2011

An empirical study of the Eclipse Project

Pitamber Tiwari

AN EMPIRICAL STUDY OF THE ECLIPSE PROJECT

by

PITAMBER TIWARI

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in
The Department of Computer Science
to
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2011

In presenting this thesis in partial fulfillment of the requirements for a master's degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

Pitamber Tiwari

05/12/2011

(Date)

THESIS APPROVAL FORM

Submitted by Pitamber Tiwari in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and accepted on behalf of the Faculty of the School of Graduate Studies by the thesis committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate of the work described in this thesis. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

May 12, 2011
Committee Chair
Dr. Wei Li (Date)

5/12/2011
Dr. Feng Zhu (Date)

5/12/2011
Dr. Mary E. Weisskopf (Date)

5-12-11
Department Chair
Dr. Heggere S. Ranganath (Date)

5/12/11
College Dean
Dr. John D. Fix (Date)

8/8/11
Graduate Dean
Dr. Rhonda Kay Gaede (Date)

ABSTRACT

School of Graduate Studies
The University of Alabama in Huntsville

Degree Master of Science College/Dept. Science/Computer Science

Name of Candidate Pitamber Tiwari

Title An Empirical Study of the Eclipse Project

In this research, we focused on the bugs (error fixes and non-error changes) in the Eclipse Project, which has 220 modules with 33,904 files, and 3,780,201 lines of code (LOC). We developed a tool to collect the bug data from the Eclipse CVS repository and Bugzilla. We analyzed the data by testing 10 hypotheses. Our research showed that total changes (error and non-error together), error changes, and non-error changes over the decade-long evolution had the Power-Law distribution at the module and file level. When we classified error changes into more detailed categories, we found that all but one bug category had the Power-Law distribution at the module level. At the file level, about two thirds of the categories did not have the Power-Law distribution. Our research showed that there was no correlation between LOC, eLOC, lLOC, Comment, and Lines and the number of bugs at module and file level. However, a small percentage of modules having a large percentage of bugs also contained most of the code size (LOC). Nevertheless, a small percentage of files having a large percentage of bugs did not contain most of the code size.

Abstract Approval: Committee Chair

Department Chair Hoggere S. Lavgavetti

Graduate Dean Rhonda Kay Eade 8/8/11

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my thesis committee chair, Dr. Wei Li, for the support and guidance without which the work presented in this thesis would have not been possible. I would also like to thank the other committee members: Dr. Feng Zhu and Dr. Mary E. Weisskopf for the guidance in correcting the draft of my thesis. I would also like to thank my friend Deel Mani Baral for providing valuable suggestions and encouragement for the pursuit of this research.

A special thanks to M Squared Technologies for providing a free student license of the software metric tool for the purpose of my research. I would like to thank Aaron Clauset and colleagues for providing the Power-Law distribution functions.

Finally, I would also like to express my deep gratitude to my mother and sister and their continual support and encouragement throughout the course of my academic career.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Software plays an important role in our daily life. Software with bugs can lead to huge losses. Emphasizing the importance of bug-free software, Lo and colleagues [Lo et al. 2009] stated "*Due to the difficulty and complexity of software systems, bugs and anomalies are prevalent. Bugs have caused billions of dollars loss, in addition to privacy and security threats."* Improving software quality is an important goal of software development. Bugs present in software code compromise software quality. Understanding bug characteristics can help us find effective methods to reduce the number of bugs in software. One characteristic of bugs is their distribution in large software systems. The analysis of bug distribution is important from both scientific and practical points of view. Practically, we can apply it to the quality management, and scientifically, such research helps to enhance the understanding of software engineering [Andersson and Runeson 2007].

In this research, we have investigated the bug distribution in the Eclipse Project [Eclipse 2010]. In the Eclipse data, there is a bug Identification Number (ID) for each logged change. A change may be an error fix or non-error change. To be consistent with the Eclipse naming convention, we use bugs to represent changes (error and non-error) in the thesis. Eclipse is a large composite open source software system that is made of many projects. Each project is made of several subprojects.

Out of several Eclipse projects, we focused on the Eclipse Project. Eclipse Project consists of five major subprojects: e4, Eclipse Project Incubator (EPI), Java Development Tool (JDT), Plugin Development Environment (PDE), and Eclipse Platform (EP). Each subproject comprises of *modules* and *module alias*. A module is made of Java source files. A module alias is a virtual module that is made of Java source files that come from other modules. For example, Module A has two source files: A1.java and A2.java, and Module B has two source files: B1.java and B2.java. A module alias MA may be made of A1.java and B2.java. Every project or subproject under Eclipse Project is a combination of several modules and/or module aliases. The focus of this research is mainly on the bug distribution in two different granularity levels: module and file.

One main difference between our research and many others is the level of granularity of study. Many of the research studies were based on bug distribution at either the module level [Basili and Perricone 1984] [Moller and Paulish 1993] [Withrow 1990] [Zhang 2008] [Fenton and Ohlsson 2000] [Ohlsson and Alberg 1996] or file level [Ostrand et al. 2004] [Watanabe et al. 2008]. Our research investigated the bug distribution at both the module and file levels in one large open source system. Unlike some research [Basili and Perricone 1984] [Moller and Paulish 1993] [Withrow 1990] [Zhang 2008] [Fenton and Ohlsson 2000], where the studies were based on the bug data from proprietary systems, our data came from an open source system and the research results can be repeated and verified. Moreover, we have based our research on a larger number of modules with 3,780,201 LOC—the total number of lines in the source code excluding the comments and blank spaces—and 5,908,044 physical lines—the total lines in the source code including the blank lines, single braces or parenthesis. We have studied 220 modules.

Automated data collection and bug-prone module or file prediction have not been fully explored in the past [Ohlsson and Alberg 1996]. In our research, we have collected the bug information—module name, file name, revision number, bug number, and bug description—from the Eclipse CVS repository [Eclipse Repository 2010] using an automated crawler tool that we developed. We used the information to examine the bug distribution in each module and file. We used graphs and charts, formula functions, and sorting functions provided by Microsoft Excel, MatLab, and Minitab to analyze the data. Using the analysis, we have answered the questions related to Pareto Principle, how bugs are distributed in modules and files (Power-Law distribution or other), and relationship between bug rate and the size of module or file. Pareto Principle and Power-Law distribution will be discussed in Chapter 3. We used Resource Standard Metrics, a metric-collection tool, to analyze the source code for each module and file. Several source code metrics such as LOC, comments, effective lines of code (eLOC), logical lines of code (lLOC), and physical lines (Lines) of code are collected using the tool. We also classified the bugs into 19 categories to examine bug distribution in each category. We collected detail bug description from bug tracking system, Bugzilla, to make the bug classification process more reliable. Many of previous researches [Basili and Perricone 1984] [Moller and Paulish 1993] [Withrow 1990] [Zhang 2008] [Fenton and Ohlsson 2000] do not classify the bug into various categories to study the detailed nature of the bug distribution. They used either the module size or the data present in bug tracking systems [Shihab et al. 2010] as the main thread of analysis. Li and colleagues [Li et al. 2006] classified the bugs into various categories by using an automated bug classification tool. We have classified the bugs manually to achieve higher precision and reliability.

Our research will help enhance the understanding of software bugs and find effective methods to improve software quality. As the demand for complex and bug-free software increases, the need for software testing continues to grow [Shen et al. 1985]. Shen and colleagues stated, "*In attempting to meet this demand, successful producers of commercial software have learned two important principles.*

1. *It is extremely important that software be delivered to customers on schedule and with the fewest number of errors possible.*
2. *The cost of correcting program errors can (and typically does) increase enormously with time to discovery.*"

Because of this, interest in various approaches to detect the programming bugs as early as possible has been growing. One of the approaches is to test the entire software product in early phase of development. Exhaustive software testing is not always possible [Pressman 2005]. Another approach is to test certain portion of the software. We can decrease the software testing cost if we focus the testing effort on the files or modules where bugs are most likely to occur. For this approach of testing, the ability to predict the presence of bug in a large software system can be very useful. Understanding bug distribution can help us develop tools that predict bug-prone software artifacts (modules and files).

CHAPTER 2

LITERATURE REVIEW

A major cause of software failure, which leads to decrease in software dependability, is a software bug. Although there are many bug detection tools and extensive research [Basili and Perricone 1984] [Fenton and Ohlsson 2000] [Hong and Kim 1998] [Lipow 1982] [Moller and Paulish 1993] [Munson and Khoshgoftaar 1992] [Ohlsson and Alberg 1996] [Shen et al. 1985] [Thayer et al. 1976] [Withrow 1990] [Zhang 2008] [Zimmermann et al. 2007], bugs still exist in huge number in large software systems. In this chapter, we will present previous research works related to our study.

2.1 Predicting Bugs

The demand for reliable software is increasing. The software vendors must make sure that they remove as many bugs as possible from the software product before they release it to the end users. Software bugs become more costly as they propagate into the later software development stages. The Pareto Principle applies to software testing as a large percentage of bugs uncovered during testing is located in a small percentage of program components [Pressman 2005]. We can get the maximum benefit from testing if we can focus most resources to this small percentage of program components. The challenge is finding these bug-prone components. There is

no particular measure that we can apply to the software product to locate the files or modules that contain most of the bugs. There are several research studies [Basili and Perricone 1984] [Hong and Kim 1998] [Lipow 1982] [Munson and Khoshgoftaar 1992] [Ohlsson and Alberg 1996] [Shen et al. 1985] [Withrow 1990] [Zhang 2008] [Zimmermann et al. 2007] that have tried to predict bug-prone modules or files.

Withrow [Withrow 1990] studied the Ada software for command and control of a military communication system built by her colleagues at the Unisys. She studied 372 packages with 114,000 lines. The reports produced during the testing and integration phase were the main source of the data for the research. She studied 487 software bugs. In the study, the number of bugs found per package ranged from zero to 39. The size of the packages, used in the study, ranged from four to 5169 lines with an average size of 316 lines. This research used bug density—defects per thousand lines of code—as an inverse measure of quality. The software quality improves as the bug density decreases. Withrow presented an analysis diagram that significantly resembles to the diagram published by Shen and colleagues [Shen et al. 1985]. The diagrams showed that as the module size increases, the bug density decreases in a regular pattern. Her research showed that the bug density declined as the module size increased up to some optimal size. The bug density rises with size beyond that optimal size. Her research discovered the optimal size to be 255 lines. The optimal size may vary depending upon the programming language and style. This research suggests that the software tester should focus on testing modules of size either less than 200 lines or greater than 500 lines, depending on the project.

Shen and colleagues [Shen et al. 1985] studied a compiler written in PL/S, a derivative of PL/I with a goal to establish an empirical basis for the optimal utilization of limited testing resource. Through the detailed analysis of three software products

developed by IBM's Santa Teresa Laboratory and their bug histories, they have found simple metrics related to amount of data and the structural complexity of programs to be of value for predicting bug-prone software modules. They discovered as the software size increases, the bug rate decreases up to an optimal size. They showed this inverse relationship was true to a size of 500 lines. Their finding also suggests that it may be beneficial to promote programming practices as Withrow's research did.

Hong and Kim [Hong and Kim 1998] researched the identification of bug-prone modules in Automated Teller Machine (ATM) switching software developed using the CHILL language. They used the discriminant analysis and classification techniques and developed 11 software metrics and used their values to determine five domain metrics for each of 35 modules using factor analysis. The research discovered that we could use the values of domain metrics to predict the bug-prone modules. The model presented in the study could give two types of misclassification, Type I error—when a non-bug-prone module is classified as bug prone—and Type II error—when a bug-prone software module is classified as not bug prone. The model had a Type I error rate of 19% (5 of the 26 modules) and Type II error rate of 33% (two of the six bug-prone software modules). The model had 21.9% of overall misclassification in the testing data set. The model performed better, with 78% accuracy, when there were 32 software modules.

Muson and Khoshgoftaar [Munson and Khoshgoftaar 1992] also used discriminant analysis to explore the possibility of predicting bug-prone modules. They used data from two systems for the application of the discriminant analysis technique. They obtained the data for the first application from research done by Lind and Vairavan [Lind and Vairavan 1989] on a Medical Imaging System (MIS). The software consisted of 400,000 lines of code written in Pascal, FORTAN, assembler,

and PL/M. However, for this study, they used 390 programs, written in Pascal and FORTAN, consisting of 40,000 lines of code. For each of the programs, 11 complexity metrics were developed. Out of the 390 programs, they randomly selected 260 metrics for the creation and calibration of a model and used 130 routines for model validation. They grouped programs based on the number of changes made. The first group consisted of programs having zero or one change, the second group consisted of programs having 10 or more changes, and the third group consisted of programs having two to nine changes. There were 126 programs in the first group and 30 programs in the second group. They discarded the third group to magnify the difference between the characteristics of the program with changes. There could be Type I and Type II errors, similar to Hong and Kim [Hong and Kim 1998], in the prediction result of the model. They observed the rate of Type I and Type II errors for the two groups of programs in the validation set. For the first group of programs, there were four out of 38 Type I bugs. For the second group of programs, there were four out of 30 Type II bugs. They also used a probability of membership in any group to examine the performance of the model. The threshold was initially set to 0.9. Out of 130 programs, the number of programs having probability of membership in either the first group or the second group greater than the threshold value was 98. They assigned 53 out of 98 programs to the new Case I group and 45 to the Case II group. Similarly, data from two Ada projects for the command and control of the military data link communication system (CCCS) were used for the second application of the discriminant analysis technique. The system consisted of 327 modules. They computed metrics for all the modules. The data from the study showed that for a threshold value of 0.9, the rates of Type II bugs were 4 % for MIS and 1% for CCCS. When the threshold was 0.8, the rates of Type II bugs were 7% for MIS and 2% for

CCCS. At the threshold value of 0.9, the discriminant model used in the study was able to classify 75% of the MIS programs and 62% of the CCCS programs. As they decreased the threshold value, the percentage of programs classified increased with the cost of the Type II bugs.

In the research conducted by Ohlsson and Alberg [Ohlsson and Alberg 1996], an empirical study was carried out at Ericsson Telecom AB to determine methods that could predict the bug-prone module before the actual coding started. There were several sub-systems within the switching system studied in this research. The studied sub-systems consisted of 20 to 40 modules. They studied 130 modules whose sizes ranged from 1000 to 6000 LOC. They used the design and complexity metrics to determine the bug-prone software modules. The purpose of their study was to identify the bug-prone modules, such that it would help in resource planning and decrease the cost associated with testing. They analyzed and designed the documents automatically using the ERIMET tool. The study discovered that it was possible to build a prediction model, based on design metrics, to identify the bug-prone software modules, before the coding started. According to the study, 60% of the bugs were located in 20% of the modules. They also concluded that the size metric is not as significant as the other metrics used in their research.

Lipow [Lipow 1982] studied the relationship between the software size and the bug rate. He based his study on Halstead's software science metrics [Halstead 1997]. He showed that the number of bugs per line of source code increased with the increase in number of lines of code. He developed a model that predicted the number of bugs per LOC.

Basili and Perricone [Basili and Perricone 1984] studied the distribution of bugs within three releases of a general-purpose program for satellite planning studies.

The program consisted of 90,000 LOC. They categorized the modules into two types: modified—developed for a previous software project and modified to meet the requirements of the new project—and new—developed for the software project under analysis. Out of 96 modules with bugs, they classified 49% as modified modules and 51% as new modules. Both types of modules had high interface bugs. They grouped the modules based on the number of LOC. Out of 370 modules, only 10 modules had more than 400 LOC and the total number of bugs detected was 215. The maximum number of modules, 107, had 51 to 100 lines of code. Change data, collected within a period of 33 months, for the study was incomplete yet sufficient for conclusive results. They performed reviews to check the bug data. The bugs due to previous change increased from 2% to 6% because of the review. The study showed that there were more bugs corrections (62%) than non-bug modification (38%). In the study of bug distribution in modules, they found that it was possible to correct 89% of the bugs by changing a single module. This result indicated very low dependence among software modules with respect to bug. They found roughly equal bug distribution between the new and the modified modules. The largest number of bugs found in the study was seven for the new and five for the modified modules. They found the average number of bugs per module to be small: between one and three. The distribution study showed that the majority of the bugs (48%) were due to incorrect functional specifications or requirements. The study showed that the existence of the modified modules had hidden cost associated with it. The data in the research indicated that the cost to correct the bug in modified modules might outweigh the coding cost saved by modified modules. The effort associated with correcting the equal number of bugs was high for modified modules as compared to the new modules. The study, considering only the executable lines of code, showed that the smaller sized modules

had higher bug rates. Large modules, however, were more complex than the smaller modules. The study indicated average complexity of the bug-prone modules to be no more than the average complexity of all the modules.

Ostrand and colleagues [Ostrand et al. 2004] studied software bug behavior with a goal of being able to predict which files in software are most likely to have bugs in the next release. Data from two large industrial software systems, an inventory system and a provisioning system (in a telecommunication industry), with large number of releases were studied. For the inventory system, the data included bugs identified in all the development stages whereas for the provisioning system the data included the bugs identified only after the unit testing. The factors considered for the study were the logarithm of the number of lines of code; whether the file was new, changed or unchanged; the file age; the square root of number of bugs identified in the previous release; the programming language used; and the release number. The study of the inventory system, for 12 releases, showed the number of lines of code as the strongest bug predictor. The predictions for the inventory system as well as for the provisioning system were accurate. As the use of a negative binomial regression model required the knowledge of statistics, they also presented a simple model. In this model, they arranged the file by length and selected the 20% of the file. The research showed that for the inventory and provisioning systems, this model provided quite acceptable results, 10% lower accuracy than the full model. The study showed strong evidence to support the Pareto Principle and discovered LOC as the strongest predictor.

2.2 Distribution of Bugs

Studying the bug distribution not only helps to improve the software quality management but also helps to get better insight of software engineering. A study performed to understand the bug distribution is helpful in predicting the factors that play vital roles in the number of bugs per module or file. Understanding these factors is a major concern to software engineers [Basili and Perricone 1984]. There are several studies [Andersson and Runeson 2007] [Basili and Perricone 1984] [Fenton and Ohlsson 2000] [Hatton 1997] [Moller and Paulish 1993] [Ostrand and Weyuker 2002] [Zhang 2008] that have focused on bug distribution and used the results in predicting the bug-prone modules and files. In this section, we will present the previous studies on the distribution of bugs.

In the research conducted by Moller and Paulish [Moller and Paulish 1993], bug distribution within three versions of a software product, consisting of 750,000 lines of source code in which 200,000 to 400,000 lines had been modified, was studied. They studied the system used for controlling the utilization of general-purpose computers. They discovered that the bug distribution was not uniform throughout the code. The study showed that a high number of bugs concentrated in a small section of modified code. When there were changes in the source code, by modification or addition, the bug rate increased. The analysis of the data showed the percentage of new or modified code and the bug rate had an inverse relationship. Programming language had no effect on the bug rate for new modules. This was not the case for the modified modules, in which the structured Assembly language performed better than the normal Assembly language. For new modules of large size, the size had no effect on the bug rate. However, the new modules with less than

70 lines of code had a higher bug rate. The bug rate for the modified code varied inversely with the module size.

Fenton and Ohlsson [Fenton and Ohlsson 2000] studied 140 and 246 modules from two releases of the software from Ericsson Telecom AB. They collected the data from two successive releases of a project for the switching system and categorized it into pre-release and post-release bugs. They considered LOC, McCabe's cyclomatic complexity, and various metrics between the modules and within a module. They used the ERIMET tool to collect the complexity metrics automatically from a design document. They collected the bug data from four different phases: function test (FT), system test (ST), first 26 weeks on sites (SI), and first year of operation (OP). They denoted FT and ST bugs as testing bugs or pre-release bugs and SI and OP bugs as operational bugs or post-release bugs. They investigated a range of hypotheses related to the Pareto Principle on the distribution of bugs, prediction of bugs, metrics for bug prediction, and benchmarking of bug data. Their study showed that 20% of the modules were responsible for nearly 60% of the bugs found during the testing. Similarly, they found that 20% of the modules having 60% of the bugs make up 30% of the system size. They also discovered that 10% of the modules for first release had 100% of the operational bugs and 10% of the second release had 80% of the operational bugs. In this study, 100% of the operational bugs were contained in modules that made up 12% of the entire system size for the first release. For the second release, 60% of the operational bugs were contained in modules that made up 6% of the entire system size. Results of the study suggest that if we want to find the bug-prone modules during operation, then we must ignore the faulty modules found during testing. The study showed that modules with 50% of the bugs in the system test were responsible for 37% of the bugs in the functional test for the first release.

For the second release, modules with 50% of the bugs in the system test were responsible for 25% of the functional test. The study discovered that 10% of the modules with most bugs were responsible for 46% of the bugs in the system test and 24% of the bugs in the functional test. The study for first release found that 93% of the testing bugs (FT and ST) were located in the modules with no operational bugs (SI and OP). For the second release, 77% of the testing bugs were located in the modules with no operational bugs. The study showed that the small modules were more susceptible to bugs than the large modules. The study did not find size metrics such as LOC a good predictor for both pre-release bugs and post-release bugs in modules. However, the study indicated that the ranking ability of LOC, as analyzed using an Albert diagram, to be good at ranking the most bug-prone modules. Using the cyclomatic complexity metric, the most complex modules had more pre-release bugs, but had less post-release bugs. They found certain complexity metrics, extracted from early designs, better than cyclomatic complexity metrics alone. The study showed weak evidence that bug densities remained constant between subsequent releases of the system. Software systems developed in similar environments tend to have similar bug densities. Fenton and Ohlsson found evidence to support some of the commonly assumed hypotheses, but in several cases the data suggested the opposite to be true.

Andersson and Runeson [Andersson and Runeson 2007] performed a research to replicate the study of Fenton and Ohlsson [Fenton and Ohlsson 2000] to have a better understanding of the nature of the result. They analyzed data from three successive telecommunication projects. The study supported the hypothesis that a small number of modules contained the most bugs and found no evidence to support the hypothesis that if a small number of modules contained the most of the bugs, then

it was because these modules constituted most of the code size. They found some evidence to support that smaller modules were less likely to be bug prone than the larger ones. Like the Fenton and Ohlsson study, their study landed some support for the hypothesis that size metrics were good predictors of pre-release bugs in modules. Their study had only limited support to the hypothesis that size metrics were good predictors of post-release bugs in modules. Their research did not provide evidence that LOC is a good predictor of bugs.

Hatton [Hatton 1997] reexamined the connection between bug density and component size. The data for the study was collected from previous research [Compton and Withrow 1990] [Moller and Paulish 1993] and from NASA Goddard data compiled by the University of Maryland's Software Engineering Laboratory. In the study, he claimed that the reliability of the system improved only by high reuse within the same system, but low reuse degraded the reliability of the system. According to the study, if the components of a system are not large, we can calculate the overall system complexity by multiplying the complexity of each component. Hatton discovered that components with less than 200 LOC and larger than 400 LOC are more likely to have bugs. His study showed that bugs are not uniformly distributed but are rather concentrated on those components with 200 to 400 LOC.

Ostrand and Weyuker [Ostrand and Weyuker 2002] studied thirteen releases of a large inventory tracking systems at AT&T with a goal of finding the bug distribution to determine bug-prone files. They detected 4734 bugs from the system consisting of 1974 files and roughly 500,000 LOC. They formulated various hypotheses regarding the bug distribution over different files, effect of module size on bug density, bug densities in different software development stages, severity of the bugs, and persistence of bugs over releases. The data in the study showed that a large

number of bugs, for each release, were concentrated in a small number of files. The data also suggested that this concentration was partly because of the size of the files. The files with more bugs had more lines of code. Nevertheless, there was strong evidence that a small percentage of lines of code, for a matured product, accounted for a large number of bugs. They also discovered that the growth of percentage of bug was more rapid than that of code size. Their study regarding bug densities across various software development stages detected the majority of the bugs, more than 80% in most cases, during the development and unit testing phases. System testing and integration accounted for very few bugs as compared to the development and unit testing. This study showed large modules did not have more bugs as compared to smaller modules; instead, the opposite was true. The data in the study also suggested that the bug densities in the files did not increase as the file size increased. They discovered that for all the releases studied, the newer files had higher bug densities compared to the pre-existing ones. There was strong evidence supporting the Pareto Principle on the distribution of bugs, however.

Several of the studies used the Pareto Principle to describe the distribution of software bugs. Andersson and Runson [Andersson and Runeson 2007], Moller and Paulish, [Moller and Paulish 1993], and Ostrand and Weyuker [Ostrand and Weyuker 2002] all confirmed that the bug distribution followed the Pareto Principle. In his research, Zhang [Zhang 2008] discovered that the Weibull probability distribution function could describe the distribution of bugs more precisely. He replicated the study described by Andersson/Runeson and Fenton/Ohlsson using the Eclipse data collected by the University of Saarlan. He used module-level data for the study. Zhang suggested that we should use Weibull probability distribution function rather than Power distribution function (generally used to describe the Pareto Principle) to

model the bug distribution in the software system as it provides better representation of actual distribution.

2.3 Bug Classification

We can classify the software bugs into various categories to understand the nature of the bugs better. The classification of bugs into categories and its analysis help predict bugs based on different criteria such as severity, cost, and frequency. It also helps developers and testers manage their work efficiently; for example, the tester can focus the testing process on those modules and files more prone to highly severe bugs rather than the less severe bugs. Different studies have created their own bug categories to analyze the system under study. In this section, we present previous studies [Chandra and Chen 2000] [Li et al. 2006] [Owens et al. 1996] [Sullivan and Chillarege 1992] on the classification of bugs.

Sullivan and Chillarege [Sullivan and Chillarege 1992] studied the software bugs reported in two IBM databases management products, DB2 and IMS, and compared the result with their previous study [Sullivan and Chillarege 1991] in the IBM MVS operating system. The data were collected by analyzing the Authorized Program Analysis Report (APAR) report filed by IBM service experts to describe the bugs. Each APAR contained the details—programming mistake, solution, failure it caused, and environment under which the bug occurred—and represented the bugs that escaped the testing. In the MVS study, they used 150 APARs. They collected 222 APARs of DB2 and 201 APARs of IMS for the study of DBMS. They used the severity field present in the APAR to analyze the high impact bugs (the highest severity level). They classified the bugs into three categories: error type, defect type, and error trigger. They classified the low-level programming mistakes as error type.

Defect type addressed the design bugs, coding bugs, timing bugs, and administrative bugs. Error trigger classified ways to execute defective code that escaped the testing process. Error type set consisted of sub-categories such as allocation management, copying overrun, data, interface, memory leak, pointer management, statement logic, synchronization, uninitialized variable, undefined state, unknown, wrong algorithm, and other. Defect types were sub-categorized to function defect, data structure/algorithm defect, assignment/checking defect, interface defect, timing/synchronization defect, and build/package/merge defect. They categorized error triggers to workload, bug fixes, client code, recovery or exception handling, timing and unknown. The study showed that MVS had 18% high impact bugs and IMS had 19% high impact bugs. DB2 had higher, 30%, high impact bugs. The data of the study showed that the three systems had different error type distributions. Undefined bugs were the largest bugs in IMS and DB2 with 18% and 20% respectively. For IMS, there were 40% undefined bugs. Synchronization errors were the main cause of system failure for MVS. The study discovered that 20% recovery, 26% unusual workload condition, and 38% unusual timing were high impact bugs for MVS. Bug fixes and error handling were major categories that triggered high impact bugs in IMS. Workload and error handling triggered high impact bugs in DB2. There were fewer bugs in IMS than DB2 and MVS triggered in the workload sub-category. The study showed that DB2 had more function-related bugs than IMS. MVS had more timing-related bugs than DB2 or IMS.

Owens and colleagues [Owens et al. 1996] used a tool, Purify, to search for memory access bugs on C and C++ programs and proved that the bugs existed even after testing and caused failures. They tested 15 software packages consisting of 31 independent programs written in C and C++. They categorized five types of

memory access bugs: uninitialized memory read bug (umr), array bounds read bug (abr), array bounds write bug (abw), free memory read bug (fmr), and free memory write bug (fmw). They based the study on dynamic bug—the sum of occurrences for memory access bug types—and static bug—the number of source lines responsible for the memory access bugs. The study showed that in the dynamic bug data, uninitialized memory read bugs occurred most frequently (66%). The reason for this result, as seen by the sample of source code, was bad coding practices. The dynamic occurrences of the array bounds read bugs were 21%. Many of these bugs appeared to be due to mismanagement of dynamically allocated arrays designed to handle strings. Free memory read and memory write bugs, together, accounted for 11% of the total dynamic bugs. However, for the static data bugs, this type of bug accounted for 45% of the total. The study discovered that the write bugs were more catastrophic. The study discovered that the dynamic bugs with regard to uninitialized memory read for C and C++ programs were similar but the static behaviors were different.

Chandra and Chen [Chandra and Chen 2000] studied three open source systems: Apache web server, GNOME (GNU Network Object Model Environment) desktop environment, and MySQL database. They used the information present in the bug report and source code to classify the bugs. They classified the bugs based on the state or event outside the system under study. They classified the bugs into two main categories: environment-dependent and environment-independent. They further classified the environment-dependent bugs into transient and non-transient. They used 50 unique bug reports out of 5220 reports, 45 unique bug reports out of 500 reports, and 44 unique bug reports out of 44,000 messages for Apache, GNOME and MySQL respectively. The study found that there were 36 environment-independent bugs, seven environment-dependent-transient bugs, and seven environment-dependent-

nontransient bugs in Apache. There were 39 environment-independent bugs, three environment-dependent-transient bugs, and three environment-dependent-nontransient bugs in GNOME. There were 38 environment-independent bugs, two environment-dependent-transient bugs, and four environment-dependent-nontransient bugs in MySQL. The study found that there were 12 environment-dependent-transient bugs and 14 environment-dependent-nontransient bugs out of 139 bugs. The study discovered that 72% to 87% bugs were independent of the state or event outside the operating environment.

Li and colleagues [Li et al. 2006] investigated the impact of various factors like change in software development process, use of bug detection tool, shift towards multi-threading architecture, user-friendly interface, security, open source development paradigm, etc., on software bug. They analyzed 29,000 bugs from the Bugzilla databases for two large open source systems, Mozilla and the Apache HTTP server. They classified the bug in three dimensions: Root Cause, Impact, and Software Component. They further divided the Root Cause into Memory, Concurrent, and Semantic categories. They divided Impact into Hang, Crash, Data Corruption, Performance Degradation, Incorrect Functionality, and Unknown. The Software Component was categorized into Core, GUI, Network, and I/O. They divided memory bug and semantic bug into various sub-categories. Due to the large number of bugs, they used automated classification. They collected the bugs and randomly divided them into two sets: a training set for learning and tuning, and a test set for accuracy evaluation. They used the training set only for training and the test set for accuracy evaluation in order to avoid the bugs caused by tuning. Their analysis of the root cause showed that the Memory bugs, 16.3% in Mozilla and 12.2% in Apache, were less than 28% to 38% as discovered by Sullivan and Chillarege [Sullivan and

Chillarege 1991] [Sullivan and Chillarege 1992]. The authors suggest that it might be the impact of software tools. The study discovered semantic bugs to be the dominant root causes. The data in the study showed that the semantic bugs, 81.1% in Mozilla and 86.7% in Apache, were more than 55% to 66% as shown by Sullivan and Chillarege [Sullivan and Chillarege 1992]. Analysis of the Impact showed that incorrect functionality, 64.3% to 69.4%, was the dominant impact and was much larger than 35% discovered Sullivan and Chillarege [Sullivan and Chillarege 1992]. The study showed that the decrease in memory bug decreased the hangs and crashes. The study showed memory bugs to be the major cause for crash, 53.8% to 57.1%, and was similar to the results shown by Sullivan and Chillarege [Sullivan and Chillarege 1991]. This study discovered that semantic bugs had more impact on incorrect functionality and were among the major cause of unavailability. According to the study, concurrency bugs caused hangs. The analysis based on software components, like core, GUI, network and I/O, etc., showed that the GUI modules were critical for software reliability. The authors suggested that future research should pay more attention to GUI modules because most of the bugs, more than 50% in Mozilla, were concentrated in GUI modules compared to network and I/O. The relation between the Root Cause and the Software Components for the Mozilla showed memory related bugs as the root cause for bugs in the core modules. Semantic and concurrency related bugs caused GUI bugs. The security related analysis showed that the number of security related bugs was increasing in both Mozilla and the Apache HTTP server. Semantic bugs had more impact on security bugs than the memory related bugs in order to expose the system to security risk. This analysis contradicts buffer overflow as the most common form of security bugs shown by [Cowan et al. 2000]. The study

discovered that concurrency bugs increased in 1999 and 2000 but decreased since. Concurrency bugs caused software hanging due to synchronization and deadlock.

2.4. Studies on The Eclipse System

Zimmermann and the colleagues [Zimmermann et al. 2007] studied three releases of Eclipse SDK—that includes the Eclipse Platform, Java Development Tools, and Plug-in Development Environment—to predict the bug-prone modules or files. They studied 2.0, 2.1, and 3.0 releases, which consisted of 6740, 7900, and 6614 number of files respectively. Like our research, they collected the bug data from the Eclipse CVS repository and Bugzilla. They extracted the reference to bugs from the message present in the history of files. They used Bugzilla for the release of the bug to distinguish between the pre-release and post-release bugs. In their research, they computed several complexity metrics, for example, number of methods call (fans out), method LOC, number of parameters, and McCabe cyclomatic complexity metrics. They showed that we could use a combination of complexity metrics to predict the bug-prone modules and files. Their research illustrated that the number of bugs increased with the increase in the complexity of the software code.

Shatnawi and Li [Shatnawi and Li 2007] studied three releases of Eclipse projects—Versions 2.0, 2.1, and 3.0—to investigate the relationship between the bad smells and class error probability in three error-severity levels. The study was conducted in the context of the post-release system evolution process. They collected the data from the Eclipse change log and the Bugzilla database. They categorized the errors High, Medium, and Low-impact severity categories. They discovered that Shotgun Surgery, God Class, and God Methods bad smells were positively associated with the class error probability in the three error-severity levels.

Rahmani and colleagues [Rahmani et al. 2010] studied five open source projects—Eclipse, Apache HTTP Server 2, Firefox, MPlayer OS X, and ClamWin Free Antivirus—to compare the prediction capabilities. They compared three reliability models—Weibull, Yamada S-Shaped and Schneidewind—using the failure data of software products. The study discovered that a reliability model that fits the failure data well might not be a right forecaster of future failure patterns. They found that Schneidewind was the best prediction model for all systems but Firefox. The S-Shaped model was discovered to be the best prediction model for Firefox. They suggested that Weibull was the worst prediction model for all the systems.

Caglayan and colleagues [Caglayan et al. 2009] studied the Eclipse Project focusing on decreasing the false alarm rate by using the repository metrics. They studied three releases of the Eclipse Project—Version 2.0, 2.1, and 3.0. They collected 6727 files with 2609 defected files from the Eclipse Project version 2.0. The Eclipse Project version 2.1 had 7886 files with 2609 defected files. Similarly, the Eclipse project version 3.0 had 10,590 files with 1821 defected files. The study found that the repository metrics decreased the false alarm rates to 23% from 32% on average.

Shatnawi and Li [Shatnawi and Li 2008] studied three releases of Eclipse projects—Versions 2.0, 2.1, and 3.0—to understand whether the software metrics can predict the class error proneness in the post-release evolution of a system. Like our study, they collected the data from two sources: the Eclipse change log and the Bugzilla database. They collected the error data from the Bugzilla database for each Eclipse release and used the change log to find the classes that were affected by the error fixes. They grouped the errors into three severity categories: High, Medium, and Low-impact. They discovered that the accuracy of the prediction decreased from

release to release although some metrics could predict class error proneness in three error-severity categories. The study also found out that the prediction could not be used to build a metrics model to identify error-prone classes with acceptable accuracy.

Jiang and colleagues [Jiang et al. 2010] used a statistical approach to create explainable regression models. For the study, they performed a case study on the Eclipse project. They used the data from three releases of Eclipse provided by Zimmermann and the colleagues [Zimmermann et al. 2007]. They were able to identify three or four independent variables that explain the majority of the impact on the dependent variables. The study on the Eclipse project found that four out of 34 code and process metrics had impact on the likelihood of finding a post-release defect. They also showed that their model could achieve comparable performance over more complex PCA-based models.

CHAPTER 3

RESEARCH OBJECTIVES

There are three objectives of our research study. The first objective is to study the distribution of bugs both at the file and module levels. The second objective of our study is to study the relationship between the size and number of bugs both at the file and module levels. The final objective is to classify the bugs into different bug categories and study the distribution of bugs within each category. Our study increases the knowledge and understanding of bugs in large open source systems.

In our study, we have decided to study the Pareto Principle and Power-Law distribution. The Pareto Principle is universal in many aspects of our life and true in software development. In general, the Pareto Principle means a small percentage of the population causes the majority of the phenomena. In software error study, it means that there is a small percentage of software artifacts that contain the majority of the errors. The Pareto Principle is also known as the 20/80 rule and can be applied in several contexts. In sales, it means that 20% of the customers are responsible for 80% of the sales. In software testing, it means that 80% of the errors uncovered during testing can be traced back to 20% of the program components.

Power-Law distribution implies that small values are common, whereas large values are extremely rare. The distribution of the form $P(x) = Cx^{-\alpha}$. Here α is a constant distribution parameter called the exponent or the scaling parameter. The

Power-Law distribution has been found in many natural and social phenomena, for example, the frequency of occurrence of unique words in a novel and the population of cities in the United States. Cumulative distributions with the Power-Law distribution are sometimes said to follow a Pareto Principle distribution. Figure 3.1 shows the histogram plot of individual data following the Power-Law distribution [Adamic 2002]. Figure 3.2 shows the histogram of the cumulative percentage of data following the Power-Law distribution.

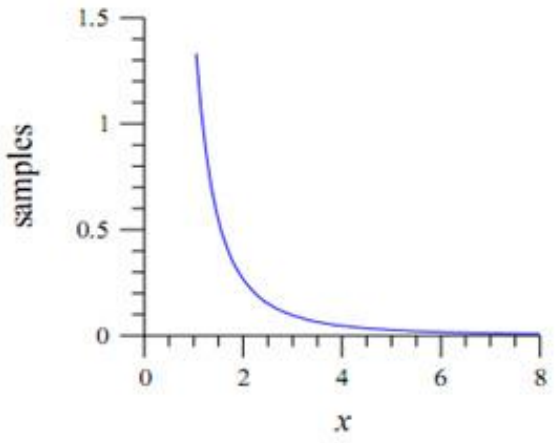


Figure 3.1: Histogram of individual data (x) following Power-Law distribution

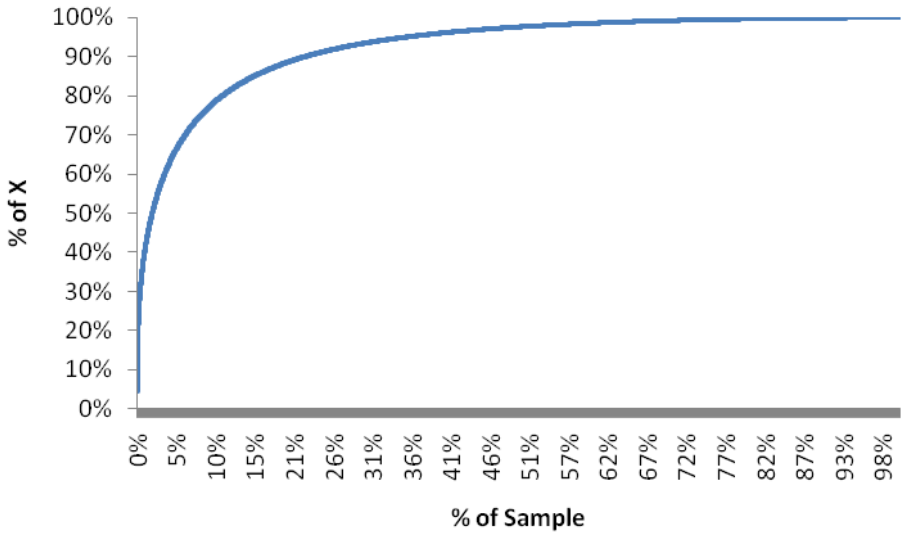


Figure 3.2: Graph of cumulative data (x) following Power-Law distribution

3.1 Research Questions

Our research will answer the following questions:

1. Does the Pareto Principle apply at module level?

2. Does the Pareto Principle apply at file level?

3. Do the LOC, eLOC, lLOC, Comment, and Lines metrics have any relationship with the number of bugs at module level?

4. Do the LOC, eLOC, lLOC, Comment, and Lines metrics have any relationship with the number of bugs at file level?

5. What percentage of changes are error fixes? What percentage of changes are non-error changes (feature enhancement, refactoring, adaptive maintenance, etc.)?

6. Do the error fixes and non-error changes have a Power-Law distribution at the module level?

7. Do the error fixes and non-error changes have a Power-Law distribution at the file level?

8. What are the various categories of bugs?

9. What is the distribution of bugs in each category at the module level?

10. What is the distribution of bugs in each category at the file level?

3.2 Research Hypotheses

By following the examples in some research [Andersson and Runeson 2007] [Fenton and Ohlsson 2000] [Gittens et al. 2005] [Timea and Barbara 2009], we set up the hypotheses as follows:

Hypothesis #1: Bugs are distributed in a Power-Law distribution at the module level; therefore, the Pareto Principle applies to modules.

Hypothesis #2: Bugs are distributed in a Power-Law distribution at the file level; therefore, the Pareto Principle applies to files.

Hypothesis #3: Modules containing the majority of the bugs contain most of the code size (LOC).

Hypothesis #4: Files containing the majority of the bugs contain most of the code size (LOC).

Hypothesis #5: The LOC, eLOC, lLOC, Comment, and Lines metrics have a linear relationship with the number of bugs in modules.

Hypothesis #6: The LOC, eLOC, lLOC, Comment, and Lines metrics have a linear relationship with the number of bugs in files.

Hypothesis #7: Both error fixes and non-error changes have the Power-Law distribution at the module level.

Hypothesis #8: Both error fixes and non-error changes have the Power-Law distribution at the file level.

Hypothesis #9: Bugs have the Power-Law distribution in each classification category at the module level.

Hypothesis #10: Bugs have the Power-Law distribution in each classification category at the file level.

We have decided to conduct tests on Hypotheses #1 and #3 to answer the first question. Similarly, the test results of Hypotheses #2 and #4 will answer the second question. We have decided to test Hypotheses #5 and #6 to find answers for the third and fourth questions respectively. We have decided to analyze the data for bug categories while testing Hypothesis #7 and find the answer to the fifth and the sixth questions. The results of the tests on Hypothesis #8 will provide the answer to the seventh question. The results of the tests on Hypothesis #9 will provide the answer to the eighth and ninth questions. Similarly, the test results of Hypotheses #10 will answer the tenth question. We have summarized the question-hypothesis relationship in Table 3.1.

Table 3.1: Question-hypothesis relationship

Question	Hypothesis
1	#1 and #3
2	#2 and #4
3	#5
4	#6
5	#7
6	#7
7	#8
8	#9
9	#9
10	#10

3.3 Research Design

To test the hypotheses, we collected bug data from 220 modules with 33,904 files from the Eclipse Project, a part of the Eclipse System. The number of bugs per module ranged from zero to 6365 and the number of bugs per file ranged from zero to 202. The total number of bugs in modules and files is 49,853. Modules and files have various sizes ranging from 20 LOC to 405,951 LOC per module and one to 48,596 LOC per file.

Aaron and colleagues [Aaron et al. 2009] reported, *"Many empirical quantities cluster around a typical value."* These quantities may vary somewhat, but they are very likely to follow a distribution, such as normal distribution, exponential distribution, Weibull distribution, Poisson distribution, or Power-Law distribution. Even though not all distributions follow a pattern, a Power-Law distribution occurs in many situations of software development.

O'Donogheu and colleagues [O'Donogheu et al. 2002] found a Power-Law distribution in frequencies of consecutive pairs of instructions executed by the virtual machine. The study conducted by Potanin and colleagues [Potanin et al. 2005] illustrated a Power-Law distribution in the incoming and outgoing references in object

graphs. Wheeldon and Counsell [Wheeldon and Counsell 2003] confirmed the Power-Law distribution in object-oriented class relationships. Hatton [Hatton 2009] illustrated the Power-Law in the equilibrium component size distribution in the system.

The identification of the range over which the Power-Law distribution exists is complicated. Commonly used mathematical methods may provide inaccurate results [Aaron et al. 2009]. There are very few situations in which all the values for x follow the Power-Law distribution. However, we can apply the Power-Law for values greater than some minimum value of x in many cases.

Relatively small numbers of modules or files are believed to have most of the bugs in software systems [Andersson and Runeson 2007] [Fenton and Ohlsson 2000] [Hatton 1997] [Moller and Paulish 1993]. This idea can be summed up by the Pareto Principle and is typically used to concentrate the effort on small and vital parts of a system [Juran et al. 1979]. We have decided to use an Arlberg Diagram, as suggested by Fenton and Ohlsson [Fenton and Ohlsson 2000], and used by Anderson and Runeson [Andersson and Runeson 2007] and Timea and Barbara [Timea and Barbara 2009], for testing Hypothesis #1 and #2. An Alberg Diagram is a line graph where the sorted cumulative total is represented by the line, for example, the line graph shown in Figure 3.2. The modules or files are sorted in decreasing order with respect to number of bugs. The cumulative percentage of bugs is plotted on the y-axis of the Alberg Diagram relative to the cumulative percentage of modules or files on the x-axis of the Alberg Diagram. For Hypothesis #1 and #2, we have decided to use the function built in Matlab as described by Aaron and colleagues [Aaron et al. 2009], to test whether the Power-Law distribution exists at the module and file level.

To test Hypothesis #3 and #4, we have decided to compute and analyze the sizes of modules and files that are responsible for a large percentage, 80% for example, of the bugs. We have decided to sort the modules and files in decreasing order relative to the number of bugs, and then plot the percentage of bugs and LOC on the y-axis with respect to the percentage of modules or files in the x-axis of the line graph.

For testing Hypothesis #5 and Hypothesis #6, we have decided to collect the number of bugs and metrics using the RSM tool, described in Chapter 7, to see whether a relationship between the metrics and bugs exists. We have decided to plot the dependent and independent variables on a graph called a *scatter plot*. A scatter plot is a graph of independent and dependent variables that helps in understanding the relationship between the variables [Bluman 1995]. In our study, the number of bugs is the dependent variable and LOC, eLOC, lLOC, Comment, and Lines are the independent variables. We will provide the details of these metrics in Chapter 7. We have decided to use the Pearson Correlation coefficient [Bluman 1995] to determine whether the two variables are related and to determine the strength of the relationship between those two variables.

We have decided to test Hypotheses #7, #8, #9 and #10 at both the file and module levels. For Hypothesis #7 and #8, we have decided to develop a list of bug categories. We will discuss the details of bug classification categories in Chapter 6. Based on these categories, we will categorize the bugs. We have decided to tabulate the number of bugs in each category. Some of the bugs represent changes that are not errors. For example, bugs related to enhancement, refactoring, and adaptive maintenance are non-error changes. Nonetheless, they all have Bug IDs in the CVS and Bugzilla systems. Other bugs may represent actual errors, such as errors in coding,

functionality, specification, memory, and event handling. For Hypotheses #7 and #8, we have two categories: error fixes and non-error changes. For Hypotheses #9 and #10, we have 19 different categories that will be discussed in Chapter 6. Once the data is collected, we have decided to test these hypotheses using the same process we use for Hypotheses #1 and #2.

CHAPTER 4

DATA COLLECTION

Open source systems not only provide a platform for engineers, scientists, students, and industries to share information and results in software development, they also provide an important data source for software engineering research. In this chapter, we present how we collected the data from the Eclipse Project.

4.1 The Eclipse Project

Eclipse, originally created by IBM and supported by a consortium of software vendors, is an open source Integrated Development Environments (IDEs) and tools. There are several projects under Eclipse and each project focuses on a technical area, although there are many common elements across all the projects. All work at Eclipse is done in projects, sub-projects and components organized in an hierarchical order [Eclipse 2010]. Among a large set of projects supported by Eclipse, we have focused our study on the Eclipse Project, one of several projects under the Eclipse. The Eclipse CVS repository arranges these projects and sub-projects in modules and module aliases. We studied the modules and files located inside these modules. Table 4.1 shows the project hierarchy in the Eclipse Project. It has five sub-projects: e4, Incubator, JDT, PDE, and Platform.

Table 4.1: List of Eclipse Projects

Eclipse Project
e4
Eclipse Project Incubator7
JDT - Java Development Tools
Core
Debug
UI
PDE - Plugin Development Environment
Build
Incubator
UI
Eclipse Platform
Debug
RelEng
Resources
Runtime
SWT
Team
Text
UA
UI

4.2 CVS Repository

The bug data logged by open source developers are crucial in software quality research as non-open source developers rarely publish their bug data for public research and analysis. Eclipse uses Concurrent Versioning System (CVS) repositories to keep track of all work and record all changes in project files, and allow developers, who are located all over the world, to collaborate efficiently. CVS is a version control system that maintains a history of the files throughout their development, allows retrieval of any stored version of file, supports production of multiple versions of files, and enables multiple developers to work simultaneously on a file without any loss of data [CVS and Eclipse 2010].

Eclipse provides a web interface for the CVS repositories. We can integrate the CVS repositories into the Eclipse Software Development Kit (SDK) and browse

the file hierarchy by clicking the directories that have forward slashes—'/'—after their names. The link names without the forward slashes are files. By clicking a file, we can view the revision history for that file. The history of a file records all the bug information related to that particular file and needed for our study. The index page for the Eclipse Project [Eclipse Repository 2010] inside the CVS repository gives the list of directories (modules) that contain the Java source files and their revision history.

4.3 CVS Module and Module Alias

Eclipse provides many plugins. For each plugin, there is a CVS module of the same name in the repository. These modules are grouped using *CVS module aliases*. A CVS module alias groups different files and directories into a single module, regardless of the file and directory location in the repository. Each component in the Eclipse repository is a combination of several plugin projects or modules. A component has a module alias, which references the plugin projects or modules that are part of the component [CVS Howto 2010]. For example, the JDT's UI component has an alias known as jdt-ui, which references all of the UI's plugin projects or modules (e.g., org.eclipse.jdt.debug.ui, org.eclipse.jdt.junit, org.eclipse.jdt.ui). We can also find module aliases for each component's tests and examples in the repository (e.g., jdt-ui-examples, jdt-ui-tests). However, there are no module aliases if there is nothing to reference.

In this study, we analyze bugs from individual modules as well as the files in those modules. A module alias may not use all the files in a module. Similarly, a project groups several modules or module aliases and may not use all the files in a module. Studying the individual modules and the files that constitute these modules

would be very helpful in understanding the bug distribution from two different granularities.

4.4 Bugzilla

Bug descriptions extracted from the revision history are short and, usually, provide very little information about the cause and nature of the bug. Mostly, the bug descriptions extracted from the CVS repository are the titles of the bug in the bug tracking system. They do not have enough information to describe the bug completely. Several times, we have found the bug description to be misleading and inappropriate. A bug description extracted from the revision history may even be empty, such that it makes the bug classification process very difficult, if possible at all.

Eclipse uses Bugzilla, a free bug tracking system, to keep track of its bugs [Bugzilla 2011]. Bugzilla helps developers track bugs and code changes that have happened throughout the resolution of the bugs. When someone reports a bug, a bug reporter enters the information that describes the bug. Depending on the nature of the bug and the knowledge of the reporter, the bug description may carry information related to the bug nature, steps to reproduce the bug, logs, and may even have suggestions to resolve the bug. Using the information present in Bugzilla, the developer understands and locates the bug and resolves the issues relating to the bug. Bugzilla facilitates the communication between developers, who can submit the changes and review changes submitted by other teammates. They post several comments and suggestions during the process of bug review and submission. The comments help developers gain a better understanding the bugs. For us, these comments and descriptions are very helpful in understanding the bug charactersitics.

To classify the bugs, we have extracted the descriptions from Bugzilla. The description extracted from Bugzilla contains the detailed behavior of the bug. It helps in making classification more reliable and accurate. The crawler tool examines the CVS repository to extract the module name, file name, revision, bug number, and bug description. After the bug number is stored in the database, the tool executes another script to take the bug number from the database and extract the description from the Bugzilla. We will describe the detailed process of collecting the bug descriptions from Bugzilla in Chapter 5.

4.5 Data Collection Methodology

It is easier to store every version of a file a developer created in a CVS repository, but storing every version of the same file would waste a tremendous amount of storage. In order to save the cost associated with disk storage, every version of a file is stored in a single file, such that only the difference between versions is stored. Since the data in the CVS repository are stored in a file, we cannot extract the data easily. Collecting the bug data from the Eclipse CVS repository is not a trivial task. Mostly, the bug data are not easily accessible. Even though Eclipse has a web interface for the CVS repository, the format of data stored in the CVS repository files does not meet the requirements of our study and we cannot easily extract them. Although there is an Eclipse plug-in, CVSChangeLog [CVSChangeLog 2011], that provides a summary of CVS log entries, it needs a tremendous amount of time and effort to collect bug information present in all the files using this plug-in manually. Over several evolution cycles, the bug data from all the Eclipse Projects is huge and difficult to collect manually. To speed up the process of data collection, we designed a crawler tool that collects the data from the online CVS repository logs. We can

access the logs accessed using a web interface to the Eclipse CVS repository. The CVS makes them available as normal HTML web pages. To collect the bug information from the log files, the crawler searches the Java files in the CVS repository and looks for bug information present in the revision log of those java files. We will explain the detailed design and working of the crawler tool in Chapter 5.

Each Java file in the CVS repository has records of revision history associated with it as shown in Figure 4.1. This revision log contains information such as file name, module name, revision number, the number of lines added or removed, version it differs from, name of modifier, date and time of modification, and messages. Messages present at the end of every revision of a file (as highlighted in Figure 4.1) carry the bug number and bug description necessary for this research. For our study, we have collected the module name, file name, revision, bug number, and bug description for each Java file.

The module name and file name can be extracted from the title of the log file, for example, the sample file is entitled '/org.eclipse.ui.intro.universal/src/org/eclipse/ui/internal/intro/universal/CustomizeAction.java'. The string enclosed between the first and second forward slash (/) is the module name, 'org.eclipse.ui.intro.universal' for the given file. The string enclosed between the last and second from last forward slash is the file name; 'CustomizeAction.java' for the given sample file. Usually, the module names start with 'org.eclipse' and the file name ends with '.java' extension.

We can find the revision number at the beginning of the revision history right after the drop-down list. A single file might have gone through several revisions, so there might be multiple revision numbers for a single file. The bug number and bug description associated with the revision are present at the end of the information as messages. Not all messages carry the bug information and bug description. For

example, two messages for Revisions 1.1 and 1.2 do not carry any bug numbers. The tool is designed to collect the information if the message starts with the keyword ‘Bug’ and has a bug number following it. For example, in Figure 4.1, we can see two bugs, Bug 218203 for Revision 1.4 and Bug 218203 for Revision 1.3. Here, both the bug numbers are the same. This means that the bug introduced in Revision 1.3 persisted in Revision 1.4. We may also find bugs that were across multiple Java files.

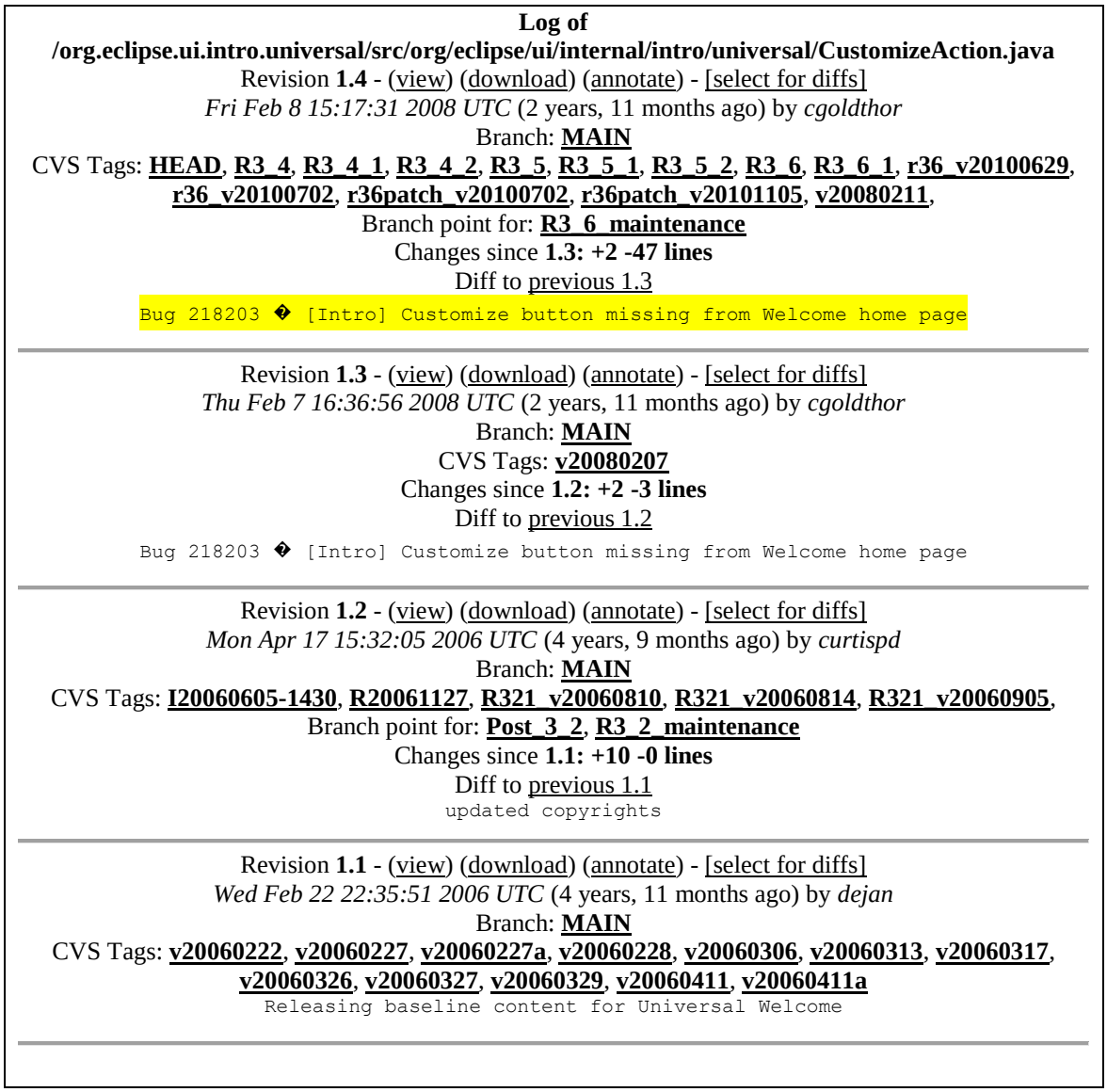


Figure 4.1: Revision log for Java files stored in the Eclipse CVS repository

We designed the crawler tool to extract information from the Java files located in the Eclipse CVS repository. As the crawler extracts information, it stores the information in a database table with the same name as the module name. This database is internal to our crawler tool and has nothing to do with the database in the CVS system. For example, org.eclipse.core is stored in a table named org.eclipse.core. The bug information stored in the database table is from the Java files located inside the modules in the CVS repository.

CHAPTER 5

THE CRAWLER TOOL

A *Crawler*, in general terms, is an automated computer program that browses the web pages of the Internet and collects the required information. Many software and search engines, such as Google, use a crawler to collect specific types of information, such as metadata. The Crawler tool we designed collects the bug information required for our study from the online Eclipse CVS repository and the Bugzilla bug tracking system. In this chapter, we provide the details of the tool design and working procedure.

5.1 Language Choice

The first step in designing the crawler tools is to pick a programming language. There are many languages available for building the tools. We considered Java, C, C++, Perl, Python, and PHP Hypertext Preprocessor (PHP) and decided to use PHP because of its plethora of built-in functions that are specifically designed to solve web-related problems [Lerdorf 2002], thus saving development effort for the tools. PHP is convenient when it comes to processing pattern-matching using regular expressions.

5.2 The Tool Design

The basic idea in the crawler design is to scan through the web pages and find the relevant information via pattern matching. There are several challenges in the design. First, the relevant bug information such as bug number, version number, file name (where the bug was found), and bug description are embedded in the mountain of irrelevant information in each web page. Therefore, we will have to design regular expressions to match the information of interest. Second, the number of pages on each website is nondeterministic, so we will have to cover all the pages dynamically for each system. Last, the retrieved bug data must be stored in persistent data storage for archiving.

The Crawler scans through the web Eclipse CVS repository pages that contain bug information of various Eclipse Projects and extracts it to store in a local database. The launching pad is the *index.php* file, which launches the user interface depicted in Figure 5.1. When a link to the module in the Eclipse CVS repository is clicked, the crawler tool will collect the bug data corresponding to that module.

Figure 5.1: Interface of the Crawler tool

5.3 The Architectural Design

The index.php file is the starting point for all the modules. Eclipse is a large system and processing all its data at once will take a long time, so we have divided the whole system into modules, as in CVS repository, for easier processing. The crawler is made up of a set of PHP files.

For all the PHP files other than the index.php and database.php for the Eclipse system, there is a folder named Eclipse. This folder contains four PHP files. Figure 5.2 shows the file hierarchy of the tool.

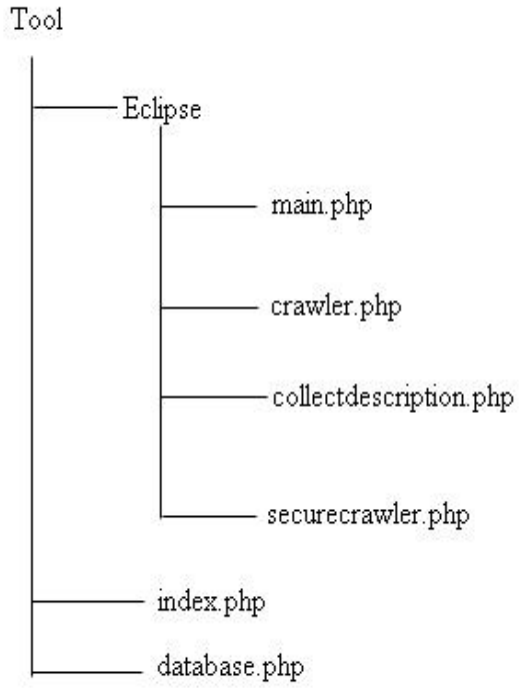


Figure 5.2: File Hierarchy for the Crawler tool

When we click a link corresponding to the module in the Eclipse CVS repository, the index.php file sends the action to the main.php in the eclipse folders. The main.php file begins with the core logic to collect the bug data for the system. A sample of the software module and corresponding actions passed to the main.php files is listed in Table 5.1.

The collected bug data are stored in a local MySQL database [MYSQL 2010], which maintains one table, with the module name, for the bugs collected from each module. All the modules present in the CVS repository share this database. The table attributes include Module Name, FileName, Revision, Bug Number and Bug Description. The Bug Number is an integer that uniquely identifies the bug. A primary key for the tables is the combination of all the attributes. The primary key guarantees that there are no duplicate entries in the table. The database.php file defines the database configuration. We have initialized the database user name, password, and host name in the database.php file. The database.php file is included in other PHP files if they need to connect to the database. After the initialization, connection to the database is established using the mysql_connect method, which takes three parameters: database name, user name, and password. Once the connection is established, the working database is selected by using the mysql_select_db method and database name as the parameter. The bug data can be exported to Microsoft Excel files via the export feature provided in MySQL phpmyadmin interface. Now, we will describe the detailed tool design of the crawler tool.

Table 5.1: Sample of modules and corresponding action passed from index to main

Eclipse Module	Action Passed
org.eclipse.ant.core	org.eclipse.ant.core
org.eclipse.compare	org.eclipse.compare
org.eclipse.help	org.eclipse.help
org.eclipse.jdt.apt.core	org.eclipse.jdt.apt.core
org.eclipse.jdt.core	org.eclipse.jdt.core

5.4 The Detailed Crawler Design for Eclipse CVS Repository

The Eclipse CVS repository contains huge amount of data and the crawler collects data by modules rather than the entire system at once. Although the Eclipse CVS repository web page does not have the any limitation on the amount of information collected, trying to collect the all data from the entire system at once takes several hours or even days.

When one of the module links is clicked, org.eclipse.ant.core for example, the tool sends the action, 'org.eclipse.ant.core', to the main.php in the Eclipse folder. The main.php file creates the Uniform Resource Locator (URL) of the Eclipse CVS repository web page and an object of the Crawler class, which is defined in the crawler.php file. It sends the URL to the constructor of the Crawler class. Upon receiving the URL, the constructor creates and allocates resources to the crawler object.

The algorithm of the crawler is as follows:

1. The index page sends an action to the main.php page as the user clicks the link representing each module in the CVS repository.
2. The main page creates the URL, database table, and activates the crawler to collect data.
3. For each web page
    a. The crawler scans the given URL and stores the entire page in a string variable.
    b. It collects the java file names located inside the module.
    c. It constructs the full link to the java file in the main page.
    d. For each file link

i. The page content that is associated with the link is stored in a string variable.
ii. The file revision information is extracted.
iii. The bug number and the bug description are extracted.
iv. The module name and the file name are extracted from the java file link.
v. The module name, file name, revision, bug number, and bug description are stored in the database.

The Sequence Diagram that depicts the algorithm for the crawler is in Figure 5.3.

The main.php file calls the get_Modules method with the title parameter to obtain the links to all the components within the module. The title is separated into different parts using the dot delimiter. The title is a module name, which consists of dot-separated names. The dot has special meaning in regular expressions, so it needs to be followed by a slash. The get_Modules method calls the PHP built-in method preg_match_all to parse the string in the variable *result*. Here, the preg_match_all call is used to extract the links of the contents inside module, with a pattern specified in regular expression. The preg_match_all method takes three parameters. The first parameter is a regular expression, *<a name="'.$escaped_title.'" href="/viewcvs\/viewvc\.cgi\/'.$escaped_title.'([^>]+)\>/*. The second is the string in the result variable. The third parameter is an array, *modules*, which is used as an output parameter to store the matched results. The regular expression is for searching the string that starts with the tag *<a*, followed by a name until it encounters >, which is again followed by the link pattern for the contents. This pattern matches all the links that are present in the module. The returned results are stored in the *module_links* variable in the main.php file.

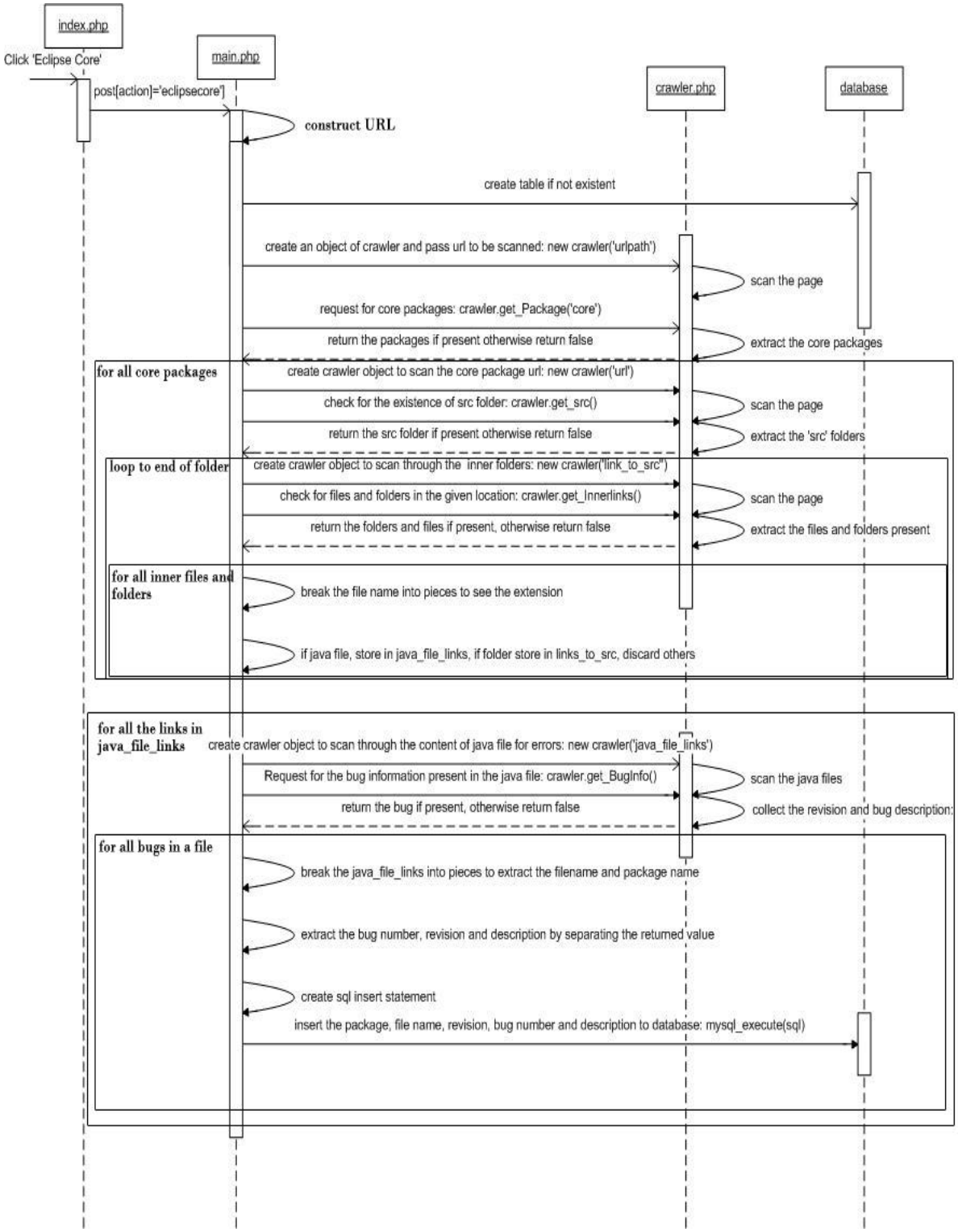


Figure 5.3: The sequence diagram for the Eclipse module

Now, the main.php file counts the number of links in the module_links and divides them into smaller groups. The module_links information is used to create the

full path of the next page, which is stored in the variable *link_to_src*. When the number of elements in the link_to_src reaches five, the main.php file creates an object of the Crawler class and passes the link_to_src as the parameter to its constructor. The constructor of the Crawler class again scans all the pages and stores the output in result. The threshold value of five is somewhat arbitrary and can be changed with the control of a variable in the program.

After creating a new crawler object, the main.php file calls the *get_LinksInsideModule* method in the Crawler class to look for the presence of folder inside the module. The presence of folder can be found from the presence of a link. The link is present in every page of the module having folder and files inside it. To extract this information, the get_LinksInsideModule method calls the preg_match_all method with three parameters. The first parameter is the regular expression, *name="([^"]+)" href="\/viewcvs\/viewvc\.cgi\/org([^"]+)*. The second is the input string, and the third is the output array, *LinksInsideModule*, used to store the matched results. This regular expression searches for any string that starts with *name* and followed by the link to the page until the double quotes (“) is encountered. This checks whether the input string has any folders or not. If a match is found, it is stored in the output array and the value is returned when it is not empty; otherwise, false is returned. The returned values are stored in the *links_inside_modules* variable in the main.php file.

Once the links to the folders present inside a module are extracted, all the files and folders inside these folders must be examined for the presence of Java files. If the links_inside_modules is not empty, the full path to the each folder is constructed in the main.php file and stored in the *link_to_src* variable. The constructed path is scanned for the presence of any files and folders. For this purpose, the link_to_src is

passed as a parameter from the main.php file to the constructor of the Crawler class when creating the Crawler object. Then the *get_InnerLinks* method of the Crawler class is called by the main.php file. This method extracts the links present in the given page. The links extracted by this method show the presence of files and folders inside the given path. The get_InnerLinks method calls preg_match_all method with three parameters to extract the links from the page. The first parameter is the regular expression, *name=" ([^"]+)" href="\/viewcvs\/index\.cgi\/org\.eclipse([^"]+)*. The second parameter is the input string, and the third is the output parameter used to store the matched outcomes. This regular expression searches for any string that starts with *name=* and followed by one or more character until the double quote is encountered, and followed by the string of HTML links. An example of string matched by this regular expression is *name="org" href="/viewcvs/index.cgi/org.eclipse.core.applicationrunner/src/org/"*. This example shows that there is a folder named 'org' inside the given path. The regular expression used in this method searches for any file or folder that is inside the given page, indicated by the format in the example. The matched results are stored in the *innerlinks* variable, which is returned if it is not empty; otherwise, false is returned. The returned value from the get_InnerLinks method is stored in the *innerpage_links* variable in the main.php file.

To identify file extension in order to determine if the link is a Java source file or a folder to be traversed, the string in the innerpage_links is broken down to extract the file name. The main.php file, after receiving the inner page links separates the innerpage_links into two parts using '/' as the delimiter and stores the result in the *separate* variable. For example, if the innerpage_links has *name="ApplicationRunner Activator.java" href="/viewcvs/index.cgi/org.eclipse.core.applicationrunner/src/org/eclipse/core/internal/applicationrunner/ApplicationRunnerActivator.java?view=log*",

the first part will be *name="ApplicationRunnerActivator. java"* and the second part will be *href="viewcvs/index.cgi/org.eclipse.core.applicationrunner/src/org/eclipse/core/internal/applicationrunner/ApplicationRunnerActivator.java?view=log"*. The second part is not helpful for extracting the file name. The first part of the *separate* is again separated into three parts using the double quotes as delimiter and the result is stored in the *java_filename* variable. The result from the above example, will be *name=* as the first part, *ApplicationRunnerActivator.java* as the second part and *href=* as the third part. The second part, which contains the file name with extension, is what we want. The other parts are ignored. Now, the name is separated into two parts by the dot delimiter and the result is stored in *java_extension_sep*. The first part of this separation is the file name and the second part is the extension of the file. If the extension is 'java', the full path to the file is constructed and stored in the java_file_links variable. If the extension is not 'java' and is not empty, the link is dropped. Otherwise, if the extension is empty, it is a folder to be traversed and is stored in the link_to_src variable. Using the link_to_src the main.php loops until the end of the path is reached. In doing this, it collects all the Java source file names and stores them in the java_file_links variable (array).

The above steps are repeated until all the core modules are examined. Once all the bug file names (with full path) are collected, these files are examined one by one to collect the bug information. For this purpose, the number of elements in java_file_links is counted in the main.php file. Each link (full-path file name) is passed to the Crawler object, which scans the content of the Java file. After the completion of the scan process, the main.php file calls the get_BugInfo method in the Crawler class to obtain the bug information including the revision, bug number, and bug description. The bug revision is embedded in a string such as 'Revision

1.8'. In order to extract the revision value, we must match all the string of the given pattern and then extract the values by removing the HTML tags . To extract the revision number, the first parameter for the preg_match_all method invocation is a regular expression, Revision *\<strong\>(.*)\<\/strong\>*. The regular expression searches for the string which starts with Revision, followed by zero or more number of characters enclosed within HTML tag . This regular expression looks for the pattern as stated in the above example. The second parameter is the result and the third parameter is revision, where the matched results are stored. After the revision is extracted, the bug number is extracted next by following a similar process with a different regular expression to match the bug number. The bug number is embedded in the bug description. So, the bug description is extracted by the get_BugInfo method first. The get_BugInfo method again calls the preg_match_all method to extract the bug description from the string stored in the result variable. The bug descriptions have the same pattern on every page. They are enclosed within the HTML tag <pre>. In order to extract the string matching the bug description pattern, the preg_match_all method uses *<pre ([^>]+)\>([^<]+)*, which is a combination of regular expressions. It searches for all the strings starting with the HTML tag <*pre*, followed by one or more characters until > is encountered, which is again followed by another one or more sets of characters until < is encountered. The matched string is stored in the *bug* variable. The bug description contains all types of descriptions. Some of the descriptions are not useful. Only the description that starts with the word 'bug' followed by a bug number is to be extracted. The bug description is separated into two parts: the first word and the remaining description. If the first word is either 'Bug' or 'bug', the remaining description string is the bug information. The resultant string and the

corresponding bug revision number are stored in the *bug_information* variable. Finally, the returned bug_information is stored in the *bug_info* variable in the main.php file.

The module name, file name, revision, bug number, and description are to be stored in the database. The link to the java file embeds the file name and module name pattern, such as *http://dev.eclipse.org/viewcvs/index.cgi/org.eclipse.core.commands/src/org/eclipse/core/commands/AbstractHandler.java?view=log*. To extract the module and files name, this link is separated into many strings, using the delimiter '/', and are stored in an array. The number of elements in the array is counted. The fifth element of the array gives the module name, for example, *org.eclipse.core.commands* in the example. The last element of the array is *AbstractHandler.java?view=log*. The last element is further separated into two parts using '?' as the delimiter. The first part of the separation gives the file name. After the extraction of module and file name, the bug revision is extracted. The bug revision is given by the even element of the bug_info array. Using the bug_info variable, bug number and bug description are extracted by separating the string into three parts. The second part is the bug number. In order to make sure that the second part is a bug number, the preg_match_all method is used to extract only the digits from the second string. The regular expression used to match the digits is */[0-9]+/*. The match stops when a non-digit character is encountered. Finally, the SQL *insert* query is created to insert the required information into the database. The module name, java file name, bug revision, bug number and bug description are inserted into the database using the insert query. The process of scanning the java file for bugs, extracting the bug information and inserting the information in the database continues for all the files in the java_file_links variable.

5.5 The Detailed Crawler Design for Bugzilla

Once the data collection from Eclipse CVS repository is complete, a detail bug description from Bugzilla can commence. As the mouse hovers over the *Collect Description from Bugzilla* button on the index page, a drop down menu appears showing the list of modules, corresponding to the database tables, from which the detail bug description is collected. When one of the menu items is clicked, Eclipse Core for example, the tool sends the action, '*eclipse_core*', to the *collectdescription.php* in the Eclipse folder. When the collectdescription.php is activated, it initializes the connection to the database by including the database.php file. It also includes the securecrawler.php.

The collectdescription.php file creates the Uniform Resource Locator (URL) of the Bugzilla web page needed for the crawler, that is, "https://bugs.eclipse.org/bugs/" and stores in a variable *urlpath*. Now, it creates a SQL statement to extract the distinct bug numbers from the database table. It executes the SQL statement, stores the resultant record set in a variable value, and loops until the value set has data in it. Inside the loop (for each bug), a link to the Bugzilla bug tracking web page is constructed using the bug number, that is, $urlpath."show_bug.cgi?id=".$value[0]. Here, $value[0] gives the distinct bug number.

Once the link to the bug tracking web page is constructed, a new securecrawler object is constructed and the link is passed to the constructor as a parameter. The securecrawler class is located in the securecrawler.php file. We called this a securecrawler because it crawls to the https—ssl encrypted—web page. The Bugzilla web page uses SSL encrypted pages for the purpose of security and to browse the web page, username and password is required. The crawler for Eclipse

CVS repository and the securecrawler for the Bugzilla differ in this sense only. Otherwise, both the crawlers are almost identical.

The algorithm of the crawler to extract bug description from Bugzilla is as follows:

1. The index page sends an action to the collectdescription.php page as the user clicks the button representing each module in the database table.
2. The collectdescription page creates the URL pointing to the Bugzilla home page.
3. The collectdescription extracts the distinct bug number from the corresponding database table.
4. For each distinct bug number
    a. The collectdescription creates the link to the bug tracking web page using the distinct bug number.
    b. The collectdescription creates an object of securecrawler located in the securecrawler.php file and passes the link, created in a, as a parameter to the constructor.
    c. The securecrawler allocates the required resources.
    d. The securecrawler scans the given URL and stores the entire page in a string variable.
    e. The securecrawler destroys the allocated resources.
    f. The collectdescription requests for the bug description.
    g. The securecrawler collects bug descriptions, corresponding to that particular bug, from the variable (from step d) that stored the page content.

h. The collectdescription process the bug description and stores in the database table along with other information collected from the Eclipse CVS repository.

As the constructor for securecrawler receives URL, it creates a PHP curl resource. It then extracts the content of the link and stores in a variable *result.* As soon as the content is copied to result, the curl resource in the securecrawler class is destroyed to utilize the resource and not to let any leaks. As soon as the content is copied in result, the control passes to the collectdescription. By calling the get_Description method located in securecrawler, the collectdescription requests the securecrawler to extract the bug description from the content stored in the variable result.

Upon receiving the request, the get_Description method in the securecrawler class checks the content stored in the result for emptiness. If the result is empty, there is nothing to extract; otherwise, there is a detail bug description representing the distinct bug number present. The bug in Bugzilla may be missing for some bugs present in the CVS, in which case the result may be empty. To extract the detail bug description, the get_Description method uses a PHP built in method *preg_match_all*. This built-in method takes three parameters as input argument, regular expression, input string, and the output array. The regular expression used to extract the bug description is <pre([^>]+)\>(.*?)\<\/pre\>. It matches any string enclosed between the HTML tag <pre></pre>. This string, matched by regular expression, gives the bug description required in the research for the classification. The second parameter to preg_match_all, input string, is the variable result, which has all the content of the web page stored as a string. The third parameter is an array, *bug_description*, which stores the matched result. Once the bug description is collected, it is returned to the

collectdescription. The returned result is stored in a variable description_bugzilla in the collectdescription. The description_bugzilla is now passed as a parameter to a built in method *addslashes*. This method returns a string with backslashes before characters—single quote (‘), double quote (“), backslashes (\), and NUL—that needs to be quoted in database queries, etc. Some of the descriptions may contain such characters, such that the addslashes method must be used before inserting the data to the database. Finally, the rows with the given bug numbers are updated using the SQL update query.

CHAPTER 6

DATA CLASSIFICATION

We started to classify the bugs after completing the data collection. The classification data can be useful in understanding the distribution of bugs in different categories. It can be helpful to understand the types of bugs the software system produced and analyze the defect and cause relationship. The bug classification data also helps manage project and software process improvement [IEEE Standard 2010].

The goal of the bug classification study is to find the occurrence of various categories of bug in the Eclipse Project. The Eclipse Project has a plethora of bug history. The history does not contain classification categories as provided by our study. Although Bugzilla provides the status of bugs, it lacks the information our classification provides. We followed the bug classification reference standard explained in Section 6.1 to classify the bugs into various categories.

6.1 Bug Categories

One of the main reasons to collect data from Bugzilla, in spite of having the data from the Eclipse CVS repository, is to make classification of bugs more accurate and reliable. The CVS repository logs and the Bugzilla database do not provide automatic bug classification. We have collected all bug attributes listed in Table 6.1 from the Eclipse CVS repository log files and the Bugzilla system.

Table 6.1: Bug attributes collected from the Eclipse CVS and Bugzilla files

Attribute	Definition
Module Name	The Eclipse Module containing the bug
File Name	The File containing the bug
Revision	Identification of the revision number in which the bug exists
Bug Number (ID)	Unique identifier for the bug
Bug Description (Title)	Brief description of the bug
Bug Detail Description	Full description of the bug behavior and condition under which it occurred (Extracted from the Bugzilla system)

Bug Description and Bug Detail Description are two attributes used in the classification of bugs into various categories. The repository log files and the Bugzilla logs do not contain the values related to the attributes, effect, mode, and type, of bug [IEEE Standard 2011]. We have defined the bug categories by combining these attributes. We have adapted the bug categories from different papers [Li et al. 2006] [Lo et al. 2009] [Pan et al. 2006], IEEE Standard Classification for Software Anomalies [IEEE Standard 2010], and books [Pressman 2005] [Beizer 1990] and added some of our own. Table A.1 in Appendix A lists the categories used in the study.

6.2 Classification Conflict Resolution

Bug categorization and classification are not trivial tasks. During the process of bug categorization, we have found that the same bug could be categorized into several different categories. There seems to be no universally correct way to categorize bugs to a particular bug category. The list shown in Table A.1 is the classification from our perspective for our study. This classification is not rigid. Bug categories, literally, can be infinite [Beizer 1990]. These categories are the result of our effort to combine various attributes such as effect, type, and mode.

Based on the history information, bug description, comments, reviews from developers, we may place a particular bug into one or another category. Even though a single character bug may pass the syntax checking by compiler, a single character bug, such as 50c for 50f, changes the statement. It results in an incorrect data initialization. This incorrect data when used in an expression leads to incorrect processing. When the output of this expression is returned as the output of a function, it leads to incorrect functioning. During the bug classification, this may be categorized into several bug categories. Looking at the situation, it may be considered a typographical bug, a data initialization bug, a coding conceptual bug, a processing bug, an incorrect function bug or sometimes we may even say this is due to lack of proper data validation [Beizer 1990].

There are several examples, where just the description of the bug is not enough. The description of the bug may be clear enough, but it may lead to a confusing situation, such as the one explained in previous paragraph. To make the bug classification more reliable, we went to the Bugzilla system for help whenever we encountered such situation. There are comments along with the bug description provided by the bug reporter, developer, and the person who resolved the bug. Although this process is time consuming, it is very helpful for correctly classifying the bug. We tried to resolve the conflicts based on the content of the comment. Sometimes, we might still place a bug in one of several categories. When the bug category was unknown, it was placed in the *general* category.

6.3 Classification Methodology

We started the bug classification after developing the bug category list. There are 49,853 bugs in 220 modules stored in 220 database tables. Out of 220 modules, 52 modules are bug free and we have classified the bugs in the remaining 138 modules in total.

We extracted the unique Bug Number (Bug ID), Bug Description, and Bug Detail Description from a single database table and stored them in a Microsoft Office Excel Worksheet. Once we copied the data to the Excel sheet, we read the Bug Description and Bug Detail Description for each bug. We assigned a suitable bug category after we were done reading. Whenever there was a conflict in bug classification, we searched for the given bug in the Bugzilla bug search page [Bugzilla a bug's life cycle 2011]. We went through the comments posted by the developers for help to resolve the conflict. We have classified a single bug into one category only.

Sometimes the Bug Description and Bug Detail Description might be missing. The Eclipse CVS repository log files might not contain the Bug Description and the Bugzilla system might be missing the Bug Detail Description. In such a case, we searched for the given bug number in the Bugzilla web page by entering the bug number into the search box. Once the bug page was loaded, we copied the title and first comment into the Bug Description and Bug Detail Description of the Excel sheet data and used this information to classify the bug. A single bug classification took us three to five minutes on average at the beginning. As we were acquainted with the process, it took us two to three minutes on average. Our approach of manual bug classification was time consuming, but more reliable than automatic bug classification with data mining tools.

CHAPTER 7

METRICS COLLECTION

In this chapter, we will present the tools used for the collection of software metrics from modules and files. We will also explain the details of the tool used, software metrics, metric calculation procedure, and present an example of metrics calculated by the tool.

7.1 Collecting Source Code from the Eclipse CVS

To calculate the number of files and LOC in a project, the project must be located in a local computer. To add the entire project from the Eclipse CVS repository to our local workspace, we need to connect to the repository and *Check Out* the entire Eclipse Project. *Check Out* is a process of importing files or projects from the repository into the local workspace. To make changes to the file or files in the repository, the user must have write permission. People who do not have the write permission to change the Eclipse code could access the Eclipse projects via anonymous access to the development CVS repository [CVS Howto 2010]. Users can check out the source code and modify it locally. Using the anonymous access, however, users cannot write the modified file back to the repository. Nevertheless, this is useful for us to get the source code of all modules to the local computer and analyze them using the metric tools. We created a connection to the CVS repository,

so that we could access the Eclipse Project files. We created an anonymous CVS connection manually. The steps we followed to set up a manual connection are as follows:

1. Run the Eclipse SDK.
2. Go to the '*CVS Repositories*' view, that is, select *Window* > *Show View* > *CVS Repositories* from the main menu.
3. Click '*Add CVS Repository*' button in the menu of CVS Repository.
4. In the new window that pops up, fill in the required information, identifying the required repository, as given below.

   Host: dev.eclipse.org

   Repository Paths: /cvsroot/eclipse (For the Eclipse platform project)

   User: anonymous

   Password: Leave blank

   Connection Type: pserver
5. Click '*Finish'*.

Once the process authenticated the login information, it showed the repository in the CVS Repositories view of the Eclipse SDK. We browsed the repository and added modules to our local workspace. An easy way to browse was to click the CVS and click the 'HEAD' tag so that the list would expand showing all the projects under it. Once we found the required project, we added the project to the local workspace by right clicking the project and selecting 'Check Out'. To add projects to our local workspace we used the following steps:

1. Run the '*Eclipse SDK'*.
2. Choose '*File* > *Import'* from the main menu of the SDK.

3. From the given list, select '*CVS > Projects from CVS'* and click '*Next'.*
4. Click '*Use existing repository location*' radio button, select the respective CVS repository from the list and click *'Next'*.
5. Click '*Use an existing module (this will allow you to browse the modules in the repository)*' radio button and select the project or projects you want to add.
6. Click '*Finish'.*

Once the check-out process completed, the project showed up in our local workspace. We saw the checked out projects by switching to Project Explorer view from CVS Repositories view in the Eclipse SDK.

7.2 Source Code Metric Tool

For the purpose of Java source code analysis, we used a metrics and analysis tool called *Resource Standard Metrics (RSM)*. *Resource Standard Metrics* is a source code metrics and quality analysis tool that supports C, C++, C#, and Java source code analysis across different operating systems [RSM 2011]. RSM is a command line program; however, there is a free and an easy to use graphical user interface that runs on top of RSM called RSM Wizard. We downloaded the free and fully functional trial version of both tools, RSM and RSM Wizard, from the website of the M Squared Technologies [RSM download 2011]. The free version of RSM limits the evaluation to 20 files at a time, but M Squared Technologies offered us a free and timed license for the duration of our project, removing the limit on the number of files per attempt by a trial version.

7.2.1 Installation of RSM

RSM and RSM Wizard were easy to install. All the required files were zipped and available for direct download. We downloaded and installed the files in Microsoft Windows Vista. The steps followed for the installation are as follows:

1. Unzip the downloaded zip file.
2. Execute the setup.exe program file and follow the directions.
3. After the completion of installation, copy the license provided by the organization, to the RSM installation directory.

7.2.2 Usage of RSM

Once the installation was complete, we were able to run the GUI and start analyzing source code. The tool could evaluate any number of files at a time due to the presence of the license file. Figure 7.1 shows the GUI of the RSM Wizard. The RSM Wizard had easy to use buttons with proper labeling on them rather than the complex command line arguments for RSM. This made it easy to use and simple to understand, such that we did not have to spend more time learning the tool. A free documentation was also available online at RSM home page [RSM 2011].

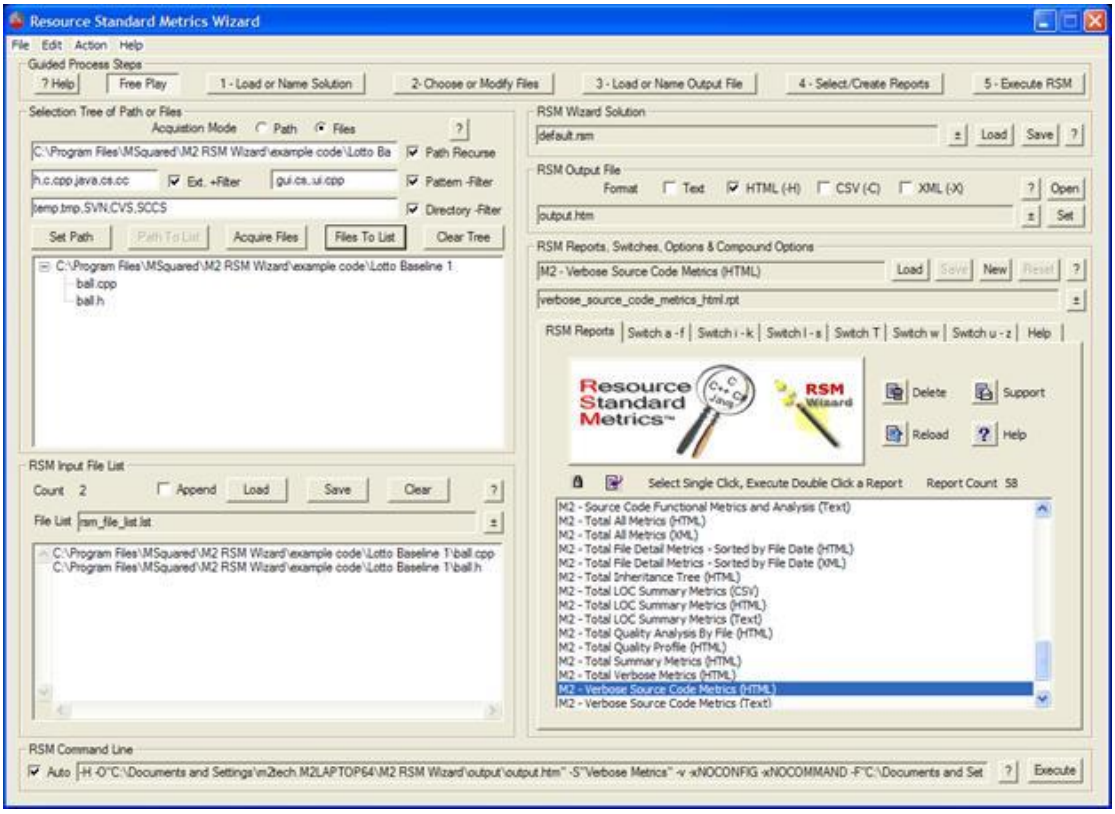


Figure 7.1: Graphical user interface of the RSM Wizard

In order to analyze the Java source code, we used the following steps:

1. Set the path of Java modules or file. To set the path, click the *'Set Path'* button and select the location of input module or file.
2. In the *'Ext. +Filter'* text box, enter Java. This filters other files and analyzes only the Java file.
3. Select the *'Path Recurse'* check box, so that the program would look for Java files inside the sub-directories recursively.
4. Click the *'Acquire Files'* button. This lists the files and directories under the selected path.
5. Click the *'Files to List'* button. This lists all the Java files available in the given location.
6. Select the output file format (e.g., Text, HTML, and CVS).

7. Set the output file name by clicking the *'Set'* button and supply the output file name.
8. Select the metrics you want in the report from the tabs available.
9. Click *'Execute'.*

After completing the analysis, we could see the report in the output file; for example, if we had selected HTML as our format, the detail analysis report appeared in the browser as plain HTML page.

7.3 Source Code Metrics

For our study, we used the default source code size metrics available for module and files. The various source code size metrics available were *Lines of Code Metric (LOC), Effective Lines of Code Metric (eLOC), Logical Lines of Code (lLOC), Comment,* and *Lines Metric.* The RSM generates these metrics by default operation [RSM metric definition 2011].

LOC is the total number of lines in the source code excluding the comments and blank lines. LOC, in general, represents the quantity of work done. RSM does not hide the process of metric determination for each line in the source file from the user. It can be viewed using the deterministic mode option in the RSM.

eLOC gives the count of the total number of source code lines that are not comments, blank lines, standalone parentheses or braces. The eLOC metric is a result of subtracting the total number of single braces or parentheses from the LOC metric. Many programmers have a practice of using single braces or parentheses to make source code clean and readable. This may lead to the increase of LOC metric. Thus, eLOC metric represents the quantity of work done more closely than the LOC metric.

lLOC represents the total number of lines in the source code that are terminated by a semicolon. The 'for' loop control statement contains two semicolons, however, it accounts for only one in the calculation of lLOC metric.

Comment represents the total lines of comments present in the source code. This metric is a general measure of the effort put by the developer in the program to make it more understandable and helpful in the maintenance phase.

Lines is the total lines in the source code including the blank lines, single braces, or parentheses. It is sometimes called the physical lines of code metric.

7.4 Metric Calculation Methodology

In this section, we show an example adapted from the documentation of RSM illustrating how the RSM tool determines the above-described metrics. In Table 7.1, the presence of a check mark (✓) indicates the particular line in the sample source code is included in the count of the respective metrics. The table indicates that the tool considered four lines of source code for the calculation of LOC, two lines of source code for the calculation of eLOC, one line of source code for the calculation of lLOC, two lines of source code for the calculation of Comment, and six lines of source code for the calculation of Lines.

Table 7.1: An example showing the process used by RSM to determine metrics

Source Code	LOC	eLOC	lLOC	Comments	Lines
if(score>90) //if statement test condition	✓	✓		✓	✓
{	✓				✓
//assign the grade				✓	✓
					✓
grade = 'A';	✓	✓	✓		✓
}	✓				✓

Figure B.1 in Appendix B shows the output of RSM for the module ‘org.eclipse.ui.intro.universal’, which has 23 Java files. The output contains all the details, including the license type, licensed to, license start and end date, etc. It shows the metrics for all individual files as well as the project summary for the module containing these files. In the project summary, it also reports the average per file metric. We used the metric data of individual files for the analysis at the file level and the project summary of each module for the analysis at the module level. From Figure B.1, for the org.eclipse.ui.intro.universal module, we see that there are 3342 LOC, 2793 eLOC, 1914 lLOC, 934 Comment, and 4460 Lines. Similarly, from the same figure, we can see that for the first file, BaseData.java, there are 16 LOC, 12 eLOC, 8 lLOC, 10 Comment, and 33 Lines. These metrics were calculated from the source code for the BaseData.java, given in Figure B.2, using the logic shown in Table 7.1.

CHAPTER 8

DATA ANALYSIS

In this chapter, we test the hypotheses and present the results.

8.1 Hypothesis #1: Bugs Are Distributed in a Power-Law Distribution at the Module Level; Therefore, The Pareto Principle Applies to Modules

This hypothesis is related to the 20-80 rule. The 20-80 rule can be applied in various contexts, but in this hypothesis it suggests that a large number of bugs are present in a small number of modules. An Alberg diagram and Power-Law distribution statistics were used to test this hypothesis. We collected 49,853 bug numbers that affected 220 modules of the Eclipse Project. The modules were sorted in descending order with relative to the number of bugs. Table 8.1 shows a summary of the distribution data for modules.

8.1.1 Power-Law Distribution for Modules Using an Alberg Diagram

We plotted the cumulative percentage of bugs on the y-axis with respect to the cumulative percentage of modules on the x-axis of the Alberg Diagram. Figure 8.1 shows the Alberg Diagram, for the distribution data in Table 8.1.

Table 8.1: Distribution of bugs in modules

Number of Modules	Number of Bugs	% of Modules	% of Bugs
11	30505	5	61.1898
22	40275	10	80.7875
33	43997	15	88.2534
44	45838	20	91.9463
55	47138	25	94.5539
66	47949	30	96.1807
77	48572	35	97.4304
88	48975	40	98.2388
99	49268	45	98.8265
110	49475	50	99.2417
121	49626	55	99.5446
132	49746	60	99.7853
143	49806	65	99.9057
154	49835	70	99.9638
165	49850	75	99.9939
168	49853	76.36	100

Figure 8.1 shows that 5% (11 modules) of the modules were responsible for approximately 61% of the total bugs, 10% (22 modules) of the modules were responsible for 80.79% (40,275 bugs) of the bugs, and 20% of the modules were responsible for about 92% (45,838 bugs) of the bugs. It also indicates that 76.36% of modules contained 100% of the bugs; that is, roughly a quarter of the modules did not report any bugs. Figure 8.1 illustrates that a large majority of bugs were concentrated in a small number of modules.

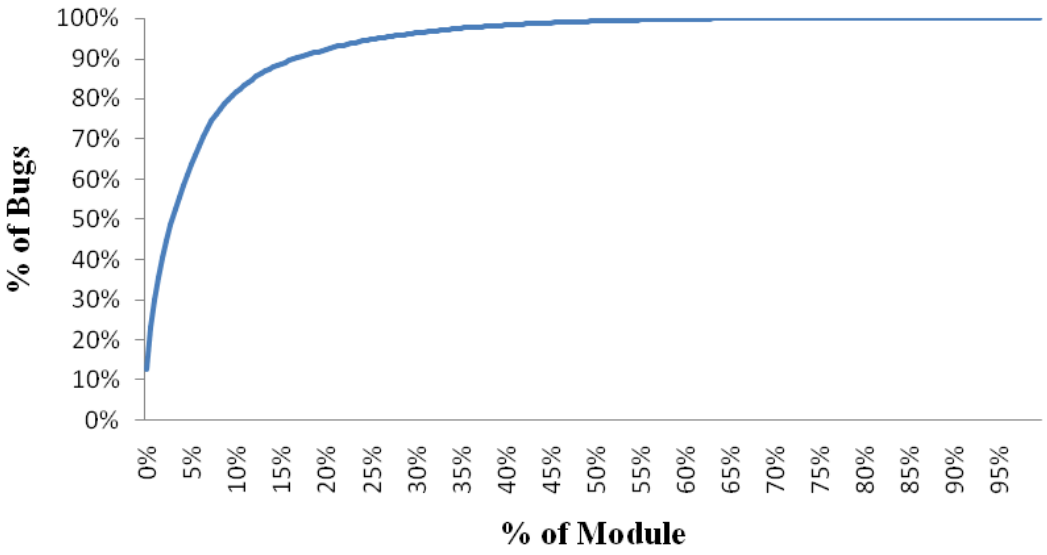


Figure 8.1: Alberg Diagram showing distribution of bugs in modules

8.1.2 Power-Law Distribution for Modules Using a Statistical Tool

We used functions built using Matlab, as described by Aaron and colleagues [Aaron et al. 2009], to test the Power-Law. Aaron and colleagues provided several Matlab functions to test the Power-Law distribution. We used the *plfit(x)* function, which estimated the lower cutoff value xmin and alpha by implementing the maximum likelihood estimator and the goodness-of-fit. This function used the number of bugs (x), given in Table 8.1, as input. The lower cutoff value xmin, as calculated by the function, was 53. To visualize the fitted distribution, we used the *plplot(x, xmin, alpha)* function, where x is the number of bugs in module and xmin and alpha are the values computed by the plfit function. This function plotted the data and the fitted Power-Law distribution in log axes. For the bug numbers given in Table 8.1, the plot is given in Figure 8.2. Figure 8.2 shows that most of the data fell on the straight line given by the Power-Law distribution.

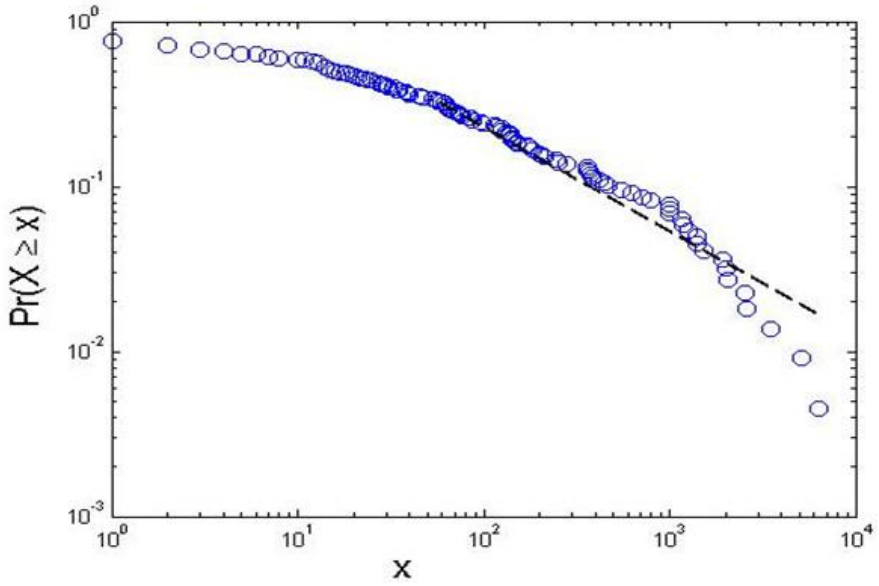


Figure 8.2: Log plot of number of bugs in modules and Power-Law distribution

Similarly, we used the function *plpva(x, xmin)* that took the number of bugs in modules (x) and lower cutoff value xmin to calculate the p-value for the Kolmogorov-Smirnov test. This function estimated the p-value over a number of repetitions. The default number of repetitions was 1000. The author claims that the p-value is more accurate as the number of repetitions increase. This was a slow process. In our case, it took roughly four hours to complete this single step. The time is dependent on the number of samples and the value of each sample. The p-value, for 1000 repetitions, computed by the function for the number of bugs in modules was 0.049. Based on the discussion by Aaron and colleagues, we can accept the Power-Law Distribution if the p-value is greater than 0.05; otherwise, the Power-Law Distribution is rejected.

Following the Alberg diagram and the log plot, as suggested in previous studies [Fenton and Ohlsson 2000] [Timea and Barbara 2009] [Andersson and Runeson 2007], we can see Power-Law distribution in a number of bugs at the module level. Hence, considering a threshold value of 0.05 and based on the analysis of Subsections 8.1.1 and 8.1.2, we conclude that Hypothesis #1 is confirmed.

8.2 Hypothesis #2: Bugs Are Distributed in a Power-Law Distribution at the File Level; Therefore, The Pareto Principle Applies to Files

This hypothesis is similar to Hypothesis #1 but at the file level. We collected 49,853 bugs data from 33,904 files located inside 220 modules in the Eclipse Project. Table 8.2 shows the distribution data for files.

8.2.1 Power-Law Distribution for Files Using an Alberg Diagram

We plotted the cumulative percentage of bugs on the y-axis of the Alberg Diagram with respect to the cumulative percentage of files on the x-axis. Figure 8.2 shows the Alberg Diagram for the distribution of bugs given in Table 8.2.

Table 8.2: Distribution of bugs in files

Number of Files	% of Files	Number of Bugs	% of Bugs
1696	5.0023	30149	60.4758
3391	10.0017	38443	77.1127
3871	11.4175	39883	80.0012
5086	15.0011	42911	86.0751
6781	20.0005	45820	91.9102
8476	25	47515	95.3102
10172	30.0023	49211	98.7122
10814	31.8959	49853	100

Figure 8.3 indicates that 5% of the files were responsible for 60% of the bugs and 12% of files contained more than 80% of bugs. Similarly, 20% of the files were responsible for more than 90% of the bugs. It also illustrates that approximately 35% of files contained 100% of the bugs; that is, more than half of the files did not report any bugs.

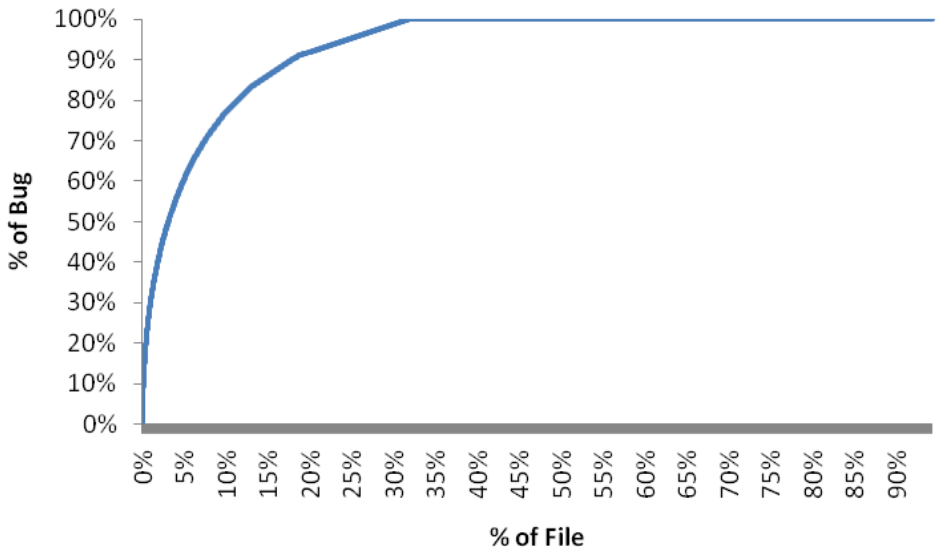


Figure 8.3: Alberg Diagram showing distribution of bugs in files

8.2.2 Power-Law Distribution for Files Using a Statistical Tool

To test the Power-Law distribution for files, we used the same functions described in Subsection 8.1.2. The lower cutoff value xmin for the data in Table 8.2 is nine. For the numbers of bugs given in Table 8.2, the plot is as given in Figure 8.4. Figure 8.4 shows that most of the data fell on the straight line given by the Power-Law distribution.

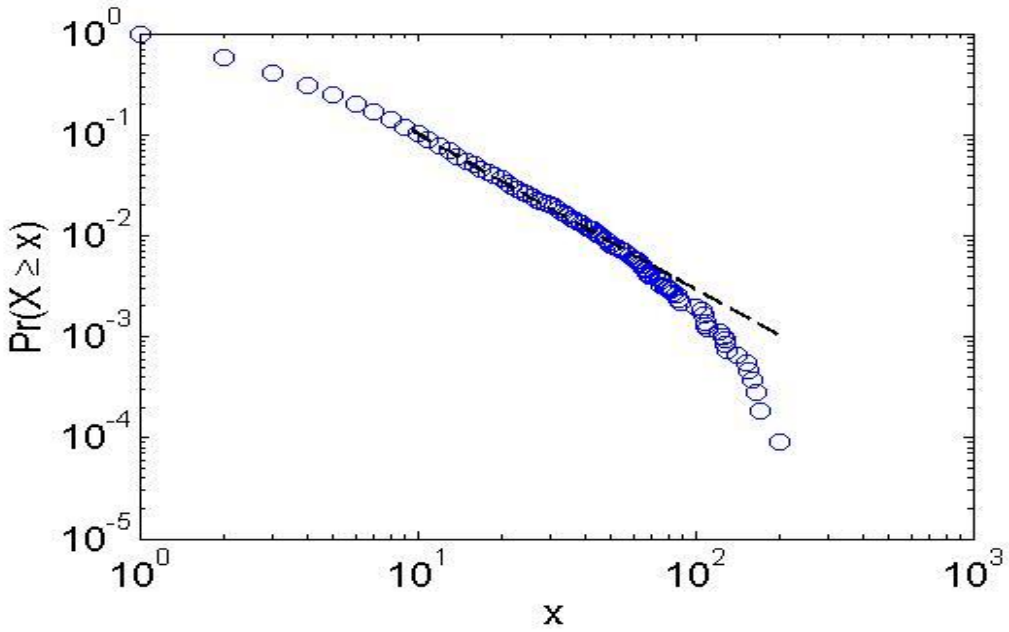


Figure 8.4: Log plot of number of bugs in files and Power-Law distribution

The p-value, for 1000 repetitions, for the number of bugs in the file is 0.106. Since, the p-value is greater than the threshold value of 0.05, we can say that the number of bugs at the file level followed the Power-Law distribution.

Based on the analysis of Subsections 8.2.1 and 8.2.2, we conclude that Hypothesis #2 is confirmed.

8.3 Hypothesis #3: Modules Containing the Majority of the Bugs Contain Most of the Code Size (LOC)

In order to test this hypothesis, we calculated the LOC for each module using the RSM tool. There were 3,780,201 LOC across 220 modules. Table 8.3 shows the LOC data for modules.

Table 8.3: Distribution of bugs and LOC in modules

Number of Modules	% of Module	Number of Bugs	% of Bugs	LOC	% of LOC
11	5	30505	61.1899	955419	25.2742
22	10	40275	80.7875	1970302	52.1216
33	15	43997	88.2534	2385225	63.0978
44	20	45838	91.9463	2569486	67.9722
55	25	47138	94.5539	3040524	80.4328
66	30	47949	96.1807	3135854	82.9546
77	35	48572	97.4304	3135931	84.9984
88	40	48975	98.2388	3136019	86.7712
99	45	49268	98.8265	3136118	88.1393
110	50	49475	99.2417	3136228	89.5554
121	55	49626	99.5446	3136349	90.4980
132	60	49746	99.7853	3136481	91.4807
143	65	49806	99.9057	3136624	93.4394
154	70	49835	99.9638	3136778	94.5781
165	75	49850	99.9939	3136943	95.3172
168	76.36	49853	100	3137111	95.3251

We sorted the modules and LOC corresponding to each module in descending order with respect to the number of bugs. We plotted the cumulative percentage of bugs and LOC on the y-axis with respect to the cumulative percentage of modules on the x-axis as shown in Figure 8.5.

Figure 8.5 indicates that 10% of the modules that were responsible for 80% of the bugs had approximately 50% of the LOC. Similarly, more than 65% of the total LOC belonged to the 20% of modules that were responsible for 91% of the bugs. We can see that small percentage of modules that had a large percentage of the bugs also had a large percentage of the total LOC.

Based on our analysis, we conclude that Hypothesis #3 is confirmed.

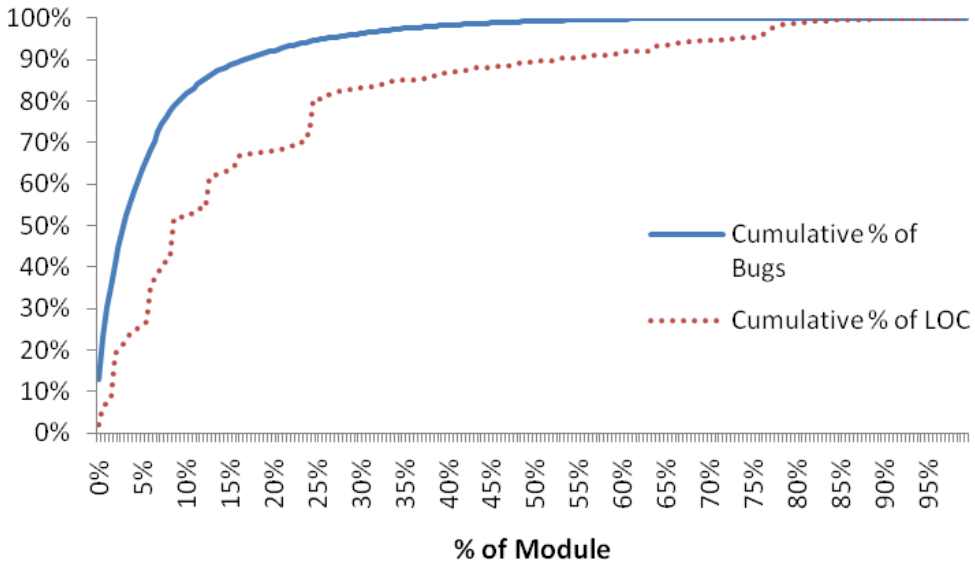


Figure 8.5: Line graph showing cumulative percentage of bugs and LOC in modules

8.4 Hypothesis #4: Files Containing the Majority of the Bugs Contain Most of the Code Size (LOC)

LOC of files may be the reason we found support for Hypothesis #2. It makes sense to test Hypothesis #4 as we confirmed Hypothesis #2. In order to test Hypothesis #4, we calculated the LOC for each file using the RSM tool. There were 3,780,201 LOC across 220 modules. Table 8.4 shows the LOC data for files.

We sorted the files and LOC for the files in descending order with relative to the number of bugs. We plotted the cumulative percentage of bugs and LOC on the y-axis with respect to the cumulative percentage of files on the x-axis. Since Microsoft Excel limits data to 32,000 points on a graph, we plotted the graph for 10,814 files only. These files contained 100% of the bugs as shown in Table 8.4. The plot for the distribution of bugs and LOC in files is as shown in Figure 8.6.

Table 8.4: Distribution of bugs and LOC in files

Number of Files	% of Files	Number of Bugs	% of Bugs	LOC	% of LOC
1696	5.0023	30149	60.4758	649563	17.1832
3391	10.0017	38443	77.1127	937014	24.7874
3871	11.4175	39883	80.0012	1000427	26.4649
5086	15.0011	42911	86.0751	1216662	32.1851
6781	20.0005	45820	91.9102	1411207	37.3315
8476	25	47515	95.3102	1538897	40.7094
10172	30.0023	49211	98.7122	1736287	45.931
10814	31.8959	49853	100	1796720	47.5297

Figure 8.6 indicates that roughly 5% of the files that were responsible for 60% of the bugs had 17% of the LOC. It shows that around 10% of the files that were responsible for 77% of the bugs had nearly 25% of LOC. Similarly, around 37% of the total LOC belonged to the 20% files that were responsible for 91% of the bugs. Roughly 31% of the files that were responsible for 100% of the bugs had 47% of the LOC. The gap between the cumulative percent of bugs and LOC in files is large, such that few files with the majority of the bugs did not have a large percentage of the code size.

Hence, we conclude that Hypothesis #4 is rejected.

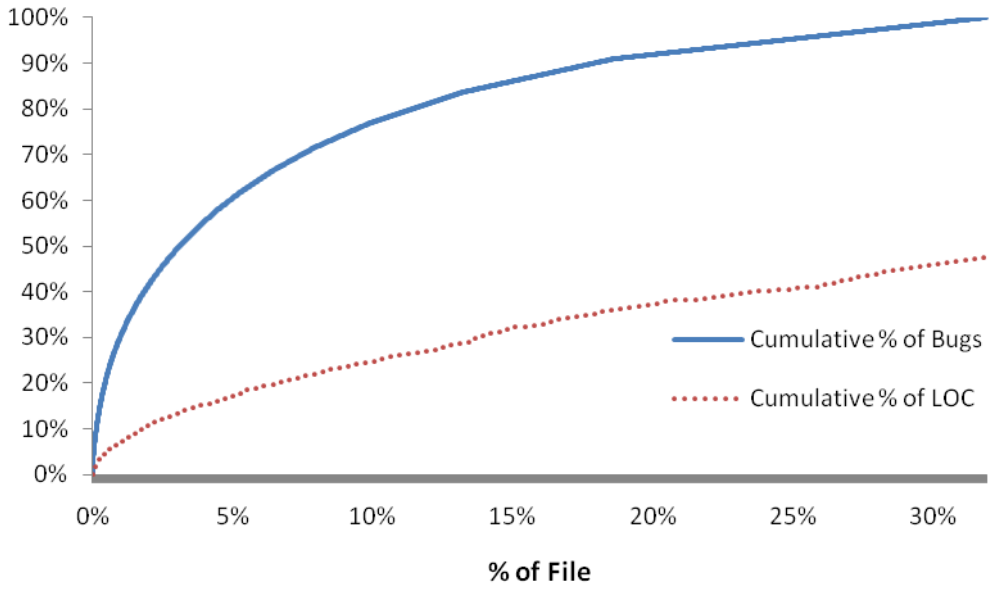


Figure 8.6: Line graph showing cumulative percentage of bugs and LOC in files

8.5 Hypothesis #5: The LOC, eLOC, lLOC, Comment, and Lines Metrics Have a Linear Relationship with the Number of Bugs in Modules

We used RSM tool to collect LOC, eLOC, lLOC, Comment, and Lines metrics, as explained in Chapter 7, for each module in the Eclipse Project to see whether a relationship exists between them. We constructed a table, in Excel, using the values. We drew scatter plots of the dependent and independent variables using Excel. Figure 8.7 (a) shows the scatter plot between LOC and number of bugs in modules. We cannot see any specific type of relationship between LOC and number of bugs from the plot of the data shown in Figure 8.7 (a). Figure 8.7 (b) shows the scatter plot between eLOC and number of bugs in modules. It does not indicate any specific type of relationship between eLOC and the number of bugs. Figure 8.7 (c) shows the scatter plot drawn between the number of bugs and lLOC. There is no indication of any specific relationship between the number of bugs and ILOC in the modules in Figure 8.7 (c). Figure 8.7 (d) shows the scatter plot between Comment and number of bugs in module. This scatter plot does not have any discernible pattern, so there is no indication of a specific relationship between Comment and number of bugs

in module. Similarly, we drew a scatter plot between the Lines and the number of bugs in modules as shown in Figure 8.7 (e). This plot does not have clear pattern. Therefore, we cannot specify any specific relationship between the Lines and number of bugs in a module.

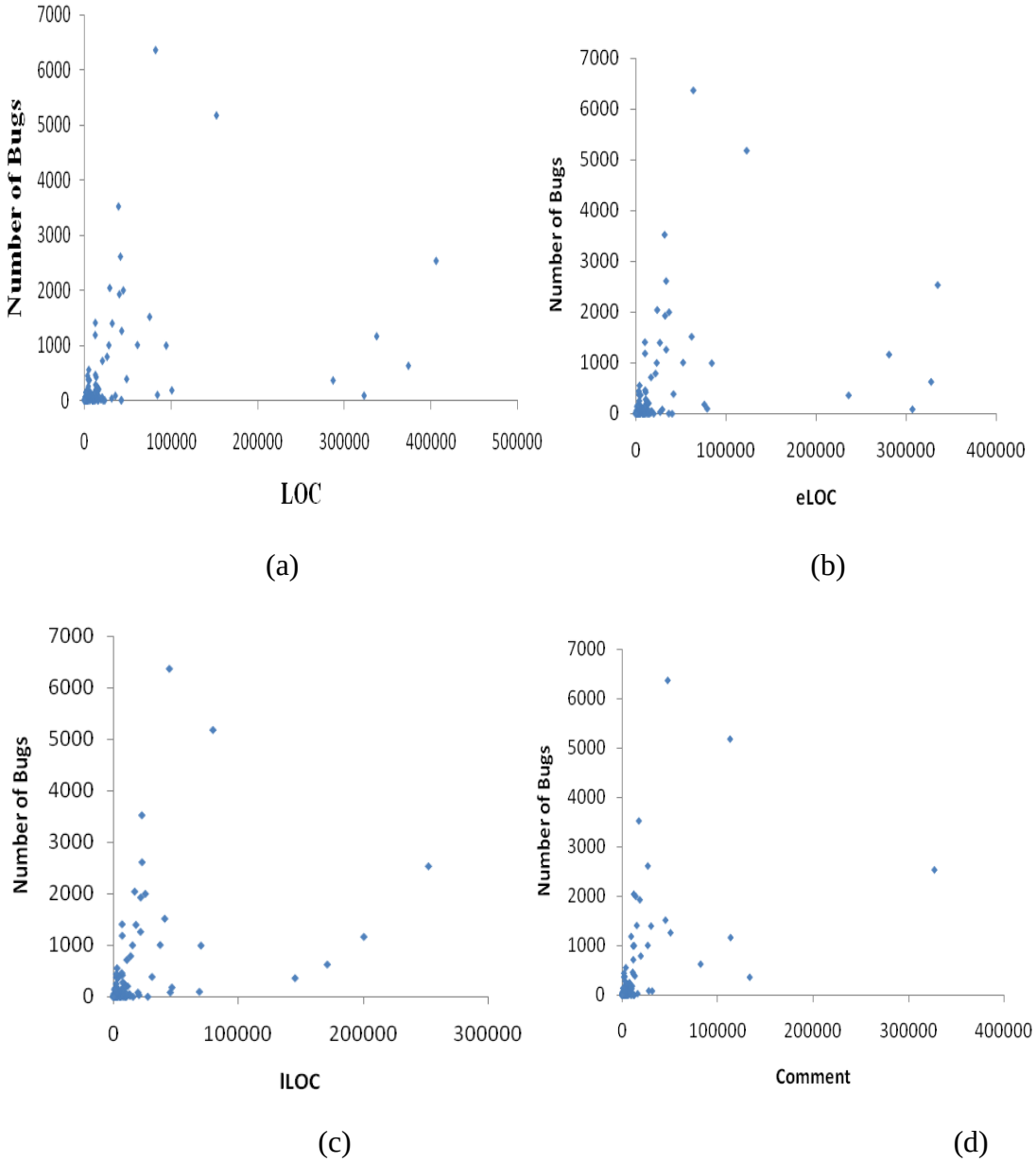
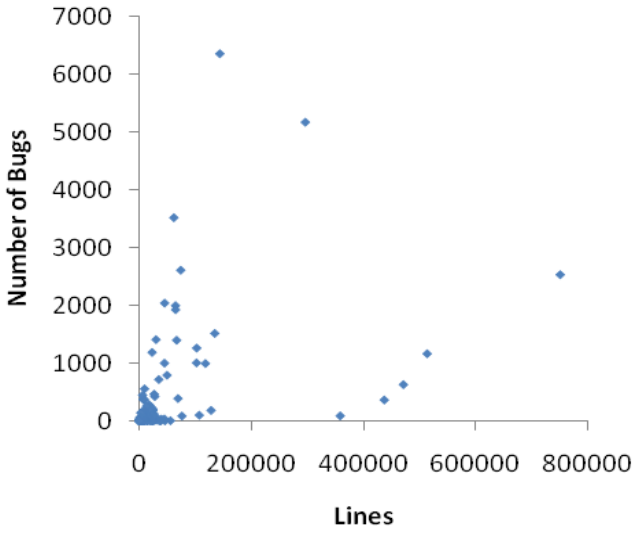


Figure 8.7: Scatter plot showing relationship between (a) number of bugs and LOC, (b) number of bugs and eLOC, (c) number of bugs and lLOC, (d) number of bugs and Comment, and (e) number of bugs and Lines in modules



(e)

Figure 8.7 Continued

Table 8.5 shows the Pearson Correlation coefficient values for LOC, eLOC, lLOC, Comment, and Lines metrics with respect to number of bugs in modules. The r value between LOC and number of bugs is 0.3929. The value of r is very low and does not indicate any relationship between LOC and number of bugs. The r value between eLOC and number of bugs is 0.3717. This value is not significant enough to indicate a linear relationship between eLOC and number of bugs in modules. The r value between lLOC and number of bugs is 0.4339, and it does not indicate a linear relationship between the two variables. Similarly, the value of r, 0.5012, between Comment and number of bugs in modules is not high enough to indicate a linear relationship between them. The value of r between Lines and number of bugs in modules is 0.4584. This value of r does not indicate any significant relationship between Lines and the number of bugs in module.

Table 8.5: Pearson Correlation coefficient values for LOC, eLOC, lLOC, Comment, and Lines metrics with respect to number of bugs in module

Metric	Pearson Correlation Coefficient (r)
LOC	0.3929
eLOC	0.3717
lLOC	0.4339
Comment	0.5012
Lines	0.4584

Based on the analysis, we conclude that Hypothesis #5 is rejected, that is, LOC, eLOC, lLOC, Comment, and Lines metrics did not have linear relationship with the number of bugs in modules.

8.6 Hypothesis #6: The LOC, eLOC, lLOC, Comment, and Lines Metrics Have a Linear Relationship with the Number of Bugs in Files

We used the RSM tool to collect LOC, eLOC, lLOC, Comment, and Lines metrics for each file in the Eclipse Project to see whether a relationship exists between them. We drew a scatter plot between LOC and number of bugs in files. The scatter plot is shown in Figure 8.8 (a). The plot does not show any identifiable pattern. Therefore, it indicates no specific relationship between the LOC and number of bugs in files. Figure 8.8 (b) shows the scatter plot drawn between eLOC and the number of bugs in the file. Figure 8.8(b) does not indicate a linear relationship between eLOC and the number of bugs in the file. Figure 8.8(c) shows the scatter plot drawn between lLOC and the number of bugs and it does not indicate a linear relationship between lLOC and number of bugs in the file. The scatter plot between Comment and the number of bugs is shown in Figure 8.8(d). Figure 8.8(d) lacks any identifiable pattern and does not indicate any relationship between Comment and number of bugs in the file. Similarly, Figure 8.8(e) shows the plot between Lines and number of bugs in the

file. This plot does not indicate any relationship between Lines and number of bugs in the file.

We calculated Pearson Correlation coefficient to analyze the relationship between the metrics and number of bugs in the file. Table 8.6 shows the result of our calculation for all the metrics. The r values for LOC, eLOC, lLOC, Comment, and Lines with respect to the number of bugs in the file are nearly equal to zero. These values do not indicate any linear relationship between the metrics and the number of bugs in the files.

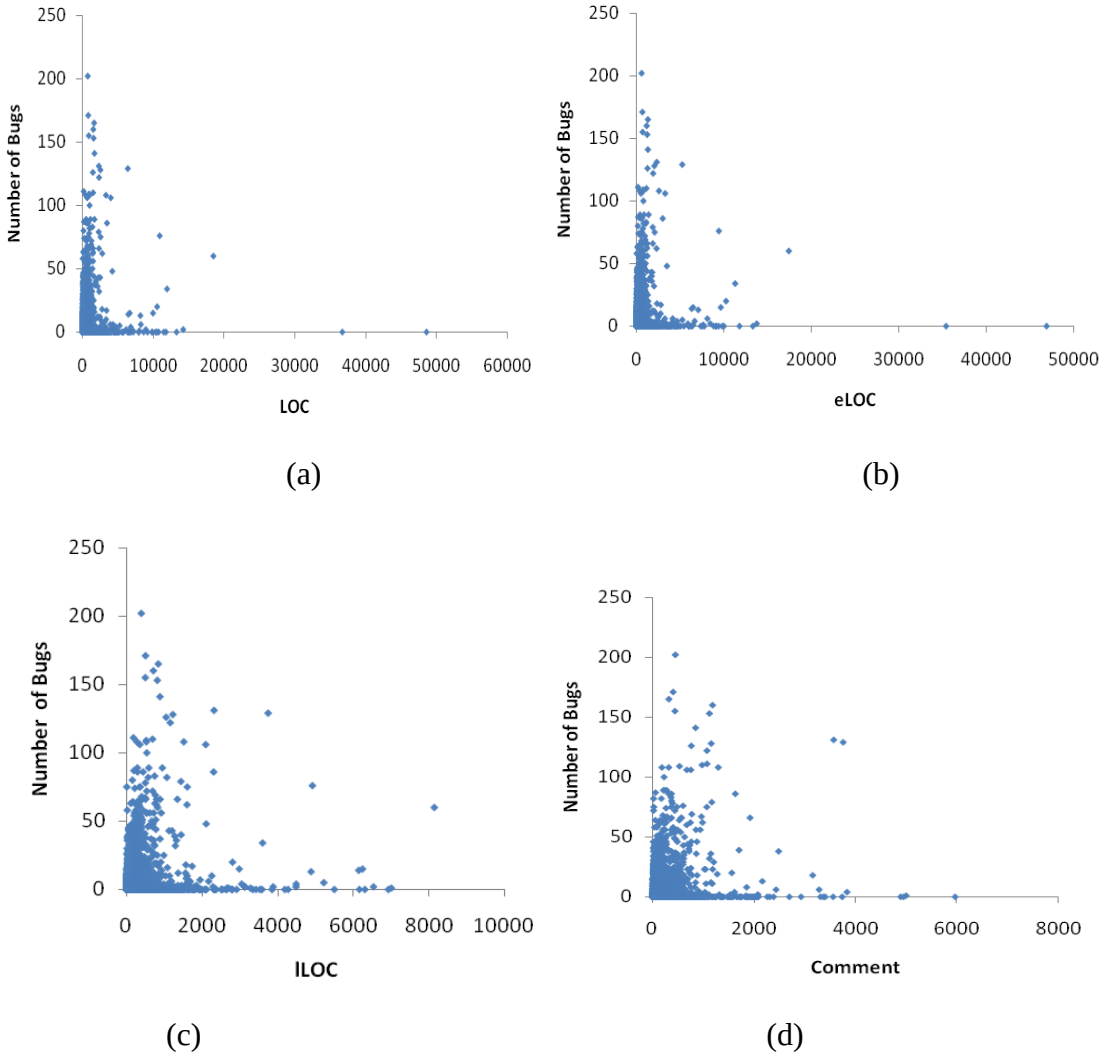
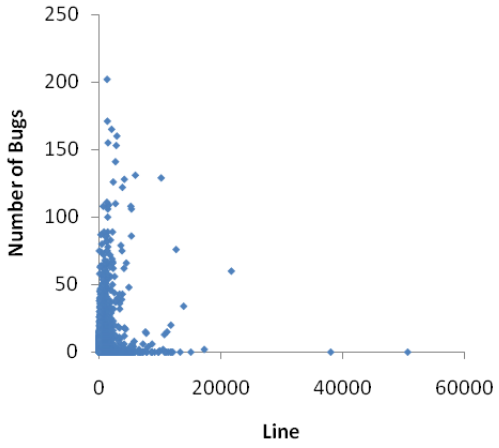


Figure 8.8: Scatter plot showing relationship between (a) number of bugs and LOC, (b) number of bugs and eLOC, (c) number of bugs and lLOC, (d) number of bugs and Comment, and (e) number of bugs and Lines in files



(e)

Figure 8.8 Continued

Table 8.6: Pearson Correlation coefficient values for LOC, eLOC, lLOC, Comment, and Lines metrics with respect to number of bugs in file

Metric	Pearson Correlation Coefficient (r)
LOC	0.1674
eLOC	0.1534
lLOC	0.2426
Comment	0.2666
Lines	0.2228

Based on our analysis, we conclude that Hypothesis #6 is rejected; that is, there was no linear relationship between the LOC, eLOC, lLOC, Comment, and number of bugs at the file level.

8.7 Hypothesis #7: Both Error Fixes and Non-error Changes Have The Power-Law Distribution at the Module Level

We classified the bugs into two groups: error fixes and non-error changes. We calculated the number of bugs in each category for 192 modules. It took us around eight months to classify the bugs in 192 modules. To classify all the bugs, it would have taken us a lot longer. Since we had classified the bugs in around 85% of the modules and one third of the total bugs, we believe that the test result will provide

some hints of the actual process. There were 9237 non-error changes and 10,182 error fixes in modules; that is, 47.57% of the bugs were non-error changes and 52.43% of the bugs were error fixes.

8.7.1 Power-Law Distribution of Error Fixes in Modules

We have listed the data for error fixes in Table 8.7. We sorted the modules in descending order with respect to the number of error fixes.

Table 8.7: Distribution of error fixes in modules

% of Module	Number of Error Fixes	% of Error Fixes
5.2083	6677	65.5765
10.4167	8073	79.287
10.9375	8173	80.2691
15.1042	8841	86.8297
20.3125	9388	92.2019
25	9608	94.3626
30.2083	9789	96.1402
35.4167	9917	97.3974
40.1042	10011	98.3206
45.3125	10082	99.0179
50	10123	99.4205
55.2083	10154	99.725
60.4167	10169	99.8723
65.1042	10178	99.9607
67.1875	10182	100

8.7.1.1 Power-Law Distribution of Error Fixes in Modules Using an Alberg Diagram

We plotted the cumulative percentage of error fixes on the y-axis of the Alberg Diagram with respect to the cumulative percentage of modules on the x-axis. Figure 8.9 shows the Alberg Diagram for the distribution of data given in Table 8.7.

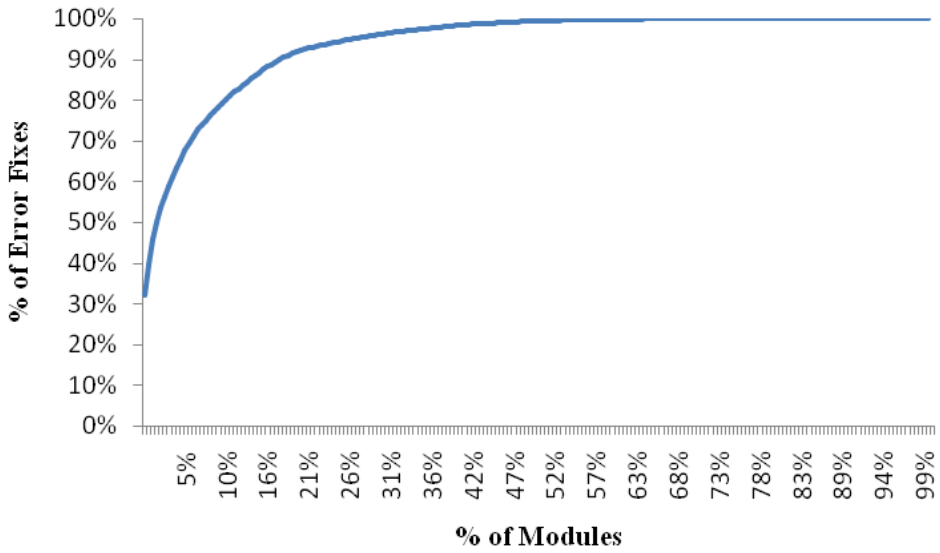


Figure 8.9: Alberg Diagram showing distribution of error fixes in modules

Figure 8.9 shows that roughly 5% of the modules were responsible for approximately 65% of the error fixes, 10% of the modules were responsible for roughly 79% of the error fixes, and 20% of the modules were responsible for about 91% of the error fixes. Roughly 67% of the modules contained 100% of the error fixes; that is, roughly a third of the modules did not report any error fixes. Figure 8.9 illustrates that a large majority of error fixes were concentrated in a small number of modules.

8.7.1.2 Power-Law Distribution of Error Fixes in Modules Using a Statistical Tool

To test the Power-Law distribution for error fixes in modules, we used the same functions described in Section 8.1.2. The lower cutoff value xmin for the data in Table 8.7 is 59. For the numbers of error fixes given in Table 8.7, the log plot is as given in Figure 8.10. Figure 8.10 shows that most of the data fell on the line given by the Power-Law distribution.

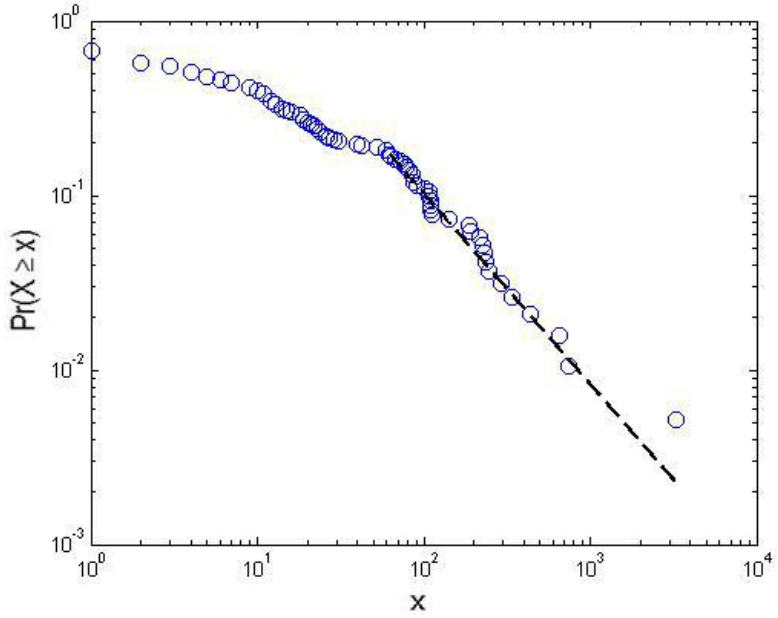


Figure 8.10: Log plot of number of error fixes in modules and Power-Law distribution

The p-value for the number of bugs in the modules is 0.556. This p-value was computed over 1000 repetitions. Since our p-value is greater than the threshold value of 0.05, we conclude that the number of error fixes at the module level followed the Power-Law distribution.

8.7.2 Power-Law Distribution of Non-error Changes in Modules

We have listed the data for non-error changes at the module level in Table 8.8. The table lists the number and percentage of modules and the corresponding percentage of non-error changes.

Table 8.8: Distribution of non-error changes in modules

% of Module	Number of Non-Error Changes	% of Non-Error Changes
5.2083	6145	66.5259
10.4167	7174	77.6659
15.1042	7763	84.0424
12.5	7451	80.6647
20.3125	8244	89.2498
25	8556	92.6275
30.2083	8774	94.9876
35.4167	8937	96.7522
40.1042	9039	97.8564
45.3125	9123	98.7658
50	9179	99.3721
55.2083	9218	99.7943
60.4167	9231	99.935
63.5417	9237	100

8.7.2.1 Power-Law Distribution of Non-error Changes in Modules Using an Alberg Diagram

We plotted the cumulative percentage of non-error changes on the y-axis of the Alberg Diagram with respect to the percentage of modules on the x-axis. Figure 8.11 shows the Alberg Diagram for the distribution of non-error changes given in Table 8.8.

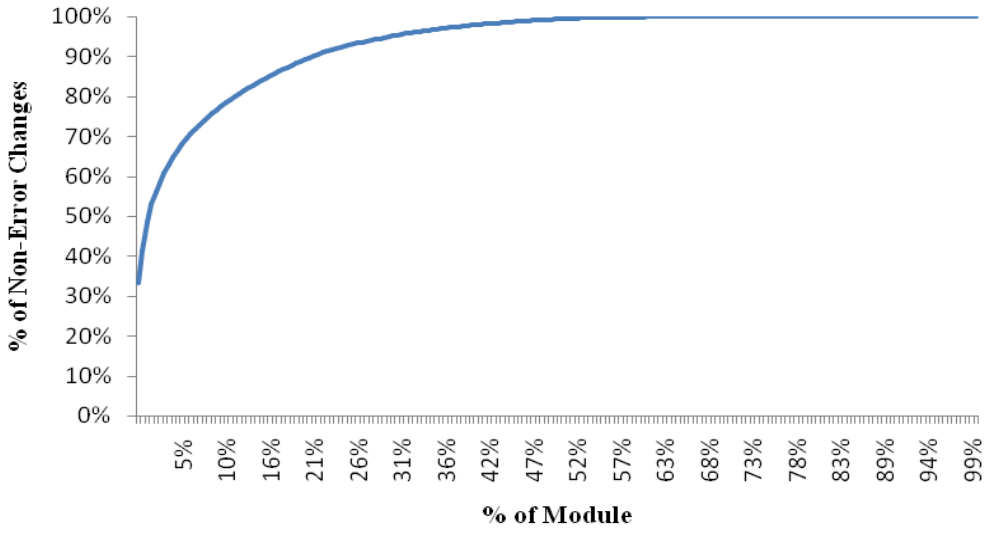


Figure 8.11: Alberg Diagram showing distribution of non-error changes in modules

Figure 8.11 indicates that roughly 5% of the modules were responsible for 66% of the non-error changes and 10% of modules contained roughly 80% of non-error changes. Similarly, 20% of the modules were responsible for more than 90% of the non-error changes. Approximately 60% of modules contained 100% of the non-error changes; that is, nearly two fifths of the modules did not report any non-error changes.

8.7.2.2 Power-Law Distribution of Non-error Changes in Modules Using a Statistical Tool

To test the Power-Law distribution for non-error changes in modules, we used the same functions described in Subsection 8.1.2. The lower cutoff value xmin for the data in Table 8.8 is 49. For the numbers of non-error changes given in Table 8.8, the plot is as given in Figure 8.12. Figure 8.12 shows that most of the data fell on the line given by the Power-Law distribution.

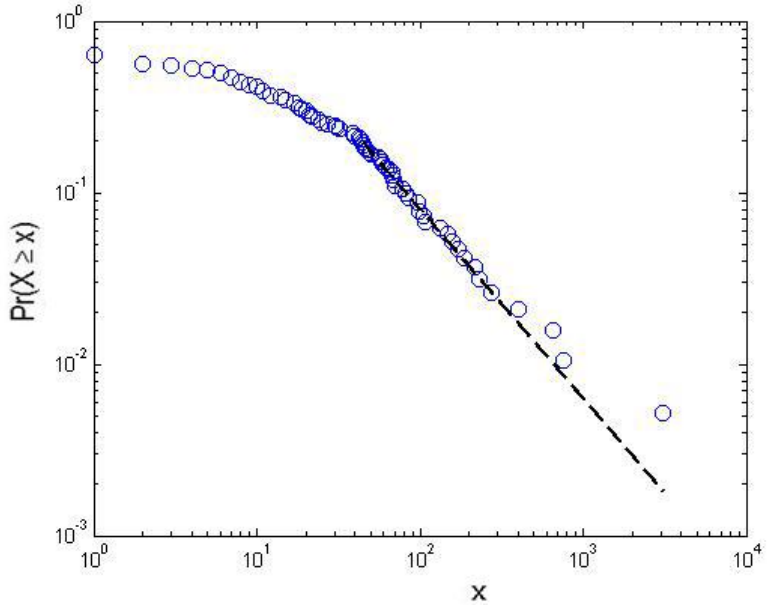


Figure 8.12: Log plot of number of non-error changes in modules and Power-Law distribution

The p-value for the number of non-error changes in modules is 0.99. This p-value was computed over 1000 repetitions. Since the p-value is greater than the threshold value of 0.05, we conclude that the number of non-error changes at the module level follow the Power-Law distribution.

8.8 Hypothesis #8: Both Error Fixes and Non-error Changes Have The Power-Law Distribution at the File Level

We classified the bugs into two groups: error fixes and non-error changes. We calculated the number of bugs in each category for 4837 files. There were 9237 non-error changes and 10,182 error fixes in files.

8.8.1 Power-Law Distribution of Error Fixes in Files

We have summarized the data for error fixes in Table 8.9. We ordered the files in descending order with respect to the number of error fixes.

Table 8.9: Distribution of error fixes in files

% of File	Number of Error Fixes	% of Error Fixes
5.0031	4735	46.5036
10.0062	6236	61.2453
15.0093	7201	70.7228
20.0124	7874	77.3325
22.8241	8146	80.0039
25.0155	8358	82.086
30.0186	8805	86.4761
35.001	9046	88.8431
40.0041	9288	91.2198
45.0072	9530	93.5965
50.0103	9772	95.9733
55.0134	10014	98.35
58.4867	10182	100

8.8.1.1 Power-Law Distribution of Error Fixes in Files Using an Alberg Diagram

We plotted the cumulative percentage of error fixes on the y-axis of the Alberg Diagram with respect to the cumulative percentage of files on the x-axis. Figure 8.13 shows the Alberg Diagram for the distribution of data given in Table 8.9.

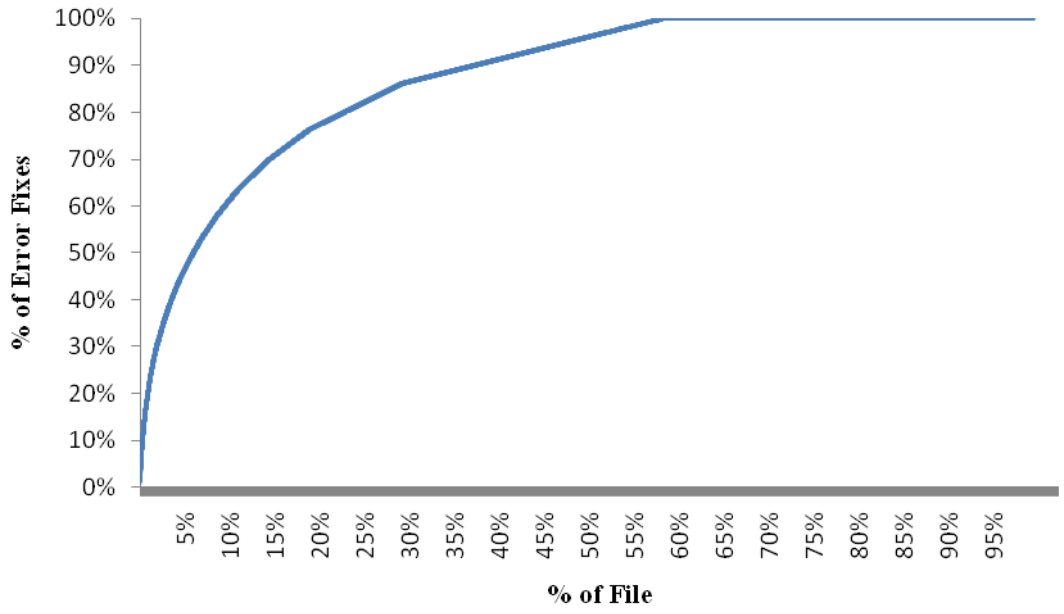


Figure 8.13: Alberg Diagram showing distribution of error fixes in files

Figure 8.13 shows that roughly 5% of the files were responsible for approximately 45% of the error fixes, 10% of the files were responsible for approximately 60% of the error fixes, and 20% of the files were responsible for about 75% of the error fixes. It also indicates that roughly 58% of files contained 100% of the error fixes; that is, roughly a two fifths of the files did not report any error fixes. Figure 8.13 illustrates that a large majority of error fixes were concentrated in a small number of files.

8.8.1.2 Power-Law Distribution of Error Fixes in Files Using a Statistical Tool

To test the Power-Law distribution for error fixes in files, we used the functions described in Subsection 8.1.2. The lower cutoff value xmin for the data in Table 8.9 is five. For the number of error fixes given in Table 8.9, the log plot is as given in Figure 8.14. Figure 8.14 shows that most of the data fell on the line given by the Power-Law distribution.

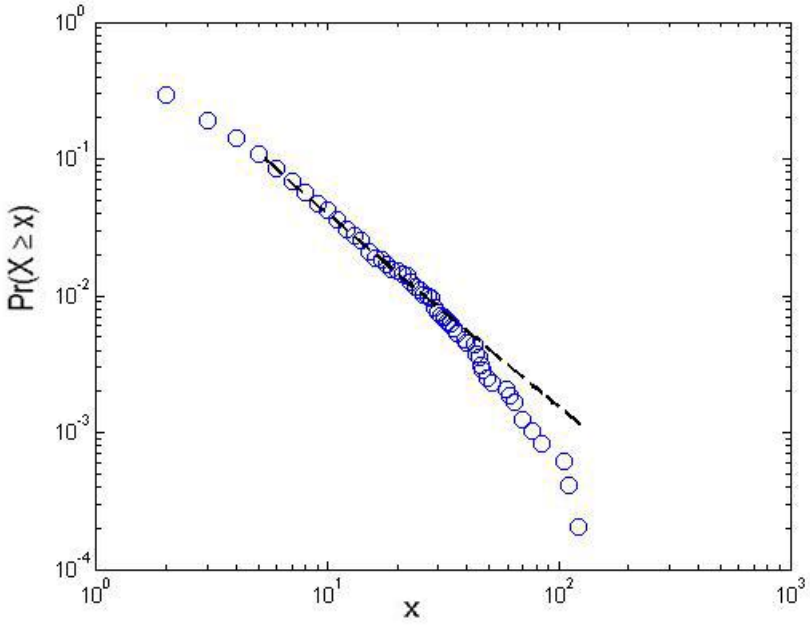


Figure 8.14: Log plot of number of error fixes in files and Power-Law distribution

The p-value for the number of bugs in file is 0.114. This p-value was computed over 1000 repetitions. Since the p-value is greater than the threshold value of 0.05, we conclude that the number of error fixes at file level followed the Power-Law distribution.

8.8.2 Power-Law Distribution of Non-error Changes in Files

We have summarized the data for non-error changes at the file level in Table 8.10C. The table lists the percentage of file and the corresponding percentage of non-error changes.

Table 8.10: Distribution of non-error changes in files

% of File	Number of Non-Error Changes	% of Non-Error Changes
5.0031	3331	36.0615
10.0062	4541	49.161
15.0093	5394	58.3956
20.0124	6039	65.3784
25.0155	6523	70.6182
30.0186	7007	75.858
35.001	7304	79.0733
36.779	7390	80.0043
40.0041	7546	81.6932
45.0072	7788	84.3131
50.0103	8030	86.933
55.0134	8272	89.5529
60.0165	8514	92.1728
65.0196	8756	94.7927
70.0021	8997	97.4018
74.9638	9237	100

8.8.2.1 Power-Law Distribution of Non-error Changes in Files Using an Alberg Diagram

We plotted the cumulative percentage of non-error changes on the y-axis of the Alberg Diagram with respect to the cumulative percentage of files on the x-axis. Figure 8.15 shows the Alberg Diagram for the distribution of non-error changes in files given in Table 8.10.

Figure 8.15 indicates that roughly 5% of the files were responsible for 35% of the non-error changes and 10% of files contained roughly 50% of non-error changes. Similarly, 20% of the files were responsible for more than 65% of the non-error

changes. Approximately 75% of files contained 100% of the non-error changes, that is, nearly one fourth of the files did not report any non-error changes.

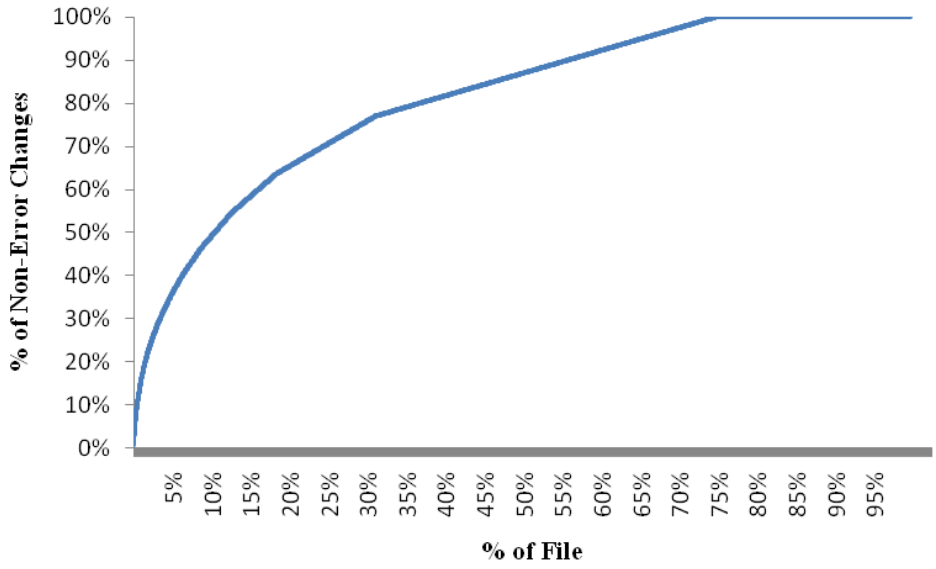


Figure 8.15: Alberg Diagram showing distribution of non-error changes in files

8.8.2.2 Power-Law Distribution of Non-error Changes in Files Using a Statistical Tool

The lower cutoff value xmin for the data in Table 8.10 is four. For the numbers of non-error changes given in Table 8.10, the plot is as given in Figure 8.16. Figure 8.16 shows that most of the data fell on the line given by the Power-Law distribution.

The p-value for the number of non-error changes in the file is 0.451. Since the p-value is greater than the threshold value of 0.05, we conclude that the number of non-error changes at file level followed the Power-Law distribution.

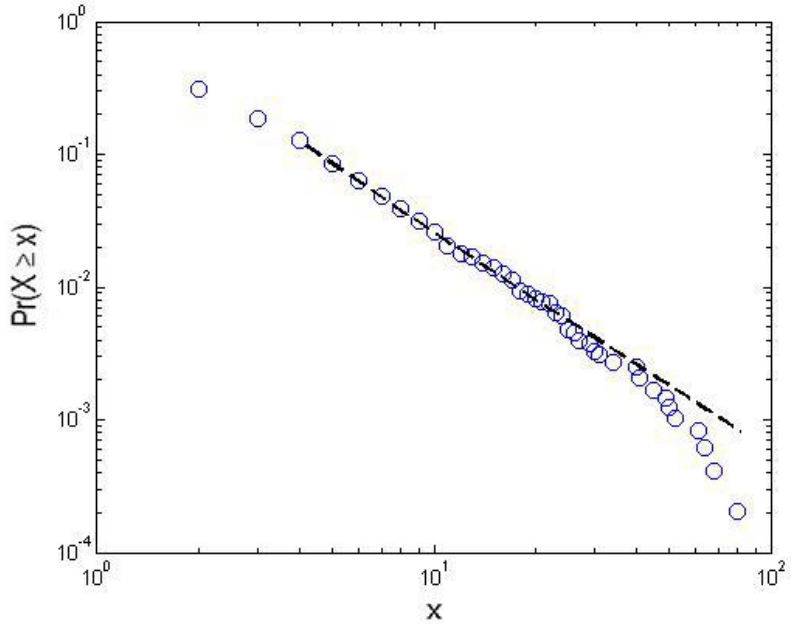


Figure 8.16: Log plot of number of non-error changes in files and Power-Law distribution

8.9 Hypothesis #9: Bugs Have The Power-Law Distribution in Each Classification Category at the Module Level

We classified 19,419 bugs from 192 modules into 19 different categories. We used the Matlab functions described in Section 8.1.2 to test the Power-Law distribution in each classification category at the module level.

We have listed the p-value and x-min for the number of bugs in each classification category at the module level in Table 8.11. The tool calculated the p-values over 1000 repetitions.

Table 8.11 shows that the p-values and x-min for the Internationalization and Third Party categories were not applicable. The Internationalization and Third Party bugs affected only three and six modules respectively. The threshold value of input data for the Power-Law distribution function is 50. The p-values for the Adaptive was below 0.05. This means that it did not follow the Power-Law distribution at the module level. The bugs in the remaining categories followed the Power-Law

distribution as shown by the p-values in Table 8.11. Table 8.11 also indicates that the classified bugs did not affect all modules. For example, out of 192 modules, the Adaptive bugs affected only 50 modules, the Cleanup bugs affected only 43 modules and so on. Sample size less than 50 may lead to finite-size bias in the calculation of p-values for the Power-Law distribution resulting in unreliable estimates [Aaron et al. 2009].

Table 8.11: Distribution of bugs in each classification category at module level

Category	**P value**	**X min**	**Number of Module Affected**	**Total Bugs**
Adaptive	0.032	5	50	474
Cleanup	0.738	2	43	553
Coding	0.312	13	59	667
Concurrency	0.925	5	32	542
Data	0.27	2	44	583
Doc	0.585	8	32	239
Enhancement	0.146	12	93	5459
Functionality	0.052	14	95	3236
Generic	0.176	5	86	1313
Gui	0.116	1	28	443
Handling	0.492	8	69	1285
I/O	0.167	8	3	255
Internationalization	n/a	n/a	31	10
Memory	0.254	6	28	254
Messaging	0.525	4	46	520
Not bug	0.788	20	81	1553
Refactoring	0.882	7	66	1751
Regression	0.409	2	40	272
Third Party	n/a	n/a	6	10

Hence, from our discussion we conclude that not all, but most classification categories at the module level followed the Power-Law distribution.

8.10 Hypothesis #10: Bugs Have the Power-Law Distribution in Each Classification Category at the File Level

We classified 19,419 bugs from 4837 files into 19 different categories. We used the Matlab functions described in Subsection 8.1.2 to test the Power-Law distribution in each classification category at file level.

We computed the p-value and xmin for the number of bugs in each classification category at the file level. The tool calculated the p-values over 1000 repetitions.

Table 8.12 shows that the p-values and xmin for the Internationalization and Third Party categories were not applicable. The Internationalization and Third Party bugs affected only 10 and 9 files respectively. The threshold value of input data for the Power-Law distribution function is 50. The p-values for the Adaptive, Cleanup, Coding, Concurrency, Enhancement, Gui, I/O, Memory, Not Bug, and Regression categories were below the threshold value of 0.05. This means that they did not follow the Power-Law distribution at the file level. The bugs in the remaining categories followed the Power-Law distribution as shown by the p-values present in Table 8.12. Table 8.12 also indicates that the classified bugs affected not all files. For example, out of 4837 files, the Adaptive bugs affected only 343 files, the Cleanup bugs affected only 404 files and so on. Sample size less than 50 may lead to finite-size bias in the calculation of p-values for the Power-Law distribution resulting in unreliable estimates [Aaron et al. 2009].

Table 8.12: Distribution of bugs in each classification categories at file level

Category	P value	X min	Number of files Affected	Total Bugs
Adaptive	0.019	1	343	474
Cleanup	0.044	1	404	553
Coding	0	1	431	667
Concurrency	0.028	1	283	542
Data	0.413	2	352	583
Doc	0.489	1	174	239
Enhancement	0	1	2481	5459
Functionality	0.608	4	1382	3236
Generic	0.634	3	815	1313
Gui	0.029	1	263	443
Handling	0.601	3	692	1285
I/O	0	1	164	255
Internationalization	n/a	n/a	10	10
Memory	0.023	1	173	254
Messaging	0.324	1	322	520
Not bug	0	1	1058	1553
Refactoring	0.326	3	952	1751
Regression	0.007	1	200	272
Third Party	n/a	n/a	9	10

Hence, from our discussion we conclude that most (two thirds) of the classification categories at the file level did not follow the Power-Law distribution.

CHAPTER 9

CONCLUSIONS

We have developed a tool to collect the Eclipse Project bug data from its public website. The Eclipse Project has 220 modules, 33,904 files (classes), and 3780201 LOC (total), and it has more than a decade of evolution history. We analyzed the data by testing ten hypotheses. Our research showed that total bugs (error fixes and non-error changes), error changes, and non-error changes over the decade-long evolution followed the Power-Law distribution at the module and file levels. When we classified the error changes into more detailed categories, we found that all but one bug category had the Power-Law distribution at the module level. At the file level, around two-thirds of the bug categories did not follow a Power-Law distribution. Our research showed that there was no correlation between LOC, eLOC, lLOC, Comment, and Lines and the number of bugs both at the module and file levels. However, a small percentage of modules having a large percentage of bugs also contained most of the code size (LOC). Nevertheless, a small percentage of files having a large percentage of bugs did not contain most of the code size.

The research studies conducted by Zimmermann and the colleagues [Zimmermann et al. 2007], Shatnawi and Li [Shatnawi and Li 2007], Shatnawi and Li [Shatnawi and Li 2008], and Jiang and colleagues [Jiang et al. 2010] used the data from both the Eclipse change log and the Bugzilla database. However, these studies did not study the modules as given in the online Eclipse CVS repository and do not

use an automated tool to collect the data as was done in our study. They did not study the bug distribution. Nevertheless, the study conducted by Ohlsson and Alberg [Ohlsson and Alberg 1996] showed that 60% of the bugs were located in 20% of the modules. Our results confirmed the findings of Fenton and Ohlsson [Fenton and Ohlsson 2000] and Ostrand and colleagues [Ostrand et al. 2004], and contradicted the findings of Lipow [Lipow 1982] and Ostrand and colleagues [Ostrand et al. 2004].

APPENDICES

APPENDIX A

LIST OF BUG CLASSIFICATION CATEGORIES

Table A.1: List of bug classification categories

S. N	Bug Category	Description
1.	Adaptive Maintenance	A type of maintenance performed to change software so that it will work in an altered environment, such as when an operating system, hardware platform, compiler, software library or database structure changes, compatibility changes etc.
2.	Coding	Bugs in the decision logic, branching, sequencing, computational logic, typo in programming Examples: General wrong concept employed in coding Syntactically incorrect program Navigation not coded correctly in source code The control flow is incorrectly implemented Incorrect processing and evaluation of expression and equations Incorrect parsing Typographical mistakes—spelling mistakes Missing, inadequate, irrelevant, or ambiguous functionality in source code Infinite loops, infinite recursion Dead code
3.	Concurrency	Bugs that happen in multithreading or multi-process environment, including data race , deadlock , and synchronizations problem

Table A.1 Continued

4.	Data	Bugs in definition, structure, mapping, access, scope, use, or initialization Examples: Incorrect object type or dimension Incorrect initial default values Incorrect duplication or failure to create a duplicate object Incorrect access to object Use of incorrect variable names
5.	Documentation	Bugs in the specifications and documents Bugs in program help
6.	Enhancement	Actions performed to add a new capability or improve the existing capability to software. Bugs that cause failure to meet the performance requirements of the product. Examples: Functions correctly but runs/respond slowly Takes longer time to perform a task Addition of a new feature
7.	Generic	Unknown bugs Bugs related to security Bugs related to scalability Bugs with missing bug description
8.	GUI	Bugs related to graphical user interface and CSS bugs Examples: Incorrect alignment of components Incorrect size and shape of interface Incorrect layout of reports Incorrect coloring
9.	I/O, Serialization	Bugs related to I/O handling, import and export of file, and serialization of objects

Table A.1 Continued

10.	Functionality	Function is incorrect, ambiguous, inconsistent, missing, incomplete, that need not be present, or does not behave as expected. Bugs related to core functionality implementation. Functions that do not meet the specification requirement or the standards with the software version, coding conventions, representation Examples: Unnecessary feature Duplicate feature Misplaced feature Missing Validation
11.	Internationalization: Localization with Resource Bundles	Bug due to adaption to various languages and regions without engineering changes. Bug due to adaption of internationalized software for a specific region or language
12.	Memory	Bugs related to memory Examples: Buffer overflow Memory access violations Memory leak Dangling Pointer NULL pointer Deference Double Free
13.	Message	Defects in message and logs, inadequate, incorrect, misleading, or missing bug messages in source code or bug logs
14.	Refactoring	Bugs due to change of source code without modifying the functional behavior to improve some nonfunctional attributes, like readability, reduced complexity, or maintainability, of the software
15.	Third Party	Bugs due to third party software

Table A.1 Continued

16.	Handling (Event/Exception)	Missing event handling or improper event handling, do not have proper exception handling, anomalies caused by improper exception handling Examples: Null pointer exception Class cast exception Missing key press or mouse movement handling Incorrect mouse click, hover, and double click handling Incorrect action selection Timer expiration
17.	Cleanup	Removing of unused and unnecessary code (may be due to copy which is not used in this feature) Examples: Removal of deprecated function Removal of unnecessary comments
18.	Regression/rollback	Bugs due to change in some function or code, change rollback, and changes not made properly Examples: Incorrect merging of codes You commit a fix, but it did not work so have to roll back
19.	Not bugs	Example, test cases, simple updates Example: File name update, copy right update, test case addition, sample examples

APPENDIX B

INPUT AND OUTPUT SAMPLE FOR RSM

```
                    Report Banner - Edit rsm.cfg File

          Resource Standard Metrics (TM) for C, C++, C# and Java
                  Version 7.75 - mSquaredTechnologies.com

  License Type: Windows Single User License, 1272 License Days
  Licensed To : ACADEMIC - Wei Li & Team
  License No. : WS2948                          License Date: Aug 07,
2010
  Build Date  : Sep  2 2009                          Run Date: Jan 22,
2011
  (C)1996-2009 M Squared Technologies LLC


_____________________________________________________________________________
_

  License File: C:\Program Files\MSquared\M2 RSM\rsm.lic
  Config. File: C:\Program Files\MSquared\M2 RSM\rsm.cfg
  Command Line: -OC:\Users\pitt\M2 RSM Wizard\output\output.txt -
FC:\Users
                \pitt\M2 RSM Wizard\input\rsm_file_list.lst

                                  File:
   E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
                      ernal\intro\universal\BaseData.java
     LOC 16       eLOC 12       lLOC 8        Comment 10       Lines
                                   33

                                  File:
   E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
          ernal\intro\universal\contentdetect\ContentDetectHelper.java
     LOC 139      eLOC 112      lLOC 78       Comment 24       Lines
                                   176
                                  File:
   E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
            ernal\intro\universal\contentdetect\ContentDetector.java
     LOC 58       eLOC 46       lLOC 32       Comment 21       Lines
```

Figure B.1: Output report generated by RSM for the module *org.eclipse.ui.intro.universal*

```
                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
          ernal\intro\universal\CustomizationContentsArea.java
  LOC 1069     eLOC 904      lLOC 669      Comment 83       Lines
                                1220

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
             ernal\intro\universal\CustomizationDialog.java
  LOC 47       eLOC 38       lLOC 29       Comment 11       Lines
                                67

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
              ernal\intro\universal\CustomizeAction.java
  LOC 20       eLOC 16       lLOC 12       Comment 10       Lines
                                38

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
               ernal\intro\universal\ExtensionData.java
  LOC 89       eLOC 75       lLOC 40       Comment 13       Lines
                                125

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
            ernal\intro\universal\ExtensionDataTransfer.java
  LOC 108      eLOC 90       lLOC 68       Comment 12       Lines
                                135

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
                  ernal\intro\universal\GroupData.java
  LOC 175      eLOC 138      lLOC 95       Comment 42       Lines
                                222

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
                  ernal\intro\universal\IntroData.java
  LOC 166      eLOC 143      lLOC 107      Comment 34       Lines
                                202

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
          ernal\intro\universal\IUniversalIntroConstants.java
  LOC 67       eLOC 66       lLOC 65       Comment 98       Lines
                                119
                               File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
                  ernal\intro\universal\Messages.java
  LOC 86       eLOC 84       lLOC 82       Comment 13       Lines
                                102
```

Figure B.1 Continued

```
                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
                    ernal\intro\universal\PageData.java
  LOC 168      eLOC 134      lLOC 93       Comment 27       Lines
                                204

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
                     ernal\intro\universal\SeparatorData.java
  LOC 15       eLOC 11       lLOC 7        Comment 12       Lines
                                30

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
            ernal\intro\universal\UniversalIntroConfigurer.java
  LOC 586      eLOC 490      lLOC 273      Comment 138      Lines
                                686

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
              ernal\intro\universal\UniversalIntroPlugin.java
  LOC 71       eLOC 55       lLOC 32       Comment 56       Lines
                                148

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
                  ernal\intro\universal\util\BundleUtil.java
  LOC 137      eLOC 114      lLOC 61       Comment 101      Lines
                                257

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
                   ernal\intro\universal\util\ImageUtil.java
  LOC 109      eLOC 91       lLOC 58       Comment 88       Lines
                                205

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
                    ernal\intro\universal\util\Log.java
  LOC 69       eLOC 59       lLOC 35       Comment 63       Lines
                                142

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
             ernal\intro\universal\util\PreferenceArbiter.java
  LOC 31       eLOC 22       lLOC 13       Comment 15       Lines
                                51

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
                  ernal\intro\universal\util\StringUtil.java
  LOC 29       eLOC 24       lLOC 14       Comment 10       Lines
                                49
```

Figure B.1 Continued

```
                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
      ernal\intro\universal\WelcomeCustomizationPreferencePage.java
   LOC 50       eLOC 37       lLOC 22       Comment 10       Lines
                                 75

                                File:
E:\workspace\org.eclipse.ui.intro.universal\src\org\eclipse\ui\int
                    ro\universal\ExtensionFactory.java
   LOC 37       eLOC 32       lLOC 21       Comment 43       Lines
                                 85

------------------------------------------------------------------
                                ---

                          ~~ Project Analysis For 23 Files ~~

   LOC 3342     eLOC 2793     lLOC 1914     Comment 934      Lines
                                4460
                  Average per File, metric/23 files
   LOC 145      eLOC 121      lLOC 83       Comment 40       Lines
                                193

------------------------------------------------------------------
                                ---

                                       ~~ File Summary ~~

    C Source Files *.c ....:          0  C/C++ Include Files *.h:
                               0
    C++ Source Files *.c* .:          0  C++ Include Files *.h* :
                               0
    C# Source Files *.cs ..:          0  Java Source File *.jav*:
                               23
    Other File Count ......:          0  Total File Count ......:
                               23


__________________________________________________________________
                                _

                           Report Banner - Edit rsm.cfg File
```

Figure B.1 Continued

```
/*******************************************************************************
    ********** Copyright (c) 2005, 2006 IBM Corporation and others.
   * All rights reserved. This program and the accompanying materials
   * are made available under the terms of the Eclipse Public License
                                v1.0
       * which accompanies this distribution, and is available at
              * http://www.eclipse.org/legal/epl-v10.html
                                *
                            * Contributors:
         *     IBM Corporation - initial API and implementation

*******************************************************************************
                                ********/
           package org.eclipse.ui.internal.intro.universal;

                      import java.io.PrintWriter;

                   public abstract class BaseData {

                          private GroupData parent;
                            protected String id;

                  protected void setParent(GroupData gd) {
                                this.parent = gd;
                                    }

                       public GroupData getParent() {
                                   return parent;
                                    }

        public abstract void write(PrintWriter writer, String indent);

                              public String getId() {
                                 return id;
                                  }
                                }
```

Figure B.2: Source code for the *BaseData.java*, input for RSM

REFERENCES

[Aaron et al. 2009] C. Aaron, S. R. Cosma, and M.E.J. Newman, "Power-Law Distribution in Empirical Data," Society of Industrial and Applied Mathematics, Vol. 51, Issue 4, Nov. 2009.

[Adamic 2002] L.A. Adamic, "Zipf, Power-Laws, And Pareto – A Ranking Tutorial," Technical report, Internet Ecologies Area, Xerox Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto, CA 94304, 2000.

[Alshayeb and Li 2003] M. Alshayeb and W. Li, "An Empirical Validation of Object-Oriented Metrics in Two Iterative Processes," IEEE Transaction on Software Engineering, Vol. 29, No. 11, pp. 1043-1049, Nov. 2003.

[Alshayeb and Li 2005] M. Alshayeb and W. Li, "An Empirical Study of System Design Instability Metric and Design Evolution in an Agile Software Process," Journal of Systems and Software, Vol. 74, No. 3, pp. 269-274, Feb. 2005.

[Alshayeb and Li 2006] M. Alshayeb and W. Li, "An Empirical Study Of Relationships Among The Extreme Programming Engineering Activities," Information and Software Technology, Vol. 48, No. 11, pp. 1068-1072, 2006.

[Andersson and Runeson 2007] C. Andersson and P. Runeson, "A Replicated Quantitative Analysis Of Fault Distribution Of Complex Software Systems," IEEE Trans. Software Engineering, Vol. 33, No. 5, pp. 273-286, May 2007.

[Basili and Perricone 1984] V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Comm. ACM, Vol. 27, No. 1, pp. 42-52, 1984.

[Beizer 1990] Boris Beizer, Software Testing Techniques, Second Ed., The Coriolis Group, 1990.

[Bell et al. 2006] R.M. Bell, T.J. Ostrand, and E.J. Weyuker, "Looking for Bugs in All the Right Place," Proc. ACM/International Symposium on Software Testing and Analysis (ISSTA2006), Portland Maine, pp. 61-71, Jul. 2006.

[Bird et al. 2009] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, "Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista," ICSE '09: Proceedings of the 31st International Conference on Software Engineering, pp. 518–528, 2009.

[Bluman 1995] A. G. Bluman, Elementary Statistics: A Step by Step Approach, Second ed., Wm. C. Brown Publishers, 1995.

[Bugzilla 2011] Bugzilla Official Website, http://www.bugzilla.org/, 2011.

[Bugzilla a bug's life cycle 2011] Bugzilla A Bug's Life Cycle, https://bugs.eclipse.org/bugs, 2011.

[Caglayan et al. 2009] B. Caglayan, A. Bener, and S. Koch, "Merits Of Using Repository Metrics In Defect Prediction For Open Source Projects," ICSE, Workshop on FLOSS, pp. 31-36, May 2009.

[Cataldo et al. 2009] M. Cataldo, A. Mockus, J.A. Roberts, and J.D. Herbsleb, "Software Dependencies, Work Dependencies, And Their Impact On Failures," IEEE Transactions on Software Engineering, Vol. 99, No. 6, pp. 864–878, 2009.

[Compton and Withrow 1990] B.T. Compton and C. Withrow, "Prediction and Control of Ada Software Defects," J. Systems Software, Vol. 12, pp. 199-207, 1990.

[Crawford et al. 1985] S. G. Crawford, A. A. McIntosh, and D. Pregibon, "An Analysis Of Static Metrics And Faults In C Software," J. Syst. Software, Vol. 5, pp. 37-48, 1985

[CVS and Eclipse 2010] CVS and Eclipse, http://www.grapeinfo.com/doc/win2000srv/filemanagement/cvs.html

[CVS Howto 2010] Eclipse CVS Repository Howto, http://wiki.eclipse.org/CVS_Howto, 2010

[CVSChangeLog 2011] CVSChangeLog, http://cvschangelog.sourceforge.net/, 2011.

[D'Ambros et al. 2009] M. D'Ambros, M. Lanza, and R. Robbes, "On The Relationship Between Change Coupling And Software Defects," Reverse Engineering, Working Conference on, Vol. 0, pp. 135–144, 2009.

[Dinkelacker et al. 2002] J. Dinkelacker, P.K. Garg, R. Miller, and D. Nelson, "Progressive Open Source," Proc. IEEE/ International Conference in Software Engineering, Florida, USA, pp: 177-184, May 2002.

[Eclipse 2010] Eclipse Official Website, http://www.eclipse.org/org/

[Eclipse Repository 2010] Eclipse Source Repository, http://dev.eclipse.org/viewcvs/index.cgi/, 2010

[Erlikh 2000] L. Erlikh, "Leveraging Legacy System Dollars For E-Business," IT Professional, Vol. 2, No. 3, pp. 17–23, 2000.

[Feller and Fitzgerald 2000] J. Feller, B. Fitzgerald, "A Framework Analysis Of The Open Source Software Development Paradigm," Proc. ACM/International Conference on Information Systems, Brisbane, Australia, pp. 58 - 69, 2000

[Fenton and Ohlsson 2000] N.E. Fenton and N. Ohlsson, "Quantitative Analysis Of Faults And Failures In A Complex Software System," IEEE TSE, Vol 26, No. 8, Aug. 2000.

[Gittens et al. 2005] M. Gittens, K. Yong, and D. Godwin, "The Vital Few Versus The Trivial Many: Examining The Pareto Principle For Software," Computer Software and Applications Conference, Vol. 1, pp. 179-185, 2005.

[Graves et al. 2000] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," IEEE Transaction on Software Engineering, Vol. 26, No. 7, pp.653-661, Jul. 2000.

[Halstead 1997] M. H. Halstead, "Elements of Software Science," New York: Elsevier, 1977.

[Hatton 1997] L. Hatton, "Reexamining the Fault Density Component Size Connection," IEEE Software, pp.89-97, Apr. 1997

[Hatton 2009] L. Hatton, "Power-Law Distributions of Component Size in General Software Systems," IEEE Transactions on Software Engineering, Vol. 35, Issue 4, pp. 566-572, 2009.

[IEEE Standard 2010] IEEE standard, "IEEE Standard Classification For Software Anomalies," IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), pp. C1 – 15, 2010

[Jiang et al. 2010] Z. M. Jiang, E. Shihab, W. M. Ibrahim, B. Adams, and A. E. Hassan, "Understanding The Impact Of Code And Process Metrics On Post-Release Defects: A Case Study On The Eclipse Project," Proc. ACM, Empirical Software Engineering and Measurement, Sep. 2010.

[Juran et al. 1979] J.M. Juran, F.M. Gryna, Jr., and F.M. Bingham, Quality Control Handbook, Third edition, McGraw Hill, New York, 1979.

[Kim et al. 2006] S. Kim, K. Pan, and E.J. Whitehead, "Memories of Bug Fixes," Proc. ACM/SIGSOFT, Portland, USA, pp. 35-45, Nov. 2006.

[Lerdorf 2002] R. Lerdort, "PHP Pocket Reference," O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2002.

[Li and Shatnawi 2007] W. Li and R. Shatnawi, "An Empirical Study of the Bad Smells and Class Error Probability in the Post-Release Object-Oriented System Evolution," Journal of Systems and Software, Vol. 80, No. 7, pp.1120-1128, 2007.

[Li et al. 2006] 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," ACM/Architectural Support for Programming Languages and Operating Systems, San Jose, California, pp. 25-33, 2006.

[Lincke et al. 2008] R. Lincke, J. Lundberg, and W. Lowe, "Comparing Software Metrics Tools," Proc. ACM/International Symposium on Software Testing and Analysis, Seattle, USA, pp. 131-142, Jul. 2008.

[Lind and Vairavan 1989] R. K. Lind and K. Vairavan, "An Experimental Investigation Of Software Metrics And Their Relationship To Software Development Effort," IEEE Trans. Software Eng., Vol. 15, pp. 649-651, May 1989.

[Lipow 1982] M. Lipow, "Number of Faults per Line of Code," IEEE Trans. Software Engineering, Vol. 8, No. 4, pp. 437-439, Jul. 1982.

[Lo et al. 2009] D. Lo, H. Cheng, J. Han, S. C. Khoo, and C. Sun, "Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach," Proc. ACM/International Conference On Knowledge Discovery And Data Mining, Jun. 2009.

[Moller and Paulish 1993] K.H. Moller and D.J. Paulish, "An Empirical Investigation of Software Fault Distribution," Proc., First Int'l SoftwareMetrics Symp. (Metrics '93), pp. 82-90, 1993

[Moser et al. 2008] R. Moser, W. Pedrycz, and G. Succi, "A Comparative Analysis Of the Efficiency Of Change Metrics And Static Code Attributes For Defect Prediction," in ICSE '08: Proceedings of the 30th international conference on Software engineering, pp. 181–190, 2008.

[MYSQL 2010] Mysql Official Website, http://www.mysql.com, 2010

[Nagappan et al. 2008] N. Nagappan, B. Murphy, and V. Basili, "The Influence Of Organizational Structure On Software Quality: An Empirical Case Study," in ICSE '08: Proceedings of the 30th international conference on Software engineering, pp. 521–530, 2008.

[O'Donoghue et al. 2002] D. O'Donoghue, A. Leddy, J. Power, and J. Waldron, "Bi-gram Analysis of Java Bytecode Sequences", In Principles and Practice of Programming in Java, Trinity College Dublin, Jun. 2002.

[Ohlsson and Alberg 1996] N. Ohlsson, H. Alberg, "Predicting Fault-Prone Software Modules in Telephone Switches," IEEE Trans. Software Engineering, Vol.22, No. 12, pp 886-894, Dec. 1996.

[Ostrand and Weyuker 2002] T.J. Ostrand and E.J. Weyuker, "The Distribution Of Faults In A Large Industrial Software System," Proc. ACM/SIGSOFT
[Ostrand et al. 2004] T.J. Ostrand, E.J. Weyuker, and R. M. Bell, "Where The Bugs Are," Proc. ACM/International Symposium of Software Testing and Analysis, Boston, USA, pp. 86-96, Jul. 2004.

[Owens et al. 1996] H.D. Owens, B.F. Womack, and M.J. Gonzalez, "Software Error Classification Using Purify," Proc. IEEE/ International Conference on Software Maintenance, Monterey CA, pp. 104-113, Nov. 1996.

[Pan et al. 2006] K. Pan, S. Kim, and E.J. Whitehead, "Bug Classification Using Program Slicing Metrics," IEEE/International Workshop on Source Code Analysis and Manipulation, pp. 31-42, 2006.

[Potanin 2002] A. Potanin, "The Fox-A Tool For Object Graph Analysis," Technical report, Victoria University of Wellington, 2002,

[Potanin et al. 2005] A. Potanin, J. Noble, M. R. Frean, and R. Biddle., "Scale-free Geometry In Object-oriented Programs," ACM Communication, Vol. 48, Issue 5, May 2005.

[Pressman 2005] R. S. Pressman, "Software Engineering: A Practitioner's Approach," fifth ed. McGraw Hill, 2005.

[Rahmani et al. 2010] C. Rahmani, A. Azadmanesh, and L. Najjar, "A Comparative Analysis of Open Source Software Reliability," Journal Of Software, VOL. 5, NO. 12, Dec. 2010.

[RSM 2011] Resource Standard Metrics,
http://msquaredtechnologies.com/m2rsm/index.htm, 2011

[RSM download 2011] M Technologies Resource Standard Metrics Download,
http://msquaredtechnologies.com/m2rsm/rsm_demo.php, 2011

[RSM metric definition 2011] Resource Standard Metrics Definition,
http://msquaredtechnologies.com/m2rsm/docs/rsm_metrics_narration.htm, 2011

[Scacchi 2002] W. Scacchi, "Understanding the Requirements for Developing Open Source Software Systems," IEEE Proc. Software, pp: 24-39, Feb. 2002.

[SDN 2010] ORACLE Sun Developer Network (SDN), http://bugs.sun.com/bugdatabase, 2010.

[Shatnawi and Li 2007] R. Shatnawi and W. Li, "An Empirical Study of the Bad Smells and Class Error Probability in the Post-release Object-oriented System Evolution," Journal of Systems and Software, Vol. 80, pp. 1120-1128, 2007.

[Shatnawi and Li 2008] R. Shatnawi and W. Li, "The Effectiveness of Software Metrics in Identifying Error-prone Classes in the Post-release Software Evolution Process," Journal of Systems and Software, Vol. 81, No.11, pp. 1868-1882, 2008.

[Shatnawi et al. 2010] R. Shatnawi, W. Li, J. Swain, and T. Newman, "Finding Software Metrics Threshold Values using ROC Curves," Journal of Software maintenance and Evolution, Vol. 22, No. 1, pp. 1-16, Jan. 2010.

[Shihab et al. 2010] E. Shihab, A. Ihara, Y. Kamei, W.M. Ibrahim, M. Ohira, B. Adams, A.E. Hassan, and K. Matsumoto, "Predicting Re-opend Bugs: A Case Study on the Eclipse Project," IEEE/Working Conference of Reverse Engineering, pp. 249 – 258, 2010.

[Stewart et al. 2005] K. J. Stewart, D. P. Darcy, and S. L. Daniel, "Observations on Patterns of Development in Open Source Software Projects," Proc. ACM/International Conference on Information Systems, St. Louis, Missouri, pp: 1-5, Jul. 2005.

[Sullivan and Chillarege 1992] M. Sullivan and R. Chillarge, "A Comparison of Software Defects in Database Management Systems and Operating Systems," FTCS 1992: 22nd Annual International Symposium on Fault-Tolerant Computing, 1992.

[Timea and Barbara 2010] I. Timea, P. Barbara, "The Vital Few and Trivial Many: An Empirical Analysis of the Pareto Distribution of Defects," Proc. Lecture Notes in Informatics, pp. 151-162, Nov. 2010

[Watanabe et al. 2008] S. Watanabe, H. Kaiya, and K. Kaijiri, "Adapting a Fault Prediction Model to Allow Inter Language Reuse," Proc. ACM/ International Workshop on Predictor Models in Software Engineering, Leipzig, Germany, pp. 19-24, May 2008.

[Wheeldon and Counsell 2003] R. Wheeldon and S. Counsell, "Power Law Distribution in Class Relationships," IEEE/International Workshop on Source Code Analysis and Manipulation, pp. 45-54, 2003.

[Withrow 1990] C. Withrow, "Error Density and Size in Ada Software," IEEE Software, Vol. 7, No. 1, pp. 26-30, Jan/Feb. 1990.

[Yu et al. 2005] L. Yu, S. R. Schach, and K. Chen, "Measuring the Maintainability of Open-source Software," IEEE/International Symposium on Empirical Software Engineering, pp: 297-303, Nov. 2005.

[Zhang 2008] H. Zhang, "On the Distribution of Faults," IEEE Trans. Software Engineering, Vol 32, No 2, pp. 301-302, Mar/Apr 2008.

[Zimmermann et al. 2007] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects for Eclipse," in PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, pp. 9, 2007.