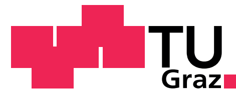
Graz University of Technology
Faculty of Computer Science
Institute for Software Technology



Doctor of Philosophy Dissertation

An Empirical Investigation into Changes and Bugs by Mining Software Development Histories

by

Javed Ferzund

Supervisor: Univ. Prof. Dipl.Ing. Dr.tech. Franz Wotawa

November 2009
Graz, Austria

To My Parents

Foreword

This dissertation was written as a partial fulfillment of the requirements for the degree of Doctor of Philosophy in Informatics at the Graz University of Technology, Austria.

The research work presented in this thesis was carried out at the Institute for Software Technology, Inffeldgasse 16b/2, 8010 Graz, Austria. This work was started in December 2006 and involved both research and development.

The subject of this thesis is to evaluate software repositories in order to develop novel approaches for software debugging. Software changes and bugs are researched to develop models for bug preiction and to study the features of bug introducing changes. A significant part of this dissertation was published at various conferences.

This work was suggested and supervised by Prof. Franz Wotawa and was partially funded by Higher Education Commission, Government of Pakistan.

Abstract

Software repositories hold enormous amount of data that can be used for software evolution studies. Finding and removing bugs from software is a challenging task. Mining development history of software can improve the debugging process. Software configuration management systems record all software changes that are made during its evolution. A significant part of these changes is used to fix bugs in software. Both bug fix and bug introducing changes can be extracted from software repositories. Bug introducing changes can be analyzed to study characteristics of source code that result in bugs. This dissertation presents two empirical studies that investigate the role of language constructs in introducing bugs and influence of programming language on post release bugs.

Revision histories of eight open source projects developed in multiple languages are processed to extract bug-inducing language constructs. Twenty six different language constructs and syntax elements are chosen for this study. Function calls, assignments, conditions, pointers, use of NULL, variable declaration, function declaration and return statement are found to be the most frequent bug-inducing language constructs. They are found in 38-62%, 30-42%, 17-40%, 11-30%, 1-22%, 11-25%, 8-12% and 8-15% of bug inducing hunks respectively. These constructs account for more than 70 percent of bug-inducing hunks. Function Calls are found to be the most dominant source of errors in all projects. Use of pointers and NULL is highly problematic in projects developed in the language C. Different projects are statistically correlated in terms of frequencies of bug-inducing language constructs. Most of the developers tend to face difficulties with similar language constructs. Statistical analysis indicates that the majority of the developers induce similar kinds of bugs independent of the project and programming language.

Within our work the development history of Mozilla project with a span of 11 years had been extracted and different code and evolution metrics had been calculated. Mozilla is a heterogeneous project developed in C, C++ and Java. Defect densities of files written in the three languages are statistically analyzed in order to find a relationship between defects and programming languages. Life

span of bugs within the three kind of programs is also calculated to compare the efforts required to fix bugs in the different languages. Statistical analyses of bug densities revealed that post release bugs are influenced by the programming language. Results of hypothesis testing showed that Java programs are less error prone than C or C++ programs, and that C programs are less error prone than C++ programs in same project. We found that the bug life time of Java programs is twice as long as for C or C++ programs.

This thesis also introduces a new set of metrics called hunk metrics and a technique to classify hunks as buggy or bug-free. The hunk classification approach uses hunk metrics as input variables to classify hunks into buggy and bug-free. Classification models are built using logistic regression and random forests, and their performance is evaluated and compared. Bug prediction abilities of individual metrics are also evaluated. The hunk classification approach is evaluated on eight large open source projects. It can classify hunks as buggy or bug-free with 81% accuracy, 77% buggy hunk precision and 67% buggy hunk recall on average. Hunk metrics related to change and history are found to be better predictor of bugs than code related hunk metrics. Predictors obtained from one project when applied to a different project could classify hunks with more than 60% accuracy.

Zusammenfassung

Software-Repositories halten enorme Menge von Daten, die für die Software-Entwicklung Studien verwendet werden. Suchen und Entfernen von Software-Fehler ist eine anspruchsvolle Aufgabe. Mining Entwicklungsgeschichte von Software zur Verbesserung der Debugging-Prozess. Software Configuration Management-Systeme erfassen alle Software-Änderungen, die während ihrer Entwicklung gemacht werden. Ein erheblicher Teil dieser Veränderungen wird verwendet, um Fehler in der Software beheben. Beide Fehler zu beheben und Fehler der Einführung von Änderungen können von der Software-Repositories extrahiert werden. Bug der Einführung von Änderungen können analysiert werden, um Merkmale der Quellcode zu studieren, die zu Fehlern. Diese Dissertation präsentiert zwei empirische Studien, dass die Rolle der Sprache zu untersuchen Konstrukte bei der Einführung von Bugs und der Einfluss der Programmiersprache über den Post Release Bugs.

Revision Geschichten von acht Open-Source-Projekte in mehreren Sprachen entwickelt werden verarbeitet, um Fehler zu extrahieren-induzierende Sprachkonstrukte. Zwanzig sechs verschiedenen Sprachkonstrukte und Syntax Elemente sind für diese Studie ausgewählt. Funktionsaufrufe, Zuweisungen, Bedingungen, Zeiger, die Verwendung von NULL, der Deklaration von Variablen, Funktion Erklärung und return-Anweisung gefunden werden, um die häufigste Fehler-induzierende Sprachkonstrukte. Sie sind gefunden in 38-62%, 30-42%, 17-40%, 11-30%, 1-22%, 11-25%, 8-12% und 8-15% der Fehler hunks bzw. veranlassen. Diese Konstrukte einen Anteil von mehr als 70 Prozent der Fehler-induzierende hunks. Function Calls finden sich als die wichtigste Quelle von Fehlern in allen Projekten. Verwendung von Zeigern und NULL ist höchst problematisch an Projekten in der Sprache C. Verschiedene Projekte entwickelt werden, korreliert statistisch in Bezug auf die Häufigkeit der Fehler-induzierende Sprachkonstrukte. Die meisten Entwickler neigen dazu, Schwierigkeiten mit ähnlichen Sprache Gesicht Konstrukte. Die statistische Analyse zeigt, dass die Mehrheit der Entwickler, ähnliche Arten von Bugs unabhängig von der Projekt-und Programmiersprache zu induzieren.

Im Rahmen unserer Arbeit die Entwicklung der Geschichte von Mozilla-Projekt mit einer Spannweite von 11 Jahren wurde extrahiert und anderen Code-Metriken und-entwicklung war errechnet worden. Mozilla ist ein heterogenes Projekt in C, C ++ und Java. Defect Dichten von Dateien in den drei Sprachen geschrieben werden statistisch ausgewertet, um einen Zusammenhang zwischen Fehler und Programmiersprachen zu finden. Lebensdauer von Fehlern innerhalb der drei Arten von Programmen ist auch geeignet, die Anstrengungen erforderlich, um Fehler in den verschiedenen Sprachen fix vergleichen. Statistische Analysen ergaben, dass Fehler Dichten nach Freigabe durch die Fehler der Programmiersprache beeinflusst werden. Ergebnisse der Hypothese Tests ergaben, dass Java-Programme weniger fehleranfällig als C oder C ++ Programme sind, und daß C Programme sind weniger fehleranfällig als C ++ Programme in einem Projekt arbeiten. Wir haben gefunden, dass der Fehler Lebensdauer von Java-Programmen ist doppelt so lang wie C oder C ++ Programmen.

Diese wird auch ein neues Set von Kennzahlen genannt hunk Metriken und eine Technik, um hunks als Buggy oder Bug-frei einzustufen. Die Einstufung hunk Metriken Ansatz verwendet als Eingangsgrößen hunks in Buggy und Bug-frei einzustufen. Klassifikation Modelle werden mit Hilfe logistischer Regression und zufällige Wälder, und ihre Leistung wird evaluiert und verglichen werden. Bug Vorhersage Fähigkeiten der einzelnen Kennzahlen werden ebenfalls bewertet. Die Einstufung hunk Ansatz basiert auf acht großen Open-Source-Projekte ausgewertet. Es kann klassifizieren hunks als Buggy oder Bug-frei mit 81% Genauigkeit, 77% Buggy hunk Präzision und 67% Buggy hunk erinnern, im Durchschnitt. Hunk Metriken im Zusammenhang mit der Veränderung und der Geschichte gefunden werden, besser zu sein als Indikator für Fehler im Zusammenhang hunk Code-Metriken. Prädiktoren, erhalten aus einem Projekt, wenn ein anderes Projekt könnte hunks mit mehr als 60% Genauigkeit zu klassifizieren angewendet.

Contents

Contents i

1 Introduction 3
1.1 Roadmap . . . 6
1.2 Empirical Analysis of Language Constructs . . . 7
1.3 Programming Languages and Bugs . . . 8
1.4 Hunk Classification . . . 8
1.5 Terminology . . . 10

2 Extraction of Data from Repositories 13
2.1 Architecture . . . 13
2.2 Database Schema . . . 15
2.3 Extraction of Hunks . . . 16
2.4 Identification of bug-inducing Hunks . . . 17
2.5 Projects Analyzed . . . 19

3 Empirical Analysis of Bug-Inducing Language Constructs 23
3.1 Extraction of Language Constructs . . . 25
3.2 Proportion of Different Hunk Types . . . 30
3.3 Most Frequent Bug-Inducing Language Constructs . . . 31
3.4 Project Similarities . . . 34
3.5 Developer Similarities . . . 36
3.6 Bug Latency . . . 40
3.7 Comparison with Non Bug-Inducing Hunks . . . 42
3.8 Summary . . . 45

4 Language Specific Bug Patterns 47
4.1 Research Hypothesis . . . 48
4.2 Project Studied . . . 48
4.3 Evolution Metrics . . . 49

4.4 Results 50
4.5 Proving hypothesis *H1* 57
4.6 Threats to Validity 61
4.7 Summary 62

5 Hunk Classification 63
5.1 The Approach 64
5.2 Tools Used 65
5.3 Hunk Metrics 65
5.4 Evaluation Criteria 68
5.5 Classification Techniques 69
5.5.1 Logistic Regression 69
5.5.2 Random Forests 71
5.5.3 Principal Component Analysis (PCA) 71
5.5.4 Point Biserial Correlation 72
5.6 Results 73
5.6.1 Correlation between Hunk Metrics and Bugs 73
5.6.2 PCA and Logistic Regression 73
5.6.3 Random Forests 75
5.6.4 Comparison of Logistic Regression and Random Forests . . 76
5.6.5 Performance of Individual Metrics 78
5.6.6 Performance of Combination of Metrics 79
5.6.7 Cross Project Predictions 82
5.7 Applications 84

6 Threats to Validity 87

7 Related Work 89
7.1 Mining Software Change History 89
7.2 Bug Prediction 90
7.3 Software Change Extraction and Analysis 92
7.4 Buggy Code Features and Locations 95

8 Future work 97

9 Conclusion 99

Bibliography 103

A List of Publications 111

List of Figures 113

List of Tables 115

Statutory Declaration **118**

Acknowledgments

I feel immense pleasure to thank the many people who directly or indirectly made this thesis possible.

I am greatly indebted to my supervisor, Prof. Franz Wotawa, for opening the door to research for me. I am thankful to him for his continuous guidance and support at each step of my research career. He gave me freedom to think and work on interesting research topics. He always encouraged me and provided advice whenever it was needed. Without his guidance, it would not have been possible to finish this dissertation.

I am grateful to Prof. Shahram Dustdar for taking time out of his busy schedule to act as second reviewer of my dissertation.

I am thankful to my colleague Syed Nadeem Ahsan for his valuable comments and suggestions on my work. We worked on many research topics together and solved the problems with discussions. We spent a very good time and enjoyed our research work.

I would like to thank the people at IST for making me feel at home. Their friendship and care made my stay comfortable. Especially, I am thankful to Petra Pichler for providing help and support whenever I needed it.

I was lucky to have many good friends from Pakistan Community at Graz. Their love and support made my life easy and joyful. We lived like a family and enjoyed our stay in Graz.

I am grateful to my parents for their continuous love and support in every matter of my life. They allowed me to do whatever I wanted. Whenever I felt downhearted they encouraged me and supported me. Without their support and guidance it would not have been possible for me to achieve precious milestones in my life.

Javed Ferzund
Graz, Austria, November 2009

Chapter 1

Introduction

Changes and bugs are interrelated in the software development process. Some changes are made to fix bugs, and on the other hand bugs are introduced by making changes to software. Change is a basic property of evolving software. When changes are made, errors may be generated in the source code, which result in software failures. These errors in turn are corrected by making changes, so changes and bugs are in a sense complementary to each other. Changes are a must for long life of software. According to Lehman's Law of Program Evolution, software needs to be continuously changed otherwise it will become progressively less useful [59]

Changes are made to the software due to several reasons such as fixing bugs, adding new functionality, performance enhancement, improving compatibility, refactoring etc. Pressman has classified software changes into four categories, correction, adaptation, enhancement and prevention [63]. Corrective changes are made to fix bugs, whereas adaptive changes are required to adjust the software to changes in the external environment. Enhancements are required to extend the functionality of the software. Preventive changes are those made to enhance the life of the software.

Software undergoes the process of aging due to continuous changes applied to it. Parnas has called the effect of continuous change as ignorant surgery. That means, different developers change the software at different times, without a thorough awareness of the software and its design [58]. Usually bugs are to be fixed in short time periods. Due to this time pressure, developers cannot understand the software fully before fixing bugs. A software system is changed by multiple developers. So changes increase inconsistency, complexity, understandability and the size of software. Sometimes these changes introduce new bugs into the source code [67].

Bugs are created due to mistakes or errors in the source code or design of soft-

ware. Software bugs vary in their complexity and severity, and need to be detected and removed before software deployment. Undetected bugs can be detrimental for life and resources [25]. In 1985 Therac-25, a radiation therapy device malfunctioned due to a software bug. It delivered lethal radiation doses and resulted in deaths and injuries [25]. In 1996 Ariane 5, the European Space Agency's rocket was destroyed a few seconds after launch, due to a bug in the guidance computer program. It resulted in a loss of 1 billion US$ [3].

Locating and removing bugs from software is a tedious and time-consuming part of software development. Developers spend a lot of time and effort to find and remove bugs, which is sometimes more expensive than writing new source code [74]. A bug life cycle consists of bug identification, bug assignment, bug fixing, quality assurance and re-assignment of bugs. Bugs are assigned to relevant developers, a process called bug triage [5]. Bugs with highest priority are fixed first and other known bugs are delivered with the software in each release.

Extensive research is going on in software debugging to produce high quality, reliable and bug-free software. Mining software repositories is a new technique to be also applied for software testing and debugging. Many bugs are not detected by the traditional testing techniques like regression testing, unit testing, code reviews and the use of debugging tools. Mining software repositories can explore useful hidden information from software repositories and bug databases [19, 65]. Since software repositories store historical information about changes and bugs, important lessons can be learned by analyzing this information.

Version control systems record changes made to the source code as software development progresses. These systems maintain a log of the changes, including date and time of change, identity of the developer and reason of the change. Bug tracking systems record information related to bugs. These systems hold information about identification, assignment and resolution of bugs. Mailing lists and communication archives record conversation between developers about particular decisions throughout the life of a software project. All this data can be pooled to conduct empirical studies involving software evolution [7, 20]. In this dissertation we focus on three goals:

The first goal of this research is to identify the language constructs which introduce bugs most of the time, thus helping in the debugging process.

The second goal of this research is to study the influence of programming language on the occurence of post release bugs.

The third goal of this research is to help developers in identifying and removing bugs, thereby reducing the testing effort and maintenance costs.

To meet these goals, this work proposes techniques to identify bug-inducing language constructs and to predict bugs in terms of hunk classification. In particular this thesis contributes to the following tasks:

- Empirical analysis of language constructs

- Identification of frequent bug prone language constructs
- Analysis of different projects, developers and programming languages for the frequencies of bug-inducing language constructs
- Analysis of bug densities of programs written in different languages
- Study of various evolution metrics obtained from programs written in different languages
- Exploration of new software metrics to be used as bug predictors
- Development of hunk classification models
- Comparison of predictor models based on statistical and machine learning techniques

The conceptual contribution of this thesis focuses on mining software development history, identification and extraction of bug-inducing hunks, definition of new software metrics, and extraction of language constructs. The technical contribution of this thesis focuses on development of bug prediction models based on metrics, approaches for change classification, and an analysis of language constructs for their role in introduction of bugs. The empirical contribution of this thesis is the application and evaluation of the proposed techniques to the release history of eight large, long lived open source software projects.

The major contributions of this thesis are:

- An approach to extract bug-inducing hunks by processing revision history of a software project. The approach makes use of configuration management systems and bug databases.
- Empirical analysis of bug-inducing language constructs in terms of their frequencies.
- Correlation analysis of different projects, developers and programming languages in terms of frequencies of bug-inducing language constructs.
- Findings about the relationship between programming language and post release bugs
- Comparative study of various evolution metrics
- Definition of 27 hunk metrics and an empirical analysis of these metrics as predictor of bugs.
- Construction of hunk classification models and their evaluation.

1.1 Roadmap

This section describes the layout of this thesis and relationship of each chapter with my selected publications.

Chapter 2 describes the techniques to extract data from software release history. We present the architecture of the database used to store and analyze data for this study. A simple approach is described to extract bug-inducing hunks from change history of a project.

Chapter 3 presents an empirical analysis of language constructs. We identify the language constructs that introduce bugs more frequently. We present a correlation analysis of different projects, developers and programming languages for the frequencies of bug-inducing language constructs. This work contributed to a publication [16] that was presented at Working Conference on Reverse Engineering (WCRE 2009).

Chapter 4 presents a case study to find the influence of programming langauge on post release bugs. We calculate and compare various evolution metrics for programs written in different languages. This work contributed to a publication [4] that was presented at International Conference on Software Engineering Advances (ICSEA 2009).

Chapter 5 describes the hunk classification approach. We define hunk metrics and present a technique to calculate them. We use statistical and machine learning techniques to build classification models. These models are evaluated on data of eight open source projects. This work contributed to two publications [18, 17]. First [18] was presented at International Conference on Software Maintenance (ICSM 2009). Second [17] is to be presented at International Conference on Software Process and Product Measurement (MENSURA 2009).

Chapter 6 discusses the threats to validity. It describes the limitations of this study.

Chapter 7 reviews the related work in the field of mining software change history, bug prediction, software change extraction, software change analysis and buggy code features and locations.

Chapter 8 discusses the future work.

Chapter 9 presents the conclusions.

1.2 Empirical Analysis of Language Constructs

Reducing bugs in software is a key issue in software development. Many techniques and tools have been developed to automatically identify bugs. These techniques vary in their complexity, accuracy and cost. Bug finding tools use predefined bug patterns, model checking and theorem proving to detect bugs. Performance of these tools can be enhanced by paying attention to those language constructs which frequently contribute to bugs. Testing effort can be focused on more risky language constructs. More test cases can be generated and models can be developed for frequently bug-introducing language constructs. Code reviews can be made with a careful examination of bug-introducing language constructs. In this way maintenance cost will be reduced as well as software quality will be improved.

Software repositories maintain record of all changes made to software. These changes are made to fix bugs, to add new features, to improve performance or to restructure the code for easy maintenance. Bug fix changes are identified by a comment recorded by a developer in the configuration management system. These changes can be traced back to the locations, where the bug was actually introduced into the source code [67, 37]. Bug-introducing changes can be extracted from software repositories and their properties can be studied.

This thesis presents an empirical study of bug-inducing changes with a focus on language constructs. One goal of this work is to identify syntax elements of a language which frequently contribute to introduction of bugs. We try to find which language constructs are more problematic. Change history of eight open source projects is analyzed to find, whether there are common language constructs which contribute to bugs. These projects are developed in different languages including C, C++ and Java. We also analyze changes made by different developers to find, whether different developers make similar mistakes.

When developers make a change, they change classes, functions, variables, selection and control structures. We analyze the bug-inducing changes to find the syntax elements which contribute to bugs. Twenty six different language constructs and syntax elements are chosen for this study. We find that most frequent bug-inducing language constructs are function calls, assignments, conditions, pointers, use of NULL, variable declaration, function declaration and return statement. These constructs account for more than 70 percent of bug-inducing hunks. Different projects are statistically correlated in terms of frequencies of bug-inducing language constructs. Developers within a project and between different projects also have similar frequencies of bug-inducing language constructs.

1.3 Programming Languages and Bugs

Comparing pros and cons of various programming languages is an interesting debate among programmers and computer scientist. There exist strong opinions for and against various programming languages. Some studies exist on comparison of programming languages. Prechelt evaluated programs written in different languages for memory consumption, runtime efficiency, reliability, program length and programming effort [62]. A similar study was conducted by Garcia et al. [24] on support for generic programming. The authors identified eight language features that support generic programming. They found that generic features are necessary to avoid awkward designs, poor maintainability, unnecessary run-time checks, and painfully verbose code.

Most of the published work in empirical software engineering that deals with bug detection or bug prediction does not compare the number of post-release bugs for programs written in different programming languages. A number of studies exist on characteristics of bugs and defect prone modules [39, 42, 43, 49, 51]. Li et al. [43] used natural language text classification techniques to analyze bug characteristics in two large open source projects. The authors found that memory-related bugs have decreased except some simple memory-related bugs such as NULL pointer dereferences, whereas security bugs with severe impacts are increasing. They also found that semantic bugs are the dominant root causes, requiring more efforts to detect and fix them. Mohagheghi et al. [49] in an empirical study analyzed the impact of reuse on defect-density and stability, as well as the impact of component size on defects and defect-density in the context of reuse, using historical data on defects, modification rate, and software size.

This thesis presents an empirical study providing insight into post release bugs. In this study programming languages are compared but in a new dimension that is software evolution. It focuses on exploring the influence of programming language on post release bugs. Various evolution metrics are compared for three different languages including C, C++ and Java. Development history of Mozilla project over the past 11 years is used for this study. It is found that Java is less error prone than C language and C language is less error prone than C++ language, at least for the Mozilla project. Although these results are hard to generalize, they provide useful insight into the relationship between programming languages and bugs.

1.4 Hunk Classification

Making changes to software is a crucial task during different phases of software evolution. Changes are required to add new features, to fix the bugs, to improve performance or to restructure the code for easy maintenance. These changes are implemented by adding, modifying or deleting the source code in different files

of software. A file can be changed at one or more places, called deltas or hunks. These hunks of source code, which are added either newly or after modifications, may introduce bugs and result in failures later on. Each hunk has a likelihood of being buggy or bug-free.

A large part of time and resources is consumed in software testing and debugging during the evolution of software. We can save this effort if we can find the parts of the source code where the probability of bugs is more and apply these resources on files which require it most.

In order to predict the number of bugs or to provide a predictor with regard to a classification schema there are two approaches possible. The first approach uses statistical methods like multiple linear regression, logistic regression, and principal components analysis [41, 52]. Linear regression can be successfully used if the dependent variables change linear with the independent variables. As most of the metrics normally correlate with each other, there is a strong need to overcome the multicolinearity problem. Principal component analysis is used in this respect to reduce the multicolinearity effect. Logistic regression can be used for binary classifications.

The second approach relies on machine learning techniques like decision tree induction, support vector machine, artificial neural networks, k-nearest neighbors to mention some of them. Machine learning techniques have the ability to learn from past data and these techniques can be employed in a variety of complex situations (see [72]).

A lot of research has been carried out on bug prediction using different approaches and at different levels of granularity. Most of the researchers have used code metrics as predictors of bugs [29, 40, 52, 55, 15, 14], while others have used process metrics as predictors of bugs [27, 35, 64]. Previous research was focused on different levels of granularity such as modules, files, classes and methods. Some researchers predicted the number of faults for modules or files [52, 55], while others focused on individual classes and methods [29, 56].

This dissertation presents a hunk classification approach that predicts bugs in smallest units of a change, which are hunks. Two prediction models are constructed using statistical and machine learning techniques. The models are built using hunk metrics of previous buggy and bug-free hunks obtained by mining the change history of a software project. Logistic regression and Random Forests are used to build the predictor models.

Our classification approach can classify hunks as buggy or bug free with 82 percent accuracy, 77 percent buggy hunk precision and 67 percent buggy hunk recall on average. Predictors obtained from one project, based on hunk metrics, can be successfully applied to other projects.

1.5 Terminology

This chapter defines various terms used in this thesis.

Software Configuration Management (SCM): It is the process of handling changes made to the software during its development. It is used to control the evolution of software projects. SCM comprises four operations: Identification, control, status accounting and audit. (IEEE Guide to Software Configuration Management. 1987. IEEE/ANSI Standard 1042-1987.)

Bug Tracking System: A bug tracking system is used to store and manage information about bugs such as when a bug is reported, who reported a bug, short description of a bug, severity of a bug, platform on which a bug is reported, module in which a bug is reported and status of a bug.

Version or Revision: These two terms are used interchangeably. A version or revision represents instance of a file at a particular time. As a software system evolves, changes are made to the files. Revisions are used to identify different instances of a changed file.

Version Control: It is an important feature of a software configuration management system, used to manage different revisions of files in a software project.

Commit: It is the process of submitting changes to an SCM system. Initially new files of a project are committed to the SCM system. Then each change to a file is committed. A commit may involve a single file or multiple files together.

Change: Software evolution is characterized by making changes to the files. A change represents a single modification stored in the SCM repository.

Change Delta: It is the result of making a change to a file. The changed lines in a file comprise a change delta.

Added Delta: It consists of the lines added for making a change.

Deleted Delta: It consists of the lines deleted for making a change.

Hunk: Changes are made to files in chunks of source code that are dispersed in a file. These chunks of contiguous source code lines are called hunks. There can be multiple hunks in a change delta.

Modification hunk: If source code lines are modified to make a change, the resulting hunk is called a modification hunk.

Addition Hunk: If new source code lines are added to make a change, the resulting hunk is called an addition hunk.

Deletion Hunk: If new source code lines are removed to make a change, the resulting hunk is called a deletion hunk.

Change log: When a developer commits a change to the SCM system, she records a message describing the purpose of the change. This message is called change log. Change logs can be processed to identify different kinds of changes.

Change Annotation: It is a basic feature of configuration management systems. An SCM system annotates each source code line with the date of modification, author of the line and the revision in which that line was changed.

Bug: A bug is characterized by a programming mistake or error in source code that results in malfunctioning of software.

Fix: A fix is characterized by replacing erroneous source code with the correct code. A fix is used to remove a bug from software.

Bug Fix Change: A change applied to software, to fix a bug is called a bug fix change.

Bug-Inducing Change: A change which resulted in malfunctioning of software later on is called a bug-inducing change or buggy change.

Bug Fix Hunk: A hunk which is part of a fix is called a bug fix hunk.

Bug-Inducing Hunk: A hunk which resulted in malfunctioning of software later on is called a bug-inducing hunk.

Bug-Fix Developer: A developer who makes changes to fix a bug is called a bug-fix developer.

Bug-Inducing Developer: A developer, modifications made by whom resulted in malfunctioning of software, is called a bug-inducing developer.

Chapter 2

Extraction of Data from Repositories

The work presented in this thesis is based on data obtained from mining software release history. Information related to changes and bugs is extracted from configuration management systems and bug databases. Source code and change information is extracted from CVS and SVN repositories. All revisions of each file are analyzed for changes made at different times. Bug information is extracted from Bugzilla and this information is mapped to revisions of files from respective software repositories.

We use our own developed modules to extract information from CVS and bug databases. The extracted information is stored into a database. This database is used for training hunk classification models as well as for analyzing language constructs.

This chapter describes the architecture of the data extraction process, the steps to extract and identify bug-inducing hunks and a schema of the database used to store hunks.

2.1 Architecture

The data extraction process used in this study involves four modules along with a fact database. The four modules are described shortly.

Log Parser It extracts log information from a software repository. Whenever a change is committed to the repository, configuration management system records the purpose of change and meta data of change. Log parser connects

to CVS and SVN, extracts log information for all revisions and stores this information into the fact database.

Annotation Parser It takes annotations for every revision of all files in a project. Configuration management systems annotate each line of code with author and date information. This information is important for analysis of changes. Annotation parser connects to CVS and SVN, extracts annotations for all files and stores this information into the fact database.

Difference Parser It takes difference of two consecutive revisions of each source file, extracts the change deltas and store this information into the fact database.

Bug Parser It extracts bug reports from a bug database and stores this information into the fact database.

Fact DataBase It holds all the information regarding files, revisions, developers, bugs, transactions and changes.

Architecture of data extraction process is depicted in Figure 2.1. Data extraction is completed in four steps:

- Log information is extracted from CVS and SVN repositories. CVS maintains log for each revision of a file separately while SVN maintains log for every revision of the project. So log information from SVN repositories is further processed to relate the log to changed files only.

- Differences are extracted between two consecutive revisions for all files. CVS and SVN provide the facility to view and get differences between two revisions. This information reveals the code additions, deletions and modifications made during the evolution of software.

- Annotations are obtained for each line of code in all revisions. This information is also extracted from CVS and SVN repositories. Annotations are helpful in studying evolutionary aspects of software.

- Bug reports are extracted from bug databases. Bug reports hold important information including descriptions, report and fix dates, developers involved in fixing and patches of code.

Details for extraction and labelling of hunks are described in the next sections.

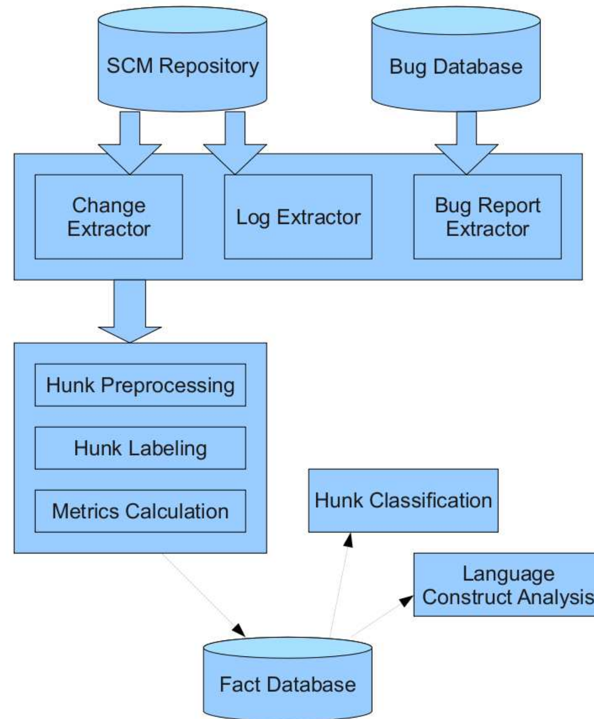


Figure 2.1: Architecture for Data Extraction

2.2 Database Schema

A simple database is designed to hold the log, difference and annotation information. This database is further analyzed to identify bug-inducing and bug-fix hunks. The database consists of three tables, details of which are given below:

CVSLog holds information extracted from log messages for each revision. A description of its attributes is given in Table 2.1.

CVSDifference holds information about change deltas between every two consecutive revisions of each file. A description of its attributes is given in Table 2.2.

CVSAnnotations holds information extracted from annotations obtained for each revision of every file. A description of its attributes is given in Table 2.3.

Table 2.1: CVSLog table description

Field	Type	Description
file	varchar(255),not null	Name and path of the file
revision	varchar(10),not null	Revision number of a file
rdate	date,not null	Revision date and time
author	varchar(50)	Name of author who made the revision
state	varchar(20)	State of the revision
linesadd	integer, not null	Number of lines added to this revision
linesdel	integer, not null	Number of Lines deleted from this revision
comment	longtext	Comments added by the author
bug	varchar(3),not null	Indicates whether a bug is fixed

Table 2.2: CVSDifference table description

Field	Type	Description
file	varchar(255),not null	Name and path of the file
revision	varchar(10),not null	Revision number of a file
hunk_id	varchar(10),not null	represents hunk identifier
hunk_text	text,not null	Contains the actual source code in a hunk
bug_induce	varchar(3),not null	Indicates a bug-induce hunk
bug_fix	varchar(3),not null	Indicates a bug-fix hunk

Table 2.3: CVSAnnotations table description

Field	Type	Description
file	varchar(255),not null	Name and path of the file
revision	varchar(10),not null	Revision number of a file
line_number	integer, not null	Position of a line in the revision
line_revision	varchar(10),not null	line modification revision
author	varchar(50)	author of the line
date	date	date and time of modification
line_code	text	Actual source code of the line

2.3 Extraction of Hunks

Evolution history of a project holds a lot of information including changes made to it. A single change can be applied to one or multiple files. Changes are made in small chunks of code, that are dispersed in a file. These chunks are called hunks.

To extract hunks from a software repository, steps illustrated in Figure 2.2 are used. Execution of these steps populates the tables mentioned in the previous section.

In the first step, log information is obtained for all revisions of each file, using the `log` command of CVS and SVN. A part of CVS log output is shown in Figure 2.4. It contains date and time, author, state, lines added and deleted, commit status and a comment added by the developer. The comment part is processed

```
1. For each file i in a project
2.     For each revision j of file i
3.             Take CVS log of revision j file i and store in a text file X.txt
4.             Extract information from X.txt and store into CVSLog table
5.             Process the comments to mark the revision as bug-fix or not
6. For each file i
7.     For each revision j  of file i
8.             Take diff of revision j and revision j-1 and store in a text file Y.txt
9.             Extract information from Y.txt into CVSDifference table
10.    For each bug-fix revision k of file i
11.            Take CVS annotate of revision k-1 and store into a text file Z.txt
12.            Extract information from Z.txt and store into CVSAnnotations table
```

Figure 2.2: Steps for Hunk Extraction

to identify bug-fix revisions as described in [47, 20].
In the second step, a difference is taken between each pair of consecutive revisions for all files, using the `diff` command of CVS and SVN. A sample of difference output is shown in Figure 2.5. It consists of different hunks, with each hunk indicating the lines added, deleted or modified between the two revisions. The lines starting with `'<'` indicate the lines removed/modified from previous revision, whereas the lines starting with `'>'` indicate the lines added into the current revision. Lines starting with `'>'` are stored into the CVS difference table for each revision. It indicates the code added either newly or after modifications. This portion of code will be used for extraction of language constructs and syntax elements.
In the third step annotations are obtained for all latest revisions preceding the bug-fix revisions, using the `annotate` command of CVS and SVN. A sample of annotations is shown in Figure 2.6. It provides for each line, the last revision in which this line was added or modified, the author who added this line, the date when this line was last added or modified and the actual code. This information helps to identify the origin of the bugs [67].

2.4 Identification of bug-inducing Hunks

Bug-inducing changes can be identified using SZZ algorithm [67, 37]. However SZZ algorithm identifies changes at file level. It does not recognize bug-inducing hunks, rather it considers whole change as bug-inducing. A manual review of bug-inducing changes have shown that not all hunks of a bug-inducing change

```
1. Get set A of bug fix file and revision pairs from CVSLog table
2.      For each file and revision pair i in set A
3.              Get a set B of hunk_ids from CVSDifference table
4.              Mark each hunk_id as bug_fix
5.              For each modification or deletion hunk_id j in set B
6.                      Filter the set C of lines which are modified or deleted in
                                hunk_id j
7.                      For each line k in set C
8.                              Get the line_revision and line_code from
                                CVSAnnotations table
9.                              Get a set D of hunk_id and hunk_text pairs from the
                                CVSDifference table  for line_revision obtained in step 8
10.                             Identify the hunk_text from set D which contains the
                                line_code from step 8, and mark it as bug_inducing.
```

Figure 2.3: Steps for identifying bug-inducing hunks

contribute to bugs. So a technique is required which can discriminate between bug-inducing and non bug-inducing hunks.

This dissertation proposes a technique for identifying bug-inducing hunks. A detail of the technique is illustrated in Figure 2.3. This technique makes use of the database described earlier. The steps to identify bug-inducing hunks are explained using an example. Suppose we have a file from Eclipse project named JDTCompilerAdapter.java. In the first step log information is extracted from the CVS repository. Figure 2.4 shows a sample of log taken for the above mentioned file. It contains information related to revision, author, date, time, lines added or deleted, status and a comment added by the developer. Comments are processed to find keywords Fix, Fixed, Patch, Bug or a numeric identifier of a bug. Such comments are highlighted using boldface in Figure 2.4. To illustrate the hunk identification process, revision 1.66 is selected in which a bug is fixed, revision 1.66 is marked as bug fix revision. To fix a bug in this revision changes were made to revision 1.65.

A difference is taken between revision 1.65 and 1.66. Figure 2.5 shows the difference of both revisions. There are two hunks in Figure 2.5, which are highlighted. First hunk indicates that lines 110-113 are changed in revision 1.65 to line 110 in revision 1.66. Lines starting with `'<'` indicate the lines removed/modified from revision 1.65, whereas the lines starting with `'>'` indicate the lines added into revision 1.66.

To identify the latest revision in which these lines were added, annotations are obtained for revision 1.65. Figure 2.6 shows the annotations organized in a tabular form. Comments are ignored and code of lines 110,115,116 and 117 is

```
----------------------------
revision 1.66

date: 2006-11-28 18:37:52 +0100;  author: oliviert;  state: Exp;  lines: +9 -7;  commitid:
5660456c73ef4567;

HEAD - Fix for 165976
----------------------------
revision 1.65

date: 2006-11-24 02:32:07 +0100;  author: oliviert;  state: Exp;  lines: +2 -2;  commitid:
63d645664b6a4567;

HEAD - Fix for 161975 and 161980
```

Figure 2.4: CVS Log

selected. These lines were recently modified or added in revision 1.38 and revision 1.29 as indicated in Figure 2.6. CVS difference table is queried to identify the hunks in which these changes were made. Figure 2.7 shows all the added hunks in revision 1.38. String comparison is used to identify the hunks in which lines 110,115 and 117 were added. The hunks containing these lines are highlighted in Figure 2.7 and these hunks are marked as bug-inducing hunks.

2.5 Projects Analyzed

For this study 8 open source projects are selected. These projects are selected due to easy availability of their development history and bug information. Table 2.4 shows some statistics of these projects. We describe the projects shortly:

Apache HTTP 1.3 is the most popular web server on the Internet, providing secure, efficient and extensible HTTP services (http://httpd.apache.org/).

Columba is an Email Client written in Java, featuring a user-friendly graphical interface with wizards and internationalization support. We selected for our study the main trunk of Columba.
(http://www.columbamail.org/drupal/)

Eclipse is an integrated development environment (IDE) for software development. We selected JDT part of Eclipse project for our study, that provides Java Development Tools (http://www.eclipse.org/).

Epiphany is a simple and easy to use web browser for GNOME desktop (http://projects.gnome.org/epiphany/).

```
Index: antadapter/org/eclipse/jdt/core/JDTCompilerAdapter.java
===================================================================
RCS file: /cvsroot/eclipse/org.eclipse.jdt.core/antadapter/org/eclipse/jdt/core/JDTCompilerAdapter.java,v
retrieving revision 1.65
retrieving revision 1.66
diff -r1.65 -r1.66
110,113c110
<       if (this.bootclasspath != null && this.bootclasspath.size() != 0) {
<                                   /*
<                                    * Set the bootclasspath for the Eclipse compiler.
<                                    */
---
>       if (this.bootclasspath != null) {
115,117c112,119
<                                   cmd.createArgument().setPath(this.bootclasspath);
<       } else {
<          this.includeJavaRuntime = true;
---
>          if (this.bootclasspath.size() != 0) {
>                                   /*
>                                    * Set the bootclasspath for the Eclipse compiler.
>                                    */
>                                   cmd.createArgument().setPath(this.bootclasspath);
>          } else {
>                                   cmd.createArgument().setValue(Util.EMPTY_STRING);
>          }
```

Figure 2.5: CVS Difference

Line No.	Revision	Stamp	Code
110	1.38	(pmulet 21-Jul-04)	if (this.bootclasspath != null && this.bootclasspath.size() != 0) {
111	1.6	(othomann 05-Sep-02)	/*
112	1.6	(othomann 05-Sep-02)	* Set the bootclasspath for the Eclipse compiler.
113	1.6	(othomann 05-Sep-02)	*/
115	1.38	(pmulet 21-Jul-04)	cmd.createArgument().setPath(this.bootclasspath);
116	1.29	(othomann 29-Sep-03)	} else {
117	1.38	(pmulet 21-Jul-04)	this.includeJavaRuntime = true;

Figure 2.6: CVS Annotations

52c52	this.attributes.log(AntAdapterMessages.getString(ant.jdtadapter.i nfo.usingJDTCompiler), Project.MSG_VERBOSE); //$NON-NLS-1$
62c62	if (!resultValue && this.verbose) {
82c82	**if (this.bootclasspath != null && this.bootclasspath.size() != 0) {**
87c87	**cmd.createArgument().setPath(this.bootclasspath);**
89c89	**this.includeJavaRuntime = true;**
92c92	Path classpath = new Path(this.project);
99c99	addExtdirs(this.extdirs, classpath);

Figure 2.7: CVSDifference table entries

Table 2.4: Description of Projects

Project	Software Type	Language	Period
Apache HTTP 1.3	HTTP Server	C	01/1996-01/2008
Columba	Email client	Java	07/2006-12/2007
Eclipse JDT	Java Development IDE	Java	06/2001-10/2008
Epiphany	Web Browser	C	12/2002-02/2009
Evolution	Groupware Client	C	04/1998-06/2007
Mozilla	Web Browser	C/C++/Java	03/1998-07/2008
Nautilus	File Manager	C	10/1999-02/2009
PostgreSQL	DBMS	C/C++	07/1996-10/2008

Evolution provides integrated mail, address-book and calendaring functionality to users of the GNOME desktop (http://projects.gnome.org/evolution/).

Nautilus is a powerful file manager. (http://projects.gnome.org/nautilus/)

Mozilla is a popular and widely used web browser. (http://www.mozilla.org/)

PostgreSQL is a widely used database management system. (http://www.postgresql.org/)

Chapter 3

Empirical Analysis of Bug-Inducing Language Constructs

As a software evolves, changes are continuously applied to the source code. Software configuration management systems record these changes made to the source code. This information can be extracted and used for software evolution studies. Log messages of a transaction help to identify reasons for software changes [47]. Bug databases hold important information related to bugs [1]. This information can be used to study characteristics and behavior of bugs. Software configuration management data combined with bug data provides a rich source for different kinds of empirical studies. In the recent years research is focused on producing good quality software with reduced costs. Particularly researchers are interested in reducing testing effort and maintenance costs. Most of the work is aimed at fault occurrence and fault prediction in the software [13, 27, 36, 54, 60, 71].

Software change history can be mined to discover interesting change patterns. Research has been conducted on different levels of granularity to find change patterns. Some researchers have studied file co-change patterns [73] , others have studied logical couplings among different modules [12, 23] and line co-change patterns [76] . More fine grained research is also conducted to find application specific patterns, to find item couplings, to predict change propagation and to find signature change patterns [31, 32, 75].

In this chapter an empirical study of changes and bugs is presented. Software change history of 8 open source projects is mined and characteristics of bug-introducing changes are analyzed. A number of language constructs are extracted from bug-introducing changes and their abilities of bug-introduction are

studied. Different language constructs are compared and more bug-prone language constructs are identified.

Revision histories of 8 open source projects are mined to extract bug-inducing hunks. These hunks are processed to extract language constructs and syntax elements which contribute to bugs. The objective of this study is to find language constructs which are more problematic. If such bug-inducing syntactic elements are identified, testing effort can be focused on the most frequent bug-inducing elements. Further developers can be careful while making changes, keeping in mind the frequent bug-inducing elements. When developers make a change, they change classes, functions, variables, selection and control structures. My first research objective is to find which language constructs or syntax elements introduce bugs most of the time. This formulates my first research question:

- *Research Question 1.* What are the most frequent bug-inducing language constructs.
  Different projects are developed for specific purpose and by a different group of developers. Further projects can be developed in different programming languages. So it would be interesting to know which language constructs commonly introduce bugs in different projects. It gives rise to the following two research questions:

- *Research Question 2.* Is the frequency of bug-inducing language constructs similar between projects developed in the same language.

- *Research Question 3.* Is the frequency of bug-inducing language constructs similar between projects developed in different languages.
  Different developers may have different programming skills, so they may feel difficulty with different language constructs and hence introduce different kinds of bugs. There can be domain specific features which increase the difficulty of developers. This observation gives rise to the following research questions:

- *Research Question 4.* Is the frequency of bug-inducing language constructs similar between developers of the same project.

- *Research Question 5.* Is the frequency of bug-inducing language constructs similar between developers of different projects.

- *Research Question 6.* Is the frequency of bug-inducing language constructs similar between developers of the same programming language.

- *Research Question 7.* Is the frequency of bug-inducing language constructs similar between developers of different programming languages.

To conduct this study, 8 open source projects developed in multiple languages and having a long development history are selected. A description of these projects is already given in Chapter 2.

3.1 Extraction of Language Constructs

Bug-inducing hunks are identified using the techniques mentioned in Chapter 2. A static source code parser is implemented in Java, which extracts different syntax elements from a given hunk. It parses the hunk and finds the occurrence of different language constructs. 26 different syntax elements are chosen, and the parser is designed to find these elements. A detail of these syntax elements is shown in Table 3.1, with examples extracted from Eclipse and Apache change data. Syntax elements presented in last five rows of Table 3.1, are extracted for Java files only, whereas pointers, include statement, define statement, structures, assertions and goto statement are not extracted for Java files.

A short description of each language construct is presented below:

Conditions: Conditional expressions provide a selection mechanism in the program. Developers implement conditions in a program to provide multiple paths of execution. Conditions usually evaluate a Boolean expression and depending on the evaluation result, execution path is selected. There can be simple and complex conditions in a program. Simple conditions involve single Boolean expression, whereas complex conditions involve multiple Boolean expressions. Further conditions are nested up to many levels.

As conditions involve Boolean expressions and use of relational operators, developers can make a mistake in selecting appropriate relational or logical operators. Usually equality operator is mistakenly used and it is sometimes missed by testing tools.

Loops: Loops provide an iteration mechanism in a program. Developers use loops to repeat a statement or group of statements many times. There are three kinds of loops, one which executes statements for the specified number of times, the other repeats statements until a specified condition becomes false, and the third one executes statements at least once even if the specified condition is false.

Developers may make a mistake in specifying the counter variable in the loop, or the controlling condition may be set wrong.

Assignments: Assignments are used to set or change the value of a variable. This value can be set using a constant, other variable or an expression. The expression may be arithmetic, logical, object instantiation or some function call.

Table 3.1: Language Constructs

Syntax Element	Symbols	Examples
Conditions	if, else, else if	if (this.compileList.length != 1) {
Loops	for, while, do while	for (int i = 0, max = pathElements.length; i< max; i++) {
Assignments	=	this.target=true;
Function Calls	Foo ( );	classpath.addExisting(new Path(null, jre_lib.toOSString()))
Function Decl./Def.	bar ( ) { }	private void addExtdirs(Path extDirs, Path classpath) {
Variable Declaration	int foo;	Map customDefaultOptions;
Pointers	Int * foo;	char *fspec;
Logical Operators	&&, ||, !	if (!resultValue && this.logFileName != null) {
Relational Operators	<, >, ==, !=, <=, >=	if (this.accessRules == null) {
Return statement	return a;	return ClasspathDirectory + this.path; $//NON-NLS-1$
Use of NULL	foo= NULL;	private Map fileEncodings = null;
Include statement	# include	# include <sys/stat.h>
Define statement	# define	# define MPE_WITHOUT_MPELX44
Structures	struct foo { }	struct utsname os_version;
Assertions	assert ( )	assert(idx < APACHE_ARG_MAX);
Arrays	int foo []	String[] dirs = extDirs.list();
Case statement	case foo:	case READING_JAR:
Goto statement	goto foo:	goto return_from_multi;
Inc-dec operator	++,- -	if (len > 2 && errstr[len-3] == .) len--;
Break statement	break;	state = destinatonPathStart; break;
Exception handlers	try, catch	try {zipFile.close();} catch(IOException e) {
Class declaration	class foo	public class ClasspathDirectory implements FileSystem.Classpath {
New operator	new foo()	this(new ZipFile(file), true, null);
Throw statement	throw foo-exception;	throw new BuildException(Jdtcom ,e); $//NON-NLS-1$
Imports	import	import org.eclipse.core.runtime.IPath;
Inheritance	extends, implements	public class ClasspathJar extends ClasspathLocation {

Developers can make mistakes in assignments by using wrong values or inappropriate expressions.

Function Calls: Functions or methods are a way to modularize programs. In object oriented programming methods act as interfaces to classes. Developers write methods or functions to perform certain tasks. Whenever that task is required, they can make a call to it. A proper syntax of a method call includes method name and its parameters. If the function or method returns a value, it should be used in an assignment expression.

Programmers can make a mistake in providing the correct parameters or arguments to a function call, or they can make a call at the wrong place.

Function Definitions: Functions or methods are required to be defined before they can be called in a program. Method definitions are an essential part of object oriented programming. Classes are incomplete without methods. Method definitions consist of signature of the method and a body of the method. Signature of a method consists of an access specifier, return type, method name and a list of parameters. Method body consists of a set of statements.

Developers can make mistakes in writing signature of a method.

Variable Declarations: Variables are used to occupy memory locations for holding data. Variables can be declared or defined in a program. Variable declaration involves a data type and a variable name, whereas variable definition additionally involves an assignment of initial value to the variable. Variables can be of simple data types or complex user defined data types. In object oriented programming, variables are also used to hold instances of classes.

Developers can make wrong declarations or incorrect instantiations, which may lead to errors in programs.

Pointers: Pointers are a kind of variables which hold memory addresses. Developers use pointers to refer different memory locations in a program. Pointers are extensively used in programs developed in C language. Pointers can be declared of any data type and they can point to memory locations of the same type.

Major draw back of pointers is memory management. Pointers can refer to wrong locations or they can occupy memory when it is no more needed. Developers can make mistakes in pointer initializations or pointer updations. They can also forget to free memory after using it.

Logical and Relational Operators: Logical operators are used to combine Boolean expressions whereas relational operators are used to construct Boolean expressions. They are normally used as part of the conditions and loops.

Developers can make a mistake in using the appropriate operator at the appropriate place.

Return Statement: Return statement is used in a method or function to return a value. If a return type is mentioned in method signatures, it should have a return statement in its body. Return statement is a way to use the results of a function execution outside the body of a function.

Developers can forget to return a value or they can make a mistake in returning the correct value.

Use of Null: Null is treated as 0 or void in C and C++. In Java it is a special literal of the null type and it doesn't necessarily have value zero. It is impossible to cast to the null type or declare a variable of this type.

Developers can make invalid use of null or they can make mistakes in assigning null.

Include Statement: Include statement is used to combine library files or other user written files in a C or C++ program.

Define Statement: Define statement is used to define macros in a C or C++ program.

Structures: Structures are a way to combine different data types into a single data type. In procedural languages structures are used to combine variables and functions. A structure represents a complex data type consisting of multiple simple data types.

Developers can make a mistake in defining the structure or assessing the elements of the structures.

Assertions: In large programs, before proceeding further it is useful to know whether a condition or set of conditions is true. To start a particular computation, developers usually make sure that the program is in a state, in which they believe it to be. It is accomplished by use of a statement called assertion. If an assertion fails, a diagnostic message can be displayed and the program is terminated.

Programmers can make mistakes in using valid assertions.

Arrays: Arrays provide a way to store collection of data items of same type at contiguous memory locations. Individual elements can be accessed by specifying the index of that element. Object oriented languages provide functions related to arrays that can be used to manipulate arrays.

Developers can make mistakes in declaring arrays or accessing the elements of an array.

Case Statement: Switch statement provides a way to have multiple execution paths based on the value of a single variable. Different values of the switch variable are provided by using case statement. During the execution of a program, statements after the matching case are executed. A default case is also provided, which is executed when none of the cases match with the current value of the switch variable.

Developers can make incorrect use of cases.

Goto Statement: Goto statement is used to shift control from one place to another place in a program. It is used in programs written in C language. Labels are used to mark locations in a program, goto statement can shift control to these labels.

Programmers can make erroneous use of goto statement.

Increment-decrement Operator: Increment operator when applied to a variable, increases its value by adding one to it. Similarly decrement operator when applied to a variable, decreases its value by subtracting one from it. These operators are short notation of an assignment expression, doing the same. Use of the operator on left or right side of the operand produces different results.

Programmers sometimes do not make use of increment-decrement operator carefully and unexpected results are produced.

Break Statement: Break statement is used in loops to stop the iterations of a loop based on some condition. Sometimes you do not want the loop to complete the specified iterations, and stop the repetition based on the state of an external variable. Break statement helps in such kind of situations.

Mishandling of break statement can produce unexpected results.

Class Declaration and Definition: Classes are the core of object oriented programming languages. Classes implement the data encapsulation, inheritance and polymorphism, that are typical features of object oriented programming. Classes are composed of data members and methods, with public, private and protected access specifiers for these two. A class can be used in a program by creating instances of it, which are called objects.

Programmers can make several types of mistakes while defining classes.

New Operator: New operator is used when a new instance of a class is required. New operator reserves memory for an instance of a class and names it with the variable for which that instance is created.

Programmers can mistakenly create wrong instantiations, or they may use wrong arguments to the constructor of a class.

Import Statement: Import statement is used to include different packages in a program. It is not very much concerned with errors, however it may indirectly involve in creation of bugs.

Inheritance: It is a typical feature of object oriented programming. A class can inherit either from a single class or multiple classes. C++ supports multiple inheritance, whereas in Java interfaces are used to implement multiple inheritance. By using inheritance, features of the parent class are transferred to the child class. The child class can have additional features of its own.

Improper handling of inheritance can result in multitude of errors which cause failure of the program.

Exception Handlers: Exception handling is a way to trap known errors in a program. It is implemented by a try and catch mechanism. Parts of the code which are known to generate errors are placed in a try block. Each try block is accompanied by a catch block, in which error handling code is placed. Exception handling prevents a program from terminating, when an error occurs.

An exception may not be trapped by the catch blocks provided and result in program failures.

Throw Statement: Throw statement is used to throw an exception of a specified type.

Invalid throw statement can result in errors, causing malfunctioning of a program.

3.2 Proportion of Different Hunk Types

Extracted hunks are grouped into four categories based on the bug information. These hunk types are:

Bug-Fix Hunks These hunks are part of bug-fix changes. A bug-fix hunk is created when a developer fixes a bug.

Bug-Inducing Hunks These hunks are origin of bugs. A bug-inducing hunk is created when a developer makes a change, which results in failure later on.

Bug-Fix-Inducing Hunks These hunks are part of bug-fix changes but introduce bugs later on. A bug-fix-inducing hunk is created when a developer fixes a bug but at the same time introduces another bug.

Clean Hunks These hunks are neither part of bug fixes nor introduce any bug.

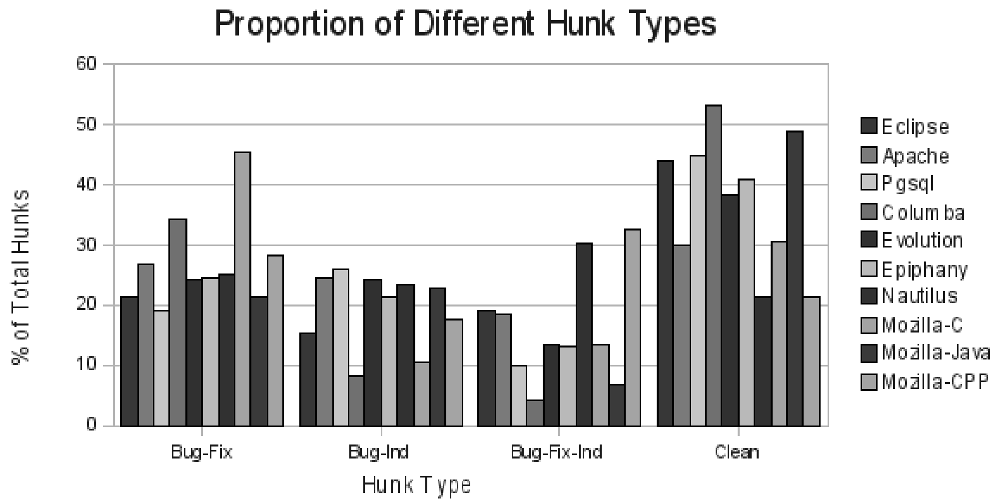


Figure 3.1: Proportion of hunk types in different projects

As development of software progresses new features are added and size of software grows. Chances of errors are increased as the number of changes increases. Bug-inducing hunks constitute a significant proportion of total hunks made in the development history of a project. Figure 3.1 shows the proportion of different types of hunks in 8 projects. Mozilla project is described with three languages separately.

All projects have more than 20% bug-fix hunks. Proportion of bug-inducing hunks is higher in projects developed in C language. Mozilla and Nautilus have a higher percentage of bug-fix-inducing hunks. Projects developed in JAVA have comparatively higher percentage of clean hunks.

3.3 Most Frequent Bug-Inducing Language Constructs

Frequencies of language constructs in bug-inducing hunks are calculated. A majority of the bug-inducing hunks involved a change to more than one language construct. To answer the research question 1, for each language construct, the proportion of total bug-inducing hunks, it was involved in is calculated. The most frequent bug-inducing language constructs are function calls, assignments, conditions, pointers, use of NULL, variable declaration, function declaration and return statement. Table 3.2 and 3.3 show the proportion of total bug-inducing hunks which contain a given language construct, expressed as percentage values. Columns from 2 to 8 in Table 3.2 indicate the percentage of total hunks involving a specific language construct for Apache, Epiphany, Evolution, C files of Mozilla, C++ files of Mozilla, Nautilus and PostgreSQL respectively. In Table

Table 3.2: Frequencies of Bug-Inducing Language Constructs(a)

Syntax Element	Ap.	Ep.	Ev.	Moz-C	Moz-CPP	Nau.	Pg-SQL
Conditions	40	22	29	28	28	21	17
Loops	11	4	7	5	4	4	6
Assignments	38	39	42	35	31	31	25
Function Calls	54	57	62	38	47	59	36
Function Declaration	12	12	11	8	8	11	7
Variable Declaration	14	24	25	16	14	18	13
Pointers	30	24	29	15	11	24	12
Logical Operators	30	16	18	17	15	15	10
Relational Operators	23	17	15	12	14	14	9
Return statement	15	9	11	13	14	8	7
Use of NULL	19	22	22	11	0.8	18	6
Include statement	0.69	7	5	1	2	5	1
Define statement	2	2	1	1	0.65	2	0.57
Structures	2	1	3	0.83	0.2	0.58	1
Assertions	0.09	0	0	0.01	0.01	0	0.07
Arrays	10	5	4	11	3	2	6
Case statement	2	2	3	2	1	1	5
Goto statement	0.38	0.35	0.57	3	0.19	0.22	0.23
Inc-dec operator	2	0.45	0.63	4	2	0.37	3
Break statement	3	2	3	3	1	1	2

3.3 columns from 2 to 4 provide values for Columba, Eclipse and Java files of Mozilla respectively.

Function calls range from 38-62%, assignments range from 30-42%, conditions range from 17-40%, pointers range from 11-30%, use of NULL ranges from 1-22%, variable declarations range from 11-25%, function declarations range from 8-12% and return statement ranges from 8-15% in the studied projects. Columba contains a high number of bug-inducing hunks involving imports and object instantiations (use of new operator). Use of increment-decrement operator, case statement and object instantiations is high in bug-inducing hunks of Eclipse. Arrays have caused more problems in Apache, Eclipse and C files of Mozilla. Number of goto statement is higher in bug-inducing hunks of Mozilla C files as compared to other projects.

More than 50% bug inducing hunks of Apache involve function calls and about 40% bug inducing hunks have conditions and assignments. Pointers are present in 30% bug inducing hunks of Apache. Function declarations, variable declarations, null, return statement and loops are present in 12%, 14%, 19%,15% and 11% bug inducing hunks of Apache respectively. About 10% bug inducing hunks of Apache

Table 3.3: Frequencies of Bug-Inducing Language Constructs(b)

Syntax Element	Columba	Eclipse	Mozilla-J
Conditions	20	31	17
Loops	8	7	4
Assignments	37	33	30
Function Calls	50	41	39
Function Declaration	8	11	10
Variable Declaration	20	12	11
Logical Operators	9	17	11
Relational Operators	12	15	10
Return statement	10	14	9
Use of NULL	5	7	4
Arrays	4	11	7
Case statement	0.59	11	5
Inc-dec operator	0.59	8	5
Break statement	0.59	3	3
Exception handlers	4	2	2
Class declaration	4	2	2
New operator	17	10	6
Throw statement	3	2	4
Imports	12	3	0.47
Inheritance	4	1	1

also involve use of arrays. Remaining language constructs are present in less than 3% bug inducing hunks of Apache.

Epiphany has almost similar proportion of language constructs to Apache, present in bug inducing hunks. However proportion of conditions, pointers, loops and return statement is comparatively less with 22%, 24%, 4% and 9% bug inducing hunks involving these constructs. Variable declarations are present in 24% bug inducing hunks of Epiphany. Surprisingly, proportion of include statements is higher in bug inducing hunks of Epiphany.

More than 60% bug inducing hunks of Evolution involve function calls and conditions are found in 29% bug inducing hunks. Proportion of other language constructs is similar to Apache, with slightly higher number of include statements.

C and C++ files of Mozilla have similar proportion of language constructs in bug inducing hunks. Both kinds of files vary in function calls, use of null and arrays. Number of function calls is higher in C++ files whereas use of null and number of arrays is higher in bug inducing hunks of C files. Function calls are present in 47% and 38% bug inducing hunks of C++ and C files respectively. Null is used in 11% bug inducing hunks of C++ files, whereas in C files this proportion is less than 1%. Arrays are present in 11% bug inducing hunks of C files and 3%

of C++ files.

Conditions, loops, assignments, function declarations, variable declarations, return statement and pointers are present in 28%, 5%, 35%, 8%, 16%, 13% and 15% bug inducing hunks of Mozilla C files respectively. C++ files have similar proportion of these constructs.

Nautilus has similar proportion of language constructs as found in bug inducing hunks of Apache.

PostgreSQL has slightly lower proportion of language constructs in its bug inducing hunks as compared to other projects. Function calls are present in 36% and assignments in 25% bug inducing hunks. Use of null and function declarations is very low in PostgreSQL as compared to other projects. Conditions are found in 17% bug inducing hunks of PostgreSQL. Other language constructs are present in less than 10% bug inducing hunks.

In Columba project, 50% bug inducing hunks involve function calls, whereas assignments, conditions, loops, variable declarations, function declarations and return statement are present in 37%, 20%, 8%, 20%, 8% and 10% bug inducing hunks respectively. Columba project surprisingly has higher number of import statement in its bug inducing hunks. About 12% bug inducing hunks contain import statement. Columba also takes a lead in the use of new operator. Object instantiations have created more bugs in Columba as compared to other projects.

Eclipse and Java files of Mozilla have more or less similar proportion of different language constructs in bug inducing hunks. Conditions, return statement and use of null have created more problems in Eclipse as compared to Java files of Mozilla. Eclipse also leads in the use of case statement and arrays in its bug inducing hunks. About 11% bug inducing hunks of Eclipse contain case statement.

Increment-decrement operator is present in 8% bug inducing hunks of Eclipse. This percentage is highest among all projects. Function calls, assignments and conditions are present in 39%, 30% and 17% bug inducing hunks of Mozilla Java files. Other language constructs are present in less than 11% bug inducing hunks.

3.4 Project Similarities

In order to answer research questions 2 and 3, we analyzed the data using Pearson Correlation. There are some language constructs specific to a particular language, so we selected the language constructs which are common to C, C++ and Java languages. Table 3.4 shows the values of correlation coefficients with $p<0.001$. Columns from 2 to 11 represent correlation values for Apache (Ap.), Columba (Col.), Eclipse (Ecl.), Epiphany (Epi.), Evolution (Evo.), Mozilla Java files (Mz-J), Mozilla C files (Mz-C), Mozilla C++ files (Mz-CPP), Nautilus (Nau.) and PostgreSQL (Pg-SQL).

The correlation coefficients range from 0.84-0.99, indicating that all projects

Table 3.4: Correlation coefficients for different projects

Project	Ap.	Col.	Ecl.	Epi.	Evo.	Mz-J	Mz-C	Mz-CPP	Nau.	Pg-SQL
Ap.	1.0	0.84	0.90	0.92	0.92	0.90	0.92	0.90	0.90	0.93
Col.		1.0	0.87	0.92	0.94	0.93	0.89	0.88	0.93	0.95
Ecl.			1.0	0.86	0.87	0.96	0.91	0.93	0.84	0.93
Epi.				1.0	0.99	0.92	0.94	0.85	0.98	0.95
Evo.					1.0	0.93	0.94	0.89	0.98	0.96
Mz-J						1.0	0.96	0.94	0.91	0.96
Mz-C							1.0	0.90	0.89	0.97
Mz-CPP								1.0	0.85	0.92
Nau.									1.0	0.93
Pg-SQL										1.0

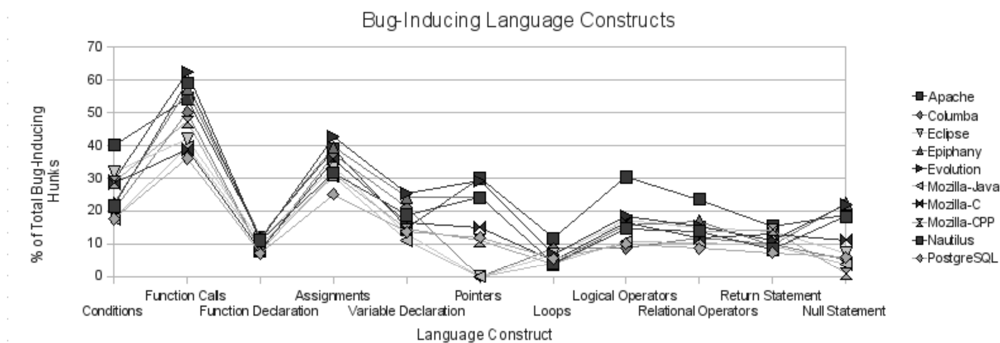


Figure 3.2: Bug-inducing language constructs in different projects (a)

are statistically correlated for the frequencies of bug-inducing language constructs. Projects developed in the same programming language are highly correlated except Eclipse and Columba, for which correlation coefficient is 0.87. Projects developed in different languages are significantly correlated but the correlation coefficients are slightly lower as compared to projects developed in the same programming language. We can see in Figure 3.2 and 3.3 that all projects have similar patterns of bug-inducing language constructs.

Highest correlation is found between Evolution and Epiphany and lowest correlation between Columba and Apache. Apache is statistically correlated to other projects, for frequencies of bug inducing language constructs with a correlation coefficient of greater than 0.9. Columba has strong correlation with Epiphany, Evolution, Nautilus and PostgreSQL having more than 90% correlation.

Eclipse is highly correlated with Mozilla and PostgreSQL. It has 86%, 87% and 84% correlation with Epiphany, Evolution and Nautilus.

Mozilla is also highly correlated with all other projects having correlation values above 90%.

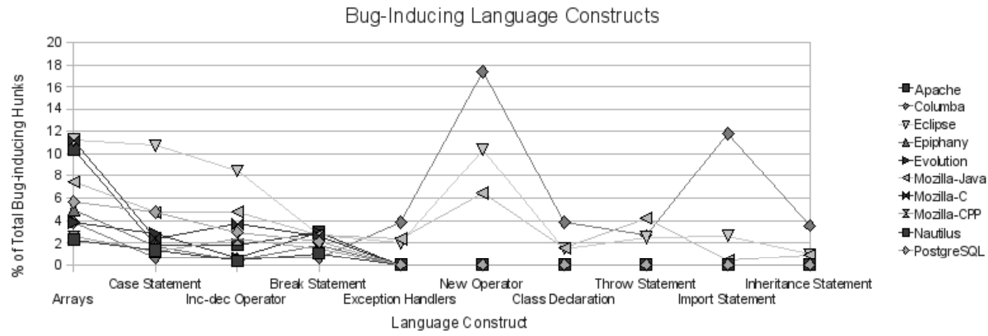


Figure 3.3: Bug-inducing language constructs in different projects (b)

Nautilus has strong correlation with all projects except Eclipse and Mozilla. It has 84% and 85% correlation with Eclipse and Mozilla C++ files. Other correlations are above 90%.

PostgreSQL has strong relationships with all projects, having correlation values above 92%.

3.5 Developer Similarities

In order to answer research questions 4, 5, 6, and 7, we calculated the frequencies of bug-inducing language constructs for each developer of all projects. We selected the 10 most bug-inducing developers from each project, except Columba in which case only 5 developers were involved in bug-inducing hunks, and applied the Pearson correlation on the selected data. Table 3.5 shows the correlation coefficients between developers of the same project, whereas correlation among developers of different projects is given in Table 3.6. Due to the space constraints we mention only the minimum and maximum values of the correlation coefficients. For detailed frequency distribution of correlation coefficients see Figure 3.4 and 3.5. Results of correlation analysis presented in Table 3.5 and 3.6 are obtained for 10 selected developers from each project. However the correlation coefficients depicted in Figure 3.4 and 3.5 are calculated for all developers. Some developers are very active and others contribute at irregular intervals. Developers having minor contributions will have weak correlation with the active developers. So the correlations in Figure 3.4 and 3.5 are as low as 0.15 and -0.1. However majority of the correlations are above 80% for developers from different projects and above 90% for developers from the same project.

Most of the developers of different projects have similar frequencies of bug-inducing language constructs. Table 3.6 shows the minimum and maximum values of correlation coefficients obtained. Developers of the projects developed in

Table 3.5: Correlation Coefficients (developers of same project)

Project	Min. Value	Max. Value
Apache	0.82	0.98
Columba	0.54	0.89
Eclipse	0.70	0.98
Epiphany	0.64	0.98
Evolution	0.95	0.99
Mozilla-J	0.76	0.97
Mozilla-C	0.31	0.97
Mozilla-CPP	0.88	0.98
Nautilus	0.89	0.99
PostgreSQL	0.33	0.97

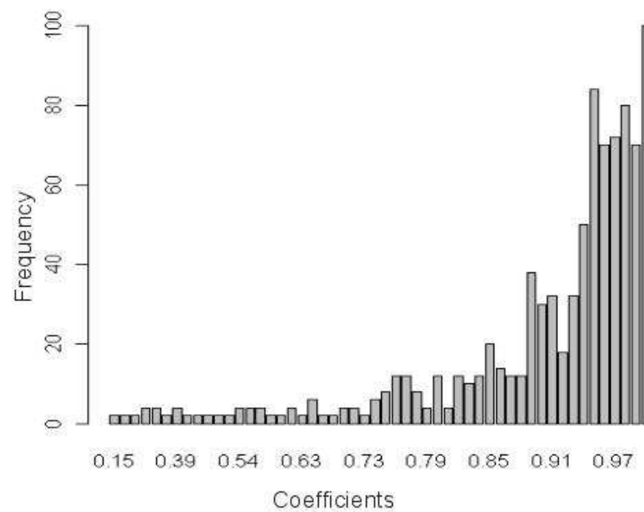


Figure 3.4: Frequency distribution of correlation coefficients (same project)

the same language have higher correlation values as compared to developers of the projects developed in the different languages. However there are a very few developers, who vary in frequencies of bug-inducing language constructs, with correlation values as low as 0.19.
Developers of the same programming language have strong correlations, with a few exceptions for each language. Table 3.7 shows the minimum and maximum values of the correlation coefficient obtained for developers of each language. There are very few developers of each language which vary from other developers of the same language.

- *Answer to Research Question 4.* Pearson correlation analysis shows that developers within the same project are strongly correlated for the frequencies

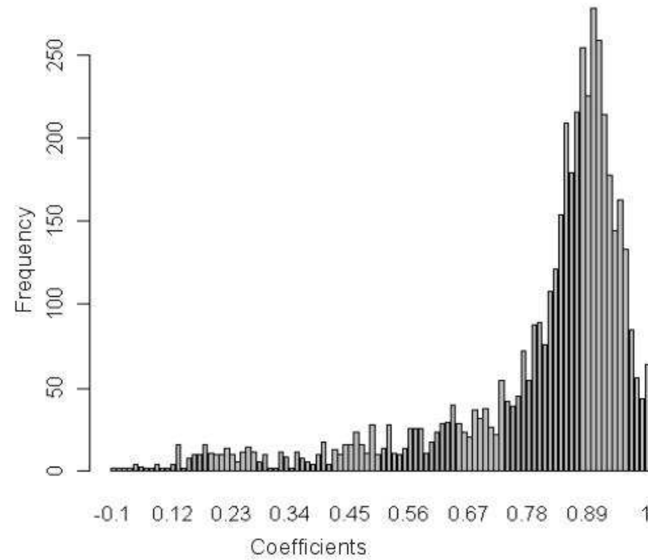


Figure 3.5: Frequency distribution of correlation coefficients (different project)

of bug inducing language constructs. The correlation coefficients within the same project range from 0.31 to 0.99.

Minimum correlation among any pair of developers of Apache is 0.82 and maximum correlation found is 0.98. Similarly, minimum correlation among any pair of developers of Columba is 0.54 with a maximum correlation of 0.89.

Results of correlation analysis on developers of Eclipse indicate a minimum correlation coefficient of 0.7 and a maximum correlation coefficient of 0.98. For developers of Java files in Mozilla similar results are found.

Developers of Evolution are strongly correlated having correlation coefficient above 0.94.

Developers of C files in Mozilla and PostgreSQL have shown similar results. In both cases, minimum correlation found among any developers is about 0.3 and the maximum correlation coefficient is 0.97.

Correlation analysis of frequencies of bug inducing language constructs for developers of Nautilus and C++ files in Mozilla has produced similar results. Minimum correlation among any pair of developers of these projects is 0.88 and maximum correlation coefficient found is 0.99.

Note that these results are for top ten developers from each project. From each project ten developers are selected which have introduced most of the bugs.

Table 3.6: Correlation Coefficients (developers of different projects)

Project Language	Min. Value	Max. Value
Same	0.82	0.98
Different	0.19	0.89

- *Answer to Research Question 5.* A correlation analysis is applied on data of developers from different projects. These projects are developed in C, C++ and Java. Results obtained indicate a minimum correlation coefficient of 0.82 among any pair of developers of different projects but developed in the same language. The maximum correlation coefficient found is 0.98 for the same set of developers.

  Correlation analyses of developers of different projects that are developed in different languages indicate a minimum correlation coefficient of 0.19, whereas maximum correlation coefficient is 0.89 for the same set of data.

- *Answer to Research Question 6.* Developers are grouped into three categories depending on the programming language. Developers of Java are grouped together irrespective of the project, similarly developers of C are grouped together and developers of C++ are grouped separately. A correlation analysis is applied on each group in order to know the relationships among developers of the same programming language. Table 3.7 shows the minimum and maximum values of the correlation coefficient obtained for developers of each language.

  Results obtained indicate a minimum correlation of 0.62 among any pair of developers of C language, whereas maximum correlation coefficient found is 0.97 for the same set.

  Correlation analyses of developers of C++ language indicate a minimum correlation coefficient of 0.88 and a maximum correlation coefficient of 0.98.

  Minimum correlation coefficient among any pair of developers of Java language is 0.54 and maximum correlation found is 0.98.

- *Answer to Research Question 7.* Developers of different programming languages are pooled together and a correlation analysis is applied on the grouped data. Last row of Table 3.6 shows the minimum and maximum values of correlation coefficients obtained among developers of different languages. Results of the correlation analysis indicate a minimum correlation coefficient of 0.19 among any pair of developers of different programming languages. Highest correlation coefficient found is 0.89 among developers of different programming languages.

Table 3.7: Correlation Coefficients (developers of same language)

Programming Language	Min. Value	Max. Value
C	0.62	0.97
C++	0.88	0.98
Java	0.54	0.98

3.6 Bug Latency

When a developer makes a change to fix a bug, configuration management system records the date and time of the commit. During the process of finding bug-inducing changes, as described in Chapter 2, date of modification for each bug-inducing change can be extracted. Interval between bug-induce date and bug-fix date can be calculated in number of days, as well as in number of revisions. In this study number of revisions made between bug-induce date and bug-fix date is calculated. This value is called bug life time or bug latency and calculated for each bug-inducing language construct. CVS maintains revisions of each file, whereas SVN maintains revisions at the project level. Whenever a change is made, CVS updates the revision of the changed file, whereas SVN increments the revision of whole project.

Bug latency values for Apache, Columba, Epiphany, Evolution and Nautilus are calculated by taking difference of project revision numbers and for the rest of the projects by taking difference of file revision numbers. Table 3.8 shows the average bug latency values, calculated in terms of number of revisions the bug existed, for five language constructs. Columns 2 to 6 indicate bug latencies for conditions, assignments, function calls, variable declarations and function declarations respectively.

In Apache project function calls are fixed on an average earlier than other language constructs. Conditions have more average bug latency than other constructs.

For Columba conditions are found more critical and they are fixed on an average earlier than other constructs. Assignments and function calls have equal bug latency and buggy variable declarations persist longer in Columba.

In Eclipse project function declarations are fixed on an average earlier than other language constructs. Conditions and function calls have equal bug latency values, similarly assignments and variable declarations have on average equal bug latency. Buggy assignments and variable declarations persist longer in Eclipse on an average.

Conditions and function calls are more critical in Epiphany and Evolution, as compared to other constructs. Function declarations persist longer in Epiphany and variable declarations persist longer in Evolution.

In Mozilla project function declarations are fixed on an average earlier than

Table 3.8: Bug Latency (Average Values)

Project	Conds.	Assig.	Funct-Calls	Var-Decl.	Funct-Decl.
Apache HTTPS 1.3	3389	2944	2562	3127	2695
Columba	206	209	209	227	213
Eclipse JDT	159	187	159	187	114
Epiphany	1979	2125	2018	2212	2832
Evolution	4532	4675	4515	5031	4987
Mozilla	124	101	106	116	91
Nautilus	1518	1739	1671	1656	1731
PostgreSQL	109	107	111	85	103

Table 3.9: Bug Latency Correlation Values between Language Constructs

	Conds.	Assig.	Funct-Calls	Var-Decl.	Funct-Decl.
Conditions	1.0	0.99	0.98	0.99	0.96
Assignments		1.0	0.99	0.99	0.98
Function Calls			1.0	0.99	0.99
Variable Declaration				1.0	0.98
Function Declaration					1.0

other language constructs. Conditions took more time to be fixed compared to other constructs.

For Nautilus conditions have on average short bug latency and assignments have long bug latency. Bug latency values of other constructs lie between these two constructs.

Variable declarations are more critical in PostgreSQL with shorter bug latencies, whereas conditions have longer bug latencies. Function calls, conditions and assignments have nearly similar bug latencies in PostgreSQL.

A correlation analysis is applied on average bug latency values of conditions, assignments, function calls, variable declarations and function declarations in the studied projects. Results of the correlation analysis are presented in Table 3.9. These language constructs are statistically correlated for bug latency. Most of the correlation coefficients are above 0.95.

It indicates that bug latencies for individual language constructs vary in similar fashion in different projects. Short bug latency indicates that the bug is critical and needs to be fixed soon. Long bug latency indicates that either the bug is minor having low priority or it is more complex to be fixed. In this study average values are used, so a more detailed study is required for some concrete conclusions. However this study represents a brief picture of bug latencies of different language constructs.

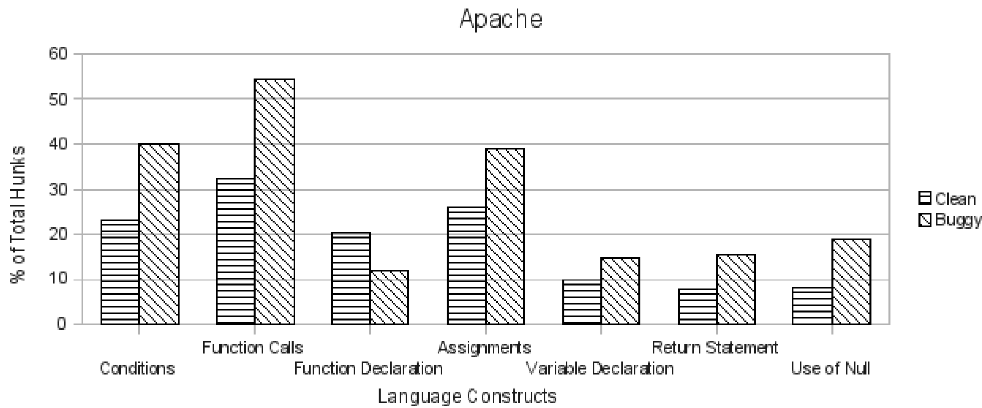


Figure 3.6: Comparison of Bug-Inducing and Clean Hunks (Apache)

3.7 Comparison with Non Bug-Inducing Hunks

Bug inducing and clean hunks are compared for the occurrence of conditions, function calls, function declaration, assignments, variable declarations, return statement and use of null. Although these constructs are also present in non bug-inducing hunks, there percentage is higher in bug-inducing hunks. Among all these constructs function declarations have different trend, they are present in higher percentage of clean hunks in all projects. It can not be stated that each time one of these constructs is used, bugs will be introduced. The context in which these constructs are used is important. However we can say these are the risky language constructs because most of the bug-inducing hunks involve these constructs.

Figure 3.6 shows that the percentage of bug inducing hunks containing conditions is about double of clean hunks in Apache project. There is a large difference between percentages of bug inducing and clean hunks involving function calls. Other constructs also constitute large proportion of bug inducing hunks as compared to clean hunks.

In Eclipse project conditions are present in more than 30% of bug inducing hunks whereas in clean hunks this proportion is less than 20%, as depicted in Figure 3.7. Function calls are present in more than 40% bug inducing hunks and 30% clean hunks. Return statement is present in equal proportions in both kinds of hunks. For the remaining constructs differences are not large but bug inducing hunks have higher percentages as compared to clean hunks.

Figure 3.8 shows that the percentage of bug inducing hunks containing return statement and using null is about double of clean hunks in Mozilla project. Remaining constructs are present in comparatively higher percentage of bug inducing hunks. Conditions constitute 20% of clean hunks and 29% of bug inducing

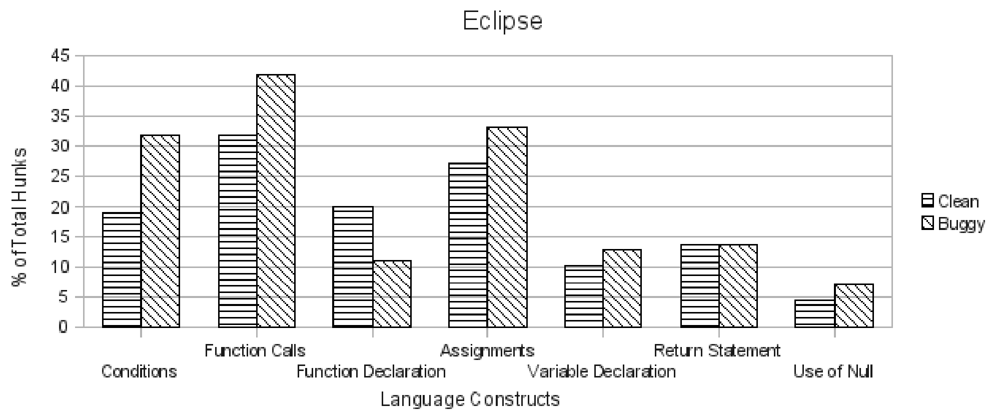


Figure 3.7: Comparison of Bug-Inducing and Clean Hunks (Eclipse)

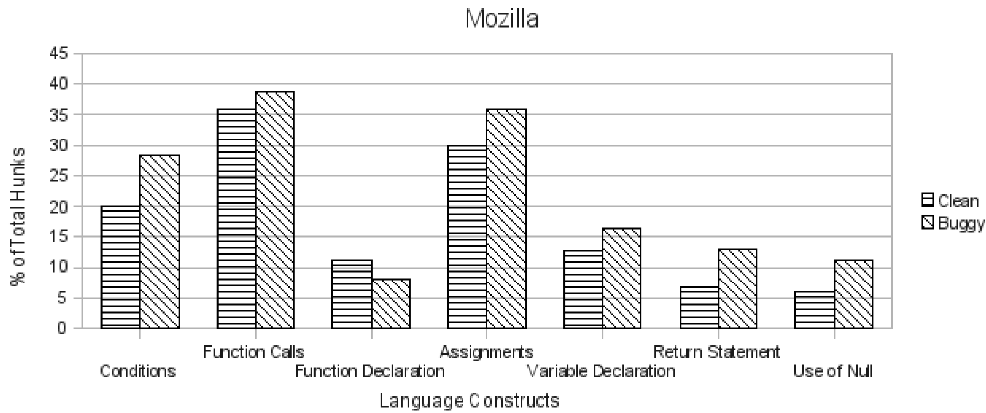


Figure 3.8: Comparison of Bug-Inducing and Clean Hunks (Mozilla)

hunks. Assignments constitute 30% of clean hunks and about 36% of bug inducing hunks.

In PostgreSQL percentage of bug inducing hunks containing return statement is about double of clean hunks, as shown in Figure 3.9. Use of null is almost double in bug inducing hunks as compared to clean hunks. Conditions are present in about 11% of clean hunks and 18% of bug inducing hunks. Assignments constitute 17% of clean hunks and 25% of bug inducing hunks. Function calls are found in 36% of bug inducing hunks and 25% of clean hunks.

Figure 3.10 depicts that conditions are present in 30% bug inducing hunks and less than 20% clean hunks of Evolution. Assignments are found in 30% clean hunks and more than 40% bug inducing hunks. More than 60% bug inducing hunks contain function calls whereas in clean hunks this proportion is less than

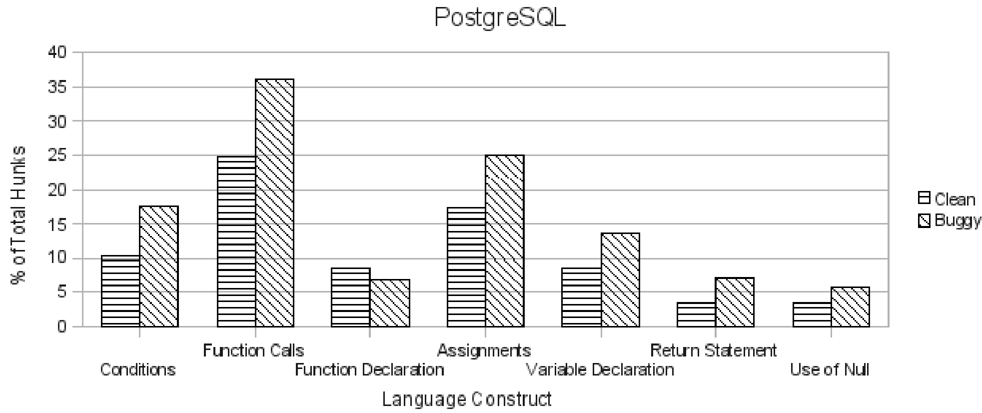


Figure 3.9: Comparison of Bug-Inducing and Clean Hunks (PostgreSQL)

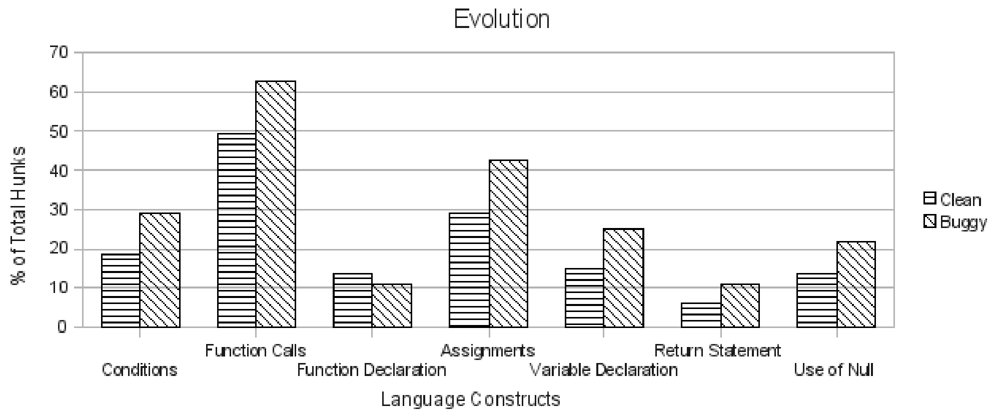


Figure 3.10: Comparison of Bug-Inducing and Clean Hunks (Evolution)

50%. Use of return statement is almost double in bug inducing hunks as compared to clean hunks.

Use of null and conditions is almost double in bug inducing hunks of Epiphany as compared to clean hunks, see Figure 3.11. Assignments are found in 26% of clean hunks and 40% of bug inducing hunks, whereas function calls are present in 58% of bug inducing hunks and 42% of clean hunks.

In Columba use of null and conditions is almost double in bug inducing hunks as compared to clean hunks, see Figure 3.12. Return statements are equally present in both kinds of hunks. Variable declarations are found in higher percentage of clean hunks, in contrast to other projects. Assignments constitute 24% of clean hunks and 38% of bug inducing hunks. Function calls are found in 50% of bug inducing hunks and 39% of clean hunks.

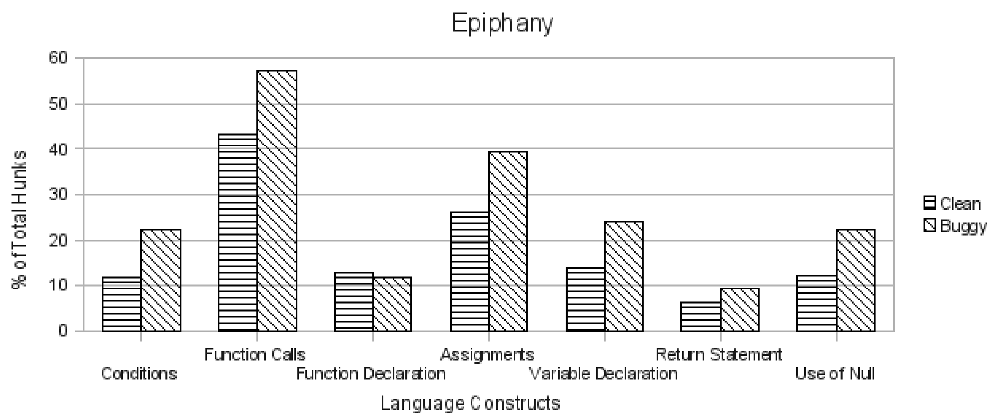


Figure 3.11: Comparison of Bug-Inducing and Clean Hunks (Epiphany)

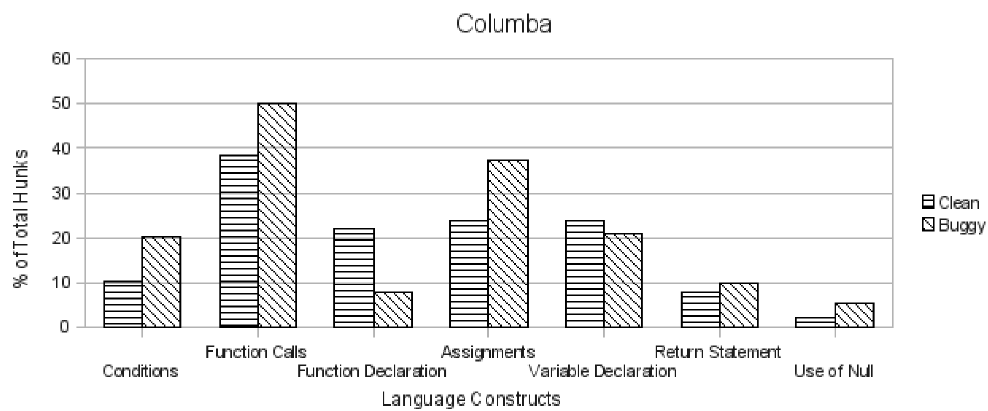


Figure 3.12: Comparison of Bug-Inducing and Clean Hunks (Columba)

Figure 3.13 shows that conditions are present in 21% bug inducing hunks and less than 14% clean hunks of Nautilus. Assignments are found in 23% clean hunks and more than 30% bug inducing hunks. About 60% bug inducing hunks contain function calls whereas in clean hunks this proportion is less than 48%. Return statement constitutes 5% of clean hunks and 9% of bug inducing hunks whereas null is used in 19% of bug inducing hunks and 12% of clean hunks.

3.8 Summary

This chapter presented an investigation into language constructs and syntax elements. In particular bug-inducing hunks were analyzed to find the frequencies of different language constructs. It is found that most of the bugs are created due

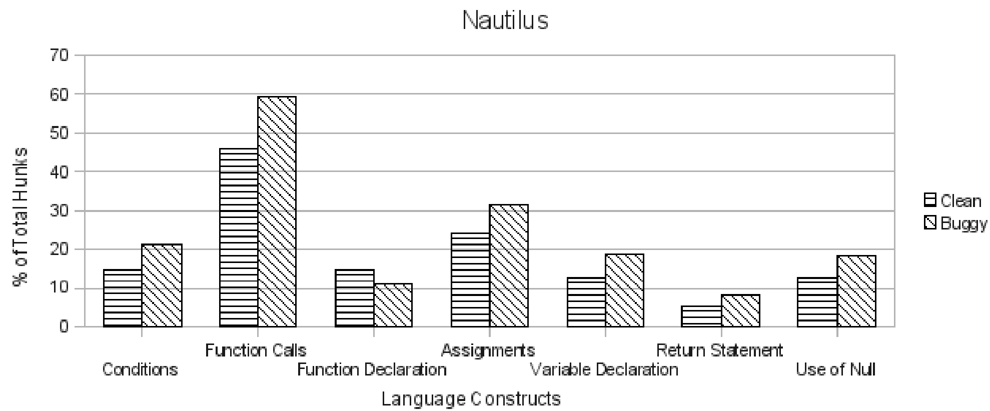


Figure 3.13: Comparison of Bug-Inducing and Clean Hunks (Nautilus)

to errors in function calls, assignments, conditions, pointers, variable declaration, function declaration and return statement. Statistical analysis showed that different projects and developers are correlated for the frequencies of bug-inducing language constructs.

These findings can be helpful during the testing and debugging process. Developers can make a priority list for testing. They can first apply testing on function calls, then on assignments, followed by conditions and so on. Applying testing resources on the frequent bug-inducing language constructs can save time and resources. Similarly if a patch of code is identified as buggy, problematic constructs can be easily identified from it. In short this study provided a means of reducing cost and improving quality of software.

Chapter 4

Language Specific Bug Patterns

During the last years there has been a growing interest in analyzing and mining the available information that is collected during all phases of the software life cycle. The used information sources are for example bug reports, which are stored in bug databases, or source code evolution information from configuration management systems (CMS). Most of the published studies focus on software quality. Researchers have tried to explore the distribution and characteristic of faults in programs [53, 13].

Most work in the empirical software engineering domain has been using open source software because of several factors. First, the source code, CMS, and bug data base information is freely available for everyone. Second, the projects like Mozilla have been developed in a distributed way. Hence, there is a larger variability in programming. Third, some of the open source programs comprise several thousands kilo lines of code (kLoc) and several thousands files. They are large enough to test available techniques in a realistic setting that would also occur in industrial practice. Because of this reasons results obtained from such projects might be generalizable which is not always the case.

Modern software projects are developed using object oriented programming languages, however a number of projects still exist in procedural languages. C language is commonly used for development of open source software projects. Different programming languages facilitate developers in writing efficient and clean code. There are some programming features specific to a particular programming language e.g; JAVA provides automatic memory management and a good exception handling mechanism. There is no multiple inheritance and no pointers in JAVA.

Programs written in different languages may have different distribution of bugs. The main goal of this chapter is to analyze whether post-release bugs are influenced by a programming language. A case study is presented to reveal whether the number of bugs per lines of code (LOC) is the same for programs written in different programming languages or not. In addition various evolution metrics are calculated and compared for different programming languages. Three common programming languages are chosen for this study, including C, C++ and JAVA.

4.1 Research Hypothesis

The research objective of this study is formulated in the following hypothesis:

Hypothesis H1: Programs written in a programming language A are more error prone in terms of more bugs per LOC than programs written in a different language B.

Hypothesis *H1* can be rejected when proving that programs written in a language A are more fault prone than programs in a language B by means of statistical inference. In this chapter hypothesis *H1* is validated up to a certain degree of significance, when applied to some languages.

When using statistical inference care has to be taken of the available information and methods. In this case proving *H1* would require to state that the mean or median of the post-release bugs per LOC of programs written in one language is really larger or smaller than the same value obtained from the programs written in the other programming language. Since, the distribution of the underlying probability variable is not known in advance, a statistical test is required that considers this case. For this purpose rank-sum test is used because it is well known to be independent on the underlying probability distribution [69].

4.2 Project Studied

For this study, Mozilla project is used because it is a heterogeneous project developed in C, C++ and JAVA. Further, it has a long development history and its information is easily available. Data is extracted from CVS and bug repositories of Mozilla using the techniques mentioned in Chapter 2. Development history of Mozilla is analyzed from 1998 to 2008. Table 4.1 shows the number of files written in different languages C, C++, and Java, as well as the lines of code for each year.

Table 4.1: Number of Source Files and Total LOC

Year	Number of Files			Total LOC (KLoc)		
	C	C++	JAVA	C	C++	JAVA
1998	1118	792	193	843	563	25
1999	1754	3365	1390	1043	1977	265
2000	2395	4958	2309	1457	2593	385
2001	2437	5207	3070	1495	2587	530
2002	2500	4762	2980	1490	2477	490
2003	2200	4845	2750	1362	2519	442
2004	2072	4776	2716	1274	2450	444
2005	2111	5141	2485	1447	2342	433
2006	2010	5183	2583	1549	2226	420
2007	2162	5016	2117	1353	2391	478
2008	2096	4704	1923	1416	2430	491

4.3 Evolution Metrics

In addition to the bug density, some other evolution metrics are calculated for each language. These metrics are used to study bug features and code evolution specific to a programming language. Bug features are studied in terms of bug density, bug frequency, bug severity, bug fix time and platform specific bug occurence. Code evolution is studied in terms of additions, deletions, code gain, number of authors and file revision frequency. Following metrics are calculated for programms written in the selected languages:

- *Authors:* The authors contributing to the file.
- *Revision frequency:* The number of revisions for each year
- *Bug frequency:* The number of corrected bugs per each year.
- *Bug density:* The number of bugs per thousand LOC (kLoc).
- *Code gain:* The sum of lines added reduced by the sum of lines removed in each file.
- *Bug fix time:* The time between fixing a bug, which is mentioned in the CVS log file, and the time where the bug was detected, which is obtained from the bug report.
- *Bug lifetime:* The time between fixing a bug and the time where the bug was introduced. The latter can be obtained from the CVS [33, 70].
- *Number of changes:* The number of changes per each file and year.

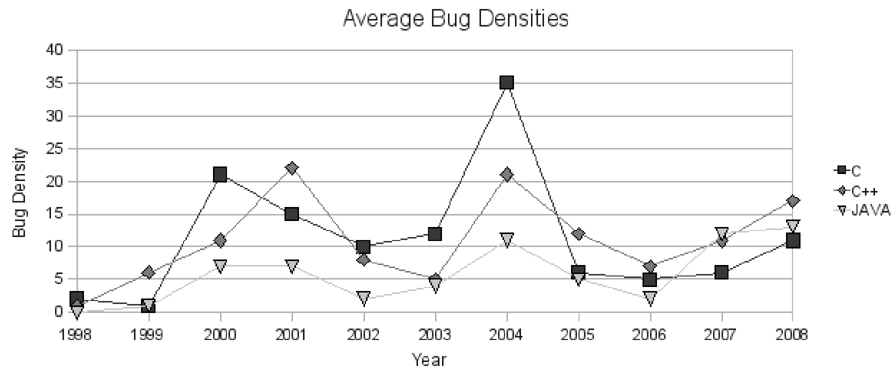


Figure 4.1: Average bug densities

4.4 Results

In this section evolution of the Mozilla project over the past years is discussed. In particular evolution metrics are compared for the three languages. Because the Mozilla project comprises C, C++, and Java files, values of different measurement categories are obtained for the three languages.

- **Average bug density:** To compute the bug density in bugs per 1000 LOC, i.e., kLOC, following equation is used:

$$bug\ density = \frac{number\ of\ bugs}{LOC} \cdot 1000$$

  The obtained results are depicted in Figure 4.1. It is evident from the figure that C++ files have higher bug densities than files written in other languages. Java files have the least bug density values except in 2007 and 2008.

- **Percentage of faulty files:** Figure 4.2 shows the percentage of faulty files in each year of development. From the figure it can be concluded that C++ files have a higher percentage of faulty files than the other languages. Java files are least likely to be faulty except in the years 1998, 2004, and 2007.

- **Average LOC per faulty file:** The results of this measure are given in Figure 4.3. On average faulty files in Java are smaller than faulty C++ files. Programs written in C have a different behavior with respect to the average number of LOC per faulty file. The size of the faulty files decreases in the initial years of Mozilla development and abruptly increase in 2004. This might be due to fixing a high number of major bugs in C files in this year.

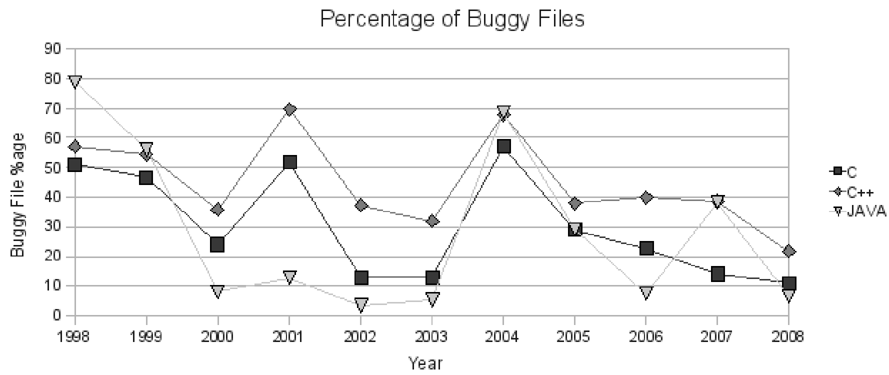


Figure 4.2: Percentage of faulty files

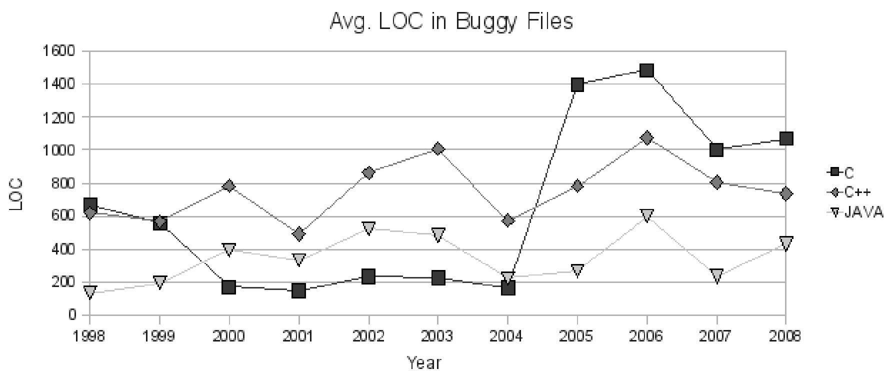


Figure 4.3: Average LOC of faulty files

- **Average revision frequencies:** Figure 4.4 shows the revision frequencies over the years. Java files show a stable behavior having a low revision frequency with exceptions in 2003 and 2006. In these years Java files have a higher revision frequency. C++ files have a higher revision frequency than the other languages. C files have revision frequency in-between C++ and Java with one exception in 2006 where C files have the highest average revision frequency.

- **Average code gain per file:** The code gain describes the increase of size of a file and is an indicator of its stability. The average code gain for the files of the Mozilla project is shown in Figure 4.5. It can be seen that Java files are more or less stable in growth whereas C++ files show a continuous decline in average code gain. C files show a mixed behavior with a high rise in code gain in 2006, which may be due to the high number of bug fixes.

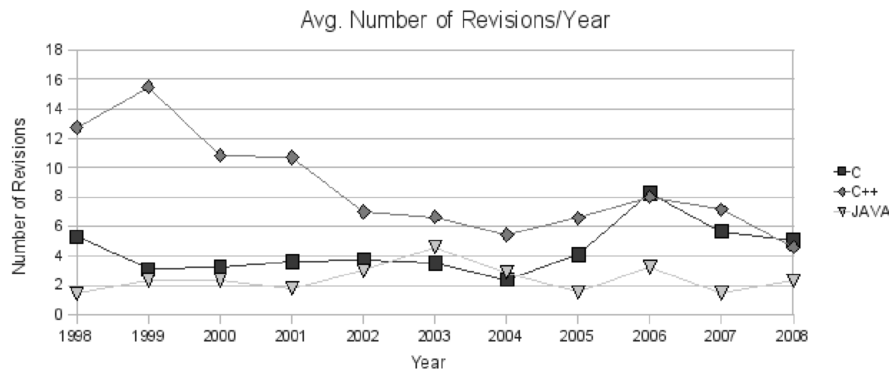


Figure 4.4: Average revision frequency

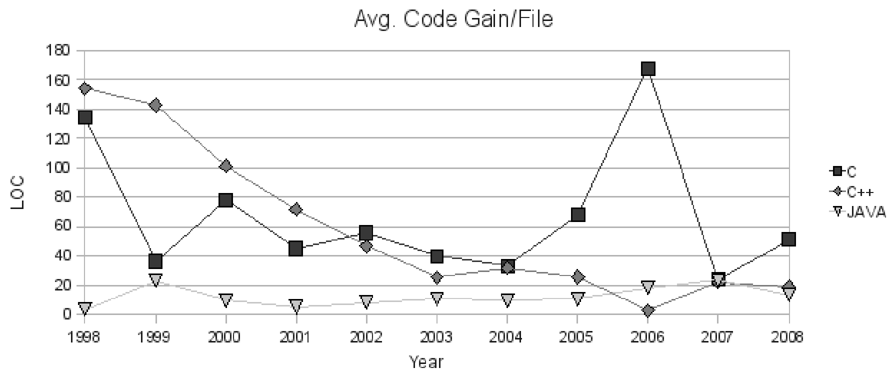


Figure 4.5: Average code gain per file

- **Bug severity distribution:** Beside the number of bugs someone is also interested in the severity of bugs and its distribution. Figure 4.6 shows the bug distribution according to severity levels. All three languages contributed a major fraction of normal bugs. Java takes the lead when considering trivial and major bugs. Most of the bugs due to enhancements are made in Java files followed by C and C++ files respectively. Most of the blocker bugs occurred in C files followed by C++ files. C++ files have the largest number of critical bugs followed by C files. From this distribution we might conclude that C and C++ are used as the programming language of choice in the kernel of Mozilla. Hence, critical or blocking bugs are created by C and C++ files.

- **Average bug lifetime:** Figure 4.7 shows the average bug lifetime for each bug severity level. It can be seen that bugs due to enhancements took more time to be fixed for C++ files. Minor bugs to be fixed took more time when

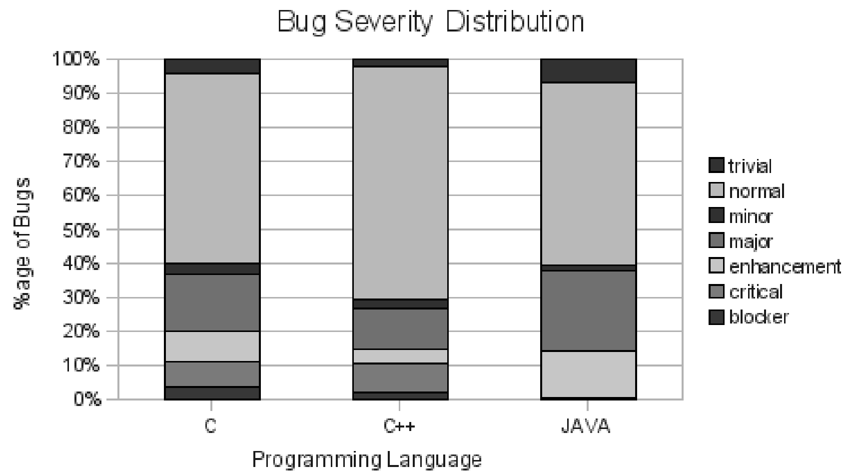


Figure 4.6: Bug severity distribution

Java was used. If we have a look at Figure 4.11 and Figure 4.10 we see that a large number of additions and deletions are made in Java files to fix minor bugs. Blocker and trivial bugs took more time to be fixed in Java files when compared with C and C++ files. Hence, what we see is that different languages have a different bug lifetime for bugs of different severity.

From the bug severity distribution and the knowledge of the number of days to fix a bug, average bug lifetime can be computed for the different languages as follows:

$$bug\ lifetime = \sum_{bug\ severity\ x} p(x) \cdot fix\ time(x)$$

where $p(x)$ denotes the probability of a bug severity, which follows from the bug severity distribution. and $fix\ time(x)$ is the average number of days necessary to fix a bug.

For the Mozilla project average bug lifetime is 175 days for C files, 192 days for C++ files, and 333 days for Java files. From this follows that bugs remain almost twice as long in the source code of Java files. This result is in line with the previous result where bugs in C and C++ files also contribute to the class of blocking and critical bugs, which have to be corrected first.

- **Average code additions:** Figure 4.8 shows a declining trend of code additions in case of C++ files. Whereas in case of C there is a decline in the first year, a stable rate for the following 5 years, and a peak in 2006 followed by a fall. Java files are almost stable with two peaks in 2003 and 2006.

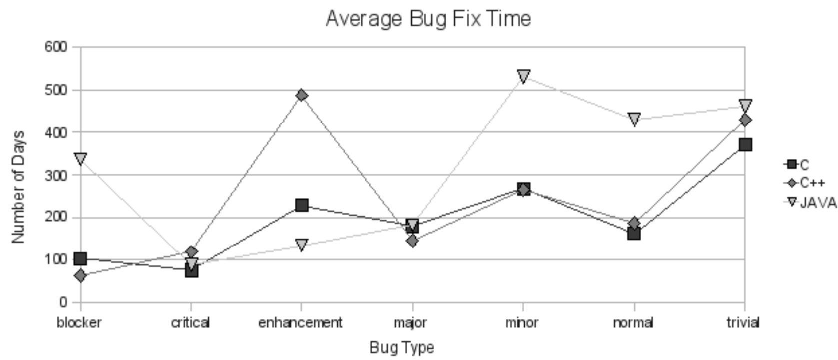


Figure 4.7: Average bug lifetime

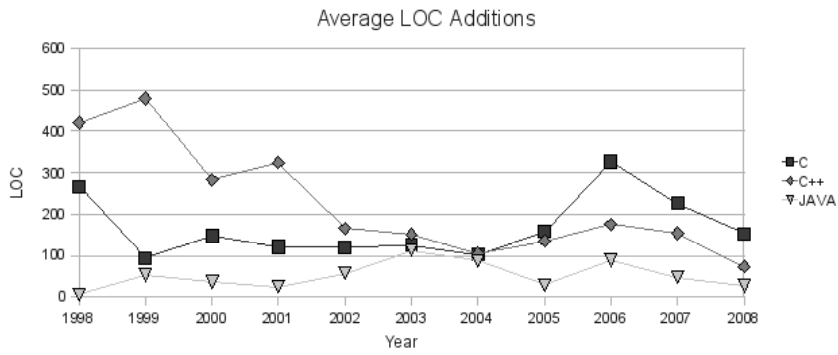


Figure 4.8: Average code additions

- **Average code deletions:** Code deletions have almost the same pattern across the time line as code additions. However deletions are less in number than additions as shown in Figure 4.9.

- **Average code deletions per bug fix:** Blocker and critical bugs involved more deletions in C++ followed by C and Java. However enhancements, major, normal and trivial bugs involved more deletions in C files followed by C++. Minor bugs involved highest deletions of all bugs and these were in Java files as shown in Figure 4.10.

- **Average code additions per bug fix:** Code additions have almost the same trend as code deletions. However additions are larger in number than deletions as shown in Figure 4.11.

- **Average number of change deltas:** In the initial years of development C++ files have higher number of change deltas. This number decreases continuously in the following years. C files have lower number of change

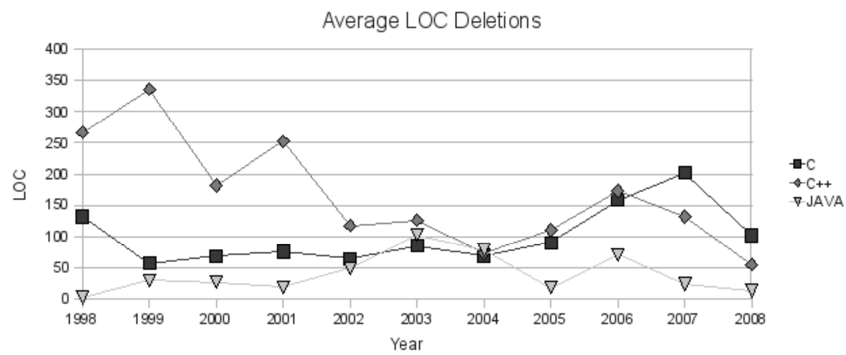


Figure 4.9: Average code deletions

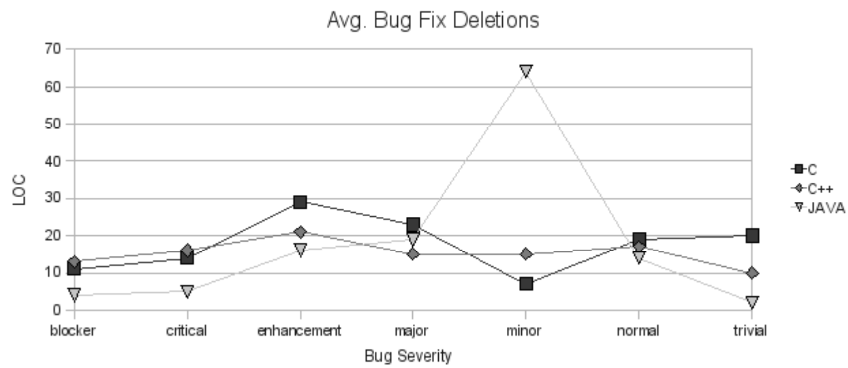


Figure 4.10: Average Code Deletions / Bug Fix

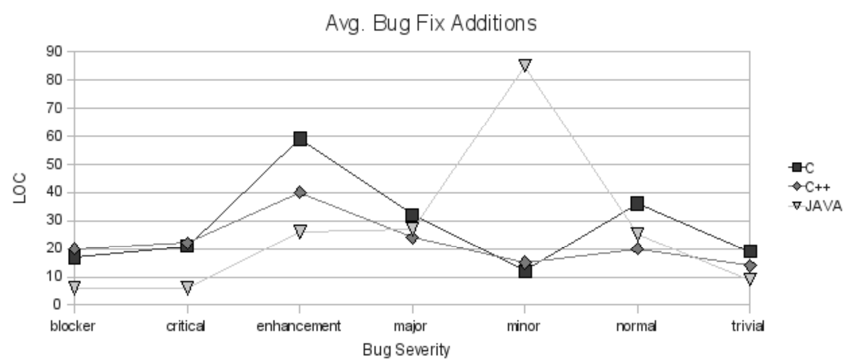


Figure 4.11: Average code additions per bug fix

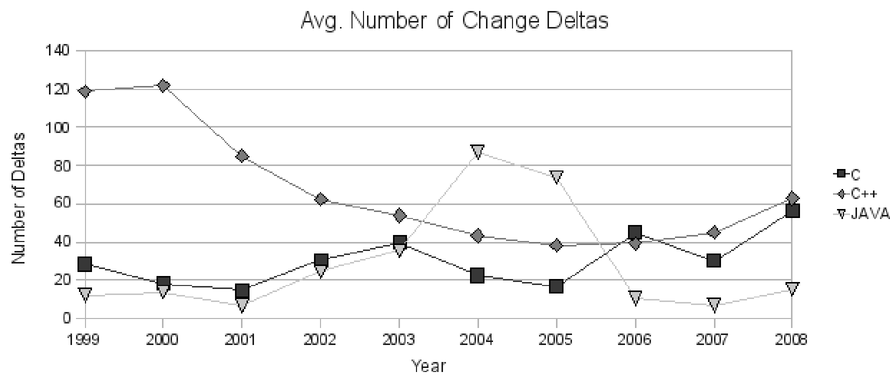


Figure 4.12: Average number of changes

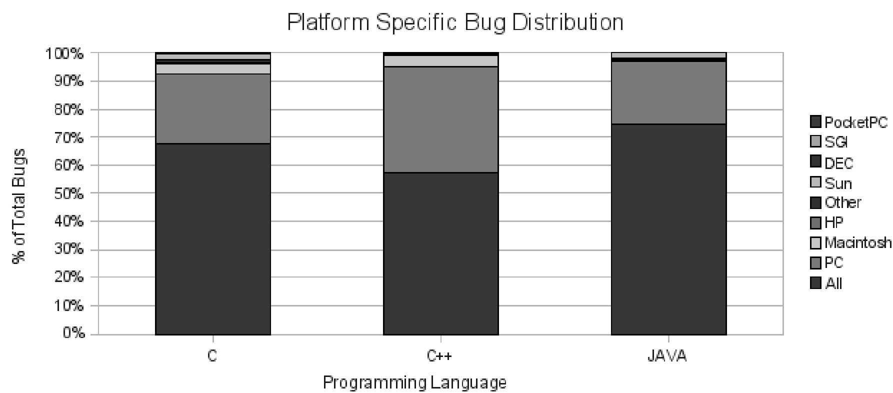


Figure 4.13: Distribution of bugs on different platforms

deltas but the pattern is different from C++ files, with ups and downs in the entire development period. Java files have very low number of change deltas with an exception in 2004 and 2005 as shown in Figure 4.12.

- **Platform specific bugs distribution:** Most of the bugs generated by three languages are reported on all platforms. However a major proportion of the bugs reported on PC and Macintosh are related to C++ files whereas majority of the bugs reported on Sun are related to C and Java. Figure 4.13 depicts different platforms on which programs written in the three languages caused failures.

- **Operating System specific bugs distribution:** A large proportion of the bugs in three languages is reported on all operating systems. However C++ is on top in the number of bugs reported on Linux and Windows followed by C language. Java files have very few bugs reported on Macintosh while C and C++ have an equal proportion of bugs reported on Macintosh.

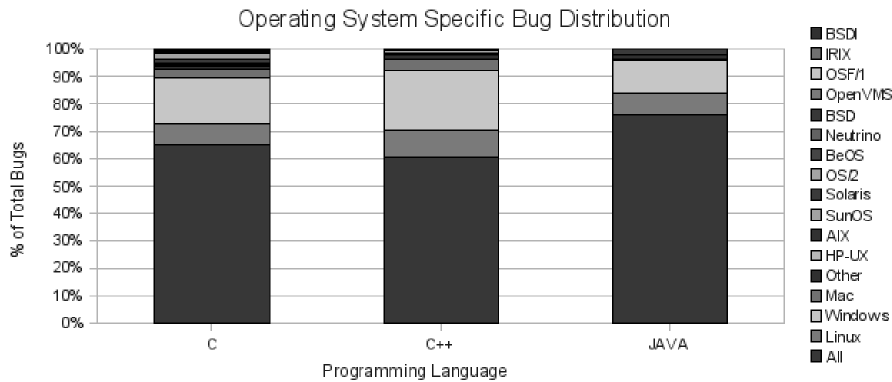


Figure 4.14: Distribution of bugs on different operating systems

Figure 4.14 depicts the types of operating systems and the proportion of bugs generated on these systems by programs of different languages.

The obtained results show that the evolution metrics have different patterns for Java, C, and C++ files in the Mozilla project. This might be due to the specific project. However, at least the results of the bug density should be generalizable because of the large number of available source files and involved programmers. In the next section, it is statistically proved that the number of bugs to be expected is influenced by the used programming language.

4.5 Proving hypothesis *H1*

In order to test hypothesis *H1* for the languages C, C++, and Java, hypothesis testing (a methodology from probability theory to draw static inference from available data under given assumptions) is used. Hypothesis testing is closely related to the procedure of interval estimation [69]. In both cases a conclusion can be drawn, which is correct for the given data set, the used statistic and probability distribution, and the desired level of significance usually denoted by α. In hypothesis testing a hypothesis H_0 is going to be proven. If the probability that the given data set $X_1, \ldots, X_n$ under the test statistic T falls within an area A, which is provided by the hypothesis H_0, is equal or larger than $1 - \alpha$, the hypothesis H_0 can be accepted. Otherwise, H_0 is said to be rejected because the observations differ significantly from the expectations.

To prove the influence of a programming language on the number of post-release bugs per LOC, we have the number of bugs and the size of the files where the bugs have been fixed. There might be remaining bugs in the files, however, since every file regardless of the used programming language is used in the same program and assuming that they are all used during program execution, there is an equal probability of detecting a bug. Hence, the probability that a bug goes

undetected in one file is equivalent for all files with no exception regarding one programming language used. As a consequence for each language

$$H_0 : f_X(x) = f_Y(x) \text{ versus } H_1 : f_X(x) = f_Y(x + c)$$

where c is a positive constant. These tests are also referred to as tests for the equality of two population medians, which is fine in this case. If we know that the median of the bugs per LOC is lower for Java programs than for C++ programs, hypothesis *H1* can be accepted for those languages.

The following rank-sum is one test for comparing two population means. In this case two independent random samples $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ are assumed. In the first step the samples are combined and ranked accordingly to increasing values. Hence, an ordered collection of size $n+m$ is obtained. Then each resulting element is assigned a rank r from 1 to $n + m$. The statistic that can be used to compare the two means is defined as follows:

$$W = \sum_{i=1}^{m} r(y_i)$$

Hence, in this case only the elements, which belong to the random sample $y_1, \ldots, y_m$ are considered. Using combinatorial theory a probability function for statistic W can be computed, and the significance level α is determined by:

$$P(W \geq w|H_0) \leq \alpha$$

Knowing the equivalence $P(W \geq w|H_0) = 1 - P(W < w|H_0)$, following inequality is determined, which must hold in order to accept H_0:

$$P(W < w|H_0) > 1 - \alpha$$

In this special case where both n and m are larger than 10, W can be approximated with a normal distribution. In this case the mean and the variance are given by:

$$\mu = E[W] = \frac{n(n + m + 1)}{2}$$

$$\sigma^2 = Var[W] = \frac{nm(n + m + 1)}{12}$$

Assuming a significance level $\alpha = 0.01$ we are able to obtain a value $w = 2.33$ if W is a Standard Normal Random Variable. Since the statistic W in general is not Standard Normal we have to standardize it using μ and σ. For values of $W(x_1, \ldots, x_n, y_1, \ldots, y_m)$ that are smaller than $2.33\sigma + \mu$, we are able to accept H_0 at the significance level of 0.01. Note that in this case the confidence in the decision is 99 percent. Alternatively, we can compute a value $Z = \frac{W(x_1,\ldots,x_n,y_1,\ldots,y_m)-\mu}{\sigma}$. If $Z > 2.33$ we accept H_0, and otherwise we reject it.

Hypothesis	Sum of ranks W	μ	σ^2	Z	Decision
$H1_0^1$	582,319,897	610,240,013	1,519,019	-18,38	reject
$H1_0^2$	747,866,055	940,156,771	2,761,791	-69,63	reject
$H1_0^3$	682,409,176	809,540,221	2,562,772	-49,61	reject

Table 4.2: Results of the rank-sum test

In the following rank-sum-test is used for testing three instances of hypothesis *H1* using the available data sets obtained from the Mozilla project:

$$H1_0^1 : f_{Java}(x) = f_C(x) \text{ versus } H1_1^1 : f_{Java}(x) = f_C(x + c)$$

$$H1_0^2 : f_{Java}(x) = f_{C++}(x) \text{ versus } H1_1^2 : f_{Java}(x) = f_{C++}(x + c)$$

$$H1_0^3 : f_C(x) = f_{C++}(x) \text{ versus } H1_1^3 : f_C(x) = f_{C++}(x + c)$$

The size of the samples for each programming language is given as follows:

Language	Sample size
Java	25,387
C++	48,074
C	22,687

Note that in this case every revision of every source file is counted as one sample. Using this information and the samples, results given in Table 4.2 can be computed.

From Table 4.2 following results can be concluded:

- The first hypothesis must be rejected with confidence 0.99. From this follows that we have to accept the alternative hypothesis that states Java programs as less error-prone than C programs.
- The second hypothesis must also be rejected. Hence, again Java programs are less error prone than C++ programs.
- The third hypothesis has to be rejected as well. It can be concluded that C files are less error prone than C++ files.

The bug density distributions given in Figure 4.15, 4.16 and 4.17 also justify the results of the rank-sum test.

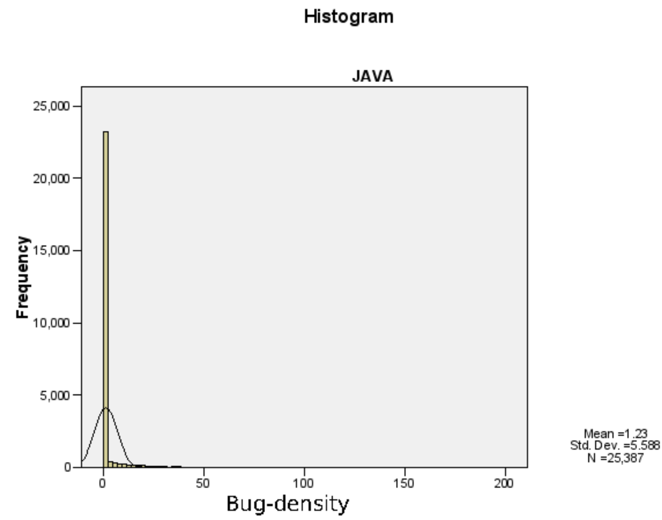


Figure 4.15: The bug density distribution of files written in Java

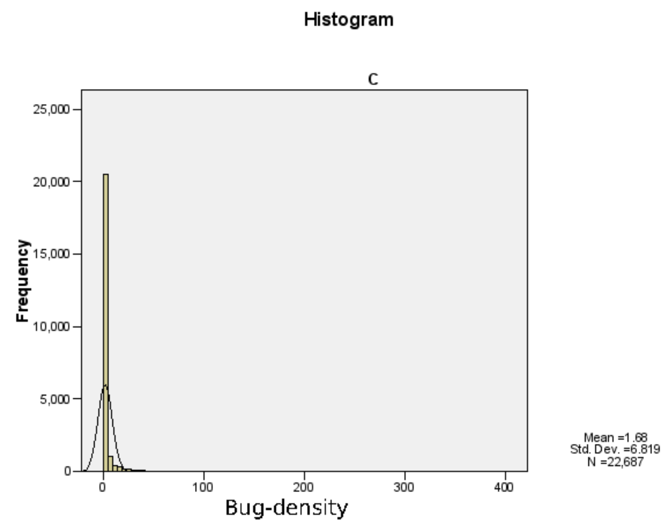


Figure 4.16: The bug density distribution of files written in C

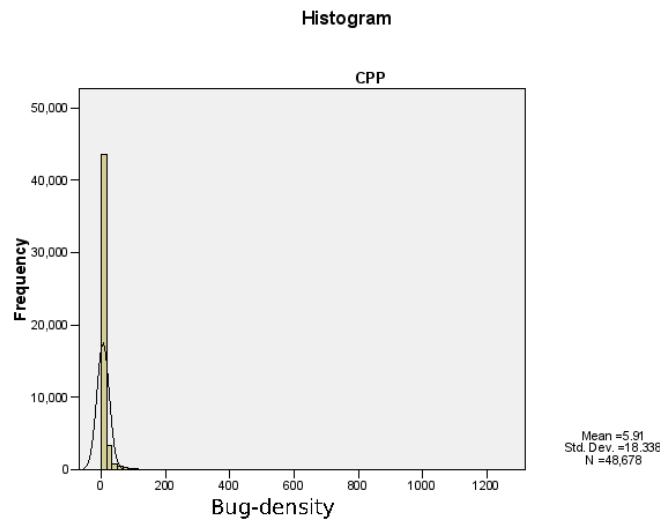


Figure 4.17: The bug density distribution of files written in C++

4.6 Threats to Validity

There are certain threats to the validity of this study. Among some of these are:

- Only one project is selected for this study, so the error patterns may be resulted from the Mozilla community rather than the programming languages.

- Although Mozilla is a heterogeneous project, the choice of programming languages for this study may be biased to a specific problem. So it is possible that the results reflect the problem rather than the programming language itself.

- No consideration is made for the features implemented in different languages. The nature of the functionality implemented in one language may have an impact on the various metrics than just the language.

- JAVA is a complete development environment, so results may be biased to the development methodology.

- There may be changes in the pool of qualified programmers for specific languages over 11 years of Mozilla development.

- There are changes in the tool support for specific languages over the years.

4.7 Summary

In this chapter empirical results obtained from 11 years of development of the open source software project Mozilla are presented. Moreover, statistical findings obtained from the development history of Mozilla are discussed. The main focus of this study is post-release bugs. In particular the hypothesis is tested whether the number of post-release bugs are influenced by the used programming language. The Mozilla project comprises source code written in Java, C, and C++ and is therefore the right project to look for in order to test the hypothesis.

In summary, this chapter has the following contributions:

- It is shown that bug lifetime is about twice as long for Java than for C and C++.
- The programming language has an influence on the number of bugs, at least for the Mozilla project. It is statistically proved that Java programs are less error prone than C or C++ programs, and C programs are less error prone than C++ programs within same project.

Although, the findings might not be generalizable they indicate a connection between post-release bugs and programming languages.

Chapter 5

Hunk Classification

Making changes to software is a crucial task during different phases of software evolution. Changes are required to add new features, to fix the bugs, to improve performance or to restructure the code for easy maintenance. These changes are implemented by adding, modifying or deleting the source code in different files of software.

A file can be changed at one or more places, called deltas or hunks. These hunks of source code which are added either newly or after modifications may introduce bugs and result in failures later on. Each hunk has a likelihood of being buggy or bug-free. This chapter describes a technique for predicting the probability of a hunk being buggy or bug-free. Software engineers and researchers face the challenge of reducing bugs to improve the quality of software. A lot of research has been carried out on bug prediction using different approaches and at different levels of granularity. Most of the researchers have used code metrics as predictors of bugs [29, 40, 52, 55, 15, 14], while others have used process metrics as predictors of bugs [27, 35, 64].

Previous research was focused on different levels of granularity such as modules, files, classes and methods. Some researchers predicted the number of faults for modules or files [52, 55], while others focused on individual classes and methods [29, 56].

Change management is an important activity in software maintenance. Changes are made to the source code as software evolves. In the past, researchers have used different change properties to predict the failure probability of changes. Researchers have shown that change properties such as size, duration, diffusion, developer expertise and type of change have strong impact on the risk of failure [48].

Features extracted from complete source code of files, change metadata and complexity metrics can be used to classify changes as clean or buggy [34]. We

Table 5.1: Statistics of Projects

Project	# of Developers	# of Revisions	# of Hunks
Apache HTTP 1.3	54	7,246	17,287
Columba	8	2,471	2,694
Eclipse JDT	17	58,565	215,824
Epiphany	52	5,217	9,035
Evolution	134	20,709	40,450
Mozilla	833	325,920	1,382,747
Nautilus	131	11,104	29,303
PostgreSQL	25	54,012	466,106

further narrow down the problem of change classification to individual units of a change, the hunks. We classify individual hunks as buggy or bug-free.

We have defined a set of hunk metrics and constructed models for hunk classification using these metrics as predictors. We used logistic regression and Random Forests to construct hunk classification models.
Kim *et al.* [34] conducted a similar study to classify software changes as clean or buggy, but our research objectives are different and go a step forward. While Kim *et al.* classified individual changes and used features extracted from complete source code, change meta data, log messages, file names and file complexity metrics, we classify individual hunks, which is a unit of change, and use only the hunk metrics. Our approach is simple and works at the smallest level of granularity.

5.1 The Approach

This chapter provides an overview of calculation of hunk metrics, labeling of hunks, preparation of data for training, hunk classification, and evaluation of classifiers.

To evaluate our approach, we extracted the change history of 8 open source projects listed in Table 5.1. The period indicates the time span used to extract the change history. The # of revisions column indicates the number of revisions extracted and the # of hunks indicates the number of hunks extracted. The # of developers indicates the number of developers involved in making these hunks.

To construct a hunk classification model following steps are used:

Preparation of Data Set Data is prepared befor it can be fed into a classifier. Data instances are created in the following way:

- Extract hunks from 8 open source project histories using the process mentioned in Chapter 2.
- Identify the bug fix hunks for each file by using the algorithm given in Chapter 2.

- Identify bug-introducing hunks by using the pseudo code given in Chapter 2.
- Label the bug-introducing hunks as buggy and others as bug-free.
- Calculate hunk metrics for each hunk.
- Combine the set of metrics of each hunk with its label indicating buggy or bug-free hunk, to make a single instance for each hunk.

Classification After preparation of data, statistical and machine learning classifiers are trained on this data.

- Train classifiers for each project, using the labeled instances.
- Evaluate classification performance of each classifier, using the measures of accuracy, recall, precision, and F-value.

Identification of Significant Metrics Some metrics may be better predictors of bugs than others, so those metrics should be selected which produce better results.

- Individual and groups of metrics are used to construct models and their performance is evaluated.

5.2 Tools Used

The random forest algorithm implemented in WEKA [2] is used for this study. To apply logistic regression, the statistical tool R is used. Random forest is used due to its ability to quickly handle large number of input variables. Output of random forest is the mode of all outputs of individual trees, so it produces better results than other machine learning classifiers. Logistic regression is used because there are two possible predictions for a hunk, buggy or bug-free. Predictive capabilities of individual as well as combination of metrics are studied.

5.3 Hunk Metrics

Software metrics deals with the measurement of the software product and the process by which it is developed. We briefly describe the categories of software metrics used so far, followed by an introduction to hunk metrics.

Classification of Software Metrics Software metrics can be classified into two major categories, product metrics and process metrics.

- Product metrics deals with the measurements of the software product itself. These metrics include measures at various stages of software development starting from requirements to installed system. Product

Table 5.2: Measurement Types

Type of Data	Possible Operations	Description of Data
Nominal	$=, \neq$	Categories
Ordinal	$<, >$	Rankings
Interval	+,-	Differences
Ratio	/	Absolute zero

metrics may include the software design complexity, the size of the final source or object code, or the number of documentation pages produced.

- Process metrics deals with the measurements of the software development process used. These metrics may include total development time, type of methodology used, or the level of expertise of the programmers involved.

Categories of Metrics Metrics can be categorized as primitive metrics or computed metrics

- Primitive metrics can be directly measured and do not need any computations. This category may include the program size metrics observed as total lines of code, number of defects found during testing, or the total development time.
- Computed metrics cannot be directly measured and require other metrics for their computation. These metrics may include productivity metrics such as LOC produced per person-month (LOC/person-month), or quality metrics such as number of defects per thousand lines of code.

Measurement Scales for Software Metrics For statistical analysis, measured data can be classified into four basic types that are nominal, ordinal, interval, and ratio. It is important to know the type of information involved before any data collection. Software metrics should belong to these categories, for their optimum utilization in empirical studies.

Good metrics should hold capabilities to be used in the development of efficient predictor models. An ideal metrics should be capable of predicting software product or process features. Thus good metrics should be simple, precise, easy to obtain, valid and robust.

In this study following hunk metrics are considered:

- *No. of Conditions* (NOCN) is the total number of conditional statements in a hunk, such as if, else if and else statement.

- *No. of Loops* (NOL) is the total number of loops in a hunk, such as for, while and do while loop.
- *No. of Function Calls* (NOFC) is the total number of functions called in a hunk.
- *No. of Function Declarations* (NOFD) is the total number of functions declared or defined in a hunk.
- *No. of Variable Declarations* (NOV) is the total number of variables declared or defined in a hunk.
- *No. of Assignments* (NOA) is the total number of assignment statements used in a hunk.
- *No. of Logical Operators* (NOLO) is the total number of logical operators used in a hunk.
- *No. of Relational Operators* (NORO) is the total number of relational operators used in a hunk.
- *No. of Return Statements* (NORS) is the total number of return statements used in a hunk.
- *No. of Arrays* (NOAR) is the total number of array declaration or access statements used in a hunk.
- *No. of Null Statement* (NON) is the total number of times NULL is used in a hunk.
- *No. of Case Statements* (NOCS) is the total number of case statements used in a hunk.
- *No. of Break Statements* (NOB) is the total number of break statements used in a hunk.
- *No. of Classes* (NOC) is the total number of classes declared in a hunk.
- *No. of Object Instantiations* (NOO) is the total number of objects instantiated using the new operator in a hunk.
- *No. of Imports* (NOIP) is the total number of import statements used in a hunk.
- *No. of Inheritance Statements* (NOIH) is the total number of inheritance statements such as extends, implements used in a hunk.
- *No. of Exception Handlers* (NOE) is the total number of exception handlers used in a hunk.

- *No. of Throw statements* (NOTH) is the total number of throw statements used in a hunk.
- *Total Hunks* (NOH) is the total number of hunks made in a revision.
- *No. of Previous Buggy Hunks* (NOBH) is the total number of buggy hunks made in the previous revisions of a file.

5.4 Evaluation Criteria

Four measures are commonly used to assess the performance of a classifier including accuracy, precision, recall and F-Measure. Accuracy is the percentage of correctly classified instances. We explain these measures with the use of the following confusion matrix.

		Predicted	
		No	Yes
Observed	No	n_{11}	n_{12}
	Yes	n_{21}	n_{22}

We represent buggy hunks with Yes and bug-free hunks with No. Accuracy is the ratio of the correct classifications to the total number of instances. Correct classifications is the sum of actual buggy hunks classified as byggy and the actual bug-free hunks classified as bug-free. Accuracy can be calculated by the following formula:

$$Accuracy = \frac{(n_{11} + n_{22})}{n_{11} + n_{12} + n_{21} + n_{22}} * 100$$

Buggy hunk precision is the ratio of actual buggy hunks predicted as buggy to the total number of hunks predicted as buggy.

$$Buggy\ Hunk\ Precision = \frac{n_{22}}{n_{22} + n_{12}}$$

Buggy hunk recall is the ratio of actual buggy hunks predicted as buggy to the total number of actual buggy hunks.

$$Buggy\ Hunk\ Recall = \frac{n_{22}}{n_{22} + n_{21}}$$

Bug-free hunk precision is the ratio of actual bug-free hunks predicted as bug-free to the total number of hunks predicted as bug-free.

$$Bug - Free\ Hunk\ Precision = \frac{n_{11}}{n_{11} + n_{21}}$$

Bug-free hunk recall is the ratio of actual bug-free hunks predicted as bug-free to the total number of actual bug-free hunks.

$$Bug-Free\ Hunk\ Recall = \frac{n_{11}}{n_{11}+n_{12}}$$

F-Measure combines both precision and recall and is a ratio of the 2 times product of precision and recall to the sum of precision and recall.

$$F-Measure = \frac{2*Precision*Recall}{Precision+Recall}$$

5.5 Classification Techniques

Many machine learning algorithms are available to be used as classifiers.

5.5.1 Logistic Regression

Logistic regression is used when the dependent variable is a binary categorical variable and the independent variables are continuous and/or categorical [38]. Logistic regression can determine the percent of variance in dependent variable explained by the independent variables and the relative importance of independents.

Linear regression cannot work when the response variable is binary. In situations where response variable is a probability that takes values between 0 and 1, logistic regression is used. It bounds the response variable to values between 0 and 1, in contrast to linear regression which allows arbitrary large or small values.

Logistic regression assumes that the response variable follows the Logit-function shown in Figure 5.1.

To understand logit-function we should know the concept of odds. The odds of an event that occurs with probability P is defined as

$$Odds \ = \ P \ / \ (1-P) \tag{5.1}$$

Figure 5.2 depicts the odds function. We can see the odds of an event goes from 0 to infinity when the probability for that event goes from 0 to 1.

In terms of odds, the logit-function can be written as

$$logit(P) \ = \ log(odds(P)) \ = \ log(P/(1-P)) \tag{5.2}$$

If we use logit-function, we can bound values of P between 0 and 1 with a linear representation for input variable X.

$$logit(P) \ = \ \alpha \ + \ \beta \ * \ X \tag{5.3}$$

Multivariate logistic regression can be represented by the equation:

$$P(X_1, X_2, \ldots, X_n) = \frac{e^{C_0+C_1.X_{i_1}+\ldots+C_n.X_{i_n}}}{1+e^{C_0+C_1.X_{i_1}+\ldots+C_n.X_{i_n}}} \tag{5.4}$$

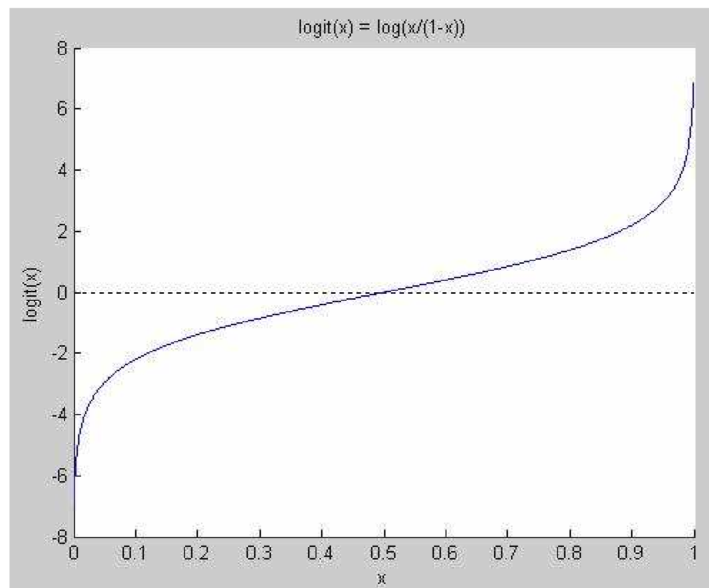


Figure 5.1: Logit Function

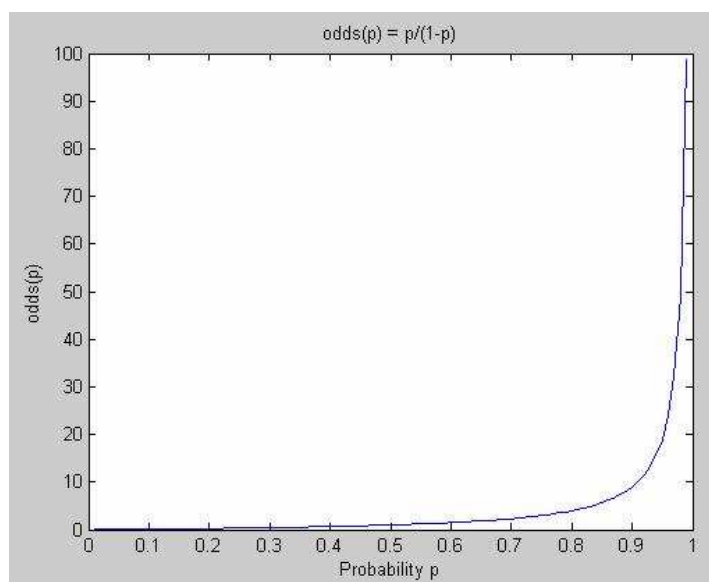


Figure 5.2: Odds Function

The X_is are the hunk metrics in our case and P is the probability of a hunk being buggy.

5.5.2 Random Forests

The Random Forest is a meta-learner comprised of many trees and operates quickly on large datasets. It uses random samples to build each tree in the forest. Attributes at each node of a tree are selected randomly and then attributes providing the highest level of learning are selected.

A detail of the working of Random Forests is out of the scope of this thesis. However a brief overview is presented here as described in [9]. Random forests use a combination of tree predictors with each tree depending on the values of a random vector sampled independently and with the same distribution for all trees in the forest. To classify a new object from an input vector, each input vector is put down each of the trees in the forest. Each tree gives a classification or votes for that class. The forest chooses the classification having the most votes among all the trees in the forest.

Each tree in the forest grows as follows:

- Suppose $\boldsymbol{N}$ is the number of cases in the training set, randomly $\boldsymbol{N}$ cases are sampled with replacement from the original data. This sample acts as training set for growing the tree.
- Suppose $\boldsymbol{M}$ is the number of input variables, a number $\boldsymbol{m} \ll \boldsymbol{M}$ is specified in such a way that $\boldsymbol{m}$ variables are selected randomly out of $\boldsymbol{M}$ at each node and the best split on these $\boldsymbol{m}$ is used to split the node. The value of $\boldsymbol{m}$ is kept constant as the tree grows.
- No pruning is applied and each tree in the forest grows to the largest extent possible.

The error rate of random forest depends on two things:

- High correlation between any two trees increases the error rate of random forest.
- Higher strength of individual trees decreases the error rate of random forest.

We used the random forest algorithm implemented in WEKA [2].

5.5.3 Principal Component Analysis (PCA)

Principal component analysis is used to identify patterns in data, and express the data in such a way as to highlight the similarities and differences among patterns. It is difficult to find patterns in high dimensional data, so PCA helps to analyze

such kind of data. PCA helps to find patterns in data and compress the data to reduce the number of dimensions, without much loss of information. To apply PCA mean (average across each dimension) is subtracted from each of the data dimensions. $\bar{X}$ is subtracted from all X values and $\bar{Y}$ is subtracted from all Y values. In this way we get a dataset having mean zero. In the next step a co-variance matrix is calculated for the data. Then eigenvectors and eigenvalues are calculated for the covariance matrix. By taking the eigenvectors of the covariance matrix, we can extract lines that characterize the data. Then the data is transformed so that it can be expressed in terms of these lines.

Eigenvectors are ordered by eigenvalues from highest to lowest, producing components in order of significance. Components with lesser significance can be ignored to reduce the data dimensions. If we have $\boldsymbol{n}$ dimensions in data and there are $\boldsymbol{n}$ calculated eigenvectors and eigenvalues, and we choose first $\boldsymbol{m}$ eigenvectors then the final data will have $\boldsymbol{m}$ dimensions. A feature vector is made by forming a matrix with the chosen eigenvectors.

$$Feature\ vector = (eig1\ eig2\ eig3\ \cdots\ eig\mathbf{n}) \tag{5.5}$$

Finally transpose of the feature vector is multiplied on the left of the transposed original data set.

$$Final\ Data = RowFeatureVector\ *\ RowDataAdjust \tag{5.6}$$

Where RowFeatureVector is the matrix with the eigenvectors in the columns transposed, and RowDataAdjust is the mean-adjusted data transposed. In this way data is represented in terms of vectors which describe patterns in the data.

Some of the hunk metrics are correlated with each other. These inter-correlations can be overcome using the principal component analysis (PCA). PCA reduces the number of dimensions without much loss of information. Principal components are extracted by using a variance maximizing rotation of the original variables. We used the extracted principal components in logistic regression.

5.5.4 Point Biserial Correlation

The point biserial correlation measures the association between a continuous variable and a binary variable [28]. It can take values between -1 and +1. Assuming X as a continuous variable and Y as categorical with values 0 and 1, point biserial correlation can be calculated using the formula

$$r = \frac{(\bar{X_1} - \bar{X_0})\sqrt{p(1-p)}}{S_x}$$

where $\bar{X_1}$ is the mean of X when Y=1 ,
$\bar{X_0}$ is the mean of X when Y=0 ,

S_x is the standard deviation of X ,
and p is the proportion of values where Y=1 .
Positive point biserial correlation indicates that large values of X are associated with Y=1 and small values of X are associated with Y=0. Point biserial correlation values greater than 0.2 are considered good.

5.6 Results

This section presents the results obtained by classifying hunks using random forests and logistic regression. Performance of individual as well as group of hunk metrics is evaluated for hunk classification. Classification accuracies are compared for random forests and logistic regression. Hunk metrics are analyzed, and those metrics are identified which can serve as better predictor of bugs.

5.6.1 Correlation between Hunk Metrics and Bugs

As a hunk can be either buggy or bug-free, point biserial correlation is calculated between each hunk metrics and the hunk type i.e buggy or bug-free. Most of the hunk metrics have positive point biserial correlation with hunk type except NOI, NOTH and NOIP having negative correlation, see Table 5.3. The majority of the correlation values are greater than 0.15, indicating that hunk metrics can discriminate between buggy and bug-free hunks. NOH has higher correlation values in all projects as compared to other metrics. It means NOH can better discriminate between buggy and bug-free hunks. NOBH has higher values for Eclipse and Mozilla as compare to other projects, the reason may be large number of revisions of these projects as compared to other projects.
Some projects have similar correlation values like Apache, Epiphany and Evolution are similar for most of the hunk metrics. Similarly Nautilus and PostgreSQL have almost similar values. It indicates the possibility of a single classification model which can be applied to different projects.

5.6.2 PCA and Logistic Regression

We applied logistic regression both with and without using PCA, but the results are almost similar in both cases. However one advantage of using PCA is that number of input variables is reduced. Logistic regression provides the probability of a hunk being buggy and the values range between 1 and 0. We used a cutoff value of 0.5 to classify hunks as buggy, it means that if $P > 0.5$, the hunk is classified as buggy and bug-free otherwise. Accuracy, precision and recall values are calculated for each project (both C and JAVA files are processed for Mozilla). The accuracy values vary from 60 percent for Nautilus to 74 percent for Mozilla. The F-Measure for buggy hunks varies from 0.11 for Mozilla to 0.61 for Nautilus

Table 5.3: Point biserial correlation between hunk metrics and hunk type

Metrics	Apache	Eclipse	Epiphany	Evolution	Mozilla	Nautilus	PostgreSQL
NOCN	0.32	0.23	0.25	0.24	0.20	0.17	0.22
NOL	0.25	0.09	0.23	0.30	0.16	0.14	0.18
NOA	0.26	0.12	0.25	0.27	0.15	0.17	0.19
NOFC	0.36	0.16	0.28	0.28	0.15	0.25	0.22
NOFD	0.16	0.12	0.23	0.25	0.13	0.23	0.19
NOV	0.18	0.09	0.25	0.26	0.09	0.18	0.18
NOP	0.27	—	0.27	0.28	0.19	0.24	0.21
NOLO	0.31	0.15	0.22	0.22	0.15	0.12	0.18
NORO	0.28	0.13	0.23	0.16	0.11	0.11	0.15
NORS	0.27	0.02	0.14	0.22	0.17	0.14	0.22
NON	0.32	0.15	0.26	0.21	0.20	0.15	0.16
NOI	-0.17	—	0.14	-0.03	-0.03	-0.02	-0.11
NOD	0.04	—	0.16	0.06	0.03	0.11	0.07
NOS	0.20	—	0.12	0.28	0.11	0.16	0.12
NOAS	0.02	—	0.01	0.01	-0.17	0.01	0.15
NOAR	0.25	0.08	0.21	0.16	0.16	0.06	0.14
NOCS	0.25	0.31	0.18	0.16	0.13	0.19	0.04
NOG	0.36	—	0.22	0.23	0.14	0.26	0.13
NOB	0.29	0.16	0.23	0.22	0.21	0.20	0.15
NOE	—	0.08	—	—	0.14	—	—
NOC	—	0.09	—	—	-0.01	—	—
NOO	—	0.04	—	—	0.05	—	—
NOTH	—	-0.03	—	—	0.09	—	—
NOIP	—	-0.01	—	—	-0.31	—	—
NOIH	—	0.15	—	—	-0.09	—	—
NOH	0.33	0.28	0.28	0.34	0.36	0.22	0.37
NOBH	0.10	0.61	0.05	0.11	0.27	0.05	0.06

and the F-Measure for bug-free hunks varies from 0.58 for Nautilus to 0.85 for Mozilla. Precision and Recall values are lower for buggy hunk as compare to bug-free hunks, see Table 5.5. We can adjust precision and recall values for buggy and bug-free hunks by changing the cutoff value. If we use cutoff value of 0.3, the precision and recall for buggy hunks is improved.

Application of PCA has not improved the results, see Table 5.4. The reason is that in majority of the hunk instances most of the hunk metrics are 0. Although there is correlation between hunk metrics but the correlation values are not so high.

Regression analysis have shown that NOCN, NOA, NOFC, NORS, NOBH and NOH are significant predictors of buggy hunks at significance level 1 % in most of the projects, see Table 5.6 and 5.7 . NOH are found significant for classifying the hunks as buggy or bug-free in all projects. NORO, NON, NOAR, NOB, and

Table 5.4: Precision P, Recall R and Accuracy A using LR with PCA

Project	A	Buggy Hunk			Bug-Free Hunk		
		P	R	F1	P	R	F1
Apache	0.65	0.68	0.36	0.47	0.88	0.64	0.74
Eclipse	0.69	0.73	0.17	0.28	0.97	0.69	0.80
Epiphany	0.68	0.63	0.20	0.30	0.94	0.69	0.79
Evolution	0.67	0.65	0.24	0.35	0.92	0.67	0.78
Mozilla-C	0.74	0.55	0.05	0.09	0.99	0.75	0.85
Mozilla-J	0.69	0.72	0.33	0.46	0.92	0.68	0.78
Nautilus	0.60	0.62	0.66	0.64	0.53	0.57	0.55
PostgreSQL	0.61	0.66	0.40	0.50	0.84	0.62	0.71

Table 5.5: Precision P, Recall R and Accuracy A using LR without PCA

Project	A	Buggy Hunk			Bug-Free Hunk		
		P	R	F1	P	R	F1
Apache	0.66	0.69	0.37	0.48	0.87	0.65	0.74
Eclipse	0.69	0.74	0.17	0.28	0.97	0.69	0.81
Epiphany	0.66	0.57	0.09	0.15	0.96	0.67	0.79
Evolution	0.66	0.65	0.19	0.30	0.94	0.66	0.77
Mozilla-C	0.74	0.56	0.06	0.11	0.98	0.75	0.85
Mozilla-J	0.69	0.73	0.33	0.45	0.92	0.68	0.78
Nautilus	0.60	0.64	0.60	0.61	0.60	0.56	0.58
PostgreSQL	0.62	0.67	0.42	0.52	0.83	0.61	0.70

NOFD are also significant in half of the projects. The set of significant hunk metrics is different in all projects with one exception, that is NOH.

5.6.3 Random Forests

Random forests have produced the most accurate results. We used 10-fold cross validation to build the classification model. In 10-fold cross validation the data is broken down into 10 sets of size n/10. The classifier is trained on 9 data sets and tested on 1 data set. This procedure is repeated 10 times and a mean accuracy is taken [72]. The accuracy values produced by our model vary from 74 percent for Epiphany to 87 percent for Eclipse, see Table 5.8. The F-measure for buggy hunks varies from 0.57 for Epiphany to 0.81 for Eclipse and the F-measure for bug-free hunks varies from 0.75 for Nautilus to 0.91 for Eclipse and Mozilla. Precision values for buggy hunks are between 66% and 84%, and the recall values for buggy hunks are between 51% and 78%.

Table 5.6: Results of Multivariate Logistic Regression (a)

Metrics	Apache		Epiphany		Evolution		Nautilus	
	Coeff.	p-value	Coeff.	p-value	Coeff.	p-value	Coeff.	p-value
constant	-0.87	**0.000**	-1.21	**0.000**	-1.11	**0.000**	-0.15	**0.000**
NOP	0.02	0.01	0.04	0.01	0.02	0.003	0.02	0.02
NOCN	0.07	**0.000**	0.07	0.04	0.02	0.15	0.02	0.22
NOL	-0.03	0.52	-0.01	0.94	0.01	0.8	-0.03	0.62
NOLO	0.03	0.05	-0.01	0.76	-0.06	**0.000**	-0.11	**0.000**
NORO	-0.06	0.002	0.12	0.003	0.07	**0.000**	0.03	0.15
NOA	-0.11	**0.000**	-0.07	0.02	0.01	0.26	-0.08	**0.000**
NOFC	0.12	**0.000**	0.07	0.001	0.07	**0.000**	0.09	**0.000**
NORS	-0.02	0.58	-0.47	**0.000**	-0.03	0.2	-0.14	**0.000**
NON	0.07	0.01	0.06	0.04	-0.01	0.26	-0.09	**0.000**
NOS	0.03	0.79	-0.05	0.87	0.08	0.22	-0.13	0.38
NOAR	0.04	0.08	-0.07	0.37	-0.07	0.002	-0.06	0.03
NOCS	0.16	0.03	-0.08	0.46	-0.1	0.001	0.14	0.02
NOG	1.04	0.01	-0.16	0.59	0.22	0.1	0.82	0.02
NOB	-0.36	**0.000**	0.21	0.23	0.07	0.14	0.08	0.43
NOV	-0.03	0.2	0.05	0.06	0.02	0.06	-0.01	0.34
NOFD	-0.08	0.002	0.16	**0.000**	0.09	**0.000**	0.14	**0.000**
NOBH	0	0.91	0	0	0.001	0.26	0.001	**0.000**
NOH	0.02	**0.000**	0.05	**0.000**	0.03	**0.000**	0.01	**0.000**

5.6.4 Comparison of Logistic Regression and Random Forests

Random forests have produced better results as compared to logistic regression. Accuracies obtained by training and applying both models are shown in Figure 5.3. Maximum and minimum accuracies obtained by applying random forests are 87% and 74% respectively. For Eclipse, Mozilla and PostgreSQL it has classified more than 80% hunks accurately. Application of logistic regression produces maximum and minimum accuracies of 74% and 60% respectively. In most of the projects, logistic regression can classify less than 70% hunks accurately.

Figure 5.4 shows the buggy hunk precision obtained by training and applying both models. Again random forest has out classed logistic regression and it produces maximum and minimum buggy hunk precision of 84% and 66% respectively. It produces more than 80% buggy hunk precision for Eclipse, Mozilla and PostgreSQL. Maximum and minimum buggy hunk precision obtained by applying logistic regression is 74% and 56% respectively. Using logistic regression, buggy hunk precision falls between 60% and 70% for most of the projects.

Buggy hunk recall obtained by applying both models is shown in Figure 5.5. Logistic regression has produced very poor recall. However in more than half projects random forest has produced more than 70% buggy hunk recall. Maximum and minimum recall obtained by applying random forests is 78% and 51%

Table 5.7: Results of Multivariate Logistic Regression (b)

Metrics	PostgreSQL		Eclipse		Mozilla	
	Coeff.	p-value	Coeff.	p-value	Coeff.	p-value
constant	-0.72	**0.000**	-1	**0.000**	-0.99	**0.000**
NOP	0.08	**0.000**	—	—	0.04	**0.000**
NOCN	0.15	**0.000**	0.04	**0.000**	0.07	**0.000**
NOL	0.1	0.01	-0.01	0.37	-0.09	0.067
NOLO	0.03	0.1	0.01	0.09	0.04	0.021
NORO	-0.16	**0.000**	-0.01	0.1	-0.1	**0.000**
NOA	0.04	**0.000**	-0.02	**0.000**	0.01	0.157
NOFC	0.06	**0.000**	0.04	**0.000**	0.05	**0.000**
NORS	0.3	**0.000**	-0.06	**0.000**	0.05	**0.000**
NON	-0.09	**0.000**	0.19	**0.000**	-0.06	0.109
NOS	0.14	0.02	—	—	—	—
NOAR	-0.14	**0.000**	-0.04	**0.000**	0.12	**0.000**
NOCS	-0.24	**0.000**	0.02	0.002	-0.03	0.041
NOG	-1.08	**0.000**	—	—	—	—
NOB	0.28	**0.000**	-0.14	**0.000**	-0.01	0.625
NOV	0.07	**0.000**	0.01	0.03	0.02	0.044
NOFD	0	0.68	0	0.85	0.01	0.610
NOE	—	—	-0.09	**0.000**	0.03	0.382
NOO	—	—	0.01	0.4	-0.15	**0.000**
NOC	—	—	-0.02	0.56	0.3	0.002
NOTH	—	—	-0.1	**0.000**	-0.09	0.001
NOIP	—	—	0	0.78	-0.5	**0.000**
NOIH	—	—	0.15	0.001	-0.4	**0.000**
NOBH	0	**0.000**	0	**0.000**	0	**0.000**
NOH	0	**0.000**	0	**0.000**	0.01	**0.000**

Table 5.8: Precision P, Recall R and Accuracy A using random forests

Project	A	Buggy Hunk			Bug-Free Hunk		
		P	R	F1	P	R	F1
Apache	0.76	0.75	0.65	0.70	0.76	0.84	0.80
Eclipse	0.87	0.84	0.78	0.81	0.89	0.92	0.91
Epiphany	0.74	0.66	0.51	0.57	0.77	0.86	0.81
Evolution	0.75	0.70	0.53	0.63	0.77	0.85	0.81
Mozilla-C	0.86	0.81	0.62	0.70	0.88	0.95	0.91
Mozilla-J	0.84	0.83	0.76	0.79	0.85	0.90	0.87
Nautilus	0.77	0.79	0.78	0.78	0.75	0.76	0.75
PostgreSQL	0.83	0.81	0.72	0.76	0.84	0.89	0.86

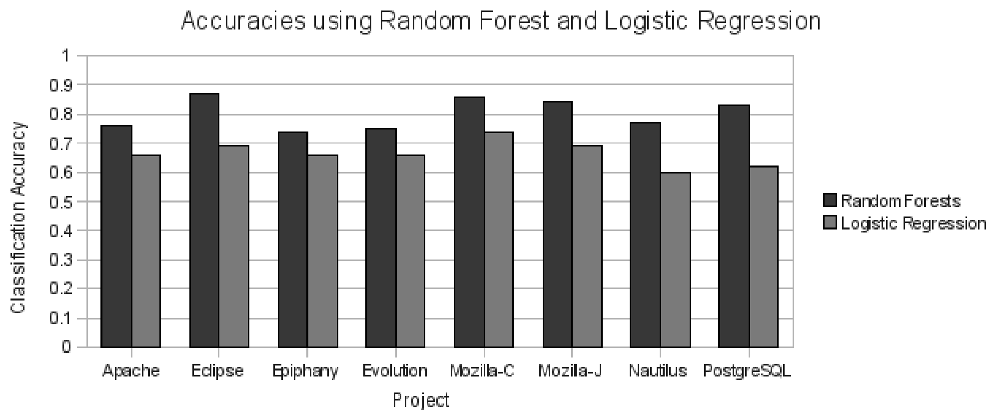


Figure 5.3: Accuracies using Random Forest and Logistic Regression

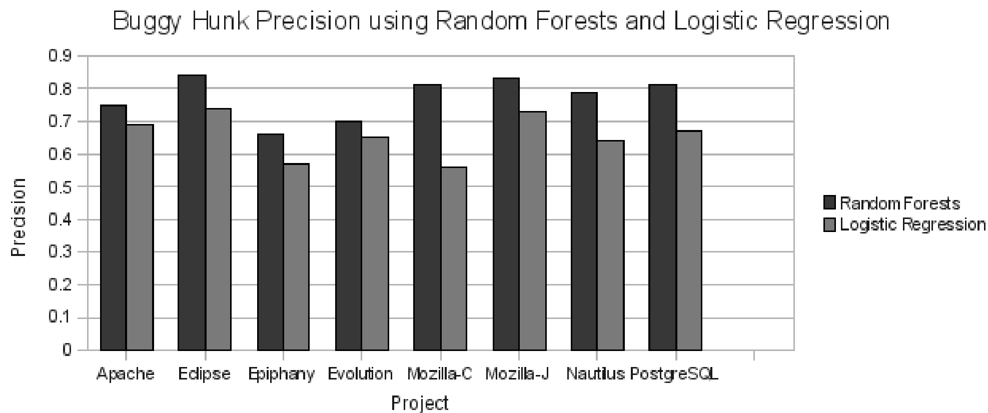


Figure 5.4: Buggy Hunk Precision using Random Forest and Logistic Regression

respectively. Buggy hunk recall obtained by applying logistic regression is less than 40% for most of the projects. It produces maximum and minimum buggy hunk recall of 60% and 6% respectively.

5.6.5 Performance of Individual Metrics

To evaluate the performance of individual metrics, we used single hunk metric as the independent variable and presence or absence of bug as the dependent variable. Our objective was to evaluate each metric separately as predictor of bugs. Most of the code related hunk metrics have produced similar results. Hunks may differ in their code contents, so different metrics may classify the same hunk differently. However overall accuracies are almost similar for code related metrics, see Table 5.9 and 5.10. Two hunk metrics have produced better results as compared to other

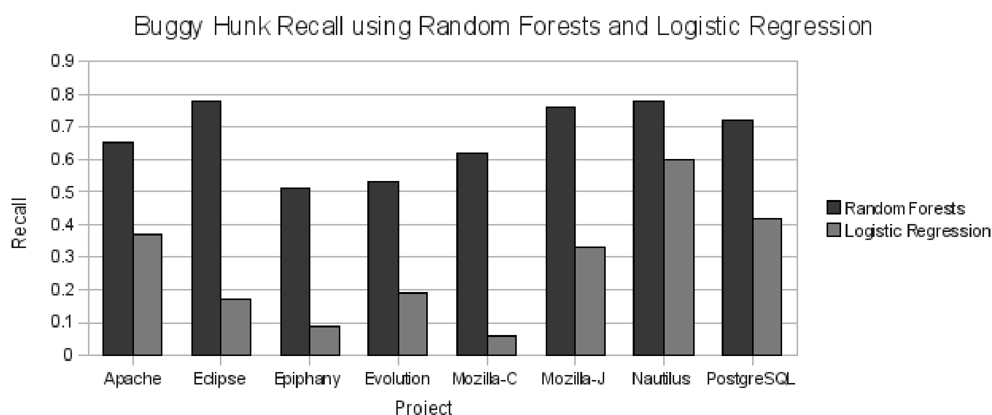


Figure 5.5: Buggy Hunk Recall using Random Forest and Logistic Regression

metrics. One of these metrics is related to size of change that is total number of hunks in a revision (NOH). Other is related to history that is number of buggy hunks found in the previous history of a file (NOBH).

Individual metrics can distinguish between buggy and bug-free hunks with 60% accuracy on an average, see Figure 5.6. For Mozilla project, function declarations, return statement, number of total hunks and number of previous buggy hunks have shown better buggy hunk precision. Whereas for Eclipse project, loops, function calls, return statements, arrays, break statement and classes have shown better buggy hunk precision, as depicted in Figure 5.7.

Individual metrics have produced very poor recall values. Among the code related hunk metrics, function calls, Null statement and case statement have produced better buggy hunk recall for the Mozilla project. Change and history related hunk metrics have produced best buggy hunk recall for both projects, see Figure 5.8.

5.6.6 Performance of Combination of Metrics

To evaluate the performance of metrics groups, we combined related metrics into three groups. The first group was composed of hunk metrics related to methods. The second group was related to classes and the third group was related to change size and history. Following is a detail of the groups:

- *Group 1.* NOCN, NOL, NOA, NOFC, NOFD, NOV, NOLO, NORO, NORS, NON, NOAR and NOB.
- *Group 2.* NOC, NOO, NOIP and NOIH.
- *Group 3.* NOH and NOBH.

Table 5.9: Precision , Recall and Accuracy for Mozilla using individual metrics

Metrics	Accuracy	Buggy Hunk			Bug-Free Hunk		
		Precision	Recall	F1	Precision	Recall	F1
NOCN	0.59	0.566	0.066	0.119	0.59	0.963	0.732
NOL	0.58	0.527	0.036	0.068	0.585	0.977	0.732
NOA	0.58	0.516	0.008	0.016	0.583	0.995	0.735
NOFC	0.60	0.577	0.144	0.231	0.601	0.924	0.728
NOFD	0.58	0.615	0.017	0.034	0.584	0.992	0.736
NOV	0.58	0.303	0.001	0.002	0.582	0.999	0.735
NOLO	0.58	0.479	0.006	0.011	0.582	0.996	0.735
NORO	0.58	0.516	0.008	0.016	0.583	0.995	0.735
NORS	0.58	0.667	0.005	0.01	0.583	0.998	0.736
NON	0.60	0.562	0.156	0.223	0.591	0.914	0.731
NOAR	0.58	0.558	0.018	0.035	0.584	0.99	0.735
NOCS	0.58	0.586	0.166	0.219	0.591	0.951	0.722
NOB	0.58	0.558	0.024	0.046	0.585	0.986	0.734
NOC	0.58	0	0	0	0.582	1	0.736
NOO	0.58	0.489	0.004	0.007	0.582	0.997	0.735
NOIP	0.58	0.5	0	0	0.582	1	0.736
NOIH	0.58	0	0	0	0.582	1	0.736
NOH	0.73	0.829	0.461	0.592	0.706	0.932	0.804
NOBH	0.77	0.783	0.624	0.695	0.764	0.876	0.816

Table 5.10: Precision , Recall and Accuracy for Eclipse using individual metrics

Metrics	Accuracy	Buggy Hunk			Bug-Free Hunk		
		Precision	Recall	F1	Precision	Recall	F1
NOCN	0.65	0.541	0.01	0.02	0.656	0.995	0.791
NOL	0.66	0.638	0.006	0.011	0.656	0.998	0.791
NOA	0.66	0.554	0.009	0.018	0.656	0.996	0.791
NOFC	0.66	0.619	0.009	0.018	0.656	0.997	0.791
NOFD	0.66	0.596	0.008	0.015	0.656	0.997	0.791
NOV	0.66	0.593	0.005	0.011	0.655	0.998	0.791
NOLO	0.66	0.578	0.01	0.02	0.656	0.996	0.791
NORO	0.66	0.604	0.008	0.016	0.656	0.997	0.791
NORS	0.66	0.625	0.006	0.011	0.656	0.998	0.791
NON	0.65	0.532	0.06	0.08	0.666	0.985	0.788
NOAR	0.66	0.616	0.003	0.006	0.655	0.999	0.791
NOB	0.66	0.601	0.007	0.015	0.656	0.997	0.791
NOC	0.65	0.639	0.001	0.003	0.655	1	0.791
NOO	0.66	0.62	0.006	0.012	0.656	0.998	0.791
NOIP	0.65	0.473	0.002	0.005	0.655	0.999	0.791
NOIH	0.66	0.548	0.008	0.015	0.656	0.997	0.791
NOH	0.75	0.839	0.326	0.47	0.731	0.967	0.833
NOBH	0.79	0.781	0.553	0.648	0.796	0.918	0.853

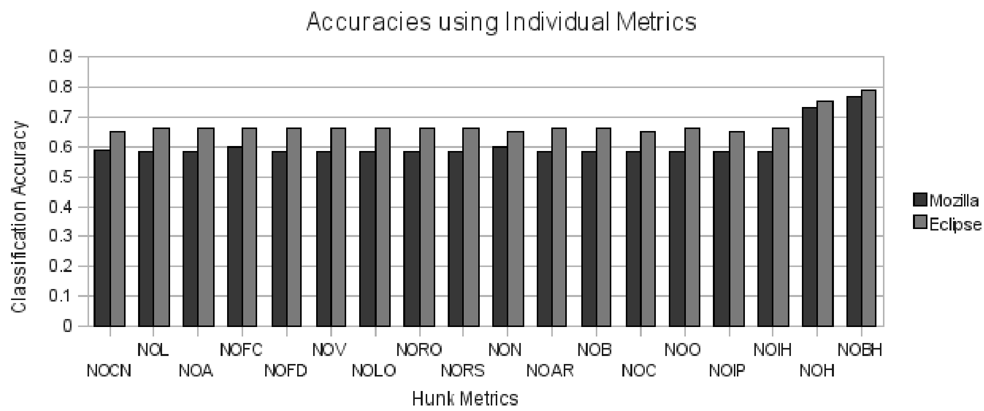


Figure 5.6: Accuracies using Individual Metrics

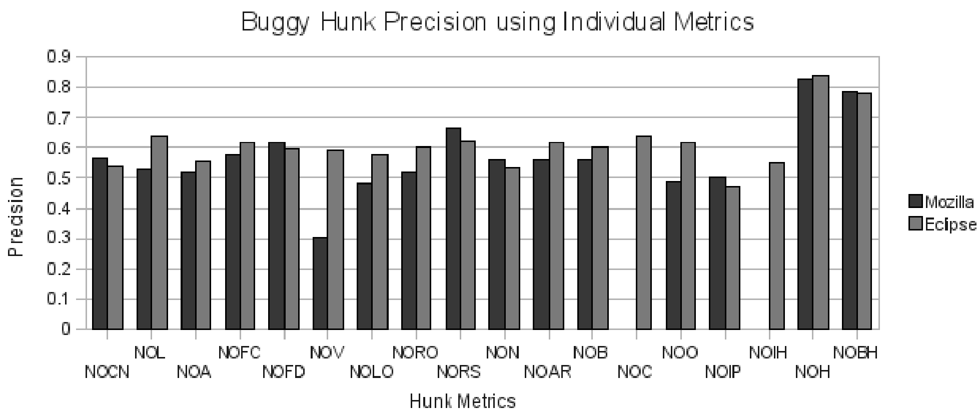


Figure 5.7: Buggy Hunk Precision using Individual Metrics

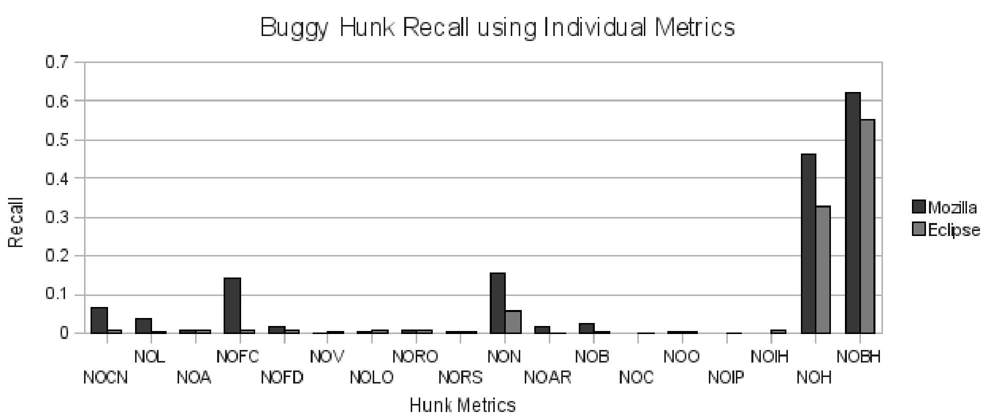


Figure 5.8: Buggy Hunk Recall using Individual Metrics

Table 5.11: Precision , Recall and Accuracy for Mozilla using metrics groups

Metrics	Accuracy	Buggy Hunk			Bug-Free Hunk		
		Precision	Recall	F1	Precision	Recall	F1
Group1	0.60	0.583	0.174	0.268	0.606	0.911	0.727
Group2	0.58	0.543	0.006	0.011	0.583	0.997	0.735
Group3	0.84	0.84	0.768	0.803	0.843	0.895	0.868

Table 5.12: Precision , Recall and Accuracy for Eclipse using metrics groups

Metrics	Accuracy	Buggy Hunk			Bug-Free Hunk		
		Precision	Recall	F1	Precision	Recall	F1
Group1	0.68	0.619	0.189	0.289	0.687	0.939	0.793
Group2	0.66	0.696	0.01	0.02	0.656	0.998	0.792
Group3	0.87	0.869	0.723	0.789	0.866	0.943	0.902

We used each group of metrics as explanatory variables and trained and tested the classifier. Group 2 produced poor results, see Table 5.11 and 5.12. One reason may be few hunks involving class declarations and inheritance statements. Group 1 produces better accuracy but recall values are poor. Group 3 produced the best results. It indicates that buggy files continue to introduce bugs in later releases.

Hunk metrics related to methods and classes can distinguish between buggy and bug-free hunks with similar accuracies, see Figure 5.9. They are equally precise also in identifying buggy hunks, as depicted in Figure 5.10. However class related hunk metrics have very poor buggy hunk recall value. Method related hunk metrics have produced slightly better results with average buggy hunk recall of 18%, as shown in Figure 5.11. The reason may be a few number of hunks involving changes to classes as compared to hunks involving changes to methods.

History and change related hunk mtrics have outperformed other two groups. History related group can distinguish buggy and bug-free hunks with 85% accuracy on an average. It has produced much better buggy hunk precision and recall values that are 85% and 74% respectively.

5.6.7 Cross Project Predictions

In order to know whether a predictor obtained from one project can be applied to other projects, we tested the constructed models across different projects. We tested the models built using random forests, because they produced better results for the same project. Projects developed in JAVA language have some additional metrics related to objects, so we made two groups. One group having JAVA projects and the other having C projects. Table 5.13 shows the classification accuracies obtained by applying predictor obtained from one project, to other projects. The accuracy values range from 49 percent to 75 percent, with most of the values greater than 60 percent. It indicates that predictors obtained from one

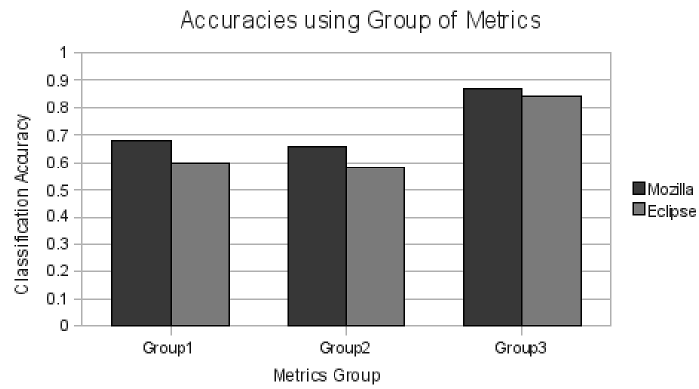


Figure 5.9: Accuracies using Metrics Groups

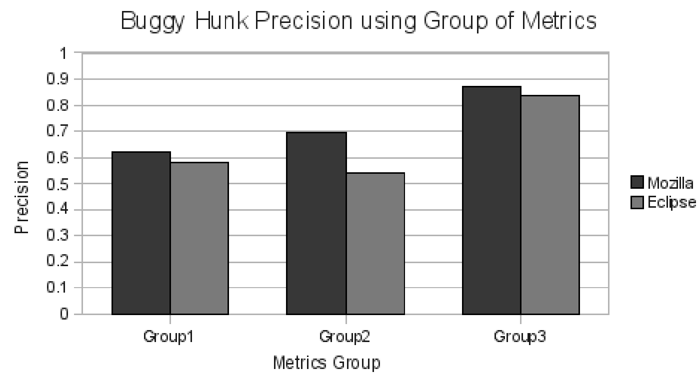


Figure 5.10: Buggy Hunk Precision using Metrics Groups

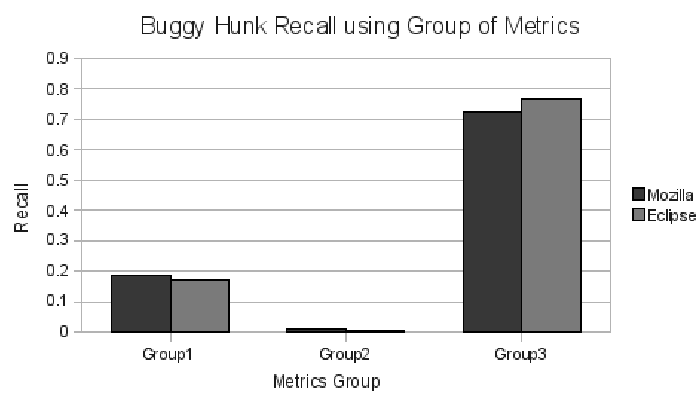


Figure 5.11: Buggy Hunk Recall using Metrics Groups

Table 5.13: Classification accuracies using models from a different project

Project	Apache	Eclipse	Epiphany	Evolution	Mozilla-C	Mozilla-J	Nautilus	PostgreSQL
Apache	—	—	0.67	0.64	0.74	—	0.52	0.65
Eclipse	—	—	—	—	—	0.61	—	—
Epiphany	0.65	—	—	0.63	0.69	—	0.54	0.63
Evolution	0.63	—	0.63	—	0.69	—	0.54	0.62
Mozilla-C	0.75	—	0.63	0.61	—	—	0.52	0.63
Mozilla-J	—	0.65	—	—	—	—	—	—
Nautilus	0.53	—	0.60	0.59	0.60	—	—	0.49
PostgreSQL	0.64	—	0.64	0.63	0.71	—	0.52	—

project based on hunk metrics can be successfully applied to other projects.

Predictor obtained from Apache project could classify hunks from Epiphany, Evolution and PostgreSQL with a similar accuracy of 64%. It could classify only 50% hunks of Nautilus accurately. However it showed better results for Mozilla project with an accuracy of 75%.

Predictor obtained from change data of Epiphany could classify hunks from other projects with an average accuracy of 63%, whereas predictor obtained from Evolution project could classify 62% of hunks from other projects correctly.

Classifier trained on historic data of Mozilla showed better results compared to other classifiers. On an average it could classify 69% hunks correctly, with best accuracies for Apache and PostgreSQL.

Predictor obtained from PostgreSQL showed results similar to the predictor obtained from Apache project. It could classify hunks from Apache, Epiphany and Evolution with a similar accuracy of 64%. It could classify only 50% hunks of Nautilus accurately, whereas for Mozilla project it also showed better results with an accuracy of 71%.

Classifiers obtained from Eclipse and Mozilla, when applied on each other, produced similar results. In both cases the accuracy of classification was about 60%.

5.7 Applications

Hunk classification approach can be used in different ways:

- Hunk classification approach can identify buggy hunks immediately after a hunk is made. It can alarm the developers about the bad code. Developers can review the code changes they have made before committing them to the repository. So hunk classifier can be used as a commit inspector.

- It can be used as part of the software development process. Developers can make changes to the source code, apply hunk classifier to check the changes,

receive notification about the change, modify the changes if required and repeat the same cycle again. One advantage of using hunk classifier is the smallest level of granularity. Developers have to inspect a few lines of code rather than the whole change.

Chapter 6

Threats to Validity

This chapter describes the threats to the validity of this work.

All analyzed projects are open source: The software systems used in this empirical study are all open source, hence they follow a different development methodology. Commercial software projects use different development and maintenance techniques, so there may be different patterns of changes and bugs. Commercial projects use skilled programmers and analysts, so bug introduction patterns may be slightly different. Time pressure is also a major difference between open source and commercial projects which can influence the change patterns.

Studied projects might not be representative: Although eight large open source projects belonging to different domains are used in this study, they cannot represent all kinds of software. Projects with better bug reporting and bug linking facilities may produce better results for classification accuracies. Real time and distributed software may have different change and bug patterns and hence different buggy hunk classification accuracies.

Quality of log comments: A careful processing is used to extract comments from configuration management systems and to identify bug fixes. However quality of the log comments can influence the results. A developer may not properly comment the change, so some bug fixes may be missed. All projects do not use a standard way of writing comments. Some projects follow a numeric bug identifier scheme to represent fix comments while others use keywords like fix, bug or patch in their comments. So some commits may be mistakenly identified as fixes.

Granularity of Versioning Systems: Configuration management systems record changes on line level. So it is difficult to identify which individual syntax element is modified during a change. There may be either a single syntax element changed in a line or multiple elements. Better techniques for identification of individual syntax elements may further enhance the accuracies of results.

Software Design Issues: In this study, changes and bugs of projects are considered which have a development history. No emphasis is given to software design and design time flaws. Different software designs may produce different change and bug patterns. It would be nice to include design time metrics and information for study of changes and bugs.

Although it is difficult to extract precise data from software repositories because of several reasons that may be mapping between bugs and source code locations, extraction of changed code or mapping of changes and bugs to the developers, we can not say that the derived conclusions are entirely wrong. Using a public data set we have to compromise on the validity of data to a certain extent. Keeping in view the available data sources, these results are acceptable.

Chapter 7

Related Work

In this chapter work related to this thesis is discussed. First different approaches and techniques are discussed for extracting valuable facts from software repositories. Next different bug prediction models and techniques are discussed and compared with the hunk classification technique. Then a discussion is made on change extraction and change analysis. Finally a review of buggy code features and code locations is presented.

7.1 Mining Software Change History

Hipikat is a tool that forms implicit group memory for a project by inferring links between stored artifacts and that then recommends relevant part of the group memory to a developer working on the task [12]. It groups four types of artifacts: bug and feature descriptions, source file revisions, messages posted on developer forums, and other project documents. It helps new comer/developer in open source project by providing an efficient and effective access to the group memory for a software development project. Hipikat can be viewed as a recommender system for software developers that draws its recommendation from a projects development history.

Kenyon is a tool that provides automated configuration retrieval from SCM to a local file system and applies fact extractors on each retrieved configuration and then saves the extracted information into a relational database using an object/relation mapping (ORM) system [8]. It reduces the time of research, automates configuration retrieval and allows user control on configuration times. Different SCM systems and multiple data input sources are supported. Kenyon provides efficient, accessible, and optional storage of extracted facts. It uses Hibernate to map its Java objects to a relational database. Hibernate provides a solution to

map database tables to a class. It copies the database data to a class. In the other direction it supports to save objects to the database. In this process the object is transformed to one or more tables. Our modules do a similar job of fact extraction from configuration management systems.

Sliwerski et al. [66] developed a prototype HATARI to detect locations in the software development history where changes have been risky in the past. It relates version archives (such as CVS) to a bug database (such as BUGZILLA) to identify and locate the risky code locations. HATARI makes this risk visible for developers by annotating source code with color bars. Furthermore, HATARI provides views to browse through the most risky locations and to analyze the risk history of a particular location.

7.2 Bug Prediction

Defect prediction studies involve different approaches including product-centric, process-centric and a combination of both. Product-centric approaches use measures obtained from static and dynamic structure of source code or measures extracted from requirements and design documents. A number of studies exist on the use of product-centric approach.

Gyimothy *et al.* [29] validated the object-oriented metrics for fault prediction in open source software. The authors used logistic regression and machine learning techniques to identify faulty classes in Mozilla. They used Chidamber and Kemerer metrics in their study. The authors evaluated eight metrics including weighted methods per class, depth of inheritance tree, response for a class, number of children, coupling between object classes, lack of cohesion on methods, lack of cohesion on methods allowing negative value and lines of code. Bugzilla database was processed and bugs were associated with classes. The authors found that coupling between object classes is the best choice for predicting faulty classes. Lines of code metrics also performed well in predicting faulty classes.

Porter and Selby [61] used classification trees based on metrics from previous releases to identify components having high-risk properties. The authors developed a method of automatically generating measurement-based models of high-risk components.

Koru and Liu [40] combined static software measure with defect data at class level and applied different machine learning techniques to develop bug predictor model. The authors analyzed the CM1, JM1, KC1, KC2, and PC1 data sets in the PROMISE repository, which belong to five software products developed by NASA. Several models were built to predict the defective modules in these products, using the static measures as predictor variables and the binary defectiveness indicator as the response variable. The authors concluded that the prediction performance was not discouraging but not very satisfactory either. However the authors have

proposed defect prediction guidelines based on their experience. They suggest to obtain static measures, aggregate measurs, collect defect data, build a prediction model, predict defect prone classes and improve prediction models. These steps are similar to our approach however we obtain defect data on the level of hunks and our model is automatically improved as more history data becomes available for a project.

Moser et al. [50] presented a comparative analysis of the predictive power of product and process metrics for defect prediction. The authors classified Java files of Eclipse project as defective or defect-free. They built classification models using logistic regression, Naive Bayes and decision trees. The authors performed a cost sensitive classification to allow different costs for prediction errors. They concluded that change data and process related metrics contain more discriminatory and meaningful information about distribution of defects in software than the source code itself. The authors used 18 change metrics to train a decision tree learner and obtained greater than 75% accuracy, 80% recall and less than 30% false positive rate. The change metrics included in their study are number of revisions, number of refactorings, number of bug fixes, number of authors, LOC added, LOC deleted, Codechurn, change set and age of a file. Their findings are similar to us as change and history related hunk metrics produce better results than the code related hunk metrics. Note that in contrast to defect prediction for files, our technique produces predictions for individual hunks.

Pan et al. [56] introduced program slicing metrics to be used as bug predictors. They used program slice information to measure the size, complexity, coupling and cohesion properties of C language programs. The slicing metrics used in their study include slice count, vertices count, edges count, edges to vertices ratio, slice vertices sum, maximum slice vertices, global input, global output, direct fan in, direct fan out, indirect fan in, indirect fan out and lack of cohesion. The authors compared bug classification capabilities of program slicing metrics with Understand for C++ suite of metrics in a number of experiments. They found that program slicing metrics produce slightly better classification accuracies than Understand for C++ metrics at the file level.

Nagappan *et al.* [52] applied principal component analysis on code metrics and developed regression models to predict the post-release defects. The authors found that there is no single set of complexity metrics that could act as a universally best defect predictor. The authors also found that predictors obtained from one project were significant for other similar projects.

Menzies *et al.* [46] showed that predictors obtained from static code attributes are useful in defect prediction with a mean probability of detection of 71 percent and mean false alarms of 25 percent. The authors found that it is more important, how the attributes are used to build predictors than which particular attributes are used. A number of attributes were used in this study including Mccabe and

Halstead complexity metrics.

Ostrand *et al.* [55] used code of the file in current release and fault and modification history of the previous releases to predict the expected number of faults in each file of the next release.
Process-centric approaches use measures extracted from the software history such as changes made to software, developers involved, size and time of changes, and age of software. Various studies are found in literature using process artifacts.

Ratzinger *et al.* [64] used regression models and decision trees to predict defects in short time frames of two months. The authors used features extracted from version control and feature tracking systems to build their models. The authors also investigated the predictability of several severities of defects in software projects.

Kim *et al.* [35] proposed a bug finding algorithm using the project-specific bug and fix knowledge base developed by analyzing the history of bug fixes. The authors implemented a tool BugMem for detecting potential bugs and suggesting corresponding fixes.

Hassan and Holt [30] presented an approach named, The Top Ten List, to predict the ten most susceptible subsystems having a fault. The authors used some heuristics to create the Top Ten List. These heuristics were based on the characteristics of software system such as recency, frequency and size of modifications as well as code metrics and co-modifications.

7.3 Software Change Extraction and Analysis

Fluri and Gall [21] proposed an approach for analyzing and classifying change types based on code revisions. Using that approach, changes on the method or class level could be differentiated and their significance in terms of the impact of the change types on other source code entities be assessed. The authors found that in many cases large numbers of lines added and/or deleted are not accompanied by significant changes but small textual adaptations. The authors presented a taxonomy of source code changes to be used for change coupling analysis and used tree edit operations in the AST to classify changes. Their classification approach could assess error-proneness of source code entities, qualify change couplings, or identify programming patterns.

Canfora et al. [11] proposed a technique to identify changes at source code line level from CVS repositories. They used Vector Space Models and the Levenshtein edit distance to determine if CVS/SVN diffs are due to line additions/deletions or if they are due to line modifications. A tokenizer was used instead of a parser to extract symbols and then compute the cosine similarity. Application of the technique on a random sample of ArgoUML snapshots indicated high precision (96%) and a high recall as well (95%). We use a different approach to identify

the bug-inducing hunks and the changed source code lines.

Fluri et al. [22] in an empirical study found that change type patterns do describe development activities and affect the control flow, the exception flow, or change the API. The authors used agglomerative hierarchical clustering to discover patterns of change types. To explore whether change types appear frequently and commonly, the authors extracted data from one commercial and two open source software systems. In contrast to general change types we study the features of bug-inducing changes.

Stoerzer et al. [68] presented an approach for change classification that helps programmers identify the changes responsible for test failures. The authors proposed several change classifiers that associate the colors Red, Yellow, or Green with changes, according to the likelihood that they were responsible for test failures. The authors used a model of atomic changes, with change categories such as added classes (AC), deleted classes (DC), added methods (AM), deleted methods (DM), changed method bodies (CM), added fields (AF), deleted fields (DF), and lookup changes (LC) (i.e., changes to dynamic dispatch). The authors considered changes to method bodies as one CM change regardless of the number of statements changed within the respective method's body. They conducted two case studies to investigate whether or not change classification can be a useful tool for focusing the attention of programmers on failure-inducing changes. In contrast we consider atomic changes as changes to individual language constructs and process the change history of a project rather than test information. We study which language constructs have more likelihood of generating bugs.

Mockus and Weiss [48] presented a model to predict the risk of new changes, based on historic information. The authors modeled the probability of causing failure of a change made to software. They used properties of a change as model parameters such as size in lines of coded added, deleted or unmodified, diffusion of the change reflected by the number of files, modules or subsystems touched, several measures of developer experience and the type of change. The authors found that change diffusion and developer experience are essential to predict failures.

Aversano et al. [6] developed a model to predict if a new change may introduce a bug or not. The authors extracted bug-introducing changes from software change history and constructed feature vectors from the source code. They represented software changes as elements of an n-dimensional vector space of terms. The constructed vectors were used to train different classifiers on data of two open source projects. The authors used K-Nearest Neighbor, simple logistic, Multi-Boosting, C4.5 and Support Vector Machines as classifiers. K-Nearest Neighbor produced better results as compared to other classifiers. This work is similar to our work but the results of change classification are poor with 63% precision and 40% recall for buggy changes. Our technique produces much better results and works at finest level of granularity.

Kim et al. [34] introduced a technique for classifying a software change as clean or buggy. The authors trained a machine learning classifier using features extracted from revision history of a software project. The features used include all terms in the complete source code, the lines modified in each change (delta), change metadata such as author, change time, and complexity metrics. The proposed model could classify changes as clean or buggy with 70 percent accuracy and 60 percent buggy change recall on average. The authors predicted faults at the file change level whereas our approach predicts faults at the smallest level of granularity, that is a hunk. Furthermore, hunk classification approach uses very less data for classification, so it is simple and easy to apply. It produces better results as compared to [34] while using less number of input variables.

Graves *et al.* [27] processed change management data to predict distribution of faults over modules of a software system. The authors found that the number of times a code has been changed is a good predictor of faults. The authors further found that modules which changed recently may have more faults than those modules which are not changed since a longer time.

Hassan and Holt [31] analyzed the development history of five open source projects to study change propagation. They proposed several heuristics to predict change propagation and validated their results using the obtained historical data.

German [26] studied the characteristics of modification requests with respect to source files and their authors. The author proposed several metrics to quantify modification requests and used these metrics to create visualization graphs for understanding interrelationships.

Gall et al. [23] developed an approach using release history information of a system to identify logical couplings and change patterns among modules. The authors used structural information about programs, modules, and subsystems, together with their version numbers and change reports to uncover hidden dependencies in the source code.

Ying et al. [73] mined software change history data to find file co-change patterns. The authors proposed that change patterns can be used to recommend potentially relevant source code to a developer performing a modification task.

Weiβgerber and Diehl presented a technique to detect changes that are likely to be refactorings and rank them according to the likelihood. The evaluation of the technique showed a high recall and a high precision, it finds most of the refactorings, and most of the found refactoring candidates are really refactorings. The proposed technique is able to find structural and local refactorings. Structural refactorings include Move Class, Move Interface, Move Field, Move Method, and Rename Class, whereas local refactorings include Rename Method, Hide Method, Unhide Method, Add Parameter, and Remove Parameter.

7.4 Buggy Code Features and Locations

Pan et al. [57] defined bug fix patterns using the syntax components and context of the source code involved in bug fix changes. Software repositories of seven open source projects, developed in JAVA, were used to extract the bug fix patterns. The authors found 45.7% to 63.3% of the total bug fix hunk pairs in these projects having the defined bug fix patterns. The most common individual patterns are method call with different actual parameter values, change in if conditional, and change of assignment expression. Correlation analysis of seven projects and five developers showed similar frequencies of bug fix patterns. This study is similar to ours, but we consider bug-inducing changes instead of bug-fix changes. Furthermore, we use software systems developed in different languages rather than same language.

Kim et al. [36] analyzed the version history of seven software systems to predict the most fault prone entities and files. The authors implemented a cache for holding locations that are likely to have faults: starting from the location of a known (fixed) fault, the location itself, any locations changed together with the fault, recently added locations, and recently changed locations. A developer can detect likely fault-prone locations by consulting the cache whenever a fault is fixed. The developed algorithm is evaluated on seven open source projects, and it is 73%-95% accurate at predicting future faults at the file level and 46%-72% accurate at the entity level with optimal options. The prediction algorithm is executed over the change history of a software project, which yields a small subset (usually 10%) of the project?s files or functions/methods that are most fault-prone. The authors base their algorithm on the observation that most faults are local, they do not occur uniformly in time across the history of a function, rather they appear in bursts. Four different kinds of locality are considered for bug occurrences including changed-entity locality, new-entity locality, temporal locality and spatial locality.

Brun and Ernst [10] proposed a technique for identifying program properties that indicate errors. They trained machine learning models on program properties that resulted from errors and then applied these models to program properties of user written code to classify and rank properties that could lead to errors. Given a set of properties produced by the program analysis, the technique selects a subset of properties that are most likely to reveal an error. Dynamic invariant detection is used to generate program properties and two machine learning tools are used to classify those properties. The authors used support vector machine and decision tree in their experiments, and found that this technique increases the relevance (the concentration of fault-revealing properties) by a factor of 50 on average for the C programs, and 4.8 for the Java programs. The authors concluded that most of the fault-revealing properties do lead a programmer to an error. They suggested that ranking and selecting the top properties is more advantageous than

selecting all properties considered faultrevealing by the machine learner. For C programs, on average 45% of the top 80 properties are fault-revealing, whereas, for Java programs, 59% of the top 80 properties are faultrevealing. In the preliminary experiments most of the fault-revealing properties lead a programmer to the error, but it is not necessary for all properties.

Li and Zhou [44] proposed a method called PR-Miner to efficiently extract implicit programming rules from large software code written in an industrial programming language such as C. It uses a data mining technique called frequent itemset mining and requires little effort from programmers without any prior knowledge of the software. PR-Miner can extract programming rules in general forms (without being constrained by any fixed rule templates) that can contain multiple program elements of various types such as functions, variables and data types. The authors also proposed an efficient algorithm to automatically detect violations to the extracted programming rules, which can be strong indications of bugs. PR-Miner was evaluated with large software code, including Linux, PostgreSQL Server and the Apache HTTP Server, having 84K-3M lines of code each. Experiments showed that PR-Miner can efficiently extract thousands of general programming rules and detect violations within 2 minutes.

Livshits and Zimmermann [45] proposed a tool called DynaMine, that analyzes source code check-ins to find highly correlated method calls as well as common bug fixes in order to automatically discover application-specific coding patterns. Potential patterns discovered through mining are passed to a dynamic analysis tool for validation and the results of dynamic analysis are presented to the user. The authors combined revision history mining and dynamic analysis techniques for discovering application specific patterns and for finding errors. DynaMine is evaluated on two widely-used, mature, highly extensible applications, Eclipse and jEdit, that collectively consist of more than 3,600,000 lines of code. The authors discovered 56 previously unknown, highly application-specific patterns, out of which 21 were dynamically confirmed as very likely valid patterns, and found 263 pattern violations by mining history data of Eclipse and jEdit.

Chapter 8

Future work

A static parser was used to extract language constructs and syntax elements. A bug inducing hunk may contain multiple language constructs. It is possible that only one construct is changed, or there may be multiple constructs changed in a single hunk. Currently all language constructs in a bug inducing hunk are considered bug inducing because configuration management systems provide information on the line level. We plan to develop techniques to identify the exact individual language construct which contributes to a bug within a hunk.

In this study, only frequencies of bug inducing language constructs are examined. No context information is extracted from the source code. Our parser scans the code of bug inducing hunks and extracts the language constructs involved. We would like to know the context in which different language constructs introduce bugs. We also want to study the coupling between language constructs for introduction of bugs.

Our parser can only extract syntactic elements and no consideration is given to semantics of the program. As same language constructs are present in the bug inducing and clean hunks, it would be interesting to know the situations in which a particular language construct can introduce bugs. For this purpose, we plan to include program control flow and data dependence information with each construct. We will enhance the parser with program analysis capabilities in future.

To study the influence of programming language on post release bugs, case study of Mozilla project is used. Although Mozilla is a large, heterogeneous project, generalized conclusions can not be drawn from a single project. We want to extend this study to a diverse set of projects as a future work.

To study the relationship between the programming language and the defect density, whole program files are used without any consideration of implemented functionality. We want to analyze the features implemented in different languages

as a future work. We would like to split this study on module level and architectural units in future. Although hunk classification approach has produced excellent results, there still exists room for improvement. Among the machine learning classifiers, only random forest is used in this study. Other machine learning algorithms can also be tried and their accuracies evaluated. It may be possible that other machine learning tools produce better precision and recall.

Machine learning algorithms can be modified to suit the specific problem needs. Modified algorithms may produce better results than existing ones in terms of accuracy, precision and recall. Hunk classification approach has used two change and history related metrics. Exploration of other process related hunk metrics remains as future work. It is possible that some other process related hunk metrics may better classify hunks as buggy or bug-free.

Online machine learning algorithms can be used to train a classification model and provide the results during the development of the project. It would be great to have a classifier which can be updated online. We plan to integrate this technique in an integrated development environment.

Chapter 9

Conclusion

This dissertation presented an empirical analysis of changes and bugs by mining software development history. Main focus of this study was to analyze features of bug inducing changes and develop a bug prediction model. Changes were studied at the finest granularity level of hunks. A technique was introduced in this thesis to identify bug inducing hunks. Different language constructs and syntax elements were extracted from bug inducing hunks and their frequencies were compared. A statistical analysis of projects and developers was presented for the frequencies of bug inducing language constructs. Bug latency values for individual language constructs were calculated and statistically analyzed. Bug densities of programs written in different languages were statistically analyzed to find the influence of programming language on post release bugs. A number of evolution metrics were calculated and compared for programs written in different languages. Finally a new set of metrics was introduced called hunk metrics and a technique was presented to classify hunks as buggy or bug free.

Bug introducing changes hold important information about the creator of bugs and the time of creation. Further bug inducing changes can be used to study features of source code which result in bugs. An algorithm for identifying bug inducing changes was proposed by Sliwerski et al. [67] . It was further enhanced by Kim et al. [37]. this algorithm can identify changes at file level. An approach was presented in this thesis that can identify bug inducing hunks. It examines all hunks involved in a change and marks only those hunks as buggy which actually contributed to bugs. Language constructs and syntax elements were extracted from bug inducing hunks of eight open source projects. Twenty six different language constructs were chosen for this study. The results show that most frequent bug-inducing language constructs are function calls, assignments, conditions, pointers, use of NULL, variable declaration, function declaration and return statement. These eight constructs are found in 38-62%, 30-42%, 17-40%,

11-30%, 1-22%, 11-25%, 8-12% and 8-15% of bug inducing hunks respectively. Overall these eight elements account for more than 70% of the bug-inducing hunks. Function Calls is found to be the most dominant source of errors in all projects. Use of pointers and NULL is highly problematic in projects developed in C language.

A correlation analysis was applied on bug inducing language constructs of different projects. The results show that different projects are statistically correlated for the frequencies of bug inducing language constructs. The obtained correlation coefficients are significant at $p<0.001$. It indicates that most of the time similar language constructs create problem in different projects.

Results of the correlation analysis show that different developers are significantly correlated for the frequencies of bug inducing language constructs. The correlation coefficients obtained within the same project range from 0.31 to 0.99. Results obtained indicate a minimum correlation coefficient of 0.82 among any pair of developers of different projects but developed in the same language. The maximum correlation coefficient found is 0.98 for the same set of developers. However majority of the correlation coefficients found either within the same project or different projects are above 0.80. The results show that most of the developers tend to face difficulties with similar language constructs. Statistical analysis indicates that majority of the developers induce similar kinds of bugs independent of the project and programming language.

Bug latency values were calculated for conditions, assignments, function calls, variable declarations and function declarations. Correlation analysis of these constructs shows that these language constructs are statistically correlated for bug latency. Most of the obtained correlation coefficients are above 0.95. It can be concluded that bug latencies for individual language constructs vary in similar fashion in different projects.

Statistical analyses of bug densities have revealed that post release bugs are influenced by programming language. Results of hypothesis testing have shown that Java programs are less error prone than C or C++ programs, and C programs are less error prone than C++ programs within same project. It is found that bug life time for Java is twice as long as for C or C++.

This thesis introduced hunk metrics and a technique to classify hunks as buggy or bug-free based on these metrics. A hunk is the smallest unit of a change and this technique works for this finest level of granularity with an average accuracy of 81%. Bug prediction models were built using logistic regression and random forests. Results have shown that random forests can better discriminate between buggy and bug-free hunks. The hunk classification technique was evaluated on eight large open source projects. It classified hunks with 77% buggy hunk precision and 67% buggy hunk recall on average.

Individual hunk metrics were analyzed for their bug prediction capabilities.

Results of multivariate logistic regression have shown that NOCN, NOA, NOFC, NORS, NOBH and NOH are significant for classifying hunks in most of the projects. Hunk metrics related to change and history are found to be better predictor of bugs than code related hunk metrics.

Predictors based on hunk metrics were also used for cross project predictions. Predictors obtained from one project when applied to a different project could classify hunks with more than 60% accuracy.

Overall, work presented in this thesis has strengthened the existing body of knowledge on bug prediction and change analysis. I hope this work will provide a base for further work on bug inducing changes and source code analysis. Mining of software change history can create awareness among developers for buggy code features and it can improve the debugging process.

Bibliography

[1] Bugzilla. `http://www.bugzilla.org/`. [cited at p. 23]

[2] Weka. `http://www.cs.waikato.ac.nz/ml/weka/`. [cited at p. 65, 71]

[3] Software bug, 2006. `http://en.wikipedia.org/wiki/Computer_bug`. [cited at p. 4]

[4] S. N. Ahsan, J. Ferzund, and F. Wotawa. Are there language specific bug patterns? results obtained from a case study using mozilla. In *Proc. of Fourth International Conference on Software Engineering Advances (ICSEA'09)*, Porto, Portugal, 2009. [cited at p. 6]

[5] S. N. Ahsan, J. Ferzund, and F. Wotawa. Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine. In *Proc. of Fourth International Conference on Software Engineering Advances (ICSEA'09)*, Porto, Portugal, 2009. [cited at p. 4]

[6] Lerina Aversano, Luigi Cerulo, and Concettina Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Proc. Ninth international workshop on Principles of software evolution*, pages 19–26, Dubrovnik, Croatia, 2007. [cited at p. 93]

[7] T. Ball, J. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk. In *Proc. ICSE Workshop Process Modelling and Empirical Studies of Software Eng.*, 1997. [cited at p. 4]

[8] J. Bevan, E. J. Whitehead Jr., S. Kim, and M. Godfrey. Facilitating software evolution with kenyon. In *Proc. Of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005)*, pages 177–186, Lisbon, Portugal, 2005. [cited at p. 89]

[9] L. Breiman. Random forests. *Machine Learning*, 45:5–32, October 2001. [cited at p. 71]

[10] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proc. of 26th International Conference on Software Engineering (ICSE 2004)*, pages 480–490, Scotland, UK, 2004. [cited at p. 95]

[11] G. Canfora, L. Cerulo, and M.D. Penta. Identifying changed source code lines from version repositories. In *Proc. Int?l Workshop Mining Software Repositories*, pages 14–21, 2007. [cited at p. 92]

[12] D. Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 408–418, Portland, Oregon, 2003. [cited at p. 23, 89]

[13] N.E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. On Software Engineering*, 26:797–814, Aug 2000. [cited at p. 23, 47]

[14] J. Ferzund, S. N. Ahsan, and F. Wotawa. Analysing bug prediction capabilities of static code metrics in open source software. In *Proc. of International Conference on Software Process and Product Measurement, LNCS Vol. 5338*, pages 331–343, Munich, Germany, 2008. [cited at p. 9, 63]

[15] J. Ferzund, S. N. Ahsan, and F. Wotawa. Automated classification of faults in programms using machine learning techniques. In *Proc. of Artificial Intelligence Techniques in Software Engineering Workshop, ECAI*, Patras, Greece, 2008. [cited at p. 9, 63]

[16] J. Ferzund, S. N. Ahsan, and F. Wotawa. Bug-inducing language constructs. In *Proc. of 16th Working Conference on Reverse Engineering (WCRE'09)*, Lille, France, 2009. [cited at p. 6]

[17] J. Ferzund, S. N. Ahsan, and F. Wotawa. Empirical evaluation of hunk metrics as bug predictors. In *Proc. of International Conference on Software Process and Product Measurement*, Amsterdam, Netherlands, 2009. [cited at p. 6]

[18] J. Ferzund, S. N. Ahsan, and F. Wotawa. Software change classification using hunk metrics. In *Proc. of 25th IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, Alberta, Canada, 2009. [cited at p. 6]

[19] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proc. 10th Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, British Columbia, Canada, 2003. [cited at p. 4]

[20] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. 19th Int'l Conference on Software Maintenance*, pages 23–32, Amsterdam, The Netherlands, 2003. [cited at p. 4, 17]

[21] B. Fluri and H. C. Gall. Classifying change types for qualifying change couplings. In *Proceedings of the 9th International Conference on Program Comprehension*, pages 35–45, 2006. [cited at p. 92]

[22] B. Fluri, E. Giger, and H. C. Gall. Discovering patterns of change types. In *Proceedings of the 23rd International Conference on Automated Software Engineering*, 2008. [cited at p. 93]

[23] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *In Proc. Int'l Conf. Software Maintenance (ICSM'98)*, pages 190–198, 1998. [cited at p. 23, 94]

[24] R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Proc. of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, Anaheim, California, USA, 2003. [cited at p. 8]

[25] S. Garfinkel. History's worst software bugs, 2005. `http://wired.com/news/technology/bugs/0,2924,69355,00.html`. [cited at p. 4]

[26] D.M. German. An empirical study of fine-grained software modifications. In *Proc. 20th Int'l Conf. Software Maintenance (ICSM'04)*, pages 316–325, 2004. [cited at p. 94]

[27] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26:653–661, July 2000. [cited at p. 9, 23, 63, 94]

[28] J. P. Guilford and B. Fruchter. *Fundamental Statistics in Psychology and Education*. McGraw-Hill, New York, 1973. [cited at p. 72]

[29] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Software Eng.*, 31(10):897–910, Oct 2005. [cited at p. 9, 63, 90]

[30] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *Proc. 21st Int'l Conf. Software Maintenance*, pages 263–272, 2005. [cited at p. 92]

[31] A.E. Hassan and R.C. Holt. Predicting change propagation in software systems. In *Proc. Int'l Conf. Software Maintenance (ICSM 2004)*, 2004. [cited at p. 23, 94]

[32] S. Kim, Jr. E. J. Whitehead, and J. Bevan. Properties of signature change patterns. In *Proc. of International Conference on Software Maintenance (ICSM 2006)*, pages 4–14, Dublin, Ireland, 2006. [cited at p. 23]

[33] S. Kim and E. J. Whitehead Jr. How long did it take to fix bugs? In *Proc. international workshop on Mining software repositories*, pages 173–174, Shanghai, China, 2006. [cited at p. 49]

[34] S. Kim, E. J. Whitehead Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Software Eng.*, 34(2):181–196, Mar/Apr 2008. [cited at p. 63, 64, 94]

[35] S. Kim, K. Pan, and E. J. Whitehead Jr. Memories of bug fixes. In *Proc. 14th ACM Symp. Foundations of Software Eng.*, pages 35–45, 2006. [cited at p. 9, 63, 92]

[36] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proc. 29th Int'l Conference on Software Engineering (ICSE 2007)*, pages 489–498, Minneapolis, USA, 2007. [cited at p. 23, 95]

[37] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead Jr. Automatic identification of bug-introducing changes. In *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 81–90, 2006. [cited at p. 7, 17, 99]

[38] D. G. Kleinbaum and M. Klein. *Logistic Regression –A Self-Learning Text*. Springer-Verlag, New York, 2002. [cited at p. 69]

[39] A. G. Koru and J. Tian. An empirical comparison and characterization of high defect and high complexity modules. *Journal of Systems and Software*, 67:153–163, Sep 2003. [cited at p. 8]

[40] A.G. Koru and H. Liu. Building effective defect-prediction models in practice. *IEEE Software*, 22:23–29, November/December 2005. [cited at p. 9, 63, 90]

[41] F. Lanubile and G. Visaggio. Evaluating predictive quality models derived from software measures: lessons learned. *Journal of Systems and Software*, 38:225–234, Sep 1997. [cited at p. 9]

[42] Y. Levendel. Reliability analysis of large software systems: Defect data modeling. *IEEE Transactions on Software Engineering*, 16:141–152, Feb 1990. [cited at p. 8]

[43] 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. In *Proc. 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, San Jose, California, 2006. [cited at p. 8]

[44] Z. Li and Y. Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. of 13th International Symposium on Foundations of Software Engineering*, pages 306–315, Lisbon, Portugal, 2005. [cited at p. 96]

[45] B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proc. of 13th International Symposium on Foundations of Software Engineering*, pages 296–305, Lisbon, Portugal, 2005. [cited at p. 96]

[46] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Software Eng.*, 33(1):2–13, Jan 2007. [cited at p. 91]

[47] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proc. 16th Int'l Conference on Software Maintenance*, pages 120–130, San Jose, California, USA, 2000. [cited at p. 17, 23]

[48] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical J.*, 5(2):169–180, 2002. [cited at p. 63, 93]

[49] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proc. 26th International Conference on Software Engineering*, pages 282–292, 2004. [cited at p. 8]

[50] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proc. of International Conference on Software Engineering (ICSE'08)*, pages 181–190, Leipzig, Germany, 2008. [cited at p. 91]

[51] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18:423–433, May 1992. [cited at p. 8]

[52] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. of 28th Int'l Conference on Software Engineering*, Shanghai, China, 2006. [cited at p. 9, 63, 91]

[53] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. In *Proc. 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 55–64, 2002. [cited at p. 47]

[54] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *Proc. of 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 86–96, Boston, Massachusetts, USA, 2004. [cited at p. 23]

[55] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Software Eng.*, 31(4):340–355, 2005. [cited at p. 9, 63, 92]

[56] K. Pan, S. Kim, and Jr. E. J. Whitehead. Bug classification using program slicing metrics. In *Proc. Sixth IEEE Int'l Workshop Source Code Analysis and Manipulation*, 2006. [cited at p. 9, 63, 91]

[57] K. Pan, S. Kim, and E. J. Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14:286–315, June 2009. [cited at p. 95]

[58] D. L. Parnas. Software aging. In *Proc. 16th International Conference on Software Engineering*, pages 279–287, 1994. [cited at p. 3]

[59] S. L. Pfleeger and J.M. Atlee. *Software Engineering –Theory and Practice*. Pearson Education, Inc., 3rd edition, 2006. [cited at p. 3]

[60] M. Pighin and A. Marzona. An empirical analysis of fault persistence through software releases. In *Proc. IEEE/ACM ISESE*, pages 206–212, 2003. [cited at p. 23]

[61] A. A. Porter and W. R. Selby. Empirically-guided software development using metric-based classification trees. *IEEE Software*, 7:46–54, Mar 1990. [cited at p. 90]

[62] L. Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33:23–29, 2000. [cited at p. 8]

[63] R. S. Pressman. *Software Engineering –A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001. [cited at p. 3]

[64] J. Ratzinger, M. Pinzger, and H. Gall. Eq-mine: Predicting short-term defects for software evolution. In *Proc. of FASE'07*, pages 12–26, Braga, Portugal, 2007. [cited at p. 9, 63, 92]

[65] A. Schroter, T. Zimmermann, R. Premraj, and A. Zeller. If your bug database could talk. In *Proc. 5th International Symposium on Empirical Software Engineering*, pages 18–20, 2006. [cited at p. 4]

[66] J. Sliwerski, T. Zimmermann, and A. Zeller. Hatari: Raising risk awareness. In *Proc. 10th European Software Eng. Conf. and 13th ACM SIGSOFT Symposium Foundations Software Eng.*, pages 107–110, 2005. [cited at p. 90]

[67] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. of Int'l Workshop on Mining Software Repositories*, pages 24–28, Saint Louis, Missouri, USA, 2005. [cited at p. 3, 7, 17, 99]

[68] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in java programs using change classification. In *Proc. Symposium Foundations Software Eng*, pages 57–68, 2006. [cited at p. 93]

[69] K. S. Trividi. *Probability Statistics with Reliability, Queuing, And Computer Science Applications*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1982. [cited at p. 48, 57]

[70] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proc. international workshop on Mining software repositories*, 2007. [cited at p. 49]

[71] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Software Eng.*, 31(6):466–480, 2005. [cited at p. 23]

[72] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2005. [cited at p. 9, 75]

[73] A.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Software Eng.*, 30:574–586, Sept 2004. [cited at p. 23, 94]

[74] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier, 2006. [cited at p. 4]

[75] T. Zimmerman, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. of Int'l Conference on Software Engineering (ICSE '04)*, pages 563–572, Edinburgh, Scotland, UK, 2004. [cited at p. 23]

[76] T. Zimmermann, S. Kim, A. Zeller, and Jr. E. J. Whitehead. Mining version archives for co-changed lines. In *Proc. of Int'l Workshop on Mining Software Repositories (MSR 2006)*, pages 72–75, Shanghai, China, 2006. [cited at p. 23]

Appendices

Appendix A

List of Publications

The work covered by this thesis led to following publications:

- J. Ferzund, S. N. Ahsan, and F. Wotawa. Bug-inducing language constructs. In *Proc. of 16th Working Conference on Reverse Engineering (WCRE'09)*, Lille, France, 2009.
- S. N. Ahsan, J. Ferzund, and F. Wotawa. Are there language specific bug patterns? results obtained from a case study using mozilla. In *Proc. of Fourth International Conference on Software Engineering Advances (ICSEA'09)*, Porto, Portugal, 2009.
- J. Ferzund, S. N. Ahsan, and F. Wotawa. Software change classification using hunk metrics. In *Proc. of 25th IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, Alberta, Canada, 2009.
- J. Ferzund, S. N. Ahsan, and F. Wotawa. Empirical evaluation of hunk metrics as bug predictors. In *Proc. of International Conference on Software Process and Product Measurement*, Amsterdam, Netherlands, 2009.

List of Figures

2.1 Architecture for Data Extraction . . . 15
2.2 Steps for Hunk Extraction . . . 17
2.3 Steps for identifying bug-inducing hunks . . . 18
2.4 CVS Log . . . 19
2.5 CVS Difference . . . 20
2.6 CVS Annotations . . . 20
2.7 CVSDifference table entries . . . 21

3.1 Proportion of hunk types in different projects . . . 31
3.2 Bug-inducing language constructs in different projects (a) . . . 35
3.3 Bug-inducing language constructs in different projects (b) . . . 36
3.4 Frequency distribution of correlation coefficients (same project) . . . 37
3.5 Frequency distribution of correlation coefficients (different project) . . . 38
3.6 Comparison of Bug-Inducing and Clean Hunks (Apache) . . . 42
3.7 Comparison of Bug-Inducing and Clean Hunks (Eclipse) . . . 43
3.8 Comparison of Bug-Inducing and Clean Hunks (Mozilla) . . . 43
3.9 Comparison of Bug-Inducing and Clean Hunks (PostgreSQL) . . . 44
3.10 Comparison of Bug-Inducing and Clean Hunks (Evolution) . . . 44
3.11 Comparison of Bug-Inducing and Clean Hunks (Epiphany) . . . 45
3.12 Comparison of Bug-Inducing and Clean Hunks (Columba) . . . 45
3.13 Comparison of Bug-Inducing and Clean Hunks (Nautilus) . . . 46

4.1 Average bug densities . . . 50
4.2 Percentage of faulty files . . . 51
4.3 Average LOC of faulty files . . . 51
4.4 Average revision frequency . . . 52
4.5 Average code gain per file . . . 52
4.6 Bug severity distribution . . . 53
4.7 Average bug lifetime . . . 54

4.8 Average code additions 54
4.9 Average code deletions 55
4.10 Average Code Deletions / Bug Fix 55
4.11 Average code additions per bug fix 55
4.12 Average number of changes 56
4.13 Distribution of bugs on different platforms 56
4.14 Distribution of bugs on different operating systems 57
4.15 The bug density distribution of files written in Java 60
4.16 The bug density distribution of files written in C 60
4.17 The bug density distribution of files written in C++ 61

5.1 Logit Function 70
5.2 Odds Function 70
5.3 Accuracies using Random Forest and Logistic Regression 78
5.4 Buggy Hunk Precision using Random Forest and Logistic Regression . 78
5.5 Buggy Hunk Recall using Random Forest and Logistic Regression . . . 79
5.6 Accuracies using Individual Metrics 81
5.7 Buggy Hunk Precision using Individual Metrics 81
5.8 Buggy Hunk Recall using Individual Metrics 81
5.9 Accuracies using Metrics Groups 83
5.10 Buggy Hunk Precision using Metrics Groups 83
5.11 Buggy Hunk Recall using Metrics Groups 83

List of Tables

2.1 CVSLog table description 16
2.2 CVSDifference table description 16
2.3 CVSAnnotations table description 16
2.4 Description of Projects 21

3.1 Language Constructs 26
3.2 Frequencies of Bug-Inducing Language Constructs(a) 32
3.3 Frequencies of Bug-Inducing Language Constructs(b) 33
3.4 Correlation coefficients for different projects 35
3.5 Correlation Coefficients (developers of same project) 37
3.6 Correlation Coefficients (developers of different projects) 39
3.7 Correlation Coefficients (developers of same language) 40
3.8 Bug Latency (Average Values) 41
3.9 Bug Latency Correlation Values between Language Constructs 41

4.1 Number of Source Files and Total LOC 49
4.2 Results of the rank-sum test 59

5.1 Statistics of Projects 64
5.2 Measurement Types 66
5.3 Point biserial correlation between hunk metrics and hunk type 74
5.4 Precision P, Recall R and Accuracy A using LR with PCA 75
5.5 Precision P, Recall R and Accuracy A using LR without PCA 75
5.6 Results of Multivariate Logistic Regression (a) 76
5.7 Results of Multivariate Logistic Regression (b) 77
5.8 Precision P, Recall R and Accuracy A using random forests 77
5.9 Precision , Recall and Accuracy for Mozilla using individual metrics . 80
5.10 Precision , Recall and Accuracy for Eclipse using individual metrics . . 80
5.11 Precision , Recall and Accuracy for Mozilla using metrics groups . . . 82
5.12 Precision , Recall and Accuracy for Eclipse using metrics groups . . . 82

5.13 Classification accuracies using models from a different project 84

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)