6-26-2023 2:00 PM

A Quantitative Analysis Between Software Quality Posture and Bug-fixing Commit

Rongji He, *Western University*

Supervisor: Konstantinos Kontogiannis, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science

© Rongji He 2023

Abstract

Software quality assessment and prediction has been a research hotspot and has become even more critical in continuous software engineering. Modifications to a software product developed following a continuous software engineering process typically commence as a sequence of frequent commits, following a philosophy of “commit small, commit often.” Continuous integration (CI) and continuous deployment (CD) are essential concepts in this development environment. The challenge then is to develop techniques and tools which allow the development team to assess the overall quality posture of a software module in the period from a bug-inducing commit (i.e., when a bug is reported) to a bug-fixing commit (i.e. when a bug is reported fixed. The hypothesis is that in this period, the quality posture of the software modules involved in a bug-inducing/bug-fixing commit pair undergoes changes which may give developers insights that a bug-fixing commit is not only within reach but also the overall quality posture of the system is improving. In this thesis, we perform a quantitative analysis of how the posture of a software module changes and whether those changes follow a pattern that can be used as a predictor for an imminent bug-fixing commit. In this thesis, the posture of a module is denoted by a vector of metrics values computed from the source code and from information extracted from GitHub and Bugzilla repositories. The results indicate that a considerable number of bug-fixing commits in many software projects is preceded by a typical posture, and the occurrences of some posture combinations are more likely than others to be succeeded by a bug-fixing commit.

Keywords: Software quality, continuous software engineering, process metrics, quantitative analysis

Summary for Lay Audience

Software development teams always pursue high-quality software, yet quality as a concept is multifaceted and hard to measure. Software bugs are significant causes of system failure and poor quality. While most of the work in this field relies on Machine Learning and software metrics to predict and fix potential quality-influencing bugs in advance, more effort must be made to understand when the actual bugs are fixed, representing the point where product quality is restored. In this thesis, we examine first whether there are any commonly occurring noticeable patterns which manifest an unhealthy system posture and second, whether the manifestations of these patterns are more probable to be followed by fixes of bugs. The findings in the thesis lead to a better understanding of quality restoration, which is an essential part of the software quality profile.

Acknowledgments

This thesis is first dedicated to the memory of my father, Jian He, who embraced me constantly with his great love, trust, and respect. Dad, you don’t know how much I miss you now and how remorseful I am for not telling you enough that I loved you when I had a chance. You are gone, but I will proceed with your belief.

I want to express my sincere gratitude to my supervisor, Dr. Kostas Kontogiannis, as the thesis would not have been possible without his support. Sometimes my tell-me-what-to-do-so that-I-can-finish-my-Master attitude makes this journey rugged enough, but his encouragement, inspiration, and patience guide me to the destination.

I am thankful and forever in debt to my girlfriend and soulmate, Heyao Bai, for her unwavering trust in me. Life is harsh, but with her presence, I fear no more.

I want to thank Marios Stavros Grigoriou, Ria Ria, and the Department of Computer Science, Western University, for their valuable help during my thesis.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ANN	artificial neural network
API	application programming interface
AUC	area under the curve
BFC	bug fixing commit
BIC	bug-inducing commit
CBO	coupling between object class
CC	cyclomatic complexity
CK	Chidamber and Kemerer metric suite
CSE	continuous software engineering
DIT	depth of inheritance tree
FP	function points
GR	gain ratio
IG	information gain
ITS	issue tracking system
KLOC	thousands of lines of code
KSLOC	thousands of source lines of code
LCOM	lack of cohesion in method
LOC	lines of code
LSSVM	least squares support vector machine
MI	maintainability index
ML	machine learning
NASA	national aeronautics and space administration
NIST	national institute of standards and technology
NOC	number of children
OO	object-oriented
P-BFC	pre-bug-fixing commit
PCC	Pearson correlation coefficients
REST	representational state transfer
SFP	software fault prediction
SLOC	source lines of code
SPCC	squared Pearson correlation coefficient
SQA	software quality assessment
SU	symmetrical uncertainty
VCC	vulnerability-contributing commits
VCS	version control system
WMC	weighted method count

Chapter 1

1 Introduction

1.1 Preface

Evolved from agile software development methodologies, continuous software engineering (CSE) has emerged as a software development paradigm which has gained significant attention in the last decade. In contrast to traditional release engineering, which is dominated by concentrated but sparse releases, CSE accelerates the software development process by enabling continuous integration and deployment through frequent short, incremental, and iterative development revisions. The wide adoption of CSE has facilitated the intensive use of version control systems (VCS) such as Git and Mercurial. A release in CSE typically manifests as a series of frequent but small commits. Frequent releases introducing new incremental features bring higher customer satisfaction but, at the same time, make software quality highly subject to change.

As software continues to be ubiquitous in our lives, techniques to assess the quality of a software system have become increasingly important in software engineering. Stakeholders need to know whether the software they use and depend on maintains high quality. Over the years, software quality has been explored in different facets: reliability, maintainability, security, efficiency, and size. Major organizations such as IEEE, ISO, PMI, NASA, and NIST have all contributed to definitions and quantifiable software quality measures. This thesis focuses on features that deal with maintainability and ease of evolution as key quality factors. Software failures caused by faults (i.e. bugs) lead to degradation of the user-perceived quality [1]. Over the past decade, the software engineering community has invested significant resources to develop techniques for identifying error-prone modules using metrics and machine learning. These efforts aimed to assign fault proneness to a given module or predict a bug-inducing commit. Despite many efforts to ensure faults are detected and fixed early, they occur inevitably. In the CSE development process, software quality declines when one or more bugs are inserted into the system in a bug-inducing commit (BIC) and is improved when bugs are fixed in a bug-fixing commit (BFC). In this agile environment, understanding how the quality of a software module in the period of certain relevant commits before a BFC evolves is a way to assess the impact of BIC on the module's quality. However, to date, little effort has been invested in examining the quality changing of a software module in the period between a BIC and a BFC.

Software metrics are well-defined measurements that reveal unique aspects of software and play a crucial role in assessing software quality. Most quantitative models for assessing software quality rely on software metrics. However, the practical applications of metrics are facing several major challenges. First, many software metrics that are used extensively in research were proposed decades ago, and many of them are becoming less suitable to measure software in this era of continuous processes [2] [3] [4]. More advanced software metrics are needed to measure the constantly evolving development process of CSE. Second, esoteric software metrics proposed in academia are seldom used practically in the industry. The industry needs metrics that are both easy to understand and collect to aid managerial decision-making in the software lifecycle.

For this thesis, we investigate a combination of source code and process metrics to define a feature vector that characterizes the current posture of a software module (i.e., a file or a package). As each metric measures particular aspect of the software, a combination of metrics describes a more detailed situation which can be used to model the software. We, therefore, define a *posture* in the CSE development process as a unique combination of several process and source code metrics. Furthermore, much research [5] [6] [7] [8] [9] [10] advocates that metrics with extreme values are manifestations of defects. In this thesis, we model software quality by capturing extreme values. A software metric is considered *abnormal* if it exhibits a very high or very low value in a software project's history. An *abnormal posture* of a module in a commit is defined as a posture where all metrics have *abnormal* values at the same time. A fault, which caused an observed bug, is fixed by one or more *bug-fixing commits (BFCs)*, and a set of commits before a BFC that might contain an *abnormal posture* that leads to this BFC are named *pre-bug-fixing commits (P-BFCs)*.

This thesis has three major objectives. The first objective is to investigate a combination of source code and process metrics which can be extracted from reconciled GitHub and Bugzilla records and can be used to define the posture (i.e., the quality profile) of a module in a given commit. The second objective is to design and implement a technique which identifies patterns of postures (i.e., patterns of metrics) which frequently occur as the quality of the system is improved from the point in time a bug-inducing commit low-quality state) occurs to the time a bug-fixing commit (higher quality state) occurs. The third objective is to conduct this analysis under different scenarios and identify whether the selection of pre-bug-fixing (P-BFC) commits plays a role in the analysis.

In our analysis, we are interested in examining a) the frequency (total occurrence) of each *posture* and b) how likely (i.e. the rate of the frequency) a *posture* occurs in *P-BFCs*. The technique can be summarized into four main steps. In the first step, we identify the BFCs and BFC-related files by reconciling commit data from GitHub and project bug information from Bugzilla. Several software process metrics are extracted from reconciled data, and we also propose a set of new process metrics based on old ones, seeking improvements from them. In the second step, we identify P-BFCs through the different strategies we proposed for this thesis. In the third step, the metric values are converted into four categorical groups to study the abnormal behaviours, namely the *very low, low, high,* and *very high* groups. In the fourth step, we leverage conditional frequency analyses to answer two research questions: a) given the occurrence of BFCs, what are the most frequent *posture* patterns that occurred commonly before them? and b) what is the likelihood of having BFC given a *posture* pattern occurring in P-BFCs? We use various project characteristics to explain and understand the distinct results afterwards. The answers to these two questions can provide insights to developers that the posture of their system leading towards a BFC.

1.2 Problem Description and Approach Outline

As discussed above, the development process of a software project which conforms with CSE processes is manifested by a sequence of frequent but small commits. In this development paradigm, each commit consists of small modifications adding very specific features to the system. These modifications are usually made to selected specific files of the project.

For our work, we assume a software system *S* which in its lifecycle produces a time-ordered sequence of commits $\mathcal{C}_s = [C_1, C_2, C_3, \ldots, C_n]$, where for any two commits of C_s, C_m and C_n, if $m<n$ then C_m is committed before C_n. Modern Version Control Systems like GitHub capture and record valuable information along with each commit related to the number of lines added, deleted, and modified. Similarly, metrics tools can capture other valuable metrics related to the committed files, such as cyclomatic complexity and information flow. In addition to the above process metrics, several other aggregate metrics can also be computed, such as the commit frequency of a file, the average commit size, etc. Assuming that each commit contains several files $\{f_1, \ldots f_i\}$, then commit $C_k \in \mathcal{C}_s$ can be represented by a matrix as follows:

$$A_{C_k} = \begin{bmatrix} f_1 & a_{f_1,m_1} & a_{f_1,m_2} & \cdots & a_{f_1,m_j} \\ f_2 & a_{f_2,m_1} & a_{f_2,m_2} & \cdots & a_{f_2,m_j} \\ f_3 & a_{f_3,m_1} & a_{f_3,m_2} & \cdots & a_{f_3,m_j} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ f_i & a_{f_i,m_1} & a_{f_i,m_2} & \cdots & a_{f_i,m_j} \end{bmatrix}$$

where f_1 to f_i are files participate in commit C_k. In this context, each row of the matrix A_{C_k} represents a vector of metrics associated with a file contained in the commit. Using the bug report of the software product *S* from a bug tracking system like Bugzilla, through a temporal reconciliation process, we are able to identify which commits fix a bug (hence a bug-fixing commit, BFC) and, more importantly, which files of a bug-fixing commit may be related to a fix, since not all files of a bug-fixing commit are necessarily related a fix. We label each file in each commit with a binary label *is_bug_fixing*, indicating whether the change made to the file in this commit is for fixing a bug (True) or not (False). In our experiment, if at least one file in a commit has an *is_bug_fixing* label value set to *true*, then the containing commit is treated as a *bug-fixing commit* (BFC). Having identified all BFCs, we carry on the study by identifying a set of commits that occurred before each BFC (hence a pre-bug-fixing commit, PBFC).

The problem statement can then be formalized as follows:

Let *S* be a software system with commits [C_1, C_2, …. C_n], where each commit C_i contains the files $f_{i,1}$, $f_{i,2}$, $f_{i,m}$ and each such file $f_{i,j}$ is modelled as a vector of metrics $V_{i,j}$ we refer to as the posture of the file $f_{i,j}$. Let also *P* be a reconciliation process between GitHub and Bugzilla records which tags a file in a bug-fixing commit as the file being the bug-fixing related file. For each bug-fixing commit, let us define an interval *I*, which refers to commits which relate to the bug-fixing commit (i.e., contain similar files). The objective is to conduct an analysis whereby we can identify commonly occurring patterns of file postures which appear in a period before a bug-fixing commit (i.e., within the interval *I*) so that we can reason as to how a file gradually moves from a low-quality posture (the bug-inducing commit) to a higher quality posture (i.e., the bug-fixing commit).

In this respect, two analyses are conducted:

- the first is to examine the conditional frequency rate (CFR): ***CFR (posture pattern | BFC),*** and
- the second is to examine the conditional frequency rate: ***CFR (BFC | posture pattern)***.

1.3 Thesis Contributions

This thesis contributes to software quality assessment in continuous software engineering processes by proposing the following:

1. A collection of source code and process metrics extracted from GitHub and Bugzilla repositories can be used to define the quality posture of a file in a commit.

2. An analysis method to identify the likelihood of patterns which may relate to bug-fixing commits (Analysis 1) and the likelihood of a bug-fixing commit given a posture pattern (Analysis 2).

3. An analysis to examine whether the type of projects and the selection of interval I (pre-bug-fixing commit period) affect the obtained results (See Strategy 1, 2, and 3 in Chapters 5 and 6).

1.4 Thesis Outline

This thesis is organized as follows. In Chapter 2, we provide research works and background knowledge relevant to this thesis. Chapter 3 describes the acquisition process of the data used in our experiment. Chapter 4 discusses how the proposed software metrics are computed. In Chapter 5, the proposed quality assessment technique is detailed. We present the experiment and results in Chapter 6. Finally, conclusions and future directions of the thesis are provided in Chapter 7.

Chapter 2

2 Background and Related Work

2.1 Software Quality Assessment

Software is penetrating and reshaping almost every aspect of our life now. The thriving growth of the software industry has made software quality essential for remaining competitive in the market. However, software quality is a vague and abstract concept, and various approaches have been proposed over the years for measuring software quality. Boehm [11] and McCall [12] presented two of the earliest software quality models, defining a set of software quality attributes and sub-attributes organized hierarchically. These early quality models led to formalizing global standards for evaluating software quality. The ISO/IEC 9126 quality reference model and its successor ISO/IEC 25010, were defined in 1991 and 2011, respectively. The SQA models mentioned above are still relatively abstract; that is, the quality attributes proposed are still high-level and often can’t be directly used to ascertain detailed quantitative assessments for a specific system under analysis. Coleman et al.’s *maintainability index* (MI) [13] was one of the first efforts to address the drawback. In their work, they focus on maintainability, a major quality attribute. They assess maintainability by a four-metric polynomial function extracted from 50 regression models, where each metric is adjusted by a weighting coefficient. Similarly, Bansiya and Davis extend Coleman et al.’s work to compute all quality attributes using metric-based polynomial equations [14]. The major research trend today is to employ machine (ML) learning techniques to achieve better quality assessment or prediction results [15] [16].

2.2 Software Fault Prediction

When the complexity of software increases over the years due to prolonged maintenance and evolution, its fault proneness increases as well. According to IEEE standard classification for software anomalies, software failure is defined as the deviation of the observed behaviour from the specified expected one and can be caused by one or more faults, defined as imperfections or deficiencies in the product which do not fulfill its requirements [17]. In order to sustain good quality, software fault prediction (SFP) has been a research hotspot. SFP involves the use of various techniques to detect pre-existed fault-prone components in advance.

However, software as a complex artifact can be hard to measure, model, or justify. Software metrics have been proposed for this reason. Many SFP models or techniques rely upon software metrics covering different system facets of a software system (i.e. structure, run-time behaviour, data flow etc.). The two main types of SFP models are regression and classification. Classification techniques aim at classifying software components into faulty or non-faulty classes, whereas the objective of a regression-based model is to predict the number of faults within a module.

2.3 Software Metrics

Software metrics are imperative building blocks for any software analytical tools, so this section is dedicated to providing comprehensive background information and related works done in the joint domain of SFP, ML, and metrics. According to the IEEE "Standard of Software Quality Metrics Methodology," a software metric is a function that takes software data as input and outputs numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality [18]. In general, software metric provides quantitative measurements of certain aspects of a software product or process, and these informative measurements can be precious guidance in decision-making.

Based on the studied literature, software metrics currently lack a clear and universal classification. Different names referred to the same classification were used across most of the papers. However, two widely agreed classes for software metrics are product metrics and process metrics [19] [20] [21]. Product metrics are metrics extracted from attributes of the final developed product, providing a comprehensive understanding of the system's size and complexity [19] [21]. Product metrics can be further classified into static metrics and dynamic metrics. Static metrics can be acquired from code, whereas dynamic metrics are based on the running of software programs. Process metrics deal with features collected during the software development lifecycle [20]. In general, process metrics scale when software evolves and gains in size.

Figure 1 gives a broad classification of software metrics, and we will present examples of proposed metrics in some classes in the following sections. We will elaborate on process

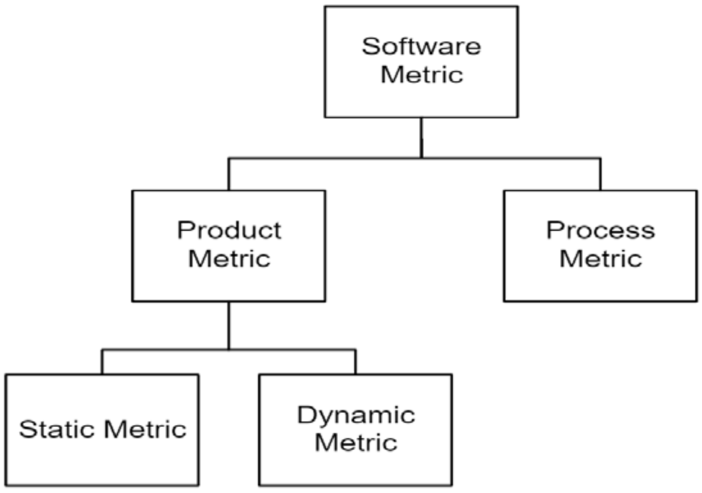


Figure 1: Metric Classification

metrics since they are the underlying metric in this thesis and will be brief on less relevant ones.

2.3.1 Static Metrics

Size Metrics are among the earliest and most widely used static metrics which measure the size of the software product. Size metric is naturally a good indicator of programmer productivity. Some obvious advantages of size metrics are intuitive, well-defined, and easy to compute [21] [22]. However, it cannot be measured in the early stage of the coding process [21]. One typical example of a size metric is the *line of code* (LOC), which measures the number of lines in code files of software. The results from a recent systematic literature review of software quality assessment or prediction models [23] [24] suggest that LOC is chosen as a primary metric in most studies. In this thesis, LOC is used in combination with other process metrics. A few refined LOC variants have been developed throughout its long history; for example, *source lines of code* (SLOC), sometimes called the *logic line of code,* is LOC that excludes comment lines, empty lines, and syntactic symbols that have no logical meaning such as the curly brackets in Java, and *Kilo-SLOC* (KSLOC), as its name suggests, often used in large-sized software projects. LOC-based size metrics suffer from certain drawbacks [14] [15] [16] [17], some of which are listed below:

- Language Depending: programs written in different programming languages that serve the same functionalities tend to have different LOC.
- Developer Variation: senior developers tend to use less LOC than junior ones when implementing the same logic.
- Lack of Universal Counting Standards: How should a statement spanned over several lines be counted? Different counting standards result in different LOC.

Function Points (FP) is another significant size metric proposed shortly after LOC by Albrecht in [25]. FP calculates the volume of functionality delivered by a software module. To compute the Function Points metric, the number of inputs, outputs, inquiries, internal files, and external interfaces to other components are taken into consideration. Quality metrics such as *defects per FP* and *defects per SLOC* are derived from FP and SLOC, respectively, and are important measures of defect density.

LOC, as part of SFP, has been in active studies constantly. LOC’s performance in predicting software defects has been proven to be both satisfactory and unsatisfactory by some researchers. Fenton and Ohlsson’s experiments [26] show that if software modules are ordered by LOC descendingly, most of the top-ranked modules also stay in similar positions when they are ordered by the number of defects. Fenton and Ohlsson’s conclusion is confirmed by Zhang in his investigation of the relationships between LOC and defects [9]. He concluded that “larger modules tend to have more defects,” and a small number of the largest modules in the system are responsible for a large share of total defects. However, both Fenton [27] and Rosenberg [28] found that size metrics are insufficient to support a predicting model alone, and they are better to be used together with other metrics.

Complexity Metric is another type of static metric. A well-known one is McCabe's *cyclomatic complexity (*CC) proposed in [29]. CC measures the number of linearly independent paths generated by the control flow statement [30]. He decomposed the software program into a control flow graph of connected and directed procedures, then the complexity of the program is the linearly independent paths in the graph which can be calculated using the following equation:

$$C = e - n + 2p \tag{2.4}$$

Where *e* is the total number of edges connecting nodes in the graph, *n* is the node number, and *p* is the number of uniquely connected graph components. The System's maintainability is tightly bound to its code complexity [19]. The Cyclomatic Complexity metric offers an explicit measure of a system's complexity that no metric could at the time it was proposed, and according to a recent study [31], it is still one of the top 10 mostly used software metrics in research.

Object-Oriented Metrics (OO metrics) reflect properties of code developed using object-oriented languages, a famous one of which is the *CK metrics suite* proposed by Chidamber and Kemerer in 1994 [32], including *Number of Children* (NOC), *Coupling between Object class* (CBO), *Depth of Inheritance Tree* (DIT), *Weighted Method Count* (WMC), *Lack of Cohesion in Method* (LOCM), etc. OO metrics also measure a system's complexity, but in comparison with McCabe's cyclomatic complexity that focuses on procedural complexity, OO metrics specifically aim at quantifying higher-level inter-class or inter-method relationships. OO metrics have become one of the most solid choices for measuring OO systems, particularly when OO methodologies are growing rapidly [33] [34].

OO metrics have gone through numerous validations in the context of software quality assurance. Singh et al. [35] conducted an empirical validation of the CK metrics suite through regression and machine learning models using a public NASA dataset. Their results showed CBO and WMC achieved good performance in predicting fault proneness, but DIT, LCOM and NOC mentioned above did poorly. Gyimothy et al. in [36] tested the fault predictability of OO metrics together with LOC on a large open-source software system named *Mozilla*. Their experiment generally reconfirmed Singh's results: CBO is a good predictor of defect-proneness of classes, but DIT and NOC are untrustworthy. Besides, they found that LOC performed well, and they concluded that it is an excellent candidate for a quick defect prediction model.

2.3.2 Software Process Metrics

If the software development process is defined as an ordered sequence of developmental activities performed on an initial build incrementally, then process metrics measure how much the software in its current state varies from previous ones [37] [38]. Software process metrics serve the purpose of improving long-term software quality by improving the process they measure [37]. Process metrics have unmatched advantages over other types of metrics in software quality assessment. First, process metrics can be accessed and maintained more easily than other metrics. Many OO metrics discussed above require extra computation and management resources and thus are unfriendly and unfeasible to software practitioners. Second, process metrics are naturally suitable for evaluating continuously delivered software, whereas OO metrics and complexity metrics are designed to capture software structural importance. Traditionally, software defect prediction research advocates product metrics. However, much recent concrete research has been done to show the great potential of process metrics in software quality prediction [39] [10] [40] [41] [42] [43] [44]. According to surveys[20] [42] [45], most quality prediction techniques are still in favour of product metrics, but process metrics are underestimated.

Table 1 (derived from section 3.2 of [20]) includes the common process metrics suite and the metrics in the suite. Chapter 4.3 presents a full list of software process metrics we used in the experiments, along with descriptions of how they are measured or computed.

Table 1: Process Metric with Examples

Metric Suite Name	**Related Metrics**
Code delta metrics [10]	*“Delta of LOC, Delta of changes”*
Code churn metrics[10]	*“Churned LOC, Deleted LOC, File count, Weeks of churn, Churn count and Files churned.”*
Change metrics[40]	*“Revisions, Refactoring, Bugfixes, Authors, Loc added, Max Loc Added, Ave Loc Added, Loc Deleted, Max Loc Deleted, Ave Loc Deleted, Code churn, Max Code churn, Ave Code churn, Max Changeset, Ave Changeset and Age.”*
Developer based metrics[46]	*“Personal Commit Sequence, Number of Commitments, Number of Unique Modules Revised, Number of Lines Revised, Number of Unique Package Revised, Average Number of Faults Injected by Commit, Number of Developers Revising Module and Lines of Code Revised by Developer”*
Requirement metrics [47]	*“Action, Conditional, Continuance, Imperative, Incomplete, Option, Risk level, Source and Weak phrase”*

Network metrics[48]	*“Betweenness centrality, Closeness centrality, Eigenvector Centrality, Bonacich Power, Structural Holes, Degree centrality and Ego network measure”*

2.3.3 Quality Assessment Using Process Metrics

This section summarizes quality assessment or prediction techniques relying mainly on process metrics.

Meneely et al. [5] examined software quality by exploring what properties vulnerability-contributing commits (VCC) commonly possess. Three types of process metrics were used in the study of the Apache HTTP Server project: *code churn*, *relative code churn*, and *30-day code churn*. The authors identified 124 VCC spanning 17 years in the project and discovered that VCC has twice as much code churn on average than non-VCCs.

An and Khomh [8] empirically studied Mozilla Firefox commit data to locate crash-inducing commits. Similar to Meneely’s study [5], seven process metrics were employed to compare the characteristics between crash-inducing commits and crash-free commit, and the conclusion was more additions and deletions of LOC are found in crash-inducing commits.

Illes-Seifert and Paech [41] studied the relationship between a file’s historical churn data and its defect count. They selected the *Frequency of Change*, *Distinct Authors*, and *Co-Changed Files* as a part of the subject metrics in the study. They employed statistical techniques and applied them to nine open-source Java projects. Their results indicated that a file’s fault-proneness positively and strongly correlated to the time it has retained in the system, and its change count and number of authors performing the changes are good indicators for defect count.

Nagappan and Ball’s work [10] is one of the first defects predictions tools that only utilized process metrics. Other than the absolute code churn metrics, relative code churn measures were formalized in their work. Statistical regression models were used to discriminate faulty files among 96189 files. After the models achieved an 89% accuracy rate, they concluded relative code churn could be valuable in fault-prone file prediction.

Nagappan et al. [40] studied how coarse grain process metrics collected from software development can be used in defect predictors as an extended study [10]. Instead of relying on process metrics in fine granularity, e.g., *Delta of LOC changes,* authors defined a new set of process metrics based on the number of changes (commits) and change bursts (concentrated commits). They applied a regression model based on the metrics set to *Windows Vista* version history data and achieved precision and recall of over 90%, which is deemed extremely satisfactory.

Rhmann et al. [39] aimed to explore the predictive performance of ensemble-based algorithms using process metrics. The hybrid algorithm used in the literature consists of Fuzzy-AdaBost and Logitboost. In addition, other machine learning techniques, including *Random Forest*, *Multilayer Perceptron,* and *J48,* were also used as a comparative experiment. *Android* projects in versions v4-v5 and v2-v5 were used in the experiment. The conclusion drawn from the experiment result is hybrid algorithm outperformed other single-learning ones.

Majumder et al. conducted a large-scale performance comparison between product metrics and process metrics in [34] through four different statistical models using 722,471 commit data mined from 700 GitHub projects. Their results showed that process metrics generate far better recalls and AUCs than product metrics and thus proved the strong predicting force of process metrics.

Nagappan and Ball [44] used simple process churn metrics together with the architectural dependency information to predict post-release failures in the *Windows Server 2003* operating system with a size of 28.3 million LOC comprised of 2075 files. Although only three process churn metrics were used in the study, i.e., *Delta LOC*, *Churn Files*, and *Churn count*, the authors sought the possible combinations of process metrics with other measures.

The statistical results indicated that the performance of the defect-predicting model could be improved significantly with the presence of OO metrics.

2.3.4 Other Software Metrics

In this section, we survey unique metrics that may evolve from but are considered distinctly out of the metric classification discussed above.

Nakamura and Basili [49] observed that architectural variations always accompany frequent software modifications, so they propose a metric that measures architectural changes based on a software property author coined “Structural Distance.” The authors argue that the CK metrics suite cannot capture detailed structure changes. Towards this effort, authors abstracted software structure into a representable graph and then used a graph kernel function to measure the structural distance between structures before and after a release. They also applied the metric to several open-source software projects as an empirical study. They concluded this metric is feasible and efficient.

Chulani et al. [50] developed a metric for managing customer view of software to improve software quality. First, the authors aggregated different service metrics data collected from customers, such as customer satisfaction reports, surveys done through telephone interviewing, and records from the customer support center. The data was then analyzed with the aid of regression tree methodology and finalized into a customizable metric set that visually reflects customer view.

Washizaki et al. [51] defined a metric suite for measuring the reusability of black-box software components. The suite encompasses five metrics: existence of meta-information, rate of component observability, rate of component customizability, self-completeness of component’s return value, and self-completeness of component’s parameter. These metrics undergo a refined process, including correlation analysis and metrics combination, before being used to evaluate components' reusability. The information required to compute these metrics can be acquired in the absence of source codes of the software; thus, the suite has high practicality.

2.4 Correlations Between Metrics

Since various metrics can be extracted from the same piece of software, some of them may be highly correlated with others. Many research papers have reported concrete evidence of correlations among software metrics. Jay et al. explored the relationship between LOC and CC complexity through a large-scale analysis of 1.2 million C, C++ and Java files [52]. They discovered CC and LOC has an almost perfect linear relationship and suggested using LOC in place of CC. Jay's conclusion is reconfirmed when Mamun et al. studied the nature of relationships between different classifications of metrics [53]. A total of 25 code metrics collected from 9572 software revisions were classified into four domains and used as the experimental basis. They found software size metric tends to have a strong correlation with other metrics, and complexity metrics are more correlated with size metrics than themselves. However, a more recent study conducted by Afriyie and Labiche argued the opposite [30]. The authors observed that the correlation between LOC and CC is also impacted by the type of code (application code or test code) and the type of software (open-sourced or commercial). They found LOC and CC exhibit almost no correlation in test code and advocated for the use of CC over LOC when applicable.

As the above studies have evidently demonstrated, correlations between metrics have also been taken into consideration in this thesis. We present one of the most widely adopted correlations, termed Pearson Correlation Coefficients (PCC), which we used to study metric correlations.

2.4.1 Pearson Correlation Coefficient

PCC is a statistical similarity metric that measures the strength and direction of a linear correlation between two random variables [54] [55]. Given two random variables, *a* and *b*, the formula is [54]-[56]:

$$\rho(a,b) = \frac{cov(ab)}{\sigma_a \sigma_b} = \frac{E((a-\mu_a)(b-\mu_b))}{\sigma_a \sigma_b} \tag{2.5}$$

where $cov(ab)$ is the covariance between $a\ and\ b$, σ_a and σ_{ba} are the e standard deviations of a and b, and $E(a)$ is the mean of a. The squared Pearson correlation coefficient (SPCC) is more convenient [42]-[44]:

$$\rho^2(a,b) = \frac{E^2(ab)}{\sigma_a^2 \sigma_b^2}\ , \tag{2.6}$$

In the context of this thesis, we work with two sample vectors and use the SPCC between two random vectors [42] [44]. First, given two random vectors of length L

$$a = [a_1, a_2 \dots a_L]^T,$$

$$b = [b_1, b_2 \dots b_L]^T.$$

The SPCC between a and b [42] [44] is:

$$\rho^2(a,b) = \frac{E^2(a^T b)}{E(a^T a)E(b^T b)}\ , \tag{2.7}$$

One of the most important properties of the PCC is:

$$-1 \leq \rho(a,b) \leq 1. \tag{2.8}$$

If $\rho(a,b) = -1$, a and b are said to have perfect negative correlation; if $\rho(a,b) = 0$, a and b are said to have no correlation, and if $\rho(a,b) = 1$, there is a perfect positive correlation between a and b.

2.5 Software Metric Selection

Software metrics are designed for different scenarios, and therefore they contribute to the performance of predictive models in different ways. A rigorous metric (i.e. feature) selection procedure is required before the use of metrics in order to reduce superfluous noise and computational complexity. The purpose of a metric selection strategy is to identify candidates of available metrics that are most relevant and suitable regarding the problem domain. Metric selection techniques can be broadly classified into two main groups, ranking and wrapper strategies [57]. Ranking-based strategies rank the metrics according to their calculated statistical scores, while wrapper strategies utilize machine learning algorithms to select metrics that yield the best accuracy. Four of the most commonly used feature-selecting methods are correlation-based (CO), information gain (IG), gain ratio (GR), symmetrical uncertainty (SU), and Relief [57]. A survey of related works is given below.

Kumar et al. [58] built a fault prediction model using Least Squares Support Vector Machine (LSSVM). A total of 10 common feature selection methods were applied to 20 object-oriented metrics to find the most effective subset. Principal component analysis was applied to reduce the dimensionality of massive data. The selected features were then fed to 3 machine-learning prediction algorithms. The results showed that there wasn’t a significant difference among the ten feature selection methods.

Ji et al. [59] proposed a refined Naive Bayes classifier capable of predicting faulty-prone components when feature data is not normally distributed. They considered six feature ranking methods to select the most suitable ones from 21 features. The classifier used information gain as the feature selection had the highest F-measure but was not significant over others. The experiments were conducted on ten software project data in three different programming languages obtained from the PROMISE repository.

The related works presented in the previous sections have several limitations. First, to find the most suitable feature set for a given type of analysis, various feature selection techniques must be applied exhaustively. More computational resources are required to support the brute-force attempts, making the approach less actionable for practitioners. Second, the results from these studies are bounded by the choices of datasets, models, and available metrics. These studies have a limited degree of reusability as no evidence is provided on whether their conclusions can be generalized to other similar works or not. Third, feature selection strategies didn’t play an important role in the experiments. One reasonable inference is that most of these strategies are initially proposed in statistical (machine learning) background and are not designed specifically for software quality assessment or prediction.

2.6 Survey of Software Quality Assessment and Prediction Techniques

Machine learning techniques have been applied extensively in software quality assessment and prediction. We have presented some fault prediction techniques developed with the usage of ML in previous chapters focusing on the software metrics side. In this section, we survey state-of-art quality assessment techniques developed after 2003.

Kumar and Rath [15] built a maintainability prediction model using a hybrid neural network and fuzzy logic approach, where maintainability is defined as the LOC additions and deletions throughout the maintenance period. In their work, fuzzy numbers were input to neurons in the network, and the weight factor was trained through backpropagation based on fuzzy logic. The proposed method obtained better performance in terms of mean absolute relative error in comparison with other quality prediction models.

Hindle et al. [6] explored the impact on software quality brought by commits of different sizes, especially large commits. They started with classifying commits into categories based on changes they made to quality, two important ones of which are: *corrective* commits---commits that fix bugs, and *perfective* commits---commits that enhance a project’s performance or improve processing inefficiency. By conducting a quantitative analysis of the proportional distribution of each category in 9 open-source projects, the authors found that large commits tend to be *perfective* changes than *corrective*, while small commits are often *corrective* than *perfective*. One drawback of their study was the classification of commits is done manually; therefore, only a small proportion of the total commit is inspected. We present an automatic way of determining commit nature in Chapter 3.5.

Zhi et al. [60] designed a quality assessment tool for Alibaba's Business Software Ecosystem based on four key aspects of source code: coding convention, code duplication, complexity, and object-oriented design. Various metrics were collected to evaluate these four aspects in a threshold-rating approach. The proposed tool has been deployed to monitor over 60 core software systems of Alibaba.

Fan et al. [61] proposed a defect prediction framework through the application of a recurrent neural network. Their work initially involved parsing targeted programs into syntax trees. The syntax trees were then converted into a mapping dictionary, and the vector form of the dictionary was fed to a recurrent neural network, allowing it to learn the syntactic and semantic features of the program. An extra layer, referred to as the “attention layer,” was employed to aggregate the critical output of the recurrent neural network into a vector which will be used in the final prediction. Results indicated their framework has an acceptable F1 measure.

Boucher and Badri [62] attempted to address fault-prone functions in software systems using an unsupervised version HySOM model proposed by Abaei et al. in [63]. The HySOM model is an ML model that relies entirely on function-level source code metrics for faulty function prediction. However, the authors managed to adapt the function with class-level object-oriented metrics as they believed such adaption produces better performance. Twelve public datasets were selected in the empirical study, and the results were compared to RF, ANN, and Naive Bayes Network model. The proposed method demonstrated better performance than other ML techniques.

Pandey et al. [64] proposed a two-staged defect prediction framework that addresses the class imbalance issue that exists in many software projects. In the first stage, a staked denoising auto-encoder was used to accurately extract general software metric data from historical datasets, and the data underwent a deep learning phase shortly after to avoid class imbalance. The deep representation of the metric data was then used in the ensemble learning comprised of ten different classifiers in the next stage. Many of the employed classifiers in this line of related work have also been discussed in Chapter 2.3.1. Though the results are not supportive of the excellence of the designed model, the model showed its potential to reduce the over-fitting problem.

Wang and Zhang are the first to utilize the deep-learning neural network encoder-decoder to predict the number of faults [65]. The model they built has a sophisticated deeper layer which is capable of capturing training characteristics. A total of 14 historical fault data sets were chosen as data basis, and the prediction results were in comparison with other mainstream ML models. The authors concluded from the results that the proposed model has suitable performance.

2.7 Other Technical Information

2.7.1 GitHub

GitHub is an online collaboration platform for software development team building on top of *Git*, a famous version control system. The collaboration is enabled through a mechanism named branch. Each project repository has one master branch, and each developer can create and work on their private branch derived from the main branch. Modifications made on a private branch will not be reflected on the main branch until they are accepted by authorities and merged into the main branch. Basic GitHub operations are:

1. commit: make changes to the local repository.
2. push: update the remote repository with respect to the local one.
3. pull: update the local repository with respect to the remote one.

GitHub has profound meaning in continuous software engineering and open-source software development. Thousands of cool open-source software are made available to the public through GitHub. As of July 2020, GitHub has over 83 million users and 200 million repositories [66].

2.7.2 Bugzilla

To efficiently document and resolve the software fault that resulted in a failure, an *issue tracking system* (ITS) such as Bugzilla or Jira is commonly used by software companies. Bugzilla is an open-source bug-tracking system developed by Mozilla in 1998 and has been under active development since then. A lot of useful features have been added to Bugzilla over time, and besides the initial purpose it served, it is now an aggregated web-based software team management system to track project issues, assign tasks, manage schedules, etc. Though Bugzilla is free, it has strict security protection and high customizability, justifying why it is trusted and used by thousands of companies around the world.

2.7.3 Tablesaw Table

Tablesaw is designed to fill the vacuum of the data analysis framework in Java. Tablesaw is employed as the core data storage and manipulation tool in the thesis. Like Python counterpart, *pandas*, Tablesaw stores data in a tabular data structure called a dataframe which supports a wide variety of built-in operations.

Chapter 3

3 Data Modelling

3.1 Metric Data Acquisition Overview

This chapter discusses the metrics acquisition process, which is adapted from the one presented in [67] and [68]. As depicted in Figure 2, we start by selecting suitable software projects based on the criteria described in Chapter 3.2. Since this thesis aims to quantify relationships between the metric-based patterns and the bug-fixing commits, we consider GitHub as the source repository of metrics and mine bug information from Bugzilla. The data of each selected project is then extracted and reconciled from the corresponding GitHub and Bugzilla repositories through custom-configured readers and analyzers complying with the GitHub and Bugzilla data models as discussed in Chapter 3.3 and Chapter 3.4. As GitHub and Bugzilla are two independent systems, a commit pushed to GitHub which fixes a bug may be reported as *closed* (i.e. fixed) to the corresponding Bugzilla repository after a delay (ranging from a few minutes to a few days). In order to associate the GitHub commit data with the correct corresponding Bugzilla bug reports and resolutions, a reconciliation process is performed on the raw data extracted from both GitHub and Bugzilla, as discussed in Chapter 3.5.

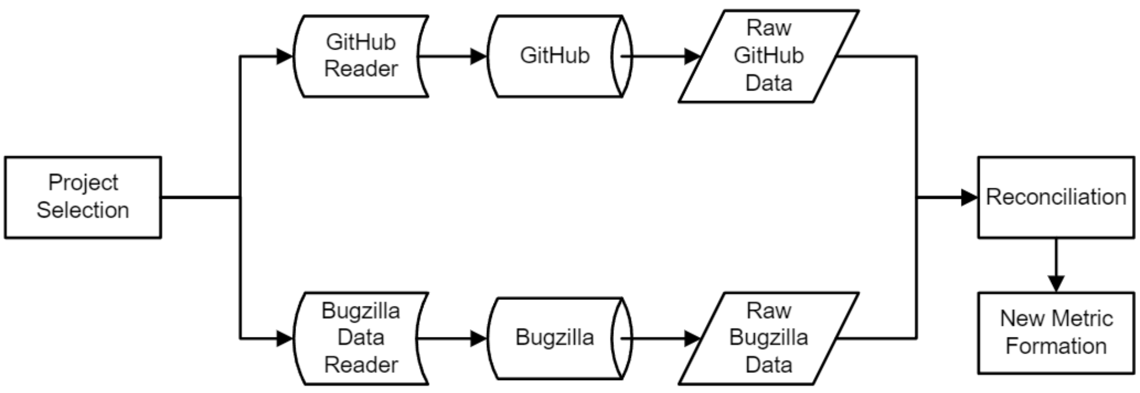


Figure 2: Metric Data Acquisition Stepwise Process

3.2 Selected Projects

A total of 25 open-source software projects are selected based on the following criteria, which are based on recommendations from Kalliamvakou’s GitHub data mining study [69], Korlepara’s data collection guideline [67], and Sarrab’s selection criteria of open source software [70]. These criteria are formulated to improve the representativeness and generalizability of the results [69] [70], but meanwhile, they increase the cost of the selection process and hence constrain the number of available projects.

1. The project must have both publicly accessible GitHub and Bugzilla repositories.

2. The project must have at least five years long active development history. We believe projects with longer histories would have a better chance of containing sufficient regular and bug-fixing commits for this thesis.

3. The project must be related to software that has gathered enough reputation, featured by GitHub stared number.

4. The project must not be collaborated using VCS other than GitHub. This criterion assures the integrity of the data we used for experiments.

A summary of the selected systems and their computed system characteristics is shown in Table 2. Since software continuous engineering is characterized by frequent commits, the table is sorted in descending order of KLOC.

Table 2: Selected Systems and Their Characteristics

Project Name	KLOC	# of commits	BFC Per Commit	# of files	Average Gap Day between Consecutive Commits	Average File Count per Commit	Average Commits per File Participate
kstars	600.44	11044	0.1651	5552	0.5778	11.0438	2.7013
kopete	512.14	16256	0.2411	2668	0.3716	4.1265	3.8375
marble	368.643	13045	0.1493	3474	0.3866	10.656	2.7618
umbrello	201.228	8656	0.3354	1254	0.783	21.3485	1.9953
k3b	164.795	26068	0.2029	898	1.076	15.9129	2.235
kdevplatform	147.218	14540	0.1575	1609	0.2561	15.5384	2.4901
gwenview	103.905	15643	0.1729	639	1.0047	21.9743	1.5906
konversation	92.834	20179	0.2049	349	0.7405	31.5433	1.4269
ktorrent	80.986	10953	0.3054	657	1.7344	12.4416	2.0106
kget	68.556	10980	0.1762	345	2.1546	17.2964	1.6721
kolourpaint	63.244	8345	0.126	412	2.5945	17.9965	1.506
elisa	62.07	5017	0.1947	207	0.744	18.1468	1.3454
plasmanm	58.121	8170	0.1794	418	0.9244	9.8665	1.5747
kmail	57.814	25011	0.2149	631	0.3361	22.9559	2.1472
ark	48.812	8393	0.2232	306	1.8869	24.6056	1.3671
lokalize	40.629	6549	0.3063	229	2.8735	25.7878	1.0326
akregator	40.496	9481	0.2016	400	1.037	10.2554	2.3077
juk	34.155	7342	0.2535	162	2.0583	33.0588	0.9153
clazy	31.844	2018	0.271	658	0.781	10.0617	1.9143
solid	28.918	2847	0.1053	401	3.9482	3.8905	2.4913
korganizer	25.458	10875	0.2093	223	0.755	13.191	2.3702
kmix	13.474	3584	0.2384	188	3.2407	20.8374	1.2482
kompare	10.644	2053	0.2196	68	6.4555	18.4855	0.9712
ktimetracker	8.521	2093	0.1686	112	3.9188	12.7838	1.5755
kontact	5.108	4925	0.1299	46	1.3472	25.052	1.453

In this thesis, we use metrics which are a) extracted directly from GitHub, b) extracted as a result of the reconciliation process, and c) as a result of new computations which aggregate existing metrics into new ones (e.g., *the number of files in each commit*, and the *number of commits* in the project can generate a new *metric average commit size*). We will discuss the compilation of these new metrics later in Chapter 4.

3.3 GitHub Data Model

The GitHub data acquisition process is mainly composed of two stages [67] [68]. The first stage is a straightforward data mining process through a customized extractor built on top of GitHub REST API. Data from stage one is crude because of its nested structure. In the second stage, we studied the GitHub data schema, as depicted in Figure 3, before further operations. GitHub has a commit-file-oriented nature, as discussed early. The process metrics used in the experiment are mainly from the *properties* module presented in Figure 3, and likewise, these metrics are subclassified into *commit-associated metrics* and *file-associated metrics*. The distinction between the two metrics' associated types is important to the categorization procedure in Chapter 5.4. Table 3 lists all metrics acquired after the reconciliation process and their associated types. Fine-grained data is acquired following the schema sequentially while preserving the overall structure. The data extractor and data resolver are implemented with Python.

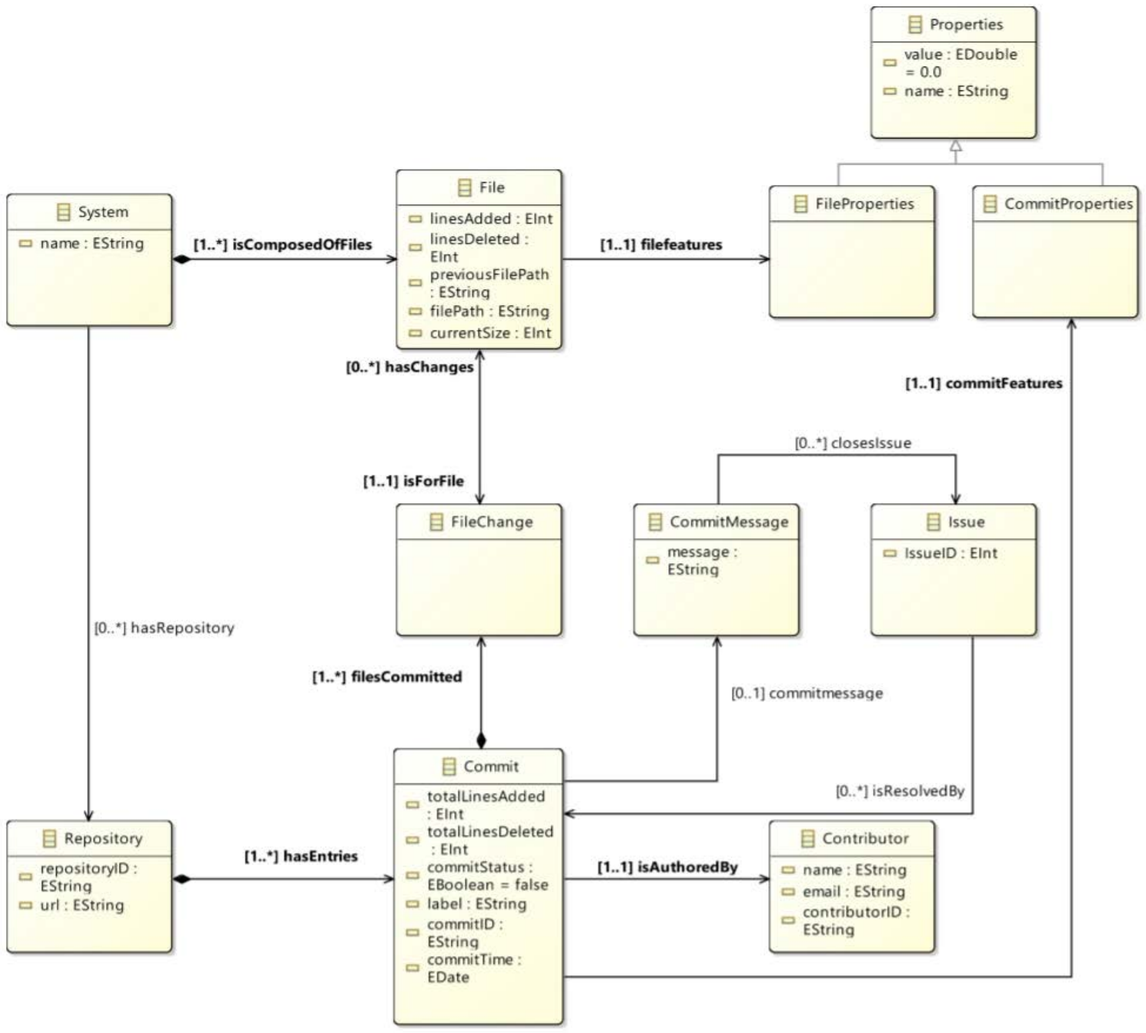


Figure 3: Domain Model for GitHub Data [67] [68]

3.4 Bugzilla Data Model

Bugzilla data was acquired in a similar fashion as GitHub data. Crude data was fetched from the Bugzilla datacenter using a Python extractor through Bugzilla REST API. The Bugzilla

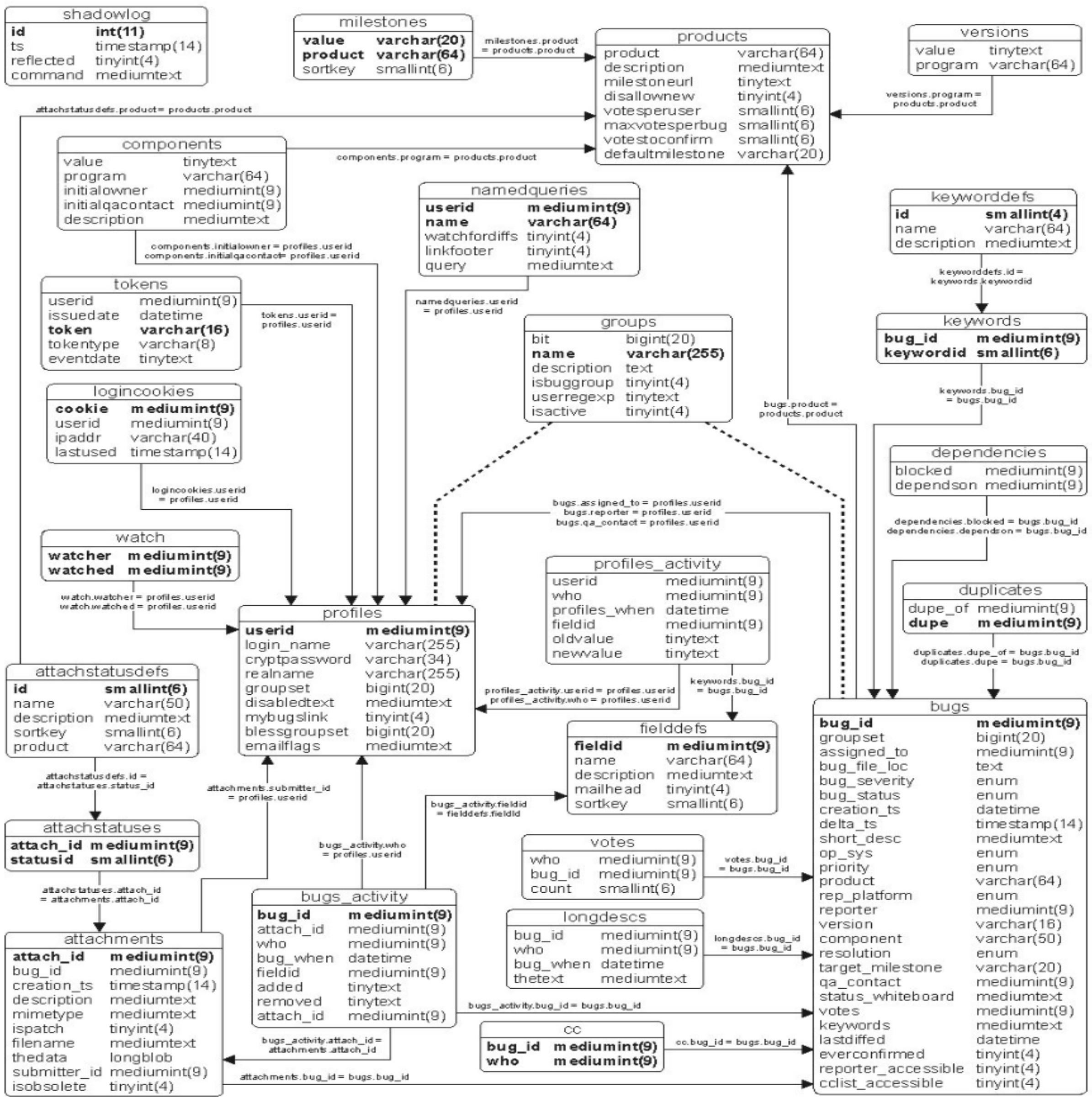


Figure 4: Bugzilla Data Model [69]

data schema is depicted in Figure 4. In this thesis, we are interested in various bug-related field attributes in the *bugs* module, including *bug_id*, *bug_status*, *creation_ts* (local datetime when the bug was created), *resolution*, *short_desc* (a description of the bug), *product*, *reporter*, and *comments* [67]. We use the *bug_status* together with the *resolution* to decide whether the bug has been fixed or not. Comment attribute is practically used to reflect file modification associated with the bug as plaintext or as changelog attachments referencing GitHub. We extracted the list of files when a bug was reported as fixed. Such information will be used in the reconciliation discussed in the next section.

3.5 GitHub and Bugzilla Data Reconciliation

In GitHub, not all files participating in a bug-fixing commit are related to a bug fix. For example, some of these files may be modified due to normal updates or refactoring operations. After crude datasets are populated from GitHub and Bugzilla, a reconciliation process is necessary to identify the exact bug-fixing file. This process is first proposed in [71]. The first task is to arrange all resolved bugs in the same chronological order of GitHub data and group them by bug ID. Next, all GitHub commit dates are iterated until one of them collides with the bug's resolution date. The search scope is narrowed down to a timeframe around this date. In the final step, for each GitHub commit within the window, we examined the number of files being a part of the commit's file list and the bug's file list at the same time, and the one with the greatest number (i.e., the largest intersection) representing the purpose of the commit is bug-fixing.

Formally, let B_k be a bug resolution reported on Bugzilla along with its timestamp t_k and a set of files that resolves the bug $R_k = \{F_k^1, F_k^2, \dots, F_k^j\}$. We then seek all the commits $c_1, c_2, .., c_m$ within timeframe $[t_k - x, t_k + x]$, each of which owes a set of commit files $M_i = \{F_i^1, F_i^2, \dots, F_i^j\}$, where M_1 is the commit files for c_1. Figure 5 depicts this process. The chosen value of x for this thesis is one month. Iteratively, the set of common files between M_i and R_k, or $M_i \cap R_k$, is calculated as $E_k = \{E_1, E_2, \dots E_j\}$. Commit c_b is marked as the bug-fixing commit corresponding to B_k if $E_b = \max(E_k)$, and files in E_b

will also be marked as buggy files. If multiple E_b exist, we select the E_b with the timestamp t_b closest to t_k.

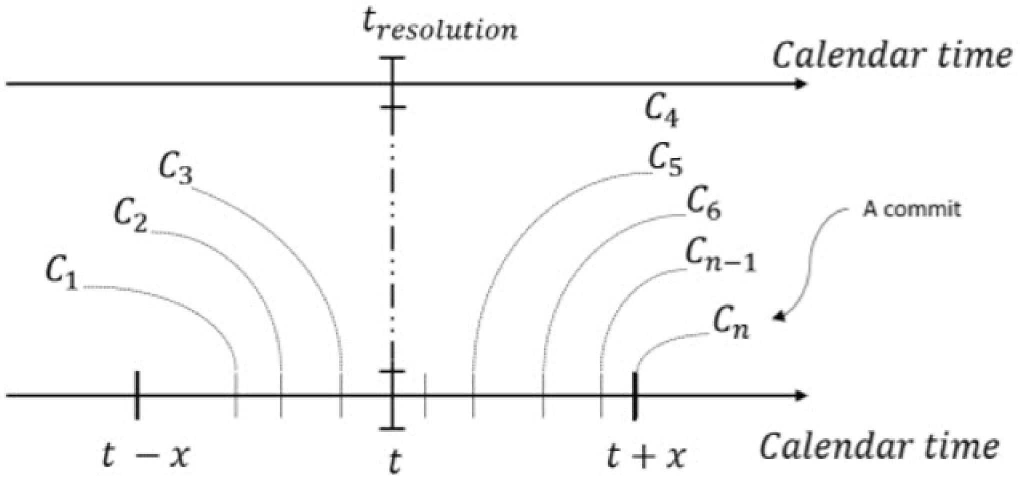


Figure 5: Timeframe Matching in Reconciliation Process. The upper timeline pertains to Bugzilla while the lower to GitHub events [73]

3.6 Metrics Acquired

In this thesis, the metrics selected after GitHub and Bugzilla records are reconciled are: *commit id, branch id, message, parent_ids, author, authored_at, committer, committed_at, commit_additions, commit_deletions, changed_files, is_bug_linked, is_fix_related, is_bug_fixing, is_merge_commit, is_refactoring, file_path, previous_file_path, file_additions, file_deletions, file_id, fractal_value, fractal_value_over_lines, and distinct_authors_to_now.*

Table 3 provides the definitions of these metrics.

For the experiments performed in the next chapter, we temporarily exclude the data that we deemed inappropriate, which includes *branch, message, parents_id, author, authored_at, committer, file_path,* and *previous_file_path.* Corrupted data entry is removed, and the numerical missing is filled with 0.

Table 3: Data Fields and Brief Explanations

Field Name	**Explanation**	**Associated Type**
commit_id	Unique identifier of the commit.	Commit-associated
branch	The name of the branch the commit is committed to.	Commit-associated
message	The message describes a commit written by the committer.	Commit-associated
parents_id	The ID of the commit that this commit is derived from.	Commit-associated
author	The developers who contribute to the modifications in a commit.	Commit-associated
authored_at	The starting time of the commit.	Commit-associated
committed_at	The time when the commit is submitted to GitHub.	Commit-associated
committer	The person who submits the commit.	Commit-associated
commit_additions	Total lines of code added in the commit.	Commit-associated
commit_deletions,	Total lines of code deleted in the commit.	Commit-associated
changed_files	The number of files changed in the commit.	Commit-associated
is_bug_linked	A Boolean value indicates if the link to the bug is provided.	Commit-associated
is_fix_related	A Boolean value indicates if the commit is related to a fix.	Commit-associated
is_bug_fixing	A Boolean value indicates if the commit is for fixing a bug.	Commit-associated
is_merge_commit	A Boolean value indicates if the commit is for merging two branches.	Commit-associated
is_refactoring	A Boolean value indicates if the commit is a refactoring commit.	Commit-associated
file_path	The absolute path of the file in the system.	File-associated
previous_file_path	The absolute path of the file before the commit.	File-associated
file_additions	Total lines of code added to the file.	File-associated
file_deletion	Total lines of code deleted from the file.	File-associated
file_id	Unique identifier of the file.	File-associated
fractal_value	The numerical value measures the diversity of the file contributors.	File-associated
fractal_value_over_lines	The fractal_value over the file’s line number.	File-associated
distinct_authors_to_now	The number of distinct contributors to file currently.	File-associated

Chapter 4

4 Enhanced Evolutionary Aggregated Metric Suite

4.1 Synthesis of New Metrics

For this thesis, in addition to LOC-derived source code metrics, as discussed in Chapter 2, we also use process-derived metrics such as *commit_additions* and *commit_deletions*, as seen in Table 3. More specifically, let $L^{c_{i-1}}$ be the system's lines of code (LOC) before a commit C_i ($L^{c_{i-1}} = 0$ for the first commit), $L^{c_i}_{addition}$ be the LOC addition in commit C_i, and $L^{c_i}_{deletion}$ be the LOC deletion in commit C_i. The current LOC L^{c_i} after C_i can be calculated as:

$$L^{c_i} = L^{c_{i-1}} + L^{c_i}_{addition} - L^{c_i}_{deletion} \qquad \text{Equation 1}$$

Similarly, a file f's current lines of code can be calculated as follows:

$$L^{c_i}_f = L^{c_{i-1}}_f + L^{c_i}_{addition\_f} - L^{c_i}_{deletion\_f} \qquad \text{Equation 2}$$

where $L^{c_{i-1}}_f$ is the file's LOC after the previous commit (C_{i-1}) , $L^{c_i}_{addition\_f}$ and $L^{c_i}_{deletion\_f}$ represent the LOC addition and deletion to f in commit C_i respectively.

4.2 Relative Change Metric Definition

Based on the process metrics discussed in Chapter 2 and in [10] and the product metrics we discussed in the previous chapter, we now define the following new aggregated metrics. They aim to measure the relative (i.e. delta) code changes during the software development process while taking size metrics into consideration. They are proposed to fill the vacuum in process change metrics or to improve flawed old ones. Some of the metrics we defined can be applied to both project and file levels.

4.2.1 Total LOC Delta

The *LOC Delta* proposed in [10] is a measure of absolute change in *LOC*. However, we deem it inadequate to reflect the magnitude of change in a commit. Considering the following example, a project of 500 LOC is modified by a commit with 400 LOC addition and deletion. The absolute delta of LOC is obviously 0 as the addition and deletion cancel out the other; however, either 400 LOC additions or deletions account for 80% of the project size. A considerable amount of change was made in the commit but was unable to be reflected by the absolute *LOC Delta*. Therefore, we defined a refined version of *LOC Delta*, which we coined *Total LOC Delta*. It measures the sum of addition and deletion rather than the absolute difference. Given the LOC addition and deletion in commit C_i, $L_{addition}^{c_i}$ and $L_{deletion}^{c_i}$ respectively, the *Total LOC Delta* in C_i, $L_{total\_delta}^{c_i}$, is defined as:

$$L_{total\_delta}^{c_i} = L_{addition}^{c_i} + L_{deletion}^{c_i} \qquad \text{Equation 3}$$

Similarly, given a file f's LOC addition and deletion in commit C_i, $L_{addition\_f}^{c_i}$ and $L_{deletion\_f}^{c_i}$ respectively, the *Total LOC Delta of* f in C_i, $L_{total\_delta\_f}^{c_i}$, is defined as:

$$L_{total\_delta\_f}^{c_i} = L_{addition\_f}^{c_i} + L_{deletion\_f}^{c_i} \qquad \text{Equation 4}$$

Though LOC addition and deletion both represent churns in code, they may relate to quality postures to a different extent. To investigate the relationship, we adopt the polynomial fashion from the *maintainability index* [13] and apply two weighting coefficients, a and b, to the additions and deletions, respectively, in Equations 4.3 and 4.4. The coefficients are determined through a heuristic process in which pairs of coefficients with different rationales are fed to Analysis 1 (discussed in 5.7.1). Each pair lead to a discovery of a different number of posture patterns. As one objective of this thesis is to address the quantitative relationship between posture patterns and BFCs, we deem a discovery of more patterns lead to a better chance of finding highly related patterns. The coefficient pair that leads to the most posture patterns is chosen. The results of the heuristic process are shown in Table 4.

Table 4: Coefficients and Pattern Discoveries

Coefficients		Rationale	strategy_1	strategy_2	strategy_3
a	b				
1	1	original coefficients	3140.52	2934.28	3865.12
1.2	0.5	increase addition's effect and reduce deletion's	3241.46	3049.17	4024.67
0.5	1.2	reduce addition's effect and increase deletion's	2697.875	2555.79	3429.33
3	1	considerably increase the addition's effect while keeping the deletion original	3178.125	2988.63	3939.83
1	3	considerably increase the deletion's effect while keeping the addition original	2593.50	2449.79	3301.50
1	0.2	considerably reduce the deletion's effect while keeping the addition original	3133.24	2914.83	3850.56
0.2	1	considerably reduce the addition's effect while keeping the deletion original	2411.32	2230.84	3068.64

Coefficients 1.2 and 0.5 lead to the most pattern discovery in all three strategies, so they are selected in this thesis. The revised formulas are:

$$L_{total\_delta}^{c_i} = 1.2 \times L_{addition}^{c_i} + 0.5 \times L_{deletion}^{c_i} \qquad \text{Equation 5}$$

and

$$L_{total\_delta\_f}^{c_i} = 1.2 \times L_{addition\_f}^{c_i} + 0.5 \times L_{deletion\_f}^{c_i} \qquad \text{Equation 6}$$

4.2.2 Total LOC Delta Weight

The *Total LOC Delta* is a direct measurement of change but is not an intuitive one. We defined a new metric *Total LOC Delta Weight*, to measure the weight percentage of *Total LOC Delta* to the project's *LOC*. This metric explains comparatively how much a LOC means to the whole project.

$$W_{total\_delta}^{c_i} = \frac{L_{total\_delta}^{c_i}}{L^{c_{i-1}}} \qquad \text{Equation 7}$$

where $L^{c_i}_{total\_delta}$ is above-mentioned *Total LOC Delta* of a commit C_i and $L^{c_{i-1}}$ is the project's *LOC* of the previous commit of C_i, denoted by commit C_{i-1}.

Similarly, this metric can also be used to measure file-level delta weight:

$$W^{c_i}_{total\_delta\_f} = \frac{L^{c_i}_{total\_delta\_f}}{L^{c_{i-1}}_{f}} \qquad \text{Equation 8}$$

Total LOC Delta Weight as a weight percentage measure is problematic when the size of the project increases constantly in its lifetime. A substantial total delta amount in the early lifespan might become dramatically less insignificant when the project gains tremendous LOC over the years. We develop a compensation mechanism to adjust the weight of later time commits.

For each commit C_i, we compute the *Average Total LOC Delta*, $\bar{x}_i$, of all its previous commits from C_1 to C_{i-1}:

$$\bar{x}_i = \frac{1}{i-1}\sum_{i=1}^{i-1} L^{c_i}_{total\_delta} \qquad \text{Equation 9}$$

We then iterate these previous commits, finding the number of commits that have a *Total LOC Delta* is greater than $\bar{x}_i$. We coin the number of these commits as the *significant commits count*.

$$S^{c_i} = \left|\{k \in \{C_1 \ldots C_{i-1}\}: k \geq \bar{x}_i\}\right| \qquad \text{Equation 10}$$

The *Total LOC Delta Weight* of each commit is compensated by multiplying it with the *significant commits count*. Later commits tend to have larger counts, so they regain significance through the adjusted multiplications.

$$W^{c_i}_{total\_delta} = \frac{L^{c_i}_{total\_delta}}{L^{c_{i-1}}} \times S^{c_i} \qquad \text{Equation 11}$$

When applied the compensation mechanism to a file *f*'s *Total LOC Delta Weight*, all the variables of Equation 4.11 are reduced to file-level metrics.

$$W_{total\_delta\_f}^{c_i} = \frac{L_{total\_delta\_f}^{c_i}}{L_f^{c_{i-1}}} \times S_f^{c_i}$$ Equation 12

4.2.3 File-Project Total LOC Delta Weight

To our best knowledge, commit-level and file-level process metrics that have been proposed so far are disjointed and unrelated. However, they are connected naturally, as any project-level changes are made up of individual file-level changes. Valuable information may be revealed from a metric that links a change in file to the overall change of its containing project. To bridge this gap, we proposed the following metric we termed *File-project LOC Delta weight*.

$$W_{file\_proj\_delta}^{c_i} = \frac{L_{total\_delta\_f}^{c_i}}{L_{total\_delta}^{c_i}}$$ Equation 13

where $L_{total\_delta}^{c_i}$ and $L_{total\_delta\_f}^{c_i}$ are from Equation 5 and Equation 6 respectively.

4.2.4 File-Project LOC Weight

Many previous studies discussed in Chapter 2 have shown that larger files are riskier to be the origin of defects, therefore leading to a Bug Fixing Commit (BFC). However, the question that arises is how large is large. The *file-project LOC weight* percentage provides intuitive project-wise comprehension of how large a file is. This metric can be calculated by dividing a file *f*'s LOC by the project's LOC:

$$W_{file\_proj\_LOC}^{c_i} = \frac{L_f^{c_i}}{L^{c_i}}$$ Equation 14

4.2.5 Total LOC Delta Percent Change

The percent change is a useful measure to describe how much a variable has changed from its previous position in a series. Finding the percent change of code deltas (i.e., the change of change) is helpful in understanding the nature of the software process. Given the LOC delta $L_{total\_delta}^{c_i}$ of commit C_i and the total LOC delta $L_{total\_delta}^{c_{i-1}}$ of its previous commit C_{i-1}, the delta rate of change is calculated as the following:

$$P_{total\_delta}^{c_i} = \frac{L_{total\_delta}^{c_i} - L_{total\_delta}^{c_{i-1}}}{L_{total\_delta}^{c_i}} \qquad \text{Equation 15}$$

Similarly, for file-level delta percent change:

$$P_{total\_delta\_f}^{c_i} = \frac{L_{total\_delta\_f}^{c_i} - L_{total\_delta\_f}^{c_{i-1}}}{L_{total\_delta\_f}^{c_i}} \qquad \text{Equation 16}$$

4.3 Metrics Used in the Experiment

Table 4 lists all the metrics used in our experiment, which will be discussed in Chapter 6 with a sequential numeric index we label them. The Index Number Uniquely References the associated metric. Metrics 1-7 are process metrics listed in Table 3, metrics 9 and 11 are size metrics we synthesize, which have been discussed in Chapter 4.1, and metrics 10 and 12-18 are relative change metrics proposed by us, which have been discussed in Chapter 4.2.

Table 5: Metrics Used in Experiment with Index and Associated Type.

Metric	Short Name	Associated Type
Commit additions	M1	Commit-associated
Commit deletions	M2	Commit-associated
Changed files	M3	Commit-associated

File additions	M4	File-associated
File deletions	M5	File-associated
Fractal value	M6	File-associated
Fractal value over lines	M7	File-associated
Distinct authors to now	M8	File-associated
Project LOC	M9	Commit-associated
Project total LOC delta	M10	Commit-associated
File LOC	M11	File-associated
File total LOC delta	M12	File-associated
Project total LOC delta percent change	M13	Commit-associated
Project total LOC delta weight	M14	Commit-associated
File total LOC delta percent change	M15	File-associated
File total LOC delta weight	M16	File-associated
File-project LOC weight	M17	File-associated
File-project total LOC delta weight	M18	File-associated

4.4 Working Example

We provide a working example to demonstrate how the proposed metrics in this chapter are synthesized or computed from available process metrics in Table 3. In the following example, we omit metrics in Table 3 that are not involved in any metric computing formula in this chapter (i.e., *committed_at*). Consider the following two commits:

commit_1*= {commit_id = c0001, commit_additions= 250, commit_deletions= 0, …*

{file_id=f0001, file_additions= 190, file_deletions = 0, …}

{file_id=f0002, file_additions= 60, file_deletions = 0, …}

}

commit_2*= {commit_id = c0002, commit_additions= 636, commit_deletions= 133, …*

{file_id=f0001, file_additions=468, file_deletions = 72, …}

{file_id=f0002, file_additions= 168, file_deletions = 61, …}

}

Assuming ***commit_1*** and ***commit_2*** are two consecutive commits in which commit_1 is committed before commit_2, they are the first and the second commit of a project, respectively.

***Commit_1*:**

- The product metric *Project LOC*, L^{c0001}, can be calculated using *Equation 1*:

$$L^{c0001} = L^{c0000} + L_{addition}^{c0001} - L_{deletion}^{c0001} = 0 + 250 - 0 = 250$$

- Similarly, the product metric *File LOC*, L_{f0001}^{c0001} and L_{f0002}^{c0001}, for file *f0001* and *f0002*, respectively, after *commit_1*, can be computed using Equation 2:

$$L_{f0001}^{c0001} = L_{f0001}^{c0000} + L_{addition\_f0001}^{c0001} - L_{deletion\_f0001}^{c0001} = 0 + 190 - 0 = 190$$

$$L_{f0002}^{c0001} == L_{f0002}^{c0000} + L_{addition\_f0002}^{c0001} - L_{deletion\_f0002}^{c0001} = 0 + 60 - 0 = 60$$

- The proposed metric *Project total LOC delta*, $L_{total\_delta}^{c_i}$, can be computed using *Equation* 5:

$$L_{total\_delta}^{c0001} = 1.2 \times L_{addition}^{c0001} + 0.5 \times L_{deletion}^{c0001} = 1.2 \times 250 - 0.5 \times 0 = 300$$

- The proposed metric *File total LOC delta*, $L_{total\_delta\_f0001}^{c0001}$ and $L_{total\_delta\_f0002}^{c0001}$ for file *f0001* and *f0002* respectively after *commit_1*, can be computed using *Equation 6*:

$$L_{total\_delta\_f0001}^{c0001} = 1.2 \times L_{addition\_f0001}^{c0001} + 0.5 \times L_{deletion\_f0001}^{c0001} = 1.2 \times 190 - 0.5 \times 0 = 228$$

$$L^{c0001}_{total\_delta\_f0002} = 1.2 \times L^{c0001}_{addition\_f0002} + 0.5 \times L^{c0001}_{deletion\_f0002} = 1.2 \times 60 - 0.5 \times 0 = 72$$

- The proposed metric *Project total LOC delta weight,* $W^{c0001}_{total\_delta}$, can't be computed for *commit_1* as the $L^{c_{i-1}}$ (LOC of previous commit) part in *Equation 11* is undefined due to *commit_1* is the first commit. We will demonstrate the computation of this metric shortly in *commit_2.*

- The proposed metric *File-project total LOC delta weight,* $W^{c0001}_{f0001\_proj\_delta}$ and $W^{c0001}_{f0002\_proj\_delta}$, for file *f0001* and *f0002* respectively after *commit_1*, can be calculated following *Equation 13*:

$$W^{c0001}_{f0001\_proj\_delta} = \frac{L^{c0001}_{total\_delta\_f0001}}{L^{c0001}_{total\_delta}} = \frac{228}{300} = 0.76$$

$$W^{c0001}_{f0002\_proj\_delta} = \frac{L^{c0001}_{total\_delta\_f0002}}{L^{c0001}_{total\_delta}} = \frac{72}{300} = 0.24$$

- The proposed metric *File-project LOC weight,* $W^{c0001}_{f0001\_proj\_LOC}$ and $W^{c0001}_{f0002\_proj\_LOC}$, can be calculated following *Equation 14*:

$$W^{c0001}_{f0001\_proj\_LOC} = \frac{L^{c0001}_{f0001}}{L^{c0001}} = \frac{190}{250} = 0.76$$

$$W^{c0001}_{f0002\_proj\_LOC} = \frac{L^{c0001}_{f0002}}{L^{c0001}} = \frac{60}{250} = 0.24$$

- The proposed metric *project total LOC delta percent change* and *file total LOC delta percent change* can't be computed for *commit_1* as the $L^{c_{i-1}}_{total\_delta}$ (*project total LOC delta* of previous commit) and $L^{c_{i-1}}_{total\_delta\_f}$(*file total LOC delta* of previous commit) part in *Equation 15* and *Equation 16*, respectively, are undefined due to *commit_1* being the first commit. We will demonstrate the computation of these two metrics shortly in *commit_2*.

***Commit_2*:**

In *commit_2*, we omit the metrics which have been shown in *commit_1* and focus on those which haven't been shown.

- To compute the proposed metric Project total LOC delta weight, $W_{total\_delta}^{c0002}$, the weighing factor significant commits count, S^{c0002}, as depicted in *Equation 11*, must be computed first. The significant commits count of commit_2 is the number of past commits with a total LOC delta greater than or equal to the average total LOC delta, $\bar{x}_i$, of all past commits. Commit_1 is the only past commit; hence, the $\bar{x}_i$ value of commit_2 is equal to Commit_1's total LOC delta, 300, and S^{c0002} is 1.

$$W_{total\_delta}^{c0002} = \frac{L_{total\_delta}^{c0002}}{L^{c0001}} \times S^{c0002} = \frac{1.2 \times 636 - 0.5 \times 133}{250} \times 1 = 2.7868$$

- The *file total LOC delta weight*, $W_{total\_delta\_f0001}^{c0002}$ and $W_{total\_delta\_f0002}^{c0002}$, for file *f0001* and *f0002* respectively, can be computed following *Equation 12*:

$$W_{total\_delta\_f0001}^{c0002} = \frac{L_{total\_delta\_f0001}^{c0002}}{L_{f0001}^{c0001}} \times S_{f0001}^{c0002} = \frac{1.2 \times 468 - 0.5 \times 72}{190} \times 1 \approx 2.7663$$

$$W_{total\_delta\_f0002}^{c0002} = \frac{L_{total\_delta\_f0002}^{c0002}}{L_{f0002}^{c0001}} \times S_{f0002}^{c0002} = \frac{1.2 \times 168 - 0.5 \times 61}{60} \times 1 \approx 2.8517$$

- The proposed metric *project total LOC delta percent change* can be computed following *Equation 15*:

$$P_{total\_delta}^{c0002} = \frac{L_{total\_delta}^{c0002} - L_{total\_delta}^{c0001}}{L_{total\_delta}^{c0002}} = \frac{696.7 - 300}{696.7} \approx 0.5693$$

- The *file total LOC delta percent change,* $P^{c0002}_{total\_delta\_f0001}$ and $P^{c0002}_{total\_delta\_f0002}$, can be computed following *Equation 16*:

$$P^{c0002}_{total\_delta\_f0001} = \frac{L^{c0002}_{total\_delta\_f0001} - L^{c0001}_{total\_delta\_f0001}}{L^{c0002}_{total\_delta\_f0001}} = \frac{525.6 - 228}{525.6} \approx 0.5662$$

$$P^{c0002}_{total\_delta\_f0002} = \frac{L^{c0002}_{total\_delta\_f0002} - L^{c0001}_{total\_delta\_f0002}}{L^{c0002}_{total\_delta\_f0002}} = \frac{171.1 - 72}{171.1} \approx 0.5791$$

Chapter 5

5 Quality Posture Assessment in Pre-Bug Fixing Commit Period

5.1 Introduction

This chapter discusses the proposed software quality assessment approach. Adopted from IEEE's definition [17], faults are inserted into the system where incorrect modifications are introduced. An assumption in this thesis is that these incorrect modifications will eventually cause some software metrics to become abnormal, i.e., showing values that are significantly higher or lower than the others, until they are fixed in a bug-fixing commit (BFC), so if a few metrics repeatedly become abnormal in the same commits, it is a strong indication of existing bug, we name such situation as an *abnormal posture*. A software's quality is degraded during the presence of bugs and is upgraded back to its previous level when it is fixed. Fixing a bug in a software project today implies significant costs in resources, time, and money. An *abnormal posture* that repeatedly occurred in commits before BFCs is particularly valuable as it can be considered as a factor that improves quality posture. We refer to those commits as *pre-bug-fixing commits* (P-BFCs). We define numerous ways to identify P-BFCs, and they are not necessarily the immediate past commits of a BFC. In the first part of the approach, we detect all *abnormal postures* in all P-BFCs of each selected project and find out the one with the most occurring rate. The *abnormal posture* found in this part is a key to understanding how a system is reaching a quality resumption point denoted by a BFC. In the second part, we examine all *abnormal postures* that occurred in all commits, then seek to predict that the system states warrant that we are going towards a Bug-Fixing Commit (BFC) by finding the posture (i.e., pattern) as this is defined by the vector of the values of the selected metrics in pre-BFC (P-BFC) commits. We enhance the prediction performance by combining postures which predict different distinct BFCs. The *abnormal posture* found in this part can guide better decision-making and resource management.

5.2 Overall Process

The workflow of the proposed analysis process is depicted in Figure 6. The analysis begins with the selection of appropriate projects based on criteria described in Chapter 3.2. The historical commit data of the selected projects is extracted from GitHub, and the reported bug information is extracted from Bugzilla; both extraction processes rely on customized data extractors. Data from two repositories are reconciled to identify bug-fixing commits and, more importantly, the bug-containing files that are fixed in a bug-fixing commit. The reconciliation process identifies with a higher degree of confidence the files fixed in a bug-fixing commit as compared to a *noisy* approach where *every* file in a bug-fixing commit is considered as being the file responsible for the bug being fixed. The reconciliation process has been discussed above in Chapter 3.5. Next, we synthesize some product metrics using available process metrics acquired after the reconciliation process, and based on these product metrics, we develop the proposed metrics suite. The metric synthesis and generation are discussed in Chapter 4, and all metrics that participate in the experiment are listed in Table 4.

Our experiment started with correlation analysis to filter our highly correlated metrics using the Pearson correlation coefficient, which has been introduced in Chapter 2.5.1. In the next step, numerical values of each metric are converted into categorical values from 1 to 4 to support the study towards metric abnormality, after which we identify and group BFCs that are committed closely in time into a *bug-fixing period* (detailed in Chapter 5.5). We then identify the P-BFC corresponding to each BFC using three strategies. Then two different analyses are then performed, as discussed in Section 5.7. The following sections of this chapter are organized in the same order as the steps depicted in Figure 6.

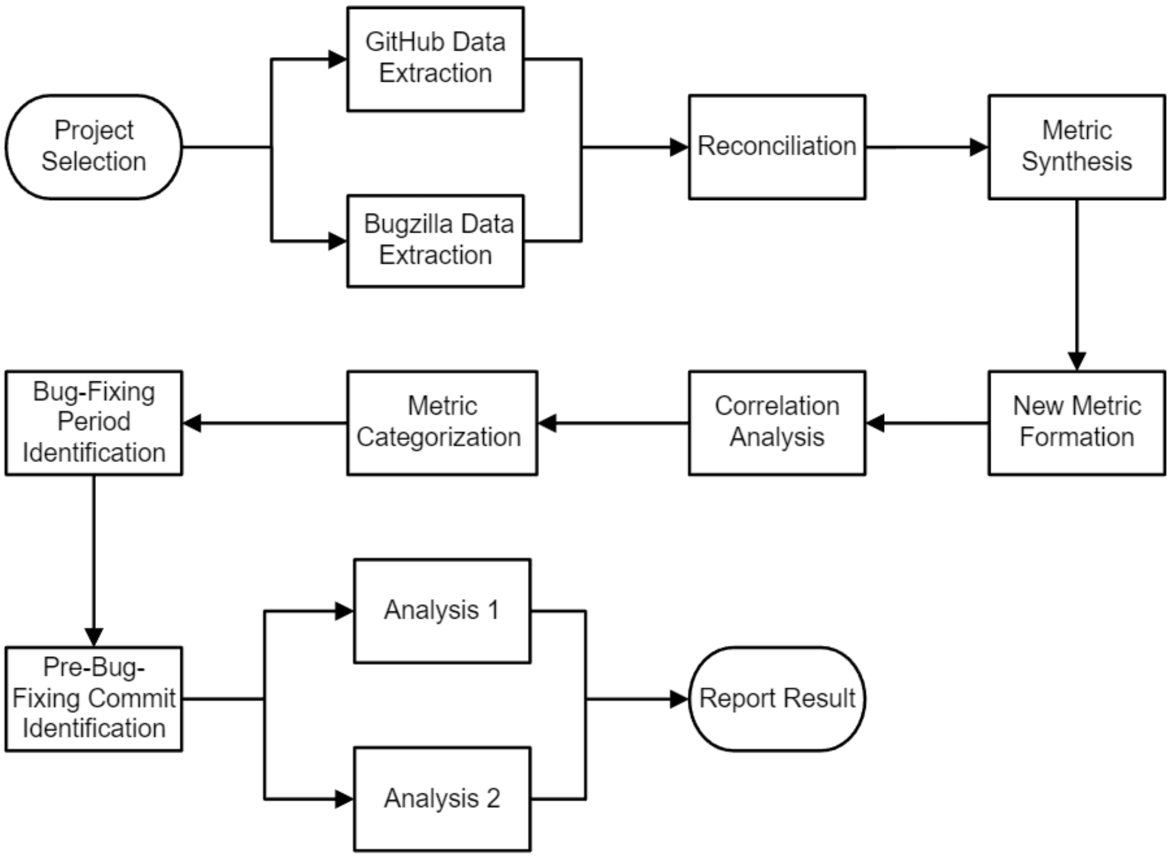
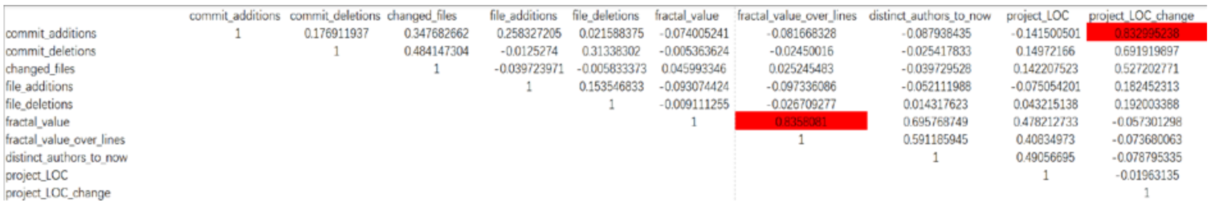


Figure 6: Overview procedure of the Posture Assessment Technique

5.3 Metric Correlations

A heavy pair-wise correlation might be introduced as the proposed metrics are either synthesized or transformed based on existing ones. The purpose of metric correlation analysis is to filter out highly correlated metrics to reduce computational complexity, as one is sufficient to represent the other one. Some commonly used statistical metric filtering techniques have been introduced in Chapters 2.5 and 2.6. The metric correlation analyses in this thesis are conducted based on the Pearson Correlation Coefficient (PCC) as introduced in



	commit_additions	commit_deletions	changed_files	file_additions	file_deletions	fractal_value	fractal_value_over_lines	distinct_authors_to_now	project_LOC	project_LOC_change
commit_additions	1	0.176911937	0.347682662	0.258327205	0.021588375	-0.074005241	-0.081668328	-0.087938435	-0.141500501	0.832995238
commit_deletions		1	0.484147304	-0.0125274	0.31338302	-0.005363624	-0.02450016	-0.025417833	0.14972166	0.691919897
changed_files			1	-0.039723971	-0.005833373	0.045993346	0.025245483	-0.039729528	0.142207523	0.527202771
file_additions				1	0.153546833	-0.093074424	-0.097336086	-0.052111988	-0.075054201	0.182452313
file_deletions					1	-0.009111255	-0.026709277	0.014317623	0.043215138	0.192003388
fractal_value						1	0.8358081	0.695768749	0.478212733	-0.057301298
fractal_value_over_lines							1	0.591185945	0.40834973	-0.073680063
distinct_authors_to_now								1	0.49056695	-0.078795335
project_LOC									1	-0.01963135
project_LOC_change										1

Figure 7: Correlation Analysis Example

Chapter 2.5.1 and are applied to each project we analyze. We choose PCC as the filtering technique because the synthesis process of the proposed metrics is mostly linear and hence doesn’t require a high degree correlation analysis. If the absolute value of the PCC between two metrics is greater than a threshold, we exclude the one with the smaller metric index from the experiment. The threshold we chose is 0.8, as statistical guidelines [53] [10] indicate that this value is associated with a “high” correlation. A partial example is shown in Figure 7. In this example, two pairs of metrics, *fractal_value* and *fractal_value_over_lines*, *commit_additions* and *project_LOC_change*, have a PCC greater than 0.8, and the one with the smaller metric index will be removed. Table 5 shows the excluded metrics of each project.

Table 6: Project-wise Excluded Highly Correlated Metrics

Project Name	Excluded Metrics
akregator	[6, 7]
ark	[1, 6, 11]
clazy	[1, 3, 6, 7, 11]
elisa	[1, 6, 7, 11]
gwenview	[1, 6, 11]
juk	[1, 2, 6, 11]
kget	[1]
kmix	[1, 2, 6, 11]
kolourpaint	[1, 2, 5, 6, 7, 11]
kompare	[1, 2, 3, 5]
kontact	[]
konversation	[1, 2, 6, 7, 11]
ktimetracker	[1, 6]
ktorrent	[1, 6, 11]
lokalize	[1, 2, 3, 6, 11]
marble	[1, 6, 7, 11]
plasmanm	[1, 11]
solid	[1, 2, 6, 7, 11, 12]
umbrello	[1, 11]
k3b	[1, 5, 6, 11]
kmail	[1, 2, 6]
kopete	[2, 3, 6, 7, 10]
korganizer	[11]
kstars	[1, 3, 6, 11]

kdevplatform	[3, 6, 11]

5.4 Metric Categorization

As metric values have a wide range, we have to discretize the values into categories (e.g., very high, high, medium, etc.). In this respect, we begin the analysis by surveying popular categorization methods. Our survey suggests two common categorization practices: a) standard deviation based and b) percentile based. The method based on standard deviation requires first calculating the mean and the standard deviation of the data and then categorizing each data point based on how many standard deviations it is away from the mean. This method generally requires that the data follows a normal distribution pattern; however, the metrics data we collected for the experiment are discrete measurements of software; therefore, it is not suitable to be categorized this way. The percentile-based method sorts metric data based on their numerical values and then groups them into percentiles. Common choices of percentiles are tertiles (33%-33%-33%), quartiles (25%-50%-25%), and deciles (10%-80%-10%) [72]. We adopt the percentile-based method and categorize the experiment data into four equal-size percentiles, namely the 0th to 25th (very low), 25th to 50th (low), 50th to 75th (high), and 75th to 100th (very high) percentile, and representing the data for the 4 percentiles with categorical values 1 to 4 respectively. Data in the lowest or highest percentile are considered abnormal. An example of metric categorization is given below. Assuming a metric vector V_{f_j} for an arbitrary file f_j is as the followings:

$$V_{f_j} = [20, \quad 2, \quad 13, \quad 35, \quad 9, \quad 5, \quad 26, \quad 19]$$

The vector is sorted to be ready for categorization. The descendingly-sorted vector is:

$$[2, \quad 5, \quad 9, \quad 13, \quad 19, \quad 20, \quad 26, \quad 35]$$

Clearly, $[2, 5]$, $[9, 13]$, $[19, 20]$, and $[26, 35]$ correspond the 0th to 25th (very low), 25th to 50th (low), 50th to 75th (high), and 75th to 100th (very high) percentile respectively. The actual values are then converted into categorical values 1, 2, 3, and 4, respectively. The converted vector V_{f_j} is then as follows:

$$V_{f_j} = [3, \quad 1, \quad 2, \quad 4, \quad 2, \quad 1, \quad 4, \quad 3]$$

The process metrics used in this work are either *commit-associated metrics* or *file-associated metrics*. *Commit-associated metrics* are shared by all files in the commit, whereas each *file-associated metric* links with a unique file. File-associated metrics data should not be categorized together with other files' data as files vary dramatically from files in size, functionality, dependency, etc. Therefore, each file's metric data are categorized independently.

5.5 Bug-Fixing Period

An interesting finding of our experiments is the clustering of bug-fixing commits (BFC) which we coined as a *bug-fixing period*. After a long series of non-BFC, BFCs tend to occur densely within the next several commits. This finding overturns our presumed convention that BFC should distribute uniformly across all commits. Since a sufficient number of pre-Bug-Fixing commits is required to form the study basis of conditional frequency rate, we treat the BFCs that have small non-BFC intervals in between as one BFC, namely a bug-fixing period. Furthermore, the frequent bug fixes within a small period are likely caused by the same or related defects introduced shortly before this period. Formally, let C_k^{BF} denote a BFC k, let C_h^{BF} the first BFC h after BFC k, and let n be the number of non-BFCs between h and k, n must satisfy $n > a$ for h and k to be considered as two separate BFC, otherwise h and k are treated as one bug-fixing period. The value of a is set to 5. An example is illustrated in Figure 8. In this example, C_7, C_9, and C_{10} are three BFCs which are separated by a non-BFC interval that is less than 5. They are together considered as one BFC and share the same pre-bug-fixing commits discussed in the next section.

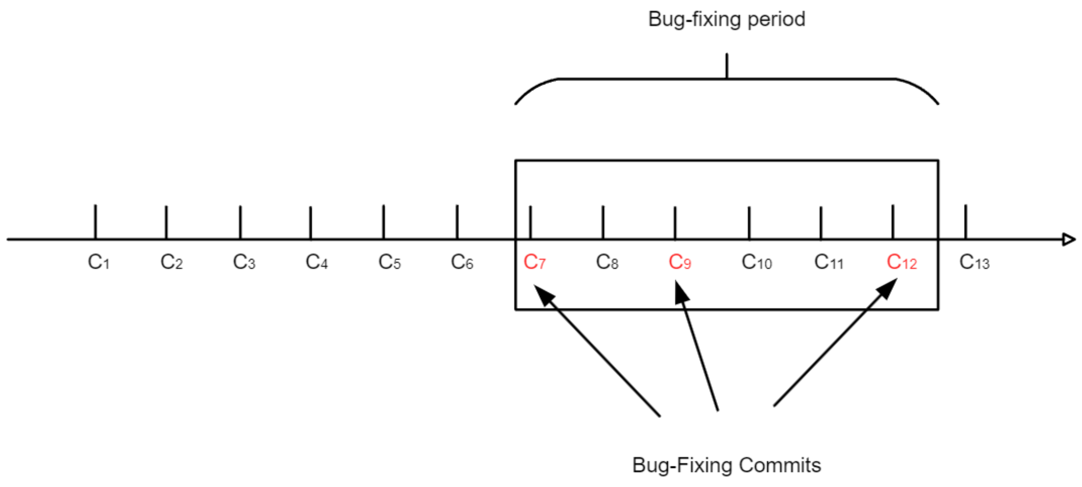


Figure 8: Bug-Fixing Period

5.6 Selection of Pre-Bug-Fixing Commits

As described in Chapter 5.1, ordinary commits occurred before a BFC contained metric value anomalies caused by defects. The statistical investigation of these commits can lead to the discovery of metrics that relate to file-level defects. For each BFC, we identify some suspicious commits prior to this BFC which might contain the manifestation of defects (i.e., extreme metrics). We refer to these commits as pre-bug-fixing commits (P-BFC). Using all commits before each commit as their P-BFC is an apparent but arduous way that results in unbalanced P-BFC sizes. Because the P-BFCs constitute the basis for our frequency calculations, we proposed different strategies for identifying P-BFCs outlined in the sections below labelled as Strategy 1, Strategy 2, and Strategy 3.

5.6.1 Strategy-1: Using a Fixed-Length Commit Segment

This strategy intuitively selects a fixed number of immediate past commits of any BFC as the corresponding P-BFC. The strategy is proposed under the assumption that after a BFC, the system can be considered temporarily defect-free, and new defects are introduced by the next normal commits. Therefore, defect-sensitive metrics should become abnormal when bugs emerge. For this thesis, we opted for the number of commits to be set to 5, which is equal to the commit-interval size presented in Section 5.4. This setting assures the P-BFC contains no BFC as the length of the *bug-fixing period* discussed above is also set to 5. Increasing the fixed fix number to include more previous commits might add noise to the P-BFC. One obvious advantage of using this strategy is it ensures each BFC has a P-BFC of the same size. For the first 5 commits in the commit history, as the total number of past commits is less than 5, all past commits are selected. An example of Strategy-1 is shown in Figure 9, where C_9 is a BFC. The segment $C_4 - C_8$ is then considered as the P-BFC associated with the BFC C_9.

5.6.2 Strategy-2: Using Fixed-Length Segment of Any BFC Files

In this strategy, we improve the P-BFC selection process by focusing only on the bug-fixing files. Theoretically, this strategy is ideal for systems where a small number of files repeatedly

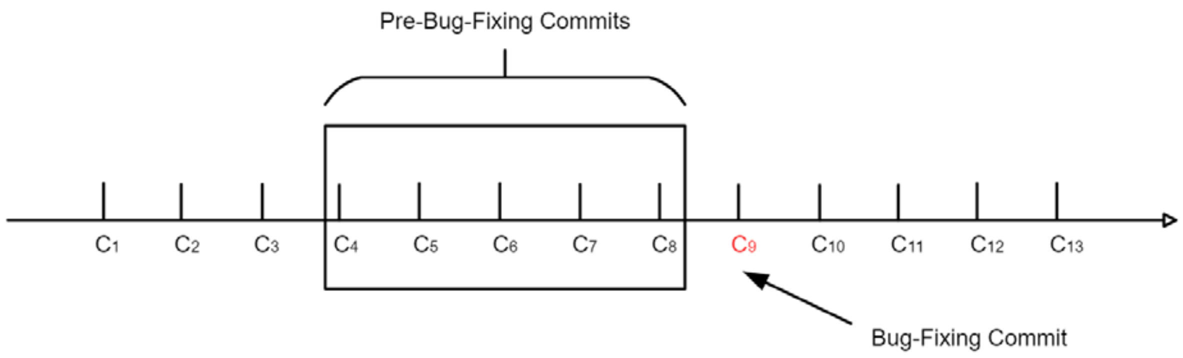


Figure 9: Example of Strategy-1

become faulty. In Strategy-2 above, for each of the bug-fixing files $f_{i,k}$ in a BFC C_k, we find the first past 5 commits prior to C_k that contain the file $f_{i,k}$ and collectively use all such found commits as the P-BFC segment period. As a past commit may contain multiple files $f_{i,k}$ all these files will be included only once in the P-BFC segment period associated with BFC C_k. An example is illustrated in Figure 10, in which commit C_{11} is a BFC with bug-fixing files f_1 and f_3. This P-BFC identification strategy traverses backward in the commit sequence to locate the first 5 commits containing f_1 and the first 5 commits containing f_3. In this example, the P-BFC segment period associated with C_{11} includes the commit C_3 (contains files f_1, and f_3 also appearing in BFC C_{11}), commit C5 (contains file f_3 also appearing in BFC C_{11}), commit C_7 (contains files f_1, and f_3 also appearing in BFC C_{11}), commit C_9 (contains files f_1 and f_4 also appearing in BFC C_{11}), and commit C_{10} (contains files f_1, and f_3 also appearing in BFC C_{11}). As it is depicted in Figure 10, commit C_4 is not included in the P-BFC associated with commit C_{11} because neither f_3 nor f_4 is in it.

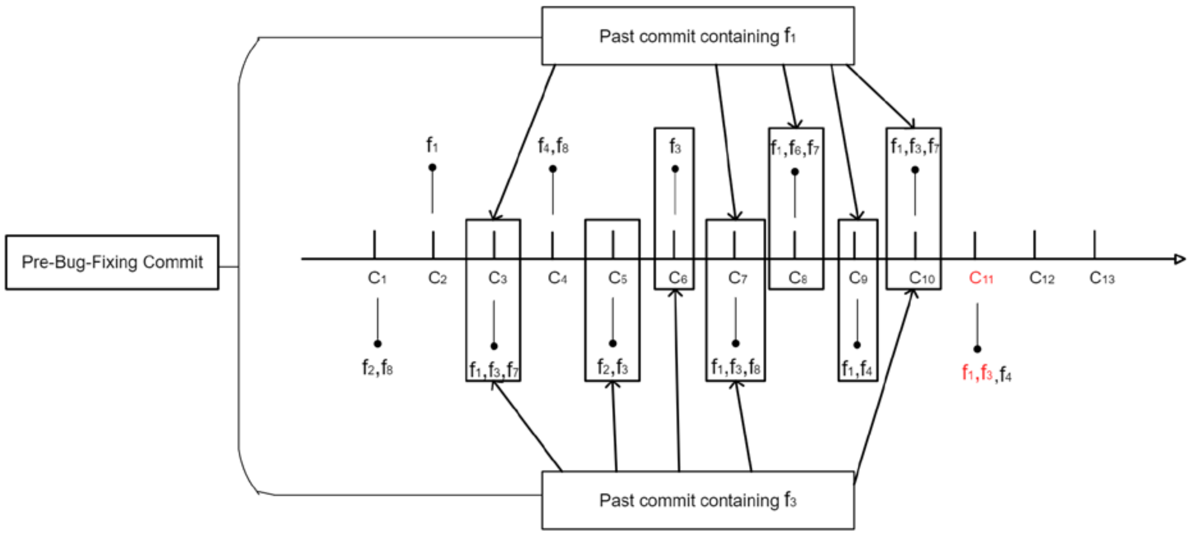


Figure 10: Example of Strategy-2

5.6.3 Strategy-3: Using Fixed-Length Segment of All BFC Files

Extending Strategy-2, we take a more aggressive step. In Strategy-3, the P-BFCs are formed by selecting the first past 5 commits that contain all bug-fixing files in a BFC. This strategy is inspired by Ria's fault forecasting model [68]. The model is based on faulty files that are committed together, which the author referred to as a co-commit relationship. One of Ria's major conclusions is that non-buggy files' co-commit history is a good indicator that these files are likely to remain non-buggy in the future. In this strategy, we seek to extend Ria's conclusion from a different angle by exploring if buggy files' co-commit history was a strong predictor of BFC. The hypothesis behind the strategy is if the bug-fixing files used to be in previous non-bug-fixing commits together, then these commits are the potential origin of the defect and, thus, are likely to contain metric abnormalities. Using the same example of Strategy 2, the new P-BFC is shown in Figure 11.

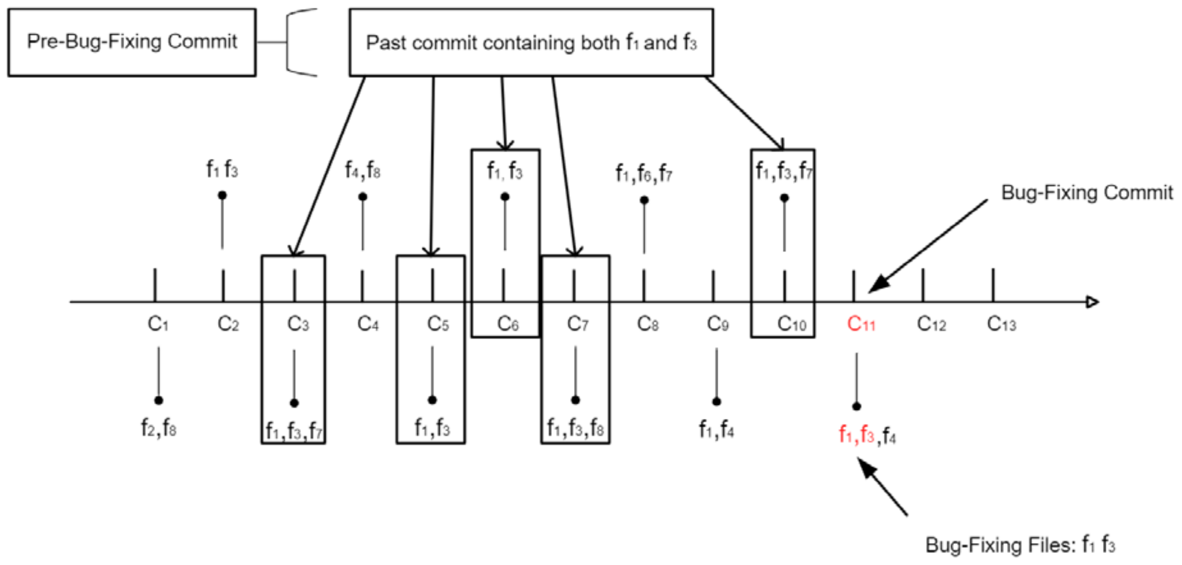


Figure 11: Example of Strategy-3

5.6.4 Working Example

Let us consider the following commits:

<u>commit_1</u>=*{commit_id =‘53963579d6cc4410cb414302b23772837d8577e1’, commit_associated_data =[…], file_data = {{file_id=‘04571199-1ed1-11eb-8195-482ae32cf5b4’, is_bug_fixing=FALSE, file_associated_data =[…]},{file_id=‘04573751-1ed1-11eb-a715-482ae32cf5b4’, is_bug_fixing= FALSE, file_associated_data =[…]}{file_id=‘04573752-1ed1-11eb-826d-482ae32cf5b4’, is_bug_fixing= FALSE, file_associated_data =[…]}, {file_id=‘04576000-1ed1-11eb-9b49-482ae32cf5b4’, is_bug_fixing= FALSE, file_associated_data =[…]}}}*

<u>commit_2</u>=*{commit_id =‘53963579d6cc4410cb414302b23772837d8577e1’, commit_associated_data =[…], file_data = {{file_id=‘04571198-1ed1-11eb-850b-482ae32cf5b4’, is_bug_fixing= FALSE, file_associated_data =[…]}}}*

commit_3=*{commit_id =‘53963579d6cc4410cb414302b23772837d8577e1’, commit_associated_data =[…], file_data= {{file_id=‘0457374f-1ed1-11eb-8453-482ae32cf5b4’, is_bug_fixing=FALSE, file_associated_data =[…]} {file_id=‘04573752-1ed1-11eb-826d-482ae32cf5b4’, is_bug_fixing= FALSE, file_associated_data =[…]}}}*

commit_4=*{commit_id =‘53963579d6cc4410cb414302b23772837d8577e1’, commit_associated_data =[…], file_data = {{file_id=‘04573751-1ed1-11eb-a715-482ae32cf5b4’, is_bug_fixing= FALSE, file_associated_data =[…]}{file_id=‘0457371c-1ed1-11eb-8530-482ae32cf5b4’, is_bug_fixing= FALSE, file_associated_data=[…]}}}*

commit_5=*{commit_id =‘53963579d6cc4410cb414302b23772837d8577e1’, commit_associated_data =[…], file_data = {{file_id=‘0457371d-1ed1-11eb-b88c-482ae32cf5b4’, is_bug_fixing=FALSE, file_associated_data =[…]},{file_id=‘04573751-1ed1-11eb-a715-482ae32cf5b4’, is_bug_fixing= FALSE, file_associated_data =[…]}{file_id=‘04573752-1ed1-11eb-826d-482ae32cf5b4’, is_bug_fixing= FALSE, file_associated_data=[…]}}}*

commit_6=*{commit_id =‘53963579d6cc4410cb414302b23772837d8577e1’, commit_ commit_associated_data=[…], file_data = {{file_id=‘0457374f-1ed1-11eb-8453-482ae32cf5b4’, is_bug_fixing=FALSE, file_ associated_data=[…]},{file_id=‘04573751-1ed1-11eb-a715-482ae32cf5b4’, is_bug_fixing=TRUE, file_associated_data=[…]}{file_id=‘04573752-1ed1-11eb-826d-482ae32cf5b4’, is_bug_fixing=TRUE, file_associated_data =[…]}}}*

We can then provide a working example of how the different strategies are applied.

Assuming *commit_1* to *commit_6* listed above are the first 6 commits of a project. *Commit_6* is identified as a bug-fixing commit as two of its files have *TRUE* for the *is_bug_fixing* label, and it is the only bug-fixing commit among the 6 commits. The actual commit-associated and file-associated metric data are omitted for simplicity reasons. During the P-BFC selection process:

Strategy_1 selects undiscriminatingly the immediate past 5 commits of the buggy commit_6 as P-BFC. In this example, commit_1 to commit_5 will be chosen.

Strategy_2 traces the past commit history of each buggy file and selects the first 5 commits that each buggy file participated in. Though buggy commit_6 has 3 files, only two of them are labelled for bug-fixing. Strategy_2 first acquires the list of buggy file IDs. In this case, the list is [‘*04573751-1ed1-11eb-a715-482ae32cf5b4*’, ‘*04573752-1ed1-11eb-826d-482ae32cf5b4*’]. An iteration of the list is performed to find 5 commits for each. In each iteration, starting from the BFC, the past commit history is traversed until 5 containing commits are satisfied or until all past commits are traversed. For file ‘*04573751-1ed1-11eb-a715-482ae32cf5b4*’, the containing commits are *commit_1*, *commit_4*, and *commit_5*, and for file ‘*04573752-1ed1-11eb-826d-482ae32cf5b4*’, the commits are *commit_1*, *commit_3*, and *commit_5*. The finding results are aggregated to remove duplicated commits, so the final P-BFC selected by strategy_2 are *commit_1*, *commit_3*, *commit_4*, and *commit_5*.

Strategy_3 aims to find 5 past commits where all the buggy files in a BFC are also co-committed together. The selection process of strategy_3 is similar to strategy_2’s. First, the buggy file list is [‘*04573751-1ed1-11eb-a715-482ae32cf5b4*’, ‘*04573752-1ed1-11eb-826d-482ae32cf5b4*’]. The commit history prior to the BFC is then traversed to find 5 commits containing all buggy files in the list until all commits have been visited. The final P-BFC selected by strategy_3 are *commit_1* and *commit_5*.

5.7 Posture Analysis

In this thesis, we conduct two types of analyses.

Analysis 1 aims to identify the conditional frequency rate of a metric pattern appearing in the pre-BFC segment (as this segment is evaluated using Strategies 1-3 as discussed above), given that a BFC occurs.

Analysis 2 aims to identify the conditional frequency rate of a BFC given a metric pattern appearing in the pre-BFC segment (as this segment is evaluated using Strategies 1-3 as discussed above), given that a BFC occurs.

5.7.1 Analysis 1: Pre-BFC Posture Frequency Given a BFC

This type of analysis aims to compute the following conditional frequency rate:

CFR (posture pattern | BFC)

The basic assumption of the proposed technique is that if defects are introduced in these P-BFCs, then they cause defect-sensitive metrics to behave abnormally until they are fixed. One common abnormal behaviour is that these metrics will demonstrate abnormal values, either very high or very low. The central idea can be summarized with a question: “Given all BFC occurrences, what postures can be found frequently before they occur?” At this point, all metric values have been converted into categorical values 1-4 (very low, low, high, very high), and the P-BFC that corresponds with each BFC has been identified. Each P-BFC corresponds to a commit state (i.e., a vector of metric values). As discussed in Chapters 1.2 and 5.6, a commit consists of several files, and a few commits identified by a strategy constitute a P-BFC. For each P-BFC, a 2-dimensional vector matrix of metrics in converted categorical values, A_{P-BFC}, can be obtained by concatenating all matrices of files of commits that constitute this P-BFC.

$$A_{P-BFC} = \begin{bmatrix} \boldsymbol{a_{f_1,m_1}} & a_{f_1,m_2} & a_{f_1,m_3} & \cdots & a_{f_1,m_j} \\ a_{f_2,m_1} & a_{f_2,m_2} & a_{f_2,m_3} & \cdots & a_{f_2,m_j} \\ a_{f_3,m_1} & a_{f_3,m_2} & a_{f_3,m_3} & \cdots & a_{f_3,m_j} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{f_i,m_1} & a_{f_i,m_2} & a_{f_i,m_3} & \cdots & a_{f_i,m_j} \end{bmatrix}$$

The concatenation process dissolves the barriers of commits, making the result matrix focus on files and their metrics. Taking the element $\boldsymbol{a_{f_1,m_1}}$ in position (1,1) in the matrix above as an example, this represents the metric $\boldsymbol{m_1}$ of file $\boldsymbol{f_1}$. Furthermore, each row of A_{P-BFC} is a metric vector associated with a file (*commit-associated metrics* are shared by all files of that commit), and each column is a vector associated with a particular metric.

Through the examination of a P-BFC's matrix, if a metric was found showing abnormal values (i.e., 1, which means *very low*, or 4, which means *very high*) with a *high frequency*, then according to our hypothesis, this metric is potentially sensitive to the bug that is fixed in the corresponding BFC. Here, *frequency rate* is defined as the division of the number of times this metric is at an abnormal value over the total number of values of this metric (i.e., the number of rows of the P-BFC's vector). As the purpose of Analysis_1 is to study the overall behaviour of posture patterns in P-BFCs, a threshold rate t is used to justify the *high frequency*. In a P-BFC, if a metric shows an abnormal *frequency rate* greater than *t*, then it is considered abnormal in this P-BFC. More specifically, a column matrix of A_{P-BFC} as shown above, represent how a metric m_k performs in the given P-BFC.

$$V_{m_k} = \begin{bmatrix} a_{f_1,m_k} \\ a_{f_2,m_k} \\ a_{f_3,m_k} \\ \vdots \\ a_{f_j,m_k} \end{bmatrix}$$

Let $-m_k$ denotes metric m_k's *very low* occurrence in V_{m_k}, and $+m_k$ denotes the *very high* occurrence in V_{m_k}. The frequency $freq(-m_k)$ and $freq(+m_k)$ can be calculated by the following equation:

$$freq(-m_k) = \frac{-m_k}{\|V_{m_k}\|}, \text{ and } freq(+m_k) = \frac{+m_k}{\|V_{m_k}\|}$$

If $freq(-m_k)$ or $freq(+m_k)$ is greater than or equal to t, metric M_k is considered *very low* or *very high* in this P-BFC, and it is a valid study subject to infer the corresponding BFC. As a single metric can be biased and partial, more metrics involvement led to more accurate and comprehensive results. The frequency analysis can be extended to explore how several metrics become abnormal at the same time (i.e., in the same P-BFC). Given the total metrics set as the following:

$$S_m = \{m_1, m_2, m_3, \ldots, m_j\}$$

Because each metric m_k has two abnormal values $-m_k$ and $+m_k$ as denoted above, the set of abnormal metrics that are involved in the experiment becomes:

$$S_{\text{abnormal}} = \{-m_1, +m_1, -m_2, +m_2, -m_3, +m_3, \ldots, -m_j, +m_j\}$$

Let S_x be a subset of $S_{abnormal}$. If S_x satisfies the following:

$$(\forall m_x)\{m_x \in S_x : freq(m_x) \geq t\}$$

Then all elements of S_x are considered simultaneously abnormal in the given P-BFC, and we refer to such an S_x as an *abnormal posture*. Notice because of the presence of other categorical values (i.e., 2, which means *low*, or 3, which means *high*) and the threshold value, neither or both of $freq(-m_k)$ and $freq(+m_k)$ can be greater than threshold t. To avoid this contradiction, we ignore the case that a metric is simultaneously abnormal in both ways. The choice of the threshold is a trade-off between precision (i.e., how significant the frequency rate is) and the number of satisfied cases. We experiment on different thresholds and report the result in Table 7. A higher threshold results in more significant frequency rates but fewer satisfied patterns, whereas a lower threshold leads to more satisfied patterns but lower precision. We opted for threshold rate t to be 0.4 for this experiment, as it has a reasonable significance level while leading to sufficient satisfied patterns.

Table 7: Different Thresholds and Satisfied Patterns

Threshold	Satisfied Patterns Cases		
	strategy_1	strategy_2	strategy_3
0.2	3796.58	3691.17	3917.83
0.3	3566.5	3466.29	3672.54
0.4	3163.04	2972.67	3244.25
0.5	2778.62	2150.38	2368.38
0.6	2520.75	1517.29	1274.83
0.7	2344.96	1071.71	960.67
0.8	2271.67	876.67	662.14

Compared to the analysis using a single metric, an *abnormal posture* of multiple metrics is a more thorough description of the P-BFC; consequently, it leads to better quality assessment. The number of subsets of a set with size *n* is 2^n. Conducting an exhaustive analysis of each subset of the total metric set is beyond the scope of this thesis. The alternative approach we used is restricting the metric subset to a fixed size *z* and exhaustively analyzing every possible subset of this size. Size *z* is set to 3 in this thesis; namely, all posture S_x that satisfy $|S_x| = 3$ will be examined. It is possible that multiple *abnormal postures* co-exist in one P-BFC. However, after applying the frequency analysis to each P-BFC, a ranked list of *abnormal posture* occurrences is obtained, where the top posture of the list marks the posture that most frequently occurred in P-BFCs, and metrics in such posture are more valuable to assess BFC-entering state.

5.7.1.1 Example

We provide an example of how 3 metrics are deemed very high or very low at the same in a P-BFC. Considering the following categorized P-BFC matrix:

$$\begin{bmatrix} m_1 & m_2 & m_3 & \dots \\ 1 & 4 & 2 & \dots \\ 2 & 4 & 3 & \dots \\ 2 & 2 & 1 & \dots \\ 3 & 1 & 4 & \dots \\ 1 & 3 & 4 & \dots \\ 1 & 4 & 1 & \dots \\ 4 & 4 & 3 & \dots \\ 1 & 4 & 3 & \dots \end{bmatrix}$$

Assuming we are interested in abnormal posture $\{-m_1,\ m_2,\ -m_3\}$---if metric m_1 was *very low*, while if metrics m_2 was *very high*, and while if metric m_3 was *very low* in this P-BFC. Hence $freq(-m_1)$, $freq(m_2)$, and $freq(-m_3)$ described above are computed.

$$freq(-m_1) = \frac{-m_1}{\|V_{m_1}\|} = \frac{4}{8} = 0.5$$

$$freq(m_2) = \frac{m_2}{\|V_{m_2}\|} = \frac{5}{8} = 0.625$$

$$freq(-m_3) = \frac{-m_3}{\left\|V_{m_3}\right\|} = \frac{2}{8} = 0.25$$

As the threshold is set to 0.4, $freq(-m_3)$ is lower than this threshold, and thus, the desired abnormal posture is not found in this P-BFC.

5.7.2 Analysis 2: BFC Frequency Given a P-BFC Posture Pattern

This type of analysis aims to compute the following conditional frequency rate:

CFR(BFC | posture pattern)

The quality posture assessment discussed above treats the occurred BFC as conditions, identifies the past investigation areas (P-BFC) based on them, and uses the most likely posture found in P-BFCs to assess the BFC-entering phase hence assessing software quality. Although the found posture has a tight correlation with imminent BFCs, they are not necessarily good predictors that suit fault mitigation jobs. BFCs and corresponding P-BFCs only comprise a small proportion of the entire commit history of a project. Every P-BFC has a posture occurrence doesn't mean all the posture occurrences are in P-BFCs. Therefore, such posture occurrences are not solid evidence that there will be an imminent BFC within the next few commits.

We design a complementary technique to assess the correlation between postures and BFCs. This technique is based on computing the total occurrences of each *abnormal posture* across the entire project, then inspecting how many of them took place in P-BFC. A frequency ratio can be obtained by dividing the P-BFC occurrences by the total occurrences. If the ratio of a posture is high enough, it indicates that the posture has a high interrelationship with BFCs; thus, the posture is a good predictor of them. Again, the central idea of the technique can be summarized with a question: "Given the occurrences of an abnormal posture, how likely will there be a BFC shortly after?". If BFCs frequently occur after a posture, this posture is a good predictor. More specifically, the metrics of each selected project, $A_{project}$, is in tabular form, which can be converted into a 2-dimensional matrix similar to A_{P-BFC} shown above:

$$A_{project} = \begin{bmatrix} a_{f_1,m_1} & a_{f_1,m_2} & a_{f_1,m_3} & \cdots & a_{f_1,m_j} \\ a_{f_2,m_1} & a_{f_2,m_2} & a_{f_2,m_3} & \cdots & a_{f_2,m_j} \\ a_{f_3,m_1} & a_{f_3,m_2} & a_{f_3,m_3} & \cdots & a_{f_3,m_j} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{f_i,m_1} & a_{f_i,m_2} & a_{f_i,m_3} & \cdots & a_{f_i,m_j} \end{bmatrix}$$

The row and column interpretation of $A_{project}$ are the same as A_{P-BFC} except that the former has a much larger row size than the latter in general. In the previous assessment, because P-BFCs are segments rather than a row of data, a threshold is set to determine whether a metric should be deemed as very high or very low. In this assessment, as the first task is to count the occurrence of *abnormal postures* by examining each row of $A_{project}$, the threshold is not required. The set of abnormal metrics, $S_{abnormal}$, has been given in the previous section, and the set of all possible unique combinations of $S_{abnormal}$ of the size of 3, $S^3_{abnormal}$, is:

$$S^3_{abnormal} = \{\{-m_1, -m_2, -m_3\}, \{-m_1, -m_2, +m_3\}, \ldots, \{+m_i, +m_j, +m_k\}\}$$

Let n be the count of posture occurrences, and z be the count of posture occurrences in P-BFC. For each unique abnormal posture in $S^3_{abnormal}$, we iterate each row of $A_{project}$ space, computing n and z. Taking posture $\{-m_1, -m_2, -m_3\}$ as an example, during the iteration of each row which is in the form of $[a_{f_i,m_1} \quad a_{f_i,m_2} \quad a_{f_i,m_3} \quad \cdots \quad a_{f_i,m_j}]$, the predicate $\left((a_{f_i,m_1} == -1) \cap (a_{f_i,m_2} == -1) \cap (a_{f_i,m_3} == -1)\right)$ is being evaluated. If the predicate is true, the corresponding posture has occurred once, and the n count for this posture is incremented by 1. If the row is contained by at least one P-BFC, the z count for this posture is incremented by 1. The *P-BFC occurring rate* of this posture (i.e., the rate of a posture occurrence in P-BFC given its occurrence), $P_{-m_1,-m_2,-m_3} = \frac{z}{n}$, can be calculated after iterating the entire $A_{project}$ space. After computing the conditional frequency rate of every posture in $S^3_{combination}$, the posture with the highest frequency rate represents the best BFC predictor.

Chapter 6

6 Experiments and Obtained Results

In this chapter, we present the setup for our experiments, present the obtained results, and discuss threats to validity.

6.1 Experiment Setup

The datasets related to the selected software projects are fetched from Bugzilla and GitHub repositories utilizing custom-made REST APIs and readers written in Python. The extracted data are then stored in the form of CSV files. The preprocessing of data and new metrics formation is accomplished by the program utilizing the Python Pandas package. The proposed framework is developed in Java with SDK version 10.0.2.

6.2 Metrics

A total of 18 metrics are used in the experiments, including 8 common process metrics, 2 synthesized static code metrics, and 8 proposed new processed metrics (Table 4). To simplify the presentation of the results, each metric is coded with an index number (1-18, see Section 4.3 Table 4). As metric values in both the highest percentile and lowest percentile are deemed abnormal and studied, an "+" is used to denote values in the highest percentile and an "-" denotes values in the lowest percentiles. For example, in result tables presented in the following sub-chapters, an "+18" is interpreted as "metric 18 is found very high". As each metric has two extreme percentiles, the total of variables that participate in the study is: 18(the number of metrics) $\times$ 2(the number of abnormal metric categorizations, *very high* and *very low*) $=$ 36. The size of the metric combination in the experiment is 3, so the total number of unique combinations is $C(36, 3) = 7140$. We chose this size as a combination size of 4 or higher is beyond the computation resource we have.

6.3 Experimental Results

We have implemented the three strategies discussed in Chapter 5.6 for selecting the P-BFC commits, all of which are used by *Analysis 1* and *Analysis 2*, as discussed in Section 5.7. The experiment is performed on 25 open-sourced software projects listed in Section 3.2. The following rank criteria are used to evaluate the results:

Table 8: Evaluation Criteria

Frequency Rate	Rank
1-0.8	Very good
0.8-0.6	Good
0.6-0.4	Acceptable
0.4-0.2	Poor
0.2-0	Very poor

6.3.1 Analysis 1 Results: P (Posture Pattern | BFC)

The obtained results of Analysis 1 are reported per strategy per project. For each strategy, only the posture with the highest conditional frequency rate is reported. All results are displayed in descending order of the rate. An example of how to interpret the result in part 1 is given in the next section.

6.3.1.1 Result Interpretation

Using the first row of Table 7 in the next section as an example, the result is interpreted as the following:

In project solid, in the 37 P-BFCs identified by Strategy-1, metric 3 (i.e., changed files) is found to be very low (indicated by the "-" sign), metric 15 (i.e., File Total LOC delta Percent Change) is found to be very low, and metric 18 (i.e., File-project LOC delta weight) is found to be very high all at the same time in the same P-BFC for a total occurrence of 28. The P-BFC occurring rate of this posture in all P-BFCs is $28 \div 37 \approx$

0.5285, *and it is the highest rate among all posture's P-BFC occurring rate in project solid.*

6.3.1.2 Analysis 1 / Strategy-1: Using a Fixed-Length Commit Segment

Table 9: Most Frequently Occurred Posture in P-BFC Identified by Strategy-1

Project Name	Most P-BFC Occurred Posture			Occurrence in P-BFC	P-BFC size	P-BFC Occurring Rate
	Metric One	Metric Two	Metric Three			
solid	-3	-15	18	28	37	0.7568
kolourpaint	-3	-10	18	79	127	0.6220
lokalize	-5	-10	18	39	69	0.5652
umbrello	-3	-10	18	180	369	0.4878
kopete	14	-15	18	431	919	0.4690
k3b	-3	-10	18	152	339	0.4484
kstars	-2	-10	18	274	628	0.4363
kontact	-3	-10	18	114	265	0.4302
korganizer	-4	-5	-12	227	550	0.4127
kdevplatform	-1	-10	18	355	866	0.4099
kmix	-4	-5	-12	46	118	0.3898
marble	-3	-10	18	270	713	0.3787
ktorrent	-3	-10	18	46	123	0.3740
akregator	-3	-10	18	106	288	0.3681
elisa	-3	-10	18	37	101	0.3663
ark	-3	-10	18	80	219	0.3653
juk	-3	-10	18	55	158	0.3481
plasmanm	-3	-10	18	56	169	0.3314
ktimetracker	-2	-10	-12	37	112	0.3304
konversation	-3	-10	18	146	462	0.3160
kget	-4	-10	-12	52	165	0.3152
clazy	-2	-10	18	33	107	0.3084
kmail	-4	-5	-12	392	1296	0.3025
gwenview	-3	-10	18	101	344	0.2936
kompare	-8	-9	-14	8	40	0.2000

Table 7 shows the most occurring postures in P-BFCs identified by Strategy-1 of each project. The posture with the highest frequency among all 25 projects is *solid* at 0.7568. This means that in 75.68% of the cases, this P-BFC posture occurs when we have a BFC, while the lowest one is from *kompare* at 20%. The average P-BFC occurring rate is 40.1%. Project *solid* and *kolourpaint* are assessed to have a posture with a P-BFC occurring rate ranked good according to the criteria in Table 6. We cross-compare these two projects with the system characteristics in Table 2 and find out that *solid* has the least *BFC per commit*. Meanwhile, *kolourpaint* has the second least *BFC per commit* among 25 selected systems. However, project *lokalize* and *Umbrello*, in third and fourth place, have the second-highest and the highest *BFC per commit* status. The cross-comparison between fifth to eighth places of results and *BFC per commit* of them supports the pattern: project *kopete* has the seventh highest *BFC per commit*, while *kontact* and *kstars* have the third and the sixth least. The observed pattern is evidence that Strategy-1 works better on projects with either relatively high or low *BFC per commit* status.

6.3.1.3 Analysis 1 / Strategy-2: Using Fixed-Length Segment of Any BFC Files

Table 10: Most Frequently Occurred Posture in P-BFC Identified by Strategy-2

Project Name	Most P-BFC Occurred Posture			Occurrences in P-BFC	P-BFC size	P-BFC Occurring Rate
	Metric One	Metric Two	Metric Three			
ktorrent	3	10	-18	65	123	0.5285
solid	-8	-15	-16	18	37	0.4865
korganizer	1	3	10	265	550	0.4818
plasmanm	2	3	10	80	169	0.4734
kopete	-5	-8	-15	420	919	0.4570
kdevplatform	1	2	10	394	866	0.4550
kompare	-8	10	-18	18	40	0.4500
kmail	3	10	-18	527	1296	0.4066
gwenview	3	10	-18	137	344	0.3983
marble	2	3	-18	281	713	0.3941
juk	3	10	-18	61	158	0.3861
ark	2	3	10	84	219	0.3836
kontact	1	10	-18	98	265	0.3698
k3b	3	10	-18	122	339	0.3599

umbrello	2	3	-18	130	369	0.3523
clazy	-5	-12	-16	35	107	0.3271
ktimetracker	3	10	-18	36	112	0.3214
kmix	3	-5	-12	37	118	0.3136
konversation	3	10	14	142	462	0.3074
kolourpaint	-8	-9	17	37	127	0.2913
akregator	1	3	10	76	288	0.2639
lokalize	10	13	-18	17	69	0.2464
kstars	2	10	13	154	628	0.2452
elisa	2	3	10	24	101	0.2376
kget	2	3	13	39	165	0.2364

Table 8 reports the most occurring postures in P-BFCs identified by Strategy-2 of each system. The posture with the highest frequency rate among all 25 projects is from *ktorrent* at 52.85%, while the lowest one is from *kget* at 23.64%. The average conditional frequency rate is 36.69%. The result is two-tiered according to the criteria in Table 6. Tier one contains the best 8 systems that are ranked acceptable, and all the rest 17 systems are ranked poor. Based on the system characteristics we computed in Table 2, we observe that of the top 5 projects in the result, *solid, kopete, plasmanm, ktorrent*, and *korganizer* have respectively the least, and the second least, the third least, the eighth least, and the tenth least *Average File Count per Commit* among the 25 selected projects. Based on the observations, we conclude that Strategy-2 works better on projects which have relatively fewer files involved in commits on average.

6.3.1.4 Analysis 1 / Strategy-3: Using Fixed-Length Segment of All BFC Files

Table 11: Most Frequently Occurred Posture in P-BFC Identified by Strategy-3

Project Name	Most P-BFC Occurring Posture			Occurrence in P-BFC	P-BFC size	P-BFC Occurring Rate
	Metric One	Metric Two	Metric Three			
ktorrent	2	10	14	96	123	0.7805
kompare	10	12	-18	31	40	0.7750
korganizer	1	10	-18	394	550	0.7164
umbrello	3	10	-18	263	369	0.7127
kopete	9	14	17	645	919	0.7018
clazy	-5	-12	-16	74	107	0.6916
lokalize	10	13	-18	47	69	0.6812
kdevplatform	1	10	-18	589	866	0.6801
juk	3	10	-18	105	158	0.6646
kmail	3	10	-18	854	1296	0.6590
ark	2	3	10	141	219	0.6438
akregator	1	2	10	183	288	0.6354
gwenview	3	10	-18	215	344	0.6250
k3b	3	14	-18	203	339	0.5988
elisa	-8	-9	17	60	101	0.5941
kmix	3	-5	-12	70	118	0.5932
plasmanm	3	10	-18	100	169	0.5917
marble	2	3	-18	417	713	0.5849
konversation	3	10	-18	268	462	0.5801
ktimetracker	3	10	-18	64	112	0.5714
solid	-8	-15	-16	21	37	0.5676
kget	2	3	13	92	165	0.5576
kontact	1	10	-18	140	265	0.5283
kstars	2	10	-18	318	628	0.5064
kolourpaint	3	10	14	64	127	0.5039

Table 9 reports the most occurring postures in P-BFCs identified by Strategy-3 of each system. The posture with the highest frequency rate among all 25 projects is from *ktorrent* at 78.05%, while the lowest one is from *kolourpaint* at 50.39%. The average conditional frequency rate is 62.98%. A total of 13 systems are ranked good and make up the tier one result, followed by the rest 12 systems with the rank of *acceptable*. We are unable to find a system characteristic in Table 2 that justifies all the top 13 best results, but when we examine the top 11 systems of the best 13 results, we observe that 10 of the 11 systems (*Umbrello, localize, ktorrent, clazy, juk, kopete, ark, compare, kmail,* and *korganizer*) are also 10 of the top 11 systems with the most *BFC per commit*. From this observation, we can conclude that Strategy-3 works better with projects that have higher *BFC per commit*.

6.3.1.5 Comparison of Results of Analysis 1 Using Different Strategies

Table 7, Table 8, and Table 9 contain the most likely occurred posture in P-BFCs of each project. We evaluate the results by counting the instances in each rank of the criteria in Table 6 and display the result summarization in Table 10. Obviously, Strategy-3 outperforms Strategy-1, and Strategy-1 slightly outperforms Strategy-2. Strategy-2 and 3 are based on a similar rationale, that is, the history of BFC files contains more manifestations of abnormal posture. However, the result of Strategy-3 vastly outperforms Strategy-2's. It means the co-committing history of BFC files in revealing forthcoming BFCs.

Table 12: Instances of Each Rank for Analysis 1

	Strategy-1	Strategy-2	Strategy-3
Very good	0	0	0
Good	2	0	13
Acceptable	8	8	12
Poor	15	17	0
Very Poor	0	0	0

6.3.2 Analysis 2 Results: P (BFC | P-BFC Posture Pattern)

In this section, we present the obtained results for Analysis-2, discussed in Chapter 5.7.2 and the selection process to select the best BFC predictor. In Analysis-1, the best posture of each system is selected only based on the frequency; however, for a posture to be used as a good BFC predictor, it must occur both frequently and frequently in P-BFCs. In addition to the *P-BFC occurring rate*, the *occurring rate* of each posture is computed by dividing the total posture occurrence by the size of the system dataset. As the posture *occurring rate* and the *P-*

BFC occurring rate discussed in Chapter 5.7.2 is equally important, we compute the product of these two factors, and for each system, the posture with the highest product value is selected as the best BFC leading posture. We present the top results of the system *ark* in Table 11 to demonstrate how the best posture of an individual system is selected. Abnormal posture {2, 3, 10} is found 2031 times in *Ark's* dataset. The *occurring rate* of this posture is calculated to be 0.1441 (the size of the dataset is omitted). Out of the 2031 posture occurrences, 1458 occurrences are found in P-BFC; therefore, the *P-BFC occurring rate* is $1458 \div 2031 = 0.7179$. The product of the *occurring rate* and *P-BFC occurring rate* is $0.1441 \times 0.7179 = 0.1034$. The product value of posture {2, 3, 10} is the highest among all postures of *Ark*, so it is the best BFC leading posture in the system. The best posture of each project under each strategy is selected through the same process and displayed in the following sections.

Table 13: An Example of the Top Results of Project *Ark*

Metric One	**Metric Two**	**Metric Three**	**Total Occurrences**	**Occurring rate**	**Occurrence in P-BFCs**	**P-BFC occurring frequency**	**Product**
2	3	10	2031	0.1441	1458	0.7179	0.1034
2	10	-18	1871	0.1327	1205	0.6440	0.0855
3	10	-18	1691	0.1199	1204	0.7120	0.0854
2	3	-18	1579	0.1120	1145	0.7251	0.0812
2	10	13	1684	0.1194	934	0.5546	0.0662
2	5	12	1590	0.1128	783	0.4925	0.0555
4	5	12	1874	0.1329	752	0.4013	0.0533
5	12	16	1746	0.1238	644	0.3688	0.0457
5	12	15	1699	0.1205	621	0.3655	0.0440
…	…	…	…	…	…	…	…

6.3.2.1 Analysis 2 / Strategy-1: Using a Fixed-Length Commit Segment

Table 14: Best BFC Predicting Posture of Each System Using Strategy-1

Project Name	**Best BFC Predicting Posture**			**Occurring rate**	**P-BFC Occurring Rate**	**Product**
	Metric One	**Metric Two**	**Metric Three**			
kmix	-4	-5	-12	0.309	0.3043	0.0940
kmail	-4	-5	-12	0.2952	0.2617	0.0772
clazy	2	10	14	0.1581	0.4696	0.0742
korganizer	-4	-5	-12	0.2709	0.2698	0.0731
juk	-4	-5	-12	0.2591	0.2708	0.0702
kget	-6	-7	-8	0.2198	0.3183	0.0700
plasmanm	-6	-7	-8	0.1961	0.3467	0.0680
umbrello	-4	-5	-12	0.2938	0.2203	0.0647
kstars	2	13	14	0.1239	0.5122	0.0635
solid	-15	-16	-18	0.3393	0.1709	0.0580
kolourpaint	-3	-10	18	0.1847	0.2987	0.0552
marble	3	10	14	0.1179	0.4592	0.0541
kontact	-2	-5	-12	0.177	0.2916	0.0516
lokalize	-4	-5	-12	0.2745	0.1834	0.0504
kopete	-5	-8	-15	0.1838	0.2683	0.0493
ktimetracker	-2	-10	-12	0.155	0.3116	0.0483
kdevplatform	-1	-10	18	0.1601	0.2986	0.0478
k3b	-4	-12	-15	0.1567	0.2972	0.0466
ktorrent	-2	-5	-12	0.1692	0.2552	0.0432
gwenview	-3	-10	18	0.1614	0.2669	0.0431
akregator	-4	-10	-12	0.1457	0.2914	0.0425
ark	-4	-12	-15	0.1544	0.271	0.0418
elisa	-3	-10	18	0.1494	0.2761	0.0413
konversation	-4	-12	-15	0.1525	0.2691	0.0410
kompare	-8	-9	-14	0.1144	0.2842	0.0325

Table 15: Posture Instances of Strategy-1

Abnormal Posture			# of instances
-4	-5	-12	6
-3	-10	18	3
-4	-12	-15	3
-2	-5	-12	2
-6	-7	-8	2
3	10	14	1
2	13	14	1
2	10	14	1
-1	-10	18	1
-2	-10	-12	1
-4	-10	-12	1
-5	-8	-15	1
-8	-9	-14	1
-15	-16	-18	1

Table 12 depicts the best BFC-predicting postures in each system based on P-BFCs identified by Strategy-2. The results are ordered by the product value of the *occurring rate* and *P-BFC occurring rate* descendingly. We observed that the highest *occurring rate* (0.3393) is from *solid*, and the highest *P-BFC occurring rate* (0.5122) is from *Kstars*. The posture with the highest product value, {-4, -5, -12}, is achieved by *Kmix*.

In Table 13, we observe that posture {-4, -5, -12} appears as the best posture for 6 different projects, and more importantly, four of the six projects with such posture, *Kmix, Kmail, korganizer*, and *Juk*, achieve the highest, the second highest, the fourth highest, the fifth highest product respectively in Table 12; meanwhile, they are found to have the fourth least, the, the twelfth least, the fifth least, and eighth least KLOC. Based on the observations, abnormal posture {-4, -5, -12} works well on medium-small projects with projects size between 57 KLOC to 13 KLOC against P-BFC identified using the fixed-length commit segment method.

6.3.2.2 Analysis 2 / Strategy-2: Using Fixed-Length Segment of Any BFC Files

Table 16: Best BFC Predicting Posture of Each System Using Strategy-2

Project Name	**Best BFC Predicting Posture**			**Occurring Rate**	**P-BFC Occurring Rate**	**Product**
	Metric One	**Metric Two**	**Metric Three**			
solid	-8	-15	-18	0.3781	0.9696	0.3666
umbrello	-4	-5	-12	0.2938	0.8618	0.2532
kmix	-4	-5	-12	0.309	0.6793	0.2099
lokalize	-4	-5	-12	0.2745	0.7607	0.2088
kolourpaint	3	10	14	0.178	1	0.1780
kmail	-4	-5	-12	0.2952	0.5984	0.1766
ktorrent	-7	-8	-12	0.1932	0.8868	0.1713
kdevplatform	1	2	10	0.1716	0.995	0.1708
juk	-4	-5	-12	0.2591	0.6549	0.1697
clazy	-5	-12	-16	0.1893	0.8487	0.1606
plasmanm	2	10	14	0.1602	0.9729	0.1558
gwenview	2	3	10	0.1547	1	0.1547
k3b	2	3	10	0.1608	0.9603	0.1544
korganizer	-4	-5	-12	0.2709	0.5634	0.1526
kget	-6	-7	-8	0.2198	0.6775	0.1489
marble	2	3	14	0.1523	0.9398	0.1431
elisa	2	10	14	0.1508	0.9291	0.1401
konversation	3	10	14	0.1343	0.9644	0.1295
ark	2	3	10	0.1441	0.8872	0.1278
kompare	-6	-7	-8	0.177	0.7014	0.1241
kopete	-5	-8	-15	0.1838	0.653	0.1200
akregator	1	2	10	0.1351	0.8752	0.1182
kontact	1	10	-18	0.1343	0.8378	0.1125
kstars	2	10	13	0.1279	0.8548	0.1094
ktimetracker	2	3	10	0.1164	0.743	0.0865

Table 17: Posture Instances using Strategy-2

Abnormal Posture			# of instance
-4	-5	-12	6
2	3	10	4
3	10	14	2
2	10	14	2
1	2	10	2
-6	-7	-8	2
2	10	13	1
2	3	14	1
1	10	-18	1
-5	-8	-15	1
-5	-12	-16	1
-7	-8	-12	1
-8	-15	-18	1

Table 14 depicts the postures of each system with the highest product value of *occurring rate* and *P-BFC occurring rate*. The P-BFCs are identified by Strategy-1. We observed that the highest occurring rate (0.3781) and product value (0.3666) in the table are both achieved by posture {-8, -15, -18} in project *solid*, and the lowest occurring rate and product value are both from posture {2, 3, 10} in project *Ktimetracker*. We also notice that the *P-BFC occurring rate* of best posture in project *Gwenview* and *Kolourpaint* are both perfectly 1. It indicates that all posture occurrences are found in some P-BFCs; however, as many posture occurrences might locate in one P-BFC, a *P-BFC occurring rate* of 1 doesn't guarantee every P-BFC has its occurrence. We will address this issue in Chapter 6.3.2.4.

Table 15 depicts the number of instances of each posture in Table 14. We observe that posture {-4, -5, -12} appears as the best posture for 6 different projects again, and the second most emerged abnormal posture is {2, 3, 10} for 4 projects. Four of the six projects possessed posture {-4, -5, -12} are *Umbrello, Kmix, Localize,* and *Kmail*, the products of which are the second highest, the third highest, the fourth highest, and the sixth highest, respectively, in Table 14. We conduct a cross-comparison between these four projects and the project features in Table 2 to understand the relationship between the posture's good performance and the adopted P-BFC identification strategy. We find *Umbrello* and *Localize* have the highest and the second highest *BFC per Commit* ratio, and *Kmix* and *Kmail*'s are seventh and tenth highest, respectively. Based on the found relationship, abnormal posture {-4, -5, -12} is suitable to predict BFC in projects with a high *BFC per Commit* ratio using Strategy-2.

6.3.2.3 Analysis 2 / Strategy-3: Using Fixed-Length Segment of All BFC Files

Table 18: Best BFC Predicting Posture of Each System Using Strategy-3

Project Name	Best BFC Predicting Posture			Occurring Rate	P-BFC Occurring Rate	Product
	Metric One	Metric Two	Metric Three			
solid	-8	-15	-18	0.3781	0.9538	0.3606
kolourpaint	3	10	14	0.178	1	0.1780
kdevplatform	1	2	10	0.1716	0.9695	0.1664
k3b	2	3	10	0.1608	0.9209	0.1481
plasmanm	2	3	10	0.1513	0.9459	0.1431
gwenview	2	3	10	0.1547	0.8856	0.1370
korganizer	1	3	10	0.1419	0.9007	0.1278
umbrello	2	3	10	0.1275	0.947	0.1208
juk	3	10	-18	0.1327	0.9061	0.1202
marble	2	3	14	0.1523	0.7879	0.1200
kmail	3	10	-18	0.1429	0.8028	0.1147
konversation	3	10	14	0.1343	0.7805	0.1048
ark	2	3	10	0.1441	0.7179	0.1034
kstars	2	10	13	0.1279	0.7955	0.1018
kget	2	3	10	0.1385	0.7273	0.1007
kontact	1	10	-18	0.1343	0.7291	0.0979
kmix	-4	-5	-12	0.309	0.3127	0.0966
ktorrent	2	3	14	0.1385	0.6154	0.0852
clazy	2	10	14	0.1581	0.5053	0.0799
akregator	1	2	10	0.1351	0.5768	0.0779
elisa	3	10	-18	0.1086	0.6982	0.0758
kompare	-8	10	-18	0.1229	0.6031	0.0741
ktimetracker	-12	-16	-18	0.1151	0.5551	0.0639
lokalize	-4	-5	-12	0.2745	0.2195	0.0603
kopete	-5	-8	-15	0.1838	0.3177	0.0584

Table 19: Posture Instances of Strategy-3

Abnormal Posture			# of instances
2	3	10	6
3	10	-18	3
3	10	14	2
2	3	14	2
1	2	10	2
-4	-5	-12	2
2	10	14	1
2	10	13	1
1	10	-18	1
1	3	10	1
-5	-8	-15	1
-8	10	-18	1
-8	-15	-18	1
-12	-16	-18	1

Table 16 depicts the best BFC-predicting postures selected from each project based on P-BFCs identified by Strategy-2. We observed that the highest *occurring rate* (0.3781) is from *solid*, and the highest *P-BFC occurring rate* (0.4696) is from *Clazy*. The posture with the highest product value, {-8, -15, -18}, is also achieved by *solid*.

In Table 17, we observe that posture {2, 3, 10} appears as the best posture of 6 different projects, and the second most emerged postures are {3, -10, 18}. Four of the six projects with posture {2, 3, 10} exhibit a product value in the top one-third, namely project *K3b*, *Plasmanm*, *Gwenview*, and *Umbrello*. We find these projects possess feature values in strong proximity, as shown in Table 2. First, the *Average Gap Day between Consecutive Commits* of every one of the four projects is close to 0.9 days ---1.0760 days for k3b, 1.0047 days for *Gwenview*, 0.9244 days for *Plasmanm*, and 0.7830 days for *Umbrello*. Second, they possess *Average Commits per File Participate* values ranging from 1.3737 to 1.7689 commits and range from the seventh highest to the thirteenth highest.

6.3.2.4 Combination of Posture and Coverage

As previously discussed, an issue encountered in Analysis 2 is a posture's *occurring rate*, and *P-BFC occurring rate* might not properly reflect its general closeness with BFCs, as a P-BFC can contain multiple occurrences of it. Moreover, the product of the two rates is too vague to reveal the predictability of a posture. Hence, we developed a more advanced methodology to evaluate the results, which is based on the number of unique P-BFCs a few postures combinedly occurs in. More specifically, from the results of Analysis 2, each abnormal posture is known to occur in a set of P-BFCs, or, in other words, to have a certain P-BFC coverage. The combined P-BFC coverage of a posture combination can be obtained by taking the union of all P-BFC sets of single posture coverage in the combination, and the coverage ratio of the posture combination can be obtained by dividing the cardinality of the union coverage set by the cardinality of all P-BFCs set.

The highest posture combination coverage ratio of each project is computed following a two-step approach. In the first step, we compute the individual P-BFC coverage of the best predicting abnormal posture of each project, as shown in Table 12, Table 14, and Table 16. In the second step, we compute the combined coverage ratio of every posture combination of sizes two and three. With the increased size of the combination, the coverage ratios of the size-three combination should be significantly higher than those of the size two, but the actual difference between them is neglectable. We present the coverage ratios of the size-two posture combination of each strategy in the following tables and append those of the size-three posture combination Appendix A to Appendix C. Table 18 to Table 20 contain the highest coverage ratio and its yielding posture combination of each project based on different P-BFC strategies. We leverage information obtained in Table 2, Table 13, Table 15, and Table 17 and other calculated statistics to evaluate and perceive the results.

Table 20: Highest Combined Coverage Ratio Using Strategy-1

Project Name	**Posture Combination**	**Coverage Ratio**
solid	{-3,-10,18},{-5,-8,-15}	0.7215
elisa	{-3,-10,18},{-4,-12,-15}	0.7095
kmix	{-4,-5,-12},{-3,-10,18}	0.7085

plasmanm	{-3,-10,18},{-6,-7,-8}	0.7067
marble	{-3,-10,18},{-4,-12,-15}	0.7035
k3b	{-3,-10,18},{-4,-12,-15}	0.7025
kget	{-3,-10,18},{-6,-7,-8}	0.7025
kolourpaint	{-3,-10,18},{-4,-12,-15}	0.7025
kompare	{-4,-12,-15},{-4,-10,-12}	0.7025
ktorrent	{-4,-5,-12},{-3,-10,18}	0.7025
korganizer	{-4,-5,-12},{-3,-10,18}	0.7012
kontact	{-3,-10,18},{-6,-7,-8}	0.6998
umbrello	{-4,-5,-12},{-3,-10,18}	0.6987
gwenview	{-3,-10,18},{-4,-12,-15}	0.6984
juk	{-4,-5,-12},{-3,-10,18}	0.6981
akregator	{-3,-10,18},{-4,-12,-15}	0.6976
kdevplatform	{-4,-5,-12},{-1,-10,18}	0.6976
ark	{-3,-10,18},{-4,-12,-15}	0.6961
kmail	{-4,-5,-12},{-3,-10,18}	0.696
konversation	{-3,-10,18},{-4,-12,-15}	0.6919
ktimetracker	{-3,-10,18},{-4,-12,-15}	0.6900
lokalize	{-4,-5,-12},{-5,-8,-15}	0.6720
clazy	{-2,-5,-12},{-4,-10,-12}	0.6565
kstars	{-4,-12,-15},{-2,-10,-12}	0.6264
kopete	{-5,-8,-15},{-15,-16,-18}	0.5450

Table 21: Highest Combined Coverage Ratio Using Strategy-2

Project Name	Posture Combination	Coverage Ratio
lokalize	{-4,-5,-12},{-5,-12,-16}	0.7355
plasmanm	{-6,-7,-8},{-5,-12,-16}	0.7250
ktorrent	{-4,-5,-12},{-7,-8,-12}	0.7191
umbrello	{-4,-5,-12},{-6,-7,-8}	0.7191
juk	{-4,-5,-12},{-5,-8,-15}	0.7158
kmix	{-4,-5,-12},{-7,-8,-12}	0.7127
korganizer	{-4,-5,-12},{-6,-7,-8}	0.7118
kmail	{-4,-5,-12},{-7,-8,-12}	0.7110
kget	{-4,-5,-12},{-6,-7,-8}	0.7030
kdevplatform	{-4,-5,-12},{-7,-8,-12}	0.6974
k3b	{2,3,10},{-7,-8,-12}	0.6908
elisa	{-5,-12,-16},{-8,-15,-18}	0.6891
kstars	{-4,-5,-12},{-7,-8,-12}	0.6869
gwenview	{-4,-5,-12},{-7,-8,-12}	0.6786
kopete	{-5,-8,-15},{-8,-15,-18}	0.6761

ark	{-4,-5,-12},{-7,-8,-12}	0.6720
kompare	{-6,-7,-8},{-7,-8,-12}	0.6706
ktimetracker	{-4,-5,-12},{-7,-8,-12}	0.6538
marble	{-4,-5,-12},{-5,-12,-16}	0.6538
clazy	{-4,-5,-12},{-5,-12,-16}	0.6437
konversation	{-4,-5,-12},{-5,-12,-16}	0.6356
kontact	{-4,-5,-12},{-7,-8,-12}	0.6183
akregator	{-4,-5,-12},{1,10,-18}	0.5991
solid	{-5,-8,-15},{-8,-15,-18}	0.5878
kolourpaint	{3,10,14},{-8,-15,-18}	0.4167

Table 22: Highest Combined Coverage Ratio Using Strategy-3

Project Name	Posture Combination	Coverage Ratio
solid	{-5,-8,-15},{-8,10,-18}	0.7568
kmail	{-4,-5,-12},{-12,-16,-18}	0.6875
plasmanm	{-5,-8,-15},{-12,-16,-18}	0.6686
juk	{-4,-5,-12},{-8,10,-18}	0.6646
kmix	{-4,-5,-12},{-12,-16,-18}	0.6525
konversation	{-4,-5,-12},{-12,-16,-18}	0.6061
korganizer	{-4,-5,-12},{-12,-16,-18}	0.6018
kget	{-4,-5,-12},{-12,-16,-18}	0.6000
kstars	{-4,-5,-12},{-12,-16,-18}	0.5939
kdevplatform	{-4,-5,-12},{-12,-16,-18}	0.5878
kontact	{-4,-5,-12},{-12,-16,-18}	0.5811
ktimetracker	{-4,-5,-12},{-12,-16,-18}	0.5714
gwenview	{-4,-5,-12},{-8,10,-18}	0.5669
kolourpaint	{3,10,-18},{-12,-16,-18}	0.5591
kompare	{-8,10,-18},{-12,-16,-18}	0.5500
elisa	{-4,-5,-12},{-12,-16,-18}	0.5347
ark	{-8,10,-18},{-12,-16,-18}	0.5342
marble	{-4,-5,-12},{-12,-16,-18}	0.5316
umbrello	{-4,-5,-12},{-12,-16,-18}	0.5149
lokalize	{-4,-5,-12},{-12,-16,-18}	0.5072
k3b	{-8,-15,-18},{-12,-16,-18}	0.4926
akregator	{-4,-5,-12},{-12,-16,-18}	0.4653
kopete	{-5,-8,-15},{-8,-15,-18}	0.4505
ktorrent	{-4,-5,-12},{-12,-16,-18}	0.4065
clazy	{-4,-5,-12},{-12,-16,-18}	0.3271

In the combined coverage ratios using Strategy-1 in Table 18, we observe posture {-3,-10,18} emerges as a part of the posture combination that produces the highest combined coverage ratio for 19 projects out of 25. At the same time, it has only an instance count of 3 in Table 13. However, the posture {-4,-5,-12} with the greatest instance count (i.e. 6) in Table 13 emerges only eight times in Table 18. The result of Strategy-3 presented in Table 20 demonstrates a similar situation: posture {2,3,10}, which is the greatest posture in Table 17, doesn't appear in any posture combination, whereas the most emerged posture {-12,-16,-18} has only instance count of 1. These observations reaffirm that the *P-BFC occurring rate* might not properly reveal the co-occurrence of postures and BFCs, and hence the coverage ratio is a better posture evaluation method.

The averages of best coverage ratios for Strategy-1, Strategy-2, and Strategy-3 are 0.6891, 0.6689, and 0.5605, respectively. The standard deviations of the three strategies are 0.0354, 0.0657, and 0.0923, respectively. Based on these statistics, Strategy-1 is the best-performed strategy with the highest average and the lowest standard deviation. Given that Strategy-1 treats the immediate past commits of fixed length as P-BFC, we can conclude that abnormal postures are more likely to be found right before a bug is fixed.

Although no posture combination predominates results of all three strategies, the single abnormal posture {-4,-5,-12} is considered as the best-performing posture considering all strategies as it appears 8 times, 18 times, and 18 times in Strategy-1, Strategy-2, and Strategy-3, respectively. The metrics of the postures are *file additions, file deletions*, and *File total LOC delta*, all of which are file-associated metrics. Considering they are all in abnormal type *very small*, it means a BFC is often preceded by a negligible modification made to some files. This conclusion coincides partially with those drawn in [41] [5] [8] [6]. We also observe that the abnormal type of metrics that made up posture combinations in the above tables is predominantly *very small*. Moreover, the metrics proposed in this thesis appear in many posture combinations, indicating that these metrics have good potentials to be applied to other related studies.

6.4 Discussion of the Results

6.4.1 Strategy-1

In Analysis-1, posture {-3, -10, 18} is found to be the most P-BFC occurring posture for 13 projects out of 25 with an average occurring rate of 0.3944. Combining the metrics of the postures and corresponding abnormal type together, posture {-3, -10, 18} depicts a situation in which a file has predominant *file-project total LOC delta weight* in commits where very few *changed files* are involved, and very few *project total LOC delta* has resulted. The average indicates approximately 40% of the BFCs in these projects are preceded by such a situation. Additionally, Strategy-1 is more suitable to be applied to a project with a low *BFC per commit* status.

The posture {-3, -10, 18} of Analysis-1 coincides with the featured posture combination of Analysis-2. The predominant posture combination of Analysis-2 is {-3, -10, 18} and {-4, -12, -15}, and it produces the highest BFC coverage ratio in 9 projects with an average coverage ratio of 0.6991. The situation depicted by posture {-3, -10, 18} has been described above, and posture {-4, -12, -15} depicts a situation in which a file has very few *file additions*, *file total LOC delta*, and *file total LOC delta percent change*. The average indicates that approximately 70% of the occurrence of the situation depicted by posture {-3, -10, 18} or {-4, -12, -15} is succeeded by a BFC.

6.4.2 Strategy-2

There is no posture which significantly overwhelms others in Analysis-1 using Strategy-2, but the posture with the most instances is again {-3, -10, 18}, which is found in 6 projects with an average P-BFC occurring rate of 0.4001, and it means approximately 40% of the BFCs in these projects is preceded by such a situation. Additionally, Strategy-2 is more suitable to be applied to a project with a low *Average File Count per Commit*.

The predominant posture combination of Analysis-2 using Strategy-2 consists of {-4, -5, -12} and {-7, -8, -12}, which produces the highest BFC coverage ratio in 9 projects with an average coverage ratio of 0.6833. The depicted situation of posture {-4, -5, -12} has been described above, and posture {-7, -8, -12} depicts a situation in which a file has very low *fractal value over lines*, has been modified by very few *distinct authors to now*, and is modified by very low *file total LOC delta*. The average indicates that approximately 68% of the occurrence of the situation depicted by posture {-4, -5, -12} or {-7, -8, -12} in the 9 projects is succeeded by a BFC.

6.4.3 Strategy-3

In Analysis-1 using Strategy-2, posture {-3, -10, 18} has the most instances of 7 with an average P-BFC occurring rate of 0.6292, and it means approximately 63% of the BFCs in these projects are preceded by such a situation. Furthermore, Strategy-3 is more suitable to be applied to a project with a high *BFC per commit*.

The predominant posture combination of Analysis-2 using Strategy-3 consists of {-4,-5,-12} and {-12,-16,-18}, which produces the highest BFC coverage ratio in 16 projects out of 25 projects with an average of 0.5481. Moreover, 7 of the rest 9 projects show a posture combination has either {-4, -5, -12} or {-12, -16, -18}. Posture {-12, -16, -18} depicts a situation in which a file in a commit has a very low *file total LOC delta*, *file total LOC delta weight*, and *file-project total LOC delta weight*. The average indicates that approximately 55% of the occurrence of the situation depicted by posture {-4, -5, -12} or {-12, -16, -18} in the 16 projects is succeeded by a BFC.

6.5 Threats to Validity

The following aspects are considered threats to the validity of this experiment.

The first point deals with the reconciliation process. The heuristic algorithm we used to label whether a commit is buggy or not relies on the data integrity from both Bugzilla and GitHub.

While GitHub’s data integrity is trustworthy, Bugzilla’s is not guaranteed. Besides, the algorithm used a lookup timeframe size of one month. However, it is possible for the lag to exceed this frame which introduces inaccuracy to the label.

The second point deals with the reality of code churn. The proposed metrics are based on the relative code churn history provided by GitHub, but many operations can result in churn in code other than the logic LOC we expect to be measured. For example, a code restructure that is as simple as moving a function to a different location in the same file will also be recorded as a file deletion and a file addition of the length of that function. Such code churns act like noise to the proposed techniques.

The third point deals with the integrity of bug information. Many projects we selected were initiated around the 2000s. We are unaware of if they started to report bugs to Bugzilla immediately after the beginning of development. Moreover, many other well-known bug-tracking systems, such as Jira, ClickUp, slack, etc., are available in the market, and we are not sure if the development team uses a secondary bug-tracking system. These situations result in an incomplete list of bugs, and consequently, the results may be skewed.

Chapter 7

7 Conclusion and Future Work

7.1 Conclusion

In this thesis, we presented a technique and its implementation in order to analyze the quality posture of system modules in the period leading to a bug-fixing commit. The concept behind this work is that once a bug-inducing commit, which manifests a system failure, occurs, then the system modules which participate in this bug-inducing commit (BIC) are considered to be in a low-quality posture. From that point on, these modules enter an intensive corrective maintenance phase until the bug is fixed, as this is manifested by a bug-fixing commit (BFC). We refer to the period between a BIC and a BFC as the pre-BFC (P-BFC) period, and the main objective of the analysis presented in this thesis is to examine whether there are any metrics patterns (i.e. quality posture) that indicate the transition of a module form a low-quality posture to a high-quality posture. We have considered two main analysis facets. The first facet was to evaluate the conditional frequency rate, *CFR (posture pattern | BFC),* of each posture pattern given that we will have an imminent BFC (i.e. next 5 or 10 commits). The second facet deals with the evaluation of the conditional frequency rate, *CFR (BFC | a posture pattern),* of a BFC that will imminently occur (i.e. in the next 5 or 10 commits) given that we observe a posture pattern occurs. In this thesis, we proposed a suite of source code as well as process metrics and a technique to discretize the metrics values to categorical values. More specifically, the approach is based on: a) converting selected datasets into categorical values in order to identify extreme values which are manifestations of an existing defect; b) compiling the set of pre-bug-fixing commits, which we refer as P-BFC, took place prior to each bug-fixing commits is identified through designated strategies; c) computing the frequency of each metric combination within P-BFC as well as its frequency rate (evaluated as a frequency of cases); d) measuring the occurrence of each combination in all commits and e) calculating the number of instances within P-BFC as well as its frequency rate. The two complementary selection processes together can be used as a metric validation tool which we used to validate the proposed metrics.

From our analysis, we draw the following conclusions. For predicting the occurrence of a pattern given an imminent bug-fixing commit, the metrics pattern that exhibits the highest prediction potential is the combination of the low value of the *Number of Changed files*, the low value of the *Project total LOC delta*, and the high value of *File-project total LOC delta weight*. With respect to predicting an imminent bug-fixing commit (i.e. improvement of quality posture) given a quality posture pattern, the pattern with the highest prediction capability is the one that includes the low value for the *Number of File Additions,* low value for the *Number of File Deletions,* and low value of *File total LOC delta*. Stakeholders of open-source software following a CSE paradigm can use the results to infer the general quality status with respect to bug-fixing commits of the system. Moreover, researchers of maintainability prediction models can consider the metrics proposed in the thesis as features.

7.2 Future Work

The work presented in this thesis can be extended in the following directions. The first extension is to design more advanced P-BFC selection strategies. P-BFC contains bug-inducing commits which influence the results profoundly, so we proposed three strategies to identify the P-BFCs. Strategy-1 is an intuitive strategy that chooses the direct history of each bug-fixing commit as the P-BFC, whereas Strategy-2 and 3 are buggy-file-based approaches. However, Strategy-2 and 3's results show no significant differences. The proposed technique is expected to yield better results under better strategies. In this respect, one could also consider tuning certain parameters of the experiments, such as the length of the P-BFC period and what commits are considered relevant to the BFC. A third extension is to consider more metrics which can be harvested by either considering dependencies between modules or the volume of functionality delivered by a module. Such metrics may deal with Information Flow, Coupling, and Cohesion.

8 References

[1] K. Naik and P. Tripathy, *Software Testing and Quality Assurance: Theory and Practice*. Somerset: Wiley, 2011.

[2] N. U. Eisty, G. K. Thiruvathukal, and J. C. Carver, “A Survey of Software Metric Use in Research Software Development,” in *2018 IEEE 14th International Conference on e-Science (e-Science)*, Amsterdam: IEEE, Oct. 2018, pp. 212–222. doi: 10.1109/eScience.2018.00036.

[3] H. D. Frederiksen and L. Mathiassen, “A Contextual Approach to Improving Software Metrics Practices,” *IEEE Trans. Eng. Manage.*, vol. 55, no. 4, pp. 602–616, Nov. 2008, doi: 10.1109/TEM.2008.2005547.

[4] M. Shepperd, “A critique of cyclomatic complexity as a software metric,” *Softw. Eng. J. UK*, vol. 3, no. 2, p. 30, 1988, doi: 10.1049/sej.1988.0003.

[5] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, “When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits,” in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Baltimore, Maryland: IEEE, Oct. 2013, pp. 65–74. doi: 10.1109/ESEM.2013.19.

[6] A. Hindle, D. M. German, and R. Holt, “What do large commits tell us?: a taxonomical study of large commits,” in *Proceedings of the 2008 international working conference on Mining software repositories*, Leipzig Germany: ACM, May 2008, pp. 99–108. doi: 10.1145/1370750.1370773.

[7] C. Andersson and P. Runeson, “A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems,” *IIEEE Trans. Software Eng.*, vol. 33, no. 5, pp. 273–286, May 2007, doi: 10.1109/TSE.2007.1005.

[8] L. An and F. Khomh, “An Empirical Study of Crash-inducing Commits in Mozilla Firefox,” in *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, Beijing China: ACM, Oct. 2015, pp. 1–10. doi: 10.1145/2810146.2810152.

[9] H. Zhang, “An investigation of the relationships between lines of code and defects,” in *2009 IEEE International Conference on Software Maintenance*, Edmonton, AB, Canada: IEEE, Sep. 2009, pp. 274–283. doi: 10.1109/ICSM.2009.5306304.

[10] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *Proceedings. 27th International Conference on Software*

Engineering, 2005. ICSE 2005., St. Louis, MO, USA: IEEe, 2005, pp. 284–292. doi: 10.1109/ICSE.2005.1553571.

[11] B. W. Boehm, J. R. Brown, and M. Lipow, “Quantitative evaluation of software quality,” in *Proceedings of the 2nd international conference on Software engineering*, Oct. 1976, pp. 592–605.

[12] J. A. McCall, P. K. Richards, and G. F. Walters, “Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality,” ROME AIR DEVELOPMENT CENTER, Griffiss Air Force Base, New York 13441, Final Technical Report, Nov. 1977.

[13] D. Coleman, D. Ash, B. Lowther, and P. Oman, “Using metrics to evaluate software system maintainability,” *Computer*, vol. 27, no. 8, pp. 44–49, Aug. 1994, doi: 10.1109/2.303623.

[14] J. Bansiya and C. G. Davis, “A hierarchical model for object-oriented design quality assessment,” *IIEEE Trans. Software Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002, doi: 10.1109/32.979986.

[15] L. Kumar and S. K. Rath, “Software maintainability prediction using hybrid neural network and fuzzy logic approach with parallel computing concept,” *Int J Syst Assur Eng Manag*, vol. 8, no. S2, pp. 1487–1502, Nov. 2017, doi: 10.1007/s13198-017-0618-4.

[16] M. O. Elish, H. Aljamaan, and I. Ahmad, “Three empirical studies on predicting software maintainability using ensemble methods,” *Soft Comput*, vol. 19, no. 9, pp. 2511–2524, Sep. 2015, doi: 10.1007/s00500-014-1576-2.

[17] “IEEE Standard Classification for Software Anomalies,” IEEE. doi: 10.1109/IEEESTD.2010.5399061.

[18] IEEE Computer Society, Software Engineering Standards Subcommittee, Institute of Electrical and Electronics Engineers, and IEEE Standards Board, *IEEE standard for a software quality metrics methodology*. New York, N.Y.: Institute of Electrical and Electronics Engineers, 1993.

[19] T. Honglei, S. Wei, and Z. Yanan, “The Research on Software Metrics and Software Complexity Metrics,” in *2009 International Forum on Computer Science-Technology and Applications*, Chongqing, China: IEEE, 2009, pp. 131–136. doi: 10.1109/IFCSTA.2009.39.

[20] S. S. Rathore and S. Kumar, “A study on software fault prediction techniques,” *Artif Intell Rev*, vol. 51, no. 2, pp. 255–327, Feb. 2019, doi: 10.1007/s10462-017-9563-5.

[21] E. E. Mills, “Software metrics,” Carnegie-Mellon University Software Engineering Ins., Pittsburgh, PA, 1988.

[22] Y. Gil and G. Lalouche, “On the correlation between size and metric validity,” *Empir Software Eng*, vol. 22, no. 5, pp. 2585–2611, Oct. 2017, doi: 10.1007/s10664-017-9513-5.

[23] M. Yan, X. Xia, X. Zhang, L. Xu, D. Yang, and S. Li, “Software quality assessment model: a systematic mapping study,” *Sci. China Inf. Sci.*, vol. 62, no. 9, p. 191101, Sep. 2019, doi: 10.1007/s11432-018-9608-3.

[24] G. Yenduri and T. R. Gadekallu, “A Systematic Literature Review of Soft Computing Techniques for Software Maintainability Prediction: State-of-the-Art, Challenges and Future Directions,” 2022, doi: 10.48550/ARXIV.2209.10131.

[25] A. J. Albrecht, “Measuring application development productivity.” Proc. Joint Share, Guide, and IBM Application Development Symposium, Oct. 1979.

[26] N. E. Fenton and N. Ohlsson, “Quantitative analysis of faults and failures in a complex software system,” *IIEEE Trans. Software Eng.*, vol. 26, no. 8, pp. 797–814, Aug. 2000, doi: 10.1109/32.879815.

[27] N. E. Fenton and M. Neil, “A critique of software defect prediction models,” *IIEEE Trans. Software Eng.*, vol. 25, no. 5, pp. 675–689, Oct. 1999, doi: 10.1109/32.815326.

[28] J. Rosenberg, “Some misconceptions about lines of code,” in *Proceedings Fourth International Software Metrics Symposium*, Albuquerque, NM, USA: IEEE Comput. Soc, 1997, pp. 137–142. doi: 10.1109/METRIC.1997.637174.

[29] T. J. McCabe, “A Complexity Measure,” *IIEEE Trans. Software Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976, doi: 10.1109/TSE.1976.233837.

[30] D. Afriyie and Y. Labiche, “Predictors of Software Metric Correlation: A Non-parametric Analysis,” in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, Hainan, China: IEEE, Dec. 2021, pp. 524–533. doi: 10.1109/QRS54544.2021.00063.

[31] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, and C. Soubervielle-Montalvo, “Source code metrics: A systematic mapping study,” *Journal of Systems and Software*, vol. 128, pp. 164–197, Jun. 2017, doi: 10.1016/j.jss.2017.03.044.

[32] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object-oriented design,” *IIEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994, doi: 10.1109/32.295895.

[33] S. Purao and V. Vaishnavi, “Product metrics for object-oriented systems,” *ACM Comput. Surv.*, vol. 35, no. 2, pp. 191–221, Jun. 2003, doi: 10.1145/857076.857090.

[34] A. A. E. Ibrahim, A. Kamel, and H. Hassan, “Object Oriented Metrics and Quality Attributes: A Survey,” in *Proceedings of the 10th International Conference on Informatics and Systems - INFOS ’16*, Giza, Egypt: ACM Press, 2016, pp. 312–319. doi: 10.1145/2908446.2908468.

[35] Y. Singh, A. Kaur, and R. Malhotra, “Empirical validation of object-oriented metrics for predicting fault proneness models,” *Software Qual J*, vol. 18, no. 1, pp. 3–35, Mar. 2010, doi: 10.1007/s11219-009-9079-6.

[36] T. Gyimothy, R. Ferenc, and I. Siket, “Empirical validation of object-oriented metrics on open source software for fault prediction,” *IIEEE Trans. Software Eng.*, vol. 31, no. 10, pp. 897–910, Oct. 2005, doi: 10.1109/TSE.2005.112.

[37] R. Xu, Y. Xue, P. Nie, Y. Zhang, and D. Li, “Research on CMMI-based Software Process Metrics,” in *First International Multi-Symposiums on Computer and Computational Sciences (IMSCCS’06)*, Hangzhou, Zhejiang, China: IEEE, Jun. 2006, pp. 391–397. doi: 10.1109/IMSCCS.2006.260.

[38] A. Hidayati, B. Purwandari, E. K. Budiardjo, and I. Solichah, “Global Software Development and Capability Maturity Model Integration: A Systematic Literature Review,” in *2018 Third International Conference on Informatics and Computing (ICIC)*, Palembang, Indonesia: IEEE, Oct. 2018, pp. 1–6. doi: 10.1109/IAC.2018.8780489.

[39] W. Rhmann, B. Pandey, G. Ansari, and D. K. Pandey, “Software fault prediction based on change metrics using hybrid algorithms: An empirical study,” *Journal of King Saud University - Computer and Information Sciences*, vol. 32, no. 4, pp. 419–424, May 2020, doi: 10.1016/j.jksuci.2019.03.006.

[40] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, “Change Bursts as Defect Predictors,” in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, San Jose, CA, USA: IEEE, Nov. 2010, pp. 309–318. doi: 10.1109/ISSRE.2010.25.

[41] T. Illes-Seifert and B. Paech, “Exploring the relationship of a file’s history and its fault-proneness: An empirical method and its application to open source programs,” *Information and Software Technology*, vol. 52, no. 5, pp. 539–558, May 2010, doi: 10.1016/j.infsof.2009.11.010.

[42] S. S. Hossain, P. Ahmed, and Y. Arafat, “Software Process Metrics in Agile Software Development: A Systematic Mapping Study,” in *Computational Science and Its*

Applications – ICCSA 2021, O. Gervasi, B. Murgante, S. Misra, C. Garau, I. Blečić, D. Taniar, B. O. Apduhan, A. M. A. C. Rocha, E. Tarantino, and C. M. Torre, Eds., in Lecture Notes in Computer Science, vol. 12957. Cham: Springer International Publishing, 2021, pp. 15–26. doi: 10.1007/978-3-030-87013-3_2.

[43] S. Majumder, P. Mody, and T. Menzies, "Revisiting process versus product metrics: a large scale analysis," *Empir Software Eng*, vol. 27, no. 3, p. 60, May 2022, doi: 10.1007/s10664-021-10068-4.

[44] N. Nagappan and T. Ball, "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007),* Madrid, Spain: IEEE, Sep. 2007, pp. 364–373. doi: 10.1109/ESEM.2007.13.

[45] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Information and Software Technology*, vol. 55, no. 8, pp. 1397–1418, Aug. 2013, doi: 10.1016/j.infsof.2013.02.009.

[46] S. Matsumoto, Y. Kamei, A. Monden, K. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE '10*, Timi&#351;oara, Romania: ACM Press, 2010, p. 1. doi: 10.1145/1868328.1868356.

[47] Y. Jiang, B. Cukic, and Y. Ma, "Techniques for evaluating fault prediction models," *Empir Software Eng*, vol. 13, no. 5, pp. 561–595, Oct. 2008, doi: 10.1007/s10664-008-9079-3.

[48] R. Premraj and K. Herzig, "Network Versus Code Metrics to Predict Defects: A Replication Study," in *2011 International Symposium on Empirical Software Engineering and Measurement*, Banff, AB, Canada: IEEE, Sep. 2011, pp. 215–224. doi: 10.1109/ESEM.2011.30.

[49] T. Nakamura and V. Basili, "Metrics of Software Architecture Changes Based on Structural Distance," in *11th IEEE International Software Metrics Symposium (METRICS'05)*, Como: IEEE, 2005, pp. 24–24. doi: 10.1109/METRICS.2005.35.

[50] S. Chulani, B. Ray, P. Santhanam, and R. Leszkowicz, "Metrics for managing customer view of software quality," in *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No.03EX717)*, Sydney, NSW, Australia: IEEE Comput. Soc, 2003, pp. 189–198. doi: 10.1109/METRIC.2003.1232467.

[51] H. Washizaki, H. Yamamoto, and Y. Fukazawa, “A metrics suite for measuring reusability of software components,” in *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No.03EX717)*, Sydney, NSW, Australia: IEEE Comput. Soc, 2003, pp. 211–223. doi: 10.1109/METRIC.2003.1232469.

[52] G. Jay, J. E. Hale, R. K. Smith, D. Hale, N. A. Kraft, and C. Ward, “Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship,” *JSEA*, vol. 02, no. 03, pp. 137–143, 2009, doi: 10.4236/jsea.2009.23020.

[53] M. A. A. Mamun, C. Berger, and J. Hansson, “Correlations of software code metrics: an empirical study,” in *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, Gothenburg Sweden: ACM, Oct. 2017, pp. 255–266. doi: 10.1145/3143434.3143445.

[54] J. Lee Rodgers and W. A. Nicewander, “Thirteen Ways to Look at the Correlation Coefficient,” *The American Statistician*, vol. 42, no. 1, pp. 59–66, Feb. 1988, doi: 10.1080/00031305.1988.10475524.

[55] J. Benesty, Jingdong Chen, and Yiteng Huang, “On the Importance of the Pearson Correlation Coefficient in Noise Reduction,” *IEEE Trans. Audio Speech Lang. Process.*, vol. 16, no. 4, pp. 757–765, May 2008, doi: 10.1109/TASL.2008.919072.

[56] J. Benesty, Ed., *Noise reduction in speech processing*. in Springer topics in signal processing, no. v. 2. Dordrecht ; London ; New York: Springer, 2009.

[57] E. Borandag, A. Ozcift, D. Kilinc, and F. Yucalar, “Majority vote feature selection algorithm in software fault prediction,” *ComSIS*, vol. 16, no. 2, pp. 515–539, 2019, doi: 10.2298/CSIS180312039B.

[58] L. Kumar, S. K. Sripada, A. Sureka, and S. Ku. Rath, “Effective fault prediction model developed using Least Square Support Vector Machine (LSSVM),” *Journal of Systems and Software*, vol. 137, pp. 686–712, Mar. 2018, doi: 10.1016/j.jss.2017.04.016.

[59] H. Ji, S. Huang, Y. Wu, Z. Hui, and C. Zheng, “A new weighted naive Bayes method based on information diffusion for software defect prediction,” *Software Qual J*, vol. 27, no. 3, pp. 923–968, Sep. 2019, doi: 10.1007/s11219-018-9436-4.

[60] C. Zhi *et al.*, “Quality Assessment for Large-Scale Industrial Software Systems: Experience Report at Alibaba,” in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, Putrajaya, Malaysia: IEEE, Dec. 2019, pp. 142–149. doi: 10.1109/APSEC48747.2019.00028.

[61] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, “Software Defect Prediction via Attention-Based Recurrent Neural Network,” *Scientific Programming*, vol. 2019, pp. 1–14, Apr. 2019, doi: 10.1155/2019/6230953.

[62] A. Boucher and M. Badri, “Predicting Fault-Prone Classes in Object-Oriented Software: An Adaptation of an Unsupervised Hybrid SOM Algorithm,” in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Prague, Czech Republic: IEEE, Jul. 2017, pp. 306–317. doi: 10.1109/QRS.2017.41.

[63] G. Abaei, A. Selamat, and H. Fujita, “An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction,” *Knowledge-Based Systems*, vol. 74, pp. 28–39, Jan. 2015, doi: 10.1016/j.knosys.2014.10.017.

[64] S. K. Pandey, R. B. Mishra, and A. K. Tripathi, “BPDET: An effective software bug prediction model using deep representation and ensemble learning techniques,” *Expert Systems with Applications*, vol. 144, p. 113085, Apr. 2020, doi: 10.1016/j.eswa.2019.113085.

[65] J. Wang and C. Zhang, “Software reliability prediction using a deep learning model based on the RNN encoder–decoder,” *Reliability Engineering & System Safety*, vol. 170, pp. 73–82, Feb. 2018, doi: 10.1016/j.ress.2017.10.019.

[66] “GitHub.” https://github.com/ (accessed Jul. 20, 2022).

[67] P. K. Korlepara, “Fuzzy and Probabilistic Rule-Based Approaches to Identify Fault Prone Files,” Western University, London, ON, 2022. [Online]. Available: https://ir.lib.uwo.ca/etd/7941

[68] Ria, “A Technique for Evaluating the Health Status of a Software Module Using Process Metrics,” Western University, London, ON, 2021. [Online]. Available: https://ir.lib.uwo.ca/etd/7855

[69] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, “An in-depth study of the promises and perils of mining GitHub,” *Empir Software Eng*, vol. 21, no. 5, pp. 2035–2071, Oct. 2016, doi: 10.1007/s10664-015-9393-5.

[70] M. S. - and O. M. H. R. -, “Selection Criteria of Open Source Software: First Stage for Adoption,” *IJIPM*, vol. 4, no. 4, pp. 51–58, Jun. 2013, doi: 10.4156/ijipm.vol4.issue4.6.

[71] K. Kontogiannis, N. Patel, and M. Grigoriou, “Continuous Compliance Data Science for Software Systems,” Western University, London, ON, 2020. [Online]. Available: https://www-40.ibm.com/ibm/cas/canada/projects?projectId=1130

[72] J. Wilhelm, “categorization method,” *How do I categorize raw data into categories “Low,” “Average,” and “High”?*, Aug. 30, 2022. https://www.researchgate.net/post/How_do_I_categorize_raw_data_into_categories_Low_Average_and_High (accessed Oct. 06, 2022).

9 Appendices

Appendix A: Highest Combined Coverage Ratio Using Strategy-1

Project Name	Posture Combination	Coverage Ratio
solid	{-3,-10,18},{-5,-8,-15},{-15,-16,-18}	0.7215
kompare	{-4,-12,-15},{-4,-10,-12},{-8,-9,-14}	0.7201
elisa	{-4,-5,-12},{-3,-10,18},{-4,-12,-15}	0.7095
kmix	{-4,-5,-12},{-3,-10,18},{-4,-12,-15}	0.7085
kget	{-3,-10,18},{-4,-12,-15},{-2,-5,-12}	0.7068
plasmanm	{-3,-10,18},{-2,-5,-12},{-15,-16,-18}	0.7067
kontact	{-3,-10,18},{-4,-12,-15},{-6,-7,-8}	0.7052
akregator	{-3,-10,18},{-4,-12,-15},{-2,-5,-12}	0.7049
k3b	{-3,-10,18},{-4,-12,-15},{-8,-9,-14}	0.7046
korganizer	{-4,-5,-12},{-3,-10,18},{-6,-7,-8}	0.7038
marble	{-4,-5,-12},{-3,-10,18},{-4,-12,-15}	0.7035
gwenview	{-3,-10,18},{-4,-12,-15},{-2,-5,-12}	0.7025
kolourpaint	{-3,-10,18},{-4,-12,-15},{3,10,14}	0.7025
ktimetracker	{-4,-5,-12},{-3,-10,18},{-4,-12,-15}	0.7025
ktorrent	{-4,-5,-12},{-3,-10,18},{-4,-12,-15}	0.7025
kdevplatform	{-4,-12,-15},{-2,-5,-12},{-1,-10,18}	0.7009
ark	{-3,-10,18},{-4,-12,-15},{-2,-5,-12}	0.6993
umbrello	{-4,-5,-12},{-3,-10,18},{-4,-12,-15}	0.6987
juk	{-4,-5,-12},{-3,-10,18},{-4,-12,-15}	0.6981
kmail	{-4,-5,-12},{-3,-10,18},{-8,-9,-14}	0.6965
konversation	{-3,-10,18},{-4,-12,-15},{-4,-10,-12}	0.6949
lokalize	{-4,-5,-12},{-4,-10,-12},{-5,-8,-15}	0.672
clazy	{-4,-12,-15},{-2,-5,-12},{-4,-10,-12}	0.6697
kstars	{-4,-12,-15},{-2,-5,-12},{-4,-10,-12}	0.6477
kopete	{-4,-5,-12},{-5,-8,-15},{-15,-16,-18}	0.545

Appendix B: Highest Combined Coverage Ratio Using Strategy-2

Project Name	Posture Combination	Coverage Ratio
lokalize	{-4,-5,-12},{-5,-8,-15},{-5,-12,-16}	0.7355
juk	{-4,-5,-12},{-5,-8,-15},{-5,-12,-16}	0.725
plasmanm	{-4,-5,-12},{-6,-7,-8},{2,10,13}	0.725
umbrello	{-4,-5,-12},{-6,-7,-8},{-5,-12,-16}	0.7211
ktorrent	{-4,-5,-12},{2,3,14},{-5,-12,-16}	0.7191
kmix	{-4,-5,-12},{-5,-8,-15},{-7,-8,-12}	0.7189
korganizer	{-4,-5,-12},{-6,-7,-8},{-5,-8,-15}	0.7131
kmail	{-4,-5,-12},{-5,-12,-16},{-7,-8,-12}	0.7121

kget	{-4,-5,-12},{-6,-7,-8},{-5,-8,-15}	0.7118
kdevplatform	{-4,-5,-12},{-5,-12,-16},{-7,-8,-12}	0.7041
elisa	{-5,-8,-15},{-5,-12,-16},{-8,-15,-18}	0.7035
gwenview	{-4,-5,-12},{2,10,14},{-7,-8,-12}	0.6913
k3b	{3,10,14},{2,3,14},{-7,-8,-12}	0.6908
kstars	{-4,-5,-12},{-5,-12,-16},{-7,-8,-12}	0.6892
ark	{-4,-5,-12},{-7,-8,-12},{-8,-15,-18}	0.6853
kopete	{-4,-5,-12},{-5,-8,-15},{-8,-15,-18}	0.6761
ktimetracker	{-4,-5,-12},{-5,-8,-15},{-7,-8,-12}	0.6732
marble	{-4,-5,-12},{-5,-12,-16},{-8,-15,-18}	0.6721
kompare	{-6,-7,-8},{-7,-8,-12},{-8,-15,-18}	0.6706
konversation	{-4,-5,-12},{3,10,14},{-5,-12,-16}	0.6528
clazy	{-4,-5,-12},{-5,-12,-16},{-8,-15,-18}	0.6505
kontact	{-4,-5,-12},{2,10,14},{-7,-8,-12}	0.6347
akregator	{-4,-5,-12},{1,10,-18},{-5,-12,-16}	0.6293

Appendix C: Highest Combined Coverage Ratio Using Strategy-3

Project Name	Posture Combination	Coverage Ratio
solid	{-5,-8,-15},{-8,10,-18},{-8,-15,-18}	0.7568
kmail	{-4,-5,-12},{-5,-8,-15},{-12,-16,-18}	0.6898
plasmanm	{-5,-8,-15},{-8,10,-18},{-12,-16,-18}	0.6864
juk	{-4,-5,-12},{-5,-8,-15},{-8,10,-18}	0.6835
kmix	{-4,-5,-12},{-5,-8,-15},{-12,-16,-18}	0.661
kontact	{1,2,10},{-4,-5,-12},{-12,-16,-18}	0.6491
konversation	{-4,-5,-12},{-8,-15,-18},{-12,-16,-18}	0.6169
kget	{-4,-5,-12},{-5,-8,-15},{-12,-16,-18}	0.6121
kdevplatform	{-4,-5,-12},{-5,-8,-15},{-12,-16,-18}	0.6074
ktimetracker	{-4,-5,-12},{-5,-8,-15},{-12,-16,-18}	0.6071
korganizer	{-4,-5,-12},{2,10,14},{-12,-16,-18}	0.6036
kstars	{-4,-5,-12},{-5,-8,-15},{-12,-16,-18}	0.6019
gwenview	{-4,-5,-12},{2,10,13},{-12,-16,-18}	0.5959
ark	{-4,-5,-12},{-8,10,-18},{-12,-16,-18}	0.5708
elisa	{-4,-5,-12},{-5,-8,-15},{-12,-16,-18}	0.5644
kolourpaint	{3,10,-18},{3,10,14},{-12,-16,-18}	0.5591
kompare	{-8,10,-18},{-8,-15,-18},{-12,-16,-18}	0.55
marble	{-4,-5,-12},{-5,-8,-15},{-12,-16,-18}	0.5358
umbrello	{-4,-5,-12},{-5,-8,-15},{-12,-16,-18}	0.5203
lokalize	{-4,-5,-12},{-5,-8,-15},{-12,-16,-18}	0.5072
akregator	{1,2,10},{-4,-5,-12},{-12,-16,-18}	0.4965
k3b	{-8,10,-18},{-8,-15,-18},{-12,-16,-18}	0.4926

kopete	{-4,-5,-12},{-5,-8,-15},{-8,-15,-18}	0.4505
ktorrent	{3,10,-18},{-4,-5,-12},{-12,-16,-18}	0.4065
clazy	{-4,-5,-12},{2,10,13},{-12,-16,-18}	0.3364