Durham E-Theses

AN INVESTIGATION OF COMMON CODING ERRORS IN OPEN SOURCE GRAPHICAL USER INTERFACE CODE

SUN, MIN,JIE

AN INVESTIGATION OF COMMON CODING ERRORS IN OPEN

SOURCE GRAPHICAL USER INTERFACE CODE

MINJIE SUN

Supervisor: Dr Elizabeth Burd and Dr Shamus Smith

A Thesis Submitted for the Degree of

Master of Philosophy

School of Engineering and Computing Sciences

Durham University

April 2011

Abstract

Introduction

This thesis investigates the occurrence of coding errors in the Open Source Software (OSS) Graphical User Interface (GUI) code. Characteristics of coding errors in the OSS GUI code are explored and analyzed so that guidelines are proposed to lower the influence of coding errors in OSS GUI software.

Background

This thesis recognizes the increased prominence of, as well as the increased total volume of OSS GUI code within modern software applications. This thesis seeks to identify whether specific types of errors manifest themselves more frequently in GUI code. The rationale behind this investigation is that: if specific errors are known to occur in specific locations then they can be more easily identified; if specific errors are due to specific causes then they can be more easily recognized in earlier stages of software development.

Methods

Common coding errors were selected and examined in example OSS code using an automatic code inspection. An analysis of results from this inspection identified the frequency and location (i.e. in GUI or non-GUI code) of the common coding error. An initial sample of OSS GUI projects was selected to be examined and a wider range of OSS GUI projects was randomly chosen to evaluate the results that were obtained from the initial sample.

Results

It was found that there are some differences in the type of errors within differing portions of source code. Certain types of coding errors were more frequently

identified in GUI code than non-GUI code and corresponding typical GUI coding error-prone behaviors were summarized.

Discussion

Awareness of these differences helps predict errors during the earlier stages of software development. Comprehension of these GUI coding error-prone behaviours contributes to prevent typical GUI coding errors as much as possible during the whole lifecycle of OSS GUI projects.

Copyright

The copyright of this thesis rests with the author and no quotation from it or information derived from it may be published without the prior written consent of the author.

Acknowledgement

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. I want to thank the Department of Computer Science, Durham University. I am deeply indebted to my supervisor Prof. Elizabeth Burd and Dr. Shamus Smith. I would like to thank Dr. Keith Gallagher, Prof. Malcolm Munro, Mr. Shane Collin and Dr. Yonghong Xiang who help me with correction. Especially, I would like to give my thanks to my wife Lingling Xu whose love encouraged me to complete this thesis.

TABLE OF CONTENTS

TABLE OF TABLES

TABLE OF FIGURES

Chapter 1   Introduction

1.1.   Introduction

The Graphical User Interface (GUI) is an important component in modern software because, increasingly, GUI is the main mode of interaction between software and its users. McMaster and Memon [McMaster2008] identified two reasons for this: modern software comprises a large percentage of GUI code; and GUI errors seriously affect users' the impression of the quality of software. Any investigation of GUI code requires a sufficient amount of code, so the Open Source Software (OSS) code is selected.

An increasingly large proportion of source code is dedicated to GUI implementation. Myers [Myers1995] stated that in general 45% to 60% of all software source code is GUI code. Memon [Memon2002] also stated that GUI code can constitute as much as 60% of an application's total code in modern software. Therefore, an investigation of errors in GUI code becomes important.

GUI errors make up a large proportion of all software errors. Mohapatra and Mohanty [Mohapatra2001] conducted a case study of errors obtained from a live project in INFOSYS Technologies Limited, India. They identified that GUI errors alone contributed 50% of all errors. A further investigation of errors in GUI code will help understand how these errors occur and enable an improvement in the maintenance of GUI code.

As one of the trends of future software development, the OSS is different from conventional software [Mockus2002]. The occurrence of errors in the OSS GUI code is therefore different from the conventional. An appropriate approach is required to investigate errors in the OSS GUI code. Recent research has focused on GUI testing

to detect fault [Xie2006, McMaster2008]. However, GUI testing may not be capable of finding all errors. Myers et al. [Myers2004] stated that "it is impractical, often impossible to find all errors in a program". Software evolution continues and errors keep being injected into programs, so an investigation of errors in a lifetime of software project is valuable as well.

1.2. Motivation and objective

The motivation of this thesis is to investigate errors that are identified in the OSS GUI code during its life time. The identified errors in GUI code are compared to non-GUI so that characteristics are analyzed. Based on the analysis, a valuable understanding of errors in GUI code is provided.

The primary objective of this thesis is to provide experimental analysis for the study of errors in GUI code. The secondary objective is to present evidence and advice for GUI maintenance by investigating errors in GUI code throughout the life cycle of projects.

Complete elimination of GUI coding errors is difficult [Myer2004] so an investigation of coding errors during software lifetime is valuable for software maintenance [Schroter2006]. The investigation of coding errors, however, provides several experimental results and advices including:

1. The types of errors that are frequently identified in GUI code
2. The place where coding errors are identified and the types of software where errors are frequently identified
3. The types of errors that are clustered in GUI code
4. The OSS features that affect the identification of coding errors in GUI code

All the above outcomes benefit the maintenance of GUI code, specifically in OSS

projects.

1.3.   Statement of the problem

Identifying the coding errors in GUI code is the fundamental issue to provide material for analysis. This issue contains three sub-problems:

1.   the range of errors (types of errors for the identification);
2.   the method to identify errors;
3.   the way to analyze errors.

To tackle the three sub-problems, a strategy is proposed. Defining the range of coding errors requires comparing and analyzing existing coding errors. There are several errors identified by existing research. However, there is little evidence of GUI-specific errors [Li2006]. This thesis attempts to investigate general coding errors and explore new types of error in OSS GUI code.

Identifying applicable coding errors is the first step. Existing research has identified many general coding errors [Chou2001, Rutar2004, Hovemeyer2004, Zheng2006]. In this thesis, the most common coding errors are selected to form a set (see Chapter 3). This set is later applied to source code to identify examples from GUI and non-GUI code. Selecting an appropriate OSS project sample set is another important issue. There are thousands of OSS projects of different sizes and written in different languages. The selection focuses on OSS Java project sample which are widely used in the available OSS source. The selection of OSS Java project is determined by the comparison among OSS projects (see Chapters 3).

Errors are identified in the changed GUI code. Reviewing changed code is an efficient way to generate data for analysis [Hovemeyer2004]. The code review is achieved by a pilot manual inspection and followed by an automatic inspection. The automatic inspections will identify examples of error and relevant information for analysis. The

manual inspection only provides guidance and feedback for the implementation of the automatic inspection; nevertheless, new types of coding errors are also explored in the manual inspection.

The investigation of coding errors in GUI code is implemented by comparing coding errors between GUI and non-GUI code. Examples of error and relevant information obtained from the automatic inspection are quantified for analysis. Appropriate statistical methods are chosen to analyze relationships between errors and potential factors, and this provides an outline of how errors occur in GUI code.

1.4. Hypothesis

The criterion of measuring this thesis is that outcomes of this thesis should provide appropriate evidence for the study of errors in GUI code and furthermore for GUI maintenance.

HYPOTHESIS: Coding errors in GUI code are different from non-GUI code; the identification of coding errors shows unique characteristic of OSS GUI code.

The hypothesis can be comprehended by examining different issues including differences of errors between GUI and non-GUI code, prediction of error in GUI and factors that are influential. These issues will be examined after the investigation is completed. The outcome will form the contribution of this thesis. Moreover, future research directions are proposed.

1.5. Thesis structure

This thesis comprises seven chapters including this introductory chapter. The outline of the remainder of this thesis is as follows:

Chapter 2 provides a survey of the literature, including that of software maintenance, OSS, GUI code comprehension, and coding errors.

Chapter 3 describes the approach to identify coding errors, including source code selection, coding error descriptions, and the process of code inspection.

Chapter 4 describes experimental results, including a description of profiles of project samples, and experimental data results from the code inspection.

Chapter 5 analyzes the data obtained from the code inspection. Based on the analysis, frequently identified coding errors in GUI will be found. The results also show where the coding errors are frequently discovered in GUI code. In addition, coding error clusters in GUI are discussed.

Chapter 6 evaluates the results obtained from the automatic inspection. The same approach is applied to another new source code sample in order to evaluate the findings.

Chapter 7 concludes this thesis, identifies the limitations, and proposes future work.

Chapter 2   Literature Review

2.1.   Introduction

GUI is a common feature in most software systems. The increasing use of complex GUI leads to an increase in the number of lines of GUI code in software systems [Memon2002]. This thesis focuses on defects of GUI code, and in this context, the presence of coding errors.

Researchers have identified many coding errors in different programming languages and systems. To identify the representative errors in GUI code, we concentrate on the most common coding errors in this thesis. The literature survey gives relevant theoretical and empirical evidences on coding errors. The design of our approach is based on the literature survey.

This chapter explores the fundamental theoretical and empirical literature, so each section corresponds to issues later addressed in this thesis. The review of OSS provides referential material for the analysis of errors in GUI code during the project life time. The review of GUI and common coding errors provides valuable guidelines for the implementation of automatic code inspection. The review of software measurement and related maintenance technology helps define the method to analyze the obtained results. The corresponding relationship between this chapter and others is shown in Figure 2.1.
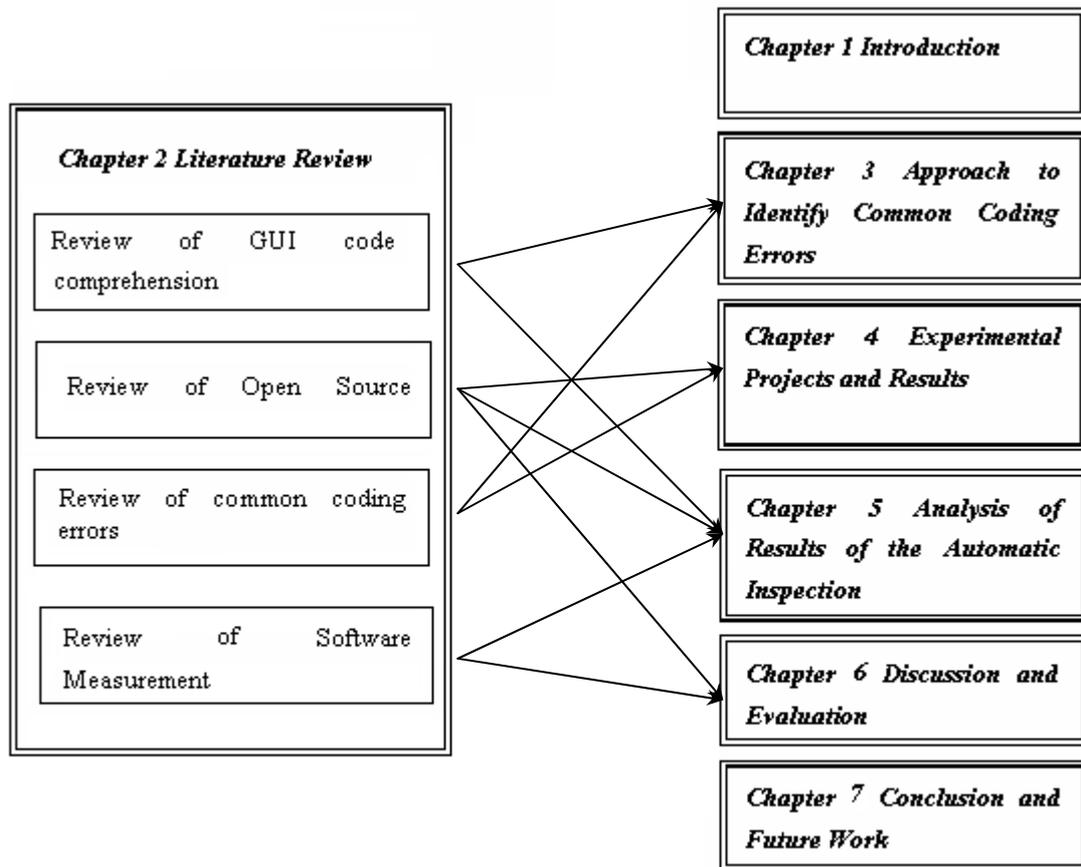
Note: arrow represents literature support

Figure 2.1 Relationship between Chapter 2 and other chapters in this thesis

2.2. GUI Code Discussion

As much as 60% of an application's total code is GUI code [Memon2002]. The comprehension of GUI code is important for the identification of errors in GUI components. This section explores GUI code and discusses relevant code analysis methods.

2.2.1. Definition of Graphical User Interface

The definition of GUI has been updated since it is first proposed in the literature. An updated version of the definition is proposed by Memon et al [Memon2003]. Memon et al. [Memon2003] identifies the following five GUI's core characteristics:

- graphical orientation;

- event driven input;

- hierarchical structure;

- the widgets it contains;

- the attributes (properties) of the widgets.

Based on the core characteristics of GUI, Memon et al. [Memon2003] provided the following definition:

A Graphical User Interface (GUI) is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events, from a fixed set of events and produces deterministic graphical output.

GUI entered the mainstream use in the 1990s when several commercial operating systems successfully adopted GUI, for example, Microsoft Window 95 and Mac operating system. Besides the operating system, GUI components were widely used in thousands of software applications. The increases in the proportion of GUI code therefore increased the total source code population.

2.2.2. Identification of GUI code

GUI code and non-GUI are different in terms of function and structure. Most modern programming languages support GUI applications. Different languages have their own definition for GUI code. To select an applicable GUI source code is import in investigating errors.

Programming language popularity

Identifying the popularity of programming languages is difficult because the usage of a language strongly depends on applications. Some methods of measuring the popularity of languages have been proposed, for example, estimating the number of existing lines of code or the number of source code files. Bieman and Murdock [Bieman2001] conducted a programming language survey based on the World Wide Web. They randomly selected a sample of 23680 URLs, and found 38124 source code

files in various languages such as C, C++, Java, and Perl. They found that Java and C++ are the most widely used Object Oriented languages in the Web and C is the most common used language overall. In addition, comparing primary programming languages of users is also used to measure the popularity of languages. Chen et al. [Chen2005] analyzed the results of responders', students' and companies' primary language and concluded Java and C++ are the most widely used languages.

The OSS development website provides an efficient way to measure the popularity of the three common programming languages (Java, C, and C++). SourceForge.net is considered as the world's largest OSS development website. From the statistical results, there were 84738 projects registered in SourceForge before 25th December 2005 (date when the OSS code was selected in this thesis). 14855 projects were written in Java, 14169 projects were written in C++, and 13961 projects were written in C. The status of the three languages had a small change till 25th June 2008 (review date). Figure 2.2 shows that Java was the most commonly used language in SourceForge. Therefore, Java was selected to be examined in this thesis.
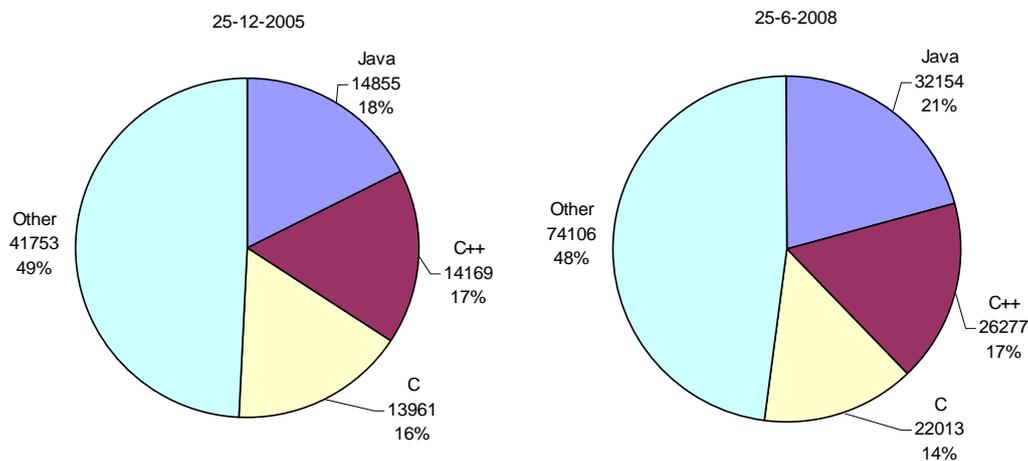
Figure 2.2 Percentage of number of projects for each language

GUI code in Java

There are three main GUI toolkits (Librarys) in Java: Abstract Window Toolkit (AWT), Swing, and Standard Widget Toolkit (SWT). The AWT and Swing are Sun

Microsystems' Java Foundation Classes (JFC), and they are both the standard Application Programming Interface (API) for providing GUI for Java programs [SunMicrosystems2006]. The SWT is a Java class library for creating GUI as part of the Eclipse project; but it is not a JFC [Eclipse2008].

Packages in the three GUI toolkits have unique names at the source code level:

- packages in AWT begin with the prefix `java.awt`;

- packages in SWT begin with `javax.swing`;

- packages in SWT begin with `org.eclipse.swt`.

If any of these prefixes is found in source code, then there exists a GUI application in the corresponding class. However, GUI code may be mixed with non-GUI code if the programmer did not strictly differentiate them. Correspondingly, relevant GUI code analysis methods have been proposed, for instance, Staiger [Staiger2007]. However, the literature shows that studies of coding errors in GUI code are not widely considered.

2.2.3. Relevant research on GUI code

The maintainability of GUI has been considered in recent years [Li2006, Li2008]. The motivation for correction of GUI is to improve quality. Xie [Xie2006] stated that the good quality of GUI code can be considered as one of the assurances of trouble-free software execution. Howell et al. [Howell2003] addressed several issues to assure the correctness of GUI code as follows:

- the layout of widgets;

- the usability of GUI;

- the performance of the overall interactive system.

However, there are few examples of error analysis in GUI code in these studies.

A number of techniques for error detection in software development have been applied to GUI testing. For example, Chen and Subramaniam [Chen2001] proposed specifications for GUI testing but no classification of errors in GUI were made. An

investigation of errors in automatically generated GUI code was delivered by Xie [Xie2006]. Xie [Xie2006] stated that about 25% of GUI events are utilized to manipulate GUI structure, and these GUI events are brought about by the automatically generated GUI code. Due to the unique characteristics of these GUI events, the code for such events normally does not interact with the code of other GUI events. Xie [Xie2006] concluded that automatically generated GUI code creates fewer errors since the execution of these events did not cause many errors for GUI.

According to Xie [Xie2006], errors in GUI code were not classified nor were the comparison of errors between GUI and non-GUI code identified. The literature on GUI correctness and testing shows the existence of errors in GUI code. It is necessary to identify the errors in GUI code for preventive purposes. The study of errors in GUI code requires suitable source code analysis. The following section will introduce two key methods of source code analysis.

2.2.4. Source code analysis

The main source code analysis methods are static source code analysis and dynamic source code analysis. Ball [Ball1999] defined source code analysis and the two methods as follows.

- Source code analysis is the process of extracting information about a program from its source code or artifacts (e.g., from Java byte code or execution traces) generated from the source code using automatic tools.
- Dynamic analysis is the analysis of the properties of a running program.
- Static analysis is the analysis which examines a program's text to derive properties.

Dynamic vs. Static

Dynamic analysis and static analysis are mainly utilized in software testing to detect errors. These two code analysis methods examine one or more versions of a program to identify existing errors or predict potential hazards in a program. The two methods each has advantages and disadvantages in identifying errors.

Since last decade, static analysis tools have been widely utilized to detect system errors in practice. For instance, Engler et al. [Engler2001] provided an empirical evaluation of errors based on their static source code analysis. Moreover, static source code analysis tools have been commonly used to identify errors due to the rapid development of OSS. For example, the *FindBugs* project applied a static source code analysis tool to Java OSS to identify errors [Hovemeyer2004]. In addition, static source code analysis has been used in GUI program comprehension. For example, Rountev et al. [Rountev2004] concluded that static code analysis can be utilized to identify the interaction of GUI programs. Yuan and Memon [Yuan2007] also concluded that static code analysis can be the main analysis technique for error detection in GUI code. Conversely, based on the research outcomes of Rountev et al. [Rountev2004], Yuan and Memon [Yuan2007] concluded that static analysis has limitations in the detection of errors in multi-language GUI implementation.

Dynamic analysis tools are also utilized to detect run-time errors. For example, Sen [Sen2008] proposed a dynamic analysis approach to detect race condition errors. The difference between dynamic and static analyses is that static analysis does not consider input. This difference means that the results obtained from static analysis can lead to much wider application than dynamic analysis. Binkley [Binkley2007] stated that only dynamic analysis results can be correctly applied to the particular input, so results obtained from static analysis are applicable to all executions of the program. Moreover, static analysis has the advantage that results are safer compared to dynamic analysis because no potentially harmful program execution is needed for static analysis [Binkley2007]. Ayewah et al. [Ayewah2007] also stated that static source code analysis can also be used to identify unusual code, such as awkward or dubious computation. Engler et al. [Engler2001] summarized the advantages of static analysis as scalable, precise, and immediate.

Dynamic analysis is considered as complementary to static analysis. Ball [Ball1999] stated that dynamic analysis has advantages in two areas: scope and precision. Static

analysis may be restricted in the scope of program where the method analyzes effectively and efficiently. Dynamic analysis examines the precise path of a program's execution where static analysis identifies all possible paths.

As a result, the combination of static and dynamic analysis is becoming the trend of code analysis approaches. Binkley [Binkley2007] summarized current code analysis approaches and gave a better process as follows: static analysis identifies potential matches of errors while dynamic analysis catches violations precisely as the program runs. Several approaches with a combination of static and dynamic methods were proposed. For example, Heuzeroth et al. [Heuzeroth2003] proposed an approach utilizing both static and dynamic analysis; Tomb et al. [Tomb2007] proposed an analysis approach based on static and dynamic techniques to identify run-time errors in Java code.

2.2.5. Source code inspection

Source code inspection is an effective and efficient method to detect errors. IEEE standard [IEEE610.12-1990] provides a definition as follows:
An inspection is a static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems. Types include code inspection; design inspection.

One of the main purposes of code inspection is to detect coding errors. The code inspection is achieved by code reading technologies. There are several commonly used code reading techniques in practice. For example, Dunsmore et al [Dunsmore2003] summarized three code reading techniques in OO program as:

- *abstraction-driven technique;*
- *use-case technique;*
- *checklist technique.*

The source code inspection process varies for different purposes. Most code inspection of errors during a program's lifetime follows the process:

- selecting project source code;

- gathering relevant information in changes by manual or automatic tools;

- analyzing data from whatever perspectives;

- presenting conclusions.

For example, Zimmermann et al. [Zimmermann2005] conducted a code inspection of code coupling over its lifetime. They utilized an automatic parser tool (ROSE) to gather relevant information of changes, applied a set of coupling rules to extract data, and used an appropriate method to analyze the data. A similar code inspection process was adopted in other research [Graves2000, Ying2004].

Identifying errors in changes is also a widely used method to gather data. Zimmermann et al. [Zimmermann2005] stated the location of errors in an OSS is hidden in version archives, so errors have to be extracted separately. Errors and other relevant information can be extracted from the changes between version archives of code.

Moreover, this thesis investigates errors identified during the lifetime of an OSS. OSS features also need to be considered when choosing a suitable approach. Even though current static and dynamic tools can precisely predict many errors in one or more versions of program, some of these identified errors may not be useful in this thesis. For example, an OSS project may have no participation from users (i.e. zero download count) but existing errors are identified by analysis tools. This result can lead to the neglect of the influence of users in OSS project. Conversely, errors identified in the changed code can provide a better reflection of the participation of users. Additionally, error pattern matching is used to detect errors in the changed code between subsequent versions.

A change, also called a directed delta, is a sequence of change operations that, when

applied to one version 1, yields another version 2 [Conradi1998]. The changes are also called "diffs" in UNIX. Concurrent Versioning System (CVS) is a commonly used version control tool to store changes between sequential data rather than complete files in the OSS code. For example, CVS is the most commonly used version control tool in SourceForge. This thesis will identify errors in diffs of the OSS code.

Extracting data from the change requires much work, so accurate automatic code inspection is a research goal. Binkley [Binkley2007] stated that the automated and semi-automated analysis of source code had remained a topic of intense research for more than thirty years. The request for automation is due to the large size of source code. In fact, about 120 billion lines of source code were maintained in 1990 [Ulrich1990], while 250 billion lines of source code were maintained in 2000 [Sommerville2000]. The automation of error detection in delta is also an adopted approach in this thesis.

2.3. Open Source Software

The purpose of investigating errors in GUI requires a large portion of source code. The OSS software is appropriate for this thesis because of their cost, traceability of code and so on. Understanding an OSS helps to identify relevant factors that are influential for coding errors in OSS GUI.

2.3.1. Open Source Software

The free software movement was launched in 1983. This movement emerged as a reaching against expensive proprietary software. The term Open Source Software "OSS" was first used in 1998 [Feller2002]. During the last decade, many OSS products have been produced and utilized, such as the Apache server, the Linux operating system and Mozilla [Feller2002, Mockus2002]. The research on OSS

maintenance is constructive for understanding the occurrence of errors. Several works on software maintenance are now based on OSS.

The core characteristics of OSS can be identified from its name, 'open'. Compare to closed source proprietary software, OSS source code is open to all users to use, modify and maintain [Feller2002]. The definition of OSS is often dependent upon its license. For example, the biggest open source development website SourceForge provides ten criteria to classify OSS. Seven of these characteristics are legal in nature, and lie outside the remit of this thesis. We are most interested in the other three characteristics:

- *free redistribution:* there should be no charge for redistribution, including OSS as software or as a component;
- *source code:* the code must be provided with programs;
- *derived works:* modifications and derived works must be allowed and they must be able to be distributed.

During the last five years, many OSS products have been widely applied in different domains, such as education [Corbesero2006] and health care [Goulde2006]. The increasing usage of OSS means the quality of OSS has become more important than ever before. For example, a single fault (i.e. from low quality of OSS) in the health care system can lead to serious consequences [Goulde2006]. Research has shown the importance of identifying key elements that influence the quality of OSS. The next section explores OSS by comparing it with proprietary software.

2.3.2. OSS vs. proprietary software

With rapid development of OSS across different domains, OSS has been widely studied. The quality of OSS is one of the widely considered topics. For instance, Raymond [Raymond1999] and Biffi et al. [Biffi2001] concentrated on OSS quality and error proneness. Many works on OSS had been carried out on small individual

OSS projects, OSS development in general and even on the community of OSS users [Capiluppi2002, Koponen2005]. In addition, the research on OSS maintenance shows how errors occurred in OSS. An overview of OSS maintenance is valuable for this thesis since it outlines a picture of errors during the life time of OSS project.

Raymond [Raymond1999] stated that errors can be corrected, and that the quality of OSS is dependent on the effort of many experienced OSS developers and maintainers. Raymond [Raymond1999] believed that OSS is more reliable while compared to proprietary software, because code is reviewed by many programmers. Koponen and Hotti [Koponen2005] investigated software maintenance activities between proprietary software and OSS. They identified that there is no retirement activity (one of the six maintenance process activities defined by ISO/IEC 15288 [ISO/IEC15288-2002]) in the maintenance of two OSS projects: Apache and Mozila, and therefore they suggested that OSS may last longer than proprietary software. In addition, the maintainability of OSS is believed to be much better than proprietary software because errors are identified and fixed in a very short period. For instance, Raymond [Raymond1999] argued that many people contribute to identify errors so errors are quickly fixed in OSS. Raymond called this 'Linus's Law'.

However, the outcome of Raymond's research [Raymond1999] is not supported by either empirical evidence or later studies [Koru2007]. According to Koru et al. [Koru2007], 55% of developers stated that compared with OSS, the peer-review in closed source software (proprietary software) is more organized.

Characteristics of OSS	Characteristics of Proprietary Software
Developed by many volunteers	Developed by limited number of people employed by the proprietor
No assignment to volunteer for a specific OSS	Strict assignment to specific software project by employers for proprietary software
No explicit system design	Accurate system and detail design
No project plan, schedule, list or deadline	Explicit plan and deadline

Source: Mockus and Herbsler (2002)

Table 2.1 Summary of the differences between OSS and proprietary software

The maintenance process of proprietary software is often undertaken by a stable team, and maintenance tasks are scheduled. Conversely, the maintenance of OSS is undertaken by many people, and there is no formal schedule or traceability for OSS maintenance.

Mockus and Herbsler [Mockus2002] conducted an investigation of comparing OSS and proprietary software, and summarized the difference between them (see Table 2.1).

Correspondingly, there are few strict maintenance processes for OSS. Godfrey and Tu [Godfrey2000] stated that evolutionary project development is one of the main characteristics of OSS. Thus, OSS is concurrently and continuously performing maintenance (such as error-fixing and functionality enhancement) while proprietary software is not [Koru2007]. The difference between OSS and proprietary software maintenance drives researchers to propose new approaches to measure OSS.

One of the common concerns between proprietary software and OSS is the impact of software size on error proneness. Koru et al. [Koru2007] addressed the importance of monitoring software size and its impact on error proneness. Emam et al. [Emam2002] stated a general agreement that software size is positively associated with error proneness. Besides the common factor of size for both OSS and proprietary software, other OSS characteristics also have impacts on the errors, so the measurements need to be understood.

2.3.3. Measuring OSS

The occurrence of errors in OSS is influenced by many factors. Samoladas et al. [Samoladas2004] stated the measurement of source code help assess its maintainability, reliability, extensibility, and portability.

The line of code (LOC) is the most common metric used to measure the size of both OSS and proprietary software. Samoladas et al. [Samoladas2004] defined LOC as one of the typical measurements for source code: *number of lines of code (LOC) measures the physical size of the program code, excluding blank lines and comments*.

Software size is a common feature used to represent software. LOC is therefore commonly used to identify the correlation between software size and other features of software. Stewart et al. [Stewart2005], Yu [Yu2006] and Koru et al. [Koru2007] all utilized source code size as measurements (i.e. number of LOC) of its key software metrics. Stewart et al. [Stewart2005] tried to identify the coupling and cohesion relationship between classes of OSS projects by considering LOC; they found that there is a significant correlation between LOC and coupling (p<0.5 in all samples) in both C++ and Java programs, but the relationship between the cohesion and ratio of comments to code is not significant.

Yu [Yu2006] used LOC to establish a maintenance effort model. Yu found that maintenance effort can be influenced by the LOC and the number of operators changed. Yu [Yu2006] also identified a positive linear correlation between the maintenance effort and the number of lines changed. Similar to Yu's research outcomes, Koru et al. [Koru2007] identified a linear relationship between the number of LOC and errors. This assumption was demonstrated by an effort model, and Koru et al. [Koru2007] concluded that there is a significant relationship between the number of LOC and error proneness. Even though the correlation does not ensure a causal relationship, LOC can be used to indicate trends in the occurrence of errors.

In addition, other features of OSS are proposed to measure OSS. For example, Crowston et al. [Crowston2003] identified the following set of metrics:
- measuring output
  - movement from alpha to beta to stable versions;

> ➢ achieved identified goals;

> ➢ developer satisfaction.

- measuring development

> ➢ number of developers;

> ➢ level of activity;

> ➢ time between releases;

> ➢ time to close errors or implement features.

- measuring developers

> ➢ individual job opportunities and salary;

> ➢ individual reputation;

> ➢ knowledge creation.

The first three features measure output of OSS projects. Features 4~7 measure the process of OSS project development, and the last three metrics measure the outcome for OSS project members. In order to investigate the impact of the ten criteria, Crowston et al. [Crowston2006] applied this set of metrics to OSS projects obtained from SourceForge. According to Crowston et al. [Crowston2006], the number of errors correlates to not only the number of developers but also the project activity index. The download count contributes to the index of activity.

In summary, several source code features have been discussed, such as development team size, error fixing time, source code size and so on. The literature shows that the LOC metric is commonly utilized to measure the source code [Samoladas2004, Stewart2005, Yu2006, and Koru2007]. The source code comprehension occupies a large amount of software maintenance time [Livadas1994]. A good source code comprehension model is required in order to apply these measurements to OSS. The next section focuses on GUI code comprehension and the source code analysis.

2.4. Coding errors

Coding errors are commonly created by both novices and experts. Many approaches have been proposed to detect, understand and prevent coding errors. There are a large number of coding errors and the effect of each coding error is different. Common coding errors overall have a more significant effect on source code with a comparison to uncommon errors. However, little evidence has been found to classify and differentiate the errors between GUI and non-GUI code, and thus common coding errors are selected to be examined in this thesis. Additionally, different types of coding errors often clustered together. Samoladas et al. [Samoladas2004] stated that coding errors are not equally generated by each component in program, for instance almost 80% of problems are generated by 20% of components. Therefore, investigation of errors clustering is also necessary to identify problematic source code in program.

2.4.1. Definition of error

The word "error" is frequently used in computer science. Additionally, similar words are also used to describe system or program anomalies, such as words "fault", "failure", and "mistake". These thesauruses may refer to different objects in computer science.

IEEE standard [IEEE610.12-1990] provides a set of definition for "error", "fault", "failure", and "mistake".

- **error**: the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition;
- **fault**: an incorrect step, process, or data definition;
- **failure**: an incorrect result;
- **mistake**: a human action that produces an incorrect result.

Based on these definitions, the relationship between them is that "mistake" causes

"error", "error" leads to "fault", and "fault" may result in "failure". Similar definitions for these terminologies are used in other literature, such as [Shittu2007].

Based on the definition provided by IEEE standard [IEEE610.12-1990], the meaning of coding error can be defined as: errors occurred at source code level while coding. This thesis only focuses on errors which occur at source code level. Additionally, the coding error can be defined as: code idioms that are likely to be errors [Cole2006].

2.4.2. Classification of coding errors

Coding errors are not completely avoidable by experts or novices. Evidence shows a large number of errors were identified. For example, Phipps [Phipps1999] conducts a survey of errors in C++ and Java code, and finds that the average number of errors per thousand lines of code is 18 for C++ and 6 for Java.

Coding errors are caused by many different reasons. However, human mistakes can be considered as one of the main reasons. Basili et al. [Basili1996] summarized five human mistakes made by programmers: omission, incorrect fact, inconsistency, ambiguity, and extraneous reasons. This categorization of mistakes mainly considers human factors but not objective factors such as language. Hovemeyer and Pugh [Hovemeyer2004] categorized mistakes into three groups considering languages and environments as follows:

- *mistakes made by programmers*, no matter how experienced a programmer is, he/she can make mistakes;
- *mistakes due to language features*, for example, Java provides opportunities for latent errors, such as forgetting Java is zero- indexed;
- *mistakes due to algorithm complexity*, especially in multi-threaded programming.

The cause of errors varies between different environments or languages where each

mistake may lead to different types of errors in source code. Specifically, the three main categories of mistakes lead to various categories of errors. Qin et al. [Qin2007] summarized three categories of errors in the execution environment:

- *memory management based errors*: this type of error is caused by programmers who make mistakes on managing memory allocation, such as *variable overflow;*

- *timing based errors*: the error is caused by multi-threaded programming mistakes, such as a shared variable being accessed by different methods;

- *user request-based errors*: This error is caused by mistakes without considering unexpected user request, for instance, buffer overflow during stack smashing attack.

The three categories of errors from the execution environment have their roots in programmers' mistake or neglect. In addition, Vipindeep and Jalote [Vipindeep2005] provided a categorization of common coding errors. They summarized five main categories for coding errors as: memory, synchronization, data and algorithm, data comparison, and redundant code. Furthermore, Smith [Smith1999] stated that the naming error is also one of the main categories of coding errors. Therefore, categorization of coding errors can be summarized as follows:

- *memory related errors;*
- *synchronization errors;*
- *data and algorithm errors;*
- *redundant code errors;*
- *data comparison errors;*
- *naming errors.*

2.4.3. Common coding errors

Several common coding errors rankings were produced based on experiential results. For example, Reilly [Reilly2008] provided top ten coding errors in Java code, and

Robinson [Robinson2004] provided top ten coding errors in C++ code. In addition, the literature has shown that the occurrences of common coding errors were compared and ranks were produced; for instance, Chou et al. [Chou2001] presented a rank of coding errors in the Linux code, and Zheng et al. [Zheng2006] presented a rank of coding errors in C++ code.

Evidence shows that the majority of software faults are caused by several coding errors. Chou et al. [Chou2001] conducted a survey on the Linux operating system and OpenBSD kernel and gave nine most common coding errors in the Linux operating system (see Table 2.2).

Error description	Number of errors	Percentage of each type of error
Null pointer	460	44.9%
Code block	293	28.6%
Index out of bound	181	17.7%
Interrupts	27	2.6%
Lock	26	2.5%
Freed memory error	17	1.6%
Memory leak	11	1.1%
User pointer dereference	7	0.7%
Inappropriate variable type	3	0.3%
Total	1025	100%

Source: Chou et al., [Chou2001].

Table 2.2 Errors identified in the Linux operating system.

According to Chou et al. [Chou2001], the most common errors in the Linux operating system code are errors *null pointer*, *code block*, and *index out of bound*. These three types of coding errors caused more than 90% faults in the Linux operating system. Similar results were also obtained by other researchers.

Zheng et al. [Zheng2006] conducted a survey of faults on three large-scale network service products written in C/C++. After examining more than three million lines of code, they found that about 90 percent of total faults identified by automatic static

analysis tools were caused by 10 types of coding errors. The most common 10 coding errors are shown in Table 2.3.

Error description	Percentage of all errors
Possible use of null pointer	45.92%
Possible access out-of-bounds	10.13%
Pointer is not freed or returned	8.11%
Memory leak	7.46%
Variable not initialized	5.64%
Inappropriate deallocation	3.41%
Suspicious use of ";"	2.47%
Control flows into case/default	2.42%
Type mismatch	2.18%
Data overrun	2.13%

Source: Zheng et al., [Zheng2006]

Table 2.3 90% errors identified by the automatic static analysis

Table 2.3 shows that *null pointer*, *out-of-bounds*, and *pointer is not freed or returned* are the most common coding errors. The three type of coding errors caused more than 60% of the errors overall. The error *code block* identified by Chou et al. [Chou2001] (see Table 2.2) is not listed in Table 2.3. Comparing results between the two researchers, it is shown that the category of memory error including *null pointer* and *index out of bounds* is the most common category of coding errors whatever the influence of environments or languages.

Other researchers summarized common coding error but they did not investigate the occurrence of each error in source code. For example, Robinson [Robinson2004] summarized the top ten coding errors in C++. He placed the ten errors in order, but he did not provide evidence to support his ranking. Robinson's top ten errors are shown in Table 2.4.

Error description
Invalid memory access error.
Off-by-one

Uninitialization
Variable type error
Loop errors
Incorrect code blocking
Returning a pointer or a reference to a local variable
Problems with new and delete
Inadequate checking of input data
Different modules interpret shared items differently

Source: Robinson, [Robinson2004]

Table 2.4 Top ten programming errors in C++ code

Table 2.4 shows that the most common coding error is *invalid memory access*, the error *null pointer* can be considered as an invalid memory access error. The second common coding error *off-by-one* is similar to the error *index out of bounds*. The error *code blocking* (see Table 2.2) is not addressed in the top three errors. Instead, the *uninitialization* error is identified as the third common error. Without empirical evidence, it is hard to justify how Robinson [Robinson2004] ranked the top ten errors. However, the top ten coding errors from Robinson [Robinson2004] show similar results to Chou et al. [Chou2001] and Zheng et al. [Zheng2006].

Several researchers have investigated common coding errors in source code written in Java [Reilly2000, Rutar2004, and Hovemeyer2004]. For instance, Reilly [Reilly2000] summarized the top ten coding errors in Java code. He ranked the ten errors but there was no data to support this ranking. Reilly's top ten errors are shown in Table 2.5.

Table 2.5 shows that Reilly's most common coding error is the *null pointer* error. It shows that the error *preventing concurrent access to shared variables by threads* (multi-threaded/concurrent programming error) is also one of the top three coding errors. However, the error *forgetting Java is zero-indexed* is in fourth position in Table 2.5. This type of error is very similar to the error *index out of bound* (see Table 2.2). Similarly, Hovemeyer and Pugh [Hovemeyer2004] identified a coding error framework that includes 19 types of error for Java programs. They did not provide

occurrence data for each error. So, it is hard to justify how they rank these errors.

Error description
Null pointer
Capitalization error
Preventing concurrent access to shared variables by threads
Forgetting that Java is zero-indexed
Writing blank exception handlers
Confusion over passing by value, and passing by reference
Comparing two objects ("==" instead of ".equals()")
Comparison assignment ("=" instead of "==")
Mistyping the name of a method when overriding
Accessing non-static member variables from static methods

Source: Reilly [Reilly2000].

Table 2.5 Top ten coding errors in Java code

In addition, Reilly [Reilly2000] considered *capitalization error* and *mistyping the name of a method when overriding* as two of the top ten common coding errors. These two types of errors can be classified to naming category errors. Specifically, the *capitalization error* is a typical Java coding error. Naming methods follow certain rules in Java, that is, every new word starts with a capital letter in a method name.

Rutar et al. [Rutar2004] conducted an investigation to compare four different debugging tools for Java. Based on the occurrence of each error identified by the four debugging tools, they rank the top 12 coding errors in Java (see Table 2.6).

Table 2.6 shows similar results to Reilly's top ten errors in Table 2.5. The first common coding error is *null pointer*, and the second error is *concurrency possible deadlock*. The error *array length may be less than zero* is ranked the fourth.

Error description
Null dereference
Concurrency possible deadlock
Exception
Array length may be less than zero
Mathematics division by zero
Unreachable code due to constant guard for conditional loop
String checking equality using == or! =
Equal objects must have equal hashcode
Stream not closed on all paths
Unused local variable
Should be a static inner class
Unnecessary statement

Source: Rutar et al. [Rutar2004]

Table 2.6 Top 12 coding errors in Java code

In the above coding errors, many names are simply different labels for very similar coding errors, for example, errors *off-by-one*, *index out of bound*, *loop error*, *Java is zero-index*, and *array length may be less than zero*. These five errors mainly represent the error *array index off-by-one* in Java because they are caused by the incorrect index value in a loop.

The error *accessing non-static member variables from static methods* and *should be static inner class* represent the same error. The error *incorrect code blocking* is the consequence of *multi-threading* error. As a result, this type of coding error can be represented by the error *concurrent access shared variable by thread*. The error *pointer is not freed or returned* and *memory leak* are similar because the former is the main cause of the latter. The errors *inappropriate deallocation* and *problem with "new" and "delete"* represent the same type of error. The errors *unused local variable* and *unnecessary statement* can be renamed as *dead code*. The coding errors decrease to 24 types after combining these reduplicate ones.

Error name	Mentioned by	Number of Researcher	Identified in
Array index off-by-one	Ch, Zh, Rb, Rl, Rt	5	Linux, C++, Java
Null pointer	Ch, Zh, Rl, Rt	4	Linux, C++, Java
Preventing concurrent access to shared variables by threads	Ch, Rb, Rl, Rt	4	Linux, C++, Java
Uninitialization	Zh, Rb	2	C++, Java
Comparing two objects ("==" instead of ".equals()")	Rl, Rt	2	Java
Comparing two assignments ("=" instead of "==")	Rl, Rt	2	Java
Pointer is not freed or returned	Ch, Zh	2	Linux, C++
Variable type mismatch	Zh, Rb	2	Linux, C++, Java
Writing blank exception handlers	Rl, Rt	2	Java
Accessing non-static member variables from static methods	Rl, Rt	2	Java
problem with "new" and "delete"	Zh, Rb	2	C++
Returning a pointer or a reference to a local variable	Rb	1	C++
Stream not closed on all paths	Rt	1	Java
Invalid memory access error	Rb	1	C++
User pointer dereference	Ch	1	Linux
Data overrun	Zh	1	C++
Dead code	Rt	1	C++, Java
Control flows into case/default	Zh	1	C++
Suspicious use of ";"	Zh	1	C++
Inadequate checking of input data	Rb	1	C++
Confusion over passing by value, and passing by reference	Rl	1	C++, Java
Capitalization error	Rl	1	Java
Mistyping the name of a method when overriding.	Rl	1	Java
Mathematics division by zero	Rt	1	C++, Java

Ch: Chou et al. [Chou2001], Rb: Robinson [Robinson2004], Rl: Reilly [Reilly2000], Rt: Rutar et al. [Rutar2004], Zh: Zheng et al. [Zheng2006]

Table 2.7 Summary of errors identified by researchers

In Table 2.7, we list the coding errors by a column "Number of Researchers", which is the number of analyses that identified certain type of coding error. This is different from the ranks of the most common coding errors because of variations in environment and languages. By comparing the results from C++, Linux and Java, it is

shown that the most common coding error types are *null pointer*, *concurrent access shared variable by thread*, and *array index off-by-one*. Some types of coding errors may not be applicable especially in our selected open source Java code. A description of how to choose appropriate types of coding errors is provided in next chapter.

2.5. Software Metrics

An appropriate analysis method is capable to reflect the reality of software. Designing an analysis approach is important for exploring occurrence of errors in OSS GUI code. Software measurement is widely considered by both academia and industry, and many valuable contributions are made in literature. Kichenham and Charters [Kitchenham2007] stated that "Evidence-based software engineering (EBSE) relies on empirical software engineering research but emphasizes the need to find and aggregate best available evidence on a specific topic and uses secondary studies such as systematic literature reviews and mapping studies as the methodological framework for finding and aggregating evidence."

Kitchenham [Kitchenham2010] compared and summarized 25 most valuable papers that contribute to the evaluation of software. These works attempt to design appropriate metrics to measure the relationship between certain software characteristics and potential causes; moreover, 13 out of 25 works are related to software fault. A summary of analysis methods [Kitchenham2010] for the works that are software fault related are as follows:

- Correlation;
- Logistic regression;
- Multiple logistic regression;
- Principal component analysis;
- Multiple regression;
- Multivariable logistic regression;

- Simulation study;

- Pareto-plots; scatter plots;

- Stepwise regression.

The main analysis methods to identify the relationship between software metrics and fault-related are correlation and regression. Moreover, a chart is also used to visually detect the relationship such as scatter plots [Fenton 2000].

Kitchenham [Kitchenham2010] also summarized the software metrics used by the 13 works. Various metrics are selected according to the relationship predictions made by different works. These metrics can be classified as follows:

- CK metrics (WMC, DIT etc.);

- Size-related (LOC, file size etc.);

- Change-related metrics (such as weighted change, number of changes, deleted and so on);

- Age-related metrics (age of code).

The CK metrics [Chidamber1994] are widely used in Object Oriented design, so they are programming-related. The CK metrics include the following aspects:

- Weighted Methods per Class (WMC);

- Depth of Inheritance Tree (DIT);

- Number Of Children (NOC);

- Coupling Between Objects (CBO);

- Response For a Class (RFC);

- Lack of Cohesion in Methods (LCOM).

Moreover, the prediction of faults by using the CK metrics is criticized by Kitchenham [Kitchenham2010]. Kitchenham pointed out that design-based metrics are more stable than code in evolving systems. Therefore, for fault prediction, OO metrics (such as CK metrics) is not as useful as system evolves and code change metrics. Data gathering is also summarized by Barbara Kitchenham [Kitchenham

2010]. According to different predications and requirements, various methods of data gathering are chosen. The observational study is the main method in gathering information for analysis in the 13 works.

2.6.   Chapter summary

In this chapter, the literature on coding errors was investigated and summarized for further selection in the next chapter. The discussion of code comprehension and source code analysis showed that OSS Java projects were the most commonly used language in SourceForge; meanwhile, static code analysis was selected to identify coding errors in the changed code. This chapter also discussed OSS project characteristics. According to the advantages of OSS projects, the source code is intended to be examined in this thesis. Since the OSS project is evolving during its life time, the investigation will review errors throughout the whole life of OSS project. An approach will be implemented to examine the OSS Java code. 24 types of coding errors were summarized base on the literature, and the most representative Java errors will be selected for further examination. Moreover, the literature on software metrics provides options for the later data analysis. The next chapter will describe the approach to examine these coding errors.

Chapter 3   Approach to Identify Common Coding Errors

3.1.   Introduction

Source code measurement, code analysis, and coding errors have been discussed in Chapter 2. Based on the discussion of the literature, a suitable Approach to Identifing Common Coding Errors (AICCE) is proposed. This chapter will describe the process of the AICCE.

The objective of using the AICCE is to provide coding error data for further investigation. After applying the AICCE to the selected source code, examples of coding errors will be quantified for further statistical analysis. More specifically, the AICCE contains five steps and one database as follows:

> step 1: selection of source code
>
> step 2: manual inspection
>
> step 3: generation of the Common Coding Error Set (CCES)
>
> step 4: automatic inspection
>
> step 5: a database to store all examples identified by both manual and automatic inspection

The process of the AICCE is shown in Figure 3.1. The first step selects open source Java projects provided by the SourceForge for coding errors identification. Step 2 identifies the coding errors (from Table 2.7) and seeks new types of coding errors in the selected source code. Step 3 assesses the results from Step 2 and refines coding errors to establish the CCES. Step 4 implements the CCES to the source code and identifies examples of the CCES. The detail of each step and database is described in the remainder of this chapter.
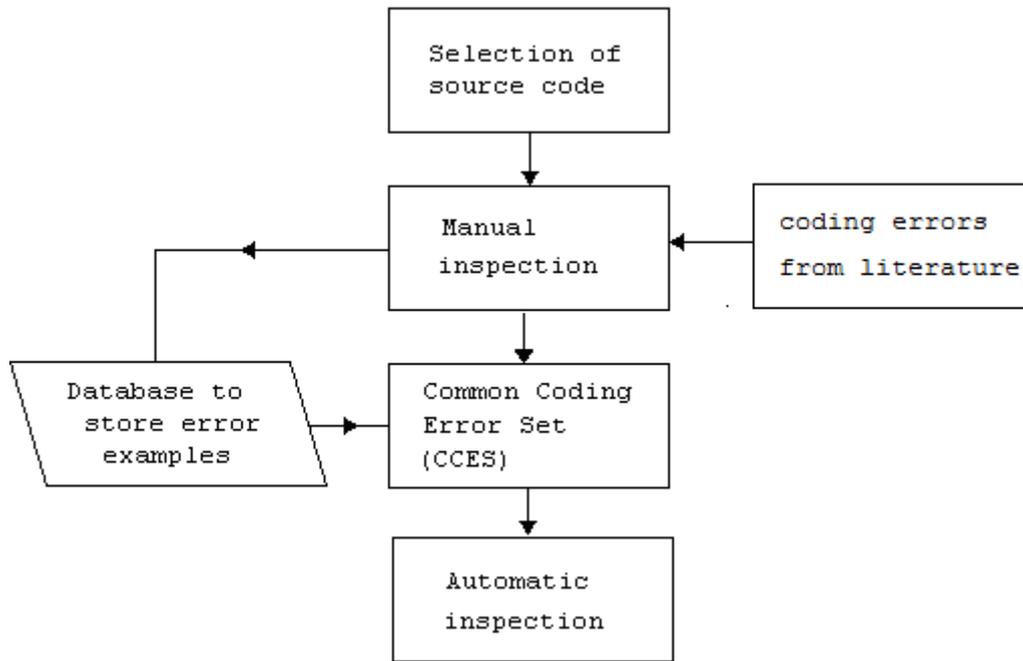
Figure 3.1 Process diagram of the Approach to Identify Common Coding Errors (AICCE)

3.2. Selection of source code

The source code was selected from SourceForge where a large number of OSS projects were registered. The statistical results provided by the SourceForge show that there were 84738 registered projects and 14855 projects were developed in Java before 25th December 2005.

3.2.1. Project sample for the manual inspection

30 projects, which their source code is web-view available, were selected for the manual inspection from 14855 Java projects before 25th December 2005. No particular method was applied to select projects as they were all chosen without applying any additional criteria. SourceForge classifies projects into seven types based on their user interface types:

- Graphical User Interface;
- Grouping and Descriptive User Interface;
- Non-interactive (Daemon) User Interface;

- Plugins;

- Textual User Interface;

- Toolkits/libraries;

- Web-based User Interface.

Based on the above user interface categories, five out of the 30 projects do not have GUI but other types of user interface, such as web-based user interface. Therefore, the results from these five projects were excluded for further investigation, and the results of the 25 projects were stored for further analysis. In addition, the category of GUI has many sub-categories, such as Java AWT, Java Swing and Java SWT. Some projects use more than one GUI package (see Figure 3.2). Figure 3.2 shows that there are 2351 Java SWT, AWT and Swing projects in the 14855 projects. The 25 projects are a sample of the population of 2351.
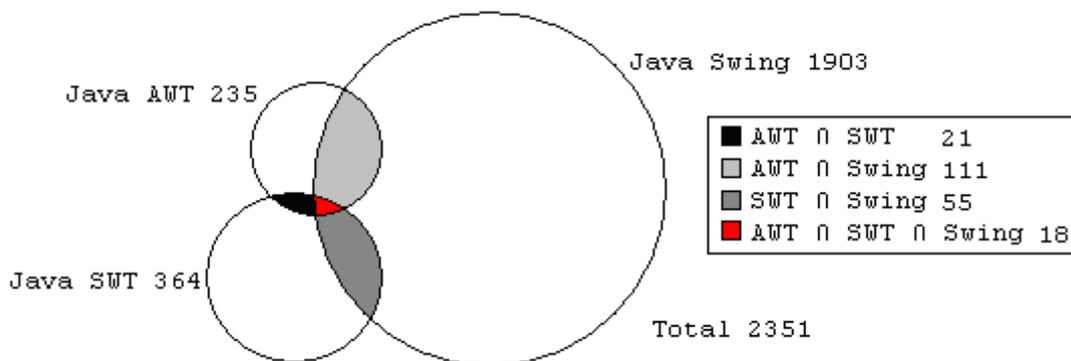
Figure 3.2 Intersections of Java AWT, SWT, Swing projects

The differences between sequential versions for each selected projects are extracted for review. Most OSS projects have a series of different versions; additionally, each single file has even larger number of versions in an OSS project. Reviewing the code that has been changed between two sequential versions narrows the scope to explore errors. As a result, the selected errors are identified in the changes between sequential versions of source code for each 25 projects.

3.3. Coding errors in manual inspection

24 types of coding errors are shown in Table 2.7. Several errors are not suitable for this thesis since they are specific to certain language, such as C++. Coding errors which are applicable to Java are selected because OSS Java projects are selected to be investigated in this thesis.

There are 15 types of common coding errors were identified in Java code (see Table 2.7), hence, the manual inspection attempted to identify the 15 coding errors in the 25 projects. Several static code analysis tools provide error patterns of these 15 types of errors. For example, Maryland University developed a static analysis tool FindBugs [Findbug 2009]. This analysis tool scans source code and identifies potential problematic code. 388 different specific type errors are defined and implemented in this program. However, this thesis focuses on the most frequently identified errors during the lifecycle of OSS project; therefore, reviewing the differences between sequential versions of code is selected for its applicability.

Many static analysis tools have been developed in recent years. The FindBugs is one of the widely used tools to identify potential errors in Java code. This tool provides a set of Java error descriptions. So, some coding problem descriptions are valuable for helping define the frequent errors in this thesis. Nevertheless, the FindBugs reviews the whole version of code instead of focusing on changes. Our actual methodology to identify errors is different from the FindBugs and other similar code analysis tools. Identifying errors in each change due to error correction is the basic principle in this thesis. According to literature and relevant existing analysis tools, the descriptions of 15 types of errors are discussed as follows:

Error 1 *"=" instead of "=="*
The equal mark *"="* represents the equality of two operands in mathematics. The double equal mark *"=="* is a relational operator to represent equivalence. The error

"=" *instead of* *"=="* is a typical coding error in many languages, such as C and Java [Shannon1996]. This error often occurs when two values are compared.

The *"="* is commonly used to represent the equality of the left hand side and the right hand side in mathematical expressions, but Java recognizes *"="* as assigning values from the right hand side to the left hand side. The *"=="* is the correct expression for the comparison of two assignments in Java.

Little evidence is found to provide referential method to identify this type of error. However, the manual inspection identified some examples in the selected source code. From the feedback of the manual walkthrough, the general appearance of this type of error is as follows: an expression of x = y (x and y represent identifier names) is found in the original code, and the expression of x == y is found in a changed subsequent version of code.

Error 2 *"=="* *instead of* *".equals()"*

The error *"=="* *instead of* *".equals()"* is a common coding error in the Java language when comparing two objects. Java recognizes *".equals()"* as the expression to compare two objects, and recognizes *"=="* for comparing two variables. Equal objects must have equal hashcode. When comparing two objects, the hashcode of two objects is compared. However, many people may ignore what they are trying to compare. Some objects may confuse programmers, such as the *String*.

This type of error has been specified in the existing tool. For instance, FindBugs [Findbug 2009] described two subtypes of this error:

- Comparison of String parameter using == or *! =* ;
- Comparison of String objects using == or *! =* .

The result of the manual inspection successfully identified three examples in the source code, and the appearance of error is as follows: an expression of x == y is

found in the original code, and the expression of `x.equals(y)` is found in the changed subsequent version of code.

Error 3 *capitalization*

The method naming error is very common in case sensitive programming languages, such as C++ and Java. A typical mistake made is that some method name is not following method naming rules.

Many languages have rules to name methods and variables. A meaningful name should be given to a method in languages such as C++ and Java. Good practice in these languages is that the method name starts with a lowercase letter and the first letter of every new word is capitalized; the variables names should be meaningful and start with a lowercase letter [Gosling2000]. Unfortunately, programmers may mistype the character in upper case, especially when the name is long.

The FindBugs [Findbug 2009] describes three subtype of this type of error. They are:

- confusing method names;
- method names should start with a lower case letter;
- very confusing method names.

Additionally, the type of error is identified in the source code by the manual inspection. The following example explains the style of this error: a method name `xyz()`(xyz represents any valid method name) is found in the original code and the name with a case changed `xYz()` is found in a subsequent version of code.

Error 4 *variable type*

The error '*variable type*' is common in many languages. Programmers may choose an inappropriate type for the variable. For example, the actual requirement for a variable type should be '*long*', but programmers selected the type '*int*'. This will cause an overflow when the value is greater than the scope of an *int*. This type of error can be caused by an incorrect understanding of variable scope. The inappropriate variable

type may lead to incorrect results. If the type of variable cannot hold the value, data overflow will occur.

Little evidence of error description has been found for this type of error in the existing analysis tools, such as FindBugs; however, the manual inspection identified one example of this error. The description of this error is as follows: A smaller range variable type is identified in the original code and a greater range variable type is identified in a subsequent version of code. For example, `int x` (x represents a variable name) is identified in the original code and `long x` is identified in a subsequent version of code.

Error 5 *uninitialization*

The error '*uninitialization*' is a common coding error for many languages such as Java and C++. Variables need to be initialized before they are used by methods. In addition, the FindBugs [Findbug 2009] classifies this type error into three subtypes as follows:

- Method defines a variable that obscures a field;
- Uninitialized read of field in constructor;
- Uninitialized read of field method called from constructor of superclass.

The manual inspection attempts to identify the possible examples for the above errors; nevertheless, the three subtypes of error described by FindBugs are not identified by the manual inspection. The main reason is that the identification of the three subtypes of error requires all relevant code to be analyzed instead of code fragment. Unlike other static analysis tool, the manual inspection only examines the comparison between sequential versions of code so the three types of error are not identified. The manual inspection identified five examples of *uninitialized* variable in source code. The trade-off between the applicability and completeness therefore selects this type of error as follows: An *uninitialized* variable x is identified in the original code and x is assigned a value in a subsequent version of code.

Error 6 *null pointer*

The error '*null pointer*' is considered the most common coding error in Java [Reilly2000], and it is also identified as common in many other languages (see section 2.5.2). This type of error will cause a program interruption because `null` is processed by other statements. Many applicable methods to identify this type of error are identified in literature. For instance, the FindBugs [Findbug2009] describes eight specific subtypes of errors as follows:

- Null pointer dereference;
- Null pointer dereference in method on exception path;
- Dereference of the result of readLine() without nullcheck;
- Immediate dereference of the result of readLine();
- Possible null pointer dereference due to return value of called method;
- Possible null pointer dereference on path that might be infeasible;
- Possible null pointer dereference;
- Possible null pointer dereference in method on exception path.

In addition, Reilly [Reilly2000] summarized two main reasons for this error as well:

- The object is not initialized;
- The return value of a function is not checked.

The result from the manual inspection shows 33 examples are identified in source code; however, the appearance of the above errors in the code differences is similar. So, this can be summarized as: a checking expression **if** `(x!= null)` (x represents an identifier name) is found in a subsequent version of code but not in the original code.

Error 7 *writing blank exception handlers*

The error '*writing blank exception handlers*' is one of the common coding errors. Many computer languages have built-in support for exceptions and exception handling, such as C++ and Java [Grimshaw1993, Gosling2000]. Methods may throw exceptions in Java, and hence corresponding exception handlers are needed. In fact,

many exceptions are difficult to predict before they occur in a program, and blank exception handlers make such exceptions even more difficult to identify. Even a simple printout exception handler is very helpful to identify exceptions, and hence promote error identification [Reilly2000]. FindBugs [Findbug2009] provides one description of this error, that is: exception created and dropped rather than thrown.

From the feedback of the manual inspection, one example is identified in source code. This type of error is described as follows: A blank exception handler expression `catch (Exception e){}` is identified in the original code and a defined exception handler `catch (Exception e){some expressions}` is identified in a subsequent version of code.

Error 8 *array index off-by-one*

The error '*array index off-by-one*' is common across many languages, especially Java because of the zero-index. This error is often related to iteration statements, such as *for* loops. The counter should point to the correct position for access in iteration, but it may point to an invalid position if the boundaries of a counter are not correctly set. Java uses a zero-index, so the smallest index is 0 and the biggest is N-1 (N is the number of loop). If the counter needs N moves, then the bound is $\geq 0 \wedge < N$, or $\geq 0 \wedge \leq$ N-1. The common mistake is that boundaries are set to $\geq 1 \wedge \leq N$, and this will throw an "*index out of range*" exception. No examples are identified in source code from the results of manual inspection; additionally, little literature is found to specific how to identify this type of error. The description of Error 8 is therefore as follows: an expression `i<=n` is found in a *for* or *while* loop in the original code and `i<n` is found in a subsequent version of code; or `i=n` is identified in a *for* or *while* loop in the original code and `i=n-1` is found in a subsequent version of code.

Error 9 *preventing concurrent access to shared variables by threads*

The error '*preventing concurrent access to shared variables by threads*' occurs when multi-threading is used. This type of error commonly occurs in various languages

including Java. Multi-threaded/concurrent programming is commonly used in many applications, so this type of algorithm requires data to be accessed by different threads simultaneously. In Java, this type of error often occurs when two or more threads access a shared variable. As a consequence, shared variables may not be correctly used or modified, and the output may become very unpredictable. A detailed method of identifying this type of error is specified in literature. For example, the FindBugs [Findbug2009] describe this type of error as: field not guarded against concurrent access Multithreaded. Based on the literature and the three identified examples from the manual inspection, the description of this type of error is defined as follows: a method `xyz()`(xyz represents a method name)is identified in the original code and `synchronized xyz()`is identified in a subsequent version of code.

Error 10 *accessing non-static member variables from static methods*

The error '*accessing non-static member variables from static methods*' is a typical coding error for Object-Oriented (OO) program languages, such as C++ and Java. This error often occurs when accessing member variables from the *static main* method because an instance of the object is required. Specifically, the literature provides methods to trace this type of error. For instance, the FindBugs [Findbug2009] specifies a detailed method that is likely to generate this type of error: TestCase implements a non-static suite method. The result from the manual inspection did not identified either of the above specific error examples nor other error examples. The specific error description is however able to be covered by the follows: a variable `x` is identified in a static method in the original code, and an instance of object `Y` and `Y.x` is found in the static method in a changed subsequent version of code.

Error 11 *stream not closed on all paths*

The error '*stream not closed on all paths*' is one of the typical coding errors in C++ and Java. The IO object is created but it is not closed finally. The literature describes four reasons to cause the stream not being closed [Findbug2009]:

- Method may fail to close database resource;

- Method may fail to close database resource on exception;

- Method may fail to close stream;

- Method may fail to close stream on exception.

In general, the description of this type of error is: a Stream object with `.close()` is not identified in the original code but it is found in the changed code. The manual inspection did not identify any examples in the change of source code.

Error 12 *dead code*

The error '*dead code*' refers to all unused statements including unused variables. This type of error is common for almost every kind of language. Little evidence is found to provide an applicable method to identify this type of error. Generally, the error can be described as follows: block of code is identified in the original code but deleted in the changed code. However, the requirement change can also lead to the deletion of code, so the original motivation of the deletion determines whether a block of code is redundant. The manual inspection did not identify this type of error in the change of source code.

Error 13 *confusion over passing by value and passing by reference*

Java uses *passing by value* and *passing by reference*. A *passing by value* occurs when a primitive data type is passed to a function, such as `char` and `int`. The original variable is not changed if the function needs to modify the variable. A *passing by reference* occurs when an object is passed to a function, such as `String` and `array`. The original object's member variables are changed if the function needs to modify them. Little evidence is found to guide the identification of this type error in the existing analysis tool. The manual inspection also did not identify any example of this type of error. A general description of this type of error is: a *passing by value* is replaced by *passing by reference* in the changed code or a *passing by reference* is replaced by *passing by value*.

Error 14 *mistyping the name of a method when overriding*

The error '*mistyping the name of method when overriding*' is common in OO language as overriding often occur. If a mistype of a method name occurs, a new method is created instead of overriding the existing one. Little evidence is identified to describe an applicable method for the identification of this type of error. The description of this error in the manual inspection is: a changed method name with a spelling correction is found in the changed code.

Error 15 *mathematics division by zero*

The error '*mathematics division by zero*' is common in various languages that have mathematic expressions such as x/y. The value of y cannot be zero otherwise the expression x/y may return an invalid value. Little literature is identified to provide applicable approach to identify this error. The manual inspection implements the description as: defining a variable not to be zero is found in the changed code.

3.4. Manual inspection process

The manual inspection identifies not only the 15 types of coding errors but also potential new types of coding error in the source code. The process applied to the source code follows four steps.

Step 1: Comparisons between different versions

The comparison of source code between versions is the first step of the manual inspection. Many errors were identified in the source code during the lifetime of the project. One of the most efficient ways to identify these errors is to analyze the changes made between different versions of code. The online-based CVS *diff* view function extracted differences between versions of the code, and marked *code added, code changed, code deleted* in different colours. The comparison of code started from the first version of code and finished with the latest version. For example, the project *sprite2D* contains hundreds of files. One of the files is `RenderContext.Java`.

This file has 17 versions. The manual inspection compared the original version and each subsequent changed version one by one. For example, comparing v1.1 to v1.2; v1.2 to v1.3, and up to v 1.n (n is the latest version number).

Step 2: Potential problematic blocks of code

Identifying problematic blocks of code is the second step of the manual inspection. Following the descriptions of the ten types of errors, potential examples of the coding

Original code	Changed code

` int strLength, curX;`
`int curWidth, curHeight;`
 	
`if (font.isDefined()) {`
`strLength = text.length();`
`curX = x + calcTextXOff(font, text,`
`justification, width);`
	`int strLength, curX;`
	`int curWidth, curHeight;`
	`if(font != null) {`
	`if (font.isDefined()) {`
	`strLength = text.length();`
	`curX = x + calcTextXOff(font, text,`
	`justification, width);`

Figure 3.3 Fragment of code from project sprite2D

errors were found in the code extracted by *diff*. For example, one possible example of the error *null pointer* is found in the changed code of file `RenderContext.Java` between version 1.2 and 1.3. This is illustrated in Figure 3.3.

In practice, many code changes could be wrongly recognized as an actual example of the error. Therefore, all suspect code blocks require a more detailed review of the code to verify.

Step 3: Examples of errors

The next step is to analyze the relevant context of the code to confirm each example of the 15 types of error, since incorrect identification of errors would bias the results. For example, the example of the error *null pointer* was confirmed by analyzing the relevant context of the code in Figure 3.3.

Step 4: Relevant information about the project and the example of errors

Finally, the confirmed example of errors was manually input into the database. The database recorded information about projects and examples of 15 types of error. The database was developed in Microsoft Access 2003. The relationship is shown in Figure 3.4.
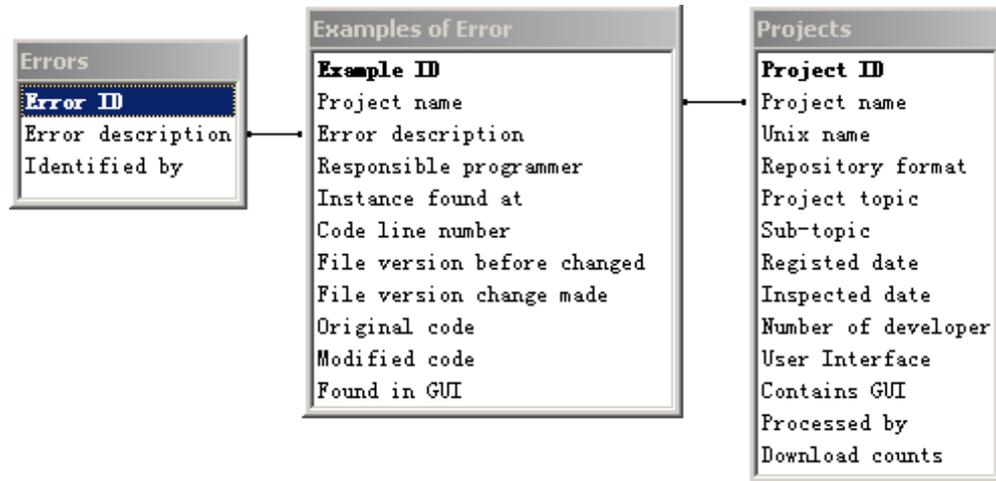
Figure 3.4 Relationship structure of information recorded for each example of errors

Figure 3.4 shows that the database contains three tables: *Projects, Example of Errors,* and *Errors*. The three tables are connected to each other. Information in the table *Projects* were obtained from SourceForge. Information in the table *Examples of Errors* was obtained from the results of the manual inspection.

The table *Projects* recorded relevant information about each project. Each project was given a unique ID. Moreover, the *Repository format* indicates the type of repository in which the project was developed, such as CVS. Other information was also recorded such as download counts, and project type. The table *Examples of Errors* recorded information about each error example. The *Example ID* is used for identification. *Example found at* and *Code line number* shows the exact location where the error example was found. *File version before changed* and *File version change made* indicate when the change is made in the lifetime of the project. The *Original code* and *Modified code* show the comparison between the original code and modified code. The *Found in GUI* indicates whether the example of errors is found in GUI or non-GUI code. The table *Errors* recorded descriptions of the 15 types of coding errors.

An explanation of each type of error and the author who mentions the error are recorded in this table.

3.5. Common Coding Errors Set (CCES)

The CCES was applied to a tool that is using a set of pattern matching rules to automatically identify errors in the changes. The tool is a simulation of the manual inspection but with efficient identification of errors. However, the results of the manual inspection shows that seven types of coding errors were identified in the 25 projects. Eight out of 15 types of coding errors were not identified by the manual inspection, and there was no new type of coding error in the changed code. Additionally, some types of errors seem not applicable to the approach of identifying errors in the automatic inspection.

The manual inspection identified seven types of errors from the set of 15. The process of manual inspection evaluates the applicability of the 15 types of errors in the automatic inspection; furthermore, the manual inspection evaluates the possibility of implementing variant methods described in existing analysis tools. The final result of the manual inspection is to generate a set of common coding errors that is applicable to the automatic inspection tool. However, the manual inspection did not identify some error examples in the code due to human error and the subjective nature of the analysis; this is shown in the latter automatic inspection.

Error No	Errors Description	Number of errors identified	Availability of existing methods	Applicability of error	Applied to Automation
1	Comparison assignment ("=" instead of "==")	0	No	Yes	Yes
2	Comparing two objects ("==" instead of ".equals()")	3	Yes	Yes	Yes
3	Capitalization error	1	Yes	Yes	Yes
4	Variable type error	1	No	Yes	Yes

5	Uninitialization	5	Yes	Yes	Yes
6	Null pointer	33	Yes	Yes	Yes
7	Writing blank exception handlers (try{}…catch{})	1	Yes	Yes	Yes
8	Array index off-by-one	0	No	Yes	Yes
9	Preventing concurrent access to shared variables by threads	3	Yes	Yes	Yes
10	Accessing non-static member variables from static methods	0	No	Yes	Yes
11	stream not closed on all paths	0	Yes	Yes	No
12	Dead code	0	No	No	No
13	Confusion over passing by value and passing by reference	0	No	No	No
14	Mistyping the name of a method when overriding	0	No	No	No
15	Mathematics division by zero	0	No	No	No
Other		0	No	No	No

Table 3.1 Ten types of error applied to automatic inspection

There are three possible reasons for not finding eight types of error: (i) the unidentified types of coding errors are not common; (ii) these errors exist in code but were not discovered by the programmers or users, so they are not able to be identified in the changed code; (iii) the manual inspection may skip some types of errors due to human error. The first explanation is in conflict to literature so it needs to be justified by the automatic inspection. The second explanation is possible but the popularity of unidentified errors should not seriously bias the results. The third explanation is the most likely cause since human error was not avoidable. Considering the result from the manual inspection and applicability of each type of error in this thesis, ten types of error are selected to be implemented in the automatic inspection. The identification of each type of error by the manual inspection is shown in Table 3.1. The common coding errors are therefore identified and later implemented to the automatic

inspection.

Applicability and feasibility are the criteria to select the final errors in CCES. If errors can be identified in code fragment which is the change between sequential versions then it will be implemented in the automatic inspection. The inapplicable errors are due to the reason that the fragment of code after *diff* is not complete so corresponding algorithm of identifying errors is not supported. As a result, ten types of errors are selected to the CCES for the automatic inspection.

3.6. Automatic inspection process

The automatic inspection was achieved by using a developed tool. This Automatic Code Inspection Tool (ACIT) was developed in Java. It implemented the CCES error patterns and applied them to the changed code. The motivation for developing the ACIT is to automatically catch the identification of CCES errors between two subsequent versions of code. This section describes the automatic inspection process and algorithm to implement each type of error.

3.6.1. Overview of the automatic inspection

The objective of the automatic inspection is similar to the manual inspection. However, the process of the automatic inspection is different from the manual inspection. The automatic inspection was mainly achieved by the ACIT. The process of the automatic inspection is shown in Figure 3.5.
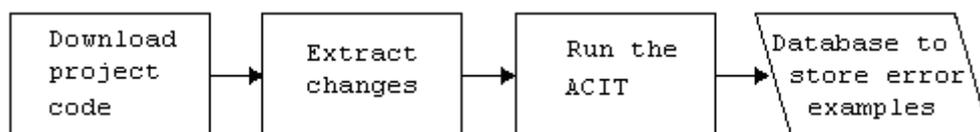
Figure 3.5 Process diagram for the automatic inspection

3.6.2. Project sample for the automatic inspection

Projects for the automatic inspection were selected six months after the manual inspection. 45 projects were selected for the automatic inspection but they are different from the 25 projects for the manual inspection.

The manual inspection reviewed the source code through web-based CVS; however, not all repositories were able to be downloaded. There were only 13 projects that their repositories of source code were downloadable under CVS. To increase the size of sample of projects for the automatic inspection, 32 new Java projects were selected from SourceForge. Figure 3.6 shows the intersection of projects processed by the manual and automatic inspection.
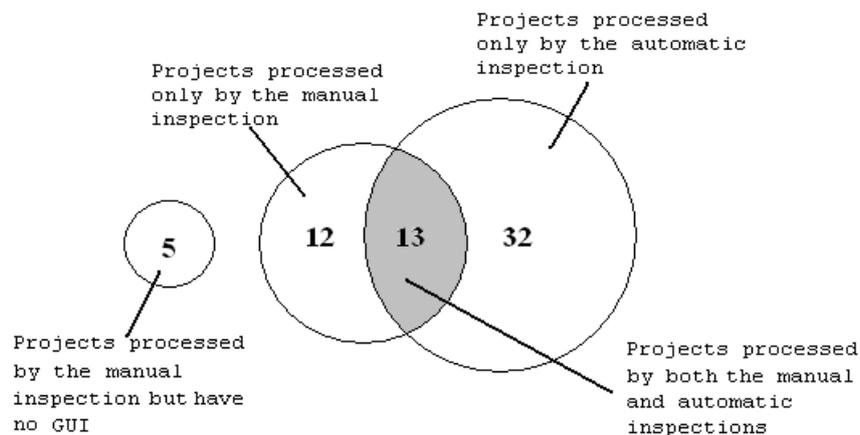
Figure 3.6 Intersection of projects processed by the manual and automatic inspections

There were some new projects registered between 25th December 2005 and 25th June 2006 so the population of Java projects had changed. The 45 projects for the automatic inspection were downloaded before 25th June 2006. There were 98909 projects, 17710 were written in Java, and 3531 Java projects had AWT, SWT, or Swing GUI. The 45 projects, which were processed by the automatic inspection, are a sample of the population of 3531.

Step 1: Download project code.

The selected 45 projects were downloaded from SourceForge, including all versions of each source file. The full versions of each file were downloaded from the CVS repository of the SourceForge. These source files were stored for further processing in UNIX.

Step 2: Extract changes.

The ACIT requires the input of changes between versions of code to identify error examples, so the changes need to be extracted from the source files. A shell script was executed to run *diff-* between any two subsequent versions of code. All changes between versions for each file were written to a text file. Thus there were 45 text files and each file contained all changes for one project. These text files at this point were ready to be processed by the ACIT.

Step 3: Run the ACIT.

The process of the ACIT:

1. Reads diff- code from .txt files and extract the original and changed code ;

2. Stores the original code and changed code in two LinkLists, and records relevant information from the header code, such as version information, line number, file directory, and so on;

3. Compares the original code and the corresponding changed code, and identifies examples; stores these examples together with relevant information in a list;

4. Writes these examples together with relevant information (such as version number) to a .txt format file; calculates and writes statistical results to the .txt file.

Step 4: Database to store error examples

Information recorded in the text file was stored in the database created in the manual inspection process. Statistical results were produced after analyzing data from the database.

3.6.3. Implementation of ten types of errors

The CCES error patterns were applied by the ACIT. The key of finding the identification of errors relied on the comparison of corresponding code between subsequent versions. This section describes details of implementing each CCES error in the ACIT.

Information read from diff- text file

The ***diff-*** function identifies three types of changes: `a` (code added), `d` (code deleted), and `c` (code changed). The ACIT reads the three identifiers to decide the type of changes. Furthermore, the character "<" indicates code that follows "<" are the original code; and the character ">" indicates code that follows ">" is the changed code. The ACIT reads the two characters to identify the original or changed code.

```
87c90@/cvsroot/arch4j/arch4j/components/base/src/org/arch4j/core/Base
ApplicationException.java%1.1 1.2
<    public void printStackTrace(java.io.PrintStream ps) {
>    public void printStackTrace(PrintStream ps) {
```

Figure 3.7 A fragment of a diff text file

There is an instruction line before every block of original and changed code in the *diff-* file. The original and changed code is displayed beneath this line of code. Information can be read from certain characters, such as "c", and "<". Figure 3.7 shows an example of a diff text file.

Error 1 *"=" instead of "=="*

To identify the error '*comparison two assignment*', four steps are applied.

    Step 1. The AWCT reads code stored in the diff- text file, recognizes the original and changed code by identifying "c" in code. All comments, which are between "/*" and "*/" or after "//", are ignored (This algorithm of ignoring comments is applied to other types of CCES errors).

Step 2. Identifies a string containing the string "==" in the changed code that is indicated by ">"; stores the string in a list.

Step 3. Seeks a string containing character "=" in the original code which is indicated by "<". If such a string is found, store in another list. If no such string is found, empty the two lists and moves on to the next block of code.

Step 4. Compares stored strings from the two lists. If the error pattern "=" instead of "= =" is matched, checks the identifiers at left and right hand sides of "=" and "==". If the identifiers are also matching, then the example of error is confirmed and output to the text file.

An example of Error 1 is shown in Figure 3.8.

```
97c97@/cvsroot/unicats-i/unicats-i/src/de/unicats/comm/Communication
Module.java%1.7 1.8
<       if (intern=true)
>       if (intern==true)
```

Figure 3.8 An example of Error 1 from project unicats-i

Error 2 *"==" instead of ".equals()"*

The implementation of identification of Error 2 is similar to Error 1. In this case, however, a string containing "*.equals()*" needs to be identified in the changed code and the string containing *"=="* needs to be identified in the original code. An example of Error 2 is shown in Figure 3.9.

```
156c156@/cvsroot/jfritz/jfritz/src/de/moonflower/jfritz/dialogs/quic
kdial/QuickDialPanel.java%1.9 1.10
<       if (e.getActionCommand() == "deleteSIP") {
>       if (e.getActionCommand().equals("deleteSIP"))
```

Figure 3.9 An example of Error 2 from project jfritz

Error 3 *capitalization*

To catch the identification of error '*capitalization*', a comparison between method

names are required. The process is as follows:

Step 1. The ACIT identifies a block of changed code that is indicated by "c".

Step 2. All method names in the original and changed code are extracted and stored in two lists. A method name in Java is after the character "." or a space " " and ends with the character ")", so the method name string is able to be extracted following this feature (the algorithm to extract a method name string follows this rule for other types of CCES errors).

Step 3. The ACIT compares all names in the two lists without considering the case of the alphabetical characters. If the same method name is identified in the two lists, the process moves on to the next step. If not, it moves on to the next block of code.

Step 4. The two methods names are compared by considering the case of each letter in the method name string. If an upper case is found in the method name from the changed code, then an example of Error 3 is confirmed and output.

Figure 3.10 shows an example of Error 3.

```
11c11@/cvsroot/unicats-i/unicats-i/src/de/unicats/ws/comm/ExternalCo
mmunicationService.java%1.1 1.2
<get externalCommunicationAddress()...
>get ExternalCommunicationAddress()...
```

Figure 3.10 An example of Error 3 from project unicats-i

Error 4 *variable type error*

To identify the *variable type* error, statements of declaring variables need to be identified. A variable declaration follows: *variable type + variable name + "=" + initial value (the initial value is alternative).* There are four variable types to define integers in Java (`byte`, `short`, `int` and `long`) and two types to define real numbers (`double` and `float`). The identification of Error 4 is made up of several

steps:

Step 1. The ACIT identifies a block of changed code that is indicated by "c".

Step 2. Strings containing "byte", "short", "int", or "double" are extracted from the original code. Strings containing "short", "int", "long", or "float" are extracted from the changed code. Furthermore, the string to declare a variable is intercepted and stored in two lists.

Step 3. Strings for the variable declaration are compared. If the same variable names with different variable types are identified in the two lists, then it moves on to next step. If not, the ACIT continues to process the next block of code.

Step 4. If the variable type matches any rules shown in Table 3.2, an example of Error 4 is confirmed and output. An example of Error 4 is shown in Figure 3.11.

	Variable type in original code	Variable type in changed code
Rule 1	Byte	short, int, long, float, double
Rule 2	Short	int, long, float, double
Rule 3	Int	long, float, double
Rule 4	Long	float, double
Rule 5	Float	Double

Table 3.2 Rules to identify variable type error

```
163c190@/cvsroot/javaseis/javaseis/src/org/javaseis/parallel/Decompo
sition.java%1.4 1.5
<       int nrem, nlive, npn;
>       long nrem, nlive, npn;
```

Figure 3.11 An example of Error 4 from project javaseis

Error 5 *uninitialization*

The key rule to identify the error '*uninitialization*' is that a variable is not assigned an initial value when it is declared in the original code, but it is assigned a value in the changed code. The process of finding Error 5 has several steps as follows:

Step 1. The ACIT identifies a block of changed code that is indicated by "c".

Step 2. The ACIT identifies a variable declaration string following the same rules used in Error 4. Strings from the original and changed code are stored in two separate lists.

Step 3. Compares strings in the two lists. If the same variable declaration is identified in two lists, then its following content is checked.

Step 4. The rule of confirmation of this error is: variable declaration + ";" in the original code and variable declaration + "=" + value + ";" in the changed code. If this rule is matched, an example of Error 5 is recorded (see error example in Figure 3.12).

```
20,23c29,32@/cvsroot/unicats-i/unicats-i/src/de/unicats/agents/Group
.java%1.1 1.2
<   public String name;
>   private String name = "";
```

Figure 3.12 An example of Error 5 from project unicats-i

Error 6 *null pointer*

The rule of the error '*null pointer*' depends on whether a checking statement is not in the original code but added in the changed code.

Step 1. The ACIT identifies a block of changed code that is indicated by "c".

Step 2. All code starts with "<" is stored in list 1. The ACIT identifies the statement if (X!=null) where X is a name of object or variable in the changed code. If this statement is identified, then this statement and all statements after it in the same code line are recorded in list 2.

Step 3. Compares the code stored in list 1 to code stored in list 2. If the code matches in the two lists (without the code if(X!=null)), then check if the code if(X!=null) is not in list 1. An example of error 6 is confirmed and recorded.

```
1921c1966,1968@/cvsroot/unicats-i/unicats-i/src/de/unicats/agents/cu
stomerAgent/MessageHandlingThread.java%1.26 1.27
<   caContent.setString(root.toString());
```

```
>   if (root != null) { caContent.setString(root.toString());
```

Figure 3.13 An example of Error 6 from project unicats-i

Error 7 *blank exception handlers (try{} catch{})*

The key rule to identify the error '*writing blank exception handlers*' is that the code
"catch(X){}" is identified in the original code but "catch(X){y}" is found in
the changed code. The process is as follows:

Step 1. The ACIT identifies a block of changed code that is indicated by "c".

Step 2. Code containing "catch(…){}" (there must be no content in "{}") is
found in the original code, and "catch(…){…}" (there must be some
content in "{}") is found in the changed code. If the rule is matched,
code is added to two lists, one for the original code and other one for the
changed code.

Step 3. The ACIT compares code from the two lists. If contents in the
"catch(…)" are matched, then an example of Error 7 is recorded (see
example in Figure 3.14).


```
407c410,412@/cvsroot/frinika/frinika/src/com/frinika/synth/Attic/myS
ynth.java%1.8 1.9
<catch(Exceptione){}
>catch(Exceptione){e.printStackTrace();}
```

Figure 3.14 An example of Error 7 from project frinika

Error 8 *array index off-by-one*

This type of error is identified in *for* or *while* loops. The algorithm to find it is that the
boundary value has been changed. The process of identifying this error follows:

Step 1. The ACIT identifies a block of changed code that is indicated by "c".

Step 2. The *for* or *while* loop statement is identified in both original and changed
code. Code in "for()" or "while()" identified in the original code is
stored in a list, and code identified in the changed code is stored in
another list. Specifically, the statement "x=n;" (x is a variable name and
n is a value) before the "while()" is also stored with the

while()statement because n is the initial value of the loop index.

Step 3. Code "for()" or "x=n; while()" is compared between the two lists. If a change of boundary value is identified, then an example of Error 8 is output. The rules are shown in Table 3.3, and an example of Error 8 is shown in Figure 3.15.

	Boundary in original code	Boundary in changed code
When "i++" or "i=i+1" identified	i<=n;	i<n;
When "i--" or "i=i-1" identified	i=n;	i=n-1;

Table 3.3 Rules to confirm loop boundary change

```
219,226c321,336@/cvsroot/jdba/jdba/src/org/jdba/db/oracle/OracleTabl
eModel.java%1.1 1.2
<   for(int col = 0; col<=getColumnCount(); col++) {
>   for(int col = 0; col <getColumnCount(); col++) {
```
Figure 3.15 An example of Error 8 from project jdba

Error 9 *preventing concurrent access to shared variables by threads*

The key word of identifying this type of error is '*synchronized*' in the changed code. The ACIT processes are as follows:

Step 1. The ACIT identifies a block of changed code that is indicated by "c".

Step 2. Adds the original code to a list line by line. Identifies the code containing the string 'synchronized' in the changed code and extracts the method name after synchronized the method name ends with "(".

Step 3. Compares the method name in the two lists. If the same name is identified in both lists and no 'synchronized' is identified before the method name extracted from the original code, then an example is confirmed and recorded (see example in Figure 3.16).

```
108c110@/cvsroot/marf/marf/src/marf/Stats/ProbabilityTable.java%1.39
1.40
<public final int size()
>public final synchronized int size()
```

Figure 3.16 An example of Error 9 from project marf

Error 10 *accessing non-static member variables from static methods*

The key to identify this coding error is to identify the expression '*instance_name.variable_name*' in the changed code and '*variable_name*' in the original code. The algorithm is as follows:

Step 1. The ACIT identifies a block of changed code that is indicated by "c".

Step 2. Identifies code containing character '.', and then confirm it is not a method by checking every character after '.' until the next symbol. A method will always follow a brace '('. Store the variable name into a list whether there is a confirmation of a variable.

Step 3. Runs through the list, check whether the variable can be identified in the original code. If the variable is identified, confirm the variable does not follow character '.'. Output the original and changed code to the text file.

This type of coding error is not identified by the automatic inspection therefore no example of Error 10 is able to be presented.

3.7. Chapter summary

In this chapter, the approach of AICCE that identifies common coding errors between code versions was described. Two sample sets of projects were selected for the manual and automatic inspections. After examining the results from the manual inspection, the refined common coding error set CCES was confirmed and applied to the automatic inspection. The results from the two inspections were from two different samples of project. Therefore, the results from the two inspections need to be evaluated. Chapter 4 analyzes the profile of the two sample sets and presents results from two inspections.

Chapter 4　Experimental Projects and Results

4.1.　Introduction

The interpretation of results obtained from the manual and automatic inspections provide evidence for further analysis. The manual inspection is a pilot for the automatic inspection and it provides results for the implementation of the automatic inspection. This chapter not only describes the profile of selected projects, but also interprets the results from the manual and automatic inspections.

The AICCE processed 57 selected projects. There were two inspections in the AICCE, and they processed two different sets of projects. The manual inspection processed 25 projects, the automatic inspection processed 45 projects, and 13 projects were processed by both the manual and automatic inspections. The intersection of shared projects is shown in Figure 3.6. The profiles of the two sets of projects are described in section 4.2.

The two inspections output two sets of data. The manual inspection not only attempted to identify common coding errors which were summarized by other researchers, but also tried to discover new coding errors in the source code. However, no new types of common coding error were actually identified. The identification of the ten types of CCES error, which were applied by the two inspections, is presented in this chapter. A description of the two sets of data is provided in section 4.3. Additionally, statistical results of the two sets of data are also presented in this chapter.

## 4.2.   The profile of the processed projects

The 57 processed projects are divided into two sets. One set is for the manual inspection, and the other one is for the automatic inspection. Different results were obtained from the two sets of project samples, so the profiles of two sets of project samples are compared.

Overview of project samples

Features of the 57 projects were gathered from SourceForge. Information was quantified to present the profiles. For example, the feature 'project size' was measured using LOC. The description of the sample includes project type, project size, development history, download counts, and development team size.

The measurement of the above features is presented as follows: Figure 4.1 shows the number of projects in each type of project for the manual inspection; Figure 4.2 is for the automatic inspection; and Table 4.1 shows the maximum, minimum, and average values of the two sets of projects for each feature.

Figures 4.1 and 4.2 show that there are seven types of projects in the manual inspection projects set and 12 types of projects in the automatic inspection projects set. The proportions of the main types of project are similar in the two sets of projects. For example, the largest proportion of the manual inspection projects set is the type Game/Entertainment, and it is also the second largest in the automatic inspection projects set. In fact, the two sample sets (two sets of projects) were selected from the same population but at different times. Therefore, features of the two sets of projects are similar. In addition, many projects are multi-type applications so it is difficult to measure exact proportions. All 57 projects were classified into one main type based on its primary type description.
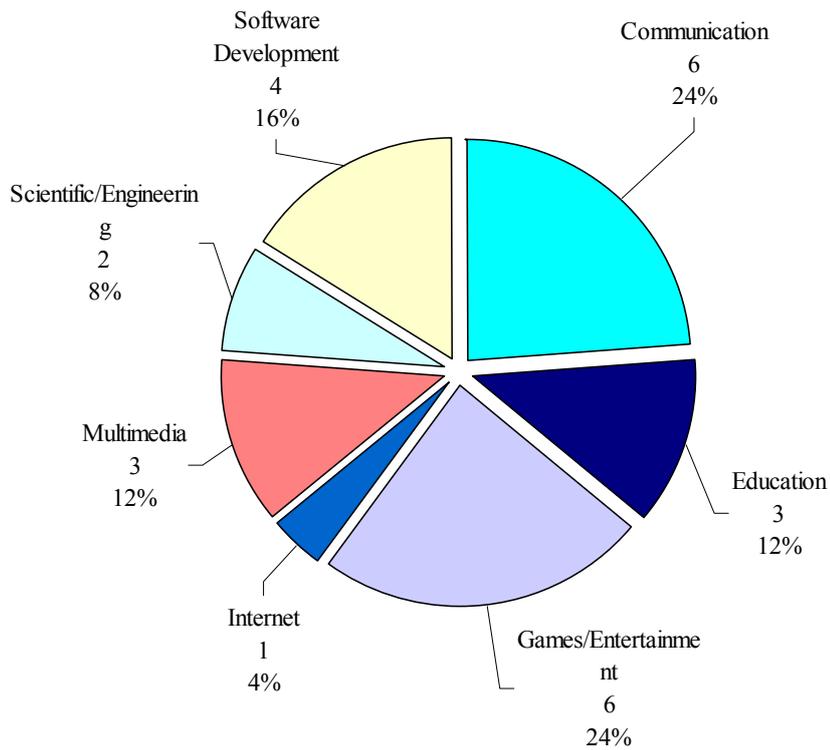
Figure 4.1 Percentage of each type of project (the manual inspection projects set)
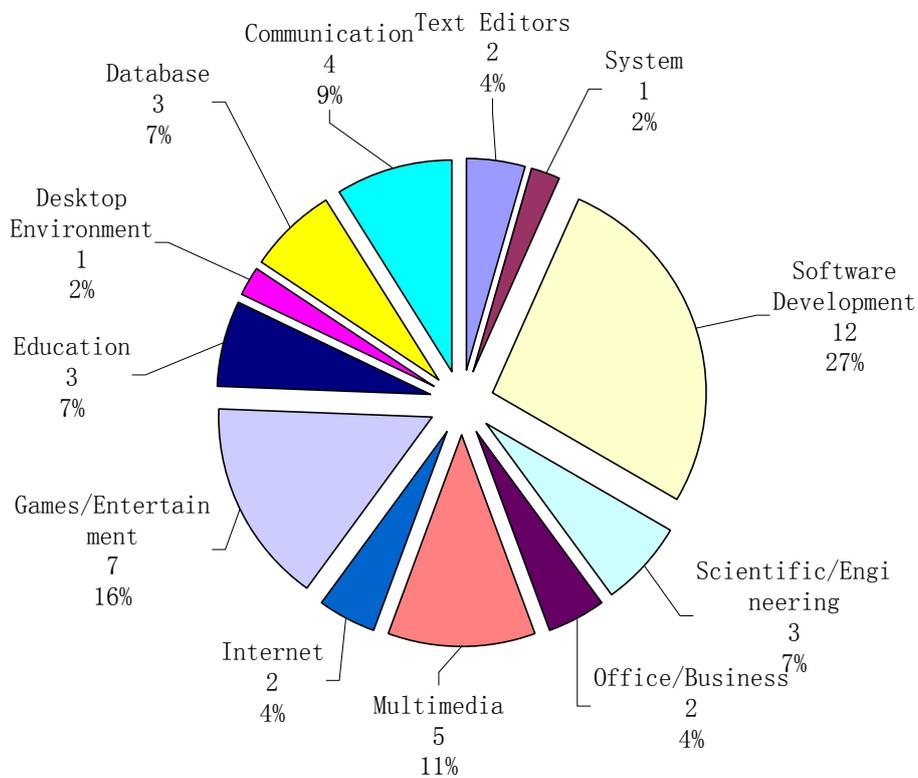
Figure 4.2 Percentage of each type of project (the automatic inspection projects set)

The 57 projects are a reflection of the population of Java GUI projects in SourceForge. The proportion of each type of project approximately reflects the distribution of different types of projects in the population. Figures 4.1 and 4.2 show that project types Software Development, Scientific/Engineering, Game/Entertainment，and Communication are the most common applications in SourceForge. Additionally, OSS features also reflect the profile of the two sample sets of projects from different perspectives. These features of the two sample sets are shown in Table 4.1.

Feature	Manual inspection projects			Automatic inspection projects		
	Max	Mean	Min	Max	Mean	Min
Project size (LOC)	57482	17466	1166	282638	34494	258
Development history (months)	63	13	2	76	26	0
Number of developers	13	4	1	30	9	1
Download counts	26159	2276	0	1562551	73798	0

Table 4.1 Features of two set of projects

Table 4.1 shows that the maximum and average values of the three out of four features for the automatic inspection projects are greater than the manual inspection projects. The minimum values are the same for three features. Table 4.1 shows that projects in the sample set for the automatic inspection have greater size, longer development times, greater development teams, and greater frequency of download. However, all these projects are selected from the same set of OSS projects hence the overall results should be similar between the manual and automatic inspections.

The literature shows that GUI code occupies a large amount of LOC in software. For example, Memon [Memon2002] identified that GUI constitutes 60% of the total software code. The results of the manual inspection show that the overall percentage of GUIGUI LOC is approximately 50% of the OSS code. The results of the automatic inspection show that the overall percentage of GUI LOC is approximately 29% of the OSS code. Considering the two inspections, the overall percentage of GUI LOC is

39% (total GUI LOC divided by total LOC). This result is slightly lower than the percentage 45% ~ 60% summarized by Myers [Myers1995].

4.3. Data obtained from the manual and automatic inspection

The manual inspection identified seven types of CCES errors and the automatic inspection identified nine types of CCES errors. The frequency of identification of each type of CCES error is different between the manual and automatic inspections because the two sample sets vary.

4.3.1. Overview of the identification of CCES

The CCES was found in 14 projects by the manual inspection and 34 projects by the automatic inspection. In other words, the CCES was identified in 56 percent of the manual inspection project samples and 76 percent of the automatic inspection project samples. The results of the manual inspection show that the CCES errors were identified in GUI code of 12 projects, in non-GUI code of five projects. The CCES errors were identified in both GUI and non-GUI code of three projects by the manual inspection. The automatic inspection identified the CCES errors in GUI code of 28 projects, and in non-GUI code of 27 projects. There are 22 projects where CCES errors were identified by the automatic inspection in both GUI and non-GUI code.

The difference between the manual and automatic inspection in identifying the CCES errors is due to various factors. One of the possible factors is that the manual inspection was prone to human instead of program.

4.3.2. The CCES ranking of errors in GUI and non-GUI code

The literature survey identified that the error '*null pointer*' (E6) is the most frequently occurring coding error in the code [Chou2001, Zheng2006, Reilly2000, Rutar2004]. The manual inspection identified the E6 *null pointer* error as the most frequently identified error in both GUI and non-GUI code. The automatic inspection identified that the E6 error is the top coding error in GUI code and third most common in non-GUI code. However, the E6 *null pointer* error was identified in more than two thirds of non-GUI code. By considering the results from the two inspections, a ranking of CCES is presented in Table 4.2. However, five types of error are not implemented in the automatic inspection so they are not shown in the table.

Table 4.2 shows the ranking of each CCES error for GUI code and non-GUI code. Errors *null pointer* (E6), *multithreaded* (E9), *variable uninitialization* (E5), '*== instead of '.equals()*' (E2), *variable type* (E4) are the most frequently identified errors overall. Table 4.2 also shows that the top five errors are similar between GUI and non-GUI code of the 57 OSS projects. Moreover, the rankings for GUI and non-GUI code almost corroborate the five top errors (E6 *null pointer* error, E9 *multithreaded* error, E5 *variable uninitialization* error and E2 *== instead of .equal()* error) in Table 2.7.

Errors *array index off-by-one* (E8) and *static method non-static member* (E10) are the most infrequently identified coding error in GUI code of the 57 OSS projects. E10 is also the most infrequently identified error in non-GUInon-GUI code. This implies that the E10 *static method non-static member* error is found sometimes but not frequent. No example of E10 error was found in GUI or non-GUI code. This result demonstrates that E10 is seldom identified after the first version of code is released for an OSS project.

The literature shows that the error '*array index off-by-one*' (E8) was identified as the

third most common coding error in C++ and Java code [Chou2001,Zheng2006, Robinson2004]. However, this thesis shows that the E8 *array index off-by-one* error was not frequently identified by either the manual or automatic inspections, especially in GUI code of the 57 OSS projects.

Ranking (the manual inspection)		Ranking (the automatic inspection)		Ranking (the literature)
GUI	non-GUI	GUI	non-GUI	
E6 (12)	E6 (21)	E6 (248)	E9 (410)	E8
E5 (5)	E9 (1)	E9 (36)	E6 (347)	E6
E2 (3)	E1 (0)	E5 (27)	E4 (230)	E9
E9 (2)	E2 (0)	E4 (21)	E5 (102)	E5
E3 (1)	E3 (0)	E2 (20)	E2 (37)	E2
E4 (1)	E4 (0)	E7 (13)	E3 (31)	E1
E7 (1)	E5 (0)	E3 (11)	E7 (10)	E4
E1 (0)	E7 (0)	E1 (0)	E8 (9)	E7
E8 (0)	E8 (0)	E8 (0)	E1 (2)	E10
E10 (0)	E10 (0)	E10 (0)	E10 (0)	E3

Note: values in the bracket represent the frequency of error

Table 4.2 Ranking of the coding error identified by the manual and automatic inspection

The automatic inspection identified a greater frequency of errors compare to the manual inspection. Table 4.3 shows the frequency of identification of each type of error between the manual and automatic inspections. Seven out of ten types of error were identified in GUI code by the manual inspection identified, and only two out of ten in non-GUI code (see Table 4.3).

The automatic inspection also identified seven out of ten types of error in GUI code and nine out of ten types of error in non-GUI. The comparison between GUI and non-GUI shows that there are great differences of frequency of E2 *= = instead of .equals()* error, E3 *capitalization* error, E4 *variable type* error, E5 *uninitialization* error, E8 *array index off by one* error, and E9 *multithreaded* error between the manual

and automatic inspections. The results also vary between the two inspections for the number of projects where the CCES errors occurred.

Manual inspection				Automatic inspection			
GUI		non-GUI		GUI		non-GUI	
Error Label	Number of errors	Error Label	Number of errors	Error Label	Number of errors	Error Label	Number of errors
E1	0	E1	0	E1	0	E1	2
E2	3	E2	0	E2	20	E2	37
E3	1	E3	0	E3	11	E3	31
E4	1	E4	0	E4	21	E4	230
E5	5	E5	0	E5	27	E5	102
E6	12	E6	21	E6	248	E6	347
E7	1	E7	0	E7	13	E7	10
E8	0	E8	0	E8	0	E8	9
E9	2	E9	1	E9	36	E9	410
E10	0	E10	0	E10	0	E10	0

Table 4.3 The frequency of identification of each type of error between the manual and automatic inspections

4.3.3. Comparison between the manual and automatic inspections

Table 4.4 shows that most types of error were successfully identified in GUI code by the two inspections. However, many errors were not identified in non-GUI code by the manual inspection. One of the possible explanations is that the manual inspection may skip some examples of error in non-GUI code since the manual inspection was prone to human error.

Error Label	Manual inspection		Automatic inspection	
	GUI code	non-GUI code	GUI code	non-GUI code
E1	0	0	0	1
E2	3	0	6	6
E3	1	0	4	6
E4	1	0	8	14
E5	4	0	14	16
E6	8	4	22	23
E7	1	0	8	7
E8	0	0	0	5
E9	1	1	9	13
E10	0	0	0	0

Note: values in cell represent the number of project in which errors were identified, the manual inspection processed 25 projects and the automatic inspection processed 45 projects

Table 4.4 The number of projects in which errors were identified

The false negative cases and false positive cases were identified in the manual inspections. The manual inspection was conducted by the author, so it is difficult to search the identification of errors in a large block of changed code but comparatively easy in a small block of changed code. Thus the false negative results possibly influenced the frequency of the identification of the CCES errors for GUI code. In this thesis, a false negative example means an actual example of the CCES errors which should have been identified but is missed. A false positive example means a false example which should not have been identified but is identified in fact.

Some examples of the CCES errors were not identified by the manual inspection due to human error. Therefore, the result from the manual inspection could be biased by the missing error examples. Some projects that were selected for the manual inspection are not available for the automatic inspection since their source code are not capable for download (empty result returned while executing command `cvs -z3 -d:pserver:anonymous@`project`.cvs.sourceforge.net:/cvsroot/`project` co -P` project). The result from the manual inspection will not be considered for further analysis.

13 projects were processed by both inspections, hence allowing comparisons. Ignoring the newly identified errors by the automatic inspection in the six month period (projects were selected for the two inspection in two different time) and confirming all error examples identified by the manual inspection were also identified by the automatic inspection, the frequency of errors are shown in Table 4.5.

Project name	Number of errors (Manual)	Number of errors (Automatic)	False negative	Negative percentage for the manual inspection
2voor12	2	4	2	50%
frinika	4	7	3	42.86%
galleon	9	27	18	66.67%
jfritz	0	20	20	100%
patcheditor	1	1	0	0%
qii	2	4	2	50%
saiph	0	0	0	0%
sink	3	14	11	78.57%
sprite2d	9	23	14	60.87%
ugt	0	12	12	100%
vitapad	0	1	1	100%
vlma	0	3	3	100%
xfngraph	0	1	1	100%
Total	2.3	9	6.7	65.30%

Table 4.5 Number of errors identified by the manual and automatic inspections

Table 4.5 shows that the difference between the automatic inspection and the manual inspection is between 0 and 20 per project. In Table 4.5, values in the fourth column represent the number of CCES errors identified by the automatic inspection minus the number of CCES errors identified by the manual inspection. The positive value in the fourth column represents the false negative example for the manual inspection. The average number of false negative example (the value is 6.7) is almost three times of the frequency of errors identified by the manual inspection (the value is 2.3) in the 13 common OSS projects. The average percentage of false negative examples is

approximate 65%.

Based on the data recorded, some false positive examples were identified in the manual inspection. The number of correct examples is 45, and the number of all identified examples is 57 in the manual inspection. Therefore, the ratio of actual to original coding error cases is 45/57 $\approx$ 78.9 %. The number of correct examples alongside original examples of each CCES errors is shown in Table 4.6.

Error type	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
Original	0	5	1	4	8	37	1	0	3	0
Corrected	0	3	1	1	5	33	1	0	3	0
Ratio (Cor/Orin)	N/A	60%	100%	25%	62.5%	89.2%	100%	N/A	100%	N/A

Table 4.6 Number of actual and original error examples identified by the manual inspection

Table 4.6 shows that no false positive example of the E3 *capitalization* error, E7 *blank exception handler* error, and E9 *multithreaded* error were identified in the manual inspection. The E4 *variable type* error has the largest portion of false positive examples in the manual inspection. The false positive results are wrongly identified due to human error. A typical example of the incorrect identification of code is the error '*varialbe uninitialization*' (E5) which is shown in Figure 4.3.

Original code	Changed code
`int arrowWidth = arrows.getImageWidth();`	`int arrowWidth = 0;`

Figure 4.3 Code fragments for project *spride2D*

Figure 4.3 shows a false positive example identified by the manual inspection. The `int arrowWidth` is already initialized in the original code, and the initial value is changed in the subsequent code but this change is not due to the '*uninitialization*' (E5). Human error wrongly identified this creating a false positive identification.

In summary, the drawbacks analyzed in this section determine that the results obtained from the manual inspection are flawed; however the results only provide experimental referential data for the implementation of the automatic inspection but careful consideration must be made before further conclusion regarding the manually identified sample is made.

<u>False negative for the automatic inspection</u>

The manual review of results obtained from the automatic inspection identified several false negative examples. Corresponding code are presented in Figures 4.4 and 4.5.

Original code	Changed code
```if(aantalSpelersButton.getText() == "1 Speler")…```	```if (aantalSpelersButton.getText() .equals("1 Team"))…```

Figure 4.4 Code fragments for E2 *==instead of .equals()* error in project *2voor12*

Original code	Changed code
```Private SpelController controller;```	```private SpelController controller = null;```

Figure 4.5 Code fragments for E5 *uninitialization* error in project *2voor12*

The automatic inspection did not identify the change shown in Figure 4.4. The design of the ACIT did not consider the change of the right hand side of "==", so this type of change was ignored by the ACIT.

The error example show in Figure 4.5 is not a correct example since it is not a variable initialization error. Thus it is not a real false negative example for the automatic inspection but a real false positive example for the manual inspection. Furthermore, some unidentified CCES error examples may also be ignored by the ACIT and these error examples require a better tool to discover them.

<u>False positive for the automatic inspection</u>

The original data obtained from the ACIT contained several false positive examples, so a manual review of the result from the automatic inspection was conducted. Even though these false positive examples were excluded in the analysis, they can still improve the ACIT in future work. One of the main reasons for these false positives is that duplicated error examples were identified by the ACIT. A same change due to the same type error was identified several times in one block of code. Details of the number of identified examples and number of correct examples are shown in Table 4.7.

Error type	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
Original	2	57	42	251	129	634	23	9	446	0
Corrected	2	57	42	251	129	595	23	9	446	0
Ratio (Cor/Ori)	100%	100%	100%	100%	100%	94.1%	100%	100%	100%	N/A

Table 4.7 Number of original examples and corrected examples identified by the automatic inspection

Table 4.7 shows that false positive examples were identified for error *null pointer* E6. The false positive examples are all duplications due to the shortcoming of the design of the ACIT. In each block of code change, a statement can appear many times in the block. Thus the changes made on this statement in the block should be considered as one change (see Figure 4.6).

Original code	`buffer.append("<isEpisode>"+video.getEpisodic()+"</isEpisode>\n");` `buffer.append("<isEpisode>"+video.getEpisodic()+"</isEpisode>\n");`
Changed code	`if(video.getEpisodic()!=null)buffer.append("<isEpisode>"+video.getEpisodic()+"</isEpisode>\n");` `if(video.getEpisodic()!=null)buffer.append("<isEpisode>"+video.getEpisodic()+"</isEpisode>\n");`

Figure 4.6 Code fragments for project *galleon*

The false positive examples were excluded in the analysis of results, so the impact of false positive is in a low level. Additionally, it still remains a possibility that some false positive examples were not identified in the examples of the CCES errors.

4.4. Chapter summary

In this chapter, the profile of two project sample sets was compared and the results are interpreted. The comparison between the two sample sets showed that the sample for the automatic inspection is different from the manual inspection. Even though most types of CCES errors were found by the two inspections, several drawbacks were identified regarding the manual inspection. As a result, the result from the manual inspection will not be included in the latter analysis. The result from the automatic inspection will be analyzed in next chapter.

Chapter 5 Analysis of Results from the Automatic Inspection

5.1. Introduction

This chapter will describe and analyze the results of the automatic inspection on 45 projects, and identify correlations between the ten types of CCES errors and projects. After comparing the correlations between GUI and non-GUI code, the frequently identified errors will be discussed.

The automatic inspection examines the ten types of CCES error in the project source code between GUI and non-GUI code. The automatic inspection tool processed the source code of 45 projects written in Java. 11 projects have no single example of any of the ten common coding errors. To highlight the results of identified errors, the results from the 11 projects are excluded. At least one of the ten common coding errors is identified in the source code of other 34 projects hence the actual number of project considered by this chapter is 34. The automatic inspection identified nine types of the CCES errors but found no identification for one type.

The outline of the remainder of this chapter is as follows. Section 6.2 will compare the distribution of all ten types of coding error between GUI and non-GUI code in order to identify projects with frequent errors in GUI and non-GUI code. Section 6.3 will analyze each of the ten types of CCES error individually so as to verify the type and number of coding errors in GUI and non-GUI code. Section 6.4 will attempt to identify the coexistence of different types of coding error in GUI and non-GUI code. An overview of the results is then followed in section 6.5.

5.2.   Comparison between GUI and non-GUI code for CCES

5.2.1.   Overview of correlations between errors and LOC over GUI and non-GUI code

Even though the nine types of CCES error are not unique, the frequency of errors appears to be dissimilar between GUI and non-GUI code. To address the difference, the correlation between the errors and the projects is investigated. The Pearson correlation coefficient is conducted on the number of errors and LOC over GUI and non-GUI code. The values are shown in Table 5.1.

	GUI errors	non-GUI errors	GUI LOC	non-GUI LOC
GUI errors	1			
non-GUI errors	0.0667	1		
GUI LOC	0.6959	0.1406	1	
non-GUI LOC	0.0131	0.6585	0.0996	1

Table 5.1Matrix of the Pearson correlation for LOC and errors

In Table 5.1, two correlations are found to be statistically significant:

- GUI LOC and GUI errors (the critical value is 0.4357 for size 34 at level p< 0.01);
- GUI LOC and non-GUI errors (the critical value is 0. 4357 for size 34 at level p< 0.01).

Based on the correlation between the common coding errors in GUI and GUI LOC, it is shown that the coding errors in GUI can be reflected by GUI LOC. A large GUI LOC can indicate a large number of coding errors. In addition, Table 5.1 shows that there is also a significant correlation between the frequency of CCES errors in non-GUI code and non-GUI LOC. A comparison and analysis between GUI and

non-GUI will be provided in next section. Moreover, there is no significant correlation between GUI LOC and non-GUI LOC in the 34 projects. Thus there is no proportional relationship between GUI and non-GUI LOC.

5.2.2. Comparison between 12 types of project over GUI and non-GUI code

The statistical results show that not only the correlation between the frequency of CCES errors and GUI LOC is significant, but also the frequency of CCES errors and non-GUI LOC is significant. In order to investigate the difference between GUI and non-GUI code, more project characteristics need to be considered, such as the project type.

The 34 projects can be categorized into 12 different types (see Table 5.2) according to their description provided by SourceForge. To compare the differences between the 12 types of project in GUI and non-GUI code, the Z-Score of error frequency for each type of project is calculated and compared. In mathematical statistics, a random variable X is standardized using the theoretical mean and standard deviation:

$$Z = \frac{X - \mu}{\sigma}$$

where μ is the mean and σ is the standard deviation. In Table 5.2, X represents the raw score of GUI error and non-GUI error, μ is the mean of the number of the coding errors in GUI code and non-GUI code, and σ represents the standard deviation of the number of coding errors in GUI code and non-GUI code.

Comparing the values of the Z-Scores for GUI and non-GUI code, eight Z-Score values for GUI are greater than the corresponding values for non-GUI code. In other words, more coding errors were identified in GUI code than non-GUI code for the eight types of project (framed in Table 5.2). The ranking of the eight types of project

are:

1. Games/Entertainment;

2. Format and Protocol;

3. Communication;

4. Office/Business;

5. Software Development;

6. Desktop Environment;

7. Education;

8. Text Editor.

Project type	Number of projects	GUI errors	non-GUI errors	Z-Score for GUI (GZ)	Z-Score for non-GUI (NZ)	GZ−NZ
Games/Entertainment	5	101	84	2.070	-0.122	2.192
Format and Protocol	1	35	30	0.109	-0.588	0.697
Communications	4	36	34	0.139	-0.554	0.692
Office/Business	1	24	38	-0.218	-0.519	0.301
Software Development	8	95	284	1.891	1.603	0.288
Desktop Environment	1	5	0	-0.782	-0.847	0.065
Education	2	4	3	-0.812	-0.821	0.009
Text Editors	2	6	10	-0.753	-0.761	0.008
Internet	1	1	1	-0.901	-0.838	-0.063
Scientific/Engineering	3	18	157	-0.396	0.508	-0.904
Database	2	36	228	0.139	1.120	-0.981
Multimedia	4	15	309	-0.485	1.819	-2.304
Average (μ)		31.33	98.17			
STDEV (σ)		33.66	115.91			

Table 5.2 Z-Score for errors between GUI and non-GUI code

In the eight types of project, the Games/Entertainment projects have the largest

variance between their GUI and non-GUI code. Hence, this type of project is regarded to have a higher frequency of identified errors in GUI than non-GUI code. The reason is that this type of project has more GUI code compared with other types. The average ratio between GUI LOC and non-GUI LOC is 0.51, and the ratio for Games/Entertainment project is 0.92. The type Games/Entertainment has the highest ratio between GUI code and non-GUI LOC in the 12 types of projects. Thus the largest number of errors was identified in the type of Games/Entertainment GUI code. Furthermore, the LOC is useful to indicate the possible identification of errors. So the more code there is the greater the change that errors will be identified.

The statistical data may not precisely reflect the actual situation of the eight types of project as some sample sizes are small. For instance, the sample of 'Format and Protocol' type only contains one project, and the sample size of 'Desktop Environment' type is also one. The difference between GUI and non-GUI code is too small for the type 'Education' and 'Text Editor'.

Exceptional cases

After investigating possible types of project containing more errors in GUI, a comparison of the distribution of errors between GUI and non-GUI code will be provided. Two scatter charts (Figures 5.1 and 5.2) represent the distribution of errors. The following two figures show that several data points are exceptional since they are divergent from the trend.

In the scatter plot shown below, the exceptional point is circled in red. These points in the scatter plots are the most influential point in determining the correlation p value overall. Points in the upper left area of plot represent high number of identified errors in small LOC and points in the bottom right area of plot represent small number of errors in large LOC.
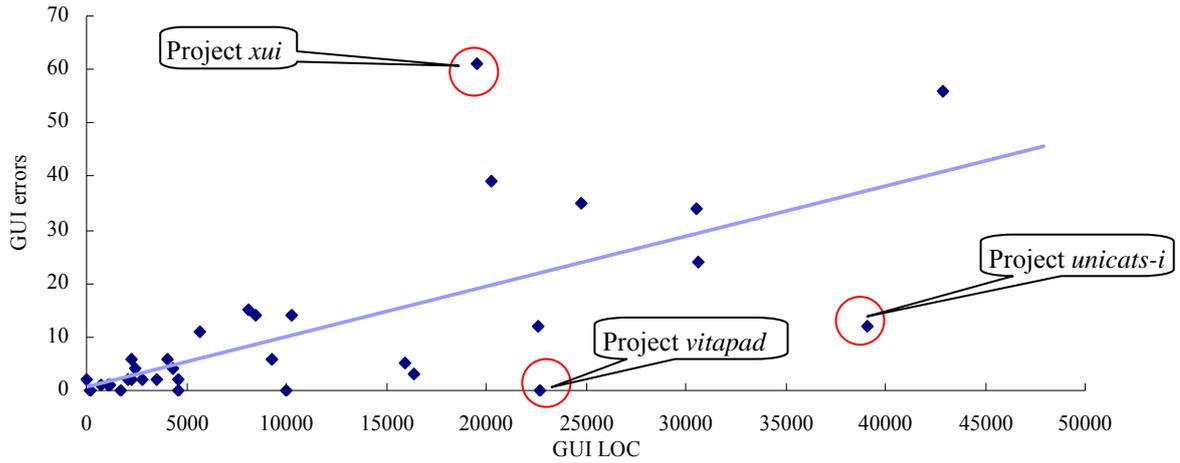
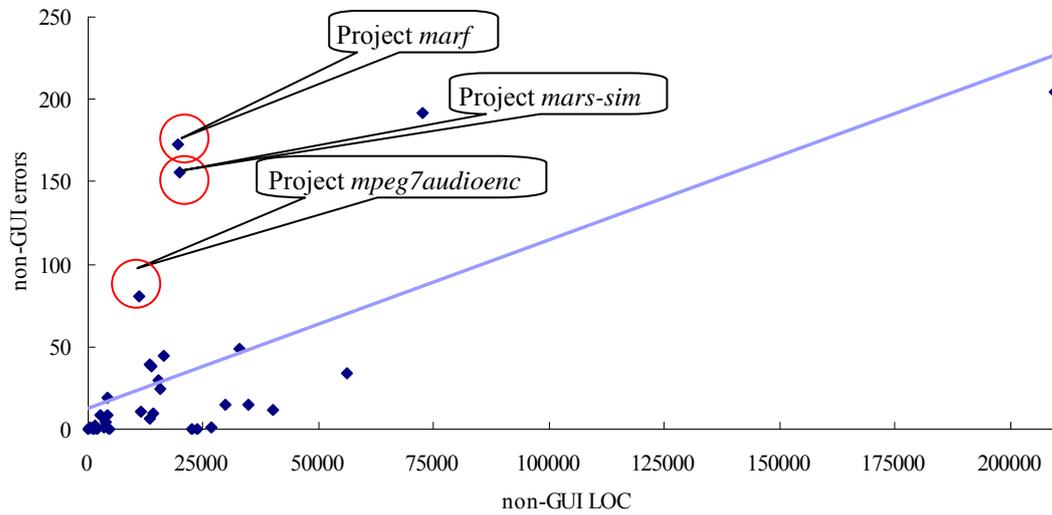**Figure 5.1 Correlation of GUI errors and GUI LOC**

Figure 5.2 Correlation of non-GUI errors and non-GUI LOC

In fact, some of the data points are divergent from the trend and they have a significant effect on the Pearson r value. The criteria to identify exceptional points are: relatively small amount of LOC but large amount of coding errors, large amount of LOC but small amount of coding errors. Three projects are considered as exceptional for GUI and three other projects are considered as exceptional for non-GUI for the same reason. A case study of the most exceptional projects is followed. The analysis mainly focuses on the projects represented by the points in the upper left area since a larger proportion of identified errors are identified.

	Project name	Errors (GUI)	Errors (non-GUI)	GUI LOC	non-GUI LOC	Project type	Number of developers	Download times
Exceptional projects for GUI	xui	61	39	19558	13423	SD	17	102935
	vitapad	0	1	22729	26789	S/E	1	371
	unicats-i	12	192	39068	72554	SD	15	32
Exceptional projects for non-GUI	marf	1	173	1184	19583	M	25	2360
	mars-sim	15	156	8099	19828	S/E	18	66202
	mpeg7 audioenc	0	80	224	10973	M	11	4244
Average for 34 projects	N/A	9.82	35.97	10845	20940	N/A	9	89877

*SD: Software Development   *S/E: Science/Engineering   *M: Multimedia

Table 5.3 GUI and non-GUI exceptional projects

Table 5.3 shows the relevant information about the six exceptional projects for GUI and non-GUI code. The three exceptional projects for GUI are *xui, vitapad* and *unicats-i*. The three exceptional projects for non-GUI are *marf, mars-sim* and *mpeg7audioenc*.

1. Project *xui*

The project *xui* is an XML RIA platform for building smart applications. Figure 6.1 shows that it has the greatest number of errors in its GUI code compared with the 34 other projects. Furthermore, the error '*null pointer*' (E6) is the most frequent error for project *xui*, and the largest value for E6 is the main reason why this project is identified as exceptional. This E6 null pointer error will be discussed further in section 5.3. From the perspective of the project characteristics, project type and download times may be the reason for the frequent occurrence of errors. The project *xui* is a tool to develop Java User Interfaces; and this program contains large GUI LOC. As shown in Table 5.3, the difference between GUI LOC of this project (19558)

is larger than the average LOC of the whole sample (10845). However, this is not the only factor that influences the results because many other projects are not considered exceptional even if they have greater GUI LOC, for instance, the project *ganttproject*. Additionally, the project *xui* is very popular for download (it has been downloaded 102,935 times), and thus more errors were reported and consequently being corrected, especially for the error *null pointer,* which can be easily identified by users.

2. Projects *vitapad and unicats-i*

Projects *vitapad* and *unicats-i* have a large amount of code but a relatively low frequency of errors in GUI code. Figure 5.1 shows that the two projects significantly affect the trend line.

The project *vitapad* is a tool to visualize biological pathways, and it has been downloaded only 317 times. This project has few coding errors identified in both GUI and non-GUI code. Without the support of an adequate number of error examples, it is difficult to analyze why the CCES is seldom identified in this project. Nevertheless, the infrequent download may explain why the smallest number of coding errors was identified in this project. Many latent errors may not have been exposed by users.

The project *unicats-i* is a tool to help extract information from the internet. This project is regarded as less problematic for GUI code, because this project has a large GUI LOC but a relatively small amount of coding errors in GUI code. For the same reason described above, the low download counts lead to fewer errors being reported.

3. Projects *mpeg7audioenc marf* and *mars-sim*

Even though the correlation between the number of CCES errors in non-GUI code and non-GUI LOC is significant, the three projects are still regarded as exceptional for non-GUI code. Without the negative influence of these three projects, the correlation is more significant. The details of each type of CCES errors for the three projects are shown in Table 5.4.

Project	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	Total
mpeg7audioenc	0	0	0	75	3	1	0	1	0	0	80
marf	0	0	0	4	7	6	1	0	155	0	173
mars-sim	0	18	2	92	0	43	1	0	0	0	157
Average for non-GUI	0	0	0	39.5	5	3.5	0.5	0.5	77.5	0	35.11

Table 5.4 Ten types of error in non-GUI code for three projects

The project *mpeg7audioenc* is an encoder to describe audio content. It has the smallest amount of GUI LOC in the 34 projects. Within such a small amount of GUI code, no single example of the coding errors was identified. The results show that the number of non-GUI LOC for the project *mpeg7audioenc* is below the average level, but the frequency of errors in non-GUI is much greater than average. More specifically, the most frequently identified error is *variable type* (E4) in non-GUI code of the project *mpeg7audioenc*. The E4 error will be discussed in section 5.3.

The project *marf* is a general cross-platform framework with a collection of algorithms for audio and natural language analysis and recognition. The key factor to make it exceptional is the E9 *multithread* error because the largest amount of E9 is identified in non-GUI code of this project. This large number of *multithread* errors was identified in 12 Java files, and all these E9 were created by one developer. Without the support of personal information about the developer, no specific conclusion can be drawn. However, the rational explanation is that the developer may lack experience of multi-threaded programming.

The project *mars-sim* is an application for Science/Engineering. Table 5.4 shows that the most frequently identified coding errors are *==instead of .equals()* (E2), *variable type* (E4) and error *null pointer* (E6). After checking the source code of the project *mars-sim*, it is identified that E2 *==instead of .equals()* error, E4 *variable type* error and E6 *null pointer* errors were created by different programmers in different classes.

Therefore, the E2, E4 and E6 errors were common in non-GUI of project *mars-sim*, and the errors were not caused by an individual programmer. The main reason was due to a small number of modification had a significant change impact throughout classes in an early stage of its development. For example, the E4 error was widely spread across the early versions of non-GUI code, and all E4 errors followed the same form that the type *int* (instead of correct type *double*) is wrongly selected. Most incorrect type variable or method definitions were found in several different classes instead of a single class.

The above analysis indicates that the project *xui* is the most interesting exceptional case for GUI code. Unlike other projects, this project has a larger number of errors identified in their GUI components. The overall number of errors in this project greatly exceeds the average level. A case study of the project *xui* is analyzed below.

Case study of exceptional case xui

From the data shown in Table 6.5, it is known that the one project has more GUI components than Non-GUI components.

Project	LOC in GUI	LOC in non-GUI	LOC
xui	19558	13423	32981

Table 5.5 Project *xui* GUI and non-GUI LOC

By matching errors lifetime with whole project lifetime, the changed LOC, errors creation of removal, and project timelines were visualized into several figures. In these figures, each bar represents an error lifetime, the vertical straight line represent versions release time, and the other two lines in the background represent total LOC changes, including addition, deletion and modification. Note in Figures 5.3 and 5.4 that the bars are only related to the x axis/timeline but not y axis/size of changes. Using the records of each error found in GUI code, eight problems were classified into two figures. The most frequently identified error *null pointer* is in Figure 5.3, and

all other types of errors were shown in Figure 5.4. The analysis continued sequentially through.

The error *null pointer* is the most frequently identified error in GUI code of project *xui*. The life of all identified error *null pointer* is shown in Figure 6.3.
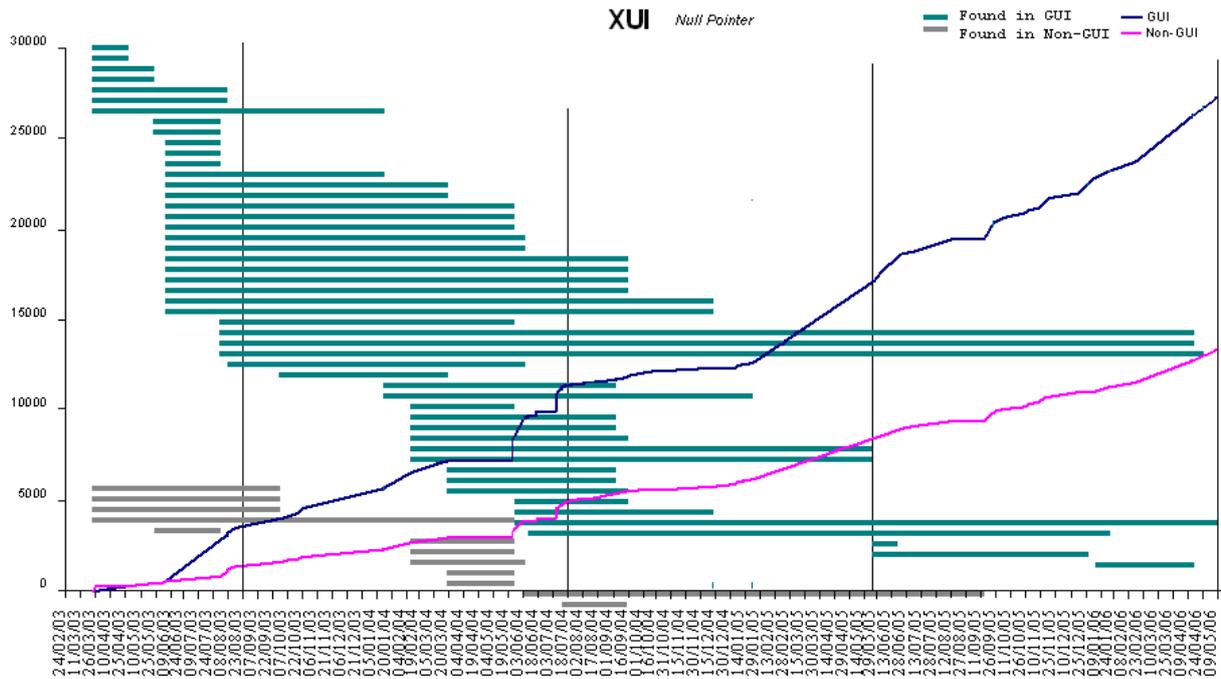

Figure 5.3 The life of error *null pointer* in project *xui*

According to project xui source code records, two sets of code duplication were identified in GUI and none was found in Non-GUI. One set contains eight duplicated errors and another contains two errors. Figure 5.3 shows that several cluster of bars with the same length. These bars represents that same changes were made in different files. The longest bars represent that certain errors were injected in earlier stage of software life but identified in very later stage. Figure 5.3 shows that the number of errors in GUI keeps decreasing even GUI LOC is increasing. Only a small number of errors are inserted into GUI code along with the project becomes mature. Moreover, the correction period took much shorter than before. The correction of errors in non-GUI code also kept decreasing. In fact, the changes made in non-GUI code were also related to GUI. Same errors were corrected not only in GUI code but also in

non-GUI code. In other word, the correction and identification of errors in GUI code help the correction of errors in the whole project.

The code duplication is the main cause for the frequency of error identification. The automatic inspection result also shows that most of the frequently identified errors are actually a same error. The reasons include code clones, high coupling, and others. These reasons are not unique for GUI. They also apply for general coding behaviors. Moreover, the life of error reflects the characteristics of error identification in GUI code.
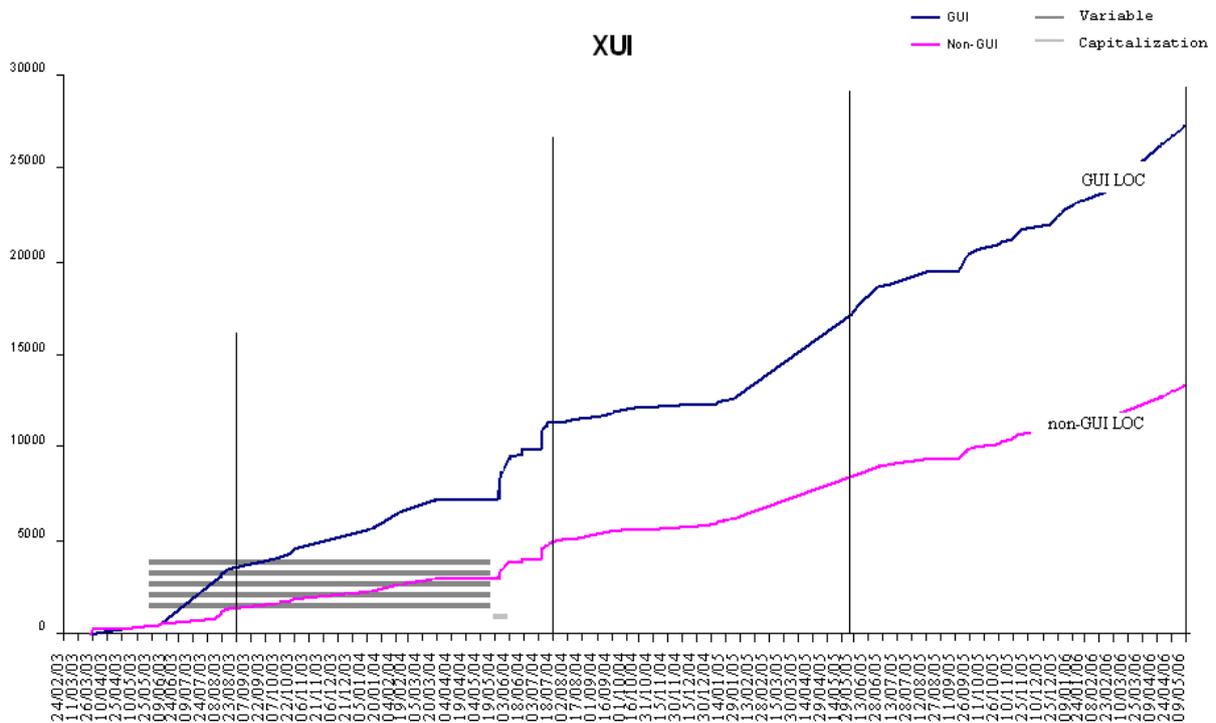
Figure 5.4 The life of errors in project *xui*

In Figure 5.4, all types of errors do not distribute successively with the development of projects. Most of these problems cases were created and corrected without any subsequence. This result does not provide evidence to indicate characteristic forces behind these problems. The block of errors in project *xui* is the variable type problem found in non-GUI components based on the records. For example, variable *private double min, max;* was created in revision 1.3 of

xui/src/net/xoetrope/awt/XMetaContent.java on 22nd May 2003, and modified to *private float min, max;* in revision 1.11 on 24th May 2004. The code change is caused by that the initial variable type is not suitable; additionally, the same reason causes the identification of the rest of same type of errors.

5.2.3. Correlation between errors and open source features

All projects examined by the automatic inspection were selected from SourceForge, and the identification of errors was influenced by specific features of Open Source Software, such as download counts and project life [Crowston2006]. To examine whether these features have an influence on the CCES errors, the Pearson r is calculated and shown in Table 6.6.

	GUI errors	non-GUI errors
Download	0.2903	0.0076
Developer	0.3974	0.3550
Project life	0.4024	0.2536

Table 5.6 Correlation between errors and open source features

Table 6.6 shows that there are four significant correlations for the 34 projects at p<0.10, p< 0.05, and p<0.02. From this point of view, the features of OSS may have an influence on GUI code, especially the influence of download count, length of project life and number of developers on GUI code. The number of developers is correlated to both GUI errors and non-GUI errors, and hence it is shown that team size may affect the identification of coding errors. Similar surveys have been conducted by other researchers. Crowston et al [Crowston2006] argued that development team size is highly correlated to the frequency of errors and the download count is also highly correlated to the number of identified errors. However, the correlation between download count and non-GUI errors is not significant (see

Table 6.6). A possible explanation is that latent errors can hide in code for a long period if the error has not been found by users. Errors occurring in GUI components are more likely to be found by users if GUI components are used frequently. Moreover, the project life is also correlated to the frequency of identified errors in GUI code. The total number identified errors increase as the OSS project continues to develop.

5.2.4. Summary

In Section 5.2, the correlation between the frequency of coding errors and the LOC was compared in GUI and non-GUI code. By analyzing statistical results, more coding errors were found in GUI than non-GUI code for eight types of project.

However, the number of projects for the types Office/Business, Format and Protocol, and Desktop Environment were too small to justify their accuracy. The results for Game/Entertainment projects may be more representative for its own type as it has more projects than other types of project.

Additionally, six exceptional projects were identified, three projects were exceptional for GUI code, and three other projects were exceptional for non-GUI code. After analyzing these exceptional projects, it was shown that certain coding errors are related to certain developers. The CCES was not identified to be located in a specific class or in the particular structure of GUI or non-GUI code. Moreover, after comparing the two groups of the exceptional projects, it was proposed that the team size, length of project life and download count may have an influence on the frequency of errors in GUI code.

5.3. Comparison of each type of error between GUI and non-GUI code

This section will examine each type of error in GUI and non-GUI code. This comparison attempts to identify possible project types where more CCES errors were identified. Table 5.7 displays the correlation of each error and LOC in GUI and non-GUI code.

Error type	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
GUI LOC	N/A	0.4300	0.1165	0.2595	0.4174	0.6320	0.5647	N/A	0.1893	N/A
non-GUI LOC	0.2419	0.1564	0.3157	0.0014	0.2616	0.3882	0.1772	0.1057	0.7130	N/A

Table 5.7 Comparison of correlation coefficients for ten types of error between GUI and non-GUI code

Comparing the values for each error, four types of error are significantly correlated to GUI LOC for the sample size of 34 at level $p < 0.01$, $p < 0.02$ and $p < 0.05$. Three types of error have significant correlation with non-GUI LOC for the sample size of 34 at $p < 0.01$ and $p < 0.05$, $p < 0.10$.

Table 5.7 shows that six errors are not significantly correlated with GUI LOC and seven types of error are not correlated with non-GUI LOC. Therefore, the LOC does not have proportional correlation on most individual type of error. The analysis will focus on the comparison of frequency of identification of each type of error between GUI and non-GUI code. Furthermore, the analysis will try to identify GUI error features, and explore underlying reasons.

5.3.1. CCES errors with significant correlation in GUI or non-GUI code

Table 5.7 shows that four types of errors (E2 *==instead of .equals()* error, E5 *uninitialization* error, E6 *null pointer* error, and E7 *blank exception handler* error) are more significant in GUI code and three type of errors in non-GUI code (E3 *capitalization* error, E6 *null pointer* error and E9 *multithreaded* error). The project types will be considered and studied to identify where these errors were identified. Additionally, exceptional projects falling outside the typical trend will be examined.

Error 2 (E2) "==" instead of ".equals()"

A comparison of values for E2 *==instead of .equals()* error indicates that E2 is more frequent in GUI code than in non-GUI code, but a comparison of the absolute values for correlations is not sufficient to verify this hypothesis. More information needs to be considered (in Table 5.8).

Before the analysis of the correlations commences, the E2 error domain needs to be identified; in other words, where the identification of E2 errors frequently occur. Data from both GUI and non-GUI is standardized to be compared by using Z-Score. Table 5.8 shows the statistical results for the type of E2 error.

Comparing the Z-Score for E2 errors between GUI and non-GUI code, more E2 are identified in GUI code in five types of project (in Table 6.8); the five project types are 'Office/Business', 'Communication', 'Format/Protocol', 'Game/Entertainment', and 'Database'. In particular, the results within the 'Games/Entertainment' and 'Communication' categories are likely to be more reliable compared to the three other types as they contain more projects than the three other types.

Project type	Number of Projects	E2 in GUI	E2 in non-GUI	Z-Score for E2 in GUI(GZ)	Z-Score for E2 in non-GUI(NZ)	Difference of Z-Score (GZ−NZ)
Office/Business	1	6	0	1.91	-0.46	2.37
Communications	4	5	0	1.46	-0.46	1.92
Format and Protocol	1	4	0	1.01	-0.46	1.47
Games/Entertainment	5	3	1	0.56	-0.31	0.87
Database	2	2	0	0.11	-0.46	0.57
Desktop Environment	1	0	0	-0.79	-0.46	-0.33
Education	2	0	0	-0.79	-0.46	-0.33
Internet	1	0	0	-0.79	-0.46	-0.33
Multimedia	4	0	0	-0.79	-0.46	-0.33
Text Editors	2	0	1	-0.79	-0.31	-0.48
Software Development	8	1	16	-0.34	1.91	-2.24
Scientific/Engineering	3	0	19	-0.79	2.35	-3.14
Average		1.75	3.08			
STDEV		2.22	6.78			

Table 5.8 Statistical Z-Score results for E2 ==*instead of .equals()* error between GUI and non-GUI code

Additionally, some of the values for the Z-Score are the same, for example, the Z-Score for the 'Multimedia' type (-0.79) is equal to the Z-Score for the 'Internet' type (-0.79). From the third column (***E2 in GUI***) in Table 5.8, it is shown that these two types of project have the same frequency of E2 found in GUI code. Based on the Z-Score formula (see section 5.2), the same values of Z-Score are produced. Therefore, the equivalence of many Z-Score does not address interesting issue for the two types of project.
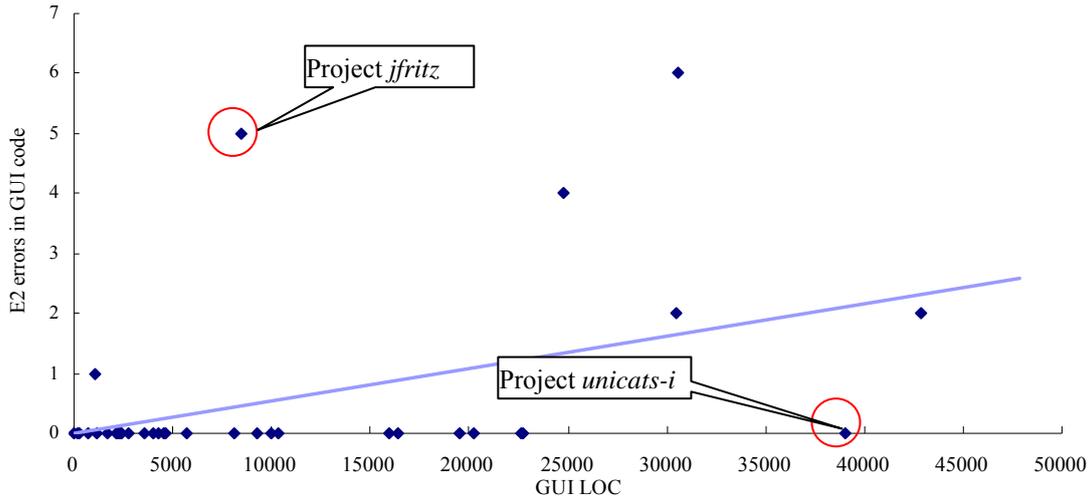
Figure 5.5 Correlation of LOC and error *"= ="* instead of *".equals()"* (E2) in GUI code

After examining the project types where E2 *==instead of .equals()* errors were frequently identified, a comparison of E2 errors between GUI LOC and non-GUI LOC is carried out. The distributions are displayed in Figures 5.5 and 5.6.
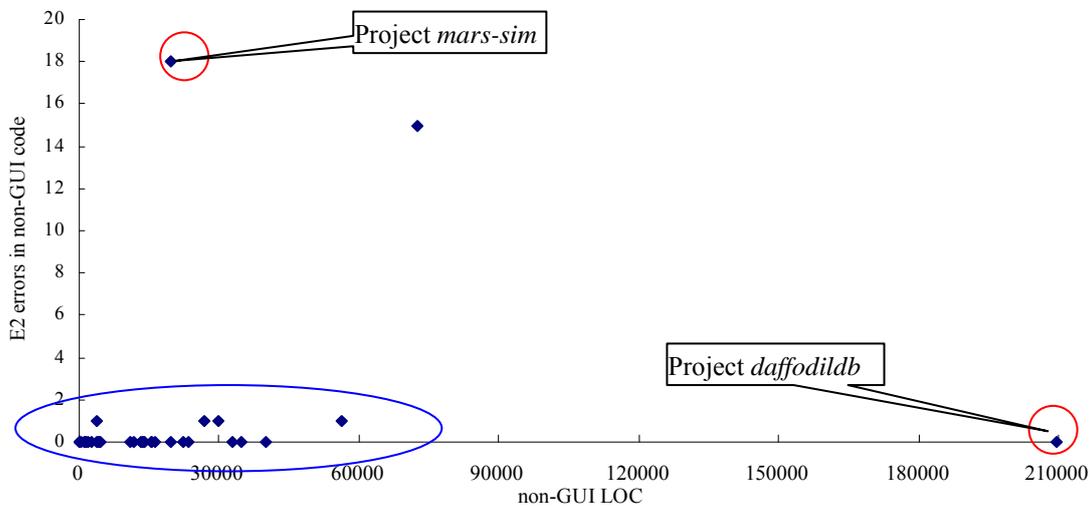

Figure 5.6 Distribution of error *"= ="* instead of *".equals()"* (E2) in non-GUI code

A comparison of the distribution of E2 *==instead of .equals()* errors between GUI and non-GUI code shows that the data points converge more closely to the regression line for GUI in Figure 5.5, and most data points are also located in one area (blue oval in Figure 5.6). The distributions of E2 in GUI and non-GUI code show that the identification of E2 error is in a low level for both GUI and non-GUI code. However,

there are four projects which have a significant negative effect on the correlation value. The study of exceptional projects may help identify particular features that significantly affect the identification of E2 error.

Figures 5.5 and 5.6 identify several potential exceptional data points for GUI and non-GUI code. After recalculating the correlations, two data points (circled in Figure 5.5) are regarded as exceptional projects for GUI code because they have the greatest negative effect on decreasing the value of the correlation. Two other data points (circled in Figure 5.6) are considered as exceptional projects for non-GUI code. The exceptional projects for GUI are project *jfritz* (in which more E2 *==instead of .equals()* errors were identified in GUI code) and project *unicats-i* (in which fewer E2 errors were identified in GUI code). The exceptional projects for non-GUI are project *mars-sim* (a large number of E2 error was identified in non-GUI code) and project *daffodildb* (which has fewer identification of E2 error in non-GUI code). The project profiles are presented in Table 5.9 and will be discussed below.

Project Name	E2 in GUI	E2 in non-GUI	GUI LOC	non-GUI LOC	Project type	Download counts	Number of developers	Life time (in month)
Jfritz	5	0	8469	11552	Communications	11844	8	14
unicats-i	0	15	39068	72554	Software Development	32	15	39
mars-sim	0	18	8099	19828	Scientific/Engineering	66202	18	26
daffodildo	0	0	38	209477	Database	13235	10	18

Table 5.9 Profile of four exceptional projects for E2 *==instead of .equals()* error

Project jfritz

The project *jfritz* is a communication tool, so interactions between the user and GUI may be very frequent. As a consequence, more errors were identified. In fact, all E2 *==instead of .equals()* errors were created by one programmer in two Java GUI classes. The two classes have no logical relationship to one another. As a result, poor programming skill is likely to account for the exceptional nature of this project. In this

case the programmer has made the same mistake again and again by confusing the comparison between **Strings**, an example is shown in Figure 5.7.

Problematic code	Changed code

`else if(type == "ping")`
 	
`returnValue = Rv.CONFIRM;`
	`else if(type.equals("ping"))`
	
	`return reactOnPing(message);`

Figure 5.7 Code fragment from project jfritz

Project unicats-i

The project *unicats-i* is a software development tool. E2 *==instead of .equals()* error was not frequently identified in GUI code but frequently identified in non-GUI code (see Table 5.8). Furthermore, this project was seldom downloaded by users, and thus E2 may remain unidentified in GUI code. There are two possibilities that cause fewer E2 errors to be identified in GUI code:

- GUI code is well implemented in this project;
- The project has a short history and small count of downloads, so many potential E2 errors exist but still remain undiscovered.

Project mars-sim

The project *mars-sim* has many identifications of E2 in non-GUI code. This project has the largest frequency of E2. Further investigation of the code identified that all E2 errors were created by one programmer in eight different Java non-GUI classes, and all these errors were similar. As a result, the large amount of E2 errors was caused by the confusion when comparing Objects. The problematic code is reproduced again and again when the comparison occurred. Figure 5.8 shows an example of E2 that was created by the programmer.

```
if (tempPerson.getLocationSituation() == Person.INVEHICLE)

allDisembarked = false;



if (tempPerson.getLocationSituation().equals(Person.INVEHICLE))

allDisembarked = false;
```

Figure 5.8 Code fragment from project mars-sim

Project daffodildb

The project *daffodildb* is very different from other projects as no E2 *==instead of .equals()* errors were identified in its non-GUI code. This project has the largest non-GUI LOC in the sample of the 34 projects, and it has been downloaded more than 10,000 times in 18 months (see Table 5.9). The possible reason is that the programmers were experienced, so they had an awareness of the E2 error when comparing two objects. Unfortunately, the information about the programmers was not available, and no further verification can be made.

Error 3 (E3) capitalization

This type of error was frequently identified in non-GUI code but not frequently identified in GUI code. The average frequency of identification is less than one in GUI code per project. Table 5.10 shows the comparison between GUI and non-GUI.

However, Table 5.10 shows that the identification of E3 *capitalization* error is more frequent in GUI code than non-GUI code for the project type 'Game/Entertainment', 'Desktop Environment', 'Format and Protocol', 'Internet', 'Multimedia', 'Office/Business', and 'Text Editors'. The Z-Score for the 'Game/Entertainment' type is more representative since the frequency of identified E3 error (the value is three) examples for this type is not equal to zero.

Project type	Number of Projects	E3 in GUI	E3 in non-GUI	Z-Score for E3 in GUI(GZ)	Z-Score for E3 in non-GUI(NZ)	Difference of Z-Score (GZ−NZ)
Games/Entertainment	5	3	0	2.5	-0.58	3.08
Desktop Environment	1	0	0	-0.5	-0.58	0.08
Format and Protocol	1	0	0	-0.5	-0.58	0.08
Internet	1	0	0	-0.5	-0.58	0.08
Multimedia	4	0	0	-0.5	-0.58	0.08
Office/Business	1	0	0	-0.5	-0.58	0.08
Text Editors	2	0	0	-0.5	-0.58	0.08
Education	2	0	1	-0.5	-0.38	-0.12
Scientific/Engineering	3	0	2	-0.5	-0.19	-0.31
Database	2	2	14	1.5	2.11	-0.61
Communications	4	0	6	-0.5	0.58	-1.08
Software Development	8	1	13	0.5	1.92	-1.42
Average		0.5	3			
STDEV		1	5.2			

Table 5.10 Statistical results for E3 *capitalization* error between GUI and non-GUI code

E3 *capitalization* error is significantly correlated to non-GUI LOC (see Figure 6.9).The two exceptional projects are *jabref* and *daffodildb*. The project *daffodildb* is also exceptional for E2 *==instead of .equals()* error in non-GUI code. However, corresponding code investigation does not identify particular features link to the identification of E3 in non-GUI code. Figure 6.9 shows that E3 was not very frequently identified across the 34 projects since E3 was identified in only five projects.
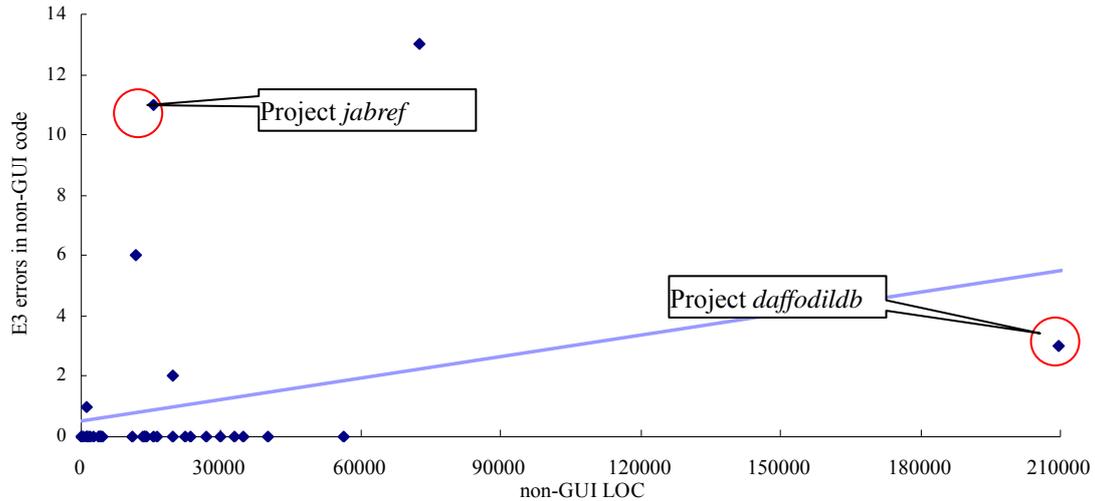
Figure 5.9 Correlation of LOC and error '*capitalization*' (E3) in non-GUI code

The project *jabref* is another exceptional project. 11 examples of E3 *capitalization* error were identified in the project *jabref*. These examples of E3 were often the same type of method. For example, a method name `fixAuthor_lastnameFirst()` was changed to `fixAuthor_lastNameFirst()`, so all corresponding method names in other source files were changed. Thus, the number of identification in this project is greater than other projects. This also occurred for other types of errors.

Error 5 (E5) uninitialization

The correlation between GUI LOC and E5 *uninitialization* errors is significant at the p< 0.02 level (see Table 5.7), and no significant correlation between non-GUI LOC and E5 errors is identified. Data shows that this type of error had been frequently identified in both GUI and non-GUI code, so the identification of E5 between GUI and non-GUI code needs to be analyzed. Table 5.11 shows the detail of the Z-Score for E5 in different project types.

Five types of project had E5 frequently identified in their GUI code. These five types are 'Format/Protocol', 'Office/Business', 'Text editors', 'Multimedia', and 'Desktop Environment'. However, the three types 'Format/Protocol', 'Office/Business', and 'Desktop Environment' only have one project each, so the results of their Z-Score

may not be applicable to other projects. In addition, the identification of E5 *uninitialization* error is different from E2 *==instead of .equals()* error in terms of project types, because E5 was identified more frequently in non-GUI code than GUI code in the project types 'Games/Entertainment' and 'Communications'.

Project type	Number of Projects	E5 in GUI	E5 in non-GUI	Z-Score for E5 in GUI(GZ)	Z-Score for E5 in non-GUI(NZ)	Difference of Z-Score (GZ−NZ)
Format and Protocol	1	5	4	1.85	-0.28	2.14
Office/Business	1	4	6	1.18	-0.16	1.34
Text Editors	2	3	1	0.51	-0.47	0.98
Multimedia	4	4	18	1.18	0.60	0.58
Desktop Environment	1	2	0	-0.17	-0.53	0.37
Games/Entertainment	5	2	7	-0.17	-0.09	-0.07
Communications	4	2	9	-0.17	0.03	-0.20
Education	2	1	0	-0.84	-0.53	-0.31
Internet	1	1	0	-0.84	-0.53	-0.31
Database	2	1	1	-0.84	-0.47	-0.37
Scientific/Engineering	3	0	0	-1.52	-0.53	-0.98
Software Development	8	2	56	-0.17	2.99	-3.16
Average		2.25	8.50			
STDEV		1.48	15.89			

Table 5.11 Statistical results for E5 *uninitialization* error between GUI and non-GUI code

The comparison between E5 *uninitialization* error examples shows that most identifications of E5 errors occurred in *String* and *Boolean* variables expression in GUI code. However, no specific features of programs were found to cause E5 errors in GUI code. Exceptional projects of E5 are analyzed to explore why E5 was frequently identified in GUI code. The scatter plots of E5 *uninitialization* error and LOC are shown in Figures 5.10 and 5.11.
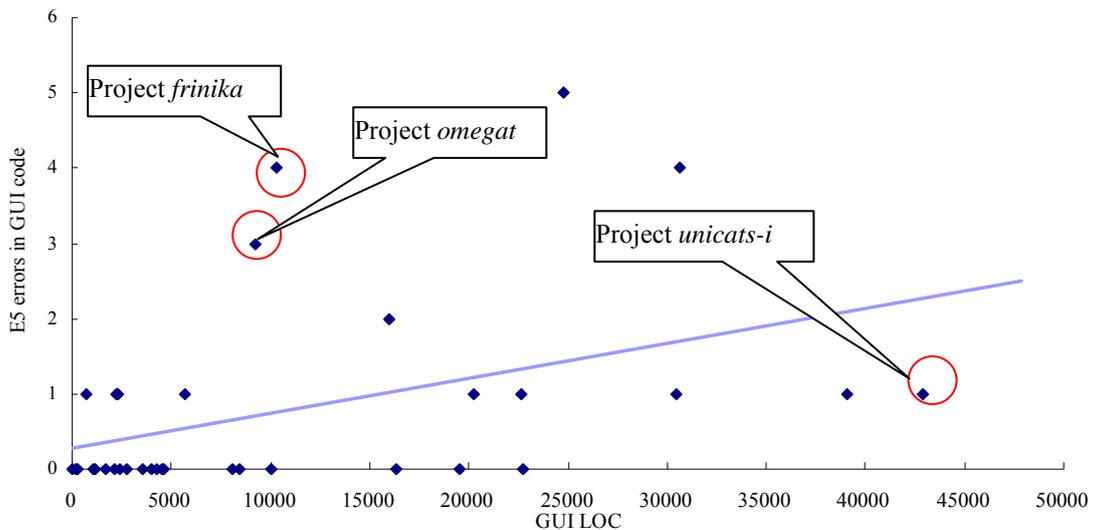
Figure 5.10 Correlation of LOC and error '*uninitialization*' (E5) in GUI code

After recalculating the correlations by removing each potential project (disjunctive data points in Figures 5.10 and 5.11), three projects are considered as exceptional for GUI, and one is exceptional for non-GUI code. The exceptional cases for GUI are the project *omegat* and *frinika*, which have a large number of identifications of E5 *uninitialization* errors in GUI code, and the project *unicats-i*, which has a small identifications of E5 errors in GUI code. The exceptional project for non-GUI is the project *daffodildb*, which has a low frequency of identified E5 in non-GUI code.

In the projects *frinika* and *omegat*, the identification of E5 *uninitialization* errors occurred in the *String* or *boolean* initialization expression. In fact, the frequent download of these two projects may cause the frequent identification of E5. For example, the project *omegat* had been downloaded 60,000 times in its two month lifetime. No other specific features or properties are found to link to the frequent identification of E5 in these two projects.
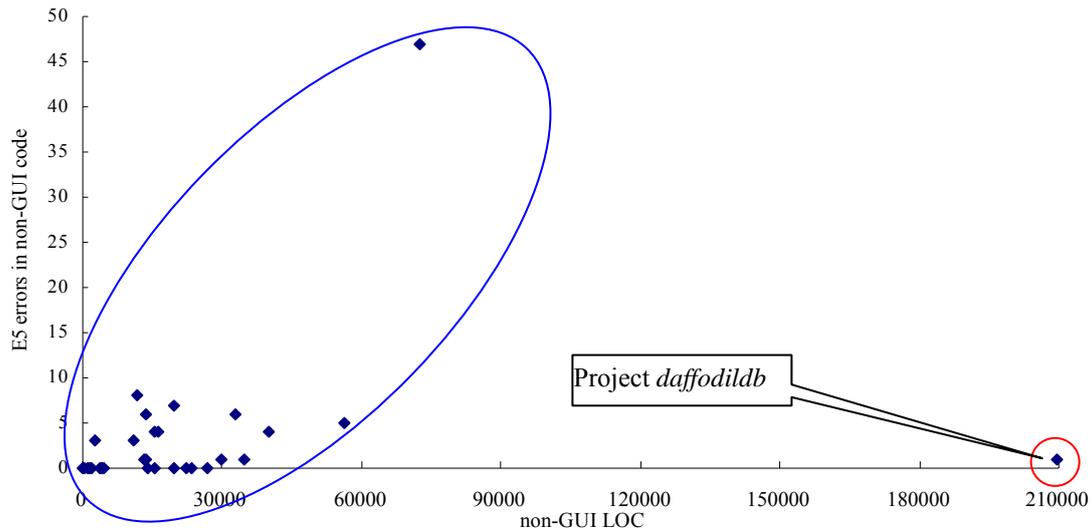
Figure 5.11 Scatter plot of error '*uninitialization*' (E5) in non-GUI code

The projects *unicats-i* and *daffodildb* have similar negative influences on the correlation value of E5 *uninitialization* error compared to E2 *==instead of .equals()* error. Thus similar characteristics may make them exceptional from others. The project *unicats-i* had a short history and small count of downloads, so a low usage of program may mean errors were yet to be identified in GUI code. Figure 5.11 shows the significant difference between the majority projects and the project *daffodildb*. This difference is also identified in other type of errors except E9 *multithreaded* error in non-GUI code. Even though the project *daffodildb* was only 18 month old, it had not only the largest non-GUI LOC but also a large count of downloads. However, the life time and download counts did not have a significant influence on the identification of E5 in non-GUI code because the two metrics are not significantly correlated to the identification of errors. A possible explanation is that the developers were experienced with most these CCES errors and seldom made mistakes overall, nevertheless, this is not able to be evaluated due to lack of information regarding the programmers background.

Error 6 (E6) null pointer

The correlation between E6 and GUI LOC is more significant than the correlation

between E6 and non-GUI LOC according to Table 5.7. This type of error is the most frequently identified error in GUI code and the second most in non-GUI code. To find out where E6 were mostly identified, the comparison of Z-Score between GUI and non-GUI is made. Details are shown in Table 5.12.

Project type	Number of Projects	E6 in GUI	E6 in non-GUI	Z-Score for E6 in GUI(GZ)	Z-Score for E6 in non-GUI(NZ)	Difference of Z-Score (GZ−NZ)
Games/Entertainment	5	71	61	1.82	0.94	0.87
Format and Protocol	1	20	13	-0.02	-0.47	0.44
Communications	4	22	22	0.05	-0.20	0.25
Education	2	3	0	-0.64	-0.85	0.21
Database	2	26	30	0.19	0.03	0.16
Desktop Environment	1	1	0	-0.71	-0.85	0.14
Internet	1	0	1	-0.75	-0.82	0.07
Office/Business	1	12	18	-0.31	-0.32	0.01
Text Editors	2	1	6	-0.71	-0.67	-0.04
Software Development	8	82	119	2.21	2.64	-0.43
Scientific/Engineering	3	10	43	-0.38	0.41	-0.80
Multimedia	4	0	34	-0.75	0.15	-0.89
Average		20.67	28.92			
STDEV		27.72	34.07			

Table 5.12 Statistical results for E6 *null pointer* error between GUI and non-GUI code

The comparison of Z-Scores between GUI and non-GUI code indicates that project types 'Games/Entertainment', 'Format and Protocol', 'Communication', 'Education', 'Database', 'Desktop Environment', 'Internet', and 'Office/Business' are the eight types of project which have greater Z-Score of E6 *null pointer* error in GUI code than non-GUI code. After checking the code where E6 errors were identified, no special features of GUI code were found to link to the frequent identification of E6 errors.

The results of E2 *==instead of .equals()* error and E5 *uninitialization* error show that the value of correlation is often significantly influenced by extreme exceptional projects, for example, the project *daffodidb* determines whether the correlation is significant for most errors. To identify the exceptional projects for the E6 error, the correlation between E6 *null pointer* error and LOC is shown in two scatter plots (Figures 5.12 and 5.13).
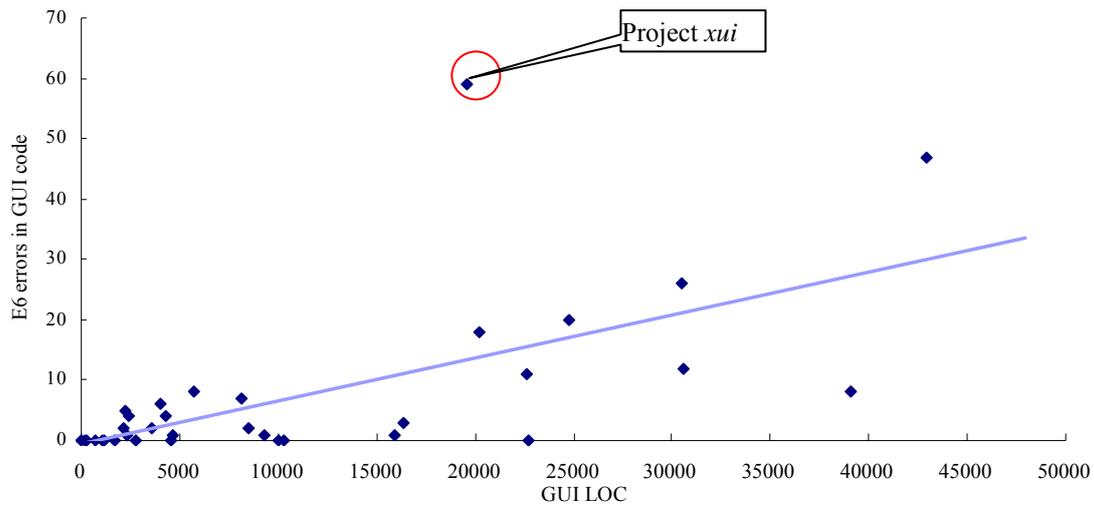
Figure 5.12 Correlation of LOC and error '*null pointer*' (E6) in GUI code
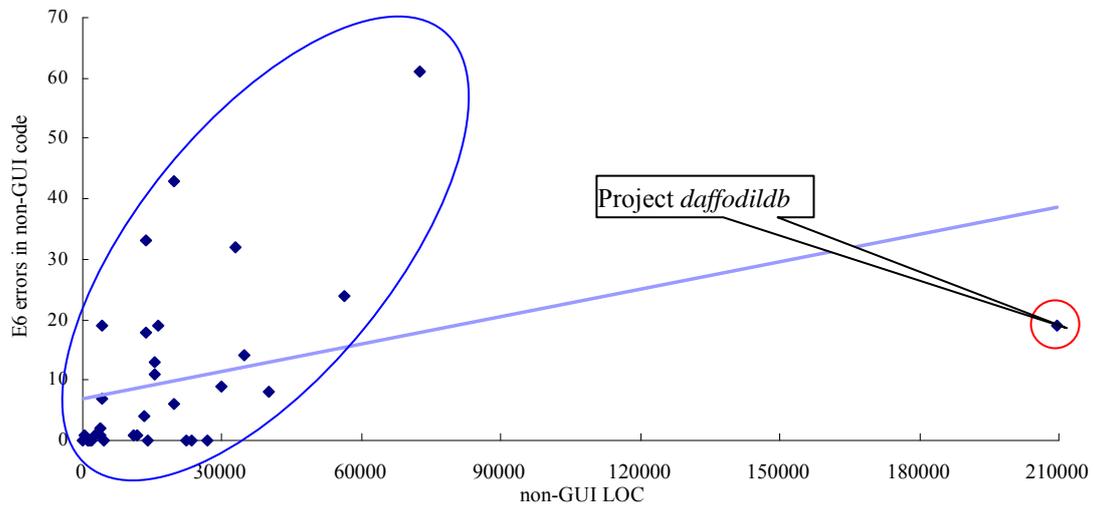
Figure 5.13 Correlation of LOC and error '*null pointer*' (E6) in non-GUI code

In Figures 5.12 and 5.13, significant numbers of data points are no longer located at the axis compared to E2 *==instead of .equals()* error and E5 *uninitialization* error.

Therefore, it shows that E6 *null pointer* error is a more frequently identified in both GUI and non-GUI code. The recalculation of the correlations identifies two projects as extreme exceptional cases. The two projects are project *xui* (which had frequent identification of E6 in GUI code), and project *daffodidb* (which had small number of identified E6 in non-GUI).

Project Name	E6 in GUI	E6 in non-GUI	GUI LOC	non-GUI LOC	Project type	Download count	Number of developers	Life time
xui	59	33	19558	13423	Software Development	102935	17	38

Table 5.13 Statistical results of E6 *null pointer* error and profile of project *xui*

The project *daffodidb* has been already discussed in previous two types of errors (E2 *==instead of .equals()* error and E5 *uninitialization* error), so the focus is on the project *xui* for E6 *null pointer* error. There are several factors to link to the frequent identification of E6 in GUI code. The profile of this project is shown in Table 5.13. E6 is the most frequently occurring coding error in Java programming according to the literature. The data from the 34 projects also shows similar results. This project had been downloaded more than 100,000 times in 38 month life time. Moreover, the frequent identification of E6 *null pointer* error was linked to only a few programmers in the project *xui*. As a result, the process of the identification of E6 in the project *xui* identified that there are a few developers made mistakes in GUI code. Most of these errors were later identified because this project was very popular for download, and a large amount of changes was made to the code due to the identification of E6.

Error 7 (E7) blank exception handler

The error '*blank exception handler*' (E7) was identified in both GUI and non-GUI code. The identification of E7 occurred in many projects in the sample. It is identified that E7 is significantly correlated to GUI LOC but no correlation between E7 and non-GUI LOC is identified. The comparison of the Z-Scores between GUI and

non-GUI code for each project type is shown in Table 5.14.

Project type	Number of Projects	E7 in GUI	E7 in non-GUI	Z-Score for E7 in GUI(GZ)	Z-Score for E7 in non-GUI(NZ)	Difference of Z-Score (GZ−NZ)
Desktop Environment	1	2	0	0.61	-0.70	1.31
Database	2	3	1	1.27	0.14	1.13
Games/Entertainment	5	3	2	1.27	0.98	0.30
Software Development	8	4	3	1.94	1.82	0.12
Communications	4	0	0	-0.72	-0.70	-0.02
Education	2	0	0	-0.72	-0.70	-0.02
Format and Protocol	1	0	0	-0.72	-0.70	-0.02
Internet	1	0	0	-0.72	-0.70	-0.02
Office/Business	1	0	0	-0.72	-0.70	-0.02
Text Editors	2	0	0	-0.72	-0.70	-0.02
Scientific/Engineering	3	0	1	-0.72	0.14	-0.86
Multimedia	4	1	3	-0.06	1.82	-1.87
Average		1.08	0.83			
STDEV		1.51	1.19			

Table 5.14 Statistical results for E7 *blank exception handler* error between GUI and non-GUI code

Table 5.14 shows that the identification of E7 *blank exception handler* error in GUI code occurred more frequently in four types of projects than other types of projects. The four types of projects are 'Desktop Environment', 'Database', 'Game/Entertainment', and 'Software Development'. Table 5.14 also shows that there is no great frequency of identified E7 for GUI or non-GUI code since the greatest frequency is 4. Compared to the E6 *null pointer* error, the identification of E7 *blank exception handler* error is relatively small. The distribution of the identification is shown in Figures 5.14 and 5.15.
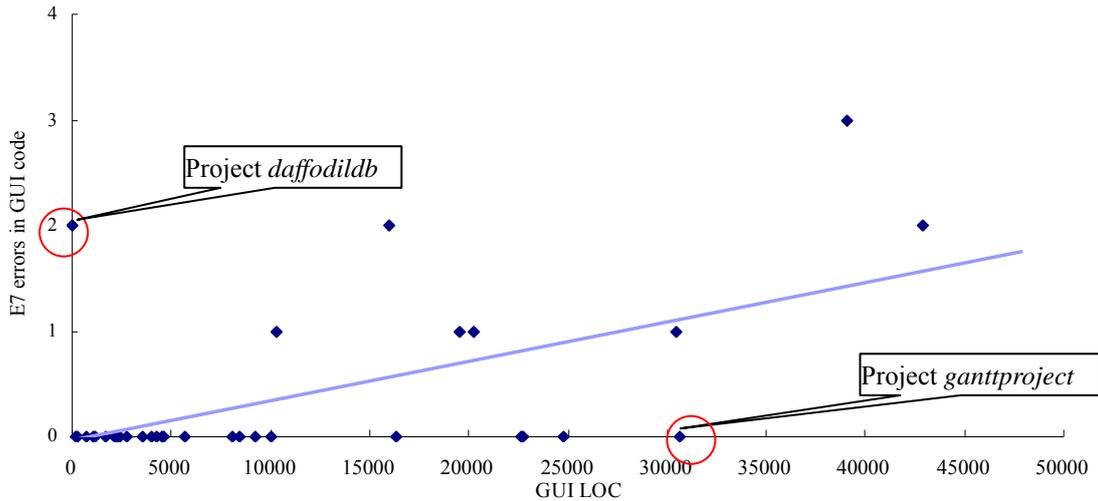
Figure 5.14 Correlation of LOC and error '*try{} catch{}*' (E7) in GUI code

Three exceptional data dot are identified for the E7 *blank exception handler* error, two dots for GUI (see Figure 5.14) and one for non-GUI code (see Figure 5.15). They are project *ganttproject* (this project had no E7 in GUI code even though it had a large GUI LOC), project *daffodildb* (this project had a great frequency of identified E7 although it had a small GUI LOC), and project *daffodildb* (it had no E7 identified although it had the greatest non-GUI LOC). The profiles of the exceptional projects (*ganttproject* and *daffodildb*) are shown in Table 5.15.
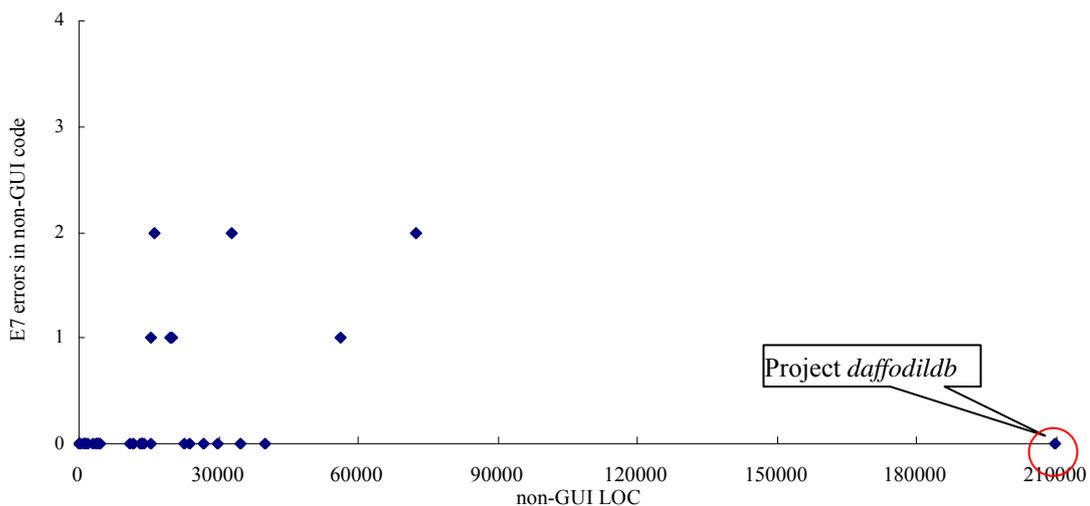
Figure 5.15 Scatter plot of E7 *blank exception handler* error in non-GUI code

The project *daffodildb* is also an exceptional case for the previous E2 *==instead*

of *.equals( )* error, E3 *capitalization* error, E5 *uninitialization* error, and E6 *null pointer* error. There are two E7 *blank exception handler* errors that were identified in GUI code and no specific features were identified to link with. Table 5.15 shows that the project *ganttproject* was frequently downloaded but no E7 error was identified in either GUI or non-GUI code in 41 months. A possible explanation is that corresponding developers may be aware of this error. However, this explanation cannot be evaluated in this thesis for lack of information.

Project Name	E7 in GUI	E7 in non-GUI	GUI LOC	non-GUI LOC	Project type	Download counts	Number of developers	Life time
ganttproject	0	0	30612	13753	Office/Business	1562551	11	41
daffodildo	2	0	38	209477	Database	13235	10	18

Table 5.15 Statistical results for exceptional projects for E7 *blank exception handler* error

Error 9 (E9) concurrent access to shared variables by threads (multithreaded)

A significant correlation is found between E9 *multithreaded* error and non-GUI LOC. Conversely, no significant correlation is identified between E9 and GUI LOC. Table 5.16 shows the difference of identification of E9.

Project type	Number of Projects	E9 in GUI	E9 in non-GUI	Z-Score for E9 in GUI(GZ)	Z-Score for E9 in non-GUI(NZ)	Difference of Z-Score (GZ−NZ)
Games/Entertainment	5	15	9	2.61	-0.39	3.00
Format and Protocol	1	4	7	0.22	-0.42	0.64
Text Editors	2	2	1	-0.22	-0.51	0.29
Communications	4	2	2	-0.22	-0.50	0.28
Desktop Environment	1	0	0	-0.65	-0.53	-0.13
Education	2	0	0	-0.65	-0.53	-0.13
Internet	1	0	0	-0.65	-0.53	-0.13
Office/Business	1	0	0	-0.65	-0.53	-0.13

Scientific/Engineering	3	0	0	-0.65	-0.53	-0.13
Software Development	8	3	53	0.00	0.29	-0.29
Multimedia	4	9	160	1.31	1.94	-0.63
Database	2	1	178	-0.44	2.22	-2.65
Average		3.00	34.17			
STDEV		4.59	64.82			

Table 5.16 Statistical results for E9 *multithreaded* error between GUI and non-GUI code

Table 5.16 shows that project types 'Games/Entertainment', 'Format and Protocol', 'Text editors', and 'Communication' are the four types of projects that had more identification of E6 *null pointer* error in GUI code than non-GUI code. According the identification of E9 *multithreaded* error in non-GUI, there are two projects are extreme different from other projects. This is shown in Figures 5.16 and 5.17.

Figure 5.16 shows that there is no predicable trend between error frequency and overall GUI LOC for E9. It also shows that the majority of projects cluster below 5000 GUI LOC. The distribution of E9 error is not centralized in a particular area so no trend is able to be predicted. Thus, the results overall may lean on the projects that contain 5,000 or less GUI LOC.
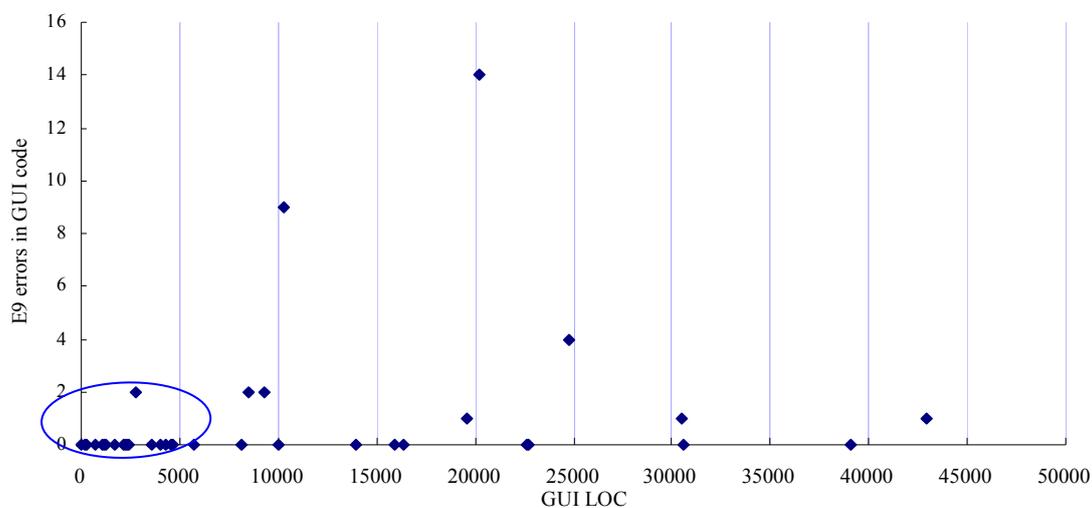
Figure 5.16 Scatter plot of E9 *multithreaded* error in GUI code
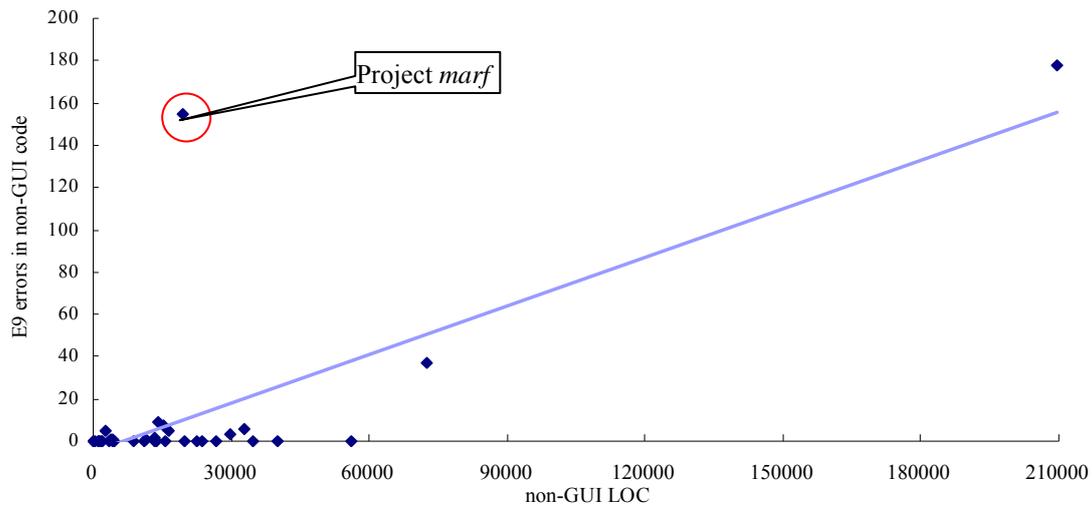
Figure 5.17 Correlation of LOC and error '*multithreaded*' (E9) in non-GUI code

Figure 5.17 shows that the project *marf* is greatly separate from the majority. After checking the code where E6 *null pointer* errors were identified, no special features of non-GUI code were found to link to the frequent identification of E6 errors. However, most of these E9 *multithreaded* errors are linked to one developer, so the developers' skill is the likely cause of the errors.

5.3.2. CCES errors which has no correlation with either GUI or non-GUI code

There are four types of errors (E1 =*instead of* == error, E4 *variable type* error, E8 *array index off-by-one* error, and E10 *static method access non-static member* error) in which no correlation is identified in GUI or non-GUI code according to Table 5.7. The distributions of the four types of errors are not proportional because no correlations are identified for the four types of errors. The scatter plot may not be useful for the analysis of the four types of errors. Only Z-Score is used to compare the identification of errors between GUI and non-GUI code.

Error 4 (E4) variable type

The error '*variable type*' was frequently identified in GUI and non-GUI code, especially in non-GUI code. The average frequency of identification is greater than 19 per project in non-GUI code. The comparison of Z-Scores is shown in Table 5.17.

Six types of project are statistically significant for E4 *variable type* errors in GUI code: 'Games/Entertainment', 'Format/Protocol', 'Science/Engineering', 'Office/Business', 'Database', and 'Software Development'. After the examination of the corresponding GUI code in these six project types, no particular cause is identified to link to the identification of E4.

Project type	Number of Projects	E4 in GUI	E4 in non-GUI	Z-Score for E4 in GUI(GZ)	Z-Score for E4 in non-GUI(NZ)	Difference of Z-Score (GZ−NZ)
Games/Entertainment	5	5	2	1.32	-0.49	1.82
Format and Protocol	1	2	6	0.10	-0.38	0.48
Scientific/Engineering	3	8	92	2.55	2.10	0.45
Office/Business	1	2	12	0.10	-0.21	0.31
Database	2	1	3	-0.31	-0.47	0.16
Software Development	8	2	19	0.10	0.00	0.11
Communications	4	0	0	-0.71	-0.55	-0.16
Desktop Environment	1	0	0	-0.71	-0.55	-0.16
Internet	1	0	0	-0.71	-0.55	-0.16
Text Editors	2	0	1	-0.71	-0.52	-0.19
Education	2	0	2	-0.71	-0.49	-0.22
Multimedia	4	1	93	-0.31	2.13	-2.43
Average		1.58	19.42			
STDEV		2.50	34.73			

Table 5.17 Statistical results for E4 *variable type* error between GUI and non-GUI code

Error 8 (E8) array index off-by-one

The error *array index off-by-one* was not identified in GUI code and only nine examples in non-GUI code. The comparison of Z-Scores is not able to be carried out because no examples of E8 were identified in GUI code.

Error 1 (E1) '=instead of ==' and Error 10 (E10) 'static method access non-static member'

These two types of errors were seldom identified in either GUI or non-GUI code. The '*=instead of ==*' error had no identification in GUI code and only two identifications in non-GUI code. The '*static method access non-static member*' error had no identification in GUI or non-GUI code. The reason is that this type of error was possibly identified in compiler stage so no identification was found afterward. Therefore, the comparison of Z-Scores becomes meaningless because the lack of data support.

5.3.3. Summary

Section 6.3 investigated each type of error and identified several types of projects in which the identification of errors was more frequent in GUI code than non-GUI code. A summary of each type of error is shown in Table 5.18.

Project type	Average LOC for GUI	Average LOC for non-GUI	Number of project	Frequently identified errors
Communications	5001	5834	4	E2 E6 E9
Database	20653	8273	2	E2 E4 E6 E7
Desktop Environment	15959	1290	1	E3 E5 E6 E7
Education	2204	1385	2	E6
Format and Protocol	24784	15384	1	E2 E3 E4 E5 E6 E9
Game/Entertainment	11849	12885	5	E2 E3 E4 E6 E7 E9
Internet	686	480	1	E3 E6

Multimedia	7025	14899	4	E3 E5
Office/Business	15779	53736	2	E2 E3 E4 E5 E6
Scientific/Engineering	15737	23394	3	E4
Software Development	11643	24290	8	E4 E7
Text Editors	5483	8472	2	E3 E5 E9

Table 5.18 Profiles of 12 types of project

Table 5.18 shows that 'Game/Entertainment', 'Format and Protocol', and 'Office/Business' project types are more likely to have frequent identification of errors in GUI components than any other type of project. However, the project type Game/Entertainment has more projects than other types. It is proposed that the 'Game/Entertainment' OSS projects may have a greater chance that errors can be identified in GUI code. In fact, the number of examined projects may be too small to predict accurate results. For example, the number of 'Format/Protocol' type projects is one, so the result from this project type may not representative.

Error type	**E2**	**E5**	**E6**	**E7**
Project name	Jfritz, unicat-i	unicats-i, frinika, omegat	xui	daffodildb, ganttproject

Table 5.19 Exceptional projects for GUI

Exceptional projects were identified for four types of error in GUI code (see Table 5.19). The project *unicat-i* is a typical example that indicates the download count sometimes determines the frequency of errors in GUI code. Therefore, the anticipation of users is an important factor to affect the maintenance of OSS GUI code.

5.4. Correlation between ten types of error in GUI and non-GUI code

According to the analysis in section 6.3, several different types of error apparently

exist in the same project; hence, some types of error seem to be correlated to each other. In this section, the possible correlations of different types of error will be analyzed. The values for the Pearson correlation coefficient are calculated to identify possible correlations between ten types of error.

Table 5.20 shows that five correlations are significant between the CCES errors. The significance is based on the analysis of 34 projects. The five correlations are shown as follows:

- E2 *==instead of .equals()* error and E3 *capitalization* error (p<0.01)
- E2 *==instead of .equals()* error and E5 *uninitialization* error (p<0.01)
- E3 *capitalization* error and E9 *multithreaded* error (p<0.02)
- E5 *uninitialization* error and E9 *multithreaded* error (p<0.05)
- E6 *null pointer* error and E7 *blank exception handler* error (p<0.05)

	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
E1	1									
E2	N/A	1								
E3	N/A	0.4392	1							
E4	N/A	0.2056	0.0229	1						
E5	N/A	0.4913	-0.0827	0.1391	1					
E6	N/A	0.2462	0.0808	0.2752	0.1381	1				
E7	N/A	-0.0427	0.0298	0.0246	0.1377	0.3747	1			
E8	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1		
E9	N/A	0.0623	0.4331	0.1224	0.3964	0.1652	0.1800	N/A	1	
E10	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1

Table 5.20 Matrix of the Pearson correlation for ten types of error in GUI code

Compared to GUI code, more significant correlations are identified in non-GUI code. 18 correlations are identified in non-GUI code (see Table 5.21). Figure 5.18 is

produced to compare the correlation between the CCES errors. Each dot represents one type of CCES error, and a circle means an example of this type of CCES error was identified. Each edge presents a significant correlation between two types of error. The strength of the correlation is represented by the width of the edge. The results of the automatic inspection show a significant difference of the correlations between GUI code and non-GUI code.

	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
E1	1									
E2	0.6236	1								
E3	0.7376	0.5237	1							
E4	0.0728	0.6278	0.0934	1						
E5	0.9557	0.5670	0.6805	0.0809	1					
E6	0.6222	0.7176	0.5380	0.3558	0.6562	1				
E7	0.4792	0.4658	0.4971	0.2130	0.5685	0.6878	1			
E8	-0.0659	-0.0736	0.0412	0.0857	0.0787	0.2728	0.3634	1		
E9	0.1106	0.0218	0.1687	-0.0353	0.1762	0.1514	0.1577	-0.1035	1	
E10	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1

Table 5.21 Matrix of the Pearson correlation for ten types of error in non-GUI code

Figure 5.18 shows that the automatic inspection identified five significant correlations in GUI (E5 *uninitialization* error and E6 *null pointer* error, E7 *blank exception handler* error and E9 *multithreaded* error). Based on the data for non-GUI, the correlations are presented in Figure 5.19. The same set of notation in Figure 5.18 is used for non-GUI code in Figure 5.19.
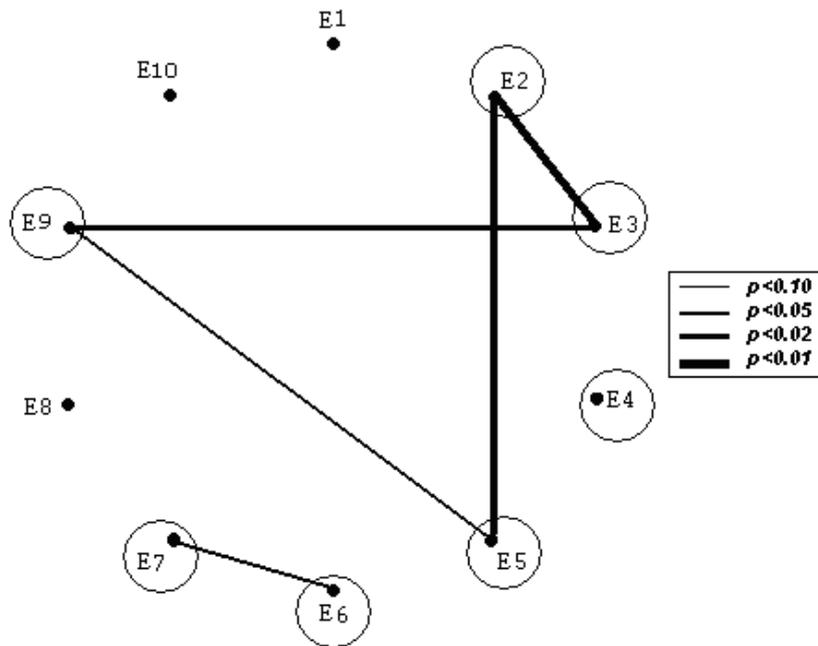
Figure 5.18 Correlations between CCES errors in GUI code

Figure 5.19 shows that the automatic inspection identified 18 significant correlations in non-GUI code. E1 =*instead of* == error, E2 == *instead of .equals()* error, E3 *capitalization* error, E4 *variable type* error, E5 *uninitialization* error, E6 *null pointer* error, and E7 *blank exception handler* error have at least two significant correlations to others.
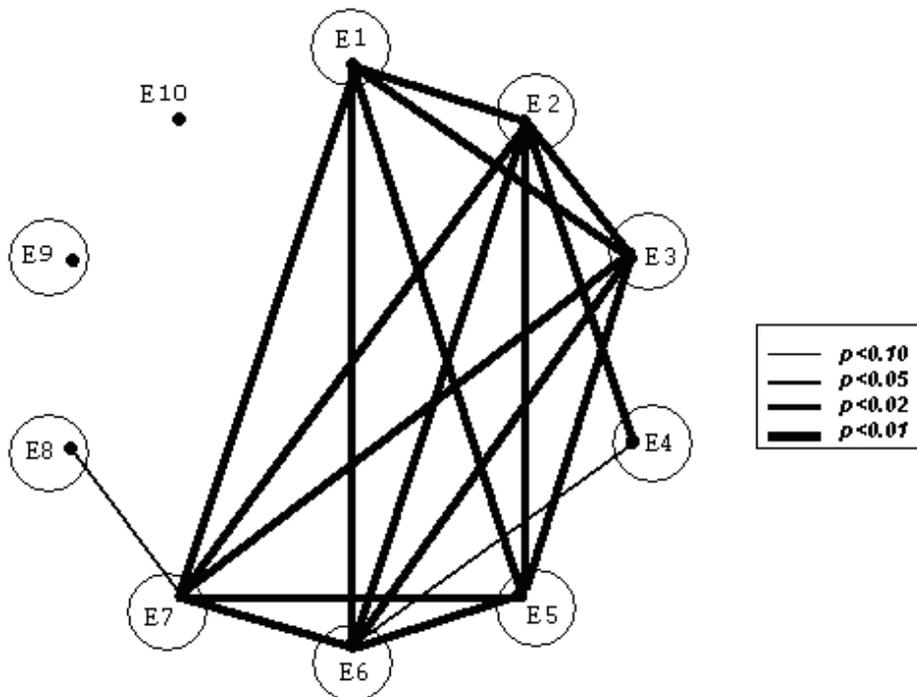
Figure 5.19 Correlations between CCES errors in non-GUI code

Considering Figures 5.18 and 5.19, the automatic inspection identified that the correlations between the CCES errors are different in GUI and non-GUI code. In other words, the clustering the CCES errors are different in GUI and non-GUI code. A statistical correlation between the CCES errors cannot guarantee a causal relationship between the errors. These clusters indicate what types of CCES error may coexist together in the code.

5.5. Chapter summary

This chapter analyzed the results from the automatic inspection. According to the analysis, nine types of CCES error were identified in the code, seven types in GUI code and nine types in non-GUI code.

The analysis of the frequency of all CCES errors showed several significant correlations. Considering the significant correlations between the frequency of all CCES errors and the LOC in GUI and non-GUI code, these results corroborate the correlation identified by Yu [Yu2006] and Koru et al [Koru2007]. Moreover, several correlations between the frequency of all CCES errors and OSS features in GUI and non-GUI code were identified. The results verified that some OSS features proposed by Crowston et al [Crowston2006] may influence the frequency of errors in GUI and non-GUI code, such as download count.

Each CCES error has been investigated. The largest proportion of the CCES errors was identified in GUI rather than non-GUI code for the project type 'Game/Entertainment'. The download count is one of the key features to affect the frequency of the identification of CCES errors since the analysis of several exceptional projects shows a similar result. The clusters of the CCES errors were investigated, several clusters were found in GUI and non-GUI code. These clusters are different between GUI and non-GUI code.

Chapter 6   Discussion and Evaluation

6.1.   Introduction

This chapter compares the results of the examined code sample and the evaluation sample. The validity of the automatic code inspection is therefore evaluated and possible drawbacks of the automatic code inspection are identified.

To evaluate the automatic code inspection, another sample of software project is selected to be examined by applying the same approach as the automatic code inspection. Excluding the chosen OSS projects in the automatic code inspection, a sample of OSS projects is selected from the same population of OSS in Sourceforge. Results are compared with the outcome from the automatic inspection. To maintain the sustainability and efficiency of the results of the two samples, the same approach is applied to the evaluation code sample. The same analysis methodologies are applied, such as quantitative analysis. Sample analysis, correlation comparison and other relevant quantitative analysis methods are also used.

The sections in this chapter describe the process of evaluation sample selection and code examination; standardizes the results and compares outcomes with the automatic code inspection. Based on the comparison of CCES between the two samples, advantages and disadvantages of analysis methodology in this thesis are identified.

6.2.   Evaluation Sample Process

The process of the evaluation sample generates results for comparison. The process of evaluation sample code selection is similar to the automatic code inspection; however, the selection of the evaluation sample is based on a completely random selection

instead of a manual selection. The process is shown in Figure 6.1.
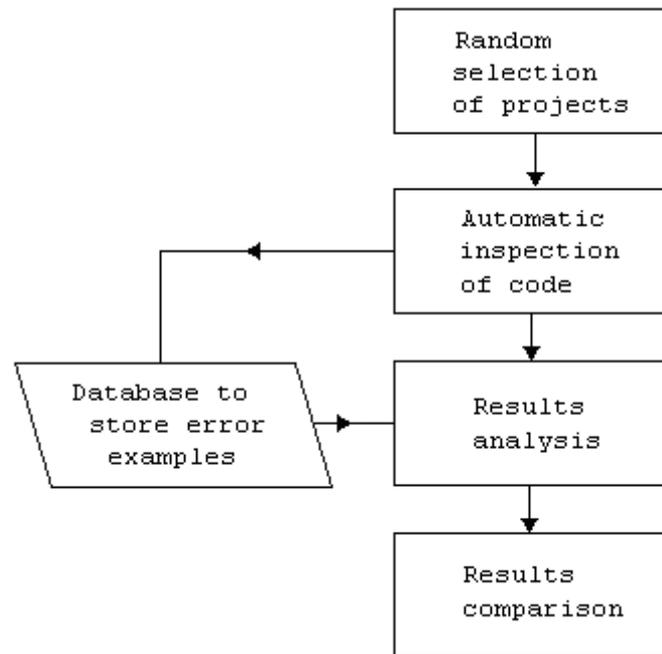
Figure 6.1 Process diagram of the evaluation sample

One of the drawbacks of the automatic code inspection is that the code is not randomly selected but is manually selected. To avoid this drawback and evaluate the automatic code inspection results, a new process of selection of projects is applied. There are several steps to randomly select the evaluation sample. These are:

Step 1. Apply the same set of criteria as the automatic inspection to identify the population of projects;

Step 2. Manually collect all relevant information for all the projects in the population;

Step 3. Prepare a text file that contains a list of project names;

Step 4. Apply the function `random( )` to the text file and randomly select a set of project name to form a sample;

Step 5. A script of automatic download and get `diff` for these selected projects is run;

Step 6. Follow the same process as the automatic code inspection.

```
#!/bin/sh
cvs -z3 -d:pserver:anonymous@jmedialibrary.cvs.sourceforge.net:/cvsroot/jmedialibrary co
-P jmedialibrary
```

Note: The text with shadow is project name which is variance between different projects.

Figure 6.2 Fragment of script

There were total 10698 Java projects with GUI components in Sourceforge on 30th April 2009, including 1034 projects with AWT, 8193 projects with Swing, and 1471 projects with SWT. The sample size determination make is based on recommendations according to Higgins' sample size table [Higgins 2001]; therefore, a sample of 264 projects was randomly selected from the population.

Upon running the sample some downloads were successful others were not. In most cases of unsuccessful downloads, they are not able to be downloaded after the execution of the script of which a fragment is shown in Figure 6.2. Valid projects are normally downloaded to the local hard drive after the command in Figure 6.2 is executed; however, some projects return nothing after its execution. Three attempts were made to randomly select the sample of 264 projects. The first random selection of 264 projects returned only 15 projects with their source code. The second random selection attempted 1000 instead of 264 but only 10 valid projects were returned. The third random selection increase the number of random selection of projects 3000 but only 38 projects turned out to be successfully downloaded.

The process of downloading the selected project is very time-consuming; the download of the first 15 projects took more than four days. The reasons for the cause of the slow download are:
1. Some projects contain a large number of files with considerable depth;
2. Sourceforge server connection is not stable.

These two factors make the download of the source code a time consuming process. The second attempt returned 10 available projects and the download process lasted more than six days. The third attempt returned 38 available projects and they were

completely downloaded in 20 days; unfortunately however seven out of 38 projects had no `diff` outcome. As a result, the size of evaluation sample is 31 projects. Even though a sample of 31 is smaller than the proposed 264, the downloading issue restricted further samples. This is one of the threats to validity of the evaluation sample and the experimental design.

6.3. Results Evaluation

6.3.1. Sample comparison

The Sourceforge project population has rapidly increased from the first sample selection to the last selection. The sample selections were made in June 2005 for manual selection and April 2009 for automatic selection. There were 3531 Java GUI projects that list in Sourceforege during the manual selection and 10698 projects during the automatic selection. In less than four years period, the number of OSS projects has increased more than three times. This increase of population provides an opportunity to evaluate the validity of the results obtained in the automatic code inspection. More specifically, the number of Swing, AWT, and SWT project is shown as follows:

	Automatic inspection projects	New evaluation projects
Swing project	2273	8193
AWT project	561	1034
SWT project	707	1471

Table 6.1 Number of project in two samples

Table 6.1 shows that the percentage of Swing, AWT, and SWT remains similar during the fast growth of the OSS projects. In other words, the characteristics of population are similar from the automatic inspection to the evaluation. Therefore, the evaluation sample is also representative for the automatic inspection sample.
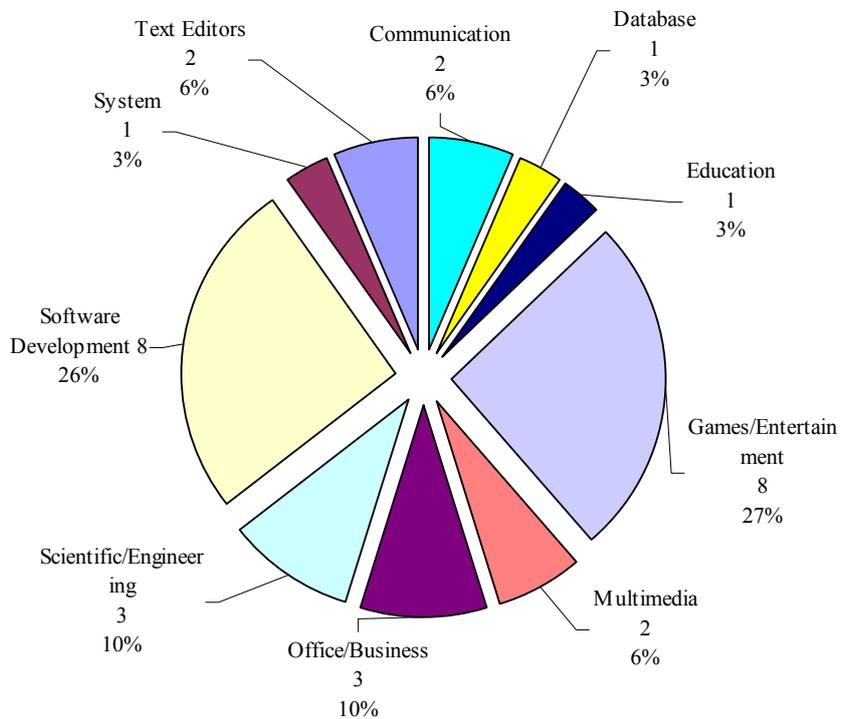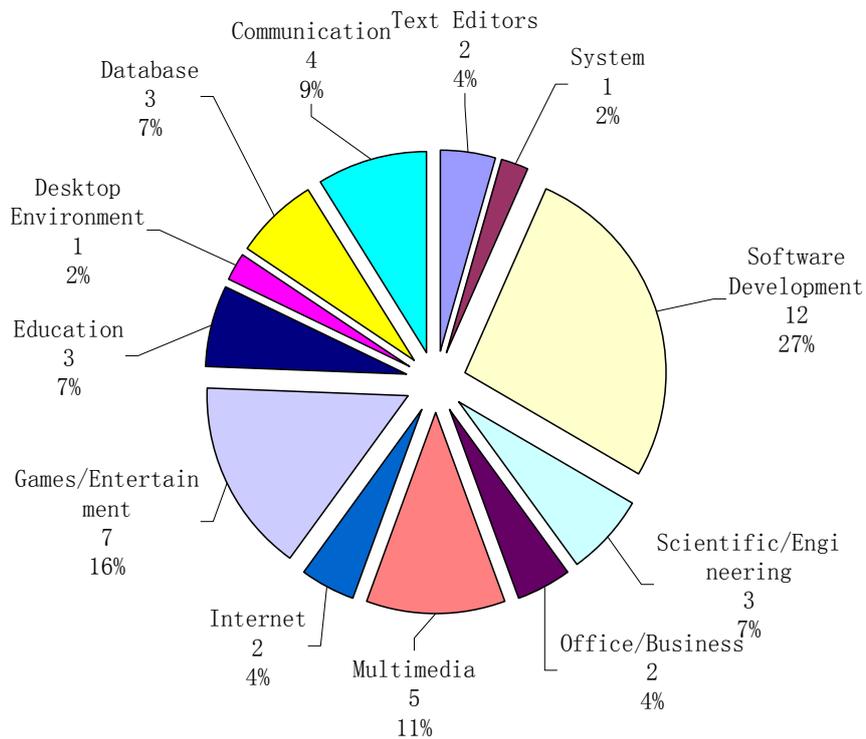
Figure 6.3 Comparison between automatic projects set and evaluation projects set

In addition, the percentage of each project type is also similar between the automatic inspection sample and the evaluation sample. The sample for the automatic code

inspection contains 45 projects and the sample for the evaluation contains 31 projects. 12 types of projects are identified in the sample of 45 and ten types of projects are identified in the sample of 31. The detail is shown in the pie chart Figure 6.3.

Figure 6.3 shows that the percentage of each project type varies little between two samples. The largest portions of the sample are Game/Entertainment and Software Development in both samples. The absolute numbers of project for other types of projects are close in the two samples; relatively, the percentage of corresponding types of projects is close in the two samples. Accordingly, the two samples are not diverse in terms of the specific elements even though the capacity of the population has been increased from the manual sample selection to the automatic sample selection.

	Automatic inspection projects			New evaluation projects		
Feature	Max	Mean	Min	Max	Mean	Min
Project size (LOC)	282638	34494	258	14681	13443	89
Development history (months)	76	26	0	65	37	2
Number of developers	30	9	1	3	1	1
Download counts	1562551	73798	0	70618	5393	0

Table 6.2 Comparison between two samples according to four characteristics

In some ways automatic inspection sample and the evaluation sample are not similar to each other. This is caused by the random selection of the projects. Table 6.2 shows differences in four main characteristics between two samples;

- the average LOC of the automatic inspection sample is much greater than the evaluation sample;
- the mean of project history of former sample is smaller than the latter sample;
- the average number of developers of the former sample is much greater than the latter;

- the former sample has greater download counts than the latter.

Even though both two samples are selected from OSS project in the same database, the order of selection is different and the population is different as well. This however provides a better and wider test object for the approach in this thesis; the results from the automatic inspection sample should be applicable in the evaluation sample since they are both representative of OSS Java GUI projects. The comparison of results from two different samples will be utilized to provide a better evaluation.

6.3.2. Results comparison

Based on the results from the analysis of data obtained in the automatic inspection, several outcomes are presented in Chapter 5. These results however need to be evaluated for the confirmation and validation. Correspondingly, a set of corresponding outcomes was generated from the evaluation sample. By comparing the two sets of outcomes from two samples, the outcome confirmed by both sample can be considered as experimental evidence. The main outcomes of the automatic inspection are:

- ranking of errors in GUI code;
- identified errors density in GUI;
- frequently identified errors in GUI;
- frequently identified errors in certain type of project;
- relationship between errors in GUI;
- relationship between errors and OSS characteristics in GUI.

In addition, the same approach was adopted for the automatic inspection sample and the evaluation sample, the detail of processing data for the evaluation sample is not fully described in this chapter as a full description of data processing and analysis can be found in Chapter 5.

Ranking of errors between in GUI code

Ranking (the automatic inspection)		Ranking (evaluation)		Ranking (the literature)
GUI	non-GUI	GUI	non-GUI	
E6 (248)	E9 (410)	E6 (63)	E5 (94)	E8
E9 (36)	E6 (347)	E4 (19)	E6 (89)	E6
E5 (27)	E4 (230)	E5 (6)	E9 (31)	E9
E4 (21)	E5 (102)	E9 (6)	E7 (17)	E5
E2 (20)	E2 (37)	E3 (5)	E3 (14)	E2
E7 (13)	E3 (31)	E8 (3)	E4 (12)	E1
E3 (11)	E7 (10)	E7 (1)	E8 (4)	E4
E1 (0)	E8 (9)	E1 (0)	E2 (2)	E7
E8 (0)	E1 (2)	E2 (0)	E1 (0)	E10
E10 (0)	E10 (0)	E10 (0)	E10 (0)	E3

Note: values in the bracket represent the frequency of error

Table 6.3 Comparison among the ranking of the identified coding errors

The ranking of CCES in GUI is similar in the automatic inspection sample and evaluation sample with the exception of a difference in one type of error. The top four frequently identified errors are the same although the ranks are in a different order; however, the most frequently identified error is E6 *null pointer* error for both samples. Table 6.3 shows the results from the two samples. Seven out of ten CCES errors were identified in the OSS GUI code in both samples. More specifically, the most common coding error *null pointer* (E6) is the top coding error in the OSS GUI. The error *"="* *instead of "=="* (E1), *array index off-by-one* (E8), and *accessing non-static member variable from static method* (E10) were not identified by the AICCE. Based on the experimental evidence, it shows that the most common coding errors in general are also frequently identified in the OSS GUI code.

Comparison of errors between GUI code and non-GUI code

The result from the automatic inspection sample shows that the frequency of identified CCES errors in the OSS non-GUI code is overall greater than the OSS GUI

code. The same result was obtained from the evaluation sample. A similar result is also obtained for each individual type of CCES error. In addition, the density of identified CCES errors in non-GUI code is also twice that of GUI code (1.6/KLOC for non-GUI and 0.8/KLOC for GUI). Based on these facts, it is summarized that the number of identified errors in non-GUI code is greater than GUI code.

Certain types of coding errors are frequently identified in the OSS GUI code

The results from the AICCE show that the errors *'==' instead of '.equals()'* (E2), *null pointer* (E6) and *blank exceptional handler* (E7) were more frequently identified in GUI code than non-GUI code in terms of error density. The results from the evaluation sample show that the error *null pointer* (E6) was frequently identified. It is summarized that E6 is particularly frequent in GUI code compared to non-GUI code.

Certain types of errors are frequently identified in GUI code of certain types of OSS projects

Several types of CCES error were relatively more frequently identified in GUI code than non-GUI code for certain type of OSS project (see Table 6.4). The result from the automatic inspection shows that the 'Game/Entertainment' project type had more types of errors identified in GUI code than non-GUI code. E2 *= = instead of .equals()* error, E3 *capitalization* error, E4 *variable type* error, E6 *null pointer* error, E7 *blank exception handler* error, and E9 *multithread* error were all frequently identified in GUI code of 'Game/Entertainment' project. However, evaluation sample results show that 'Database' and 'Text Editor' are not the case. By comparing the results from two samples, the project types 'Database' and 'Text Editor' are the project that more errors were identified in GUI code than non-GUI code. These two types of project can be found in Table 6.4 that have the greatest number of double ticks, for instance, there are two ticks for E3 *capitalization* error in A (automatic inspection sample) and E (evaluation sample) for Database project. The double ticks indicate that errors are frequently identified in GUI in respect of Z-score from the automatic inspection sample and evaluation sample.

Considering the results from two samples, it is summarized that project Game/Entertainment and Text Editor have more E3 *capitalization* error identified in GUI than non-GUI code; project Database and Scientific/Engineering have more E4 *variable type* error in GUI; project Multimedia, Office/Business and Text Editor have more E5 *uninitialization* error in GUI, and so on.

	E1		E2		E3		E4		E5		E6		E7		E8		E9		E10	
	A	E	A	E	A	E	A	E	A	E	A	E	A	E	A	E	A	E	A	E
Communications			✔								✔			✔			✔			
Database			✔				✔	✔		✔	✔	✔	✔	✔						
Desktop Environment					✔				✔		✔		✔							
Education								✔		✔	✔	✔		✔						
Format and Protocol			✔		✔		✔		✔		✔						✔			
Game/Entertainment			✔		✔	✔	✔				✔		✔			✔	✔			
Internet					✔						✔									
Multimedia					✔			✔	✔	✔		✔		✔						
Office/Business			✔		✔		✔		✔	✔	✔			✔		✔				
Scientific/Engineering							✔	✔						✔		✔				
Software Development							✔					✔	✔							
System								✔		✔		✔		✔						
Text Editors					✔	✔		✔	✔	✔				✔			✔	✔		

Note: A represents the automatic inspection sample and E represents the evaluation sample

Table 6.4 Frequently identified errors in GUI than non-GUI code

One type of coding error causes an another type of coding error in GUI code

No causal relationship can be confirmed even though the experimental results show there are several correlations between CCES errors in GUI code. Since there are no causal relationship between errors, one error can be a predictor for another. Figure 5.18 (see section 5.4) shows that five correlations were identified between errors. These correlations are:

- E2 *= = instead of .equals()* error and E3 *capitalization* error;
- E2 *= = instead of .equals()* error and E5 *uninitialization* error;
- E3 *capitalization* error and E9 *multithread* error;

- E5 *uninitialization* error and E9 *multithread* error;

- E6 *null pointer* error and E7 *blank exception handler* error.

The evaluation analysis identified 15 correlations between errors. The detail is shown in Figure 6.4.
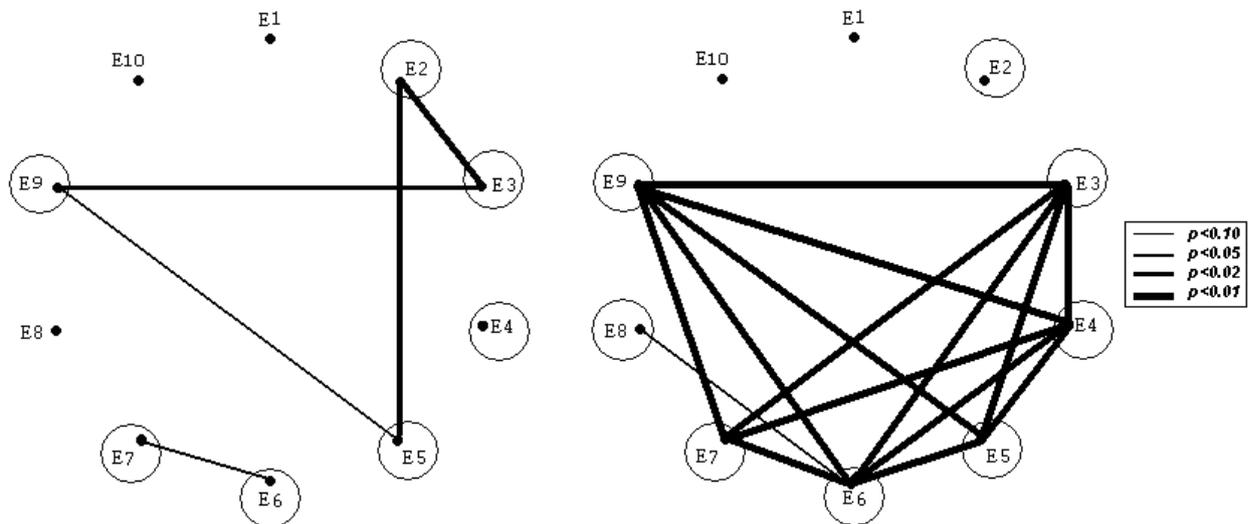
Figure 6.4 Correlations between errors in the automatic projects set and evaluation projects set

By comparing the two results, three correlations between errors in the two sample project populations were identified:

- E3 *capitalization* error and E9 *multithreaded* error;

- E5 *uninitialization* error and E9 *multithreaded* error;

- E6 *null pointer* error and E7 *blank exception handler* error.

These correlations on their own do not confirm any causal relationship but they indicate that these types of errors can be regarded as predictive of each other in GUI code.

Characteristics of OSS are correlated to the identification of coding errors in the OSS GUI code

The total number of identified errors for each error type in the automatic inspection sample is greater than the evaluation sample. This difference is caused by the characteristics of the sample. The automatic inspection sample has more LOC and download counts; these two factors influence the number of identified errors in code,

especially, the download counts for the OSS characteristic. The influence of specific characteristics could be indicated by the correlation between it and the number of errors. Table 6.5 shows the details of correlation between errors and four characteristics.

	GUI errors		Non-GUI errors	
	Automatic inspection sample	Evaluation sample	Automatic inspection sample	Evaluation sample
LOC	0.6959	0.980	0.6585	0.946
Download	0.2903	0.797	0.0076	0.114
Developer	0.3974	0.263	0.3550	0.125
Life	0.4024	0.126	0.2536	0.607

Table 6.5 Comparison between correlations

Table 6.5 shows that the factor LOC has an influence on both GUI and non-GUI errors. Projects with more LOC have greater capacity for potential errors in general. Furthermore, the correlation values for GUI are only slightly greater than non-GUI. The LOC has the capability to indicate potential errors in both GUI and non-GUI code. With a consideration of literature, LOC is widely used as a metrics to predict defect in software, for instance the metric is supported by the work of Cartwright [Cartwright 2000].

Unlike the LOC, download count is a potential critical predictor to indicate the capacity of errors in OSS GUI. A greater download count creates opportunity for a greater number of errors to be identified in GUI during the lifetime of the project so it could be regarded as a predictor for the total of revealed errors in OSS GUI code. The evidence can be found in Table 6.4. No correlation between download and errors are found in non-GUI in either the automatic inspection sample or the evaluation sample.

The result from the evaluation sample indicates that the characteristics, including developer and project life are not reliable predictor for the number of errors in GUI or non-GUI code. Table 6.5 shows different results between GUI and non-GUI in both

automatic inspection sample and evaluation sample. The above discussions are confirmed. Download count however is suitable predictor for the number of identified error in GUI. The possible explanation for this is that errors will be likely identified during the interaction between users and developers.

6.4. Chapter Summary

In this chapter, the results from the automatic inspection sample and evaluation sample are compared. The rank of CCES in GUI was evaluated, and the implementation of the AICCE was assessed. The results from the automatic inspections sample and evaluation sample were discussed. The evaluation between the two samples shows that examples of the CCES were commonly identified in both GUI and non-GUI code, and download count is a critical predictor to indicate the number of identified errors in OSS GUI. In addition, several correlations between errors are examined; and frequently identified errors in certain type of project are evaluated.

Chapter 7   Conclusion

7.1.   Introduction

This thesis explored common coding errors in the OSS GUI code. After applying the CCES to source code samples, a comparison between GUI and non-GUI code was conducted. Based on the results from the AICCE, ten conclusions are drawn for the hypotheses proposed.

The interest of errors in GUI code motivated the investigation for this thesis. The results of the literature survey drove this work to a specific approach: the static analysis of common coding errors in GUI and non-GUI code across revision histories. The literature also indicated the appropriate metrics for the approach, such as developer size, development history.

By comparing and summarizing common coding errors identified in the literature, a set of 15 coding errors were selected to be examined in a sample of 25 OSS projects by the manual inspection. After the review of the results from the manual inspection, the ten most common of the 15 coding errors were selected for the CCES and later applied to 45 projects by the automatic inspection. However, different results were obtained from the manual inspections and the automatic inspection. The evaluation of the two inspections indicates that the manual inspection was influenced by false negative examples, and the automatic inspection was slightly affected by the false positive examples. The outcomes of this thesis rely on the automatic inspection and the evaluation inspection. After considering the empirical evidence from the literature, several findings of this thesis were verified.

The results from this thesis provide experimental data and conclusions for the study of

errors in OSS GUI code and further for OSS GUI maintenance. Nevertheless, the investigation in this thesis is the beginning of a journey to improve maintenance of GUI software; several new directions were proposed for future research.

7.2.   Contribution of this thesis

The investigation of common coding errors in the OSS GUI code presents evidence to evaluate the hypothesis. The main contribution of this thesis is that a series of correlations were identified in OSS GUI code. Based on experimental evidence, conclusions are made. These could be valuable for further practice and research.

The hypothesis defined in Chapter 1 of this thesis was:

HYPOTHESIS *Coding errors in GUI code are different from non-GUI; the identification of coding errors codes show unique characteristics in the OSS GUI code.*

CONCLUSION *There are significant differences in errors between GUI code and non-GUI code, furthermore, some characteristics for OSS turned out to be useful to predict errors in GUI code; nevertheless, conflicts of results between two different samples indicate that these findings should be considered as experimental references instead of universal guidelines for coding errors in OSS GUI code.*

Based on the findings, several conclusions are drawn including:
- The most frequently identified coding error in OSS GUI code is *null pointer* (E6). The identification of E6 is correlated to E7 *blank exception handler* error. Maintenance should therefore be aware of the possible co-occurrence of the two types of error. OSS programmers and maintainers may keep in mind that the creation of E6 *null pointer* error is most likely to be made in code, especially in GUI code. The programmer may notice the possibility of making

this error during programming, such as the check of variable before the variable is used by others. The maintainer may notice that the identification of E6 *null pointer* error may accompany with E7 *blank exception handle* error. The exploration of E7 may need to be conducted while an E6 error is identified. However, the E6 and E7 errors are correlated but no concrete evidence to conclude a relationship between these two types of errors. Programmers or maintainers may not consider this correlation as a causal relationship but a referential suggestion.

- The total number of errors in the OSS non-GUI code is greater than the OSS GUI code and the total number of non-GUI LOC is greater than GUI LOC. The CCES defines general coding errors for both the OSS GUI code and non-GUI code. None of the errors in CCES should be described as a GUI specific error. In terms of coding, there are no specific GUI coding errors that were identified in this work. Programmers and maintainers may notice that the majority of coding error in GUI is not specific GUI coding errors but general coding errors. Nevertheless, this finding is not verified or approved by other research, as a result, there is possibility that specific GUI coding errors remain undiscovered and were not identified in this thesis. Programmers and maintainers may not completely ignore to seek specific GUI coding errors.

- More errors were identified in GUI than non-GUI code for the Education, Office/Business, and Text Editors projects. More specifically, more errors examples of each type of error were identified in GUI than non-GUI for the above projects. Programmers and maintainers, in general, should be more careful about the creation of errors in Database and Text Editors programs. More specifically, E3 *capitalization* error needs to be considered in Game/Entertainment program during GUI development and maintenance; E5 *uninitialization* error in GUI requires more concerns in Multimedia and Office/Business programs. The detail is shown in Table 7.1. In addition, these findings are extracted from the selected two samples, and no behind reasons are clearly shown through the analysis in this thesis; they may draw attention

of programmers and maintainers interest but not strictly guide their work.

	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
Database				✔		✔	✔			
Education						✔				
Game/Entertainment			✔							
Multimedia					✔					
Office/Business					✔					
Scientific/Engineering				✔						
Text Editors		✔			✔				✔	

Table 7.1 Identified coding errors in GUI code for each type of project

- Correlations were identified in GUI code for E3 *capitalization* error & E9 *multithreaded* error, E5 *uninitialization* error & E9 *multithreaded* error, and E6 *null pointer* error & E7 *blank exception handler* error. Programmers and maintainers may be aware of the co-existence of the three pairs of errors. Especially, GUI maintainers may search the paired errors when another is identified in GUI code. This may help an early discovery of errors. On the other hand, no clear causal relationship could be confirmed between these errors; therefore, these findings may not be used as concrete evidence to assure the necessity of coexistences.

- LOC can be used as a predictor for errors not only in GUI code but also in non-GUI code. In other words, this metrics is not unique for the prediction of errors in GUI code. However, the OSS characteristics download count can be regarded as a potential predictor for errors in the OSS GUI code. Based on the record of download count, the potential existence of errors in GUI can be predicted. The OSS program with low count of download is most likely to have more errors unidentified in GUI code. The activities of reporting errors are seldom occur in such kind of OSS program. The identified errors in this work are based on code changes so there should be several unidentified errors remain in GUI code. A comprehensive review of all complete versions of code is needed for identifying the hidden errors. This can be a future research for further confirmation.

The above conclusions are the main outcome of the investigation in this thesis. As an investigation, it intended to identify relationship between errors and software metrics. In fact, evidence and potential relationships in this thesis enrich the experimental evidence in the software maintenance field. Little evidence and related work of GUI error were found in the literature, this thesis attempted to explore potential predictor for further study in GUI errors.

7.3.   Limitation and future work

Limitations and improvements

The outcomes indicate that the investigation conducted in this thesis does not provide significant evidence to form some concrete causal relationship between errors and predictors. This requires further design of analysis methods by considering the potential causal relationship between errors and to identify any appropriate predictors. This is the main limitation of this thesis.

The attempt to discover new types of coding errors in GUI code returned no specific results. The exploration of new types of error was conducted by the manual inspection. However, potentially new types of GUI coding error could be ignored by the manual inspection since the exploration of new type of GUI errors may need a complete comprehension of code instead of reviewing of a code fragment or line by line. Additionally, adding more people to the team could improve the manual inspection. For instance, a multi person manual inspection of full comprehension of code may identify unexplored new type of GUI error. The multi person manual inspection also reduces the human mistakes since the identification of errors are reviewed and confirmed by multi persons.

The effect of the structure of the program on errors in GUI code is not considered by

this thesis. No systematic classification of characteristics of GUI code was made, such as classes, methods in GUI code. The Object Oriented metrics, such as CK metrics [Chidamber and Kemerer 1994], may be applied to identify predictor for errors in GUI and non-GUI. Using these metrics may identify the causal relationship between the metrics and errors in GUI. With a consideration of these metrics, causes of error in GUI could be identified by comparing GUI and non-GUI code. As a result, the frequently identified GUI errors due to the differences of code between GUI and non-GUI can be identified.

Future work

The work presented in this thesis addresses many interesting directions for further research. Besides the extension of the investigation in this thesis, there is a new issue: The investigation in this thesis identified that the download count in the OSS code has predictive effort on the identification of errors in GUI code. However, this thesis does not provide further evidence to indicate how the download affects the identification of errors in GUI during the evolution of OSS software. One of the most likely explanation could be the participation of users help maintainer identify errors in GUI. As long as an OSS has been frequently downloaded by users, more and more errors in GUI may be identified and reported by users; these reports could possibly lead to the error correction during the evolution of OSS. A large number of the interactive maintenance activities between users and maintainers cause the majority of errors in GUI are corrected quickly, and furthermore the relevant errors in non-GUI are corrected. This could be one of the most interesting topics for future research. The conduction of this research follows:

     1. all code changes are identified including addition, deletion, and changed;

     2. changes made due to errors are identified, classified and quantified;

     3. reasons for these changes are classified and quantified;

     4. a case study of some exceptional project is conducted, and possible modes of how errors are corrected due to users is also conducted;

5. a set of appropriate predictors are selected to form an error evolution model for OSS GUI code;

6. a further evaluation of the defined predictors is conducted and results are confirmed.

Furthermore, an investigation of programmer competences can also identify factors that drive programmers to make mistakes. Interviews with programmers' managers should be conducted to obtain information, such as what kind of features that causes the programmer to make mistakes. By analyzing the examples, it is possible to identify a pattern of how each type of coding error is created in GUI code. Based on these patterns, a GUI-specific error detecting tool could be developed.

References

[Ayewah2007]   N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating Static Analysis ERROR Warnings on Production Software", Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, p 1-8, 2007.

[Ball1999]   T. Ball "The Concept of Dynamic Analysis", ACM SIGSOFT Software Engineering Notes, Volume 24, Issue 6, p 216 – 234, 1999.

[Basili1996]   V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sorumgard, and M.V. Zelkowitz, "The Empirical Investigation of Perspective-Based Reading," Empirical Software Engineering, Volume 1, Number 2, p 133- 164, Springer, 1996.

[Bennett2000]   K. H. Bennett and V. T. Rajlich, "Software Maintenance and Evolution: a Roadmap", Proceedings of the Conference on the Future of Software Engineering, p 73 – 87, ACM, 2000.

[Bartlett2001]   J. E. Bartlett, J. W. Kotrlik, C. C. Higgins, "Organizational research: Determining appropriate sample size in survey research", Information Technology, Learning, and Performance Journal, Vol. 19, pp.43 – 50, 2001.

[Bhatt2006]   P. Bhatt, K. Williams, G. Shroff, A. K. Misra, "Influencing Factors in Outsourced Software Maintenance", ACM SIGSOFT Software Engineering Notes, Volume 31, Issue 3, p1-6, May 2006.

[Bieman2001]   J. M. Bieman and V. Murdock, ""Finding Code on the World Wide Web: A Preliminary Investigation", First IEEE International

Workshop on Source Code Analysis and Manipulation, p 73-78, 2001.

[Biffi2001]   S. Biffi, B. Freimut, and O. Laltenberger, "Investigating the Cost-Effectiveness of Reinspections in Software Development", Proceedings of the 23rd International Conference on Software Engineering, Toronto, Canada, p 155-164, May 2001.

[Binkley2007]   D. Binkley, "Source Code Analysis Road Map", Future of SoftwareEngineering, MN, U.S.A, p 104-119, 2007.

[Capiluppi2002]   A. Capiluppi, P. Lago and M. Morisio, "Characterizing the OSS Process", Proceedings of International Conference on Software Engineering, 2nd Workshop on Open Source Software Engineering, Orlando, Florida, May 2002.

[Cartwright2000]   M. Cartwright, M. Shepperd, "An Empirical Investigation of an Object-Oriented Software System", IEEE Transactions on Software Engineering, vol. 26, No. 8, 2000.

[Chen2001]   J. Chen and S. Subramaniam, "A GUI Environment to Manipulate FSMs for Testing GUI-based Applications in Java", Proceeding of the 34th Hawaii International Conferences on System Sciences, Volume 9, p 9061, Jan 2001.

[Chen2005]   Y. Chen, R. Dios, A. Mili, L. Wu and K. Wang, "An Empirical Study of Programming Language Trends", IEEE Software, Volume 22, Number 3, p 72-78, May 2005.

[Chidamber1994]   S. R. Chidamber, C. F. Kemerer, "A metrics suite for object-oriented design", IEEE TSE 20 (6), 476–493, 1994.

[Chou2001]   A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An

Empirical Study of Operating Systems Errors", ACM SIGOPS
Operating Systems Review , Proceedings of the Eighteenth ACM
Symposium on Operating Systems Principles SOSP '01, Volume
35, Issue 5, p73-88, 2001.

[Cole2006]   B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K.
Stephens, "Improving Your Software Using Static Analysis to
Find Bugs", ACM SIGPLAN Conference on Object-oriented
Programming Systems, Languages, and Applications, p 673 – 674,
2006.

[Conradi1998]   R. Conradi and B. Westfechtel, "Version Models for Software
Configuration Management", ACM Computing Surveys, Volume
30, Issue 2, p 232-282, 1998.

[Corbesero2006]   S. G. Corbesero, "Rapid and Inexpensive Lab Deployment Using
Open Source Software", Journal of Computing Sciences in
Colleges, Volume 22, Issue 2, p228-234, December 2006.

[Crowston2003]   K. Crowston, H. Annabi, and J. Howison, "Defining Open Source
Software Project Success", Proceedings of the 24th International
Conference on Information Systems (ICIS 2003), Seattle, WA,
p327-340, 2003.

[Crowston2006]   K. Crowston, J. Howison, and H. Annabi, "Information Systems
Success in Free and Open Source Software development: Theory
and Measures", Software Process--Improvement and Practice
(SPIP), John Wiley & Sons Ltd, Volume11, Issue 2, p123-148,
2006.

[Dunsmore2003]   A. Dunsmore, M. Roper, and M. Wood, "Practical Code
Inspection Techniques for Object-oriented Systems: An

Experimental Comparison", IEEE Software, Volume 20, Issue 4, p 21 – 29, 2003

[Eclipse2008]   Eclipse, "SWT: The Standard Widget Toolkit", [Available online] http://www.eclipse.org/swt/, the website is last accessed on 25/06/2008.

[Emam2002]   K. E. Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis, and S. N. Rai, "The Optimal Class Size for Object-Oriented Software", IEEE Transaction on Software Engineering, Volume 28, Issue 5, p494–509, 2002.

[Engler2001]   D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code", ACM SIGOPS Operating Systems Review, Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles SOSP '01, Volume 35, Issue 5, p57-72, October 2001.

[Erlikh2000]   L. Erlikh, "Leveraging Legacy System Dollars for E-business", IEEE IT Proceedings, p17-23, May/June 2000.

[Feller2002]   J. Feller and B. Fitzgerald, "Understanding Open Source Software Development", Addison Wesley Professional, 2002.

[Fenton 2000]   N. E. Fenton, N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System", IEEE Transactions on Software Engineering, vol. 26, No. 8, p797-814, 2000.

[Findbug 2009]   FindBugs, University of Maryland, "FindBugs™ - Find Bugs in

Java Programs", [Available online]: http://findbugs.sourceforge.net/, the website is last accessed on 05/06/2009.

[Godfrey2000]   M. W. Godfrey and Q. Tu, "Evolution in Open Source Software: A case study", Proceedings of the International Conference on Software Maintenance, Washington, DC, USA, p 131, 2000.

[Gosling2000]   J. Gosling, B. Joy, G. Steele, and G. Bracha, "The Java Language Specification (Java)", Addison-Wesley, 2000.

[Goulde2006]   M. Goulde and E. Brown, "Open Source Software: A Primer for Health Care Leaders", California Health Care Foundation, March 2006.

[Graves2000]   T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History", IEEE Transaction on Software Engineering, Volume 26, Number 7, July 2000.

[Grimshaw1993]   A. Grimshaw, "Easy-to-use Object-oriented Parallel Processing with Mentat", Computer, Volume 26, Number 5, p39–51, 1993.

[Grubb2003] P.   Grubb and A. A. Takang, "Software Maintenance: Concepts and Practice", World Scientific Publishing Company, 2003.

[Heuzeroth2003]   D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe. "Automatic Design Pattern Detection", IEEE International Workshop on Program Comprehension, p 94 – 103, 2003.

[Hooimeijer2007]   P. Hooimeijer and W. Weimer, "Modeling Bug Report Quality", Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, p34-43, 2007.

[Hovemeyer2004]   D. Hovemeyer and W. Pugh, "Finding Bugs is Easy", Conference on Object Oriented Programming Systems Languages and Applications, p132-136, 2004.

[Howell2003]   C. J. Howell, G. M. Kapfhammer, and R. S. Roos, "An Examination of the Runtime Performance of GUI Creation Frameworks", Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java, ACM International Conference Proceeding Series, , Volume 42, p 171 − 176, 2003.

[IEEE610.12-1990]   IEEE Std. 610.12-1990. "IEEE Standard Glossary of Software Engineering Terminology", 1990.

[ISO/IEC15288-2002] ISO/IEC 15288, "Software Engineering: Software Life Cycle Processes", ISO/IEC, 2002.

[Kersten2005]   M. Kersten and G. C. Murphy. "Mylar: A degree-of-interest Model for IDEs. In AOSD '05: Proceedings of the 4th International Conference on Aspect-oriented Software Development, p159–168, 2005.

[Kitchenham2010]   B. Kitchenham, "What's up with software metrics? − A preliminary mapping study", The Journal of Systems and Software 83 , p37–51, 2010.

[Kitchenham2007]   B.A. Kitchenham, S.M. Charters, "Guidelines for Performing Systematic Literature Reviews in Software Engineering", EBSE Technical Report EBSE-2007-01, Keele University and Durham University, Version 2.3, 2007.

[Ko2005]   A. J. Ko, H. Aung, and B. A. Myers, "Eliciting Design Requirements for Maintenance-oriented Ides: A Detailed

Study of Corrective and Perfective Maintenance Tasks", In ICSE '05: Proceedings of the 27th International Conference on Software Engineering, p126–135, 2005.

[Koponen2005]   T. Koponen and V. Hotti, "Open Source Software Maintenance Process Framework", International Conference on Software Engineering: Proceedings of the fifth workshop on Open Source Software engineering, p 1-5, 2005.

[Koru2007]   A. G. Koru, D. Zhang, and H. Liu, "Modeling the Effect of Size on Error Proneness for Open-Source Software" Third International Workshop on Predictor Models in Software Engineering (PROMISE'07), IEEE, p 10, 2007.

[Li2006]   P. Li and E. Wohlstadter, "View-based Maintenance of Graphical User Interfaces", Proceedings of the 7th International Conference on Aspect-oriented Software Development, p 156-167, ACM, 2008.

[Li2008]   Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have Things Changed Now? : An Empirical Study of Bug Characteristics in Modern Open Source Software", Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, p 25 – 33, ACM, 2006.

[Livadas1994]   P. E. Livadas and D. T. Small, "Understanding Code Containing Preprocessor Constructs", Proceedings of the 3rd Workshop on Program Comprehension, Washington, DC, IEEE Computer Society Press, Los Alamitos, CA, p 89-97, 1994.

[McMaster2008]   S. McMaster and A.M. Memon, "Call-Stack Coverage for GUI Test Suite Reduction", IEEE Transactions on Software

Engineering, Volume 34, Issue 1, p 99-115, 2008.

[Memon2001]   A. M. Memon, E. Pollack, and M. L. Soffa, "Hierarchical GUI Test Case Generation Using Automated Planning", IEEE Transaction on Software Engineering., Volume 27, No 2, p 144-155, 2001.

[Memon2002]   A. M. Memon, "GUI Testing: Pitfalls and Process", Computer, IEEE Computer Society, Volume 35, Issue 8, p 87 – 88, 2002.

[Memon2003]   A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing", Proceedings of the 10th Working Conference on Reverse Engineering, p27-64, Nov 2003.

[Mockus2000]   A. Mockus and D.M. Weiss, "Predicting Risk of Software Changes", Bell Labs Technical Journal, April-June 2000, pp.169-180.

[Mockus2002]   A. Mockus and J. D. Herbsleb, "Two Case Studies of Open Source Software Development: Apache and Mozilla", ACM Transactions on Software Engineering and Methodology (TOSEM), Volume11, Issue 3, p 309 – 346, July 2002.

[Mohapatra2001]   S. Mohapatra and B. Mohanty, "Error Prevention through Error Prediction: A Case Study at Infosys" ICSM '01: Proceedings of the International Conference on Software Maintenance, p 260 – 272, 2001.

[Mookerjee2005]   R. Mookerjee, "Maintaining Enterprise Software Applications", Communications of the ACM, Volume48, Issue11, p 75 – 79, 2005.

[Myers1995]   B. A. Myers, "User Interface Software Tools", ACM Transaction Computer Human Interact, Volume 2, Issue 1, p 64–103, 1995.

[Myers2004]   G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas, The Art of Software Testing, John Wiley and Sons, 2004.

[Phipps1999]   G. Phipps, "Comparing Observed Bug and Productivity Rates for Java and C++", Software Practice and Experience, Volume 29, Number 4, p 345–358, Wiley & Sons, Inc, 1999.

[Qin2007]   F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan, "Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures", ACM Transactions on Computer Systems, Volume 25, Issue 3, p235-248, August 2007.

[Raymond1999]   E. S. Raymond. "The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary", O'Reilly and Associates, p19-p30, Sebastopol, CA, USA, 1999.

[Reilly2000]   D. Reilly, "Top Ten Errors Java Programmers Make", [Available online] http://www.javacoffeebreak.com/articles/toptenerrors.html, the website is last accessed on 31/01/2008.

[Robinson2004]   J. Robinson, Software Design for Engineers and Scientists, Elseveir, 2004.

[Rountev2004]   A. Rountev, S. Kagan, and M. Gibas, "Evaluating the Imprecision of Static Analysis", In Proceedings of the ACMSIGPLAN- SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, p 14-16, 2004.

[Rutar2004]   N. Rutar, C. B. Almazan, and J. S. Foster, "A Comparison of Bug Finding Tools for Java", Proceedings of the 15th International Symposium on Software Reliability Engineering, ISSRE '04 (ISSRE'04), Volume 00, p 245-256, 2004.

[Samoladas2004]   I. Samoladas, I. Stamelos, L. Angelis, and A. Oikonomou, "Open Source Software Development Should Strive for Even Greater Code Maintainability", Communications of the ACM, Volume 47, Issue 10, p83-87, October 2004.

[Schroter2006]   A. Schroter, T. Zimmermann, R. Premraj, and A. Zeller, "Where Do Bugs Come from?", ACM SIGSOFT Software Engineering Notes, Volume 31, Issue 6, p 1-2, 2006.

[Sen2008]   K. Sen, "Race Directed Random Testing of Concurrent Programs", Conference on Programming Language Design and Implementation, Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, p 11-21, ACM, 2008.

[Shannon1996]   B. Shannon, "C Style and Coding Standards for SunOS", Sun Microsystems, Inc., Version 1.8, 19 Aug 1996.
[Available online]: http://jp.opensolaris.org/os/community/documentation/getting_started_docs/cstyle.ms.pdf, last accessed on 30/01/2007.

[Shittu2007]   Hakeem Shittu, "WinRunner in Simple Steps", Genixpress, 2007.

[Singh2007]   Y. Singh and B. Goel, "A Step Towards Software Preventive Maintenance", ACM SIGSOFT Software Engineering Notes

Page, Volume 32, Issue 4, Number 10, July 2007

[Smith1999]   D. Smith, "Designing Maintainable Software", Springer-Verlag, 1999.

[Sommerville2000] I. Sommerville, "Software Engineering (6th Edition)", Addison-Wesley, 2000.

[Stafford2003]   J. A. Stafford, "Software Maintenance as Part of the Software Life Cycle", Technical Report, Department of Computer Science, Tufts University, 2003. [Available online at]: http://hepguru.com/maintenance/Final_121603_v6.pdf

[Staiger2007]   S. Staiger, "Static Analysis of Programs with Graphical User Interface", Proceedings of the 11th European Conference on Software Maintenance and Reengineering, p 252-264, IEEE Computer Society, 2007.

[Stewart2005]   K. J. Stewart, D. P. Darcy and S. L. Daniel, "Observations on Patterns of Development in Open Source Software Projects", Fifth Workshop on Open Source Software Engineering (5-WOSSE), St Louis, MO, USA, p1-5, May 2005.

[SunMicrosystems2006]

Sun Microsystems, "JavaTM Platform, Standard Edition 6 Overview", 2006,   [Available online] http://java.sun.com/javase/6/docs/technotes/guides, the website is last accessed on 25/06/2008.

[Tomb2007]   A. Tomb, G. Brat and W. Visser, "Variably Interprocedural Program Analysis for Runtime Error Detection", International

Symposium on Software Testing and Analysis, Proceedings of the 2007 International Symposium on Software Testing and Analysis, p 97 – 107, ACM, 2007.

[Ulrich1990]   W. Ulrich, "The Evolutionary Growth of Software Engineering and the Decade Ahead", American Programmer, Volume3, Issue10, p12-20, 1990.

[Van2000]   V. H. Van, "Software Engineering: Principles and Practices", 2nd Edition, John Wiley & Sons, West Sussex, England, 2000.

[Vipindeep2005]   V. Vipindeep and P. Jalote. "List of Common Bugs and. Programming Practices to Avoid Them", Technical report, Department of Computer Science and Engineering, Indian institute of technology, 2005.

[Xie2006]   Q. Xie, "Developing Cost-Effective Model-Based Techniques for GUI Testing", ICSE'06, Proceedings of the 28th International Conference on Software Engineering, p997-1000, May 2006.

[Xie2007]   Q. Xie and A. M. Memon, "Designing and Comparing Automated Test Oracles for GUI-based Software Applications", ACM Transactions on Software Engineering and Methodology, Volume 16, No 1, 2007.

[Ying2004]   A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll, "Predicting Source Code Changes by Mining Change History", IEEE Transaction on Software Engineering, Volume. 30, Number 9, p 574-586, Sept. 2004.

[Yu2006]   L. Yu, "Indirectly Predicting the Maintenance Effort of

Open-source Software", Journal of Software Maintenance and Evolution: Research and Practice, Volume 18, p 311-332, 2006.

[Yuan2007] X. Yuan and A. M. Memon, "Using GUI Run-Time State as Feedback to Generate Test Cases", 29th International Conference on Software Engineering (ICSE'07), p 396-405, 2007.

[Zheng2006] J. Zheng, N. Nagappan, J. P. Hudepohl and M. A. Vouk, "On the Value of Static Analysis for Fault Detection in Software", IEEE Transactions on Software Engineering, Volume 32, Issue, 4, p 240-253, April 2006.

[Zimmermann2005] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining Version Histories to Guide Software Changes", IEEE Transactions on Software Engineering, Volume 31, Issue 6, p 429 – 445, 2005.

Appendix

GUI	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
2voor12	0	1	0	0	1	0	0	0	0	0
frinika	0	0	0	0	0	0	1	0	2	0
galleon	0	0	0	0	0	0	0	0	0	0
Isql	0	0	1	0	0	0	0	0	0	0
Jdip	0	0	0	0	0	0	0	0	0	0
jfritz	0	0	0	0	0	0	0	0	0	0
jgammon	0	0	0	0	0	0	0	0	0	0
kennismanager	0	0	0	0	0	0	0	0	0	0
lmnga	0	0	0	0	0	0	0	0	0	0
mdevinf	0	0	0	0	0	1	0	0	0	0
nimbus-protocol	0	0	0	0	1	1	0	0	0	0
patcheditor	0	1	0	0	0	0	0	0	0	0
Qii	0	0	0	0	0	2	0	0	0	0
rptools	0	0	0	0	0	0	0	0	0	0
saiph	0	0	0	0	0	0	0	0	0	0
schedsim	0	0	0	0	2	3	0	0	0	0
sears	0	0	0	1	0	1	0	0	0	0
Sink	0	0	0	0	1	2	0	0	0	0
sprite2d	0	0	0	0	0	1	0	0	0	0
sudoku	0	0	0	0	0	0	0	0	0	0
turbogps	0	1	0	0	0	1	0	0	0	0
Ugt	0	0	0	0	0	0	0	0	0	0
vitapad	0	0	0	0	0	0	0	0	0	0
vlma	0	0	0	0	0	0	0	0	0	0
xfngraph	0	0	0	0	0	0	0	0	0	0

Data obtained from GUI code by the manual inspection

non-GUI	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
2voor12	0	0	0	0	0	0	0	0	0	0
frinika	0	0	0	0	0	0	0	0	1	0
galleon	0	0	0	0	0	9	0	0	0	0
isql	0	0	0	0	0	0	0	0	0	0
Jdip	0	0	0	0	0	0	0	0	0	0
jfritz	0	0	0	0	0	0	0	0	0	0
jgammon	0	0	0	0	0	0	0	0	0	0
kennismanager	0	0	0	0	0	0	0	0	0	0
lmnga	0	0	0	0	0	0	0	0	0	0
mdevinf	0	0	0	0	0	0	0	0	0	0
nimbus-protocol	0	0	0	0	0	1	0	0	0	0
patcheditor	0	0	0	0	0	0	0	0	0	0
Qii	0	0	0	0	0	0	0	0	0	0
rptools	0	0	0	0	0	3	0	0	0	0
saiph	0	0	0	0	0	0	0	0	0	0
schedsim	0	0	0	0	0	0	0	0	0	0
sears	0	0	0	0	0	0	0	0	0	0
sink	0	0	0	0	0	0	0	0	0	0
sprite2d	0	0	0	0	0	8	0	0	0	0
sudoku	0	0	0	0	0	0	0	0	0	0
turbogps	0	0	0	0	0	0	0	0	0	0
Ugt	0	0	0	0	0	0	0	0	0	0
vitapad	0	0	0	0	0	0	0	0	0	0
vlma	0	0	0	0	0	0	0	0	0	0
xfngraph	0	0	0	0	0	0	0	0	0	0

Data obtained from non-GUI code by the manual inspection

GUI	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
2voor12	0	0	0	0	0	2	0	0	0	0
daffodildb	0	0	0	0	0	0	2	0	0	0
druid	0	0	0	0	0	0	0	0	0	0
easyos	0	0	0	0	2	1	2	0	0	0
ephx	0	0	0	0	0	0	0	0	0	0
fate	0	0	0	0	0	0	0	0	0	0
findbugs	0	0	0	0	0	6	0	0	0	0
frinika	0	0	0	0	4	0	1	0	9	0
galleon	0	0	0	0	1	11	0	0	0	0
ganttproject	0	6	0	2	4	12	0	0	0	0
gface	0	0	1	0	0	1	0	0	0	0
ghpmathproj	0	0	0	0	0	0	0	0	0	0
grapher	0	0	0	0	0	2	0	0	0	0
gumbo	0	0	0	0	0	0	0	0	0	0
internet-café	0	0	0	0	0	0	0	0	0	0
jabref	0	2	2	1	1	26	1	0	1	0
jazzy	0	0	0	0	0	0	0	0	0	0
jfreereport	0	0	0	0	0	0	0	0	2	0
jfritz	0	5	5	0	0	2	0	0	2	0
jmri	0	2	0	3	1	47	2	0	1	0
juiml	0	0	0	0	0	0	0	0	0	0
jtrade	0	0	0	0	0	0	0	0	0	0
marf	0	0	0	1	0	0	0	0	0	0
mars-sim	0	0	0	8	0	7	0	0	0	0
mpeg7audioenc	0	0	0	0	0	0	0	0	0	0
musiccenter	0	0	0	0	0	0	0	0	0	0
myrpg	0	0	0	0	0	0	0	0	0	0
omegat	0	0	0	0	3	1	0	0	2	0
ourgrid	0	0	0	0	0	0	0	0	0	0
patcheditor	0	1	0	0	0	0	0	0	0	0
prefuse	0	0	0	0	0	3	0	0	0	0
profiler4j	0	0	0	0	0	0	0	0	0	0
qii	0	0	0	0	0	4	0	0	0	0
saiph	0	0	0	0	0	0	0	0	0	0
sink	0	0	0	0	1	5	0	0	0	0
sisw2006	0	0	0	0	0	0	0	0	0	0
soapui	0	4	0	2	5	20	0	0	4	0
sprite2d	0	0	0	0	0	4	0	0	0	0
triplea	0	0	3	2	1	18	1	0	14	0
ugt	0	0	0	2	1	8	0	0	0	0

unicats-i	0	0	0	0	1	8	3	0	0	0
vitapad	0	0	0	0	0	0	0	0	0	0
vlma	0	0	0	0	1	1	0	0	0	0
xfngraph	0	0	0	0	1	0	0	0	0	0
xui	0	0	0	0	0	59	1	0	1	0

Data obtained from GUI code by the automatic inspection

non-GUI	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
2voor12	0	0	0	0	0	0	0	0	0	0
daffodildb	0	0	3	3	1	19	0	0	178	0
druid	0	0	0	0	0	0	0	0	0	0
easyos	0	0	0	0	0	0	0	0	0	0
ephx	0	0	0	0	0	0	0	0	0	0
fate	0	0	0	0	0	0	0	0	0	0
findbugs	0	1	0	0	5	24	1	3	0	0
frinika	0	0	0	14	4	19	2	0	5	0
galleon	0	0	0	0	1	14	0	0	0	0
ganttproject	0	0	0	12	6	18	0	2	0	0
gface	0	0	0	0	0	0	0	0	0	0
ghpmathproj	0	0	0	0	0	1	0	0	0	0
grapher	0	0	0	2	0	0	0	0	0	0
gumbo	0	0	0	0	4	8	0	0	0	0
internet-café	0	0	0	0	0	0	0	0	0	0
jabref	0	0	11	0	0	11	1	1	0	0
jazzy	0	1	0	0	0	2	0	0	1	0
jfreereport	0	0	0	1	0	0	0	0	9	0
jfritz	0	0	1	0	8	1	0	0	1	0
jmri	0	0	0	1	6	32	2	2	6	0
juiml	0	0	0	0	0	0	0	0	0	0
jtrade	0	0	0	0	0	0	0	0	0	0
marf	0	0	0	4	7	6	1	0	155	0
mars-sim	0	18	2	92	0	43	1	0	0	0
mpeg7audioenc	0	0	0	75	3	1	0	1	0	0
musiccenter	0	0	0	0	0	0	0	0	0	0
myrpg	0	0	0	0	0	0	0	0	0	0
omegat	0	0	0	1	1	4	0	0	0	0
ourgrid	0	0	0	0	0	0	0	0	0	0
patcheditor	0	0	0	0	0	0	0	0	0	0
prefuse	0	0	0	0	0	0	0	0	0	0
profiler4j	0	0	0	0	3	1	0	0	5	0
qii	0	0	0	0	0	0	0	0	0	0

saiph	0	0	0	0	0	0	0	0	0	0
sink	0	0	0	0	0	7	0	0	1	0
sisw2006	0	0	0	0	0	0	0	0	0	0
soapui	0	0	0	6	4	13	0	0	7	0
sprite2d	0	0	0	0	0	19	0	0	0	0
triplea	0	1	0	1	1	9	0	0	3	0
ugt	0	0	0	0	0	0	0	0	0	0
unicats-i	2	15	13	15	47	61	2	0	37	0
vitapad	0	1	0	0	0	0	0	0	0	0
vlma	0	0	1	0	0	0	0	0	0	0
xfngraph	0	0	0	0	0	1	0	0	0	0
xui	0	0	0	3	1	33	0	0	2	0

Data obtained from non-GUI code by the automatic inspection

GUI	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
scartamus	0	0	0	0	0	0	0	0	0	0
tailor	0	0	0	0	0	0	0	0	0	0
biogenesis	0	0	0	0	0	1	0	0	0	0
brico	0	0	0	0	0	0	0	0	0	0
carabiner	0	0	0	0	0	0	0	0	0	0
dabench	0	0	0	0	0	0	0	0	0	0
gate	0	0	2	13	6	27	1	0	4	0
jasmin	0	0	0	0	0	0	0	0	0	0
jbackup	0	0	0	0	0	0	0	0	0	0
jcolony	0	0	0	0	0	0	0	0	0	0
jnine	0	0	0	0	0	0	0	1	0	0
j-waste	0	0	0	0	0	0	0	0	0	0
kabeja	0	0	0	0	0	0	0	0	0	0
kroak	0	0	0	0	0	0	0	0	0	0
loke	0	0	0	0	0	0	0	0	0	0
magicbeans	0	0	0	0	0	0	0	0	0	0
mates	0	0	0	2	0	0	0	1	0	0
matrex	0	0	1	0	0	30	0	1	2	0
octopus	0	0	0	0	0	0	0	0	0	0
olitext	0	0	0	0	0	0	0	0	0	0
openphysics	0	0	0	0	0	0	0	0	0	0
originalsynth	0	0	0	1	0	0	0	0	0	0
pazaak	0	0	0	0	0	0	0	0	0	0
photopolis	0	0	0	0	0	0	0	0	0	0
solium	0	0	0	0	0	0	0	0	0	0
sqlbuilder	0	0	0	0	0	0	0	0	0	0

supermercado	0	0	0	0	0	0	0	0	0	0
teamedit	0	0	0	0	0	0	0	0	0	0
visualpseudo	0	0	0	0	0	0	0	0	0	0
xhack	0	0	2	3	0	0	0	0	0	0
xholon	0	0	0	0	0	5	0	0	0	0

Data obtained from GUI code of evaluation sample

GUI	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
scartamus	0	0	0	0	0	0	0	0	0	0
tailor	0	0	0	0	0	0	0	0	0	0
biogenesis	0	1	0	5	1	3	0	0	1	0
brico	0	0	0	0	0	0	0	0	0	0
carabiner	0	0	0	0	0	3	0	0	0	0
dabench	0	0	0	0	0	0	0	0	0	0
gate	0	1	1	2	12	52	2	4	1	0
jasmin	0	0	0	0	0	0	0	0	0	0
jbackup	0	0	0	0	0	0	0	0	0	0
jcolony	0	0	0	0	0	0	0	0	0	0
jnine	0	0	0	0	0	0	0	0	0	0
j-waste	0	0	0	1	7	1	0	0	0	0
kabeja	0	0	3	0	2	0	0	0	0	0
kroak	0	0	0	0	0	0	0	0	0	0
loke	0	0	0	0	0	0	0	0	0	0
magicbeans	0	0	0	0	0	0	0	0	0	0
mates	0	0	0	0	1	1	0	0	0	0
matrex	0	0	8	0	0	12	0	0	29	0
octopus	0	0	0	0	0	1	0	0	0	0
olitext	0	0	0	1	0	0	0	0	0	0
openphysics	0	0	0	0	0	0	0	0	0	0
originalsynth	0	0	0	0	0	0	0	0	0	0
pazaak	0	0	0	0	1	4	1	0	0	0
photopolis	0	0	0	0	0	0	0	0	0	0
solium	0	0	0	0	0	2	0	0	0	0
sqlbuilder	0	0	0	0	1	0	0	0	0	0
supermercado	0	0	0	2	0	0	0	0	0	0
teamedit	0	0	0	0	0	0	0	0	0	0
visualpseudo	0	0	0	0	0	0	0	0	0	0
xhack	0	0	2	0	3	7	0	0	0	0
xholon	0	0	0	1	66	3	14	0	0	0

Data obtained from non-GUI code of evaluation sample