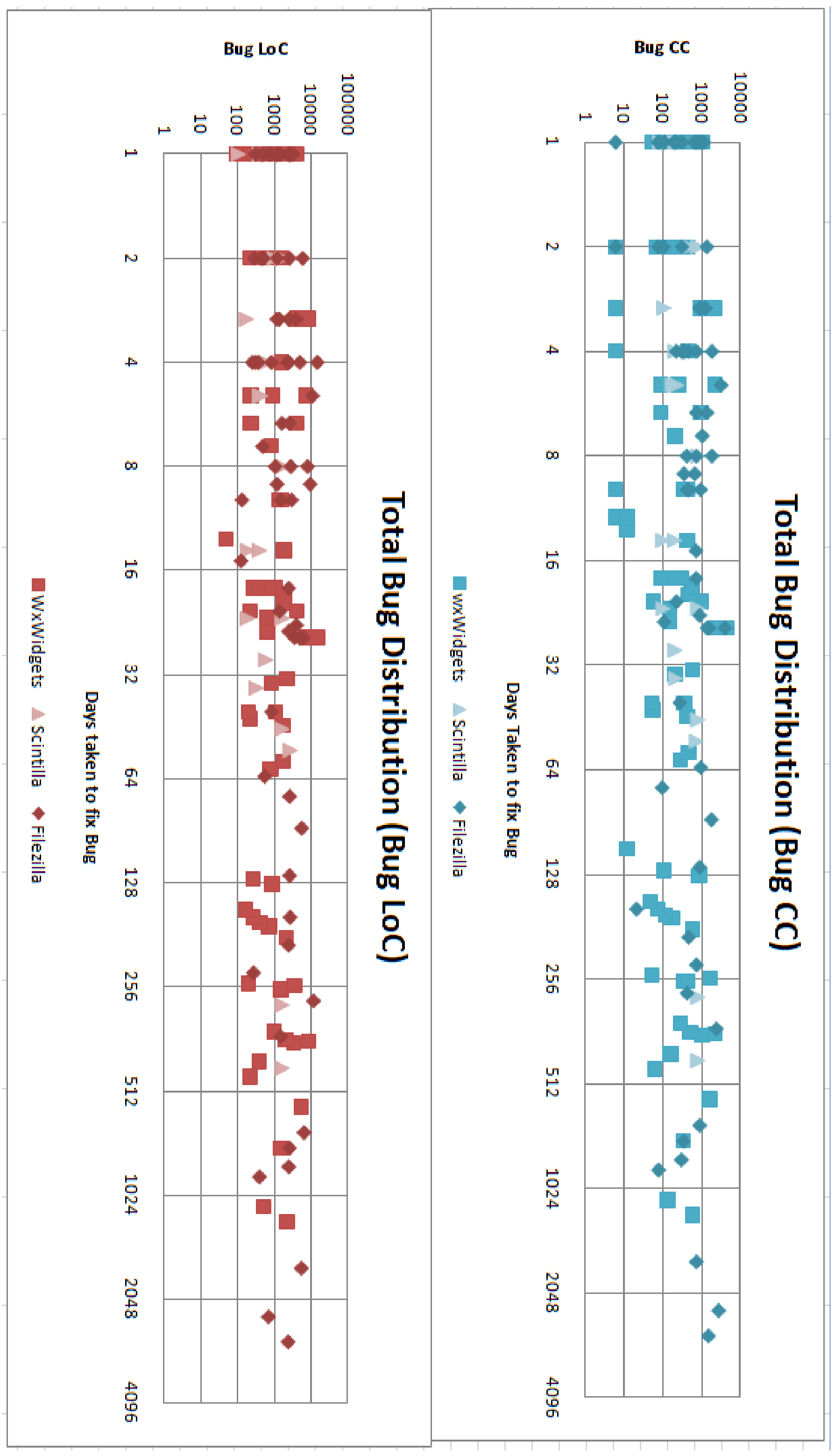


Figure 15 Total Bug Distribution on log-log graph

4.3.2 Bug Classification Group Features

Based on the bug distribution, we define the following groupings of the bugs:

- ✧ 0 to 10 days
- ✧ 11 to 64 days
- ✧ 64 days and more

Using these groups, other attributes of the bugs were analyzed to see if what other characteristics are common. The graph (Figure 16) below shows the bug priority and frequency of bugs in each group. It can be seen that in all projects frequency, the majority of bugs are fixed in 0-10 days. It is interesting to note the Filezilla, the blocker bugs appear in the 0-10 days, however in wxWidgets has in bugs more than 64 days old.

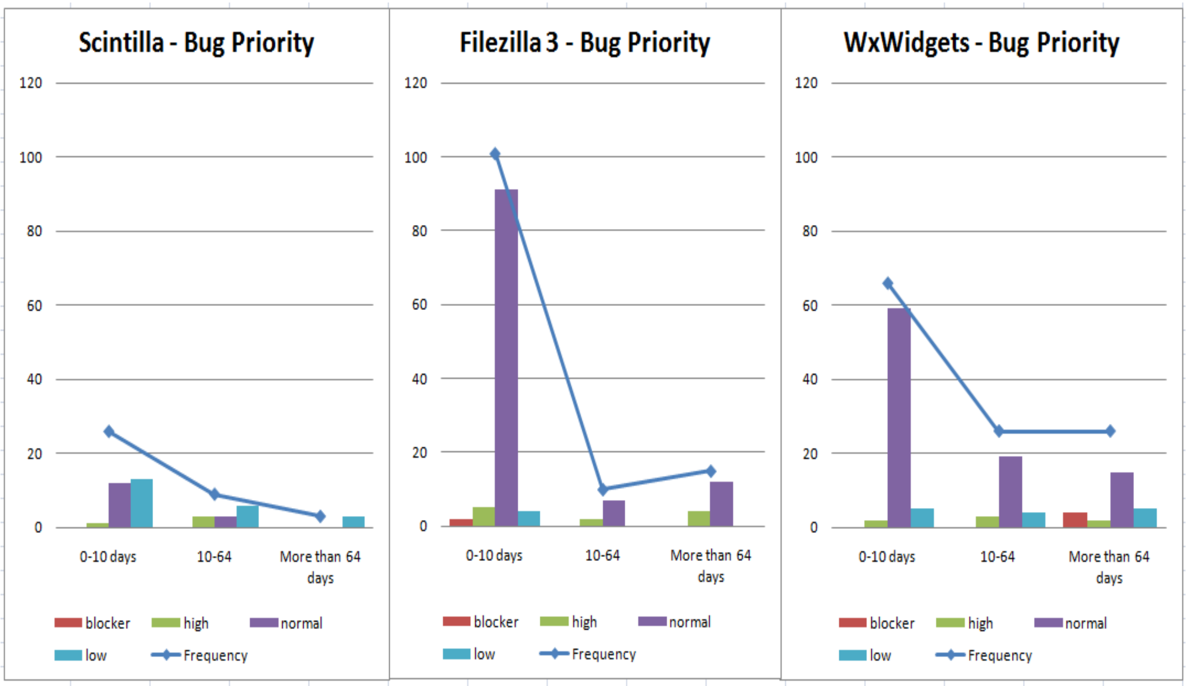


Figure 16 Bug Classification Analysis by Priority and Frequency

4.4 Program Slicing Metrics

This section describes the results of applying the program slicing metrics in the research. The first section gives a general comparison of the metrics at project level. After, each project is individually analyzed according to the proposed bug classification.

4.4.1 General Characteristics

The graphs below (Figure 17) show the general program slicing statistics for at the function level. As you can see, Scintilla has the highest cyclomatic complexity per function as well as the highest lines of code per function. WxWidgets and Filezilla both that equal CC and LoC ratings. The results validate our metrics as program analysis usually state that smaller projects with less lines of code have more complex functions and more lines of code [21].

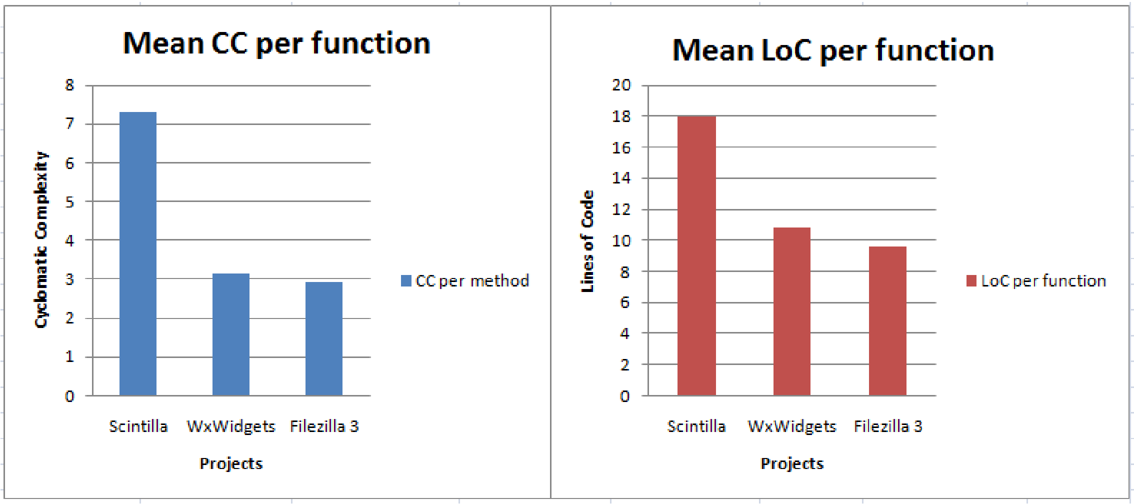


Figure 17 Comparison of Mean Program Slicing Metrics across Test Projects

4.4.2 Bug Classification Analysis

This subsection presents the program slicing metrics statistics when the proposed bug classifications were applied.

4.4.2.1 Bug CC Analysis

The graphs in Figure 18 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 bug that have more than 64 days have low complexity and a lower distribution.

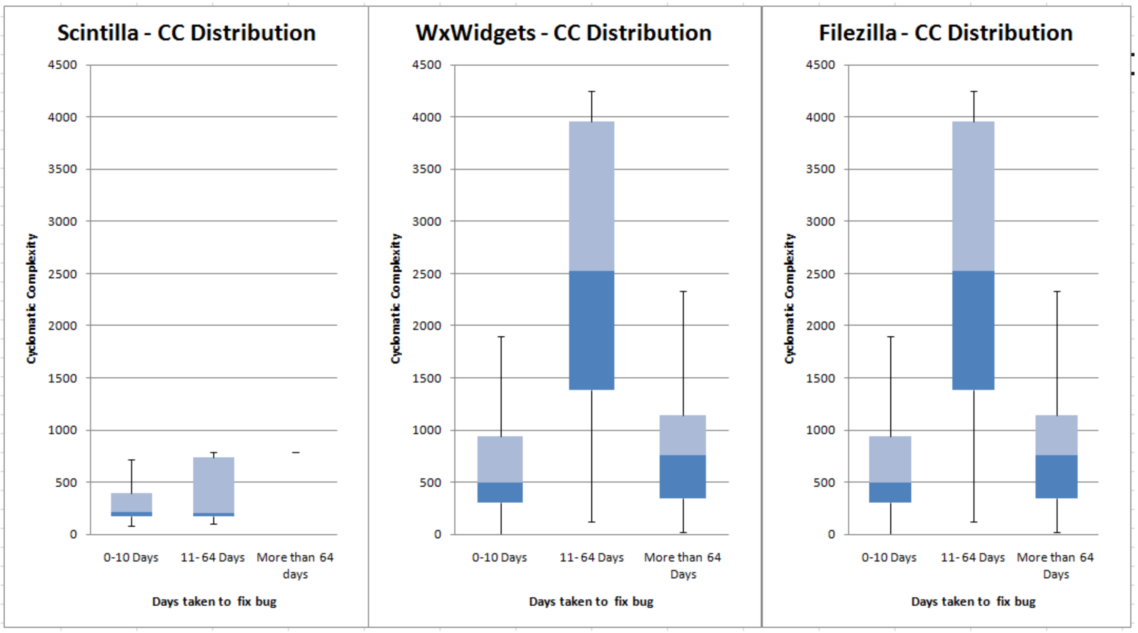


Figure 18 Cyclomatic Complexity Distribution using Bug Classification across Projects

4.4.2.2 Bug LoC Analysis

The box plots illustrated in Figure 19 show the Bug LoC for bugs within the bug classifications proposed in 4.2.2. The graphs clearly show that bugs that take 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 to be able to make concrete analysis.

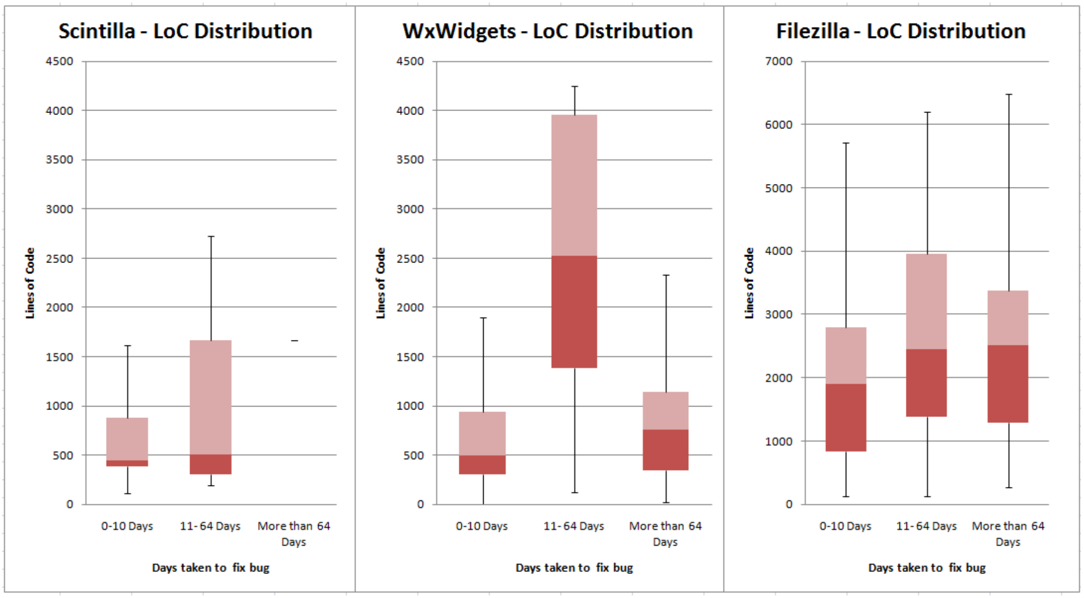


Figure 19 Lines of Code Distribution using Bug Classification across Projects

4.5 Micro Process Analysis

This section presents the micro process analysis. The analysis involves studying the execution of the processes explained in sub section 2.4.1.

4.5.1 Bug Fixing Process

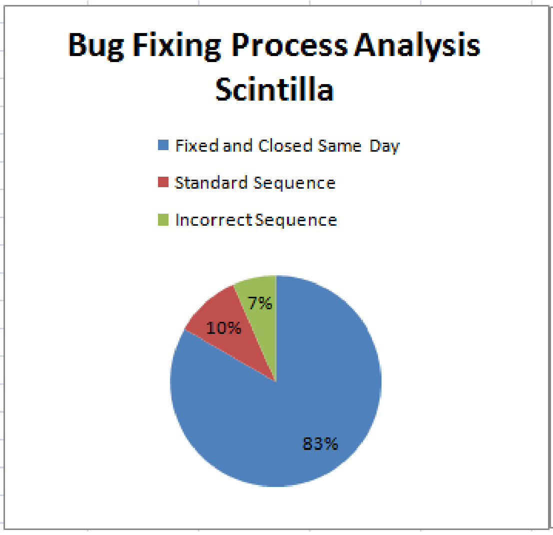
This subsection presents the comparisons of the execution sequences of the bug fixing process for the three projects. Refer to Appendix A, B and C for details. As seen the results were raw and hard to understand so additional analysis was required.

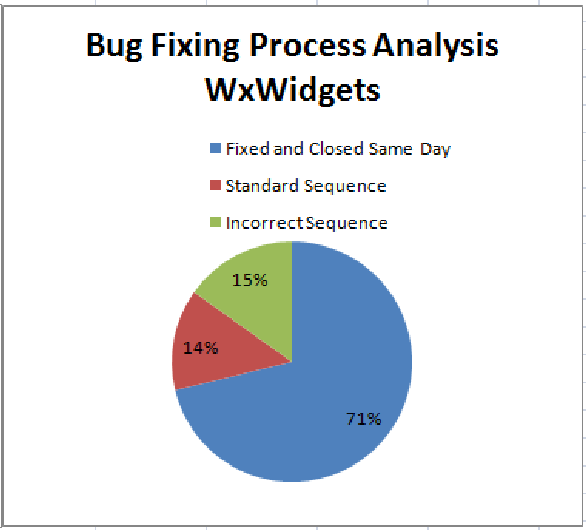
The execution was divided into these three groups to assist with the analysis. The groups are based on the model to fix bugs as outlined in section 2.3.1. The bugs with correct bug fixing process are then divided to bugs that were closed on the same day of the bug fix:

- ✧ *Fixed and Closed Same Day* – These groups of bugs contain bugs that are closed on the same day. It can be used to view bugs that were fixed but not closed just yet.
- ✧ *Standard Sequence* – This group contains the bugs that have all the processes sequenced in correct order.
- ✧ *Incorrect Sequence* – This group of bugs have incorrect sequences of the bug process compared the proposed model. The term 'incorrect' refers to abnormal sequence and behavior with the bug fixing process.

During the analysis, it was found that both Filezilla and wxWidgets did not utilize the 'assign bug' process that we use in our proposed MPA model.

The graphs in Figure 20 suggest that most bugs are usually fixed and closed in the same day. This suggests that with these projects, the verification of the fix is usually one day.





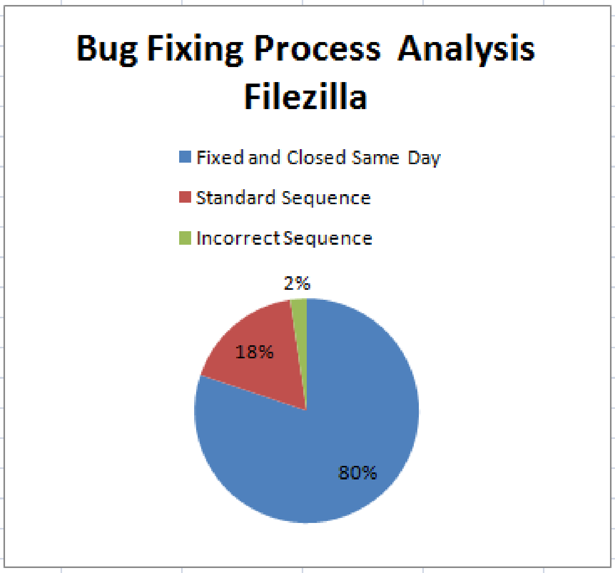
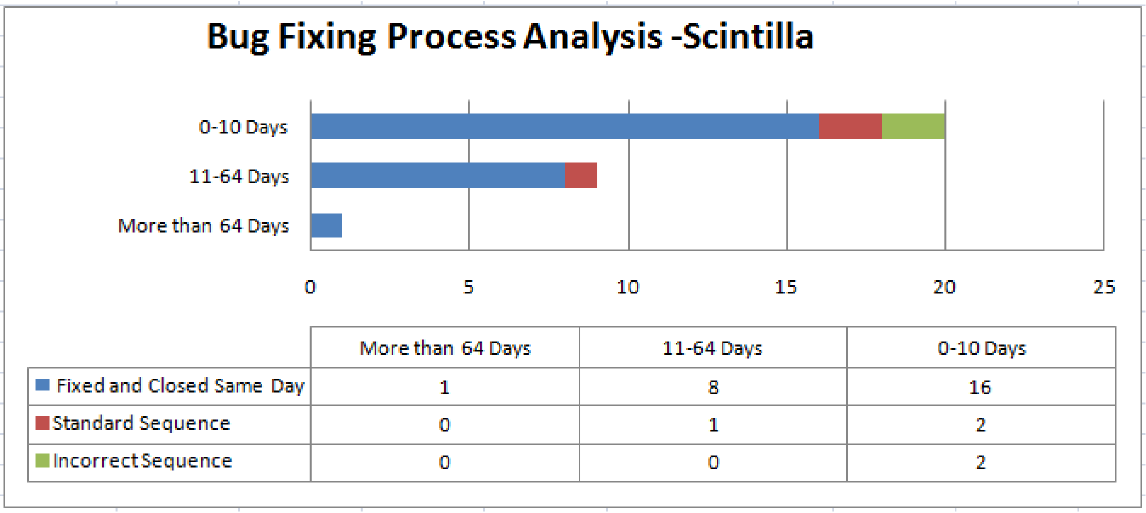


Figure 20 Graphs illustrating the distribution the grouped by the execution of bug fixing process across test projects (fixed and closed same day, standard sequence, incorrect sequence)

The next section shows a more detailed look at the micro process analysis for each project. Here the bugs are grouped according to the proposed bug classifications from section 4.3.2.

4.5.2 Bug Fixing Process using the Bug Classification

4.5.2.1 Scintilla



	More than 64 Days	11-64 Days	0-10 Days
🟦 Fixed and Closed Same Day	1	8	16
🟥 Standard Sequence	0	1	2
🟩 Incorrect Sequence	0	0	2

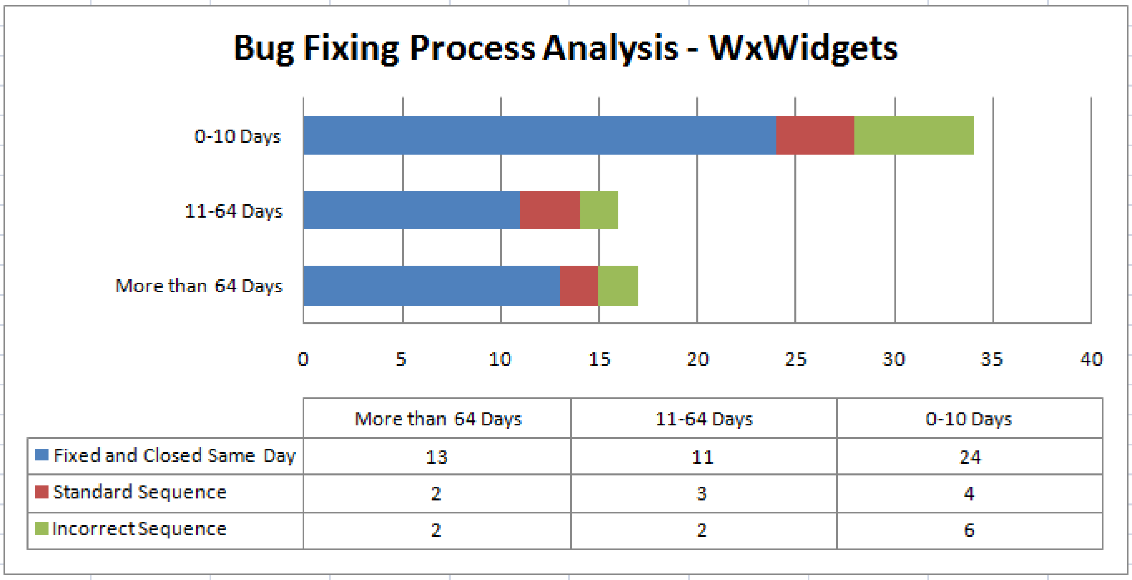
Figure 21 Scintilla execution of bug fixing process grouped by bug classification

The Figure 21 shown above indicates that 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.

4.5.2.2 WxWidgets

The Figure 22 on the next page illustrates wxWidgets bug fixing process grouped in the proposed bug classifications.

As seen in Figure 22, 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.



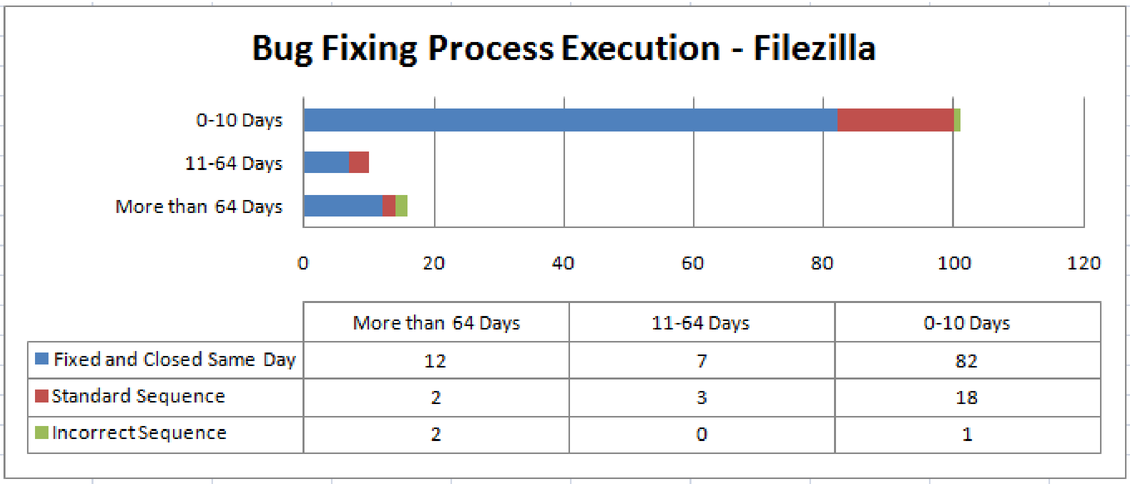
	More than 64 Days	11-64 Days	0-10 Days
🟦 Fixed and Closed Same Day	13	11	24
🟥 Standard Sequence	2	3	4
🟩 Incorrect Sequence	2	2	6

Figure 22 WxWidgets execution of Bug fixing process grouped by bug classification

4.5.2.3 Filezilla

The Figure 23 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 Filezilla, the bugs that were fixed and closed in the same day occur in all bug groups, however the most is found in bugs that took up to 10 days to fix. It is interesting to note that bugs that took 11 to 64 days did not have any incorrect sequence in its bug fixing process.



	More than 64 Days	11-64 Days	0-10 Days
🟦 Fixed and Closed Same Day	12	7	82
🟥 Standard Sequence	2	3	18
🟩 Incorrect Sequence	2	0	1

Figure 23 Filezilla execution of Bug fixing process grouped by bug classification

4.5.3 Program Slicing Metrics against Bug Fixing Process Execution

In this section the program slicing metrics are applied to the different groupings of bugs based on their process execution discussed in section 4.5.1.

4.5.3.1 Bug CC

The graphs in Figure 24 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.

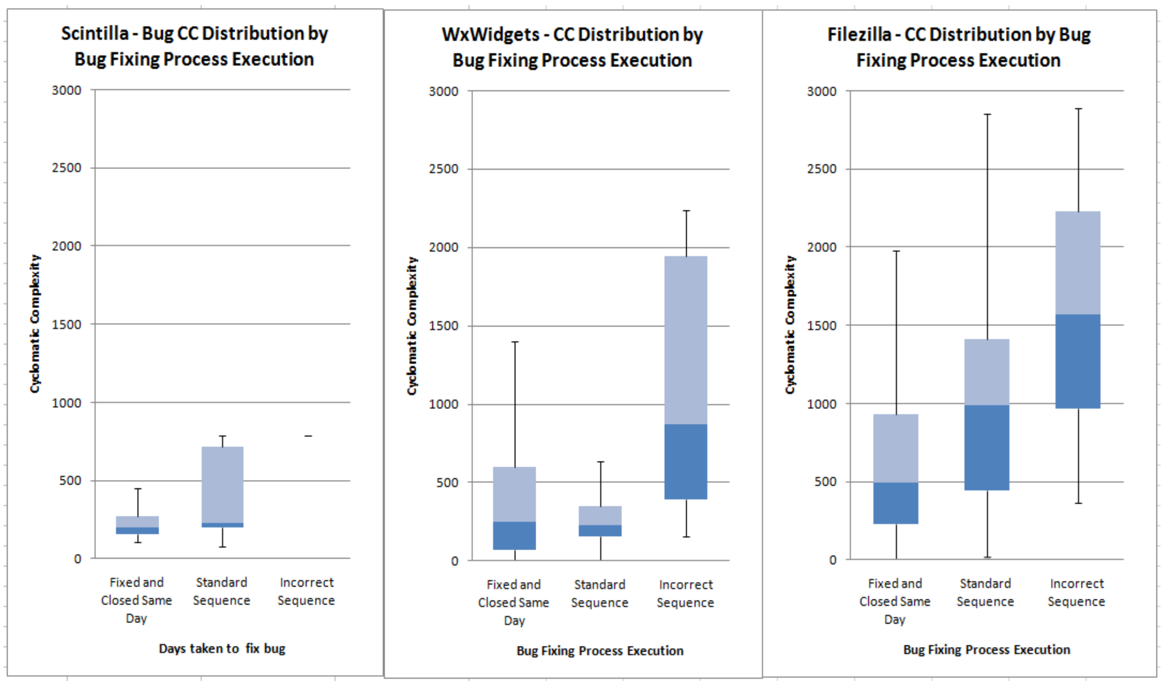


Figure 24 Bug CC by Bug Fixing Process

4.5.3.2 Bug LoC

Figure 25 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 (section 4.5.2.1), 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.

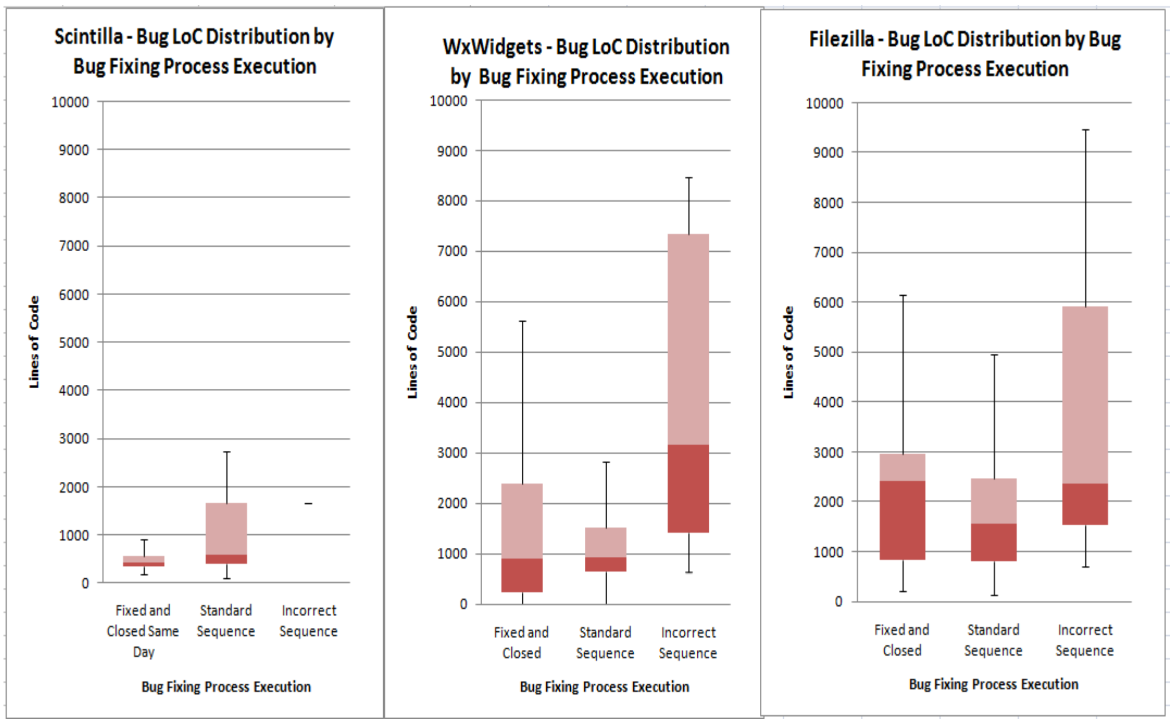


Figure 25 Bug LoC by Bug Fixing Process

5. Discussion

5.1 Overview

This chapter discusses the findings of the research. In Section 5.2, the results are discussed in relation to both the hypothesis and objective of the thesis. Section 5.3 discusses the application of the findings in the research its contribution to the field of Software Process Improvement.

5.2 Approach Challenges

The thesis 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 Open Source Software analysis data validation questionable [22].

Another limiting factor was the program slicing metrics. Due to the nature of the tool, the projects were limited to only C programs. This limits the scope to only projects to that specific language, excluding big java based projects such as eclipse project[12], which was a prime candidate for analysis.

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 rather than the validation of the approach.

[12] http://www.eclipse.org/

5.3 Findings 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.

Further analysis was done on the groupings resulting with many interesting findings. Firstly most bugs are fixed within ten days of being detected. Also priority is not a factor with the bug classification, as shown in this experiment Filezilla fixed very urgent bugs within 10 days while wxWidgets addressed urgent bugs after 64 days.

To better understand the bug classification, the distributions of program slicing metrics were analyzed according to the bug groupings. Findings in section 4.2 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. 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 were 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 for the thesis 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.

5.4 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 closed in the same day and have a higher likelyhood to have errors in its process execution.

In response to the hypotheses, the findings support Hypothesis 1 as our research was able to successfully classify bugs with similar micro process executions and have a particular program slicing metric. 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. However, further research it needed to have more confidence in these hypotheses.

5.5 Application of Research

The 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.

If successful, the research can used to help create ‘prediction models’ for bugs. For example, based on code, a bug could be classified, 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.

6. Future Works

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

More advance program slicing metrics such as lines of code affected in forward slicing and backward slicing will be explored. This a more precise metric, than the lines of code used in this study. It is envisioned that this will give more precise analysis.

Lastly, a larger sample data will be experimented to further validate the results, Current work only shows three projects. It would be preferable to test more projects.

7. Conclusion

The main objective of the research is to provide a proof of concept that there is indeed a relationship between a bugs attributes and how its related processes are executed. The results indicate that there is a relationship between the bug characteristic and its executed process. Using program slicing metrics, similar bugs showed similar process execution.

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 thesis contributes as a starting point towards this goal.

Overall goal of the research is to create a tool to enable easier classification of bugs and identification of bug more prone to incorrect micro process execution. Further work is proposed to refine the metrics as well increase the sample test group as well as development of a predictive model and tool.

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis.

….First and foremost to my supervisors and co-supervisors with this thesis: Nara Institute Science and Technology: Prof. H. Iida, Prof. K. Matsumoto, Prof. K. Nishitani and Assistant Prof. S. Kawaguchi. Thank you for your comments and guidance.
…..To Assistant Prof. S. Morisaki and the Micro Process Analysis Research group. This work has been the foundation of my thesis.
….To the Software Design and Software Engineering Laboratories. This has been my Japanese family. To all my labmates, especially sempai Kyohei for always lending an ear and guiding me with both my research and my life in Japan. Also to Kimiharu for helping with both social and research related work. To the rest of the 2nd year master students in no particular order: Shinichi, Tetsuya, Yoshihiko, Keisuke and Koji for help during my classes and just making me feel at home. It must have been hard with communication but I appreciate everyone always willing to help.
…..To the Japanese Government for the funding to make this all possible. My gratitude to the Japanese Embassy in Papua New Guinea for giving this opportunity to me.
…..To my filipino family here in Japan. To my trusty sidekick Rusty, thanks for it all. CC, for lending me your time.
…..Finally to my family back home. Mum and Dad, this is for you. I do it all for you. Wini and family, Elma and family, Emily and Guise thanks for being my role model and cheering me over the line.

‘nakamu pono gaua’

References

[1]. Herbsleb, James; Carleton, Anita; Rozum, James; Siegel, Jane; & Zubrow, David. "*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. Kulpa and Kent A. Johnson. (2003)."*Interpreting the CMMI: A Process Improvement Approach*" Auerbach Publications, 414 pages. ISBN: 0-8493-1654-5. (CSUE Body of Knowledge area: Software Quality Management)

[3]. Schmauch, C. H. 1995 "*ISO 9000 for Software Developers*". 2nd. ASQ Quality Press.

[4]. S. Beecham, T. Hall and A. Rainer, *"Software process problems in twelve software companies: an empirical analysis"*, Empirical Software Engineering 8 (2003), pp. 7–42

[5]. N. Baddoo and T. Hall, *"De-Motivators of software process improvement: an analysis of practitioner's views"*, Journal of Systems and Software 66 (1) (2003), pp. 23–33

[6]. M. Niazi, M. Ali Babar, *"De-motivators for software process improvement: an analysis of Vietnamese practitioners' views"*, in: International Conference on Product Focused Software Process Improvement PROFES 2007, LNCS, vol. 4589, 2007, pp. 118–131.

[7]. J.G. Brodman and D.L. Johnson, *"What small businesses and small organizations say about the CMMI"*, Proceedings of the 16th International

Conference on Software Engineering (ICSE), IEEE Computer Society (1994)

[8]. A. Rainer and T. Hall, *"Key success factors for implementing software process improvement: a maturity-based analysis"*, Journal of Systems and Software 62 (2) (2002), pp. 71–84

[9]. C. Yoo, J. Yoon, B. Lee, C. Lee, J. Lee, S. Hyun and C. Wu, *"A unified model for the implementation of both ISO 9001:2000 and CMMI by ISO-certified organizations"*, The Journal of Systems and Software 79 (7) (2006), pp. 954–961.

[10]. Armbrust, O., Katahira, M., Miyamoto, Y., Münch, J., Nakao, H., and Ocampo, A. 2009. *"Scoping software process lines"*. Softw. Process 14, 3 (May. 2009), 181-197.

[11]. S. Morisaki, H. Iida, "*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), pp.26-30, Munich, Germany, Aug. 2007.

[12]. Weiser, M. 1981. "*Program slicing*". In Proceedings of the 5th international Conference on Software Engineering (San Diego, California, United States, March 09 - 12, 1981). International Conference on Software Engineering. IEEE Press, Piscataway, NJ, 439-449.

[13]. Tracy Hall, Paul Wernick, "*Program Slicing Metrics and Evolvability: an Initial Study*," Software Evolvability, IEEE International Workshop on, pp. 35-40, IEEE International Workshop on Software Evolvability (Software-Evolvability'05), 2005.

[14]. Kai Pan, Sunghun Kim, E. James Whitehead, Jr., "*Bug Classification*

Using Program Slicing Metrics," Source Code Analysis and Manipulation, IEEE International Workshop on, pp. 31-42, Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06), 2006

[15]. WATSON, A. AND MCCABE, T. 1996. *"Structured testing: A testing methodology using the cyclomatic complexity metric"*. National Institute of Standards and Technology, Gaithersburg, MD, (NIST) Special Publication, 500-235.

[16]. Xu, B., Qian, J., Zhang, X., Wu, Z., and Chen, L. 2005. *"A brief survey of program slicing"*. SIGSOFT Softw. Eng. Notes 30, 2 (Mar. 2005), 1-36. DOI= http://doi.acm.org/10.1145/1050849.1050865

[17]. Venkatesh Prasad Ranganath , John Hatcliff, *"Slicing concurrent Java programs using Indus and Kaveri"*, International Journal on Software Tools for Technology Transfer (STTT), v.9 n.5, p.489-504, October 2007

[18]. Wang, T. and Roychoudhury, A. 2008. *"Dynamic slicing on Java bytecode traces"*. ACM Trans. Program. Lang. Syst. 30, 2 (Mar. 2008), 1-49.

[19]. Paul Anderson , Mark Zarins, *"The CodeSurfer Software Understanding Platform"*, Proceedings of the 13th International Workshop on Program Comprehension, p.147-148, May 15-16, 2005

[20]. Bevan, J., Whitehead, E. J., Kim, S., and Godfrey, M. 2005. *"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, September 05 - 09, 2005). ESEC/FSE-13. ACM, New York, NY, 177-186.

[21]. McCabe, Thomas J. 1976 “*A Complexity Measure*”. IEEE Transactions on Software Engineering. SE-2,

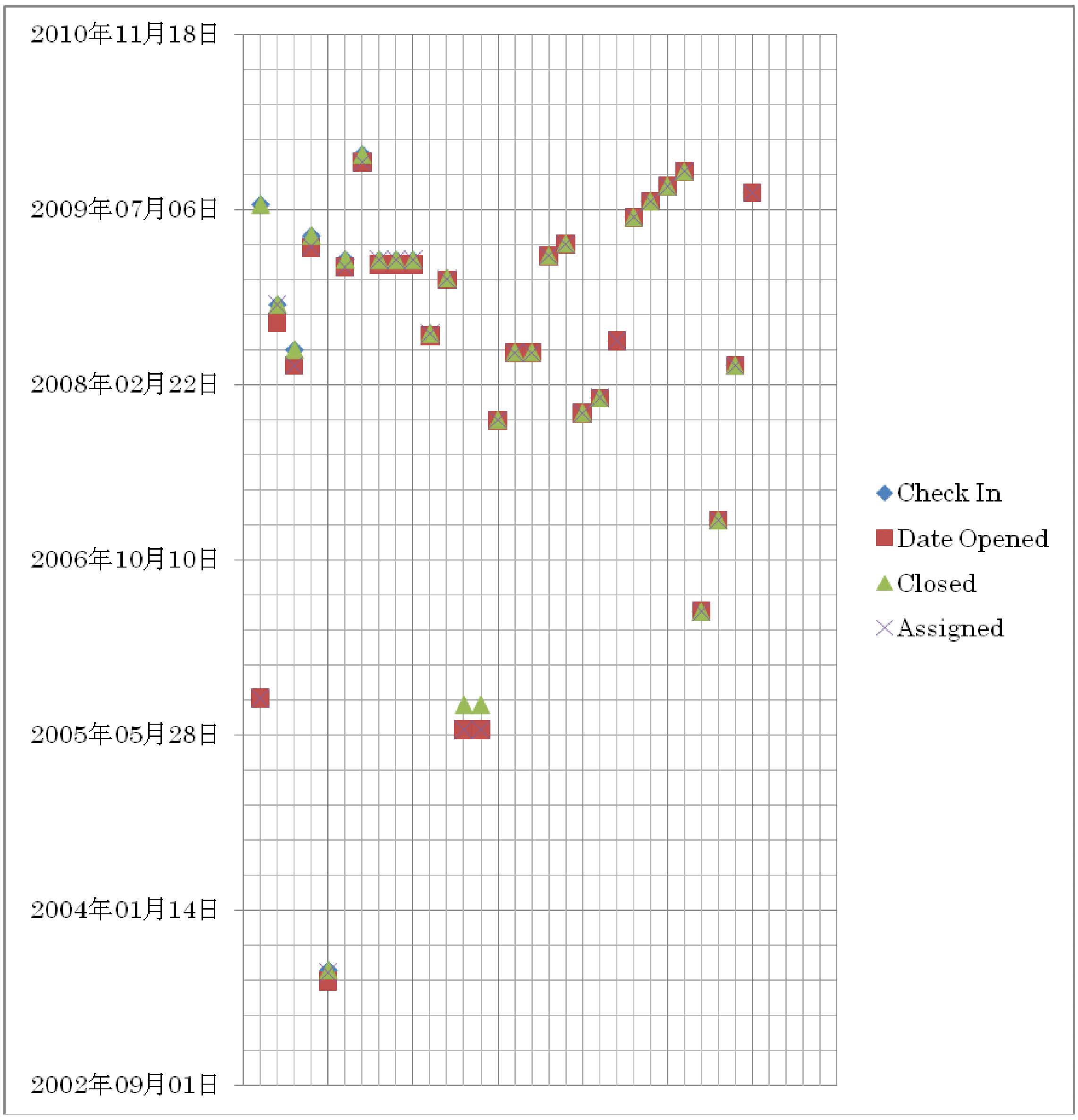
[22]. J. Howison and K. Crowston, "*The perils and pitfalls of mining SourceForge.*," presented at Mining Software Repositories Workshop,International Conference on Software Enginnering (ICSE 2004), Edinburgh, Scotland,May 25., 2004.

Appendix

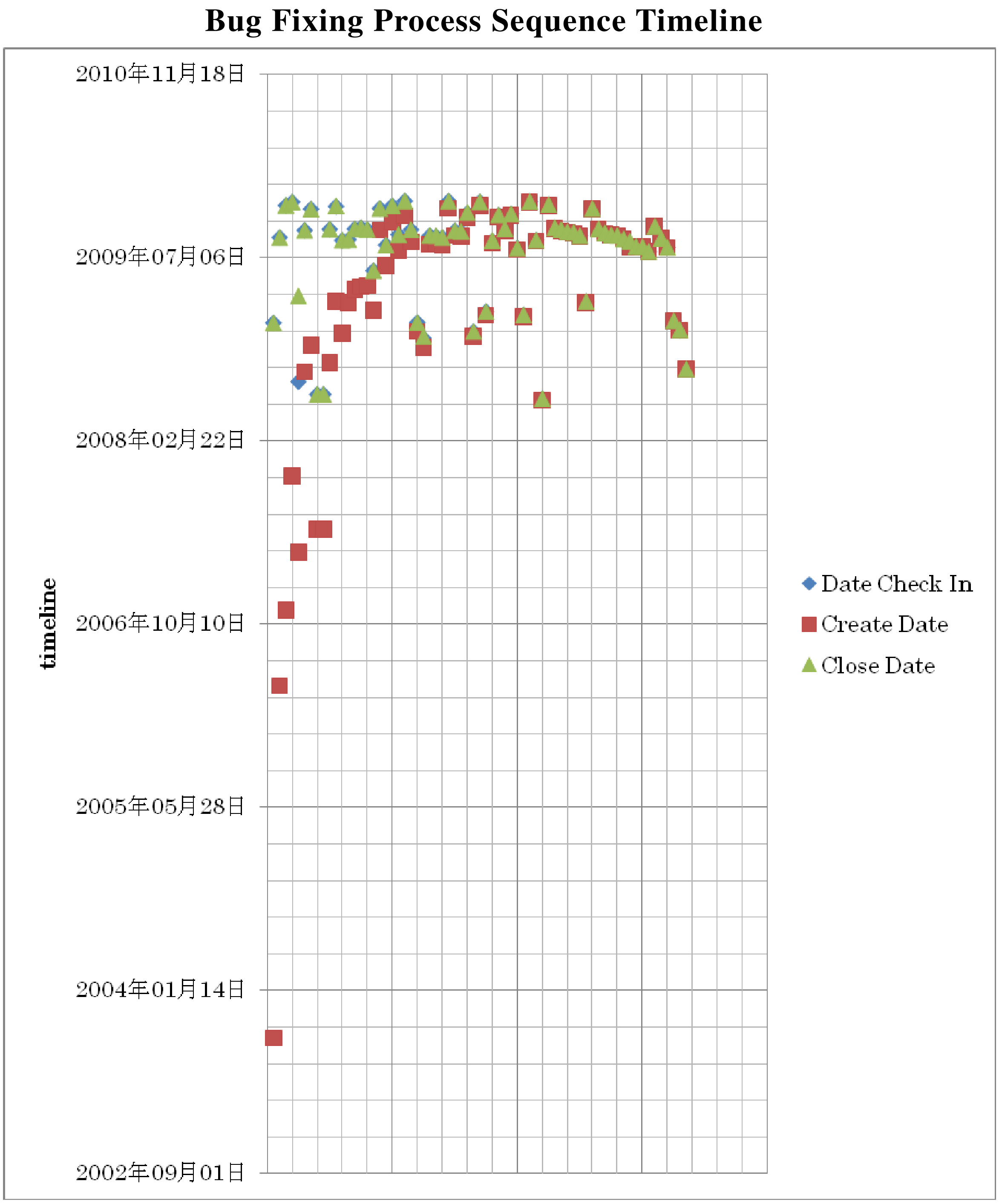
Bug Fixing Process Executions

Appendix A –Scintilla

Bug Fixing Process Sequence Timeline

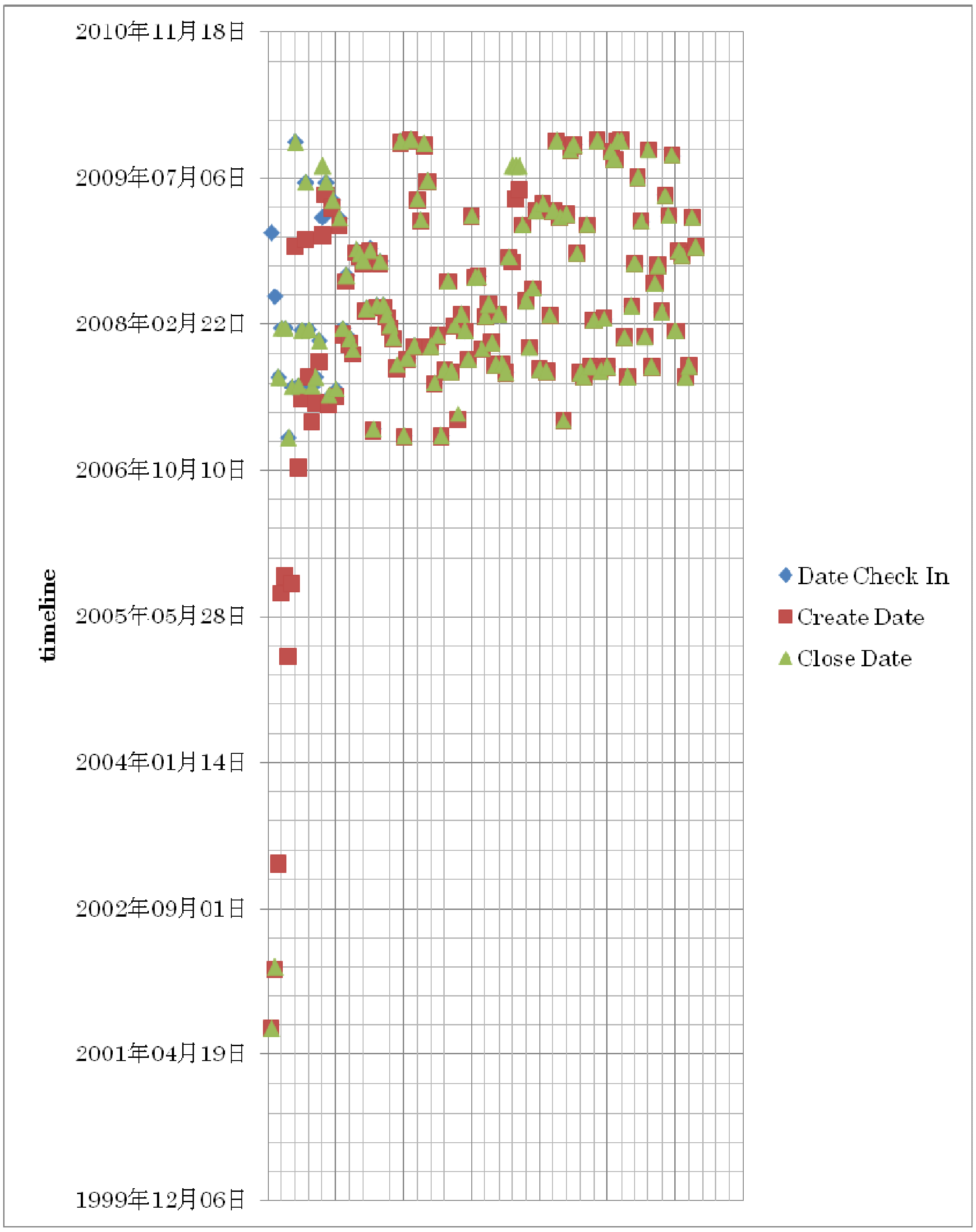


Appendix B -WxWidgets



Appendix C -Filezilla

Bug Fixing Process Sequence Timeline


2010年11月18日
2009年07月06日
2008年02月22日
2006年10月10日
2005年05月28日
2004年01月14日
2002年09月01日
2001年04月19日
1999年12月06日
timeline
◆ Date Check In
■ Create Date
▲ Close Date