2015

MiSFIT: Mining Software Fault Information and Types

Billy R. Kidwell
University of Kentucky, kidwell.bill@gmail.com

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Billy R. Kidwell, Student

Dr. Jane Huffman Hayes, Major Professor

Dr. Miroslaw Truszczynski, Director of Graduate Studies

MISFIT

MINING SOFTWARE FAULT INFORMATION AND TYPES

DISSERTATION

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science in the College of Engineering
at the University of Kentucky

By

Billy R. Kidwell

Lexington, Kentucky

Director: Dr. Jane Huffman Hayes, Professor of Computer Science

Lexington, Kentucky

2015

ABSTRACT OF DISSERTATION

MISFIT
MINING SOFTWARE FAULT INFORMATION AND TYPES

As software becomes more important to society, the number, age, and complexity of systems grow. Software organizations require continuous process improvement to maintain the reliability, security, and quality of these software systems. Software organizations can utilize data from manual fault classification to meet their process improvement needs, but organizations lack the expertise or resources to implement them correctly.

This dissertation addresses the need for the automation of software fault classification. Validation results show that automated fault classification, as implemented in the MiSFIT tool, can group faults of similar nature. The resulting classifications result in good agreement for common software faults with no manual effort.

To evaluate the method and tool, I develop and apply an extended change taxonomy to classify the source code changes that repaired software faults from an open source project. MiSFIT clusters the faults based on the changes. I manually inspect a random sample of faults from each cluster to validate the results. The automatically classified faults are used to analyze the evolution of a software application over seven major releases. The contributions of this dissertation are an extended change taxonomy for software fault analysis, a method to cluster faults by the syntax of the repair, empirical evidence that fault distribution varies according to the purpose of the module, and the identification of project-specific trends from the analysis of the changes.

KEYWORDS: Software Faults, Software Fault Classification, Software Taxonomy, Mining Software Repositories, Software Evolution

Billy R. Kidwell
Student's Signature

April 9, 2015
Date

MISFIT
MINING SOFTWARE FAULT INFORMATION AND TYPES

By

Billy R. Kidwell

Jane Huffman Hayes, Ph.D.
Director of Dissertation

Miroslaw Truszczynski, Ph.D.
Director of Graduate Studies

April 9, 2015
Date

DEDICATION

This dissertation is dedicated to:

My wife, Nora Mae Kidwell, for her limitless patience, love, and support

My son, Alexander Raylan Kidwell, for providing new meaning to life

My parents, Danny and Donna Kidwell, for instilling a strong work ethic, teaching me persistence, and encouraging me to excel at whatever I do

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Jane Hayes, for her guidance and support. She has been an excellent advisor and mentor throughout my time at UK. I thank Dr. Judy Goldsmith, Dr. Zongming Fei, and Dr. Albert Lederer for serving on my committee and providing guidance on this dissertation. I would also like to thank Dr. James Lumpp, Jr. for serving as my outside committee member. Thanks to Dr. Wasilkowski, Dr. Finkel, Dr. Klapper, and Dr. Truszczynski for their assistance as directors of Graduate Study.

Thanks to NASA for partially funding this research. Thanks to Dr. Allen P. Nikora for his input and support during this research. Thanks to Hewlett-Packard for supporting my Ph.D. Thanks to Anthony Wiley, Russ Wolfe, and Rob Guckenberger for supporting my academic efforts as my manager. Thanks to Carla Griesch, Lucas Cockerham, Steve Stogner, Matt Downs, Chris Wells, and all of the other co-workers that have endured conversations about this dissertation.

Thanks to Dr. Davide Falessi for his guidance and mentoring on our research on failure classification. I would like to thank Dr. Robert Gillespie, Dr. William Pierson, Professor Joseph Fuller, and Professor Mike Clark for serving as great professors and role models during my undergraduate studies.

I would like to thank fellow graduate students Wenbin Li, Dr. Hakim Sultanov, Dr. Wei-keat Kong, and Dr. Ashlee Holbrook for their collaboration.

I would like to thank all of the friends and family that have encouraged me during this process. I especially want to thank Kevin Magsig, Bill Woodson, James Klawon, and my father, Danny Kidwell for their encouragement.

I could not have completed this work without the love and support of my wife and best friend, Nora. Thank you for being there for me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction and Overview

Software companies are building increasingly complex systems. At the same time, market pressures require that they do so in less time, while customers are demanding higher quality. Increasingly, today's software teams are distributed across the country, or across the world. Balancing these factors is a major problem for software development organizations. In order to reduce time and increase quality, software organizations must continually improve their software development practices.

The most measurable aspect of software quality is the number of faults, or bugs, that are discovered in a software product. A simple metric to assess the quality of a product might be a count of the faults reported by customers. However, this metric is problematic in at least two ways. First, it does not provide actionable feedback about where improvements can occur and second, it occurs too late to make any corrections.

Software fault classification provides precise feedback about the software development process. Modern fault classification schemes include multiple attributes, such as the severity of the fault, the activity that found the fault, and the type of fault that occurred. If the scheme is carefully designed, the type of fault can provide evidence of when the fault was introduced [1]. The longer the fault goes without detection, the more expensive the fault is to repair [2]. The goal of using fault classification schemes is thus to prevent faults and find as many faults as possible, as early as possible.

Prior research has shown that fault classification has been used successfully to measure and improve the software development process [3], prevent faults [4][5], design tests [6], plan quality assurance activities [7]–[9], and evaluate the effectiveness of quality assurance activities [10][11].

Studies cite a number of different challenges for practitioners. The developer that repaired the fault is required to determine the classification. The use of the fault description and a secondary group, such as the quality assurance

team rather than the developer that fixes the fault, results in low agreement [12]. Fault classification is also dependent on the experience of the classifier [13]. Other studies reported challenges in getting consistent data [5], [14]–[16] and a need to customize fault classification schemes for a domain, organization, or project [5], [17]–[19]. I have seen anecdotal evidence of these challenges in my professional experience as a software engineer. Based on this anecdotal evidence, I believe that these barriers prevent the widespread adoption of fault classification in industry.

Automation is applied to fault classification in several ways. Natural language processing has been used to analyze the text of fault reports and detect duplicates [20], [21]. Duplicate detection increased process efficiency by eliminating wasted work. Automation has also been used to automatically determine if a fault represents corrective maintenance [22], determine the customer impact of a fault [23], and predict the severity of a fault [24], [25].

1.1 Problem Statement

As software becomes more important to society, the number, age, and complexity of systems grow. Software organizations require continuous process improvement to maintain the reliability, security, and quality of these software systems. Software organizations can utilize data from manual fault classification to meet the process improvement needs of organizations, but organizations lack the expertise or resources to implement them correctly. This dissertation addresses the need for the automation of software fault classification. Validation results show that automated fault classification, as implemented in the MiSFIT tool, can group faults of similar nature. The resulting classifications result in good agreement for common software faults with no manual effort. The evolution of faults over seven releases are examined with the aid of the classified fault data.

1.2 Research Thesis

The goal of this research is to provide an automated method to categorize software faults based on the syntactical changes that repair the fault. Specifically, I categorize Java source code changes according to an extended change taxonomy and apply clustering to the results to form a project-specific fault taxonomy.

I present a new method implemented in a tool, MiSFIT (Mining Software Fault Information and Types), which can be utilized to process historical information from software repositories, classify syntactical changes, and cluster software faults. The overall thesis of this research is that software fault classification can be automated by leveraging the information in the source code changes that repair the fault. The use of the method described in this dissertation provides a project-specific taxonomy that evolves with the programming language and the programming practices of the software development team.

To evaluate the thesis, I apply the extended change taxonomy to classify the source code changes that repaired software faults from an open source project. MiSFIT clusters the faults based on the changes. I manually inspect a random sample of faults from each cluster to validate the results. The automatically classified faults are used to analyze the evolution of a software application over seven major releases. The validation results in the following contributions:

- an extended change taxonomy for software fault analysis,
- a method to cluster faults by the syntax of the repair,
- empirical evidence to support prior findings that fault distribution varies according to the purpose of the module [26], and
- project-specific trends identified through the analysis of the changes.

1.3 Scope of the Research

For this project, I restrict my attention to object-oriented systems written in the Java programming language. I limit the investigation of faults to those that

appear in source code. I eliminate from consideration any fault in requirements documents, design models, or documentation that do not appear in the source code.

1.4 Relevance

Software fault classification provides many benefits, but the primary users are software organizations with mature development processes. Software organizations need methods to improve development processes in order to improve quality and reduce time to market. Unfortunately, manual fault classification is expensive to implement correctly. An automated method to classify faults can provide valuable information for improving software development processes.

In addition, many open source software projects are available today and provide researchers with an enormous amount of data that was previously unavailable. The manual classification of the faults in open source projects is difficult. Open source projects are highly dependent on volunteers to contribute to the development effort, and the development processes are immature by software engineering standards. As a result, access to the information to classify software faults retroactively is difficult to obtain. However, the source code and problem reports for these projects are readily available. An automated method of fault classification can provide additional data about the nature of software faults to advance our understanding of software engineering.

1.5 Overview of Dissertation

This section describes the organization of the dissertation. Chapter 2 discusses background information and surveys the current literature on software fault classification. Chapter 3 introduces the MiSFIT tool and presents the research approach. Chapter 4 presents an existing change taxonomy and an extension that makes it adequate for analyzing software faults. Chapter 5 presents the clustering of software faults based on the syntactic information in the fix. Chapter 6 extends this work by examining software faults from seven

versions of an open source software project. Chapter 7 concludes the dissertation and discusses future work.

Chapter 2

Background and Related Work

I begin this chapter with terms, definitions, and background information on the software development lifecycle. Once established I introduce software fault classification by presenting a common fault classification scheme, the Orthogonal Defect Classification (ODC) scheme. The remainder of this chapter is a review of the literature in software fault classification. In this review, I explore the benefits, challenges, and future of software fault classification.

2.1 Terms and Definitions

The IEEE defines a software *fault* as an “incorrect step, process, or data definition in a computer program” [27]. The terms *defect* and fault are often used interchangeably in the literature. An *error* causes the introduction of a software fault in the creation of a software artifact. Faults are introduced in requirements, architecture, design, or source code and may be detected at any stage after introduction, including testing and maintenance of the software. A software fault remains latent until a set of operating conditions or inputs trigger the fault, causing the fault to manifest itself as a *failure*.

A software *failure* is the failure of a software system to operate within the specifications of that system. The failure may be an incorrect output, system crash, or a failure to perform its operations under non-functional constraints related to performance, security, or availability. The cause of software failures can be complex. In some cases, failures are difficult to reproduce. Failures may only occur in rare conditions, or one fault may hide the existence of another. When this occurs, fixing a fault may appear to introduce a new fault, when in fact it reveals a hidden fault. A better understanding of the complex relationship between faults and failures is an open area of research and essential for improving the prevention and detection of software faults [28].

The term *bug* is often used in industry as a synonym for a software fault, failure, or error. Due to the ambiguous nature of the term, this dissertation avoids its use as much as possible.

A failure is documented in a database that is used within the software development organization. This database is referred to by terms such as issue tracking system, bug tracking system, or problem tracking system. This dissertation refers to the database as a problem tracking system, and to a single report of a failure or possible failure as a problem report. Practitioners attempt to keep each problem report isolated to a single failure, but this is not always possible. In practice, the source code fix for a single problem report may address a number of related issues that are uncovered during the investigation and repair of the issue. In extreme cases, changes may need to occur to the architecture or high-level design to address a fundamental flaw or changing need of the system. The problem report is a record of a failure, including its detection, investigation, and repair.

This section has provided terms and definitions that are useful throughout this dissertation. The next section introduces fault classification by providing an overview of a commonly used fault classification scheme.

2.2 An Overview of the Software Development Lifecycle

Modern software processes are iterative and incremental in nature. The complexity of software requires the decomposition of software into smaller parts and their assembly into working systems. The history of iterative, incremental development dates back as far as the 1960s [29]. Iterative, incremental software development is an improvement on the waterfall development process. Royce introduced what we now refer to as the waterfall development process in 1970 [30]. Figure 1 illustrates an adapted version of the development process from Royce's paper. The waterfall process model provides a useful foundation for the phases and activities involved in software development. For interested readers, Larman and Basili provide an overview of the history of iterative and incremental development processes [29].

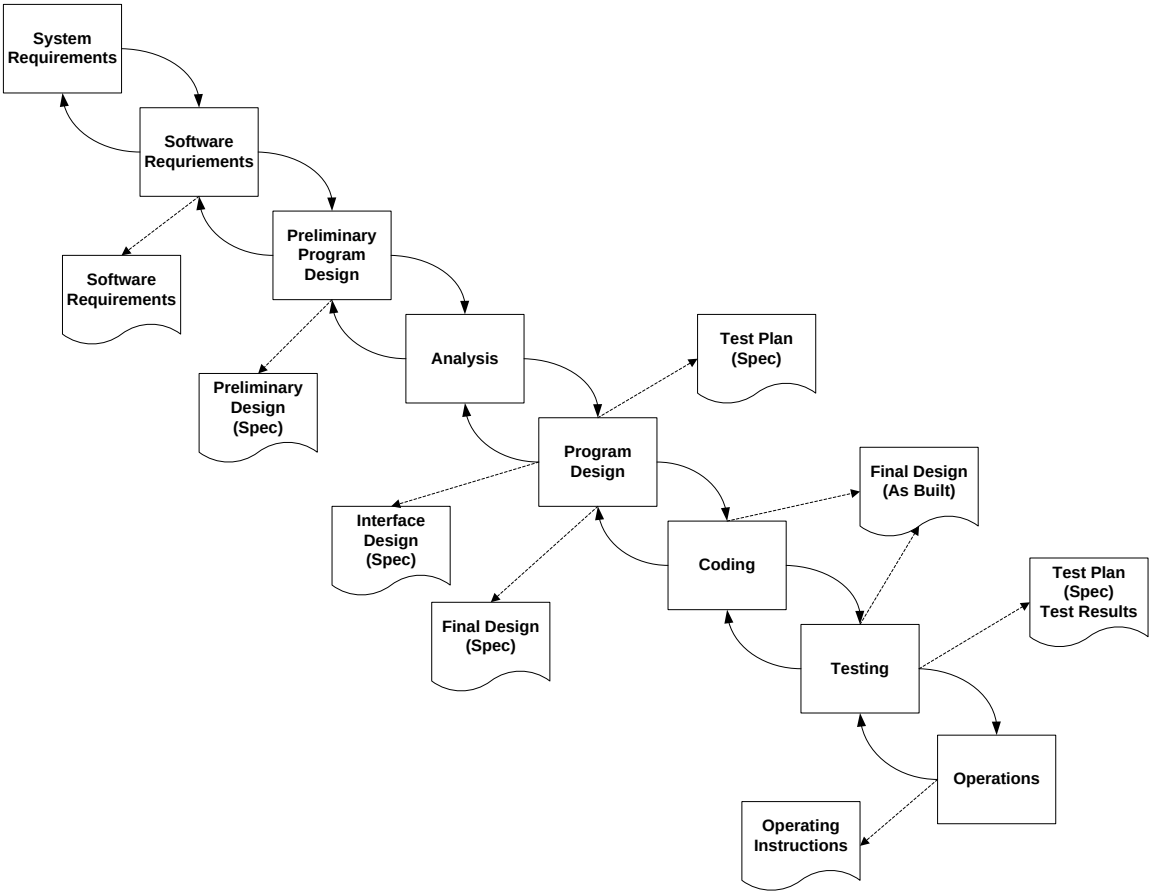

Figure 1 - Waterfall Software Development Process

The **system requirements phase** identifies the requirements for the system in the context where it will exist. The **software requirements phase** is concerned with collecting all of the requirements of the system. These requirements include functional requirements, as well as non-functional requirements such as performance, reliability, and usability. The software requirements phase results in a software requirements document as an artifact of this phase.

Royce introduces the **preliminary program design phase** to reduce risk in large development projects [30]. An important tenet of the waterfall model is that problems in a development phase should affect at most one previous phase. Without the preliminary design phase, problems with timing, storage, and other constraints identified during testing can affect the requirements phase. The addition of the preliminary design phase reduces the risk of this problem. The preliminary design phase is also known as the **high-level design phase** or the

architecture phase of a project. The focus is on the high-level structure of the software and meeting non-functional requirements.

The **analysis phase** of a software development project involves modeling the problems that the system will solve. In the context of a space guidance and control system, this might involve numerous equations for determining the appropriate flight path of a rocket. In contrast, the analysis phase for a business system focuses on understanding the logical entities and business rules to complete a transaction.

The **program design phase**, also known as detailed design or low-level design, is the activity that produces the specification for the coding phase. The interfaces of modules, as well as the data structures and algorithms, are determined during this activity. An Interface Design document and a Final Design document capture the specification. In addition, a Test Plan document is created that will guide the verification of the software after coding.

The **coding phase**, or implementation phase, involves the development of the software. Artifacts from the **program design phase** are the basis of the development effort. The Final Design document includes any changes that occur in the coding phase. The Test Plan document guides the **testing phase**. The testing effort validates the functional and non-functional properties of the system with respect to the requirements and specification. Problems found in the testing phase may affect the design, and result in changes to the Final Design document. The output of this phase is the final test plan with test results.

Once the **testing phase** is completed, the software transitions to an environment for operational use. This transition to operations includes an Operation Instructions document.

Royce's contributions were a two-stage design process, an emphasis on documentation, and the use of an early simulation, or prototype, to reduce risk for original work [30]. It is interesting to note that these observations occur within the constraints of US government-contracting models in the 1960s and 1970s. Software processes have changed over the decades, but the waterfall model

remains a useful example of the phases and activities involved due to its simplicity.

2.2.1 Verification and Validation

In software engineering terms, **verification** is the process of evaluating an artifact to determine whether it meets the conditions to exit the current phase of the software development lifecycle (SDLC) [27]. The artifact may be a requirements document, design document, a model, or a software component. In contrast, **validation** is the evaluation of a system at the end of the development process to determine whether it satisfies certain requirements [27].

It is important to detect and eliminate faults in any artifact. Faults that remain undetected and move on to the next phase, which I refer to as **escaped faults**, result in additional costs. The additional cost will vary depending on several factors, e.g., the complexity of the project and the method of delivery. Research literature estimates the cost of an undetected fault that escapes into operations to be 5:1 for small, non-critical systems up to 100:1 for large, complex systems [2].

The waterfall process described above produces several artifacts. Each of these artifacts is subject to a review on any large software project. Review of the Software Requirements document aims to detect ambiguous requirements, conflicting requirements, and any lack of completeness. Review of the Preliminary Design document (or Software Architecture document) evaluates the design to validate it can meet non-functional requirements (e.g., performance, security, reliability). The reviews of additional design documents verify that the design will meet the business requirements. The review of the Test Plan document verifies completeness with respect to the requirements. In addition, inspection of the code itself can uncover faults that may be difficult to find during testing. Some faults, e.g., poor documentation of code and failure to follow coding guidelines, cannot be detected by testing and require code inspection.

The verification of artifacts is important to uncover faults early and make the project run efficiently. Consider the example where a design has a fault that

escapes to the release phase. A customer may detect this fault during operations, requiring a fix. This forces the software organization to make a design change to software after release. The design change becomes more complicated due to backwards compatibility issues. Changes in design may also cause requirements to be re-visited. The software undergoes design, analysis, coding, and testing again in order to release the change. It is easy to see how these costs add up, and why early detection or prevention of faults increases software productivity and quality.

2.2.2 Software Maintenance and Evolution

The maintenance of software systems differs from that of hardware systems. Software does not wear out like hardware components, but it must constantly evolve to respond to changes in its environment. Lehman classifies systems into three types, according to how they may change [31]. **S-systems** are formally defined systems based on a specification. S-systems do not change often. If the real world problem that the system solves changes, it often means that a new problem has emerged, and a new system is necessary, rather than a change to an existing problem. The basis of **P-systems** is a practical abstraction of a problem. In this case, the problem is too complex for a complete, formal specification. P-systems change more often than S-systems, since the abstraction may be incomplete, and changes to the abstraction result in changes to the system.

Lehman's third type of system is the **E-system** [31]. An E-system is embedded in the real world. As the world changes, the system must be evolved or abandoned. A useful example of an **E-system** is tax preparation software. Tax laws change every year, requiring updates to these systems. Many software systems fall into this category and are subject to constant change.

Lehman introduced eight laws of software evolution [31][32]. These laws have been studied and improved over a period of thirty years [33]. The laws of software evolution, as published by Lehman [32], are summarized below.

I. **Continuing Change.** An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.
II. **Increasing Complexity.** As a program is evolved its complexity increases unless work is done to maintain or reduce it.
III. **Self-Regulation.** The program evolution process is self-regulating with close to normal distribution of measures of product and process attributes.
IV. **Conservation of Organisational Stability (invariant work rate).** The average effective global activity rate of an evolving system is invariant over the product life time.
V. **Conservation of Familiarity.** During the active life of an evolving program, the content of successive releases is statistically invariant.
VI. **Continuing Growth.** Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.
VII. **Declining Quality.** E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.
VIII. **Feedback System.** E-type programs constitute Multi-loop, Multi-level Feedback systems and must be treated as such to be successfully modified or improved.

The first law, **Continuing Change**, reflects the definition of E-type systems. As the real world evolves, the E-type system must be updated in order to remain satisfactory and useful. The law of **Increasing Complexity** states that the successive changes to the system will increase the entropy of the system unless the complexity is constrained and effort expended to reduce the complexity. The law of **Self-Regulation** states that software systems exhibit measurable and predictable behaviors [34]. The fourth law, **Conservation of Organisational Stability**, states that the amount of useful work achievable for a system is invariant. This is in agreement with Brooks' conclusions that adding resources to a software project may reduce the effective rate of productive output [35]. This counter-intuitive phenomenon is due to increased communication and other overheads as the number of contributors grows.

The fifth law, **Conservation of Familiarity**, states that over time, the effects of subsequent releases will make little difference in the overall functionality of the software. The sixth law, **Continuing Growth**, refers to the need to add functionality continually. Unlike the first law, which results from changes in the real world, this law results due to the need to scope software

systems. Out of scope features eventually become more important to users and must be added. The seventh law, **Declining Quality**, results because the assumptions made during the design and implementation phase are based on the present state of the system and the world. As the system and the real world evolve, these assumptions are likely to change and result in faults in the system. The eighth law, **Feedback System**, describes the software development process as a feedback system. For example, the system will continually grow until it becomes more expensive to expand, as a result the organization may reduce the size of the system in order to add required new functionality. Once the system size is reduced, however, it will only be a matter of time before the system is again too large for affordable growth.

2.2.3 Conclusions

This section provides background information on the software development lifecycle, verification and validation of software, and the evolution of software systems. The development of large software systems is a complex endeavor that involves numerous technical and human factors. In the following section, we build upon this background knowledge to discuss techniques to monitor and improve the software development process.

2.3 An Introduction to Fault Classification

In this section I introduce the concept of fault classification by example. Orthogonal Defect Classification (ODC) was developed at IBM by Chillarege et al. as a method of in-process feedback to developers [1]. The process bridges the gap between causal analysis and statistical defect models. Chillarege et al. characterize causal models as qualitative and high effort. Statistical defect models are quantitative, but occur late in the development process. The ODC is currently at version 5.2 [36] and has evolved based on changes in technical needs (e.g., incorporating concepts from Object-Oriented programming) and pragmatic concerns (e.g., addition of user documentation, build, and language support categories). The ODC consists of multiple attributes, each concerned

with a specific property of a fault. These attributes are designed to be **orthogonal** in two ways. The attributes are orthogonal to each other, in that they capture different information about the fault. The attribute values are designed such that only one value applies, providing orthogonal attribute values. As an introduction, I provide an overview of commonly used attributes and applications of ODC from the literature.

The key attribute of the ODC scheme is the *defect type*. This attribute captures the semantics of the fix applied to correct the fault [1]. In addition, a qualifier indicates whether something was incorrect, missing, or extraneous. The defect type categories are based on research that identified relationships between the semantics of fault fixes and the software development process [37]. A subset of the fault types and process associations are shown in Table 1. This relationship is essential to understanding when a fault is injected into the software. The knowledge of when the fault injection occurred provides feedback on the phase of the process that must improve, but also enables other forms of diagnosis, which I will discuss in the next section.

Table 1 - ODC Defect Types and Process Associations

Defect Type	Process Association
Function	Design
Interface	Low Level Design
Checking	LLD or Code
Assignment	Code
Timing/Serialization	Low Level Design
Algorithm	Low Level Design

A second attribute of importance in the ODC is the *defect trigger* [1]. The defect trigger describes the situation in which a latent defect is *triggered* in a customer environment [10]. The trigger is identified early in the lifecycle of a fault, when the fault is discovered and recorded. The trigger is an effective means of diagnosing the verification process [38]. Examples of a defect trigger include Design Conformance, Logic/Flow, Backward Compatibility, Workload/Stress, and

Rare Situations. Triggers map to verification activities such as Design Review, Code Inspection, Unit Test, Function Test, and System Test. The defect trigger also reflects the skill and knowledge of the tester. This property of triggers can be used to determine if more experienced reviewers, or reviewers with more knowledge of the system, are required to perform the review. Chaar et al. describe the use of defect triggers to assess verification activities [38].

Using only the qualifier, defect type, defect removal activity, and defect trigger, a number of different scenarios in the software development process can be analyzed. By using the association of defect types to process phases, it is possible to determine whether the fault detection occurs in the earliest possible verification activity. When faults escape the earliest possible verification activity, that activity is a candidate for improvement. After improvement activities, measurements occur against the current baseline. With the addition of historical data, it is possible to determine whether an activity is finding a sufficient number of each type of fault while that activity is in progress. Project managers can make adjustments earlier in the process when this type of data is available. These attributes provide important data for process improvement.

In addition to these attributes, ODC includes attributes such as the impact of the fault on the customer, the age of the code that contains the fault (e.g., new, pre-existing, rewritten), and the source of the fault (e.g., outsourced, re-used, ported). It is easy to see how additional attributes can provide additional diagnosis. For example, the impact attribute can be used to determine which defect types are prone to high impact customer problems. The source of the fault might help diagnose problems with outsourced work, re-used code libraries, or portability problems.

In this section I have provided an introduction to fault classification by describing the Orthogonal Defect Classification (ODC) scheme. I discussed the primary attributes, defect type and defect trigger, as well as their role in measuring the software development process. In the following sections, I will explore the impact of fault classification more broadly. The next section describes the process for the literature survey.

2.4 Literature Survey

The primary focus of this chapter is to review the literature for practices and applications of fault classification. The goal of this literature survey is to identify the claimed benefits of fault classification, analyze evidence related to its use, and present a direction for the research and application of fault classification. For this survey I selected a purposive sample of central and pivotal articles in the field. My selection criteria appear below. The analysis is presented by concept, with chronological ordering within each concept.

For each publication, I am interested in answering a number of key questions. First, I am interested in claims of benefits from the use of fault classification and the validation of these claims. Next, I am interested in challenges that arise from the use of a fault classification scheme. Finally, I am interested in the degree to which the fault classification scheme is automatable.

To locate articles, I performed a search using the key terms *software, "fault type", "defect type", and taxonomy*. I selected these terms based on a number of searches, many of which result in false positives for the term *classification*. I reviewed the 43 results and narrowed the list to 18 results by reading the abstracts of the resulting papers. In reviewing the results, I kept papers/articles that met the following criteria:

- About software, rather than hardware or power faults
- From a Journal, Conference, or a Thesis/Dissertation
- Presents
    - a fault classification scheme, or
    - applications of a fault classification scheme, or
    - a software engineering process that is impacted by fault classification
- Includes
    - new results, or significant validation of previous results

From these 18 results, I expand the list by reviewing the bibliography of the work and exploring sources that meet my criteria. In total there were 81 articles, papers, reports and book chapters that were reviewed for information collection. After eliminating redundant sources and sources that did not provide results that were relevant for my purposes, I used 54 sources.

In the following sections, I present the information that was collected and analyzed for this literature survey. I first focus on the benefits of fault classification as they have been recorded in the literature. Next, I discuss the challenges that have been published. With these benefits and challenges explained, I move on to recent innovations and thoughts on the future of fault classification research. Finally, I compare recent innovations to the research in this dissertation.

2.5 The Benefits of Software Fault Classification

This section discusses the benefits of software fault classification as recorded in the literature. Readers that are interested in adopting a fault classification scheme may find the guidelines presented by Freimut to be useful [39]. This chapter discusses the benefits of software fault classification in the broad areas of process improvement, verification and validation, and empirical knowledge.

2.5.1 Process Improvement

Knuth provides a description of the change classifications that he used for enhancements and bugs for ten years while developing the TEX system [40]. Knuth reports that his classification may appear ad hoc, but represents the best way for him to make sense of his experience on the project. Knuth presents nine classifications for bugs, which he denotes by a single capital letter (code), a name, and a short description. The author provides numerous examples to clarify each category. Table 2 below presents the classifications.

Knuth does not claim that his classification scheme is useful to anyone but himself, so it is not surprising that ambiguities are possible. For example, if a surprising scenario causes an incorrect result in an algorithm, it is not clear which classification applies. I argue that the most important contribution of this classification scheme is increased awareness about the use of fault classification for process improvement, in this case, applied to an individual.

Table 2 – Knuth's Fault Classifications

Code	Name	Description
A	algorithm awry	incorrect algorithm
B	blunder or botch	author knew what he ought to do, but wrote something else
D	data structure debacle	information not properly handled, such as memory leaks
F	forgotten function	error of omission, forgot to include a piece of functionality
L	language liability	misuse or misunderstanding of the programming language
M	mismatch between modules	forgot conventions built into a subroutine when it was used
R	reinforcement of robustness	add validation to prevent crashes and erroneous conditions
S	surprising scenario	unforeseen interactions force a change in design
T	trivial type	typed the wrong thing (e.g., '+' instead of '-'), excludes syntax errors caught by the compiler

Bridge and Miller introduced the ODC scheme to Motorola with the aim to better measure and improve the software development process [3]. Bridge and Miller describe how existing inspection data maps to ODC defect types in order to leverage historical data that is already in place. Many companies are interested in making use of existing historical data in order to take advantage of fault classification methods. Bridge and Miller describe one way to leverage existing data and describe how Motorola uses fault classification for process improvement.

Perry and Evangelist conduct an empirical investigation of software interface faults in a real-time system. The system is 350,000 non-commented lines of C source code. They construct a taxonomy by randomly selecting 84 faults, inspecting the faults, and determining if they matched an existing category, or warrant a new category [41]. In all, they define sixteen categories. They determine that 68.6% of the faults are interface faults [42]. Inadequate error processing, inadequate post processing, coordination of changes, and inadequate functionality are the most significant categories of errors in their

study. They also find that nearly three-fourths of the interface faults originate in the implementation phase, and not during the design phase.

Leszak et al. also use the taxonomy developed by Perry and Evangelist to investigate the impact of defect analysis [14]. They report five major findings. First, the cost of fixing faults grows linearly with phase when the retesting efforts are not considered. This implies that retesting costs represent a large part of the costs for faults found late in the process. They also find that the majority of faults do not originate in early phases and the distributions per subsystem reveal large differences. The authors claim that human factors significantly influence the injection of software faults, and that root cause analysis has a low and tolerable effort (reporting 19 minutes per fault) [14].

The group of studies by Perry and Evangelist [41], [42] and Leszak [14] contribute a number of interesting findings that impact current knowledge on software faults. The studies are limited to real-time systems, so further evidence is needed to generalize beyond that domain. The studies found a large percentage of interface faults, and many were introduced during implementation. Many quality improvement initiatives begin with the improvement of requirements and design. Initiatives targeting requirements and design improvements would not reduce the number of faults that occur during coding, so they would not have a large impact on the quality of these systems. These studies primarily contribute research knowledge to the software engineering community and validate it empirically in an industrial setting. The latter also contribute to the understanding of process improvement with fault taxonomies.

Yu investigates the distribution of faults in a telecommunication switching system. Yu finds that nearly half of the faults are coding faults, and that a majority of these coding faults are preventable [4]. Root cause analysis is applied, resulting in the creation and adoption of a set of guidelines to prevent the introduction of coding faults. The classification of coding defects in the case study is coarse, with three major categories. These are logic faults, interface faults, and maintainability faults. The results of adopting these guidelines are measured with metrics for average fix cost per fault, average implementation

cost, and average testing cost per source code line. The study shows a 34.5% reduction in coding faults, saving an estimated US$7M in product rework and testing. These results suggest that efforts to reduce coding faults by examining fault types and performing a root cause analysis can result in significant savings.

Lutz and Mikulski studied the high impact anomalies of seven operational, safety-critical systems using ODC [43]. Many unexpected classification patterns revealed implied software requirements, prompted changes to documentation and procedures, and helped the authors measure assumptions made about the system and its operational environment. The authors recommend the analysis of the most severe anomalies in safety critical software for better maintenance as well as improving future systems.

Robinson et al. report on the successful application of the top two levels of Beizer's classification scheme, described more fully in the Test Design section, to implement a defect-driven improvement process in industry [44]. The report indicates that approximately four-thousand defects were classified across four organizations. The effort required to perform the retrospective classification is estimated at one person-year. The results indicate quantitative and qualitative improvements in the process. The results include a reduction in the number of file changes after formal test and an improved perception of software quality by groups that test and certify the software.

Børretzen and Dyre-Hansen investigate the fault profiles of five business-critical industrial applications to determine where process improvement activities should be considered [45]. They find that the most common fault types are *function* and *GUI* fault types. *Assignment* fault types are also frequent. In terms of severity, the *relationship* fault type (associations among procedures, data structures, or objects) has the highest share of critical faults, faults with the highest severity rating. *GUI* and *Data* faults are among the least severe. Based on the results, the authors propose increased effort in the design phase to counter function faults and relationship faults.

Shenvi reports on the adoption of ODC at Philips for fault prevention [5]. Shenvi's case study is an industrial project to develop software for a DVD player.

The case study focuses on the reduction of function faults, which the authors note are particularly costly. Various practices are adopted, including a requirements workshop, design overview, automated tools for traceability improvement, and tailored checklists. The result was a decline in function defects from 28% to 12% [5].

Seaman et al. describe their experience mapping defect data from multiple, heterogeneous data sets into a single, comprehensive data set [18]. The motivation for aggregating data from multiple projects is to optimize the planning of early lifecycle verification and validation activities and demonstrate tradeoffs. The effort included data from 2,529 inspections from 81 projects across five NASA centers.

Seaman et al. present challenges in combining the data and recommendations for designers of fault categorization schemes [18]. The recommendations align with those of Freimut [39]. The classification scheme is based heavily on the ODC scheme. It is interesting to note the differences that evolved from its use in practice and subsequent aggregation with similar classification schemes. In particular, logic faults are separated from the algorithm/method type. The interface type is renamed internal interface, and a separate fault type is added for the user interface. Performance corrections in an algorithm are classified as an algorithm defect in the ODC scheme, but Seaman et al. provide a separate category for non-functional defects.

Process improvement is a critical area for software companies. Higher quality software is demanded by customers, while software companies continue to feel schedule pressures and operate with constrained resources. In this section the literature on the use of fault classification schemes for process improvement were reported.

From this literature, one can conclude that the scope of process improvement is broad. On one end of the spectrum, Knuth’s classification scheme [40] was devised for his own use so that he could make personal improvements. In contrast, Seaman et al. aggregate data from 81 NASA projects in order to improve processes across multiple projects.

The literature also addresses process improvement at multiple phases of the software development lifecycle. For example, Shenvi discusses reducing function faults and concentrates on requirements processes [5]. Børretzen and Dyre-Hansen recommend increased attention to the design phase [45]. Yu focuses primarily on faults that are introduced while code is implemented [4], and Robinson et al. focus on cost and efficiency during testing [44].

In conclusion, process improvement and fault prevention have broad implications for companies across all phases of the software development lifecycle. Fault classification provides valuable information for measuring the development process, and is thus an integral part of process improvement activities.

2.5.2 Verification and Validation

Software verification and validation (V&V) activities are concerned with the detection of software faults. Fault classification plays an important part in the design, planning, evaluation, and measurement of V&V techniques.

Test Design

One important use for a fault taxonomy is to aid testers in test design [6]. In this context, it pays to have a large number of fault categories that generate ideas about problems. These problems are the basis for test cases. Vijayaraghavan and Kaner provide an example of how tester uses a taxonomy for this purpose and how it improves completeness of the testing scenarios [6].

Beizer introduced a fault taxonomy to aid software testing [46]. Beizer's taxonomy is hierarchical with nine top-level categories. Vinter provides an update to Beizer's taxonomy [47]. The classification uses four digit numbers to indicate the placement of the fault in the hierarchy. The classification captures multiple aspects of a software fault and is thus not orthogonal. For example, a domain boundary closure is classified as "243X: Boundary closures," while other control logic errors are classified as "3128: Other control flow predicate bugs." Beizer

advised the use of a taxonomy as a statistical basis of a testing strategy, as well as a tool for test design [46].

A fault taxonomy aids test design in two ways. A taxonomy provides a set of possible fault conditions for a tester to consider when they are designing tests. In addition, baseline information about the expected number of faults in each category of the taxonomy provides a way to plan the amount of testing effort for each fault type.

Fault Injection and Mutation Testing

Fault Injection provides a way to evaluate the fault tolerance of a software system. The process of injecting faults into software to assess the fault tolerance of the system is a recommended practice in industries such as the automotive and aerospace industries [48]. Fault injection experiments require knowledge of the distribution of different fault classes to reflect typical behavior during operation. The injection of faults allows the evaluation of fault tolerance for different design choices.

Mutation analysis involves the injection of faults into software, but with a different goal. Mutation analysis provides a way to measure the quality of test cases that have been developed for a program [49]. A mutation system injects a program with faults to create multiple versions of the system using mutation operators. These faults represent small syntactic changes to the program such as replacement of one arithmetic operator with a different arithmetic operator (called the AOR mutant). The mutation system executes test cases against the source program, and then mutant programs. Since these mutant programs may have errors, the test cases may detect them – marking the mutant as dead. Once a mutant is marked as dead, that mutant program is removed from the set and tests are no longer executed against it. A mutation score is used to determine how effective test cases performed against the mutants. The score is the ratio of dead mutants to remaining mutants. Testers can add new test cases to improve the score, and thus improve the test suite.

One of the key problems in mutation testing and fault injection is the need to inject faults that are representative of software faults that are observable in the field [50]. Chistmansson and Chillarege report on a technique for fault injection using field data classified using ODC [50]. The defect trigger helps determine an operational profile, and the defect type is used to select appropriate types of defects. As a result testers can be assured that the faults generated by mutants are representative of faults that have occurred in the past, and that the investment in mutation analysis provides real benefits. These benefits include a measurably comprehensive test suite, as well as risk mitigation for the company.

Fault injection provides a method to test the reliability of a system when a fault occurs and mutation analysis provides a way to evaluate and improve software test suites. Fault classification data provides information about the types of faults that should be injected into a system. Without this information, these methods are less effective and may provide misleading results. The techniques require a representative sample of software faults in order to provide valid results.

Inspection

Kelly and Shepard extend ODC to compare the effectiveness of software inspection techniques for computational code [16]. The extended fault classification scheme, ODC-CC, is used to evaluate inspection techniques. Kelly and Shepard associate each fault type with the “level of understanding” that is necessary to identify the fault. For example, discovering a fault by comparing code to naming conventions requires less understanding than discovering a fault for logic or error handling. These faults are more difficult to identify during inspection. The study finds that the use of the task-directed inspection technique finds more of the difficult faults than the control inspection technique.

Hayes et al. define a fault link as a relationship between the type of fault and the types of components in which they occur [26]. To validate the utility of fault links they use fault link information to customize code review checklists. Hayes et al. find that the customized checklists can improve the number of faults

that found by 170-200% and the number of hard to find faults by 200-300%. This approach demonstrates the use of fault classification data, along with properties of the software, to improve code inspections.

Two studies that focus on fault classification data and inspections were identified. Many other studies address inspection as one possible V&V technique. Kelly and Shepard use fault classification data as a means to validate improved inspection techniques [16]. Hayes et al. use inspections to validate fault links, providing a practical method to improve inspections, as well as a novel way to consider the use of fault classification data [26].

Planning V&V Activities

One important use of fault classification is the planning of V&V activities. The relationship between testing techniques and the types of faults they detect is non-trivial. When data about detection techniques and the fault types they can detect are present, it allows the development of strategies for multiple purposes. One strategy may broadly cover many fault types with fewer techniques, while another strategy focuses on high risk fault types.

A report for the U.S. Nuclear Regulatory Commission and the Electric Power Research Institute contains detailed taxonomies for faults, and for detection methods [7]. The report provides guidelines for the verification and validation of both conventional software and expert systems. In the report, Miller et al. conducted a literature survey to identify methods for the verification and validation of software [7]. The report classifies methods according to the most appropriate phase in the software development lifecycle. The report also characterizes methods according to their ease-of-use and fault detection capabilities. Two measures are developed to allow quantitative comparisons, a Cost-Benefit Metric and an Effectiveness Metric [7]. The metrics allow the ranking of methods according to the goals of a software development organization or project.

Vegas et al. present a characterization process for testing technique selection [8]. The characterization schema includes the defect (fault) type.

Historical information about which testing techniques discovered which types of faults can be used to aid technique selection in future efforts.. Components often exhibit similar types of faults as they have in the past, so the history supplies helpful empirical data about the selection of the most effective testing technique.

Inspection is an important practice in verification and validation of software. It is not always clear, however, when it should be applied, and to what extent. Runeson et al. analyze several empirical studies to answer this question and provide some practical findings [9]. They find that inspections are more efficient and effective at finding design specification defects. Functional and structural testing more effectively find code defects. Runeson et al. suggest design specification inspections to find design faults early, and a balance of code inspection and testing techniques to find faults in code.

Zheng et al. evaluate the ability of static analysis to detect faults in three large industrial software systems at Nortel Networks [51]. Zheng et al. find that static analysis is an affordable means of fault detection, and that it is most effective at detecting Assignment and Checking faults. Furthermore, statistical analysis indicates that the number of static analysis faults can be effective for identifying problematic modules in a software system. The use of static analysis may allow organizations to focus on the detection of more complex faults. One of the findings in this dissertation is that complex faults are more likely to be *problematic faults,* which require multiple rounds of changes for repair. Static analysis is easily applied to new projects, while existing projects may require more significant effort for adoption. This is because static analysis checks for current best practices in software development, and older programs are likely to have multiple violations due to advances in software development practices.

Li et al. develop an extension of ODC for black-box testing called ODC-BD [52]. ODC-BD is validated against faults from 39 industry projects and two open source projects. Li et al. also validate the use of the taxonomy to reduce effort during defect analysis and improve testing efficiency [52].

Planning the verification and validation of software effectively and efficiently is an important, practical concern as well as an open area of research.

In this section, I have discussed several studies with different approaches to planning these activities. Broad approaches, such as that described by Miller et al. [7] and Vegas et al. [8] require knowledge of fault classes that are targeted by a technique.

Other studies focus on particular methods. Runeson et al. seek to choose between inspection and testing techniques [9]. Zheng et al. focus on understanding the types of faults detected by static analysis [51]. Li et al. provide a different approach by focusing on black-box testing, but extending the ODC classification scheme in order to customize it to the needs of black-box testing. These studies provide valuable empirical knowledge about individual techniques and the types of faults detected by their use.

Evaluating V&V Effectiveness

Fault classification can be used for process improvement that targets verification and validation (V&V) activities, such as review, inspection, and testing. Studies in this section seek to determine how faults that are discovered by customers escaped V&V activities, or to understand high severity failures. This information is essential to formulating V&V strategies and meeting quality targets in software projects.

Sullivan and Chillarege studied faults that cause high severity failures in a high-end operating system [10]. Their research focused on *overlay* failures, which result in corrupted program memory. The study confirms their impact by measuring the probability of such a fault to achieve a severity 1 rating, and its probability of being flagged as “highly pervasive” by customers [10]. They find that most of these faults are due to boundary condition and allocation problems. This is contrary to the common belief that timing or serialization problems are the primary cause of these high severity failures. Based on these findings, the number of these faults could be greatly reduced by applying better testing of boundary conditions, which is much less effort than timing/serialization tests.

Chillarege and Bassin describe their use of ODC to systematically determine how faults escaped V&V activities into the field [11]. The trigger

provides valuable information on how the failure can be reproduced. The authors note that each trigger has a different distribution based on time. Tactically, this information can be used to focus testing on issues that will be found immediately following the release, while testing for faults that are found after longer time periods could be delayed and fixed in subsequent patches. For example, the authors find that documentation and backward compatibility failures are generally uncovered quickly, while lateral compatibility failures peak almost a year later. This information is valuable in order to prioritize testing efforts for software products.

The trigger attribute is often used in combination with other attributes to assess the state of verification and validation (V&V) activities. Chaar et al. present expected distributions for triggers and fault types and demonstrate their use to troubleshoot V&V activities [38]. Chillarege and Prasad expand on this concept by focusing on the trigger and activity [53]. By comparing current values to benchmarks, Chillarege and Prasad are able to determine that code quality is poorer than expected and that inspections should have caught more of the faults. These observations led to recommendations to correct the situation, but also led to guidance for avoiding the situation in the next release.

Similar to the need to plan an effective V&V strategy, it is necessary to evaluate its effectiveness. Fault classification data provides feedback that allows corrective action. The development of software is simply too complex and is impacted by too many factors for consistent success through experience alone. In this section I discussed multiple ways that researchers have applied different attributes of ODC in order to investigate software faults. These studies investigated high severity faults [10], determined how faults escaped verification and validation activities, and evaluated and controlled the verification and validation process.

Software Security

Studies show that security vulnerabilities have major economic impact on software vendors, including a direct impact on stock price [54]. Technology

trends such as cloud-computing, mobile devices, and the widespread use of software in critical applications make software security a growing concern. Research into prevention and detection of these problems is relevant, and necessary for improvement. The use of fault classification designed for this purpose can aid in software security improvement practices.

Du and Mathur present a classification scheme that is designed to determine the effectiveness of software testing techniques in revealing security errors [55]. The scheme consists of attributes for the cause, impact, and fix for the fault. The scheme was validated by inspecting security vulnerability reports from public security vulnerability databases.

More recently, Hunny et al. extended the Orthogonal Defect Classification scheme to create a security specific scheme that they refer to as the Orthogonal Security Defect Classification (OSDC) scheme [56]. The authors validate their scheme against security vulnerabilities recorded against several versions of the Firefox and Chrome browsers. They found that some fault classes were more commonly associated with security vulnerabilities that occurred in multiple releases. For example, the *exploitable logic error* fault class was consistently a large percentage of security vulnerabilities across versions. They recommend more attention during high-level design and implementation, as well as additional effort during code review, unit test, and function test to mitigate this concern. Their goal is to apply OSDC during development and allow teams to benefit from in-process feedback to aid in adoption of a secure development lifecycle [56].

2.5.3 Empirical Knowledge

While many of the studies previously mentioned contribute to empirical knowledge, they are focused on specific activities and applications. In this section I focus on studies that were developed specifically to address empirical questions about the nature of software faults.

Dyre-Hansen investigates 901 faults from online bank and financial systems [15]. Dyre-Hansen finds that the majority of faults in these systems are

function faults (27%) and GUI faults (19.5%). Relationship faults and Timing/Serialization faults tend to be the most severe faults, while GUI and Data faults tend to be less severe. Dyre-Hansen found little correlation among the different projects and the distribution of the fault types using ODC [15].

Hamill and Goševa-Popstojanova conducted an empirical investigation and characterization of software faults and failures based on data extracted from change tracking systems for large-scale, real world projects [28]. The study finds that requirements and coding faults contribute to about 33% of the total faults each, and that the next highest category is “data problems” at 16%, where “data problems” include structural and interaction problems with the data repository. To further investigate this distribution, the authors group projects based on the number of releases and compare their results with other studies. From these comparisons they conclude that the percentage of coding faults is significant, being roughly equal to the number of requirements and design faults combined. They also conclude that interactions between components cause problems, and that other defect types are less significant and may be influenced by the domain of the software. The percentage of coding faults, requirements faults, and data faults were found to be surprisingly consistent across projects with different domains, programming languages, processes, and people. These findings lend empirical evidence that coding faults are a common problem in software development projects.

These studies provide useful data that may be used to improve the state of the art in software engineering. For example, Hamill and Goševa-Popstojanova reveal that 33% of faults are introduced during implementation. Many projects begin improvement efforts with the requirements phase, but this evidence provides reason to carefully consider a more balanced approach. It is also interesting that the distribution of fault types across projects that was observed by Dyre-Hansen [15] exhibits no pattern, while Hamill and Goševa-Popstojanova are able to find consistent patterns [28] when using a higher level classification (e.g., Requirements, Design, Data, Coding). Understanding the nature of software faults in large systems is an important research area, with

practical implications for industry and research. Relatively few studies exist that consider this problem in conjunction with the type of faults that occur.

2.6 Manual Fault Classification Challenges

Above we described many advantages of software fault classification. Advantages include applications in process improvement, verification and validation, and in empirical software engineering research. Despite the multiple advantages there are many challenges to the adoption and use of fault classification practices. In this section I review literature that illuminates these challenges.

"The range of efforts to create defect classification schemes [..], and the long history, in which there has been no single, widely used scheme, suggests that defect classification is hard, and repeatable orthogonal classification is itself difficult."
- Kelly and Shepard [16]

The quote above summarizes my beliefs on the challenges of software fault classification, still accurate fourteen years after it was published. To explore these challenges I look at: 1) research that directly studies challenges in software fault classification, and 2) evidence from work that I have already discussed, where the focus of the study is a benefit, rather than a challenge.

2.6.1 Empirical Studies of the Challenges of Fault Classification

The studies in this section are focused on challenges in fault classification. These studies focus on the repeatability of fault classification, its effectiveness, and the orthogonality of the classification. Other considerations include efficiency and experience requirements.

El Emam and Wieczorek conduct a study to determine whether fault classification using ODC is repeatable [57]. The authors use the Kappa

coefficient to measure agreement between classifiers and found that in general there is good agreement ($\kappa > 0.62$) and in some cases excellent agreement ($\kappa > 0.82$). The authors point out confusion between the *Data* and *Assignment* defect types by combining the types and showing the impact on measurements. While these results seem promising, the results cannot be generalized. Their results were for a single organization, and only studied the inspection activity.

Henningsson and Wohlin conducted a study to determine whether a group separate from the developers can correctly classify software faults based on the fault descriptions [12]. The authors find that agreement is low, but that the participants are confident in their decisions. This illustrates the impact of human fallibility on fault classification. The authors also conclude that training is required, but that education alone does not explain the low agreement.

Falessi and Cantone explored the effectiveness, efficiency, orthogonality, and discrepancy of software fault classification using ODC [13]. They find that all effectiveness, orthogonality, and discrepancy are dependent upon experience. They found that the mean time to classify a defect was 5 minutes and the median 6.7 minutes. The authors provide information about affinity between some defect types in the ODC scheme and recommend improvements in documentation and definition of these types in order to improve the repeatability of fault classification. The affinity of a fault type A with respect to a fault type B measures the percentage of faults of type A that are classified as A or B. Falessi and Cantone find that when the most frequent classification (MFC) is *Relationship*, 90% of the categorizations from participants are *Relationship* or *Interface/OO Messages*. They also find that when the MFC is *Checking*, 95% of the classifications are *Checking* or *Algorithm/Method*. Finally, Falessi and Cantone found that faults with an MFC of *Assignment/Initialization*, *Algorithm/Method*, or *Checking* are classified as one of these classifications 90% of the time. In other words, these three classifications are often interchanged by participants.

Several interesting conclusions can be drawn from these studies. First, orthogonality is indeed difficult to achieve. Without orthogonal attributes and attribute values it is difficult to get agreement on the correct classification of a

fault, and thus difficult to get actionable data. The studies by El Emam and Wieczorek [57] and Falessi and Cantone [13] both identify affinity between fault types. The study by Henningsson and Wohlin [12] indicates that the description of the fault alone is insufficient to classify faults reliably. Perhaps more concerning, is the high confidence of participants in their decisions, even when they are incorrect [12]. Thus, the impact of the human classifier cannot be understated. An additional perspective on this dependency is that of the experience of the classifier. Falessi and Cantone find that many aspects of the fault classification activity are impacted by experience [13]. Orthogonality, available information, and experience are thus three major challenges that have been explored in empirical studies. Studies seem to indicate that the time to perform classification is modest, including Falessi and Cantone which explicitly measure this aspect of fault classification [13].

2.6.2 Fault Classification Challenges from Research and Practice

In this section I explore the fault classification challenges that have been reported from industrial and research literature that was focused on the benefits of the technique. I have arranged these observations into high level topics. The first is the problem of consistent data. The second is time commitment. A third area of concern is the customization of fault classification schemes.

Data Consistency

Consistent data is necessary in order to make good decisions based on that data. A number of studies cited problems with the consistency of data that was collected. Leszak et al. reported that 30% of the data collected from engineers was inconsistent [58]. They conclude that additional training may be necessary. However, training seems to be only one aspect of inconsistency.

Dyre-Hansen found that 21.5% of problem reports were either not faults, or duplicates, while 12% were classified as unknown [15]. The large percentage of unknown fault reports represents a significant problem in data consistency. The percentage is large enough to negatively influence decisions based on the

distribution of the faults. For example, if a large percentage of the unknown fault reports represent design issues, but the correctly identified faults indicate that implementation faults are the largest category, the corrective actions will be applied to the incorrect phase of the software development lifecycle.

Shenvi points out that some faults could belong to one or more type according to the ODC scheme [5]. It is unclear whether this is a problem with the scheme, a problem within that particular domain, or perhaps due to the interpretation of the information. Kelly and Shepard noted differences in interpretation as well as a reliance on skill and experience [16], so it is likely that multiple factors play a part.

Seaman et al. point out that quality assurance activities are necessary to mitigate factors such as these [18]. Quality assurance activities on fault data uncover problems that suggest additional training, but may also uncover needs for changes to the classification scheme. Changes to the scheme may include new fault categories and changes to existing categories that are often misclassified.

Time

Although studies have shown that the time to classify faults is small [13], [58], [59], additional evidence suggests that other time commitments may cause resource problems. Despite an estimate of nineteen minutes to perform root cause analysis on each fault, Leszak et al. reported that the complexity of the scheme caused stakeholders to lose track of the classification effort due to project pressures [58].

While analysis is a larger time commitment than classification, studies revealed other time constraints that impact cost. For example, Bhandari et al. estimate fault classification at 4 minutes per defect [59]. However, they do not account for training and they estimate 10-20% of an individual's time for data collection and analysis. It is also possible that the individuals needed for data collection and analysis are highly skilled individuals with multiple competing priorities.

I conclude that the time commitment of adopting a fault classification scheme and the associated practices are not well understood. In order to truly measure the cost, it is necessary to take multiple factors into account. First, there is the time to classify a fault. While this time commitment is modest, it is also frequent. An average of four minutes per fault for one thousand faults is the equivalent of 67 man hours of effort. While I believe that this investment is reasonable, it is likely one of the smallest resource requirements required to adopt fault classification.

In addition to the time for classification, there is the time necessary for training staff. Education is clearly necessary to end up with consistent data, although it is not itself sufficient for ensuring consistency. The scheme must be clearly documented, with relevant examples, and strict guidelines [39]. The training activities, along with the time commitments to develop guidelines and examples for operation and for the training itself, are likely to be a significant investment of time in most organizations.

Finally, one must also consider the time investment of quality assurance for fault classification data. This includes reviewing faults, recording findings, and providing feedback on corrective measures. Corrective measures include training and changes to the fault classification scheme.

Customization of Fault Taxonomies

A number of factors may require customization of fault taxonomies. Some factors are obvious, such as the goals of the organization. Others are less understood. Ploski et al. investigate fault classification schemes in order to better understand how fault injection studies should select a fault density and frequency of fault classes [60]. They conclude that the distribution of software faults is dependent on project-specific factors such as the maturity of the software, the operating environment and the programming language. Furthermore, they state that it is not obvious how these factors should be considered, or systematically discovered. This section contains examples illustrating the needs for customization, as well as some recommended approaches.

Studies by Shenvi [5] and Freimut et al. [17] specifically cite a need for domain specific customization of fault classification schemes. Freimut et al. present such a customization approach that was used and validated at Robert Bosch GmbH in the Gasoline Systems business unit [17]. Seaman et al. discuss the challenges associated with customization in NASA, when the data is aggregated [18]. The broad customization of the schemes within the same organization suggest that the domain is only one factor that contributes to customized schemes.

Hayes presents a process for tailoring and extending a requirements fault taxonomy for specific projects and types of projects within NASA [19]. The process of tailoring the fault taxonomy enables a project to better meet its objectives with regard to quality and safety.

In this section I have presented a number of factors that require customization of fault classification schemes. While the factors are varied, and relatively poorly understood, the result is that customization of fault classification schemes are needed and impact the success of their adoption in organizations.

2.7 Automated Fault and Failure Classification

Researchers have looked at automated methods of understanding fault and failure information for various purposes. This includes detection of duplicate problem reports, determining the best developer to fix a fault, and automated classification. In this section I discuss these efforts and relate it to my research.

2.7.1 Duplicate Reports

Podgurski et al. created an automatic way to classify software failures for software that is instrumented to detect failures [20]. The authors believe that the instrumentation of software to provide execution profiles when failures occur will increase the number of problem reports, and increase the number of failure reports for the same underlying fault. The authors describe a process to select a subset of features, perform automated cluster analysis, and compliment it with visualization of the data. Podgurski et al. find that small, tight clusters were quite

likely to contain failures with the same cause [20]. A few large, non-homogenous clusters existed with sub-clusters that contain similar causes. In some cases failures from the same cause were split. The technique reduces the average amount of time and effort necessary to diagnose a failure.

Runeson et al. apply Natural Language Processing techniques to the text of fault reports in order to identify duplicates [21]. The technique is validated at Sony Ericsson where approximately 40% of the marked duplicates were identified. Runeson et al. interviewed developers and testers and were able to confirm that detection of 40% of duplicates represented a significant cost savings [21].

2.7.2 Fault vs. Enhancement

Antoniol et al. classify problem reports from Mozilla, Eclipse, and JBoss to determine if the report describes a fault or another activity (e.g., enhancement or refactoring) [22]. Issue descriptions were used to distinguish faults with a precision between 64% and 98% and a recall between 33% and 97%. This work is complimentary to the research presented in this dissertation. The technique presented by Antoniol et al. provides an effective pre-processing step to eliminate non-corrective maintenance activities from consideration.

2.7.3 Classification of Fault Impact

Huang et al. present AutoODC, an approach to automating ODC classification by treating it as a supervised text classification problem [23]. AutoODC requires experts to annotate the text of the problem report. Once annotated the system classifies the Impact attribute of ODC. Although Huang et al. claim that this technique can be applied to other attributes of fault classification, no evidence of this has been presented. The work in this dissertation focuses on the fault type, or defect type in ODC, which characterizes the nature of the fault fix. Therefore, in its current state, the work of Huang et al. complements the research in this dissertation by automating a different attribute of the fault.

2.7.4 Automatic classification of fault severity

Menzies and Marcus developed a system called SEVERIS which uses the text of problem reports to automatically classify the severity of the faults [24]. SEVERIS performs its classification and compares it to the manually assigned severity. Discrepancies can be reviewed and corrected by supervisors. SEVERIS was validated on NASA robotics projects. The reported *F*-measure varied for projects and severity levels. Three of the measurements were greater than 0.90 and many instances were greater than 0.7.

Lamkanfi et al. performed a similar study to predict the severity of problem reports on three open source systems [25]. Lamkanfi et al. predicted the severity of faults from Mozilla, Eclipse, and Gnome. They concluded that a training set of approximately 500 reports per severity was needed to gain consistent results.

The severity of an issue is important to determine the priority with which it is addressed. Severity levels are often subjective, so automated support can help compensate for human error or inexperience. These studies complement the research in this dissertation by automating the severity attribute of a fault.

2.7.5 Automated Classification of Fault Family

Thung et al. propose an automated categorization of software faults into three families: *control and data flow*, *structural*, and *non-functional* [61]. Thung et al. use features from bug reports and from the source code that fixes the software fault. A multi-class classification algorithm is used to classify the faults. The approach was evaluated on 500 manually labeled faults from three open source systems. An F-measure of 0.692 and an accuracy of 0.778 was achieved [61].

Tan et al. use the text of the problem report to classify 109,014 faults into semantic bugs and memory bugs [62]. The purpose of the automated classification is to reduce manual effort in building bug benchmarks for the evaluation of fault detection tools.

This dissertation differs from the approaches of Thung et al. [61] and Tan et al. [62] by providing more granular fault types that are not pre-determined. In this dissertation we utilize the syntax of the fault fix to group faults, and are not limited by the completeness or correctness of the fault description. Thung et al. use statistics on program elements in addition to the text [61]. However, they only consider a handful of program elements in their classification scheme, and only classify faults into three fault types. The research in this dissertation provides flexibility in the number of fault types and is able to consider all source code changes.

2.7.6 Bug Fix Patterns

Pan et al. present twenty-seven automatically extractable bug fix patterns as a new approach to software fault classification [63]. They are motivated to find the most common types of software faults for a specific system and whether the frequency of these software faults are common across systems. Their validation finds that 45.7-63.6% of bug fix changes can be classified using their method. The changes are classified based on locations within the file that have changed, rather than classifying the fault itself. The most common patterns identified are changes of the parameters in method calls, changes to conditional expressions in an *if* statement, and changes to assignment expressions. Six of the seven projects have similar bug fix pattern frequencies. An analysis of five developers in the Eclipse project shows a surprising consistency in the rate at which developers introduce certain types of software errors.

Merkel and Nath manually apply the bug fix patterns introduced by Pan et al. as a software fault classification for a Java-based system [64]. They randomly select 100 commits (373 file revisions) from 476 commits that are identified as fixes. They suggest four possible new bug fix patterns. The suggestions are method return value changes, scope changes, loop-related changes, and changes to string literals. Their results lend additional evidence that the bug fix pattern approach is useful, and also demonstrate that the patterns are not comprehensive.

There are two major differences between the Bug Fix Pattern approach and the research in this dissertation. The first difference is what is classified. Bug Fix Patterns classify a section of code that has been altered. This means that many such patterns could be present in a single fault fix. In contrast, this dissertation categorizes the entire fault using information about all of the changes. It may be possible to use the Bug Fix Patterns as a higher level change type in order for these techniques to be integrated. The second primary difference is the identification of patterns. The Bug Fix Patterns are identified manually, and then their detection in source code is automated. The work in this dissertation takes a different approach. I classify the source code changes and find patterns through the use of clustering. This automates the pattern recognition.

2.8 Discussion

This chapter began with the introduction of fault classification. In order to provide a concrete example, an overview of the Orthogonal Defect Classification (ODC) scheme was presented. This scheme was selected for this purpose due to its large record of use in industry and research.

The benefits of fault classification are broad. I began the discussion of benefits with the most widely cited benefit of fault classification, that of process improvement. Process improvement is critical for software companies, and its applications range from reducing coding defects, improving verification and validation activities, to changing processes that impact multi-project organizations.

Verification and validation are also benefitted by fault classification in multiple ways. A fault taxonomy can serve as a guide to testers that are designing tests, guide the injection of faults for reliability testing, aid in planning quality-related activities, and aid in the measurement of their effectiveness.

Research on fault classification is far from complete. There are multiple issues that make the classification of software faults difficult. Getting consistent data requires a useful scheme that is properly customized for the environment

and domain. The scheme must be well documented, and training must be conducted. In addition, there is no substitute for the skill and experience of the classifiers.

Researchers have recognized the limits of manual fault classification and have investigated automation solutions. Studies have attempted to limit duplicate problem reports, separate corrective maintenance from other issues, and automatically determine the impact and severity of software faults.

Relatively few studies have addressed the automatic classification of faults according to their fault type. Thung et al. successfully distinguish three broad categories of faults by using information from the text of the problem report in addition to information from the source code changes [61]. Pan et al. provide an automated method to classify source code changes, but the classification occurs for every pair of changes in the source code that repair a fault [63].

I believe that the future of fault classification lies in the automation of the work. Automated approaches that can deliver on benefits that have been recorded, as well as address major challenges, can drastically impact how software organizations approach fault classification. This paradigm shift should reduce the cost of ownership that is present in fault classification practices today and make the practices more accessible to organizations that can benefit from these practices.

Chapter 3

Mining Software Fault Information and Types

This chapter describes my approach for mining and categorizing faults based on syntactical change data. I present MiSFIT (Mining Software Fault Information and Types), a process, and toolset for mining software fault information. My approach consists of three phases. Each phase builds on the results of the last. The first phase extends a change taxonomy. The resulting change taxonomy provides a method to categorize and count the syntax changes in a fault repair. The second phase provides a method to cluster software faults based on the syntax of the fault repair. The final phase applies the automatically clustered faults to the analysis of software faults over several releases of an open source software project.

3.1 Extending a Change Taxonomy

This research investigates the extension and application of fine-grained source code changes to the analysis of software faults. Fluri et al. introduced ChangeDistiller, a tool that can identify the fine-grained source code changes from two versions of source code [65].

The algorithm and change taxonomy implemented in ChangeDistiller are designed to analyze change couplings [65], [66]. The taxonomy is not adequate for the analysis of software faults due to its treatment of source code statements. From a change coupling perspective, the insertion of an *if statement* or a *method invocation* have an equally small probability of causing changes in other parts of the source code. However, from a software fault perspective, the difference in these two changes strongly informs the classification of a fault.

I extended the change taxonomy and made changes to the application in order to capture information that was relevant to software faults. My first research question, which I must address before going further, is whether this extended taxonomy has information that is relevant to software fault analysis.

RQ4.1: Can an extended change taxonomy provide additional information about source code changes that is useful in the analysis of software faults?

The details of the extended change taxonomy are discussed in Chapter 4. Chapter 4 also describes the tool, MiSFIT, which I developed to collect the software fault data.

3.2 Clustering Software Faults

As previously mentioned, clustering is a machine learning technique that groups data instances into natural groups [67]. Clustering is therefore useful when a training set is not available. In this study, I cluster software faults based on the types of syntactic changes that occurred to repair the fault.

A clustering solution is often evaluated for its internal and external quality. I expect a clustering solution for software faults to be stable from one version of software to the next. Changes in the distribution of fault types must be reasonable, and explained. In addition, I want to know that the clusters convey beneficial information to users of such a system. The goal of clustering the faults, as with fault classification, is to enable analysis of faults at a macro level. This leads to two important research questions for clustering software faults.

RQ5.1: Can clustering of fault fixes by syntactic changes result in consistent clusters for a software project?

RQ5.2: Does the automatic categorization of faults by syntactical change provide beneficial information regarding the nature of the software fault?

3.3 Software Fault Evolution

Software evolution is the study of large, long-lived systems. Due to changing business requirements and environments, combined with changes in user expectations, successful software is constantly changing. Software undergoes changes to correct faults, enhance functionality, and manage

complexity (controlling maintenance costs). Chapter 6 looks at the evolution of software faults with the benefit of classified fault data.

With the addition of fault type, I can look at interesting questions about the evolution of software systems. For example, do the same types of faults tend to occur in the same locations? Do developers tend to fix the same types of faults? Some faults require multiple attempts to repair. I refer to these faults as *problematic fault* fixes. Do these problematic fixes tend to occur more often for certain fault types? These types of question led to the following research questions.

RQ6.1: Over time, do the same types of faults tend to occur in a given subcomponent?

RQ6.2: Are certain fault classes more likely to be fixed by single or multi-file changes?

RQ6.3: Do developers tend to fix the same types of faults?

RQ6.4: Are pre-release fault distributions predictive of post-release fault distributions?

RQ6.5: Are problematic fault fixes distributed evenly among fault classes?

Chapter 4

An Extended Change Taxonomy for Software Fault Analysis [1]

This chapter presents an extension to an existing change taxonomy and its application to the analysis of software faults. In this chapter I present the existing taxonomy, including the algorithm and tool that supports the taxonomy. I then describe my method for extending the taxonomy for analyzing software faults. Finally, I present an experiment that shows that my extended taxonomy provides useful information for the software faults in my case study.

This research investigates the extension and application of fine-grained source code changes to the analysis of software faults. Fluri et al. introduced ChangeDistiller, a tool that can identify the fine-grained source code changes from two versions of source code [65]. The algorithm and change taxonomy implemented in ChangeDistiller are designed to analyze change couplings [65], [66]. A version of ChangeDistiller is available under an open source license[2]. The change taxonomy consists of more than forty change types. Four of these change types identify the insert, update, delete, or re-ordering of a statement. In order to extend the taxonomy, I expand these four change types by appending the type of statement that was changed.

4.1 A Taxonomy of Source Code Changes

Fluri and Gall present a taxonomy of source code changes for change analysis [66]. The taxonomy is based on the comparison of abstract syntax trees. The commonly used textual differencing approach is not sufficient, since textual changes may include formatting changes and updates to comments which are cosmetic. The taxonomy models changes to abstract syntax trees as operations on the nodes of the tree, specifically, insert, update, move, and delete changes.

---

[1] © 2014 IEEE. Reprinted, with permission, from Bill Kidwell, Jane Huffman Hayes, Allen P. Nikora, "Toward Extended Change Types for Analyzing Software Faults", Proceedings of the 14th International Conference on Quality Software (QSIC), Oct. 2014.

[2] https://bitbucket.org/sealuzh/tools-changedistiller/

In addition to defining the taxonomy, Fluri and Gall also associate a *significance level* to each change type. These significance levels are low, medium, high, and crucial. The value is based on the probability that the change will result in additional changes in the source code. For example, changing the name of a method requires a change to each method invocation of that method, resulting in a high significance level. The change taxonomy is presented here in two parts. The first part, presented in Table 3, represents changes to declaration parts in the source code. The second part, presented in Table 4, represents changes to the body of a class or method.

Table 3 - Fluri and Gall's Change Taxonomy - Declaration-Part

Change Type	Significance	Description
Class Renaming	High	Changing the name of a class.
Decreasing Accessibility Change	Crucial	Changing accessibility on a class, method or attribute to a less accessible state (e.g., public to private).
Attribute Type Change	Crucial	Changing the type of an attribute (e.g., from integer to float).
Attribute Renaming	High	Renaming an attribute without modifying the type of the attribute.
Final Modifier Insert	Crucial	Adding a final modifier to a class, method, or attribute. This prevents a class or method from being overridden. It prevents an attribute from being modified.
Final Modifier Delete	Low	Removing a final modifier from a class, method, or attribute. This allows derivation for classes or methods and allows modification for attributes.
Increasing Accessibility Change	Medium	Changing accessibility on a class, method or attribute to a more accessible state (e.g., private to protected).
Method Renaming	High	Changing the name of a method without changing the return type or parameters.
Parameter Delete	Crucial	Removing a parameter from a method.
Parameter Insert	Crucial	Inserting a new parameter in a method.

Table 3, continued

Change Type	Significance	Description
Parameter Ordering Change	Crucial	Changing the order of one or more parameters in a method.
Parameter Type Change	Crucial	Changing the type of a parameter in a method.
Parameter Renaming	Medium	Renaming a method without changing the type of the method.
Parent Class Delete	Crucial	Removing an inheritance or extension association with a parent class or interface.
Parent Class Insert	Crucial	Adding an inheritance or extension association with a parent class or interface.
Parent Class Update	Crucial	Changing an inheritance or extension association with a parent class or interface.
Return Type Delete	Crucial	Changing the return type of a method to void.
Return Type Insert	Crucial	Adding a return type to a method.
Return Type Update	Crucial	Changing the type of the value returned by a method.

Declaration-part changes include changes to method signatures, changes to a class name, and to an attribute's type. They also include changes to the accessibility of a class, method, or attribute. These changes are the most significant changes in terms of change propagation.

Body-part changes represent either the addition/removal of methods/attributes to a class or changes within a method. Changes within a method can be further divided based on whether they change condition expressions, impact the control structure of the method (thus changing the nested depth), or move the location of a statement to a new block.

4.2 Extending the Change Taxonomy

As previously mentioned, the existing change taxonomy is inadequate for software fault analysis due to the treatment of statements. The majority of fault fixes impact statements within a method. In order to understand the type of change that is applied, more precise information about the type of statement is necessary.

Table 4 - Fluri and Gall's Change Taxonomy - Body-Part

Change Type	Significance	Description
Additional Functionality	Low	Addition of a function.
Additional Object State	Low	Addition of an attribute.
Condition Expression Change	Medium	Change to a condition expression in an *if statement* or loop.
Decreasing Statement Delete	High	Deletion of a statement that results in a decrease in the nested depth of the method.
Decreasing Statement Parent Change	High	Change to the location of a statement that results in a decrease in the nested depth of the method.
Else-Part Insert	Medium	Addition of an else block to an *if statement*, or case block within a switch.
Else-Part Delete	Medium	Removal of an else block from an *if statement*, or case block within a switch.
Increasing Statement Insert	High	Addition of a statement that increases the nested depth of the method.
Increasing Statement Parent Change	High	Change to the location of a statement that results in an increase to the nested depth of the method.
Removed Functionality	Crucial	Removal of a function.
Removed Object State	Crucial	Removal of an attribute.
Statement Delete	Medium	Deleting a statement from a method.
Statement Insert	Medium	Adding a new statement within a method.
Statement Ordering Change	Low	Changing the order of statements within a method.
Statement Parent Change	Medium	Changing the parent of a statement (e.g., moving a statement within an *if* block).
Statement Update	Low	Updating a statement within a method.

The contextual information collected by ChangeDistiller allows the extension of the *statement delete*, *statement insert*, *statement update*, and *statement ordering change* change types. I use the *changed entity* information available from the ChangeDistiller API to identify the type of statement that was altered, such as an *if statement* or *method invocation*. For example, a change of type *statement insert* and a changed entity of *method invocation* will result in an extended change type of *statement insert method invocation.* I translate this value to *insert method call* for readability.

The extension of these change types more than doubles the number of change types. The theoretical size is equal to the number of statement level entities in the language multiplied by the four node operations. I only record change types that are actually observed. The source code entities that were observed are listed in Table 5. Along with the entity type, I indicate whether it was seen as part of a statement insert ("I"), statement delete ("D"), statement ordering change ("M"), or statement update ("U").

Table 5 - Entities Observed in Extended Change Types

Entity Type	I	D	M	U		Entity Type	I	D	M	U
ASSERT_STATEMENT	x	x		x		POSTFIX_EXPRESSION	x	x	x	x
ASSIGNMENT	x	x	x	x		PREFIX_EXPRESSION	x	x	x	x
BREAK_STATEMENT	x	x	x			RETURN_STATEMENT	x	x	x	x
CATCH_CLAUSE	x	x	x	x		SUPER_CONSTRUCTOR_INVOCATION	x	x		x
CLASS_INSTANCE_CREATION	x	x	x	x		SUPER_METHOD_INVOCATION	x	x	x	x
CONSTRUCTOR_INVOCATION	x	x		x		SWITCH_CASE	x	x	x	x
CONTINUE_STATEMENT	x	x	x			SWITCH_STATEMENT	x	x	x	x
DO_STATEMENT	x	x	x			SYNCHRONIZED_STATEMENT	x	x	x	x
ENHANCED_FOR_STATEMENT	x	x	x			THROW_STATEMENT	x	x	x	x
FOR_STATEMENT	x	x	x			TRY_STATEMENT	x	x	x	
IF_STATEMENT	x	x	x			VARIABLE_DECLARATION_STATEMENT	x	x	x	x
LABELED_STATEMENT	x	x				WHILE_STATEMENT	x	x	x	
METHOD_INVOCATION	x	x	x	x						

Note that the vast majority of these source code entities are statements, but postfix expressions and prefix expressions are also included. These expression types were added because a loop is deconstructed into the initializer expression, condition expression, and update expression.

4.3 Case Study

In order to validate the extended change taxonomy I extract the source code changes of fault fixes from two versions of the Eclipse Platform. I chose Eclipse version 2.0 and Eclipse version 3.0 for the case study in this research. In this section I describe the Eclipse platform and provide information about the versions that I selected.

The Eclipse platform was developed as a common basis for integrated development environments (IDEs) [68]. Multi-tier applications use a number of different technologies, which require a diverse collection of tools. The Eclipse platform was developed with open application programming interfaces (APIs) to allow the integration of multiple tools in a single platform. Eclipse accomplishes this level of integration through a component-oriented architecture. Besides a minimal base, the Eclipse Runtime, all functionality is added through Java modules called Plug-ins [68].

Eclipse 2.0 was released on June 7, 2002. According to available sources, the primary focus was quality improvement and performance, with a lesser emphasis on new features [69]. Eclipse 2.0 consisted of 3 subprojects, the Eclipse Platform, the JDT (Java development tooling), and the PDE (Plug-in development environment).

Beginning with Eclipse Version 3.0, Eclipse became a Rich Client Platform [70]. This required Eclipse to change its underlying architecture. The Eclipse project adopted the OSGi Service Platform. Gruber et al. describe the transition from a proprietary framework to a framework based on OSGi [70]. This change is significant for my purposes, since the two versions of the product are separated by approximately 2 years and represent a significant change in architecture.

Eclipse 3.0 was released on June 21, 2004. The development plan for Eclipse 3.0 outlines a number of themes for each subproject [71]. The Eclipse Platform focused on user experience, more responsive UI, and rich client platform capabilities. The JDT focused on support for other JVM-based languages and improved user experience for Java developers. The PDE

subproject worked on support for the new plug-in format, testing, and improving the scalability of its model implementation.

Multiple artifacts for Eclipse are publicly available. The source code for Eclipse 2.0 and 3.0 is kept in a Concurrent Versioning Systems (CVS) repository. The problem tracking system is a customized version of Bugzilla[3]. Some descriptive statistics for Eclipse 2.0 and Eclipse 3.0 are given in Table 6.

Table 6 - Descriptive Statistics for Eclipse Versions

Version	Fault Fixes	Files Involved	Lines Added	Lines Removed	Start Date	End Date
Eclipse 2.0	3335	13047	208257	124313	1/8/2002	9/27/2002
Eclipse 3.0	8160	45096	1440617	1140349	12/22/2003	12/21/2004

Multiple researchers have used the Eclipse source code and problem tracking system to conduct software engineering research. Zimmermann et al. mined Eclipse 2.0, 2.1, and 3.0 to build software prediction models [72]. The data sets from these prediction models are publicly available[4]. Moser et al. extended this research by comparing the ability of change metrics to predict faults [73]. Moser et al. concluded that change metrics, such as the number of changes that are made to a file, are more effective at predicting faults than static metrics, such as the number of source code lines or the complexity of a method.

Krishnan et al. investigated the use of change predictors to predict fault-prone files in a product line [74]. The study by Krishnan et al. treats the Eclipse platform as a product line, and each project as an application that is delivered from that product line. They found that prediction results improve significantly as the product evolves. Krishnan et al. also made their dataset, scripts, and databases publicly available. This research builds upon the Krishnan et al. set of artifacts.

[3] https://bugs.eclipse.org/bugs/
[4] https://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/

4.4 Data Collection

The first step in data collection is to transform the data in the database into a format that can be used to drive my process. I use Pentaho Data Integration tool (aka Kettle) as the Extraction, Transformation, and Loading (ETL) tool[5]. The resulting database schema is a star schema, a common approach for business intelligence databases, which includes dimension tables and fact tables. The schema is depicted in Figure 2.

Each file is described in the *file_dim* table, including the full path and the date/time that the file was added to the system. Each file has one or more revisions in the *file_revision_dim* table. The revision number, as well as the number of lines added and removed, is captured as recorded by CVS. Since a fix can be attributed to multiple faults, the *fix_commit_fact* table has one entry per commit, per problem report. This results in a many-to-many relationship between the *fix_commit_fact* table and the *file_revision_dim* table.

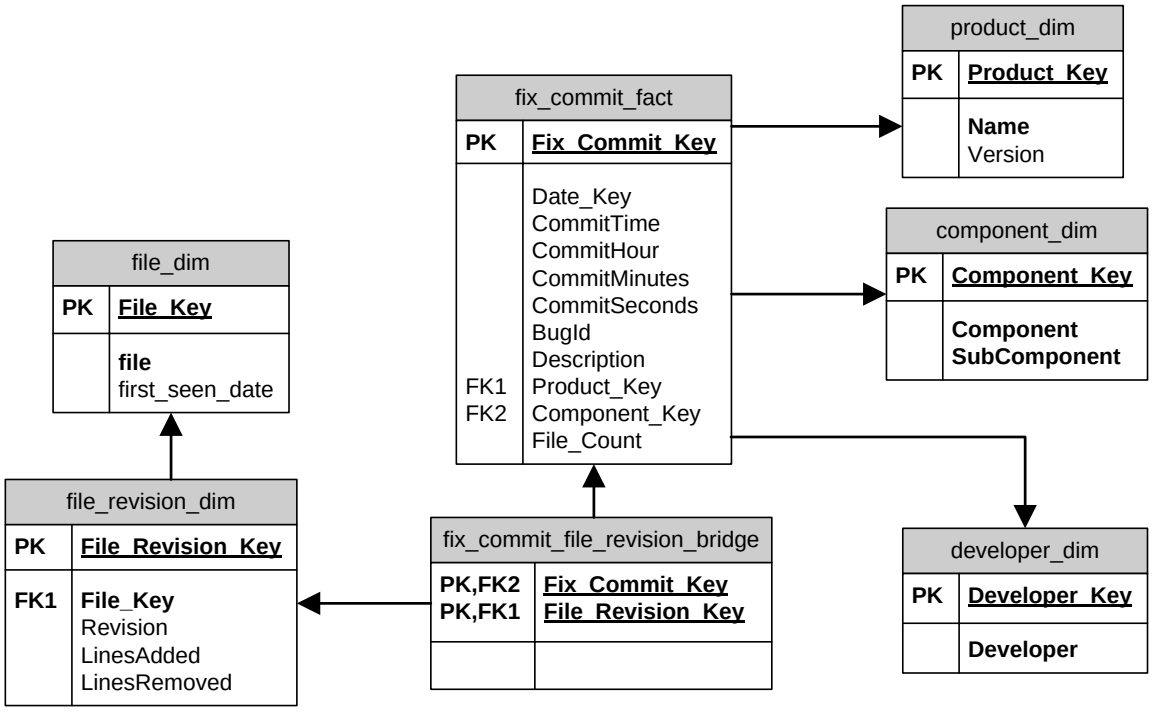

Figure 2 - Star Schema for Eclipse Fault Fix Data

[5] http://community.pentaho.com/projects/data-integration/

The *product_dim* table, the *component_dim* table, and the *developer_dim* table contain information about the product and version, the component and subcomponent, and the developer that committed the files to CVS. These tables can be used to query information from the *fix_commit_fact* table based on these attributes.

4.4.1 Data Collection Workflow

MiSFIT processes each fault according to a simple workflow. File revisions before and after each fault fix are retrieved from the CVS source code repository and stored locally. The workflow is service-based, with each service pulling work from a message queue, performing a single task, and putting the work on the next queue. The workflow is shown in Figure 3 and described in more detail below.

The primary advantages of this approach are scalability, reliability, flexibility, and modularity. Scalability is achieved by adding additional instances of each service. Multiple instances can safely pull from a single queue. The message queue also provides reliability. If a service fails while processing work the item is returned to the queue after a timeout period. This allows another instance of the service to pick it up and process it. The system is flexibile because I can add or remove processing steps easily. Finally, modularity is high because each service performs a simple task. The overall complexity of each service is relatively simple.

The initiation controller formats the data into an XML file with the following fault data: product, release, component, subcomponent, fixDate, bugId, author, and description. In addition, for each file I include the path, revision, lines added, lines removed, and the date/time in which the file was first seen. The xml file is placed in a local file store, and a message is placed on the *Fetch Queue*.

The *File Fetch Service* retrieves the message from the fetch queue. The service reads the xml, and for each file, it retrieves the version of the file before and after the stated revision. These files are placed in a local file store and the xml file is updated with their location. Their locations are recorded as two

attributes on the file, *preRepair* and *postRepair*. Once this is completed, MiSFIT stores the updated archive file in a document repository and removes the message from the *Fetch Queue*. The message is then placed on the *Distill Queue.*

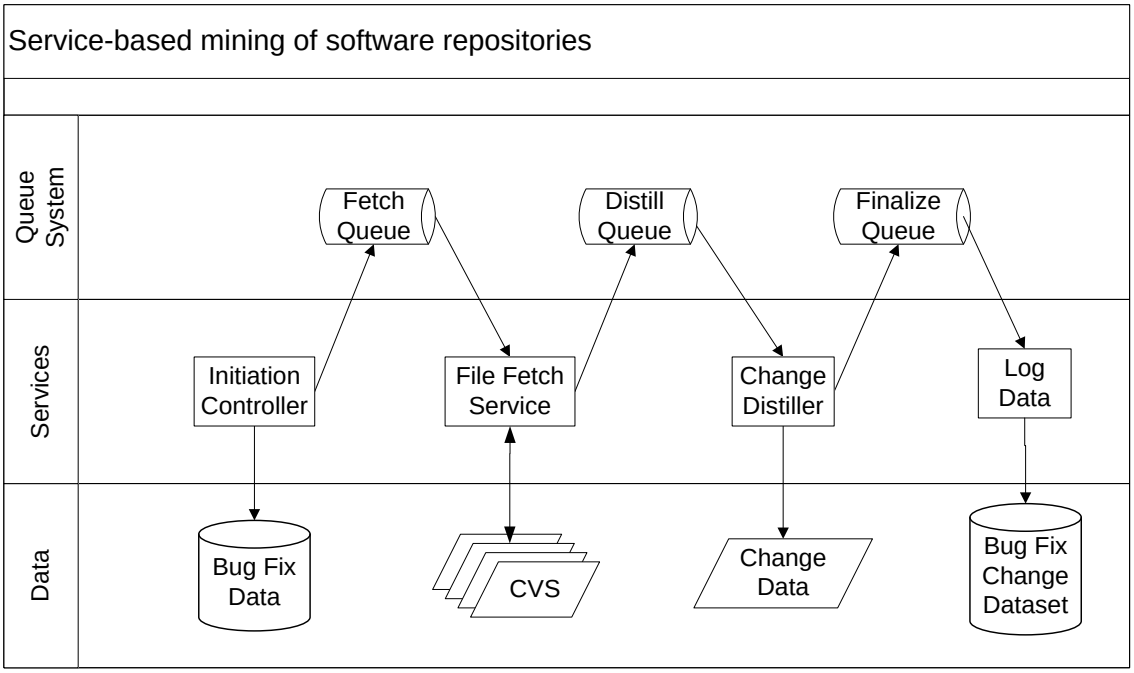

Figure 3 - A Service-based source code mining

I am using the Evolizer toolset, and specifically the ChangeDistiller component, from the University of Zurich to collect the syntactic change types between two versions of a file [75]. By default this tool acts in a batch mode, processing all of the versions for all of the files in a given project. For the Eclipse source code, this presented problems. There are many individual projects in the system, and there are a large number of changes that are of no interest to this research (do not repair faults). I utilized the Stand-alone ChangeDistiller tutorial[6] on the tool's website as a basis for an OSGi plugin. This allows us to treat the ChangeDistiller as a service. MiSFIT provides two files and the ChangeDistiller service provides a list of the change types that occurred between the two versions. The Change Distilling process is discussed in more detail below. I then

[6] https://www.evolizer.org/wiki/bin/view/Evolizer/Tutorials/StandaloneChangeDistiller

add the change types to the xml, update the local file store, and place the message on the *Finalize Queue*.

The *Log Data* service is responsible for parsing the xml and updating a relational database with the information. The use of a relational database makes it easy to perform reporting and data export to a variety of formats for analysis.

4.4.2 Change Distilling Process

The fine-grained source code changes are extracted for each pair of files using the ChangeDistiller tool [75]. Fluri et al. describe the change distilling process, where the abstract syntax trees of each revision of the source code are compared and source code changes are extracted [65].

I use the *changed entity* information available from the ChangeDistiller API to identify the type of statement that was altered, such as an *if statement* or *method invocation*. All of the information for each change is recorded in an SQL database and the extension is performed through the use of an SQL script. A database trigger is used to append the changed entity's type to the change type. Once the database is populated with all of the source code changes, a query is used to collect the type and count of source code changes that are recorded for each fault in the dataset.

4.5 Validation

Validation of the taxonomy occurs in two phases. In this section I describe my work to validate that the extended change types provide useful information for fault fixes. In the next chapter I continue validation by clustering these faults and manually inspecting a subset of the faults. My rationale is that in order to be useful, the extended change types must occur frequently in fault fixes. If these change types are infrequent in fault fixes, then the additional granularity that is gained by adding the extended types adds no new information. On the other hand, if multiple extended change types occur frequently I should consider these extended change types as features and evaluate their usefulness for understanding the data.

In this section, I evaluate the frequency of extended change types in software fault fixes as compared to the original change taxonomy. The top twelve change types that are extracted from fault fixes in Eclipse 2.0 and 3.0 are the same, and are presented in Table 7 with frequency of occurrence.

Table 7 - Top Twelve Change Types for Fault Fixes

Change Type	Eclipse 2.0		Eclipse 3.0	
	Commits	Percent	Commits	Percent
Insert If *	1512	52.39%	3415	52.21%
Insert Method Call *	1391	48.20%	3039	46.46%
Insert Var Decl *	1145	39.67%	2637	40.31%
Statement Parent Chg	1098	38.05%	2555	39.06%
Add Functionality	979	33.92%	2205	33.71%
Update Method Call *	958	33.19%	2095	32.03%
Insert Assignment *	937	32.47%	2238	34.21%
Delete If *	934	32.36%	2239	34.23%
Delete Method Call *	861	29.83%	1883	28.79%
Insert Return *	777	26.92%	1750	26.75%
Update Var Decl *	734	25.43%	1850	28.28%
Cond Expr Change	731	25.33%	1853	28.33%

The first column indicates the change type. Change types that were introduced by my extension to the taxonomy are denoted by an asterisk (*). The second and fourth columns provide the number of commits that are associated with a fault fix that contained at least one instance of the change type for each version of the software. The third and fifth columns provide a percentage of the total number of commits that include the change type.

The total number of extended change types in this list provides evidence that the extended change types provide additional granularity that is useful in the analysis of software fault fixes. The change types occur with surprising consistency between the two versions. This led us to question whether the frequency between the two versions is consistent. The following hypotheses are used for investigation.

H_0: The frequency distributions of extended change types in Eclipse 2.0 and Eclipse 3.0 are not the same (α=0.05).

H_A: The frequency distributions of extended change types in Eclipse 2.0 and Eclipse 3.0 are the same (α=0.05).

The data is not normally distributed, so the non-parametric Wilcoxon signed rank test is performed to test the hypothesis. The test was performed against the number of commits for each extended change type in the dataset. The test indicates that there is no significant difference in the frequency of the change types, with a *p*-value of 0.0005. I reject H_0 in favor of the alternative and conclude that the occurrence of change types is consistent in these two versions of the software.

4.6 Conclusions

In this chapter I have described an extended change taxonomy and validated its usefulness for fault analysis. First, I described the change taxonomy provided by Fluri and Gall [66], including its limitations with regards to analyzing software faults. I provided a proposed extension that utilizes information that is collected by the ChangeDistiller tool [75].

As a case study, I selected two versions of Eclipse. I included software faults from multiple Eclipse projects in the analysis. Data collection began with the extraction and transformation of an existing research database provided by Krishnan et al. [74]. From this starting point, a service-based workflow that utilizes a message queue system to coordinate work was described. The data collection workflow is used throughout this work.

In order to move forward with in-depth analysis of the data I need to validate the usefulness of the extended change taxonomy. I found that nine of the top twelve change types in software faults from my case study are extended change types. In addition, I discovered that there is no significant difference in the distribution of these extended change types in Eclipse 2.0 and Eclipse 3.0 when only fault fixes are considered. These results provide evidence that the extended change taxonomy provides useful information and that additional research is warranted.

Chapter 5

Clustering Software Faults[7]

This chapter describes a process for clustering software faults based on the changes that were made to repair the software fault. The goal is to characterize the software fault from the fix that repaired it using an automated process. In this chapter I describe the clustering tools, my clustering process, and my validation of the clustering results.

5.1 Clustering Software Faults

The input to the clustering process is a vector. The features of the vector are the extended change types. One hundred and one extended change types were present in the Eclipse 2.0 dataset and one hundred and nine change types were present for Eclipse 3.0. The change types were presented in Chapter 4.

A summary of the process is depicted in Figure 4. The files involved in the fault fix are extracted from the source code repository. The abstract syntax tree is instantiated and processed to extract the change types. Each change type is a feature in the vector and the frequency of a change type for a particular fault is recorded as the value of that feature for the fault's vector in the dataset.

[7] © 2014 IEEE. Reprinted, with permission, from Bill Kidwell, Jane Huffman Hayes, Allen P. Nikora, "Toward Extended Change Types for Analyzing Software Faults", Proceedings of the 14th International Conference on Quality Software (QSIC), Oct. 2014.

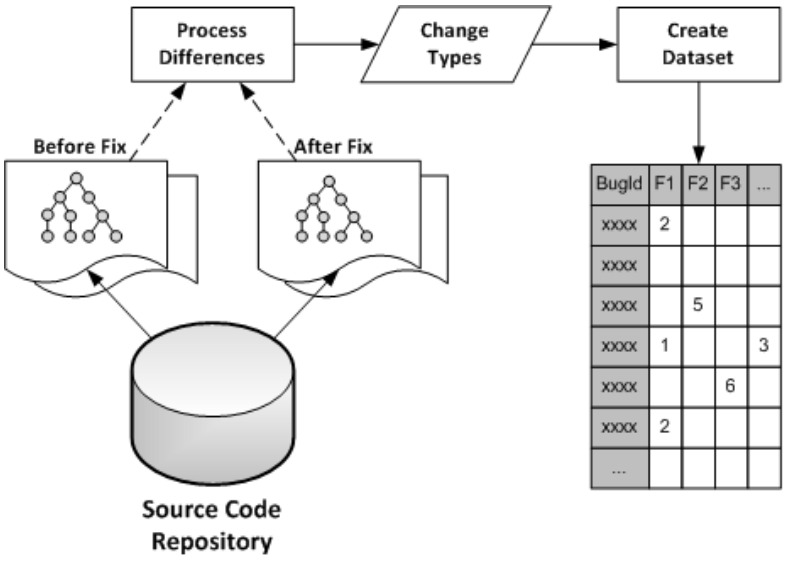

Figure 4 - Dataset Creation Overview

For example, Bug # 10009, shown below in Figure 5, consisted of four changes: JavaDoc comments were inserted, an *if statement* was added, the dispose method call was updated, and the parent of the method call was changed. For this fault the vector has the following values: Insert_If = 1, Statement_Parent_Change = 1, Update_Method_Call = 1. The changes to comments are recorded, but discarded for purposes of this study.

```
+       /**
+        * @see AbstractUIPlugin#shutdown()
+        */
        public void shutdown() throws CoreException {
                JDIDebugModel.removeHotCodeReplaceListener(this);
                JavaDebugOptionsManager.getDefault().shutdown();
-               getImageDescriptorRegistry().dispose();
+               if (fImageDescriptorRegistry != null) {
+                       fImageDescriptorRegistry.dispose();
+               }
                super.shutdown();
        }
```

Figure 5 - Source Code Changes for Bug 10009

5.2 Measurements

The CLUTO clustering toolkit is used to perform clustering of the data [76]. CLUTO was selected based on its inclusion of cosine similarity as a distance measure and visualization features that aid in the analysis of the clusters. CLUTO creates a hierarchical clustering solution when the repeated bisection approach is used [77]. The hierarchical solution provides views of the data at different levels of granularity, and in my case allows us to compare hierarchies in data from multiple datasets.

A complimentary project, gCLUTO, provides an easy method to get familiar with the tool and visualize data [78]. The gCLUTO interface provides a convenient method to try different clustering parameters and visualize the results. It also provides the Mountain Visualization, which we discuss in more detail below.

CLUTO treats the clustering problem as an optimization process which seeks to maximize or minimize a particular criterion function [76]. All documents are initially partitioned into two clusters. One of the clusters is selected and bisected. This process is repeated k-1 times to arrive at k clusters. CLUTO provides seven different criterion functions that can be used to guide the clustering process. A simple, greedy scheme is used to optimize the selected criterion function [79]. During multiple iterations of refinement, each instance in a cluster is visited in random order and moved to the cluster that improves the criterion function's value. This iterative refinement is repeated until no instances are moved. In order to avoid the selection of a local maximum or local minimum, the entire process is repeated ten times and the best solution is selected.

CLUTO offers multiple similarity measures. My initial analysis of clustering tools identified the cosine similarity as the most effective measure to produce reasonable clusters in the size range of 7-20 clusters. For two vectors v_i *and* v_j, the cosine similarity function [80] is defined as follows:

$$similarity = \cos(\theta) = \frac{v_i \cdot v_j}{\|v_i\|\|v_j\|}$$

The cosine similarity ranges from zero (completely orthogonal) to one (identical), since the frequencies of the change types are always non-negative. The internal similarity is the average similarity between all objects of the cluster. An internal similarity near one represents a “tight” cluster. I focus my evaluation of clusters on the internal similarity since I am trying to group software fixes with similar syntax. To maximize the internal similarity I limit my evaluation to the use of the I1 and I2 criterion functions. The external similarity is the average similarity between the objects of each cluster with the rest of the objects. An external similarity near zero represents a cluster that is well-separated from other clusters in the data set. I report the external similarity but do not use it for evaluation.

I define *n* as the number of fault vectors, *k* as the number of clusters. *S* is the set of vectors that I want to cluster. S_1, S_2, …, S_k denotes each of the *k* clusters. I define n_1, n_2, …, n_k as the size of the *k* clusters. The composite vector D_i, is defined by the sum of all vectors in cluster S_i.

$$D_i = \sum_{v \in S_i} v$$

The centroid vector is obtained by averaging the features from all of the vectors in cluster S_i.

$$C_i = \frac{D_i}{|S_i|}$$

I1 maximizes the sum of the average pairwise similarities between the instances in the cluster. The I1 criterion function is defined [81] as:

$$\text{maximize } I1 = \sum_{r=1}^{k} n_r \left(\frac{1}{n_r^2} \sum_{v_i, v_j \in S_r} \cos(v_i, v_j) \right)$$

The innermost term of this equation is the cosine similarity between two instance vectors. The similarity is calculated between every two instance vectors in the cluster and these similarities are summed. The average is calculated by dividing by the squared size of the clusters, and this is weighted by multiplying by the size of the cluster. I1 maximizes weighted average for all clusters. A useful

way to visualize this criterion function is presented by Ted Pedersen[8]. You can imagine that each instance in the cluster is a point, and that you are connecting a string between each set of points. The length of the string connecting the points represents the distance, which is the inverse of the similarity. The goal is to end up with a tight ball of string.

I2 maximizes the similarity between each instance and the centroid of the cluster, similar to the vector-space of the *K*-means algorithm [81]. The I2 criterion function is defined as:

$$\text{maximize } I2 = \sum_{r=1}^{k} \sum_{v_i \in S_r} \cos(v_i, C_r)$$

The innermost term of this equation is the similarity between each instance vector in the cluster and the cluster's centroid. This similarity is summed for all instance vectors in the cluster. The I2 criterion function maximizes this for all clusters in the solution. This criterion function can also be visualized, but in this case, as a flower[8]. Imagine that a piece of yellow string is stretched from the centroid to each point in the cluster. Again, the length of the string is to be minimized. In this case, you end up with a small, round flower.

CLUTO provides metrics to aid in cluster analysis. For each cluster, the internal similarity (**iSim**) and external similarity (**eSim**) are reported, along with their standard deviations (**iSDev** and **eSDev**).

Clusters are numbered from zero to *k*-1. The clusters are ranked by subtracting the ISim value from the ESim value, and sorting largest to smallest [76]. The size is the number of instances that have been assigned to this cluster. The ISim, as described above, is the average internal similarity of the cluster. The ESim is the average similarity of each instance in the cluster with items from the other clusters.

CLUTO reports a number of features that account for the internal similarity of a particular cluster. These are referred to as *descriptive* features [76]. A

[8] http://sourceforge.net/p/senseclusters/mailman/message/692149/

percentage is provided with each feature. An example of the output from CLUTO for a cluster is given in Table 8.

Table 8 - Example Cluster Metrics from Cluto

Cluster 0 Size: 113 ISim: 0.732 ESim: 0.095

Descriptive:	UPD_VAR_DECL	97.3%
	INS_METH_CALL	0.6%
	ADD_FUNC	0.5%
	INS_VAR_DECL	0.4%
Discriminating:	UPD_VAR_DECL	51.6%
	INS_IF	11.1%
	INS_METH_CALL	8.9%
	STATEMENT_PARENT_CHANGE	4.9%

The descriptive and discriminating features are ranked from largest contribution to the similarity of the items in a cluster, to the lowest. The number of features reported is configurable. In this cluster the UPD_VAR_DECL feature (Update Statement: Variable Declaration) accounts for 97.3% of the similarity between instances in the cluster. The same feature differentiates the instances in the cluster from instances in other clusters by 51.6%.

The descriptive features are used in this study to characterize and label each of the clusters and make a conjecture about the types of faults that belong to the group. Labeling of the clusters is entirely based on the statistical prominence of the features in the cluster, and not based on subjective evaluation of the results. I use a cutoff threshold of 10% in order to name the cluster. All features with a discriminating feature value equal to or above 10% are included in the cluster name (e.g., Statement Parent Change + Insert If). This allows us to compare clusters from different datasets.

5.3 Experimental Design

The purpose of this study is to analyze software faults and the naturally occurring groups that result from clustering the faults. The frequency of the syntactical elements that were changed in the fix for the fault are used as the input to the clustering algorithm. My goal is to understand how effectively the syntax of the changes can characterize the nature of the software fault, and

ultimately to determine whether I can use this clustering as a form of automated fault classification.

The study is described using the Goal/Question/Metric (GQM) template for goal definition [82][83].

> *Analyze the* ***clustering of software faults***
> *for the purpose of* ***characterizing fault classes***
> *with respect to their* ***effectiveness***
> *from the point of view of the* ***researcher***
> *in the context of* ***two versions of a large, open source system****.*

5.3.1 Variables

The mean internal similarity (iSim) is used to measure the effectiveness of a clustering solution. This value is calculated by calculating the mean value from the iSim value for each cluster in the solution.

5.3.2 Evaluation of Criterion Functions

In order to proceed with the clustering and inspection of the faults, I must choose the most appropriate criterion function. Clustering is performed for fault data for Eclipse 2.0 and Eclipse 3.0. I repeat the clustering for all values of k from 2 to 20. The number of fault types in a fault taxonomy should be manageable and not too large [39]. Based on this recommendation, I expect there to be seven to ten fault types. I choose a broad range of numbers to be inclusive. I use the following hypotheses for investigation.

H_0: There is no difference in the mean internal similarity of clusters when using the I1 and I2 criterion functions (α=0.05).

H_A: The mean internal similarity of clusters when using the I1 criterion function is greater than the mean internal similarity of clusters when using the I2 criterion function (α=0.05).

The mean internal similarity for each of these methods is presented in Table 9. The number of clusters, *k*, is shown in the first column. The remaining columns report the internal similarity for each method, for each version. A graph of these values for the Eclipse 2.0 dataset is presented in Figure 6. A similar graph for Eclipse 3.0 is displayed in Figure 7.

I perform a one-tailed paired samples Wilcoxon signed rank test on the similarity data for I1 and I2 to evaluate the hypothesis. A paired t-test was considered, but the data does not pass a test for normality, and thus the non-parametric test is used. I perform the test independently for both versions of Eclipse. For Eclipse 2.0, the p-value = 3.815e-06 and for Eclipse 3.0, the p-value = 3.624e-05. In both cases I am able to reject the null hypothesis in favor of the alternate hypothesis.

Table 9 - Mean Internal Similarity
© 2014 IEEE

	Eclipse 2.0		Eclipse 3.0	
k	**I1**	**I2**	**I1**	**I2**
2	0.292	0.282	0.297	0.289
3	0.329	0.317	0.333	0.322
4	0.404	0.401	0.412	0.415
5	0.475	0.429	0.439	0.443
6	0.497	0.449	0.526	0.468
7	0.517	0.462	0.546	0.494
8	0.535	0.487	0.551	0.510
9	0.561	0.495	0.566	0.528
10	0.567	0.499	0.571	0.531
11	0.577	0.506	0.584	0.539
12	0.580	0.503	0.591	0.571
13	0.584	0.511	0.601	0.569
14	0.593	0.514	0.612	0.574
15	0.597	0.521	0.614	0.576
16	0.602	0.543	0.617	0.580
17	0.606	0.549	0.601	0.585
18	0.607	0.555	0.604	0.587
19	0.621	0.561	0.615	0.599
20	0.624	0.567	0.622	0.630

Zhao and Karypis provide an analysis of document clustering solutions using the I1 and I2 criterion functions in their comparison of criterion functions [79], [81]. In general, all criterion functions have different sensitivities based on the tightness of the clusters and the degree of balance in the resulting solution. Zhao and Karypis analyze the I1 and I2 functions to explain how the I1 criterion function can lead to several pure, tight clusters and a single large, poor quality cluster. This poor quality cluster is referred to as a “garbage collector” and results from the function’s tendency to exclude peripheral documents from the pure clusters.

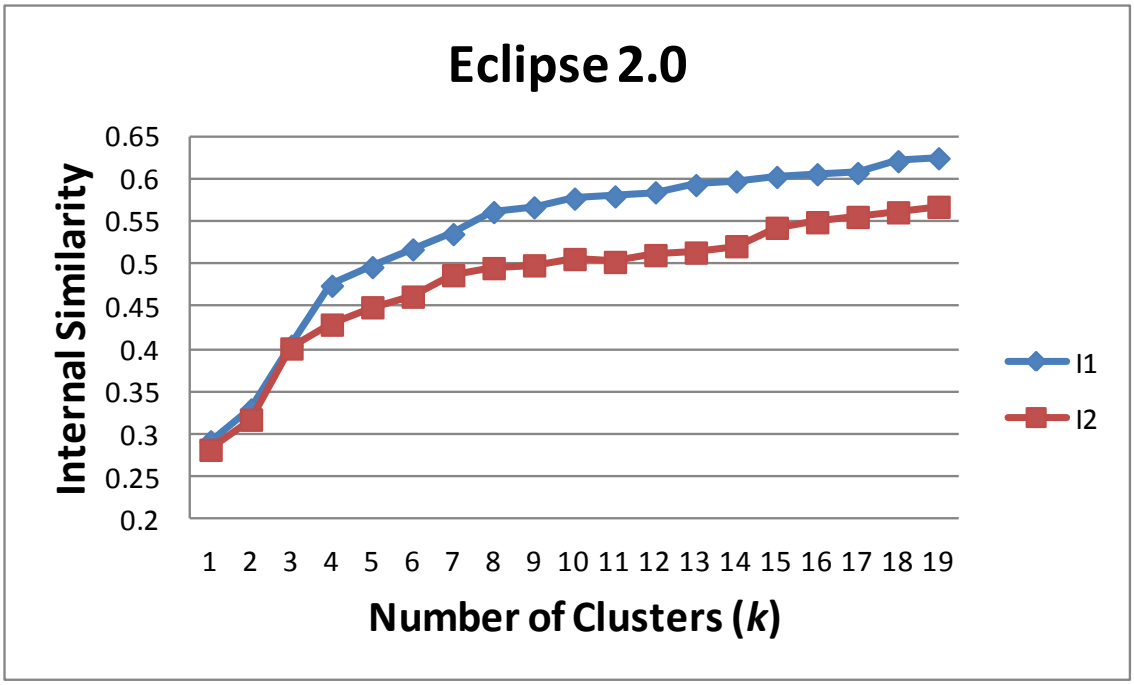

Figure 6 - Mean Internal Similarity of Eclipse 2.0

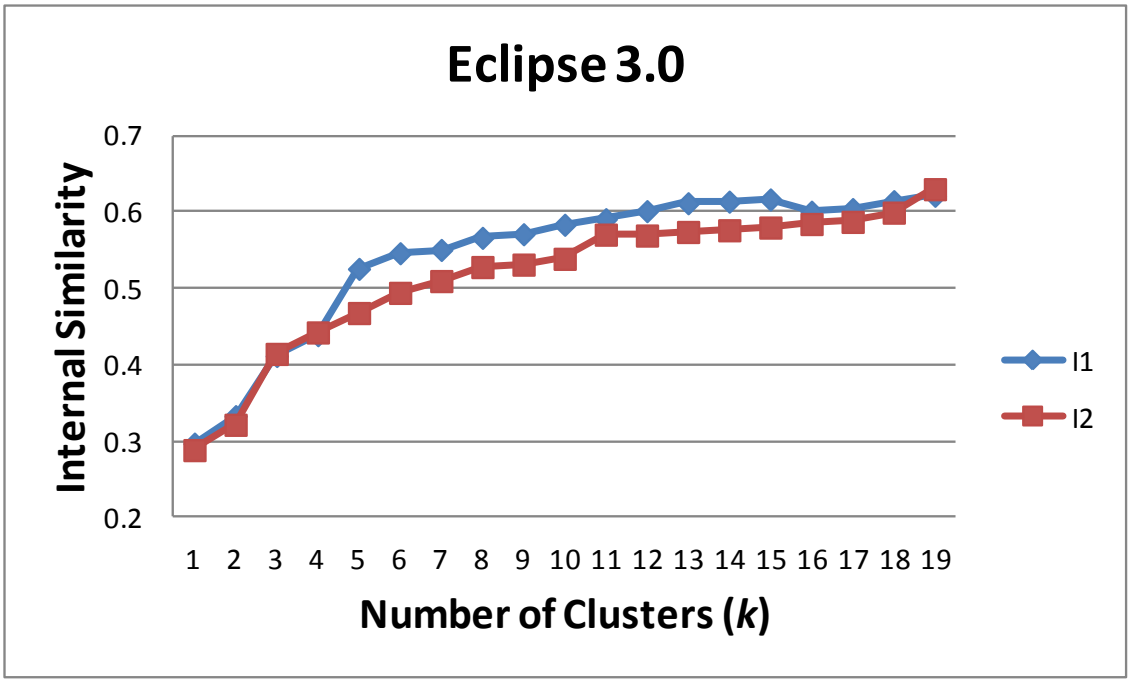

Figure 7 - Mean Internal Similarity of Eclipse 3.0

Zhao and Karypis conclude that this property of the I1 criterion function may be useful in noisy data sets [79]. This helps explain the superiority of the I1 criterion function in my experiment, and suggests that more analysis of the instances in the "garbage collector" may allow the taxonomy to be refined. While these faults occur infrequently, there may be patterns of changes over several releases, or across multiple projects.

5.3.3 Consistency of Clusters for Eclipse 2.0 and 3.0

In this section I analyze the consistency of the clustered fault fixes for Eclipse 2.0 and Eclipse 3.0 at k=10. I choose this value of *k* due to similarities in the descriptive features across the two versions of Eclipse. The groups appear to stabilize at this value of *k*. Other researchers have also used a value of k=10, it is on the high end of the number of fault classifications that are recommended by best practices [39]. I label each cluster based on the descriptive features reported by CLUTO. The top five descriptive features of each cluster are reported, regardless of their significance. In clusters where a single feature dominates it is possible to use the largest value as the label for the cluster. To properly represent the clusters with multiple features I use a threshold value of 10% to label the clusters. For example, Cluster 4 below reports descriptive features as Insert Return (47.3%), Insert If Statement (36.4%), Delete Return (5.0%), Insert Variable Declaration (3.5%), and Insert Method Call (1.7%). It is interesting to know that these features occur together, but the first two features identify the nature of the faults in the cluster. This cluster is labeled "Insert Return + Insert If Statement." The threshold value of 10% allows this labeling to occur automatically.

The cluster features, sizes, and similarities are reported in Table 10. The first row reports on the clusters that are described by the update of a variable declaration. In Eclipse 2.0, this cluster included 94 faults, 3.3% of the total, while in Eclipse 3.0 the cluster includes 261, 4.3% of the total. The last row of the table contains totals for the number of faults in each data set.

Table 10 - Comparison of Clustered Faults

	Eclipse 2.0		Eclipse 3.0	
Cluster (Descriptive Features)	**Size**	**iSim**	**Size**	**iSim**
Upd Var Decl	94 (3.3%)	0.789	261 (4.3%)	0.724
Cond Expr Chg	139 (4.8%)	0.708	244 (4.0%)	0.834
Add Func	132 (4.6%)	0.678	441 (7.2%)	0.599
Upd Method Call	266 (9.2%)	0.663	494 (8.1%)	0.654
Ins If + Ins Return	164 (5.7%)	0.58	0 (0.0%)	-
Ins If + Stmt Parent Chg	446 (15.5%)	0.57	908 (14.9%)	0.584
Ins Meth Call	434 (15.0%)	0.566	756 (12.4%)	0.582
Del Meth Call + Ins Meth Call	279 (9.7%)	0.525	669 (11.0%)	0.513
Ins If + Ins Meth Call + Ins Var Decl	554 (19.2%)	0.504	1049 (17.2%)	0.515
Ins Assign + Upd Assign	376 (13.0%)	0.084	706 (11.6%)	0.128
Ins Assign + Ins If	0 (0.0%)	-	567 (9.3%)	0.579
Total	**2884**		**6095**	

Notice that Eclipse 2.0 has a cluster described by the insertion of *if* and *return* statements, while Eclipse 3.0 has a cluster that is described by the insertion of *assignment* and *if* statements. In order to compare the clustering solutions, I treat these as empty clusters in the versions where they do not occur. I use the following hypotheses for investigation.

H_0: There type and size of clusters in the is no significant correlation in the clustering solutions of Eclipse 2.0 and Eclipse 3.0 at k=10 (α=0.05).

H_A: The clustering solutions of Eclipse 2.0 and Eclipse 3.0 at k=10 are correlated (α=0.05).

To test the hypothesis, Pearson's correlation coefficient is calculated. A Shapiro-Wilk test for normality was performed to verify that the data is normally distributed. The value of *r* for the data is 0.778, with a *p*-value = 0.004, allowing us to reject the null hypothesis and conclude that the cluster types and sizes are correlated.

5.4 Manual Inspection of Faults in Each Cluster

In this section I present clustering results on Eclipse 2.0 fault fixes using the I1 criterion function and setting *k*=10. The Eclipse 2.0 dataset consists of 101 fine-grained source code change types after expanding statement insert, update, delete, and ordering change types and eliminating changes to comments and source code documentation. There are 2884 faults in the dataset with Java source code changes. Faults with zero Java source code changes, e.g., those requiring only changes to properties or xml configuration files, are not included in the analysis. CLUTO reports a number of metrics for the clusters. These metrics are presented in Table 11.

Table 11 - Cluster Statistics for Eclipse 2.0, *k* = 10

Cluster Id	Size	iSim	iSDev	eSim	eSDev
0	94	0.789	0.124	0.077	0.052
1	139	0.708	0.134	0.112	0.073
2	132	0.678	0.125	0.129	0.058
3	266	0.663	0.136	0.118	0.069
4	164	0.58	0.084	0.212	0.073
5	446	0.57	0.093	0.203	0.065
6	434	0.566	0.091	0.208	0.066
7	279	0.525	0.09	0.207	0.084
8	554	0.504	0.082	0.246	0.059
9	376	0.084	0.057	0.083	0.081

The CLUTO manual provides a full description of these metrics [76]. A summary is presented here. The Cluster Id is a zero-based integer assigned to each cluster. The Size is the number of faults that were assigned to the cluster. The column labeled iSim is the mean internal similarity of the faults in the cluster. The column labeled iSDev is the standard deviation of the mean internal similarities. Similarly, the eSim column is the mean similarity of the faults in the cluster with the faults that are not in the cluster, or the external similarity. The eSDev column is the standard deviation of the mean external similarity for the faults in the cluster. The clusters are ranked by subtracting the external similarity

from the internal similarity and arranging them in decreasing order. This positions tight, distinct clusters at the top of the list.

5.4.1 Data Visualization

The CLUTO toolset provides tools to visualize clustering results [76]. A modified version of the cluster plot visualization for the results that I manually analyzed is presented in Figure 8. The columns in the visualization are the clusters, with the size of each cluster in parentheses. The tree structure aids in understanding the relationships between clusters. For example, cluster 6 and 7 are very similar clusters, and contain similar source code changes. The rows of the visualization provide a subset of the 101 source code changes that were used as features during the clustering process. The darkness of the cells is based on the intensity of the feature within each cluster. For example, in the first column we see that cluster 5 is described by the *statement parent change* and *insert if statement* change types. The label for descriptive features is repeated to the left of each occurrence. As an example, Cluster 1, on the far right of the illustration, is described by conditional expression changes (COND EXPR CHG).

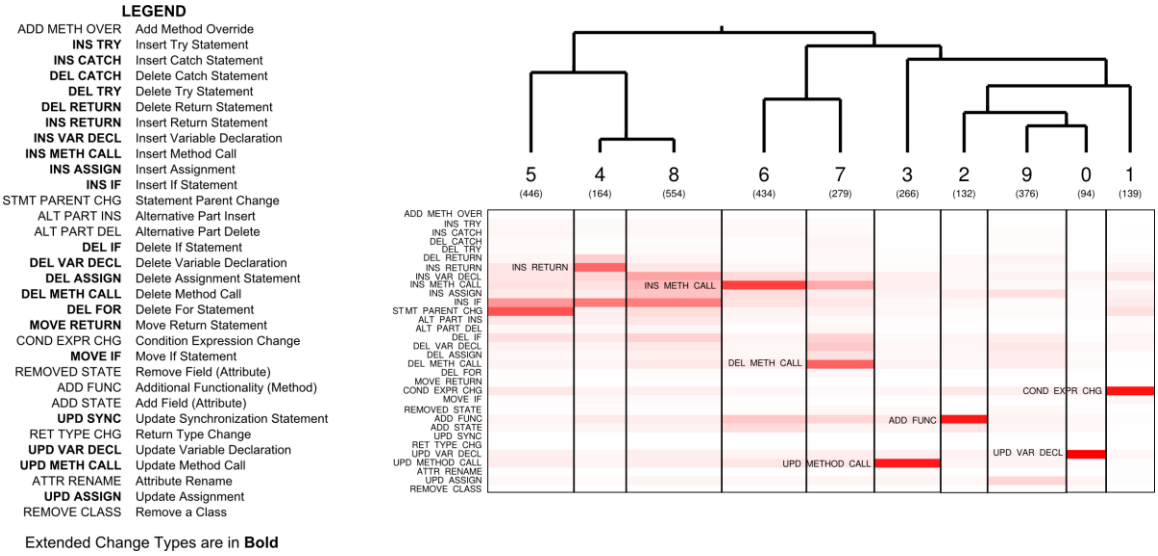

Figure 8 - Visualization of Clusters for Eclipse 2.0
© 2014 IEEE

A second visualization of the clusters is provided by gCLUTO. The mountain visualization aids the user in understanding high-dimensional data in a

lower-dimensional representation [78]. The visualization conveys the number of objects, internal similarity, external similarity, and standard deviation.

The mountain visualization for the Eclipse 2.0 dataset from gCLUTO is provided in Figure 9. Each peak represents a single cluster. The distance between two peaks conveys the relative similarity of the two clusters. This information is consistent with the tree structure in the matrix visualization (Figure 8). For example, the relative locations of clusters 0, 1, 9, and 2 are similar.

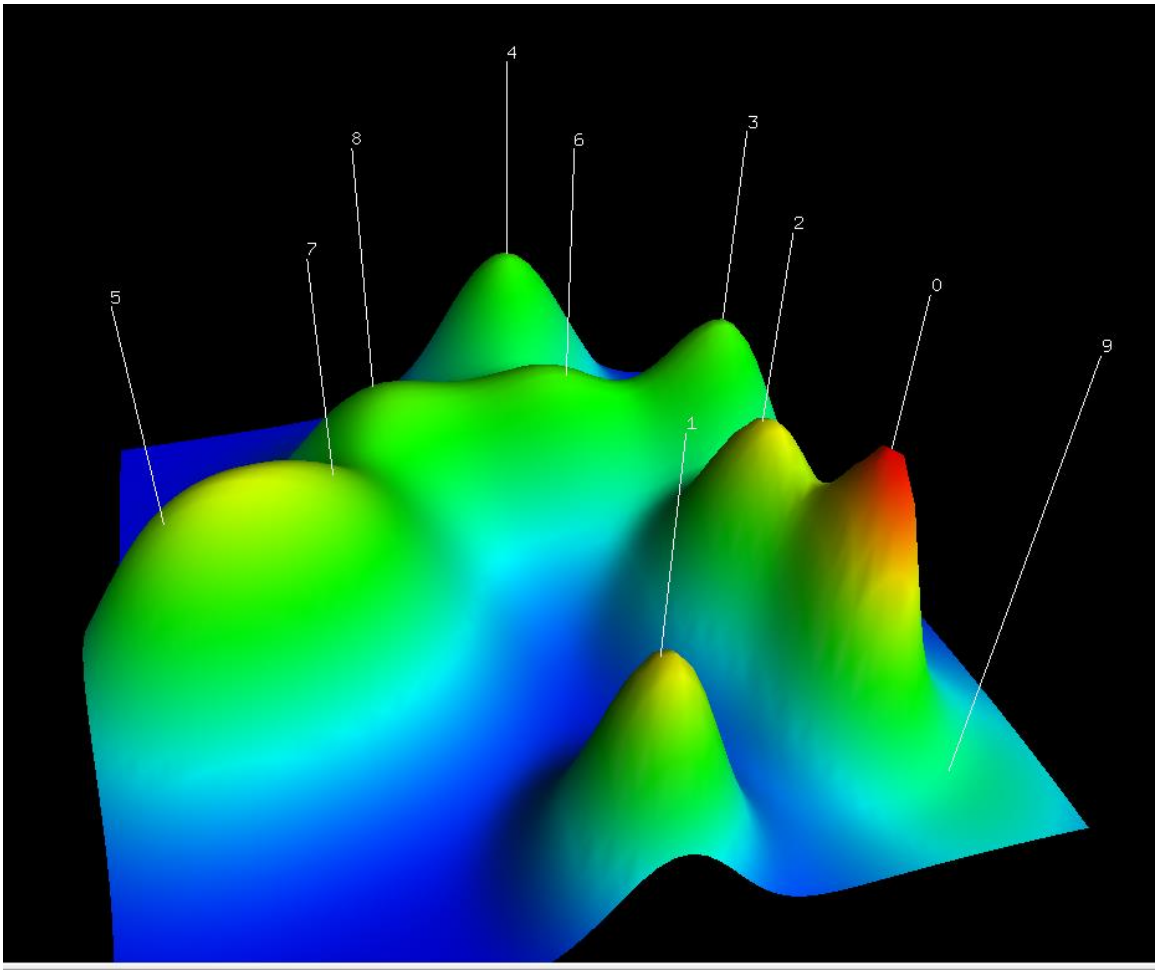

Figure 9 - Mountain Visualization of Clusters for Eclipse 2.0

The height of each peak is proportional to the internal similarity of the cluster. This can be seen by cluster 0 (iSim=0.789) and cluster 9 (iSim=0.084). The volume of the peak is proportional to the size of the cluster. Cluster 5 consists of 446 instances, and cluster 1 consists of 139. The color of the peak

represents the internal standard deviation. Red represents data with low deviation, while blue represents data with high deviation [78].

5.4.2 Manual Inspection Process

For each cluster I present internal clustering metrics, features that explain the clusters, and then conduct a manual inspection of five to eight faults. I randomly select the faults from each of the clusters for manual inspection. The fault reports for these faults are available on the Eclipse foundation Bugzilla web site[9].

In order to inspect these faults, a taxonomy is necessary. The primary question that I am seeking to answer is whether the syntactic patterns of the fault fixes in the cluster characterize the nature of the faults. In order to test this with the manual inspection, I first use the descriptive features and develop a set of expectations. The expectations relate the dominant syntactical features to the types of faults that are expected. During the manual inspection, I am trying to determine whether the fault that is being inspected falls within those pre-determined expectations.

Cluster 0: Update Variable Declaration

Faults in this cluster are expected to be the result of incorrectly initialized variables.

Cluster 1: Condition Expression Change

Faults in this cluster are expected to be simple logic changes. Some complex logic changes may also occur where there are multiple condition statements that check similar conditions and must change in unison.

Cluster 2: Additional Functionality

Addition of new functionality or overriding an inherited method.

[9] https://bugs.eclipse.org/bugs/

Cluster 3: Update Method Call

A method was used incorrectly, for example, incorrect parameters were passed or incorrect version of a method was called.

Cluster 4: Insert If + Insert Return

I expect the most common faults in this cluster to be unchecked pre-conditions. More complex changes may be algorithmic changes.

Cluster 5: Statement Parent Change + Insert If

Faults in this cluster are likely to be logic changes. These can range from checking faults to more complex logic changes.

Cluster 6: Insert Method Call

Faults in this cluster are expected to be missing functionality or interface faults where a required method was not called.

Cluster 7: Delete Method Call and Insert Method Call

Faults in this cluster are expected to require the removal of extraneous code, or are expected to be interface faults where the incorrect method was being called.

Cluster 8: Insert If + Variable Declaration + Method Call

Faults in this cluster are expected to be changes to algorithms or changes in behavior. These types are faults are expected due to the large number of change types that characterize the cluster.

Cluster 9: Garbage Collector

I expect faults in this cluster to be varied and uncommon. My aim in manually inspecting this cluster is to determine if any pattern can be found.

5.4.3 Manual Inspection Results

Cluster 0 – Update Variable Declaration

Cluster 0 is the tightest and smallest cluster in the selected solution. The *update variable declaration* change type explains over 98% of the similarity of the faults in the cluster. I expect faults in this cluster to represent faults where a variable is either uninitialized or incorrectly initialized. The metrics for this cluster appear in Table 12.

Table 12 - Cluster 0 Metrics

Cluster Id	0	Descriptive Features	
Size	94	Update Variable Declaration	98.5%
iSim	0.789	Condition Expression Change	0.4%
iSDev	0.124	Insert Variable Declaration	0.2%
eSim	0.077	Update Assignment	0.2%
eSDev	0.052	Additional Functionality	0.1%

Two of the five faults in this category fall in the expected category (10483 and 16828). In Bug 11110, a condition expression change is edited to check for null references. A portion of the change appears in Figure 10. The change requires the intermediate variable **window** on the new line 167. The window variable is used in the new condition on the new line for 168. This change is obfuscated because it occurs in a variable declaration for an anonymous class, an instance of **Runnable** that is declared on line 165.

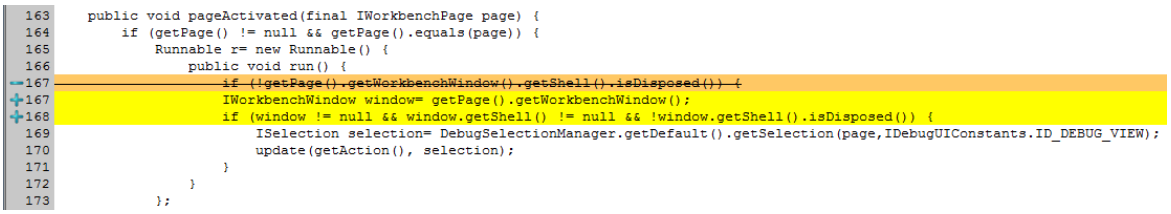

Figure 10 – Bug 11110: Fault fix to check for Null Pointer

The faults inspected from this cluster appear in Table 13. These descriptions explain my interpretation of the source code changes and allow other researchers to improve upon these results.

Table 13 - Faults Inspected for Cluster 0

Bug Id	Expected	Description
10483	Yes	Bug 10483 includes updates to variables that are subsequently used in method calls. These changes in values were necessary to support differences in operating systems.
11110	No	The changes were made within a variable declaration, but were within an anonymous class.
16828	Yes	Bug Id 16828 is fixed by changing the variable declaration for the point where a tooltip is displayed, thus avoiding overlap with other components and undesired interactions during usage.
18923	No	The fix for Bug 18923 has a number of updated variable declarations due to the fact that variable names were changed. These changes cause this fault to belong to this cluster, but do not characterize the fault.
23824	No	Bug 23824 is an interface fault. The project folder should be cast to type *ICVSRemoteFolder*, changing the call that was used to fetch the parent folder.

Cluster 1 – Condition Expression Changes

The presence of a *conditional expression change* in faults that belong to Cluster 1 explain 94.7% of the similarity values for these items. Simple logic errors are expected to belong to this cluster. Complex algorithmic faults requiring extensive logic changes may also be represented here. Four of the five faults I inspect are logic errors, while the fix for Bug 18787 is a more complex logic change. The metrics for this cluster are presented in Table 14.

Table 14 - Cluster 1 Metrics

Cluster Id	1
Size	139
iSim	0.708
iSDev	0.134
eSim	0.112
eSDev	0.073

Descriptive Features	
Condition Expression Change	94.7%
Statement Parent Change	1.7%
Insert Variable Declaration	1.4%
Insert If	0.7%
Insert Assignment	0.3%

Logic problems are a common cause for software faults and the source code changes are often small and contained. A small number of dominant change types easily characterize faults with these characteristics. The faults inspected from this cluster are described in Table 15.

Table 15 - Faults Inspected for Cluster 1

Bug Id	Expected	Description
15951	Yes	Bug 15951 was fixed with a single conditional expression change to repair a forgotten case for unmanaged remote files.
18482	Yes	Bug 18482 added the classpath to a conditional expression.
18787	Yes	Bug 18787 was a more complicated logic error. A condition and cast were added to the conditional expression, but the behavior of the *getSignature*() method was also changed.
21185	Yes	Bug 21185 added a predicate to consider the style of the component during the comparison.
21370	Yes	Bug 21370 fixed a failure that froze the editor. The fault was due to a problem with pattern matching that was repaired by changing a >= operator to a > operator so that the first character was not unread when the end sequence was not detected.

Cluster 2 – Additional Functionality

The similarity in Cluster 2 is explained by the addition of one or more new methods (95.2%). The metrics for this cluster are provided in Table 16. I expect faults in this cluster to include additions of new features and functionality. I investigate six faults in this cluster.

Table 16 - Cluster 2 Metrics

Cluster Id	2
Size	132
iSim	0.678
iSDev	0.136
eSim	0.118
eSDev	0.069

Descriptive Features	
Additional Functionality	95.2%
Additional State	1.2%
Condition Expression Change	1.0%
Insert Assignment	0.5%
Update Variable Declaration	0.3%

Five of the faults met my expectations for this category. The sixth, Bug 15513, is fixed by overriding a method of the base class. This type of fault logically belongs to the group, so I add it as an additional consideration for this cluster. The faults inspected from this cluster are described in Table 17.

Table 17 - Faults Inspected for Cluster 2

Bug Id	Expected	Description
11265	Yes	Bug 11265 required the addition of two convenience constructors to replace a source locator API that had been deprecated.
12297	Yes	The fix for Bug 12297 enhances the algorithm that checks for synchronization of local and server resources in the CVS module.
12573	Yes	The fix for Bug 12573 adds a *WM_NOTIFY* method in order to address a platform specific fault on Windows operating systems.
15513	New	Bug 15513 required that the *setToolTipText* method of the base class be overridden. This example exposes an additional type of fault that must be considered due to this syntax change.
15699	Yes	Bug 15699 was fixed by adding a method to provide an order to the components that should be placed on a dialog.
18473	Yes	The fix for Bug 18473 added a function that would indicate whether the context-sensitive help window was currently displayed.

Cluster 3 – Update Method Call

The faults in Cluster 3 are characterized by the update of a method call (95.4%). The metrics for this cluster are provided in Table 18. The faults in this cluster are expected to be interface faults that involve the incorrect use of methods. Five faults in this cluster are manually inspected.

Table 18 - Cluster 3 Metrics

Cluster Id	3
Size	266
iSim	0.663
iSDev	0.136
eSim	0.118
eSDev	0.069

Descriptive Features	
Update Method Call	95.4%
Additional Functionality	1.1%
Update Variable Declaration	0.5%
Insert Method Call	0.5%
Insert Variable Declaration	0.4%

Two of the five faults that I manually inspect from this cluster meet my expectations for changes. The faults inspected from this cluster are described in Table 19. I discuss the problematic samples from this cluster below.

Table 19 - Faults Inspected for Cluster 3

Bug Id	Expected	Description
12449	No	In the fix for Bug 12449, one of the parameters was an anonymous class, and logic was changed in the anonymous class.
14742	Yes	The fix for Bug 14742 changes a parameter value from false to a value that is retrieved from the user's preferences.
20421	No	The fix for Bug 20421 also involved an anonymous class as a method parameter. In this case the logic checked a precondition and returned if it was not honored.
21824	No	The fix for Bug 21824 wraps a function call to display the busy indicator while the code executed.
23447	Yes	The updated method calls in Bug 23447 were primarily to resolve the direct access of member variables. Changing the code to use getter/setter methods simplified the logic and corrected the reported failure.

The most unexpected finding in this cluster is the impact of anonymous classes. Three of the five faults that I manually inspect in this cluster have methods updated where the argument is an anonymous class. The changes to the anonymous class are logic changes. An example is shown in Figure 11 from Bug # 20421. Similar to the anonymous class encountered in cluster 0, the true nature of the change is hidden. The addition of lines 77-81 check a precondition and return false if it is false. However, it occurs within the anonymous class that is passed to the accept method on line 68. Bug #12448 exhibits a similar problem with an anonymous class. Bug # 21824 is repaired by wrapping a method call in

```
68  event.getDelta().accept(new IResourceDeltaVisitor() {
69      public boolean visit(IResourceDelta delta) throws CoreException {
70          IResource resource = delta.getResource();
71
72          if(resource.getType()==IResource.ROOT) {
73              // continue with the delta
74              return true;
75          }
76
77          if (resource.getType() == IResource.PROJECT) {
78              // If the project is not accessible, don't process it
79              if (!resource.isAccessible()) return false;
80          }
81
82          String name = resource.getName();
83          int kind = delta.getKind();
84          IResource[] toBeNotified = new IResource[0];
```

Figure 11 – Bug 20421: Additional condition check

an anonymous class.

Cluster 4 – Insert If and Return Statements

Cluster 4 is the first cluster with two dominant descriptive features. The addition of a *return* statement explains 47.3% of the similarity and the addition of an *if* statement explains 36.4% of the similarity. The metrics for this cluster are presented in Table 20. I expect simple faults in this cluster to be checking faults. More complex faults with multiple instances of *if* statements and/or multiple instances of *return* statements may represent more complex logic faults.

Table 20 - Cluster 4 Metrics

Cluster Id	4
Size	164
iSim	0.580
iSDev	0.084
eSim	0.212
eSDev	0.073

Descriptive Features	
Insert Return	47.3%
Insert If Statement	36.4%
Delete Return	5.0%
Insert Variable Declaration	3.5%
Insert Method Call	1.7%

Five faults are manually inspected in this cluster and all of them meet expectations. The faults inspected from this cluster are described in Table 21. Two of the five were checking faults. Two of the fixes were minor logic changes. Bug 14061 had extensive changes to the program logic.

Table 21 - Faults Inspected for Cluster 4

Bug Id	Expected	Description
12210	Yes	The fix for Bug 12210 was an update to code that uses the Visitor design pattern [84]. When a node is visited, the class must determine if a simple name or a variable declaration is being visited and act appropriately.
12590	Yes	Bug 12590 appears to be a checking fault. The author added a check to see if the selected item was a local variable when the *rename* function was invoked.
13417	Yes	Bug 13417 was fixed by adding a check for blank text on a tooltip.
14061	Yes	Bug 14061 was a complex logic fault that resulted in duplicate menu items when the *SubContributionItem* class is used. In addition to the logic changes, new functionality was also added.
18274	Yes	Bug 18274 is related to Bug 14061. In the fix for Bug 18274, a check was added for this type and an *unwrap* method was called when it was encountered.

Cluster 5 – Insert If Statement and Statement Parent Change

The faults in Cluster 5 are characterized by a *statement parent change* (63.1%) and the insertion of one or more *if* statements (22.7%). The cluster metrics are provided in Table 22. Similar to Cluster 4, I expect logic faults that range from checking faults to more complex logic faults. I manually inspect five faults in this cluster.

Table 22 - Cluster 5 Metrics

Cluster Id	5	**Descriptive Features**	
Size	446	Statement Parent Change	63.1%
iSim	0.570	Insert If Statement	22.7%
iSDev	0.093	Delete If Statement	2.0%
eSim	0.203	Insert Method Call	1.9%
eSDev	0.065	Insert Variable Declaration	1.5%

Bug 14025 is the only fault in this cluster that does not meet my expectations. The change requires logic changes, but includes new functionality as well. The faults inspected from this cluster are described in Table 23 below.

Table 23 - Faults Inspected for Cluster 5

Bug Id	Expected	Description
13024	Yes	Bug 13024 changed the code to account for blank text for a tooltip. The changes were complex because different implementations were necessary for each operating system.
14025	No	Bug 14025 required a new instruction set in the abstract syntax tree to deal with the length member variable on arrays.
17176	Yes	The fix for Bug 17176 reordered logic in one method. The reordering was recorded as a deletion and insertion of the *if* statements, but as a statement parent change for the code in the statement block. Although this was unexpected based on the change types, the fault was a logic fault due to order of checks
18468	Yes	Bug 18486 was mislabeled in the CVS repository. That commit was actually for Bug 18468. The fault repaired was a checking fault. Under certain conditions the view needed to be refreshed.
19985	Yes (see Note)	Bug 19985 was fixed by changing the way the end of a line was written. Improvements to the code were made along with the change in logic. The *if* statement inserts appear to be somewhat misleading, since the *if* statement was moved and the condition expression was changed.

Cluster 6 – Insert Method Call

The similarity of faults in Cluster 6 is explained primarily through the insertion of method calls (78.5%). A small part of the similarity is explained due to the addition of methods (6.7%). The cluster metrics are provided in Table 24. I expect this cluster to contain faults due to missing functionality and misuse of methods. Seven faults from this cluster were manually inspected.

Table 24 - Cluster 6 Metrics

Cluster Id	6
Size	434
iSim	0.566
iSDev	0.091
eSim	0.208
eSDev	0.066

Descriptive Features	
Insert Method Call	78.5%
Additional Functionality	6.7%
Insert Variable Declaration	2.9%
Additional State	2.6%
Insert Assignment	2.4%

The faults inspected from this cluster are described in Table 25. Three of the faults address missing functionality (10823, 11308, and 18067). Three of the

faults are interface faults (17490, 17981, and 21654). The fix for Bug 16160 repairs a dependency problem and is unexpected in this cluster.

Table 25 - Faults Inspected for Cluster 6

Bug Id	Expected	Description
10823	Yes	The fix for fault 10823 requires changes to four classes and the addition of two new "Action" classes. The fault is a change of functionality to support advanced users. The Action classes follow the Command design pattern [84].
11308	Yes	The fix for Bug 11308 changed the project to use relative paths to allow project portability.
16160	No	The fix for 16160 repaired a dependency problem in the *CVSUIPlugin* class.
17490	Yes	The fix for Bug 17490 added method calls to enable context-sensitive help.
17981	Yes	The fix for Bug 17981 added method calls to enable shortcut keys (mneumonics).
18067	Yes	The fix for Bug 18067 was a change in behavior that included refreshing the viewer under certain conditions.
21654	Yes	Bug 21654 was a GTK specific issue and was repaired by adding a GTK specific method call.

Cluster 7 – Delete Method Call

The faults in Cluster 7 are explained by the removal of method calls (56.6%) and partially explained by the insertion of new method calls (16.2%). The metrics appear in Table 26. I expect the faults in this cluster to include the removal of extraneous code and moving method calls to new locations. Since the changes imply restructuring of the code, functional defects and refactoring may also be present in these faults.

Table 26 - Cluster 7 Metrics

Cluster Id	7	**Descriptive Features**	
Size	279	Delete Method Call	56.6%
iSim	0.525	Insert Method Call	16.2%
iSDev	0.090	Delete Variable Declaration	6.9%
eSim	0.207	Delete If Statement	4.5%
eSDev	0.084	Additional Functionality	4.3%

The faults inspected from this cluster are described in Table 27. Three of the five fall into the category of extraneous method calls or functionality (14800,

16051, and 16445). The other two fixes in this cluster involve extensive changes to current program flow, and include refactoring.

Table 27 - Faults Inspected for Cluster 7

Bug Id	Expected	Description
14197	No	The fix for Bug 14197 was a significant change in existing functionality and included code refactoring.
14288	No	The fix for Bug 14288 made fundamental changes to the way that the search functions. These changes included removal of some functions and the insertion of others. This could be considered an algorithmic or functional fault.
14800	Yes	The fix for Bug 14800 removed method calls to fix the behavior.
16051	Yes	The fix for Bug 16051 removed method calls to fix the behavior.
16445	Yes	Bug 16445 repaired a functional defect where information was requested from the user that was not necessary.

Cluster 8 – Insert If, Variable Declaration, Method Call, and Assignment

The faults in Cluster 8 are explained by the insertion of *if statements* (40.3%), *variable declarations* (19.5%), *method calls* (11.1%), and *assignment statements* (9.0%). The metrics are provided in Table 28. Given the nature of these changes, the faults in this cluster are expected to be algorithmic or functional changes to behavior.

Table 28 - Cluster 8 Metrics

Cluster Id	8	**Descriptive Features**	
Size	554	Insert If Statement	40.3%
iSim	0.504	Insert Variable Declaration	19.5%
iSDev	0.082	Insert Method Call	11.1%
eSim	0.246	Insert Assignment	9.0%
eSDev	0.059	Delete If Statement	5.0%

Seven faults in this cluster are manually inspected. The faults inspected from this cluster are described in Table 29. Five of the faults manually inspected fall into this broad category of changes. Bug 15506 is fixed by adding a busy indicator. The CVS commit for Bug 19270 included changes for another bug, which makes automated analysis challenging.

Table 29 - Faults Inspected for Cluster 8

Bug Id	Expected	Description
10714	Yes	The fix for Bug 10714 corrected behavior when a view was closed. The software was not always properly setting focus to the last view that was active.
14614	Yes	Bug 14614 was an issue with the way that CVS tag decorators were displayed that resulted in duplicate tags. The fix was an update to the algorithm.
15506	No	The fix for Bug 15506 wraps the code in a *Runnable* class to show the busy indicator. This required code to store results and handle exceptions, then communicate these to the main program.
15755	Yes	Bug 15755 was repaired by changing the initial search location and the precedence of additional locations.
19270	No	The fix for Bug 19270 was checked in with the fix for Bug 6295. Bug 19270 appears to be a checking fault that required new code to retrieve a user preference for comparison. The fix for Bug 6295 corrected a problem where the *save as* option resulted in a read-only file.
22448	Yes	Bug 22448 was corrected by changing the algorithm to handle an edge case where the first button in the second row of a toolbar caused a screen resize.
24134	Yes	The fix for Bug 24134 changed the way that compile was invoked.

Cluster 9 – Garbage Collector

As mentioned previously, the last cluster acts as a "garbage collector" when the I1 criterion function is used. The metrics and descriptive features are provided in Table 30. The variation in change types and the scores for each descriptive feature support previous findings about the nature of the last cluster when I1 is used as the criterion function [79].

Table 30 - Cluster 9 Metrics

Cluster Id	9
Size	376
iSim	0.084
iSDev	0.057
eSim	0.083
eSDev	0.081

Descriptive Features	
Update Assignment	24.6%
Insert Assignment	12.7%
Delete Variable Declaration	8.0%
Update Return	6.6%
Remove Functionality	6.5%

I expect this cluster to have varied faults that are uncommon or simple faults obfuscated by implementation details. These may represent a set of faults

for which automated classification is not possible or warranted due to their infrequent nature. Eight faults from this cluster are manually inspected. The faults inspected from this cluster are described in Table 31.

Table 31 - Faults Inspected for Cluster 9

Bug Id	Expected	Description
10144	n/a	Bug 10144 called for the promotion of *org.eclipse.ui.views.framelist* to a public API. The change includes the check-in of the files in their new location and updates to use the new namespace.
11474	n/a	The fix for Bug 11474 changed the way that an error condition is checked. The method that was previously used was deleted from the class and the error message was changed.
12996	n/a	Bug 12996 is a concurrency fault. The changes to correct the fault included the deprecation of old methods and changes to the parent class.
13470	n/a	The fix for Bug 13470 adds methods to externalize (and thus translate) string values.
13625	n/a	Bug 13625 is fixed by removing deprecated functions.
15583	n/a	The fix for Bug 15583 changes a literal value to correct a missing mnemonic in a menu item. This fault is interesting because the true nature of the fault is obfuscated because it is a change within a variable declaration.
16027	n/a	The fix for Bug 16027 required a large number of files to be changed. The changes included the removal of a number of getter methods and the update to method parameters. The latter changes were obfuscated because the method calls were part of a return statement.
20430	n/a	Bug 20430 was changed by updating a single assignment that set the minimum width.

There was no discernible pattern to these changes. Some of the changes were large, while others were small and infrequent. It is important to note that the fix for Bug 16027 includes some changes that were hidden because they were part of a return statement.

5.4.4 Discussion

The manual inspection resulted in mixed results for 2 clusters, but many of the clusters provide promising results. A summary of agreement and disagreement is given in Figure 12. Cluster 0 (Update Variable Declaration) and Cluster 3 (Update Method Call) had poor results.

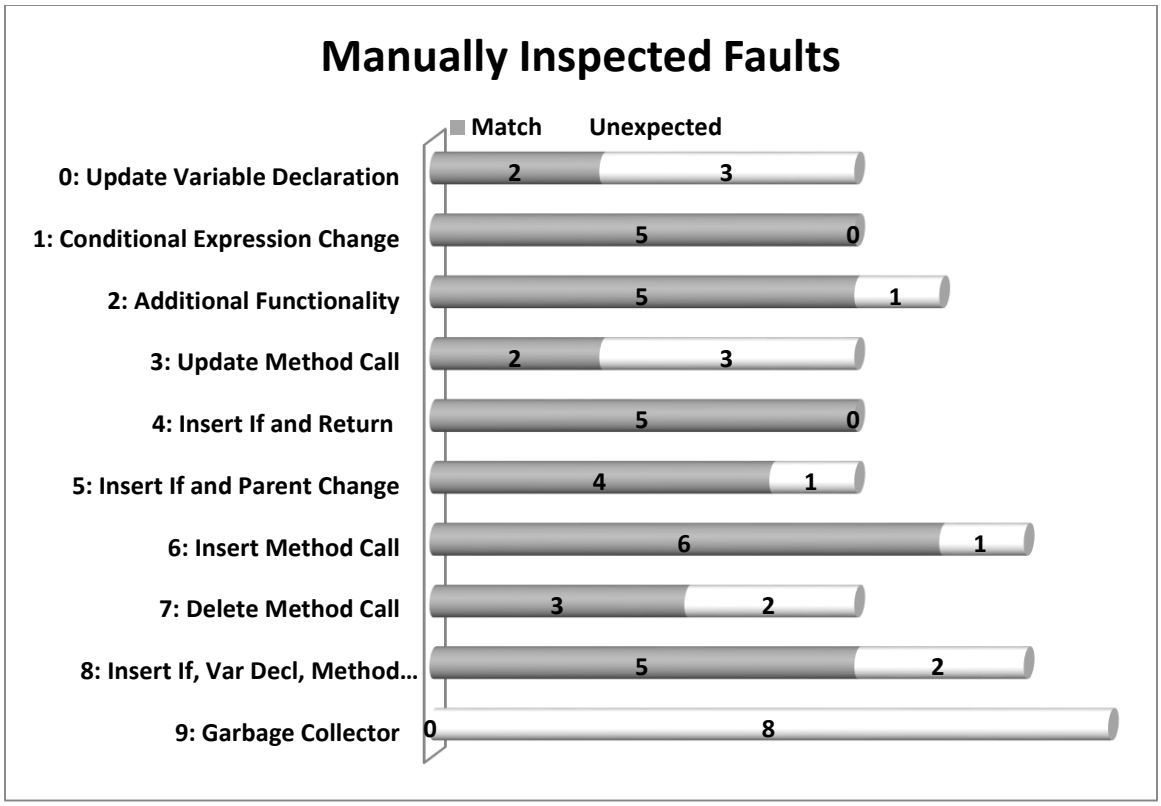

Figure 12 - Summary of Manual Inspection Results

In most fault classification studies where the agreement of two classifiers are studied, Cohen's Kappa is calculated to determine the level of agreement between classifiers. If I disregard the faults in the garbage collector and calculate Cohen's Kappa for these results, I find $\kappa=0.717$. According to the scale presented by Landis and Koch, 0.717 represents Good agreement [85]. Thus, when uncommon faults are not considered, these results may be comparable to that of human fault classifiers [12], [13], [57].

There are many difficulties in extracting useful information from the syntactical changes. For the faults that I inspected I saw changes such as variable renaming and refactoring. These changes introduce noise into the syntactical changes that are used to cluster faults. Similarly, many commits to the software repository will address multiple faults. These changes cannot easily be separated. These types of problems can be mitigated by disciplined check-in procedures. Research has been done on non-essential changes, such as renaming of variables, which may be applicable to this problem [86]. There have

also been studies on identifying refactored source code from changes [87], and determining whether a problem report should be classified as a fault or an enhancement from the text [22]. It may be possible to apply these techniques to improve results.

However, the most significant problem that I can address based on my manual inspection results is the way that the ChangeDistiller tool handles anonymous classes. This problem may be exacerbated by the Eclipse architecture. Anonymous classes are commonly used as event handlers, and the component-based architecture of Eclipse relies heavily upon event handlers. In the next section I address the problem of anonymous classes in variable declarations, assignments, method calls, and return statements.

5.5 Improving ChangeDistiller for Anonymous Classes

In this section I describe updates to the ChangeDistiller application that handle changes that occur within anonymous classes. I have made these changes publicly available[10].

It is interesting to note that the Change Distilling algorithm does not specify a stopping point for comparison [65]. The ChangeDistiller implementation extracts the changes to the granularity required for the change taxonomy defined by Fluri and Gall [66]. The changes described in this section have to detect that an element is an anonymous class and change the behavior of the program appropriately to properly classify the changes.

The ChangeDistiller tool uses the Visitor design pattern [84]. Each abstract syntax tree node is visited as the tree is traversed. The visit function for each node accepts a visitor class. The JDT API defines an *ASTVisitor* class and this is used as the basis of the ChangeDistiller algorithm. Returning true from the visitor results in a traversal of the child nodes, while returning false does not.

Anonymous classes are contained in an instance of a *QualifiedAllocationExpression* in the JDT API. I modified the *visit* method for this type to traverse the children of the element. I also modified the visit method's

[10] https://bitbucket.org/bill_kidwell/tools-changedistiller

local declarations, method calls, and return statements. The changes result in traversal of statements within anonymous class methods.

My changes have been effective for every case that I found during this research and my testing. However, there are limitations. The changes are not designed to deal with any changes within the qualified allocation expression except for statement level changes. I did not test structural changes, such as the addition of methods. I did not see any of these changes during my inspection of fault changes for Eclipse.

5.5.1 Updated Clustering Results

The results of clustering after the changes to ChangeDistiller produced a similar set of clusters. The metrics from these results are presented in Table 32. The tightest cluster has an internal similarity of 0.709 (compared to 0.789) and the garbage collector cluster has an internal similarity of 0.147, an improvement over the previous result of 0.084.

Table 32 - Updated Clustering Results for Eclipse 2.0

Cluster Id	Size	iSim	iSDev	eSim	eSDev
0	167	0.709	0.125	0.115	0.068
1	116	0.702	0.126	0.109	0.079
2	212	0.654	0.118	0.136	0.079
3	211	0.651	0.087	0.178	0.093
4	323	0.653	0.093	0.222	0.060
5	480	0.596	0.082	0.215	0.068
6	233	0.593	0.065	0.289	0.064
7	282	0.461	0.090	0.187	0.088
8	526	0.510	0.083	0.242	0.071
9	260	0.147	0.085	0.091	0.074

The differences in the two results begin to become apparent when I investigate the descriptive features of the clusters. In Table 33 the descriptive features for the clusters are displayed side-by-side. The *Condition Expression Change* Cluster moved up one position in rank. The slight reduction in the similarity of the *Update Variable Declaration* cluster is likely due to the fact that variable declarations with changes in anonymous classes now have a different

set of feature values. It is also interesting to note that the size of Cluster 4 has increased dramatically.

Table 33 - Descriptive Feature Comparison

Original Results				Updated Results			
Rank	Size	Descriptive Features	Perc.	Rank	Size	Descriptive Features	Perc.
1	139	Condition Expression Change	94.70%	0	167	Condition Expression Change	96.20%
0	94	Update Var. Declaration	98.50%	1	116	Update Variable Declaration	96.30%
3	266	Update Method Call	95.40%	2	212	Update Method Call	93.60%
2	132	Additional Functionality	95.20%	3	211	Additional Functionality	85.20%
6	434	Insert Method Call	78.50%	4	323	Insert Method Call	89.30%
5	446	St. Parent Change	63.10%	5	480	St. Parent Change	67.90%
		Insert If	22.70%			Insert If	17.90%
7	279	Insert Method Call	16.20%	6	233	Insert Method Call	44.20%
		Delete Method Call	56.60%			Delete Method Call	28.40%
				7	282	Delete Method Call	40.40%
						Delete Var Declaration	22.60%
						Delete If Statement	13.40%
8	554	Insert If	40.30%	8	526	Insert If	38.30%
		Insert Var. Declaration	19.50%			Insert Var. Declaration	27.90%
		Insert Method Call	11.10%				
9	376	Update Assignment	24.60%	9	260	Update Assignment	42.00%
		Insert Assignment	12.70%			Insert Assignment	26.70%
4	279	Insert If	36.40%				
		Insert Return	47.30%				

The next change of interest is the change in Cluster 5. The percentage of contributions from the *Insert If* statement has dropped, while *Statement Parent Change* has grown. This change is likely due to changes in the membership of this cluster. The size has only changed from 446 to 452. Cluster 6 also exhibits large changes in the significance of the descriptive features, as well as a drastic change in size.

Cluster 7 changed dramatically between the two versions. The logic errors from the original Cluster 7 are likely to be in a different cluster. Cluster 8 and Cluster 9 appear to be similar, except for a decrease in size in Cluster 9. In the

next section I revisit the manual inspections. I analyze faults that have stayed in the same clusters as well as faults that have changed clusters.

5.5.2 Manual Inspection of Changes

After updating the ChangeDistiller code and repeating the clustering process, I pulled the new cluster results for the manually inspected data. Thirty-six of the fifty-eight faults are in the *equivalent cluster* in the new results. Nine additional faults changed membership to a cluster with *similar descriptive features*. The remaining thirteen faults changed membership to new clusters. In this section we analyze faults in each of these categories.

Equivalent Clusters

The majority of faults that remain in an equivalent cluster meet the expectations set for that cluster. This includes five faults that remain in the garbage collector cluster. Three faults remain in an equivalent cluster and do not meet expectations. Bug 18923 remains in the **Update Variable Declaration** cluster. As mentioned during the manual inspection, this fault includes variable name changes that had no impact on behavior. Bug 23824 also remains in the **Update Variable Declaration** cluster. Bug 23824 involves an incorrect cast. Bug 15506 remains in the **Insert If + Insert Var Decl** cluster. In Bug 15506 existing code is wrapped in an anonymous class instance.

The large number of faults assigned to similar clusters provides some evidence of stability. None of the faults from **Condition Expression Change** cluster or the **Additional Functionality** cluster change membership. Fault fixes that require complex changes, or that include refactoring continue to be difficult to cluster correctly. Some small changes, such as the casting problem and adding an anonymous class to wrap existing functionality, also present challenges.

Similar Descriptive Features

It is worth noting that the **Insert If + Insert Return** cluster does not exist for the updated clustering results. It is also interesting that the cluster that appears in the new solution is very different (**Delete Method Call + Var Decl + If Statement** cluster). One possible explanation is that the **Insert If + Insert Return** cluster is currently a subcluster, and will emerge if *k* is increased. This might also lead to an expectation that all of the faults from this cluster are currently in a different cluster, but this is not the case.

Three of the faults from the original **Insert If + Return** cluster have changed membership to the **Insert If + Insert Variable Declaration** cluster. Bugs 12590, 13417, and 18274 are checking faults, and the fixes appear to be simple in the syntactic sense. This supports the idea of a subcluster within the **Insert If + Insert Variable Declaration** cluster.

If a subcluster exists, faults that are more complex do not necessarily reside within the subcluster. Bug 12210 and 14061 changed membership to the **Delete Method Call + Var Decl + If** cluster. Both of these faults involved more extensive logic changes than the others. I increased *k* until a cluster emerged with the *Insert If Statement* and *Insert Return Statement* as the dominant descriptive features. The cluster emerged at $k = 13$. Four of the five faults were in this cluster, but Bug 12210 remained in the **Delete Method Call + Var Decl + If** cluster.

The **Insert Method Call** cluster and the **Insert Method Call + Delete Method Call** cluster also had several membership changes. This is seen in the changes to sizes and the feature contributions. Bug 16160 is not expected in the **Insert Method Call** cluster. The fault moved to the **Insert Method Call + Delete Method Call** cluster. The fix is a structural change to avoid referencing an internal class directly. The fault does not belong in the new cluster either. Bug 11308 is a change in behavior. It moves to the **Insert If + Insert Variable Declaration** cluster. Complex faults in this cluster are expected to be complex logic changes or complex changes to behavior, so it belongs in the new cluster.

Bug 10823 is a similarly complex change to logic and behavior that moves to the **Insert Method Call + Delete Method Call** cluster. The fault meets the expectations of this cluster.

Bug 14197 moves from the **Delete Method Call** cluster to the **Insert Method Call + Delete Method Call** cluster. As noted above, Bug 14197 is a significant change in functionality, so it meets the expectations of this cluster, where it did not meet the expectations of the **Delete Method Call** cluster. The fix for 14197 includes refactoring that makes it difficult to characterize via its syntax. Bug 16445 also moved from the **Delete Method Call** cluster, but moved to the **Delete Method Call + Var Decl + If** cluster, where it meets the expectations of that cluster.

Bug 19270 contains multiple fault fixes (includes Bug 6296). In addition to the fact that two fixes are included, a number of statements are removed during the restructuring of the files to fix the problems. The fault moved from the **Insert If + Var Decl + Method Call** cluster to the **Delete Method Call + Var Decl + If** cluster. The fault does not belong in either cluster.

Based on the analysis of the faults in this category, it seems apparent that larger values for *k* could provide better results in some cases. It may be difficult to identify a value that provides the fine-grained patterns that we seek and makes the clusters meaningful to practitioners. In addition, many changes in this category were complex. Some of the difficulty in clustering complex faults may be due to the removal of code.

New Clusters

The aim of the changes to ChangeDistiller was to avoid problems identified with anonymous classes. Anonymous classes affect three of the manually inspected faults. Bug 11110 moved to the **Condition Expression Change** cluster, where it is an expected member. Bug 12449 involves the addition of code to handle the delete action when the delete key is pressed. This fault moved to the **Insert Method Call** cluster, where it is an expected member. Bug 20421 involves logic changes that are obfuscated by an anonymous class,

which is passed as a parameter. This fault moved to the **Statement Parent Change + Insert If** cluster, where it is an expected member of the group. All three incidents that involve changes in anonymous classes are in correct clusters.

Additional faults from manual inspection that changed clusters appear below in Table 34. The Kappa statistic for these results improved slightly to κ = 0.735.

Table 34 - Additional Manual Inspection for New Results

Bug	Original Cluster	New Cluster	Expected
21824	Update Method Call	Insert Method Call	Yes
14025	Insert If + Stmt Parent Change	Update Method Call	No
21654	Insert Method Call	Update Var Decl	No
14288	Delete Method Call	Condition Expr Change	No
10144	Garbage Collector	Additional Functionality	Yes
11474	Garbage Collector	Condition Expr Change	Yes
12996	Garbage Collector	Delete Method Call + Var Decl + If	No

5.5.3 Discussion

The changes to the ChangeDistiller program did improve the clustering of faults with anonymous classes, but overall made only incremental improvement. I take this as a positive sign that additional changes could make further improvements. Some code check-ins contain multiple fault fixes, refactoring, or changes to variable names. These fault fixes will be difficult to classify in an automated manner.

5.6 Conclusions

In order to further validate the extended change types introduced in Chapter 1 the CLUTO clustering toolkit is used to cluster the fault fixes. Using the

repeated bisection clustering method and the cosine similarity, the I1 criterion function performs better than the I2 criterion function with respect to the average internal similarity of the clusters in the resulting solution. The ability of I1 to create tight clusters and one cluster that acts as a “garbage collector” in a noisy data set aids the investigation [79].

The results of clustering where *k*=10 are analyzed. The similarity of the cluster is explained by one to four features that are shared by the faults in the cluster. These descriptive features are used to automatically label the cluster. The clusters for Eclipse 2.0 and 3.0 and their sizes were compared. The occurrence and size of the clusters were correlated, indicating that the clustering of these change types is consistent in these two versions of the software.

A subjective analysis of a subset of faults in each cluster provides guidance on the types of faults characterized by different source code change types. Many fault fixes are in agreement with our expectations based on the syntactical changes that were made to the fault. For example, faults fixed with changes to condition expressions that are inspected in this study are in line with expectations.

Several of the faults that were inspected exposed limitations in the taxonomy. ChangeDistiller stops the comparison of the abstract syntax trees at the statement level due to its intent in analyzing change couplings. As a result, update changes to variable declarations, assignments, or return statements do not provide the granularity necessary for fault analysis. There were a surprising number of problems with anonymous classes as method parameters, and within variable declarations, that also require more granular information about the change. These findings indicate that the comparison must be extended beyond differences in statements, to differences in arguments and expressions.

The ChangeDistiller program was updated to handle the common problems that we saw with anonymous classes. The data was collected with the updated program and the clustering process was repeated. The faults that involved anonymous classes were now in the expected clusters, but other problems emerged. The results seem to indicate that more clusters are

necessary for useful results. The number of clusters will involve a trade-off between the precision of the patterns in the groups, and the usefulness of the clusters to practitioners. In addition, the number of faults that changed membership due to deleted statements is significant. Weighting deleted statements might provide a method to improve these results further.

I encountered a number of common software repository mining problems during the manual inspection. Code refactoring that is included in a commit for a bug fix can make automated analysis difficult. A simple change, such as renaming a variable for readability, should be handled at the semantic level of analysis. More complex refactoring changes will still make automated analysis difficult. Developers sometimes include multiple bug fixes in a single commit, as evidenced by Bug #19270. Bug #18468 was mislabeled as Bug #18486, which can be problematic when bug database information is cross-referenced with the syntactical changes.

I conclude that the current taxonomy provides a useful start for the automated analysis of software faults. Incremental improvements are necessary, and based on the improvements reported above, can measurably improve the effectiveness of the method. In the next chapter we utilize the improved version of the ChangeDistiller tool to investigate the distribution of faults across several versions of an open source software project.

Chapter 6

Software Fault Evolution

In this chapter I analyze the evolution of software faults over multiple releases for a major component of the Eclipse product line. The Eclipse Java development tools (JDT) project is analyzed over seven versions of its release. I investigate a number of questions about the evolution of software faults that are made possible by automated fault classification. These questions include an investigation of fault distribution by subcomponent, between single and multi-file fixes, among developers that fixed the faults, among pre-release and post-release fault fixes, and of fixes that appear problematic.

The study can be described using the Goal/Question/Metric (GQM) template for goal definition [82][83].

> *Analyze the **distribution of software faults***
> *for the purpose of **understanding software evolution***
> *with respect to the **consistency of the distributions***
> *from the point of view of the **researcher***
> *in the context of **an open source Java development environment**.*

6.1 Case Study

An overview of the Eclipse JDT is available on the Eclipse.org website [88]. The project provides a full-featured Java IDE built on the Eclipse platform. The site describes five JDT plug-ins, the plug-ins are summarized here. The JDT APT (Annotation Processing Tools) adds annotation support, which was introduced in Java 5 (1.5). The JDT Core provides APIs for building Java applications, navigating Java elements (e.g., packages, classes, methods, and fields), code assist, and refactoring. The JDT Debug plug-in provides debugging support. The JDT Text plug-in provides a full featured Java editor with syntax coloring, code assist, code formatting, and other common source code editor features.

The Eclipse project coordinates releases for multiple projects, such as the Eclipse Platform and the Eclipse JDT, at the same time. The release dates for the versions that I investigate are shown in Table 35. Faults that are fixed between the Start date and the Release date are considered pre-release fault fixes. Faults that are fixed between the Release date and the End date are considered post-release fault fixes. Eclipse also schedules service releases for each version after the Release date. The timing of the service releases is not considered in this study.

Table 35 - Eclipse Release Timelines

Version	Start	Release	End
2.0	1/1/2002	6/7/2002	9/29/2002
2.1	9/30/2002	3/28/2003	9/26/2003
3.0	12/1/2003	6/21/2004	12/30/2004
Europa (3.3)	1/1/2007	6/29/2007	12/31/2007
Ganymede (3.4)	1/1/2008	6/25/2008	12/31/2008
Galileo (3.5)	1/1/2009	6/24/2009	12/31/2009
Helios (3.6)	1/1/2010	6/23/2010	12/31/2010

In this study I am investigating the Eclipse JDT project as a component of the Eclipse product line. I look at the subcomponents of the JDT based on the Java packages. The subcomponents are listed in Table 36 with the number of fault fixes that included source code changes for each version. The total in the right-most column indicates the number of faults in the subcomponent across all studied versions. The Version total row at the bottom of the table presents the total number of faults across all subcomponents for the given version of Eclipse.

Table 36 - Fault Fixes for Eclipse JDT Subcomponents by Version

Subcomponent	2.0	2.1	3.0	3.3	3.4	3.5	3.6	Total
org.eclipse.jdt.ui	639	815	842	566	425	238	235	3760
org.eclipse.jdt.core	184	444	684	458	407	300	277	2754
org.eclipse.jdt.debug.ui	341	196	222	138	49	40	27	1013
org.eclipse.jdt.debug	234	98	98	40	19	27	15	531
org.eclipse.jdt.launching	97	81	55	29	20	9	12	303
org.eclipse.jdt.junit	12	45	57	34	12	17	8	185
org.eclipse.ltk.ui.refactoring			9	18	8	7	5	47
org.eclipse.jdt.apt.core				31	9	2	1	43
org.eclipse.ltk.core.refactoring			12	13	6	1	7	39
org.eclipse.jdt.compiler.apt				10	7	8	3	28
org.eclipse.jdt.compiler.tool				8	3	3	7	21
org.eclipse.jdt.apt.pluggable.core				8	4	3	1	16
org.eclipse.jdt.core.manipulation				10				10
org.eclipse.jdt.junit.runtime			5	1			3	9
org.eclipse.jdt.junit4.runtime				2		2	2	6
org.eclipse.jdt.apt.ui				3				3
Version Total	**1507**	**1679**	**1984**	**1369**	**969**	**657**	**603**	**8768**

It is interesting to note that the top 4 subcomponents account for more than 90% of the fault fixes over the seven releases. It is also evident from the Version Total row that the fault fix count for the first three releases is trending up, while the fault fix count for the last four releases is trending down. Most likely this is due to the maturation of the product and the process. The trend is depicted in Figure 13.

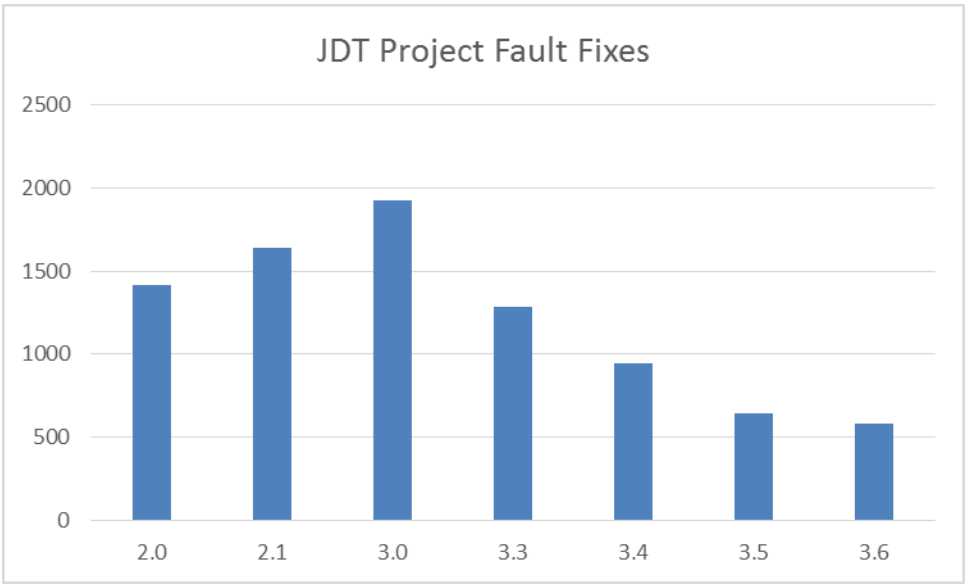

Figure 13 - JDT Project Fault Fixes by Version

6.2 Data Collection

In this section I describe the data collection for this study. The database that was published by Krishnan et al. was again used as the basis for my data collection [74]. The source code that was used for collecting the change type frequencies was no longer available in a public CVS repository. The Eclipse project migrated to the use of Git, a distributed revision control system. In this section I describe how I altered MiSFIT to support the use of Git.

6.2.1 Git Data Collection Changes

The first step of the migration is to match CVS file and revision numbers to Git commits and files. Each commit in a Git repository has an identifier, and may contain multiple files. CVS, on the other hand, tracks changes for each file separately, even if changes occur at the same time. When converting from CVS the Eclipse project chose to maintain historical information. The tools used to convert the repository combine files checked in simultaneously into individual commits.

I use the Eclipse EGit project as an interface to the Git repository[11]. For each fault fix, I query the file name, author name, commit date, and commit comment from the database. Using this information, I was able to query the Git repository and retrieve commit and file information for 92.3% of the changes (1813/22889 could not be retrieved).

All but 15 of the unidentified files are part of a feature branch in CVS. A feature branch occurs when the code is isolated from other developers in order to get a feature working, then merged back into the mainline branch for testing and release. The other 15 files were manually investigated and are not available in the repository.

Because these seven versions occurred over nearly a decade, it was necessary to adjust my process to handle new constructs in the Java programming language. Eclipse 2.0, 2.1, and 3.0 are parsed and examined using version 1.4 of the Java Development Kit (JDK). Eclipse 3.3 is parsed and examined using version 1.5 of the JDK. The remaining versions are parsed and examined using version 1.6 of the JDK.

I also found and corrected a number of issues in the database. The removal of special characters (e.g., apostrophe (‘) and backslash (\)) caused issues when matching information by description. I altered the Perl script provided by Krishanan et al. [74] to maintain these characters and improve the matching.

I found multiple problems with incorrectly identified Bug Ids in the database. In CVS, the Bug Id is entered as free form text in the comment. Multiple conventions are used. I found multiple instances where other numbers in the comments caused problems. For example, the comment “Fixed bug 187226: Compiler warning in I20070516-0010” resulted in two records, one for 187226 and one for 0010. I constructed a query to identify similar problems and removed the erroneous entry. I also found problems where build numbers in the form of dates cause problems. The entry “JRT 20020305” was logged as Bug Id 200203.

[11] http://eclipse.org/egit/

I investigated all entries with identifiers that matched dates in the YYYYMMDD format and removed those that were errors.

In this section I have described modifications to the MiSFIT system in order to collect data from the Eclipse JDT Git repositories. I utilized the EGit project to interface with the Git repositories and fetch files as they were needed. I also used the EGit project to mine information about the commits and expand my database. Other steps in the data collection process were changed minimally.

6.2.2 JDT Clustering Results

The resulting clusters for 8096 fault fixes that were processed for seven versions of the Eclipse JDT project are described in Figure 14 and illustrated in Figure 15. The clusters are similar to those in Eclipse 2.0 and Eclipse 3.0. The expectations for these clusters are as follows:

0. Logic faults involving condition expressions
1. Interface faults, likely involving incorrect parameters or calling the incorrect version of a method
2. Faults that involve missing functionality
3. Interface faults or missing functionality
4. Logic faults involving a failure to check necessary conditions
5. Incorrectly initialized variables or incorrect assignments

Eclipse JDT: #Rows: 8096, #Columns: 128, #NonZeros: 1036288

Cluster 0, Size: 624, ISim: 0.779, ESim: 0.135

Descriptive: **COND_EXPR_CHG 97.5%**, INS_VAR_DECL 0.8%, STATEMENT_PARENT_CHANGE 0.6%, INS_IF 0.3%, UPD_VAR_DECL 0.2%

Cluster 1, Size: 512, ISim: 0.653, ESim: 0.126

Descriptive: **UPD_METHOD_CALL 93.0%**, INS_VAR_DECL 1.2%, UPD_VAR_DECL 1.1%, INS_METH_CALL 0.7%, ADD_FUNC 0.6%

Cluster 2, Size: 525, ISim: 0.597, ESim: 0.141

Descriptive: **ADD_FUNC 88.3%**, INS_METH_CALL 3.3%, ADD_STATE 1.8%, COND_EXPR_CHG 1.2%, INS_IF 1.0%

Cluster 3, Size: 840, ISim: 0.606, ESim: 0.179

Descriptive: **INS_METH_CALL 87.3%**, INS_IF 3.1%, INS_VAR_DECL 2.2%, DEL_METH_CALL 2.1%, ADD_STATE 0.9%

Cluster 4, Size: 1461, ISim: 0.635, ESim: 0.214

Descriptive: **STATEMENT_PARENT_CHANGE 74.7%**, **INS_IF 11.6%**, COND_EXPR_CHG 5.4%, INS_VAR_DECL 1.3%, DEL_IF 1.2%

Cluster 5, Size: 499, ISim: 0.487, ESim: 0.105

Descriptive: **UPD_VAR_DECL 80.6%**, **UPD_ASSIGN 14.0%**, INS_VAR_DECL 1.5%, COND_EXPR_CHG 1.0%, DEL_VAR_DECL 0.4%

Cluster 6, Size: 707, ISim: 0.557, ESim: 0.208

Descriptive: **INS_RETURN 43.2%**, **INS_IF 40.3%**, STATEMENT_PARENT_CHANGE 4.2%, INS_VAR_DECL 3.2%, DEL_RETURN 3.2%

Cluster 7, Size: 1246, ISim: 0.555, ESim: 0.246

Descriptive: **INS_VAR_DECL 39.1%**, **INS_IF 24.5%**, **INS_ASSIGN 16.1%**, INS_METH_CALL 5.8%, STATEMENT_PARENT_CHANGE 4.7%

Cluster 8, Size: 976, ISim: 0.450, ESim: 0.178

Descriptive: **DEL_METH_CALL 33.0%**, **DEL_VAR_DECL 24.1%**, **DEL_IF 14.2%**, DEL_ASSIGN 4.9%, INS_METH_CALL 3.9%

Cluster 9, Size: 706, ISim: 0.146, ESim: 0.086

Descriptive: **INS ASSIGN 42.8%**, **UPD RETURN 25.1%**, REMOVE FUNC 5.1%, DEL ASSIGN 4.9%, ADD STATE 2.9%

Figure 14 - Fault Clusters for Eclipse JDT

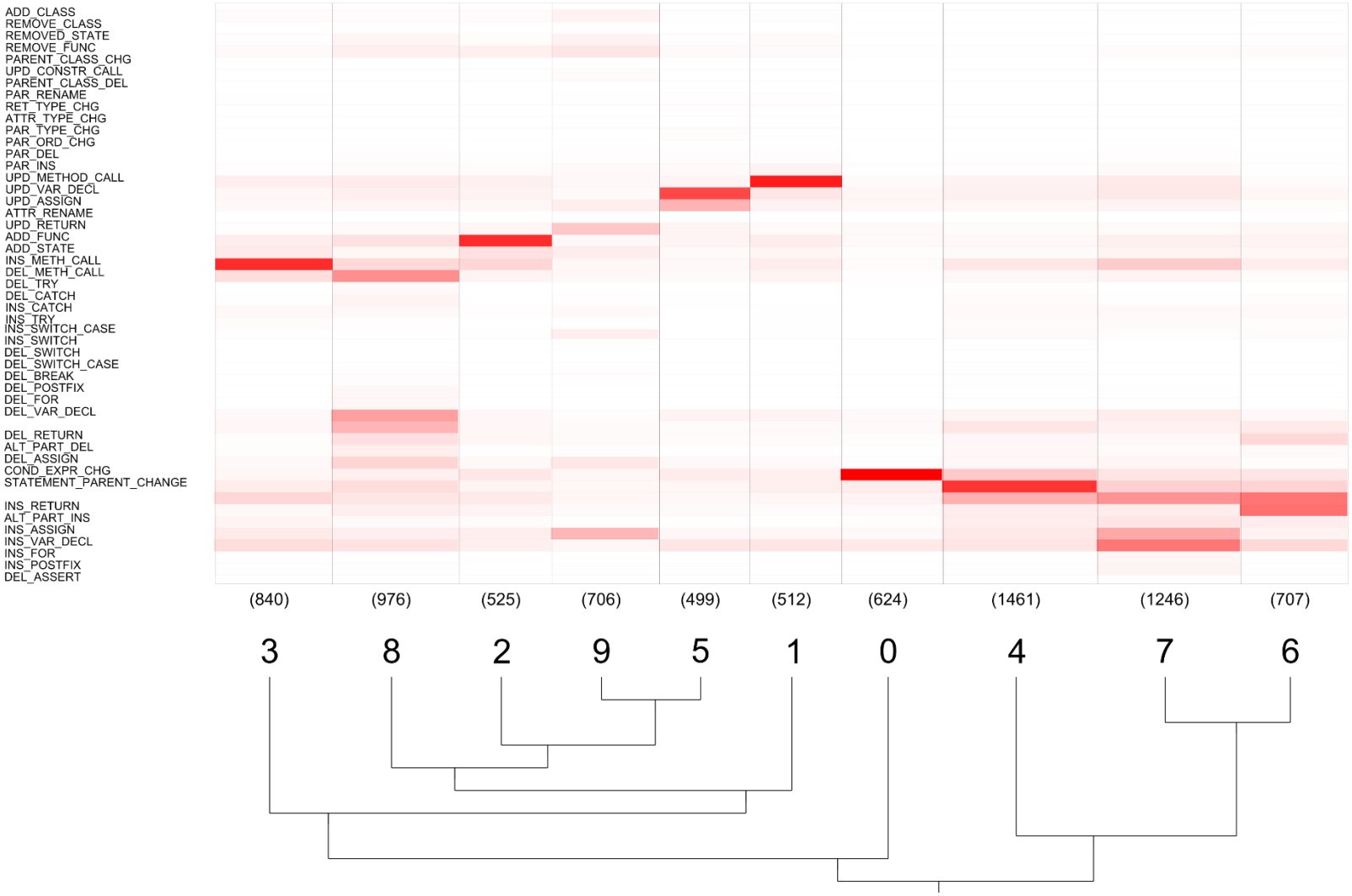

Figure 15 - Matrix Visualization of Clusters from Eclipse JD

6. Logic faults; primarily a failure to check pre-conditions
7. Faults with incorrect algorithm or behavior
8. Faults that require the removal of extraneous behavior
9. Rare, varied faults that should be manually inspected

6.3 Experimental Design

In this study I undertake analysis of the *fault profile*, that is, the frequency of fault occurrence in each fault class. Each cluster is treated as a fault class. As mentioned by Freimut [39], the use of the chi-square test can be used to test whether faults are distributed uniformly, or whether they are statistically independent.

6.3.1 Distribution of faults by subcomponent

For my first research question I want to know whether there is a relationship between a fault's class and the subcomponent in which it is observed. If such a relationship exists, the distribution of faults among fault classes will differ for each subcomponent.

RQ6.1: Over time, do the same types of faults tend to occur in a given subcomponent?

I define f_{S0} as the frequency of fault class zero (0) in subcomponent s. S is the set of all subcomponents of the Java Development Toolkit that had fault fixes. F_s is a vector composed of the frequencies for individual fault classes f_{s0}, f_{s1}, …, f_{sn}. F_{sE} is a vector composed of the expected frequencies of individual fault classes for subcomponent s. F_{sE} is calculated by assuming that the distribution of faults for the JDT project are reflected in each of the subcomponents. For each subcomponent, the total number of faults in that subcomponent is multiplied by the frequency of each fault class in the JDT over all seven releases.

My independent variable is the subcomponent. My dependent variable is the distribution of the faults, F_S. My null hypothesis is that fault classes from the subcomponents of JDT are distributed evenly.

$$H_0: \forall\ s\ \in\ S, F_s = F_{sE}$$

My alternative hypothesis is that fault classes are not distributed evenly.

$$H_A: \exists\ s\ \in\ S, F_s \neq F_{sE}$$

I calculated the expected frequency for all subcomponents in the JDT that contained faults. Six of these subcomponents had an adequate number of faults to meet the minimum requirements of a X^2 test (expected frequency >5 for each category). I performed a X^2 goodness of fit test individually for each subcomponent. The resulting *p*-Value of each test is given in Table 37. Items in bold were significant at the α = 0.05 level.

Table 37 - Fault distribution for JDT subcomponents

Subcomponent	*No. of Faults*	*p*-Values
org.eclipse.jdt.core	2577	**4.52E-43**
org.eclipse.jdt.debug	501	8.24E-02
org.eclipse.jdt.debug.ui	984	**3.85E-17**
org.eclipse.jdt.junit	171	**5.54E-04**
org.eclipse.jdt.launching	284	5.43E-02
org.eclipse.jdt.ui	3673	**1.82E-12**

Two of the subcomponents, org.eclipse.jdt.debug and org.eclipse.jdt.launching, have a distribution that is very similar to the expected frequency. For these two subcomponents, the null hypothesis cannot be rejected. These two subcomponents have the same fault classes in similar proportions. The similarity can be seen in Figure 16 below.

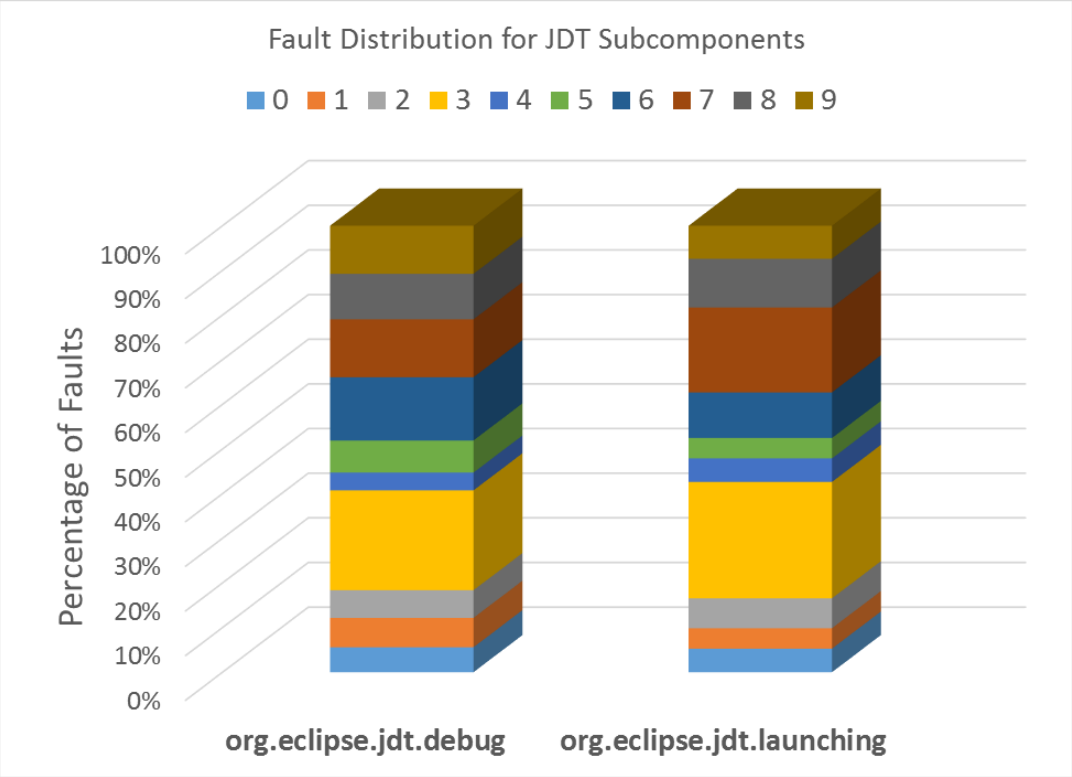

Figure 16 – Similar Fault Distributions for two subcomponents

The distribution of faults for the four remaining subcomponents differs significantly from the distribution seen at the JDT project level. The distribution of faults in these subcomponents can be seen in Figure 17.

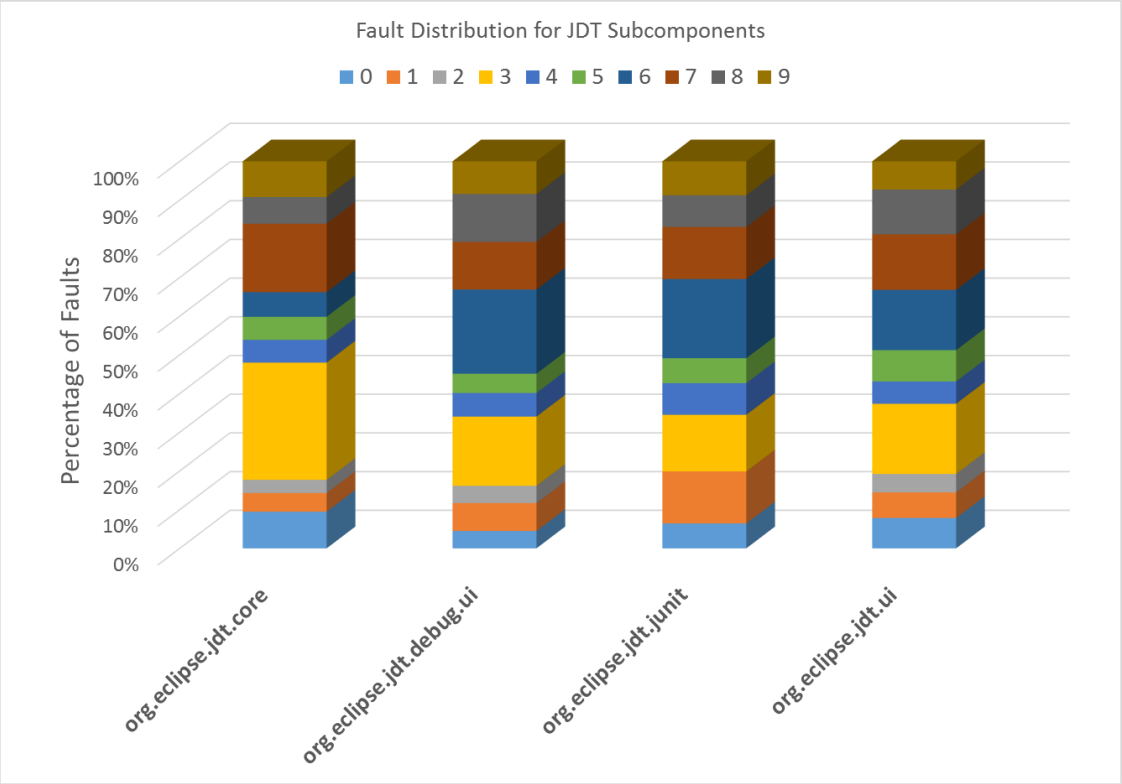

Figure 17 - Fault Distribution for four JDT subcomponents

The JDT core subcomponent (org.eclipse.jdt.core) has a large proportion of faults for cluster 3 (Additional Functionality). It also has a much lower proportion of faults in cluster 6 (Insert Return and Insert If).

The JDT Debug UI subcomponent has a significantly smaller proportion of faults in Cluster 0 (Condition Expression Change) and Cluster 3 (Additional Functionality). It has a significantly larger proportion of faults in Cluster 6 (Insert Return and Insert If).

The JDT JUnit subcomponent has zero faults in Cluster 2 (Update Method Call) and contains a large proportion of faults in Cluster 1 (Update Variable Declaration).

The JDT UI subcomponent has the largest number of faults for the studied time period. Similar to the JDT Debug UI, the JDT UI subcomponent has a significantly smaller proportion of faults in Cluster 3 (Additional Functionality), and a larger proportion of faults in Cluster 6 (Insert Return and Insert If). Unlike

the JDT Debug UI subcomponent, the proportion of Cluster 0 (Condition Expression Change) is equal to the expected proportion. The similarity in the two subcomponents may be due to their similar purpose in the architecture. This led us to perform a test of independence between the fault distributions between the two subcomponents. I normalized the values and investigated the following hypotheses.

My null hypothesis is that the distribution of faults for the two UI subcomponents are equal.

$$H_0: F_{jdt.ui} = F_{jdt.debug.ui}$$

My alternative hypothesis is that faults are from different distributions.

$$H_A: F_{jdt.ui} \neq F_{jdt.debug.ui}$$

The $X^2 = 0.0296 < X^2_{0.05,\ 9} = 16.92$. Thus, the null hypothesis cannot be rejected, indicating that the normalized distribution of faults among the fault classes is not significantly different. The normalized distributions are shown in Figure 18. This finding suggests that the fault distribution is a function of the purpose of the subcomponent, rather than (or perhaps in addition to) the project in which it resides. An analysis of additional projects, along with a categorization of subcomponent types, is necessary to better understand this relationship.

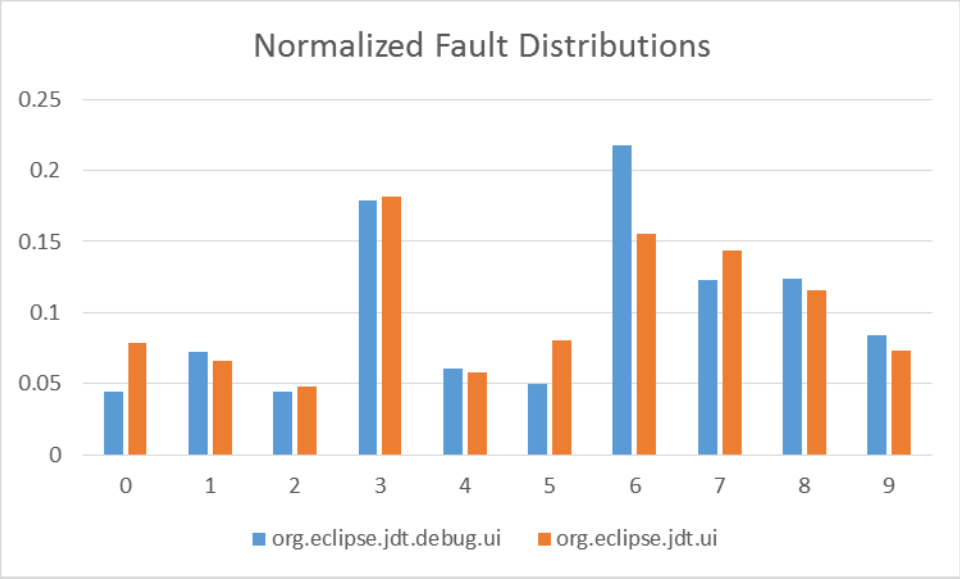

Figure 18 - Normalized Fault Distributions

6.3.2 Fault distribution for single and multi-file fixes

My next research question investigates the size of a fault fix with respect to the number of files that are altered. Intuitively, one might suspect that faults fixed within a single file are less complex in nature. However, what does this imply for the classification of the fault based on syntactical change data?

RQ6.2: Are certain fault classes more likely to be fixed by single or multi-file changes?

I filter the file count data so that unclassified changes and changes to comments are excluded. Note that unclassified changes represent 88 of 19946 file revisions. Changes to comments have no impact.

There are 4867 single file fault fixes and 3219 multi-file fault fixes. The average number of files changed for a fault fix is 2.54 and the median number of files is one. The standard deviation is 5.6 files.

I perform a X^2 goodness of fit test to determine if the single-file fix frequencies have a distribution similar to the multi-file fix frequencies.

My null hypothesis states that the distribution of faults for single file and multi-file fault fixes are equal.

I define F_{SF} as the vector of observed frequencies for all fault classes that are repaired with a change to a single Java file. I define F_{MF} as the vector of expected frequencies for all fault classes that are repaired by changing more than one Java source file.

$$H_0: F_{SF} = F_{MF}$$

My alternative hypothesis is that the distribution of fault classes differs for single file and multi-file fixes.

$$H_A: F_{SF} \neq F_{MF}$$

The X^2 test is significant at the α=0.05 level, allowing the rejection of the null hypothesis and leading to the conclusion that these distributions are significantly different. The distributions are shown in Figure 19.

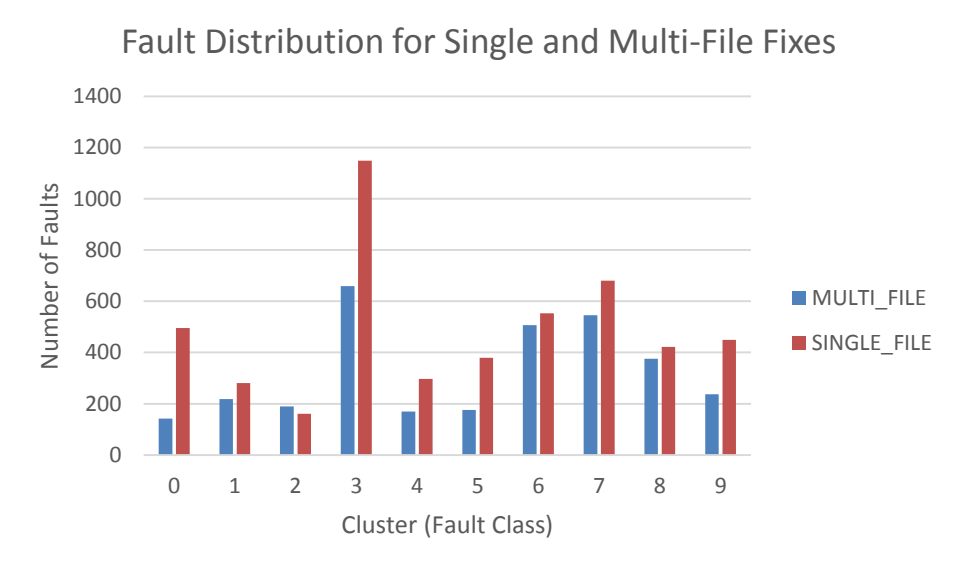

Figure 19 - Fault Distribution for Single and Multi-File Fixes

There are a number of interesting observations that can made from the distribution. Cluster 0 (Condition Expression Changes) and Cluster 4 (Statement

Parent Change and Insert If) both represent logic changes. As one might expect, these types of logic changes appear to be much more common in single file changes. On the other hand, Cluster 3 (Additional Functionality) is also more common in single file changes. This suggests that many new functions are called only within their class, or exposed through public APIs. Cluster 5 (Insert Method Call) is also more common in single file changes. This may indicate that interface faults are often fixed on the caller side.

6.3.3 Fault distribution in terms of developer

In this section I look at the distribution of faults among the authors of the fixes.

RQ6.3: Do developers tend to fix the same types of faults?

I start with 35 developers that committed fault fixes to the JDT for one of the seven versions in the case study. Eighteen of the 35 fixed enough faults that the assumptions of X^2 could be met (expected value > for all cells). As with previous tests, I calculate an expected distribution based on the distribution of faults in the JDT project. The number of fault fixes that were logged for each author is multiplied by the frequency of each fault type to arrive at the expected values. The independent variable is the author of the fault fix. The dependent variable is the distribution of the fault fixes.

I define *A* to be the set of all authors that committed fault fixes to the JDT project in the studied releases. Let *a* be an author that exists in *A*. F_a is a vector with the distribution of faults by fault class. F_{aE} is the expected distribution based on the number of faults fixed by author *a,* and the frequency of each fault class in the JDT project.

My null hypothesis is that fault fixes from the authors of the JDT project are distributed evenly.

$$H_0: \forall\, a \in A, F_a = F_{aE}$$

My alternative hypothesis is that fault classes are not distributed evenly for each author.

$$H_A: \exists\, a \in A, F_a \neq F_{aE}$$

Of these eighteen, the null hypothesis can be rejected for fourteen. The distribution of the faults for the remaining four authors was not statistically different than the distribution of faults for the JDT project. The data that was compared, as well as the *p*-value for the X^2 test, is provided in Table 38. Rows in bold are significantly different from the expected distribution. The data is ordered based on the number of total fixes committed by the author.

Table 38 - Fault Distribution for Fault Fix Commits by Author

Author	0	1	2	3	4	5	6	7	8	9	All	p-value
maeschli	70	61	37	190	61	53	139	152	86	72	921	4.05E-01
dmegert	67	43	38	149	43	63	117	105	95	56	776	5.90E-02
darin	**40**	**36**	**36**	**164**	**26**	**46**	**120**	**109**	**92**	**68**	**737**	**1.85E-03**
mkeller	**52**	**42**	**18**	**87**	**39**	**47**	**78**	**87**	**71**	**56**	**577**	**3.61E-03**
othomann	**28**	**28**	**11**	**94**	**19**	**23**	**39**	**74**	**36**	**48**	**400**	**4.37E-02**
oliviert	**23**	**21**	**11**	**108**	**20**	**35**	**27**	**85**	**22**	**43**	**395**	**1.28E-06**
dbaeumer	**23**	**26**	**26**	**51**	**16**	**29**	**66**	**66**	**51**	**24**	**378**	**3.21E-05**
akiezun	**20**	**24**	**24**	**50**	**18**	**49**	**55**	**21**	**39**	**23**	**323**	**1.45E-11**
darins	**13**	**28**	**12**	**56**	**19**	**12**	**73**	**32**	**35**	**30**	**310**	**2.56E-07**
bbaumgart	**27**	**13**		**57**	**17**	**31**	**59**	**44**	**35**	**14**	**297**	**2.61E-03**
ffusier	**22**			**113**	**18**	**12**		**59**	**15**	**25**	**264**	**8.27E-11**
jlanneluc	**25**	**16**	**12**	**71**	**18**	**14**	**16**	**44**	**23**	**15**	**254**	**3.11E-02**
pmulet	**39**			**107**	**15**	**12**	**15**	**29**	**20**	**17**	**254**	**2.82E-13**
daudel	**37**	**12**	**12**	**63**	**12**	**15**	**14**	**42**	**12**	**34**	**253**	**2.61E-06**
jeromel	15	12	12	47	21		19	41	17	14	198	6.17E-02
jburns	**11**	**14**		**36**	**21**		**37**	**28**	**20**	**14**	**181**	**1.57E-02**
lbourlier	11	14		36		13	13	26	19	18	150	3.98E-01
kent	**17**			**67**		**14**		**18**			**116**	**7.97E-13**

The relative distribution data is presented graphically in Figure 20. From the chart I can see that proportions of each type vary considerably. It is clear that faults from Cluster 3 (Additional Functionality) are quite prominent for all authors. Additional factors, such as which area of the code the author generally works, may need to be explored to better understand the distribution.

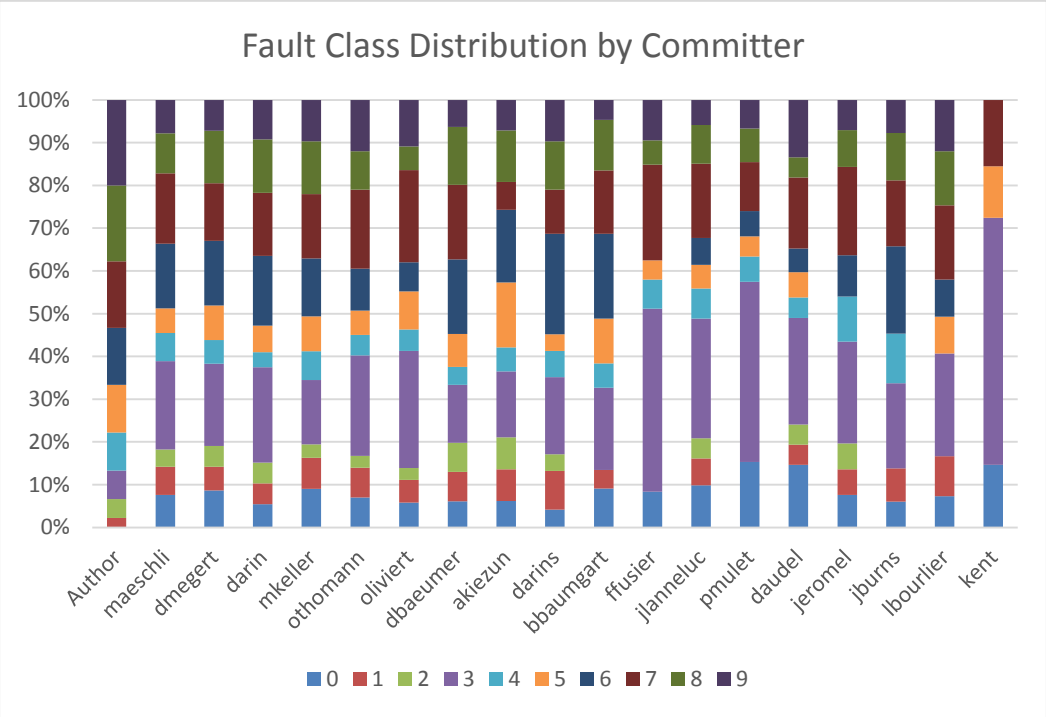

Figure 20 - Fault Distribution for Fault Fix Commits by Author

6.3.4 Fault distributions for pre-release and post-release fixes

For my next research question, I want to determine whether the distribution of pre-release faults is indicative of post-release faults. This may tell us whether certain fault classes require additional attention to prevent their occurrence as post-release faults.

RQ6.4: Are pre-release fault distributions predictive of post-release fault distributions?

The independent variable is the pre-release/post-release state of the fault fix. The dependent variable is the distribution of the faults. To test this hypothesis I calculate the expected frequency of post-release faults for each fault class based on the frequency of its occurrence in pre-release fault fixes.

F_{POST} is a vector composed of the observed frequencies of post-release faults for all fault classes. F_{POSTE} is a vector composed of the expected frequencies of all fault classes for post-release faults in a version of the Eclipse JDT project.

My null hypothesis is that the distribution of faults for pre-release and post-release faults are from the same distribution.

$$H_0: F_{POST} = F_{POSTE}$$

My alternative hypothesis is that faults are from different distributions.

$$H_A: F_{POST} \neq F_{POSTE}$$

The values for the X^2 goodness-of-fit test for each version are given in Table 39. Three of four versions exhibit a significantly different distribution (the null hypothesis can be rejected at α=0.05), while the other four exhibit a distribution that is not significantly different than that of pre-release faults.

Table 39 - *p*-values for Chi-Square Goodness-of-Fit Test

Version	*p*-Value
2.0	0.7019
2.1	0.4006
3.0	**0.0008**
3.3 Europa	**0.0113**
3.4 Ganymede	**0.0018**
3.5 Galileo	0.1047
3.6 Helios	0.2151

The relative distributions for each version is depicted in Figure 21. From this illustration I can see that the relative distribution is similar in most cases, and that variations tend to represent a handful of fault classes that occur in higher or lower frequencies than expected post-release. Cluster 0 (Condition Expression

Change) is significantly higher post-release for release 3.0 and 3.3 (Europa). Conversely, Cluster 6 (Insert Return/Insert If) is significantly lower than expected. In version 3.4 (Ganymede) the cluster with a larger proportion of faults is Cluster 1 (Update Variable Declaration) while Cluster 4 (Statement Parent Change/Insert If) and Cluster 5 (Insert Method Call) are both smaller than expected.

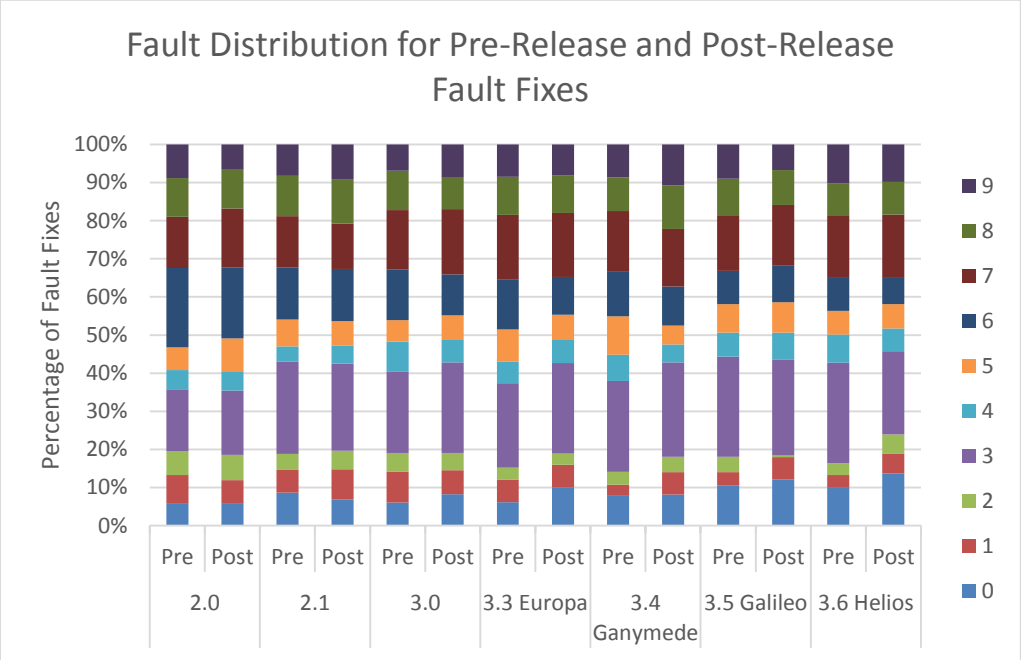

Figure 21 - Pre-Release/Post-Release Fault Fix Distribution

6.3.5 Fault distribution for problematic fixes

While mining data from the JDT, I noticed that some faults require multiple commits before they are fixed. In some cases, this can be attributed to minor issues that are rectified quickly. I refer to changes that require multiple rounds of changes as *problematic fixes*. For my next research question I investigate the fault classes for these changes.

RQ6.5: Are problematic fault fixes distributed evenly among fault classes?

I only looked at faults where the file was changed more than once, with at least a four hour time lapse between changes. Of the 4054 files that are involved in fault fixes, 1708 files meet this criterion and represent 840 fault fixes. I calculate the expected distribution based on the overall distribution of each fault class.

The independent variable for this test is the status of the fault fix as problematic. The fault fix belongs to the set of faults that required multiple changes to repair. The dependent variable is the distribution of the faults among the fault classes. I define F_{PR} as a vector composed of the observed frequencies of problematic fault fixes for all fault classes. F_{PRE} is a vector composed of the expected frequencies of all fault classes for problematic fault fixes in a version of the Eclipse JDT project.

My null hypothesis is that the distribution of problematic fault fixes is the same as the distribution of faults in the JDT project.

$$H_0: F_{PR} = F_{PRE}$$

My alternative hypothesis is that problematic fault fixes are from a different distribution.

$$H_A: F_{PR} \neq F_{PRE}$$

The X^2 goodness-of-fit test for homogeneity against the expected distribution is significant for $\alpha = 0.05$, indicating that these faults are not distributed as expected. The data is depicted in Figure 22.

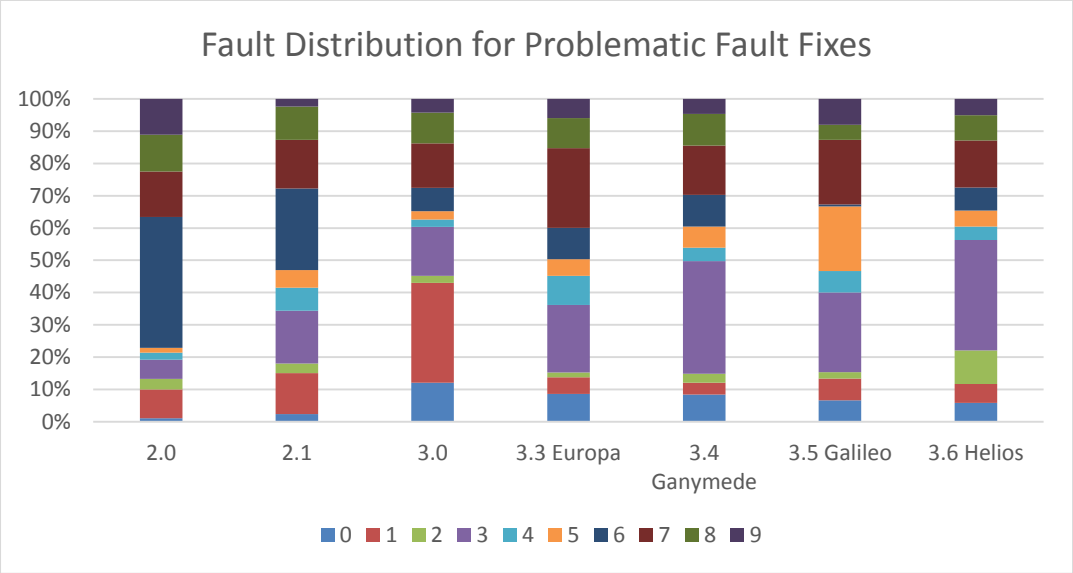

Figure 22 - Fault Distribution for Problematic Fault Fixes

I made a number of interesting observations from this data. Cluster 0 (Conditional Expression Change), Cluster 2 (Update Method Call), Cluster 4 (Statement Parent Change/Insert If), and Cluster 5 (Insert Method Call) have a consistently low frequency. This indicates that these types of changes are less likely to be problematic fault fixes. Cluster 1 (Update Variable Declaration) had an interesting increase in frequency for version 3.0 of Eclipse. The frequency of that type decreased in subsequent releases. Cluster 6 (Insert Return/Insert If) decreases in frequency in subsequent releases. These types of changes may become less complex as the software matures.

Cluster 3 (Additional Functionality) increases in relative frequency for later releases. It is the most consistent contributor to problematic faults. This indicates that faults that must be resolved through additional functionality are more likely to require multiple rounds of changes, and are likely more complex.

Cluster 7 (Insert Variable Declaration/Insert If/Insert Assignment), Cluster 8 (Delete Method Call/Delete Variable Declaration/Delete If), and Cluster 9 (Garbage Collector) seem to occupy 30-40% of these problematic faults for all releases. This is consistent with the idea that these clusters with lower internal similarity, and more descriptive features, represent more complex changes. The

increased complexity of the fix makes the probability that a fault is problematic, and must be re-visited, more likely.

6.4 Conclusions

In this chapter I analyze the distribution of software faults, as classified by the clustering of syntactic changes. As a case study I consider seven versions of the Java Development Tools (JDT), a development environment built on the Eclipse platform. For these seven releases, 8768 fault fixes with Java source code changes are included in the analysis.

For my first research question, I examine the distribution of software faults in six subcomponents of the JDT. If there is no difference in the distribution of faults in these subcomponents, I expect the distribution of the subcomponent to be similar to that of the JDT project. Two of the six distributions are not significantly different from the distribution at the project level. The remaining four have distributions that differ significantly from the expected distribution. During this investigation, I found that the normalized distributions of the two user interface subcomponents (org.eclipse.jdt.ui and org.eclipse.jdt.debug.ui) are not significantly different. This is an indication that the fault distribution may vary based on the purpose of the subcomponent in this project.

I also investigate the distribution of faults for single and multi-file fault fixes. Logic changes appear to occur more frequently in single file fixes, as one might expect, but additional functionality also occurs more often in single file fault fixes. This is a surprising finding, and may be due to Eclipse's component-based architecture.

My third research question looks at the distribution of faults committed by developers. Eighteen developers had enough faults to analyze using the X^2 test. Of these eighteen developers, fourteen had distributions that are significantly different from the expected distribution. I found that faults repaired by adding functionality were common for all authors.

The distributions for pre-release and post-release faults provided mixed results. The two earliest releases (Eclipse 2.0 and Eclipse 2.1) had post-release

fault distributions that are not significantly different from those of pre-release faults. This may indicate poor quality. Eclipse versions 3.0, 3.3, and 3.4 have post-release fault distributions that are significantly different from pre-release fault distributions. I also notice that faults repaired by additional functionality have a reduced relative frequency. This may be a sign of improved quality and stability. However, the last two releases (Eclipse 3.5 and 3.6) return to post-release fault distributions that are not significantly different from pre-release. Given the reduced number of fault fixes for these versions, this may indicate that few new features are added, and fault distributions have reached a steady-state.

I define the concept of a *problematic fix*, a fix which requires multiple attempts for resolution. In order to minimize coincidental problems I limit the investigation to fixes where a second commit occurs after a four hour lapse. The period of four hours was chosen to eliminate small mistakes that do not represent problematic constructs. For example, a developer may forget to include a file with a check-in, and as a result, must add the file after the initial transaction. The four hour period likely eliminates simpler problem cases, but preserves those that require significant re-work.

Initialization faults (Cluster 1 – Update Variable Declaration) and logic faults (Cluster 6 – Insert If and Return) seem to decrease in relative frequency over time. In converse, the relative frequency of Cluster 3 (Additional functionality) seems to increase over time. Cluster 7 (Insert If + Variable Declaration + Assign), Cluster 8 (Delete Method Call + Var Declaration + If) and the “garbage collector,” Cluster 9 (Insert Assign + Update Return), consistently make up 30-40% of the problematic fixes. Since clusters are ranked by the tightness of the cluster, these clusters represent more complex faults. It appears that faults in these clusters are more likely to encounter difficulty when repaired.

The findings in this chapter show how the distribution of fault classes can be analyzed for software projects in order to gain insight into the evolution of a software system. This level of large-scale analysis can be used to gain insight into the development process and the quality of the product that is being developed. Many software development organizations have not adopted fault

classification due to the overhead involved in getting consistent, high quality data. Automated classification provides access to this data, and historical data, at much lower cost.

Chapter 7

Conclusions and Future Work

7.1 Threats to Validity

In this chapter I discuss the threats to validity for each of the preceding three chapters, discuss the contributions in this dissertation, and conclude with a discussion of future work.

Wohlin et al. describe four areas where the validity of the results may be threatened [89]. I discuss threats in each of these four areas.

Conclusion Validity concerns the statistical significance of the result. It is important that the relationship between the treatment and the outcome are properly measured in order to draw proper conclusions. In order to counter this threat during statistical tests, the pre-requisites of each statistical test are confirmed.

In Chapter 4 and Chapter 5, data is checked for a normal distribution using the Shapiro-Wilk test. In cases where the data is not normally distributed, the non-parametric Wilcoxon signed rank test is used. Pearson's correlation coefficient is used in Chapter 5 when the data is normally distributed.

The X^2 goodness-of-fit test is used to test hypotheses in Chapter 6. This test is not recommended if the frequency for any category is less than five. In order to meet the pre-requisites, only data that met this criterion was used for the statistical tests.

Internal Validity is concerned with my ability to correctly measure the influence of the independent variables on the dependent variables and the elimination of possible confounding variables that may lead to incorrect conclusions. The manual inspection of a random subset of faults from each cluster is an important component of this research, but the sample size may be too small for statistically significant results. In addition, there is a mono-operation bias that could be eliminated by allowing independent review and classification of the results. This is a common problem in fault classification studies, since most organizations that have classified fault data will not share it. In this dissertation I

have made my manual classification notes publicly available so that other researchers can build on this work and improve upon my results.

There may be undetected problems in the software that is used to collect data for this study. We build on an existing dataset to help limit this threat [74]. I made updates to the dataset and associated scripts in order to remove some errors, but other errors may exist. I utilize the ChangeDistiller tool [75] to collect change information, but also altered this program. ChangeDistiller may have had errors that affect these results or I may have introduced problems when I made changes. Both versions of the ChangeDistiller tool are publicly available so that other researchers can identify problems and improve results.

The data in the problem tracking database, and the comments in the version control system depend on the software developer to get accurate information. I found one instance where a fault identifier was mistyped, and other faults are likely to be similarly mislabeled.

Construct Validity refers to how well the independent and dependent variables in the study measure what is intended. Classification of software faults by the syntax of the fix is difficult due to the uncertainty of the developer's intent. While simple changes are easier to interpret, complex changes can be difficult to understand based on the frequency of changes alone. The alternative would be to use the description of the fault. This method has similar problems because the relationship between the symptom recorded and the underlying fault may not be clear. Henningsson and Wohlin found that use of a description alone for fault classification resulted in low agreement [12]. To counter this threat I used a large number of software faults for analysis. In addition, the clustering method isolates faults that may be infrequent. Gaining more precise data from the syntax of the source code is discussed further in the future work section.

External Validity refers to the ability to generalize the results of the study. I do not claim that these results can be generalized outside of the Eclipse project. I analyzed all of the faults from two versions of Eclipse, and all of the faults from seven versions of the Java development tools project. This provides a vertical slice (all projects for two versions) and a horizontal slice (seven versions for one

component) that allow me to investigate different aspects of the method. The consistency in the clusters for these different slices provides strong evidence of the validity of this approach within Eclipse projects.

There are a number of additional factors that must be considered before the results of the experiment can be generalized. I will discuss the development community, architecture, domain, and programming language as factors that impact external validity.

The Eclipse community consists of a number of open source contributors and a process for coordinating multiple projects. Other projects include different developers and different processes that could lead to different findings. One possible project to further generalize these results without considering other factors is the NetBeans development platform, which has a similar purpose and underlying architecture[12]. The evaluation of commercial software is also an important direction to extend the work, since the development process is likely to be very different.

Eclipse uses a very modular, component-based architecture. This architecture influences the way that code is structured, and the way that software faults are repaired. For example, the finding that additional functionality is often added with changes to a single file may be due to the component-based structure of Eclipse. A study of development environments with different architectures could improve our understanding of which results can be generalized, and may also provide insight into the quality impacts of different architectural decisions.

The domain of the software also has an impact on our ability to generalize the results. The domain can influence the complexity of the software, the types of operations that need to be performed, and the types of non-functional requirements that must be met, such as performance and reliability. Each of these factors lead to the use of different data structures and algorithms, which may exhibit different types of faults.

[12] http://wiki.netbeans.org/OSGiAndNetBeans

My study was limited to the Java programming language. While I would expect similar results from other strongly-typed, object-oriented programming languages, additional studies are needed to confirm these studies. In addition the use of dynamic scripting languages and functional programming have become increasingly popular, and these languages will have an influence on the way that faults are repaired.

7.2 Contributions

This dissertation has presented a method and toolset to automatically classify software faults from the syntax of the source code fix. Other researchers focus primarily on the use of the text in the problem report for classification [61], [62] or only identify pre-determined syntax patterns in the repair [63]. Fault classification research has shown that the textual description of the fault is insufficient for fault classification [12]. The results in this study support the notion put forth by DeMillo and Mathur that “syntax is the carrier of semantics” [90].

The following contributions were made in this dissertation towards the goal of providing automated fault classification of software faults:

1. The change taxonomy published by Fluri and Gall [66] was extended to support the analysis of software faults. I found that the change types occur often for fault fixes in two versions of the Eclipse project, and that the frequency of occurrence for the change types is correlated, indicating a consistency of occurrence.
2. A method to cluster faults using the syntax of the fault fix is described. The frequency of change types from the extended change taxonomy are used as an input vector to the clustering algorithm. The CLUTO clustering toolkit is used to perform clustering [76]. The cosine similarity function is used as the internal similarity measure. The resulting clusters were consistent for two versions of Eclipse. The use of the I1 criterion function reduces noise in the data by creating a single, low similarity cluster with data

that does not match other clusters [79]. This low quality cluster isolates faults that occur infrequently and may require manual classification.

3. Changes to the ChangeDistiller tool were made to overcome limitations with respect to the handling of anonymous classes. These changes resulted in measurable improvements to the results and indicate that additional incremental improvements are possible.
4. The MiSFIT (Mining Software Fault Information and Types) toolset is presented. The toolset provides a flexible workflow to process fault information in a reliable and scalable manner.
5. Analysis of the software fault distribution for individual subcomponents of the JDT indicates that the distribution varies by the purpose of the subcomponent. This supports prior evidence that faults vary by the purpose of the component [26].
6. Single file fault fixes in the JDT included a large percentage of faults that required additional functionality to repair the fault. This is a surprising finding that may be due to Eclipse's component-based architecture.
7. I found that the relative frequency of faults that require additional functionality is high for all developers within the Eclipse JDT.
8. When analyzing the distribution of software faults that were *problematic*, requiring multiple changes to repair, it was discovered that algorithmic faults, faults repaired by the removal of code, faults repaired by the addition of functionality, and infrequent faults are more likely to be problematic to repair. This indicates that these types of fault fixes may benefit the most from review before they are committed.

7.3 Future Work

The results of this dissertation indicate that the classification of software faults by the syntax of the fix is a useful method to analyze software faults. This work can be furthered in a number of ways.

The syntax of software fault fixes can be complex for multiple reasons. Some non-essential changes (e.g., renaming a variable) produce “noise” in the data. Kawrykow and Robillard developed DiffCat, a tool to filter out these changes from source code [86]. Similarly, Thung et al. further this research by narrowing the essential changes to the root cause [91]. The use of these tools can greatly reduce the number of syntactical elements that are considered for classification and lead to more precise classifications.

Multiple fault fixes are sometimes committed to a software repository in a single transaction. This may be because the two reported failures are caused by the same underlying fault. However, it may also be due to the fact that the faults are close together, and working on them together was more efficient for the software developer. The latter situation results in a need to identify multiple root causes in a single set of source code changes.

Selection of the CLUTO toolkit for clustering was based on several requirements, including a need for a pre-existing tool to perform clustering. While CLUTO contains several clustering algorithms, a more extensive comparison of clustering techniques is a possible area for future work. In addition to clustering, other statistical and machine learning techniques could be utilized to classify software faults. The discovery of a superior classification method would help advance this research.

One possible application of this research is the development of a decision support system (DSS) to aid a classifier in the fault classification process [92]. Such a decision support system can be used to improve the efficiency and consistency of the fault classification task where expert opinion is needed for fault classification. The DSS would also provide a valuable tool for researchers to evaluate and improve upon the method and tools in this dissertation.

The use of the ChangeDistiller application for extracting the source code changes limits this work to the Java programming language. Extending ChangeDistiller to work on additional programming languages can expand the scope of the research in this dissertation and improve the external validity of the study.

As I described in the review of current literature, fault links define a relationship between the types of components and the types of faults that occur in the components [26]. Past research on fault links has been conducted using manual classification of components and faults [26], [93], [94]. This research provides a method to automate the fault classification. There are multiple techniques to classify the component or module. For example, Marinescu defines *Detection strategies*, an approach that utilizes static code metrics and rules to identify design flaws in object-oriented software [95]. The study of fault links that are associated with these design flaws could aid our understanding of their impact. A more general way to classify classes or components is the use of stereotypes, which define the role of the class. Dragan et al. provide an automated method of identifying method and class stereotypes from source code [96], [97]. A better understanding of fault links can further aid in verification and validation improvement activities, and may also provide a mechanism to perform tradeoff analysis for refactoring and restructuring activities.

Buse and Zimmermann hypothesize that the application of analytics to software development activities can aid in decision-making for project managers and developers [98]. They argue that software development has several properties that make analytics applicable, and cite the successful application of analytics to other fields with similar properties. Based on a survey of project managers and developers, they suggest several areas where software analytics could be used.

I argue that fault classification data is applicable to many of the software analytics themes that are presented by Buse and Zimmermann [99]. Furthermore, automation of fault classification data is necessary to drive broad industry adoption. The extension of this work to build software analytics systems

that aid in decision making is, therefore, a promising area of future research. The combination of the automated fault type data from MiSFIT with other automated techniques to separate faults from enhancements [22], predict severity [24][25], and predict the customer impact [23] provide a powerful toolset for fault analysis. This data can be analyzed from multiple perspectives along with additional information such as quality metrics and effort data to drive informed decisions to improve efficiency and quality.

References

[1] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, “Orthogonal Defect Classification-A Concept for In-Process Measurements,” *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, 1992.

[2] B. Boehm and V. R. Basili, “Top 10 list [software development],” *Computer*, vol. 34, no. 1, pp. 135–137, Jan. 2001.

[3] N. Bridge and C. Miller, “Orthogonal defect classification using defect data to improve software development,” *Software Quality*, vol. 3, no. 1, pp. 1–8, 1997.

[4] W. D. Yu, “A software fault prevention approach in coding and root cause analysis,” *Bell Labs Technical Journal*, vol. 3, no. 2, pp. 3–21, 1998.

[5] A. A. Shenvi, “Defect prevention with orthogonal defect classification,” in *Proceeding of the 2nd annual conference on India software engineering conference*, Pune, India, 2009, pp. 83–88.

[6] G. Vijayaraghavan and C. Kaner, “Bug taxonomies: Use them to generate better tests.,” presented at the Software Testing, Analysis & Review Conference (Star East), Orlando, FL, 2003.

[7] L. A. Miller, E. H. Groundwater, J. E. Hayes, and S. M. Mirsky, “Guidelines for the Verification and Validation of Expert System Software and Conventional Software: Survey and Assessment of Conventional Software Verification and Validation Methods. Volume 2,” Nuclear Regulatory Commission, Washington, DC (United States). Div. of Systems Technology; Electric Power Research Inst., Palo Alto, CA (United States). Nuclear Power Div.; Science Applications International Corp., McLean, VA (United States), NUREG/CR--6316-Vol.2; SAIC--95/1028-Vol.2, Mar. 1995.

[8] S. Vegas, N. Juristo, and V. Basili, “Packaging experiences for improving testing technique selection,” *Journal of Systems and Software*, vol. 79, no. 11, pp. 1606–1618, Nov. 2006.

[9] P. Runeson, C. Andersson, T. Thelin, A. Andrews, and T. Berling, “What do we know about defect detection methods? [software testing],” *IEEE Software*, vol. 23, no. 3, pp. 82–90, May 2006.

[10] M. Sullivan and R. Chillarege, “Software defects and their impact on system availability-a study of field failures in operating systems,” in *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, 1991, pp. 2–9.

[11] R. Chillarege and K. A. Bassin, “Software Triggers as a function of time - ODC on field faults,” presented at the DCCA-5: Fifth IFIP Working Conference on Dependable Computing for Critical Applications, 1995.

[12] K. Henningsson and C. Wohlin, “Assuring fault classification agreement - an empirical evaluation,” in *2004 International Symposium on Empirical Software Engineering, 2004. ISESE ’04. Proceedings*, 2004, pp. 95–104.

[13] D. Falessi and G. Cantone, “Exploring Feasibility of Software Defects Orthogonal Classification,” in *Software and Data Technologies*, J. Filipe, B. Shishkov, and M. Helfert, Eds. Springer Berlin Heidelberg, 2008, pp. 136–152.

[14] M. Leszak, D. E. Perry, and D. Stoll, “A Case Study in Root Cause Defect Analysis,” in *Software Engineering, International Conference on*, Los Alamitos, CA, USA, 2000, p. 428.
[15] J. Dyre-Hansen, “Analysis of fault reports from online-systems,” Norwegian University of Science and Technology, Department of Computer and Information Science, Depth Study TDT4735, Dec. 2006.
[16] D. Kelly and T. Shepard, “A case study in the use of defect classification in inspections,” in *CASCON ’01: Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research*, Toronto, Ontario, Canada, 2001.
[17] B. Freimut, C. Denger, and M. Ketterer, “An Industrial Case Study of Implementing and Validating Defect Classification for Process Improvement and Quality Management,” in *Software Metrics, IEEE International Symposium on*, Los Alamitos, CA, USA, 2005, p. 19.
[18] C. B. Seaman, F. Shull, M. Regardie, D. Elbert, R. L. Feldmann, Y. Guo, and S. Godfrey, “Defect categorization: making use of a decade of widely varying historical data,” in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, Kaiserslautern, Germany, 2008, pp. 149–157.
[19] J. H. Hayes, “Building a requirement fault taxonomy: experiences from a NASA verification and validation research project,” in *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, 2003, pp. 49–59.
[20] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, “Automated support for classifying software failure reports,” in *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon, 2003, pp. 465–475.
[21] P. Runeson, M. Alexandersson, and O. Nyholm, “Detection of Duplicate Defect Reports Using Natural Language Processing,” in *29th International Conference on Software Engineering, 2007. ICSE 2007*, 2007, pp. 499–510.
[22] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, “Is it a bug or an enhancement?: a text-based approach to classify change requests,” in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, New York, NY, USA, 2008, pp. 23:304–23:318.
[23] L. Huang, V. Ng, I. Persing, R. Geng, X. Bai, and J. Tian, “AutoODC: Automated generation of Orthogonal Defect Classifications,” in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2011, pp. 412–415.
[24] T. Menzies and A. Marcus, “Automated severity assessment of software defect reports,” in *IEEE International Conference on Software Maintenance, 2008. ICSM 2008*, 2008, pp. 346–355.
[25] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, “Predicting the severity of a reported bug,” in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*, 2010, pp. 1 –10.

[26] J. H. Hayes, I. R. Chemannoor, and E. A. Holbrook, “Improved code defect detection with fault links,” *Softw. Test. Verif. Reliab.*, p. n/a–n/a, 2010.
[27] “IEEE Standard Glossary of Software Engineering Terminology,” IEEE Computer Society, IEEE Std 610.12-1990, Dec. 1990.
[28] M. Hamill and K. Goševa-Popstojanova, “Common Trends in Software Fault and Failure Data,” *Software Engineering, IEEE Transactions on*, vol. 35, no. 4, pp. 484–496, 2009.
[29] C. Larman and V. R. Basili, “Iterative and Incremental Development: A Brief History,” *Computer*, vol. 36, no. 6, pp. 47–56, 2003.
[30] W. Royce, “Managing the Development of Large Software Systems,” in *Proc. Westcon*, 1970, pp. 328–339.
[31] M. M. Lehman, “Programs, life cycles, and laws of software evolution,” *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, Sep. 1980.
[32] M. M. Lehman, “Laws of software evolution revisited,” in *Software Process Technology*, C. Montangero, Ed. Springer Berlin Heidelberg, 1996, pp. 108–124.
[33] M. M. Lehman and J. F. Ramil, “Software evolution—Background, theory, practice,” *Information Processing Letters*, vol. 88, no. 1–2, pp. 33–44, Oct. 2003.
[34] S. L. Pfleeger and J. M. Atlee, *Software Engineering: Theory and Practice*, 4 edition. Upper Saddle River N.J.: Prentice Hall, 2009.
[35] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*, Anniversary edition. Reading, Mass: Addison-Wesley Professional, 1995.
[36] “Orthogonal Defect Classification - IBM,” 22-Mar-2013. [Online]. Available: http://researcher.watson.ibm.com/researcher/view_group.php?id=480. [Accessed: 14-Feb-2015].
[37] R. Chillarege, W.-L. Kao, and R. G. Condit, “Defect type and its impact on the growth curve [software development],” in *, 13th International Conference on Software Engineering, 1991. Proceedings*, 1991, pp. 246–255.
[38] J. K. Chaar, M. J. Halliday, I. S. Bhandari, and R. Chillarege, “In-Process Evaluation for Software Inspection and Test,” *IEEE Trans. Softw. Eng.*, vol. 19, no. 11, pp. 1055–1070, Nov. 1993.
[39] B. G. Freimut, “Developing and Using Defect Classification Schemes,” Fraunhofer IESE, IESE-Report 072.01/E, Sep. 2001.
[40] D. E. Knuth, “The errors of TeX,” *Software: Practice and Experience*, vol. 19, no. 7, pp. 607–685, 1989.
[41] D. E. Perry and W. M. Evangelist, “An Empirical Study of Software Interface Faults,” 1985.
[42] D. E. Perry and W. M. Evangelist, “An Empirical Study of Software Interface Faults - An Update,” in *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences.*, Hawaii, 1987, vol. II, pp. 113–126.
[43] R. R. Lutz and I. Carmen Mikulski, “Empirical Analysis of Safety-Critical Anomalies During Operations,” *IEEE Trans. Softw. Eng.*, vol. 30, pp. 172–180, Mar. 2004.

[44] B. Robinson, P. Francis, and F. Ekdahl, “A defect-driven process for software quality improvement,” in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, Kaiserslautern, Germany, 2008, pp. 333–335.
[45] J. Børretzen and J. Dyre-Hansen, “Investigating the Software Fault Profile of Industrial Projects to Determine Process Improvement Areas: An Empirical Study,” in *Software Process Improvement*, 2007, pp. 212–223.
[46] B. Beizer, *Software Testing Techniques, 2nd Edition*, 2 Sub. International Thomson Computer Press, 1990.
[47] B. Beizer and O. Vinter, “Bug Taxonomy and Statistics,” Software Engineering Mentor, 2630 Taastrup, Technical Report, 2001.
[48] D. Cotroneo and H. Madeira, “Introduction to Software Fault Injection,” in *Innovative Technologies for Dependable OTS-Based Critical Systems*, Springer, 2013, pp. 1–15.
[49] A. J. Offutt and R. H. Untch, “Mutation Testing for the New Century,” W. E. Wong, Ed. Norwell, MA, USA: Kluwer Academic Publishers, 2001, pp. 34–44.
[50] J. Christmansson and R. Chillarege, “Generation of an error set that emulates software faults based on field data,” in *Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS ’96)*, 1996, p. 304.
[51] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, “On the value of static analysis for fault detection in software,” *Software Engineering, IEEE Transactions on*, vol. 32, no. 4, pp. 240–253, 2006.
[52] N. Li, Z. Li, and X. Sun, “Classification of Software Defects Detected by Black-box Testing: An Empirical Study,” in *Proceedings of 2010 Second World Congress on Software Engineering (WCSE 2010)*, Wuhan, China, 2010, pp. 234–240.
[53] R. Chillarege and K. Ram Prasad, “Test and development process retrospective - a case study using ODC triggers,” in *International Conference on Dependable Systems and Networks, 2002. DSN 2002. Proceedings*, 2002, pp. 669–678.
[54] R. Telang and S. Wattal, “An Empirical Analysis of the Impact of Software Vulnerability Announcements on Firm Stock Price,” *IEEE Transactions on Software Engineering*, vol. 33, no. 8, pp. 544–557, Aug. 2007.
[55] W. Du and A. P. Mathur, “Categorization of software errors that led to security breaches,” *In Proceedings of the 21st National Information Systems Security Conference*, 1998.
[56] U. Hunny, M. Zulkernine, and K. Weldemariam, “OSDC: Adapting ODC for Developing More Secure Software,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, New York, NY, USA, 2013, pp. 1131–1136.
[57] K. El Emam and I. Wieczorek, “The repeatability of code defect classifications,” in *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, 1998, pp. 322–333.

[58] M. Leszak, D. E. Perry, and D. Stoll, “Classification and evaluation of defects in a project retrospective,” *Journal of Systems and Software*, vol. 61, no. 3, pp. 173–187, Apr. 2002.
[59] I. Bhandari, M. Halliday, E. Tarver, D. Brown, J. Chaar, and R. Chillarege, “A case study of software process improvement during development,” *IEEE Transactions on Software Engineering*, vol. 19, no. 12, pp. 1157–1170, Dec. 1993.
[60] J. Ploski, M. Rohr, P. Schwenkenberg, and W. Hasselbring, “Research issues in software fault categorization,” *SIGSOFT Softw. Eng. Notes*, vol. 32, no. 6, p. 6, 2007.
[61] F. Thung, D. Lo, and L. Jiang, “Automatic Defect Categorization,” in *Reverse Engineering, Working Conference on*, Los Alamitos, CA, USA, 2012, vol. 0, pp. 205–214.
[62] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, “Bug characteristics in open source software,” *Empir Software Eng*, vol. 19, no. 6, pp. 1665–1705, Jun. 2013.
[63] K. Pan, S. Kim, and E. Whitehead, “Toward an understanding of bug fix patterns,” *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, Jun. 2009.
[64] R. Merkel and S. K. Nath, “An Analysis of a fault classification scheme for Java software,” Swinburne University of Technology, Melbourne, Australia, Center for Software Analysis and Texting Technical Report 2010-002, May 2010.
[65] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall, “Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction,” *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.
[66] B. Fluri and H. C. Gall, “Classifying Change Types for Qualifying Change Couplings,” in *14th IEEE International Conference on Program Comprehension (ICPC’06)*, Athens, Greece, 2006, pp. 35–45.
[67] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*, 2nd ed. Morgan Kaufmann, 2005.
[68] J. desRivieres and J. Wiegand, “Eclipse: A platform for integrating development tools,” *IBM Systems Journal*, vol. 43, no. 2, pp. 371–383, 2004.
[69] “Eclipse Project DRAFT 2.0 Plan,” *eclipse.org*, 21-Dec-2001. [Online]. Available: http://www.eclipse.org/eclipse/eclipse_project_plan_2_0_rev1221.html. [Accessed: 17-Feb-2015].
[70] D. Gruber, B. J. Hargrave, J. McAffer, P. Rapicault, and T. Watson, “The Eclipse 3.0 platform: Adopting OSGi technology,” *IBM Systems Journal*, vol. 44, no. 2, pp. 289–299, 2005.
[71] “Eclipse Project 3.0 Plan (Final),” *eclipse.org*, 02-Jun-2004. [Online]. Available: http://eclipse.org/eclipse/development/eclipse_project_plan_3_0.html. [Accessed: 17-Feb-2015].

[72] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting Defects for Eclipse,” in *Predictor Models in Software Engineering, 2007. PROMISE’07: ICSE Workshops 2007. International Workshop on*, 2007, p. 9.
[73] R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” in *Proceedings of the 30th international conference on Software engineering*, Leipzig, Germany, 2008, pp. 181–190.
[74] S. Krishnan, C. Strasburg, R. R. Lutz, and K. Goševa-Popstojanova, “Are change metrics good predictors for an evolving software product line?,” in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, New York, NY, USA, 2011, pp. 7:1–7:10.
[75] H. C. Gall, B. Fluri, and M. Pinzger, “Change Analysis with Evolizer and ChangeDistiller,” *IEEE Software*, vol. 26, no. 1, pp. 26–33, 2009.
[76] G. Karypis, “CLUTO: A Clustering Toolkit,” University of Minnesota, Department of Computer Science, Minneapolis, MN, Technical Report #02-017, Nov. 2003.
[77] Y. Zhao and G. Karypis, “Evaluation of hierarchical clustering algorithms for document datasets,” in *Proceedings of the eleventh international conference on Information and knowledge management*, New York, NY, USA, 2002, pp. 515–524.
[78] M. Rasmussen and G. Karypis, “gCLUTO - An Interactive Clustering, Visualization, and Analysis System,” 2004.
[79] Y. Zhao and G. Karypis, “Empirical and Theoretical Comparisons of Selected Criterion Functions for Document Clustering,” *Mach. Learn.*, vol. 55, no. 3, pp. 311–331, Jun. 2004.
[80] G. Salton, *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.
[81] Y. Zhao and G. Karypis, “Criterion Functions for Document Clustering: Experiments and Analysis,” University of Minnesota, Department of Computer Science, Minneapolis, MN, Technical Report UMN CS 01-040, 2001.
[82] L. C. Briand, C. M. Differding, and H. D. Rombach, “Practical guidelines for measurement-based process improvement,” *Softw. Process: Improve. Pract.*, vol. 2, no. 4, pp. 253–280, Dec. 1996.
[83] C. M. Lott and H. D. Rombach, “Repeatable software engineering experiments for comparing defect-detection techniques,” *Empirical Software Engineering*, vol. 1, no. 3, pp. 241–277, Jan. 1996.
[84] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1995.
[85] J. R. Landis and G. G. Koch, “The Measurement of Observer Agreement for Categorical Data,” *Biometrics*, vol. 33, no. 1, pp. 159–174, Mar. 1977.
[86] D. Kawrykow and M. P. Robillard, “Non-essential Changes in Version Histories,” in *Proceedings of the 33rd International Conference on Software Engineering*, New York, NY, USA, 2011, pp. 351–360.

[87] P. Weissgerber and S. Diehl, “Identifying Refactorings from Source-Code Changes,” in *21st IEEE/ACM International Conference on Automated Software Engineering, 2006. ASE ’06*, 2006, pp. 231–240.
[88] “Eclipse Java development tools (JDT) Overview,” *eclipse.org*. [Online]. Available: https://eclipse.org/jdt/overview.php. [Accessed: 22-Feb-2015].
[89] C. Wohlin, P. Runeson, and M. Höst, *Experimentation in Software Engineering: An Introduction*, 1st ed. Springer, 1999.
[90] R. A. Demillo and A. P. Mathur, “A Grammar Based Fault Classification Scheme and its Application to the Classification of the Errors of TEX,” Software Engineering Research Center; and Department of Computer Sciences; Purdue University, Technical Report, 1995.
[91] F. Thung, D. Lo, and L. Jiang, “Automatic recovery of root causes from bug-fixing changes,” in *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 92–101.
[92] B. Kidwell, “A decision support system for the classification of software coding faults: a research abstract,” in *Proceeding of the 33rd international conference on Software engineering*, New York, NY, USA, 2011, pp. 1158–1160.
[93] I. R. C.M., “Fault Links: Identifying Module and Fault Types and Their Relationship,” Master’s Thesis, University of Kentucky, 2004.
[94] J. H. Hayes, I. R. C.M., V. K. Surisetty, and A. Andrews, “Fault Links: Exploring the Relationship Between Module and Fault Types,” in *Dependable Computing - EDCC 2005*, 2005, pp. 415–434.
[95] R. Marinescu, “Detection strategies: metrics-based rules for detecting design flaws,” in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings*, 2004, pp. 350– 359.
[96] N. Dragan, M. L. Collard, and J. I. Maletic, “Reverse Engineering Method Stereotypes,” in *22nd IEEE International Conference on Software Maintenance, 2006. ICSM ’06*, 2006, pp. 24–34.
[97] N. Dragan, M. L. Collard, and J. I. Maletic, “Automatic identification of class stereotypes,” in *2010 IEEE International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.
[98] R. P. L. Buse and T. Zimmermann, “Information Needs for Software Development Analytics,” in *Proceedings of the 34th International Conference on Software Engineering*, Piscataway, NJ, USA, 2012, pp. 987–996.
[99] B. Kidwell and J. H. Hayes, “Toward a Learned Project-Specific Fault Taxonomy: Application of Software Analytics,” presented at the 2015 IEEE 1st International Workshop on Software Analytics (SWAN), Montréal, Canada, 2015, pp. 1–4.

Appendix

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of the University of Kentucky's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

Vita

Place of Birth:

Beckley, West Virginia

Educational Institutions Attended and Degrees Already Awarded:

West Virginia University Institute of Technology, Montgomery, WV
Bachelor of Science, December 1997
Major: Electrical Engineering
Minor: Computer Science

Research Positions Held:

UK Laboratory for Advanced Networking, College of Engineering
Software Verification and Validation Research Lab, Research Assistant
Oct. 2011 – Sept. 2014

Professional Positions Held:

Hewlett Packard, Lexington, KY

April 14, 2008 – present

Software Architect (May 2014 – present)

R&D Program manager (Nov 2010 – May 2014)

Senior Software Developer (April 2008 – Nov 2010)

Affiliated Computer Services, Inc., Lexington, KY

July 1, 2002 – April 1, 2008

Technical Systems Architect (July 2005 – April 2008)

Division Software Manager (May 2003 – June 2005)

Senior Programming Analyst (July 2002 – May 2003)

Digital Freight, Inc., Lexington, KY

March 2001 – July 2002

Software Quality Engineer

Future Phase Technology, Lexington, KY

August 1999 – March 2001

Software Engineer

Eagan, McAllister and Associates, Inc., Lexington Park, MD

January 1998 – August 1999

Associate Engineer

Professional Activities:

Program Committee Member, The 24th IEEE International Symposium on Software Reliability Engineering (ISSRE), 2013.

Grants:

Past Support

Research Assistant, NASA, "Predicting Fault Types and Fault Links in Object-Oriented Systems using Historical Data," with Allen Nikora (JPL), Kishor Trivedi, and Jane Hayes (advisor), $843K for three years, Oct. 2011 – Oct. 2014.

Scholastic and Professional Honors:

Tau Beta Pi, Engineering Honor Society

Eta Kappa Nu, Electrical Engineering Honor Society

Publications

Billy Kidwell, Jane Huffman Hayes. 2015. Toward a Learned Project-Specific Fault Taxonomy: Application of Software Analytics. In *Proceedings of the First International Workshop on Software Analytics (SWAN)*. IEEE Computer Society, Montréal, Canada.

Billy Kidwell, Jane Huffman Hayes, and Allen P. Nikora. 2014. Toward Extended Change Types for Analyzing Software Faults. In *Proceedings of the 2014 14th International Conference on Quality Software* (QSIC '14). IEEE Computer Society, Washington, DC, USA, 202-211. DOI=10.1109/QSIC.2014.10 http://dx.doi.org/10.1109/QSIC.2014.10

Davide Falessi, Bill Kidwell, Jane Huffman Hayes, and Forrest Shull. 2014. On failure classification: the impact of "getting it wrong". In *Companion Proceedings of the 36th International Conference on Software Engineering* (ICSE Companion 2014). ACM, New York, NY, USA, 512-515. DOI=10.1145/2591062.2591122 http://doi.acm.org/10.1145/2591062.2591122

Billy Kidwell. 2011. A decision support system for the classification of software coding faults: a research abstract. In *Proceedings of the 33rd International Conference on Software Engineering* (ICSE '11). ACM, New York, NY, USA, 1158-1160. DOI=10.1145/1985793.1986028 http://doi.acm.org/10.1145/1985793.1986028