TECHNIQUES AND TOOLS FOR MINING PRE-DEPLOYMENT TESTING DATA

by

Brian Yan Lun Chan

A thesis submitted to the Department of Electrical and Computer Engineering

In conformity with the requirements for

the degree of Master of Science (Engineering)

Queen's University

Kingston, Ontario, Canada

(September, 2009)

Abstract

Pre-deployment field testing in is the process of testing software to uncover unforeseen problems before it is released in the market. It is commonly conducted by recruiting users to experiment with the software in as natural setting as possible. Information regarding the software is then sent to the developers as logs. Log data helps developers fix bugs and better understand the user behaviors so they can refine functionality to user needs. More importantly, logs contain specific problems as well as call traces that can be used by developers to trace its origins. However, developers focus their analysis on post-deployment data such as bug reports and CVS data to resolve problems, which has the disadvantage of releasing software before it can be optimized. Therefore, more techniques are needed to harness field testing data to reduce post deployment problems.

We propose techniques to process log data generated by users in order to resolve problems in the application before its deployment. We introduce a metric system to predict the user perceived quality in software if it were to be released into market in its current state. We also provide visualization techniques which can identify the state of problems and patterns of problem interaction with users that provide insight into solving the problems. The visualization techniques can also be extended to determine the point of origin of a problem, to resolve it more efficiently. Additionally, we devise a method to determine the priority of reported problems.

The results generated from the case studies on mobile software applications. The metric results showed a strong ability predict the number of reported bugs in the software after its release. The visualization techniques uncovered problem patterns that provided insight to developers to the relationship between problems and users themselves. Our analysis on the characteristics of problems determined the highest priority problems and their distribution among users.

Acknowledgements

I would like to thank my professor Dr. Ying Zou for her guidance in the research and for the opportunity to undertake this study. I would also like to thank Dr. Ahmed Hassan for his suggestions throughout the course of my work which helped clarify my direction.

Special thanks goes to Dr. Bram Adams for his tireless feedback regarding the material that I have written. The quality of the content would not be the same without his advice. I am also grateful to Anand Sinha for providing the opportunity to study at RIM so I may gather better data for my study.

I would also like to thank the entire Software Engineering Group for their constant support and good companionship throughout my time here. Thanks to Lionel, Derek and Jin for their collaboration in certain papers and help.

Finally I want to thank my brother, who encouraged me to undertake this endeavor in the first place. Also to my parents who have always wanted me to do this.

Table of Contents

List of Figures

List of Tables

List of Equations

List of Abbreviations

FAR	Failure Accumulation Rating
IFFP	Initial Failure Free Period
OFR	Overall Failure Rating
FMR	Failure Memory Rating
MTBF	Mean Time Between Failures
API	Application Interface
PSD	Problem Signature Distribution
CC	Cool-down constant
FIC	Failure Increase Constant
FMV	Failure Memory Value
CPN	Colored Petri Net
COTS	Commercial off the Shelf
UX	User Experience
GUESS	Graph Exploration Systems

Chapter 1
Introduction

1.1 Background

Field testing of software applications in pre-deployment is an important and necessary step before its release into the market. Field testing helps uncover problems due to unexpected user behaviors and unforeseen bugs in a natural setting. Monitoring software for the application is used in field testing to capture real time information in order to generate logs. This enables the collection of field data without the developer's interference. Logs are eventually generated by the monitors and sent to a central repository to be analyzed by developers to find and resolve significant problems. Common information in logs includes the reporting users, error messages and call traces. Log analysis is meant to improve applications before its release. Despite the usefulness of these logs, developers tend to rely on post deployment data to handle problems. They wait for bug reports to be reported by customers, which delays problem resolution and lowers the overall quality of the software.

A common problem in analyzing logs in any stage of deployment is harnessing useful data from its bulk. Information reported may not become apparently useful until it is considered with the broader scope of all the logs. In some cases, developers focus on problems reported by specific users or ones that are more recently reported as in situations described by Musa *et al.* [39]. The narrowness of their scope makes it difficult to ascertain how the average user feels about the application since they cannot assess the impact different problems have for all the users. For example, there may be similarities in problem patterns across multiple logs which indicate serious bug that is overlooked because developers are too focused on analyzing isolated problem cases. Also if using Musa's approach, developers may be unaware whether certain problems are

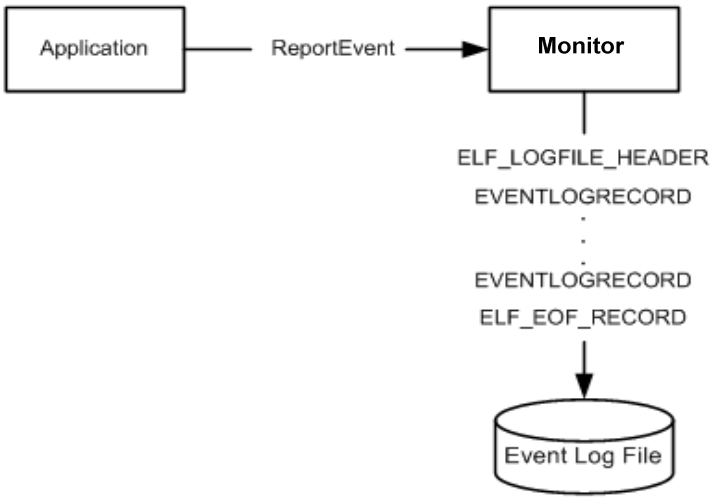


Figure 1-1. Log generation example

isolated to specific users or are reported by the majority of the population. This leads to speculation as to which problems have the highest priority. It also makes them unable to determine how easily it would be to replicate a problem among all the users who used the application. This means the information gathered from logs is not used to its full potential. In the following sub-sections we give an overview of the background.

1.1.1 Log Generation Overview

Logs files store user activity information that is generated by the applications. Figure 1-1 shows the process of capturing events from the application and creating a log. As shown in Figure 1-1, an application reports specific events that are captured by monitoring software. Monitors capture any event recorded such as keystrokes or functions accessed by the tester which is eventually combines into a log. In the figure, the messages shown represent the generic structure

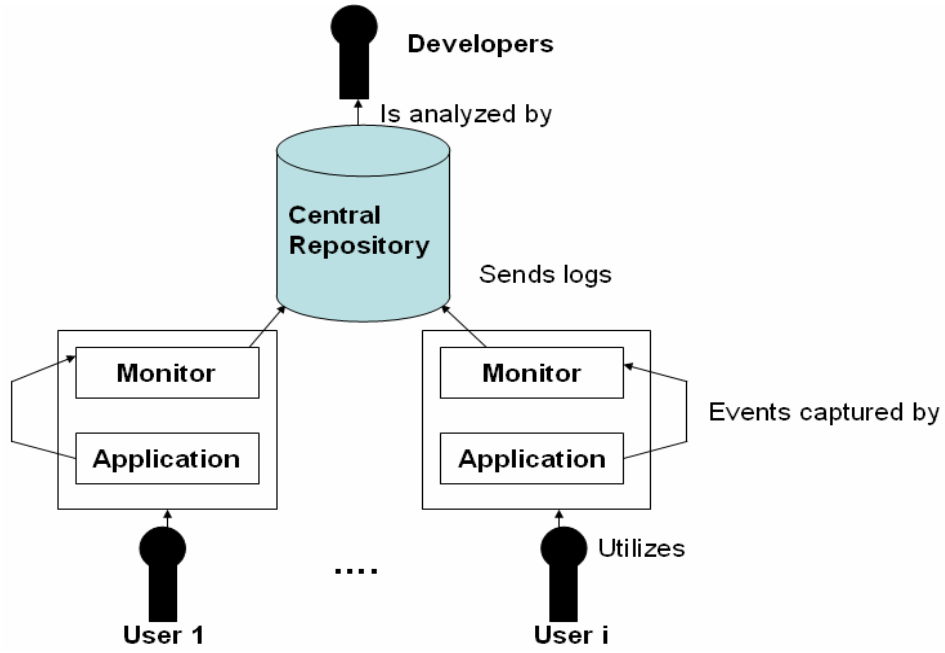


Figure 1-2. Field testing environment example

of the log. For example ELF_LOGFILE_HEADER denotes the starting point of the file, EVENTLOGRECORD is a specific event or problem report while ELF_EOF_RECORD represents the end of file. Usually monitors create a log after a certain size of data has been accumulated or if a serious problem has been recorded. Each log is a sequence of events that is written chronologically to when it happened.

1.1.2 Field Testing in Pre-Deployment

Field testing before the application is released into the market involves developers recruiting users to use the software application in as natural setting as possible. These users experiment with the functionality of the application as they see fit, while all their actions are internally logged as logs. Developers usually set a beta testing period where the testers can use their applications, whereupon at the end, all the log information is sent back to the developers in a central

repository. Figure 1-2 shows how log data is sent by users and passed on to developers for analysis. As depicted, each user has the same application where they experiment on. The application generates event information on any actions taken by the user as well as any consequent problems. This information is captured and stored by the monitor until sufficient information has been accumulated to create a log. The most important information needed from field testing is the sequence of events leading to a reported problem. This allows developers to trace its origin and get a better idea of how to resolve it.

1.2 Problem Definition

In the current field testing environment, there are several limitations to the analysis process that reduces the value of the log data. We highlight the major issues found:

1) Unable to accurately predict the user perceived quality in pre-deployment:

Field testing of software application is an important and critical step before the wide release of a software application. Metrics such as the Mean Time Between Failures (MTBF) as described by Amari *et al.* [3] are commonly used to quantify the quality of a software application and to measure the progress of field testing. Decisions such as the duration of beta testing and product release dates are influenced on metrics like MTBF.

However, we believe that a single MTBF metric fails to capture the user perceived quality in software applications before release. Developers are unable to accurately predict the disposition of customers to their software would be once put into the mainstream. A good understanding of occurrence of faults is needed before release in order to predict how well the software will do once it is available to the customers. For example, two users test two different applications. User 1 encounters a large number of failures at the beginning of his beta testing period while User 2

only encounters problems at the last stages. Even if both users encountered the same number of problems, they did not experience it at the same time period and would have different opinions on the quality of software. This distinction could not be found simply by using standard metrics like MTBF.

2) Data from users is analyzed individually without assessing the potential problems that affect the entire user base:

In the current state of practice such as works by Schwab *et al.* [50], developers may examine problems individually without a global view on the relation between the different problems and the relation between the users reporting these problems. For instance, it may be the case that a particular set of problems co-occur frequently together; therefore studying and resolving any one of these problems might lead to the resolution of all these problems. Also several users can report the same set of problems (i.e., share the same problem profile); therefore recruiting a few of these users to verify any fixes would be a faster option instead of deploying the fix across the field blindly and awaiting problem reports. It may also be the case that a large number of problems are reported, however very few of these problems have a wide impact on many users. Fixing problems with high impact is usually a high priority effort.

3) The priority of problems and the suitability of users to replicate them cannot be predicted by developers:

In the current state of testing, developers cannot accurately assess the priority of different problems reported by users since every developer has their own set of criteria such as complexity defined by Harrison *et al*. [27]. Consider a situation where an application used by users report Problem A and Problem B. Problem A is reported with a high frequency but will have negligible effects on a user that uses the application on a day to day basis. Problem B is reported only very

rarely but when it occurs, the application crashes for the users. Developers can easily replicate the situation causing Problem A and fix the problem but Problem B cannot be replicated because it happened infrequently and the users recruited could not replicate it. Since Problem B has more severe consequences and harder to capture, some developers may consider it higher priority but others may consider Problem A to be more important due to its frequency. Another serious issue associated with problem priority is the potential inability to recruit a proper user to replicate the problem. Even after identifying the top priority problem signatures, developers would need to know the proper users that report it. However all users report a different set of problem signatures developers cannot assume a user who reports problem A will report problem B. Therefore, developers must be able to recruit the proper user for each problem.

1.3 Objectives

The thesis aims to provide techniques to analyze log data in bulk in order to maximize information extracted from field testing. The objectives of the thesis are described as follows:

1) ***Attempt to establish user experience metrics to predict the perceived quality in software during pre-deployment.*** We attempt to design metrics in order to give a more accurate assessment of the software quality before its release. This is done to determine whether the disposition of users would be positive or negative if the software was released in its current state. Ultimately this should help developers assess whether the best release date for the product based on the user disposition. The purpose for designing multiple metrics is to capture different types of user perspectives when they utilize the software. To ensure they show unique information, the metric results should have low correlation. We intend to show that the metrics can assess the user disposition accurately by showing a strong correlation between the metric results and post

deployment results such as 1. Metric scores to bug report data for the software and 2. Metric scores to usage periods for the software.

2) ***Explore the use of graph visualization for easy identification of problem patterns.*** Visualization is a useful technique for abstracting vast quantities of low level information into identifiable patterns. We aim to provide a visualization technique that will allow developers to view low level data like problems and users that report problems in a visual format. Our main objective is to help developers identify patterns between problems and users that may not be apparent when analyzing problems at a lower level. We also hope to potentially find patterns that have generic characteristics within the visualizations to help filter user log data. By visualizing the field testing results across multiple product releases, developers can compare the progress of field testing efforts for different releases.

3) ***Assess the validity in visualizing the file trace of problems to find files of interest.*** Many problems are considered different due to the varying call trace sizes and files accessed prior to the problem report. It may also be the case however that the different call traces will have accessed the same files at a certain point of their execution. We attempt to provide a visualization technique that will allow developers to see the call trace of all their users at once. The main objective of creating this technique is to help developers identify files of interest among different call traces. By identifying common files accessed between different call traces, developers may get a better idea on how to isolate problems.

4) ***Devise techniques to prioritize problems and identify users for replication.*** Problems can be rated by priority according to its frequency of occurrence among all the user logs. While this is a useful characteristic, it does not consider other important factors such as how distributed it is among user logs and the pervasiveness among users themselves. We consider problems that are frequently reported by few people to be lower priority that problems reported by a large variety of people with comparable frequency. Using this argument, we attempt to develop a measurement technique that utilizes entropy of a problem to assess its priority. The objective of this technique is to give the highest priority to problems which are reported by the most users and in a distributed fashion. We also attempt to create a visualization scheme that will highlight the users that report these highest priority problems. The objective of this visualization is to eliminate guesswork by developers as to which users they should recruit to replicate a specific problem.

1.4 Organization of Paper

The remaining chapters of this thesis are organized as follows:

- Chapter 2: We review related work about log analysis, previous metrics used to assess software quality, bug isolation, pattern recognition, entropy and complexity in systems, and user behavior analysis.
- Chapter 3: We describe in detail the metrics that were designed, including what component of user experience they were meant to measure. We conduct a case study to assess the independence of the metrics to each other, their ability to predict the number of bug reports issued after post release, and evaluate the user perceived quality by correlating the software usage period to the metric scores. In addition a discussion on the threats and limitations

regarding its usage is included.

- Chapter 4: We discuss the process of how to create the visualizations using log data and then describe in detail the components of each graph type. In addition we also discuss the criteria for weighing and filtering in the graphs. We also conduct a case study to identify useful patterns extracted from each graph type and their significance to developers.
- Chapter 5: We describe how to recombine the structure of problems in logs to identify common files to find the origin of a problem. We also identify patterns in the entire call tree to describe characteristics of problem structures. A case study is conducted to show the patterns identified in the file structure visualization and how they characterize different problems.
- Chapter 6: We define the definition of entropy as well as what constitutes a problem by developer standards. We also discuss the definition of cluster entropy and individual entropy and the process of generating the results. We conduct a case study to show statistical and visual evidence of the effectiveness of our analysis technique in identifying the top priority problems.
- Chapter 7: We draw conclusions from our work and discuss future work to be conducted.

Chapter 2
Related Work

2.1 Field Data Collection

Evaluation of User Perceived Quality

User perceived quality of software pertains to the disposition of the user and how favorably they regard the software after using it for a given amount of time. The usability of an application is evaluated using five attributes[48][57]: (1) learnability, which measures the ease of learning the functionality of an application; (2) low error rate, which measures the number of mistakes that users make while using the application; (3) memorability, which measures the ease of remembering the functionality of the application; (4) efficiency, which measures the ease of use and the level of productivity that the users of the application can attain; and (5) user satisfaction, which measures the enjoyment of the users who are using the application. Our work is concerned with the perceived quality of an application. It relates most closely to the 5^{th} attribute, i.e. user satisfaction. Users are very satisfied with applications which rarely fails, have limited footprint, and do not consume too much battery.

Similar to our work, Gonzalez *et al.* [22] condenses the vast amounts of field information into metric scores to evaluate usability of mobile users using data mining techniques. Reliability concerns the maximum failure rate of an application. It is an important factor to determine user satisfaction. Studies, such as Cheung [13], argue that reliability of a system is in effect a reliability analysis of different modules or (nodes) and detailing their individual success rates. A sensitivity analysis technique is developed to assess this effect. Everett [20] takes a similar approach to Cheung, stating the modularity of programs during testing should be maintained. Each component should be stressed individually to assess their reliability before superimposing

the components for final assessment. Our approach in contrast relies on observing all the components being executed together to help optimize the software. Mockus [40] applies a similar approach to our study by using logging information from the system. Mockus defines reliability as the amalgamation of three aspects: total runtime, number of outages and the duration of outages. A framework is proposed to assess system reliability using information readily available in most systems such as bug fixes issued and alarm information. We use similar concepts such as number of outages and duration but rely more on observing the failures to assess reliability. Buckley *et al.* [9] argues that user satisfaction and service delivery directly affect each other. They measure service variables and customer attributes to show that the return user satisfaction (and thereby investment) is exponential to the amount of effort applied by developer fixes. We try to reduce the overhead (i.e. developer fixes) by resolving as many problems as possible during pre-deployment. Current approaches to collect the data for evaluating user perceived quality can be categorized into four classes [21]: 1) direct observation, 2) laboratory methods, 3) self-report, and 4) automatic logging. Our technique is most connected to automatic logging. Each class offers different types of data for evaluating the usability. In contrast to our technique, direct observation as mentioned by Schwab *et al.* [50] provides rich qualitative information for the usage and user reaction. However, it can only be applied to a small number of participants at a time. The results are also subject to the interpretation of the observer. Laboratory methods offer an environment that rigidly controls settings and configurations for experimentation such as works proposed by Schmidt [49]. The artificial setting and usage are not representative of the natural context and are hard to generalize. Self-reporting survey mechanisms such as those described by Hochstein *et al.* [29] rely on users to manually submit on-line reports to gather user's feedback on the software. This is not dissimilar to our automatic logging technique described in chapter 1. However unlike our technique, many users often do not submit

the survey. While our logging does not always successfully report the data back, it would have a higher success than online report. To gain a realistic understanding of the usability of mobile software, data must be collected from users in their actual setting. Automatic logging also as described by Hochstein *et al.* [29] (i.e., tracing, profiling and instrumentation) mechanisms are used to record usage information. These mechanisms scale well across users and collect large amounts of data without requiring the intervention of users. For example, INGIMP, a standalone application, instruments the graphic user interfaces of open source applications, collects real-time usability data from open source applications and submits all the data to a website for analysis [52]. HUIA developed by Au *et al.* [6] evaluates the static and dynamic properties of the user interface, and recommends methods of improvement. Another automatic logging technique is proposed by Musa [39]. Similar to our approach, different users exercise different subsets of the features provided by an application. User profiles can be built by instrumenting the infield use of an application. These profiles can be used to compare different users using various similarity and dissimilarity metrics (e.g. [36] and [18]). Using this knowledge one can reduce the number of needed test cases and determine similar user groups. When studying large distributed software applications, detailed instrumentation is not possible due to its high overhead. Such detailed instrumentation consumes too many resources and produces a large amount of data which is challenging to transmit back to central repositories for further analysis. In our work we make use of already-collected data, the in-field problem reports, to derive an approximation of the operational distribution of an application – its problem profile. In contrast to an operational profile, users with different usage patterns might not have the same problems.

Christmansson et al. inject software errors directly into an application and observe the outcome [14]. Such fault injections approaches could be used to verify the soundness of the

clusters in our visualization by checking if a representative user of a cluster would exhibit similar problems to other users in the clusters.

2.2 Pattern Recognition and Identification of Bugs through Visualization

Visualization is a tool to abstract large amounts of data and condense it into a useful representation to find information that could not be seen at low levels. Visualization has two primary categories of use in software systems: pattern recognition and bug isolation. Pattern recognition involves finding relationships between components that may provide insight into internal workings or better understanding a problem. We use this approach primarily to identify patterns between problems and users. The majority of bug isolation visualization involves looking at lines of code from a high level view and highlighting potential sections that caused the error based on the description of the problem. Our bug isolation technique is similar but rather than isolating sections one at a time, it involves linking the sequences of code that would eventually cause the problem itself.

Pattern Recognition Visualization Techniques

Visualizing relationships between components is often employed through the use of edges. Prior work which visualizes relations between software artifacts (e.g., Harald et al. [26]) employs basic threshold filtering techniques. For example, only edges above a specific threshold are shown to reduce clutter and avoid over-plotting, in contrast our approach uses a statistical filtering algorithm. We apply a similar technique to filter edges so that only statistically significant edges remain within the visualization graphs. The statistical filtering is possible due to the large size of the data used in our case study.

Rozinat *et al.* [46] use a visualization scheme to map relationships between components in a simulation model, called Colored Petri Net (CPN). Similar to our tree node approach, their models can map a sequence of events (i.e. call trace data) into visual format. However, additional data, such as time and resources can be annotated to the model. The results of our technique could potentially be enhanced using CPN to visualize the call traces.

Various approaches are used to study and compare different profiles. Dickinson et al. use cluster analysis techniques [19], while others like Orso et al. [43] and Jones et al. [32] use visualization techniques. Approaches similar to Dickinson et al. study the relation between different operational profile (e.g., users) using mathematical techniques. While approaches such as Orso et al. and Jones et al. study the similarity of profiles by mapping them to the Lines of Code and visualizing the relation between the different lines to identify the location of faults. In contrast, our approach visualizes relations at a higher level of abstraction, at the level of field problems and users instead of lines of code. Our analysis could be done at the level of lines of code; however this would require a more detailed instrumentation of the application which is not feasible for application during field-testing.

In our work, we make use of spring-based layout algorithms to produce clusterings of field reports and users. There is a large and active area of research in the use of clustering techniques to understand software artifacts [31, 41, 53, 56, 55, 44]. In contrast, our work is primarily visual whereas the aforementioned techniques produce textual output representing the different clusters. To produce such clusters users must specify cut-off values for the clustering algorithm. Our work makes use of a spring-based layout algorithm to perform visual clustering. The visualization aspect of our work helps users in exploring various cut-off values interactively. Using the cut-off values, textual representation of the clusters could be produced from the graphs using commands similar to our pattern identification scripts.

Bug Isolation Visualization Techniques

There have been various attempts to visualize and study large scale systems to identify bugs. While our technique observes tries to observe bugs at a higher abstracted level, previous studies tend to focus at low level components. For example, Liblit *et al.* [37] uses instrumented predictors to observe code segments and report failures if the program fails. They also introduce an assertion statement framework [38] the code to track down faulty software. Hovemeyer *et al.* [30] isolate bugs by creating a bug pattern detector. The detector analyzes the code structure and determines which sections match known faulty patterns. These techniques could help us pinpoint the source of a bug based on files that are flagged by our visualization. In terms of real time visualization of bugs, Orso *et al.* [42] have developed a tool called GAMATELLA which allows users to manipulate execution data in an interactive fashion by providing functions such as instrumentation and treemap nodes. This tool could potentially help our study fix errors in real time by injecting sample code into a call path with a problem façade and visualize the results.

2.3 Entropy and Complexity of Systems

Entropy has seen a wide variety of usages in relation to software. Shannon entropy is defined in this paper for software, but is more commonly associated with signal processing [51]. Current approaches to using entropy in software systems are divided into two major categories: analyzing software complexity and testing software system security. In contrast, we employ entropy to assess the priority of different problems discovered in software. Software complexity as described by Hassan *et al.* [28] relates to the cost of maintaining a piece of currently implemented software. This involves the effort and time required to perform maintenance or changes in the code itself while measuring the impact to the whole system. This concept is

similar to our use of entropy, as higher complexity will affect more components while high priority problems will affect a greater number of users. Hassan *et al.* [28] attempted to show that a development process that has a higher rate of entropy negatively impacts the quality of the software produced. They take a set of files modified within a project and determine the probability of each file being modified. File modifications that are done at a time of high complexity results in a greater number of faults introduced in the system. Their study could be potentially useful in determining whether there is a correlation to the priority of problems we discover and the number of modifications done to the files that produce the problems. Bianchi *et al.* [7] defines the entropy as the number of links between components. A software system is composed of separate components which are connected through relationships or links. Their study attempts to show a direct correlation between the number of links among components to the number of defects and maintenance effort within the system.

Hafiz *et al.* [25] treats a system as an information source and uses entropy to extract different types of information. Specifically the study promotes the theory that files which are more functional and descriptive provide a larger amount of entropy. They apply Shannon, Hartley and Renyi entropy measures [51] as part of their case study. However since they measure average information content, the module size do not affect their measures. Kim *et al.* [33] attempts to define new software complexity metrics (class complexity and inter-object complexity) based on the traditional Shannon entropy. Their research specifically targets object oriented systems. Harrison *et al.* [27] believes that the complexity of a program is inversely proportional to the information content of its operators. They believe a program is a set of symbols and if a larger set of symbols is present, then on average the chance of them appearing decreases. The greater variety of symbols creates greater entropy, hence greater complexity. This is similar to our concept in assessing priority of problems in the sense that a problem with a

higher number of users reporting it invariably creates a higher entropy and hence higher priority. Chapin *et al.* [11] contends that entropy metrics are readily available for any system and by observing any abrupt chances in the entropy of a system, insight can be gathered as how maintenance of the software should be performed. [12] also by Chapin brings an interesting perspective on integrated systems. He believes many of the Commercial Off the Shelf (COTS) are integrated with a variety of reusable software and miscellaneous components to form new software systems. To this effect he proposes an entropy based metric to show how hard it would be to maintain such as system. It would be interesting to see if the highest priority problems identified in our studies are influenced by COTS software and whether in-house source code is inherently better quality.

2.4 Chapter 2 Summary

In this chapter we discussed work on techniques that are related to the fields we implement in subsequent chapters including metrics, visualization and entropy. In the next chapter, we introduce the metrics that quantitatively measure the user perceived quality of applications using user log data. Subsequent chapters discuss the techniques to find visualize patterns in user logs and how to find the priority of problems from user logs respectively.

Chapter 3
Using Metrics to Predict User Perceived Quality in Software after Release

Metrics such as the Mean Time Between Failures (MTBF) can be useful in quantifying the quality of a software application and to measure the progress of field testing. They measure characteristics such as the distribution of failures in a time period and their frequency. Metrics give an indication between the state of the software and the user perceived quality. For example, decisions such as when to stop the field testing and when to release a product may be influenced by metrics like MTBF. Current metrics do not cover all the characteristics of failures within a time period and therefore are less accurate in indicating user perceived quality. We propose four new metrics that compensates for the lack of information in certain areas of failure analysis. Using these metrics, developers can realize the state of failures if the software was released in its current form. While they cannot predict the exact user perceived quality they can get an indication of whether the feedback would be positive or negative.

3.1 User Quality Experience Metric Definitions

For all metrics, a high score is desirable over a lower score. We summarize the proposed metrics in Table 3-1, as well as the traditional MTBF metric which tracks the average time between the occurrences of a failure across all users of an application.

Table 3-1. Summary of the Proposed Metrics

Metric Name	Description	Interpretation of Score
Mean Time Between Failures (MTBF)	The average failure free period of a user in between failure events.	High scores indicate that the average user has a long failure free period in between failures
Initial Failure Free Period (IFFP)	The average number of days before a user encounters the first failure.	High scores indicate that there is a long period before the average user encounters the first failure thereby improving their impression of the application
Failure Accumulation Rating (FAR)	The number of failures that the majority of user for an application encounter	High scores indicate the majority of users report a small number of failures
Overall Failure Rating (OFR)	The percentage of users that report a failure on any given day	High scores indicate a low percentage of users report failures on a given day. Scores are lower if the average percentage is higher
Failure Memory Rating (FMP)	The amount of undesirable events that is remembered by the user.	High scores indicate the average user hardly remembers any undesirable events (i.e., failures). Low scores mean they remember many undesirable events

3.1.1 Initial Failure Free Period (IFFP)

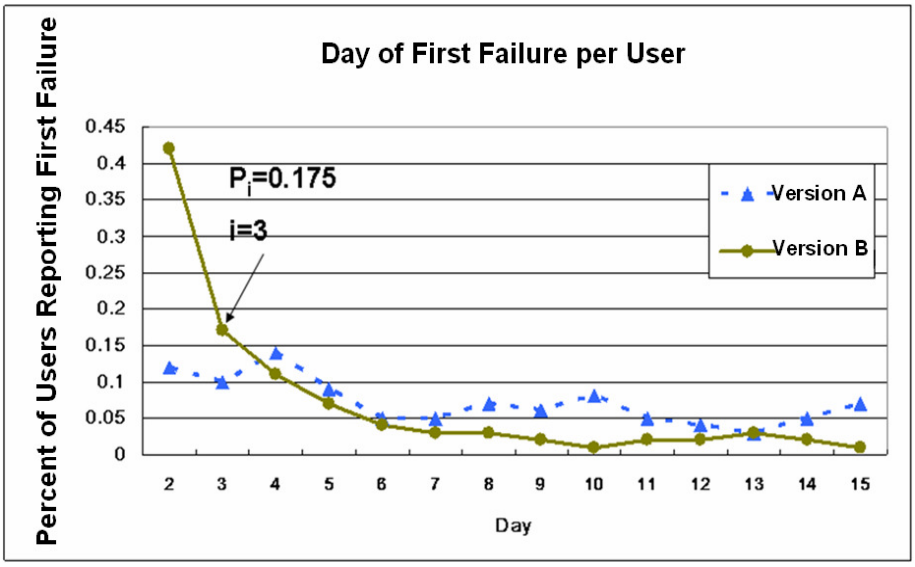


Figure 3-1. Sample data input for Initial Failure Free Period metric

The Initial Failure Free Period (IFFP) metric measures the average amount of time before the user faces the first failure. It is desirable that users use an application for a longer period of time before they exhibit their first failure. The overall perceived quality of an application is lower if the majority of users of a software application encounter failures earlier during their period of usage rather than later. In any case, a large number of failures encountered early on can be detected by users during the field testing period. If the number of bug encountered initially is significantly larger than anticipated, developers can push back the release date accordingly. We define the IFFP metric as follows (1).

$$IFFP = \sum_{i=1}^{i=T} P_i \times i \qquad \textbf{Eq. (3-1)}$$

T represents the length of the testing period. i represents the i-th day in a testing period. The values of i ranges from 1 to T. P_i refers to the percentage of users that reported a failure on the i-th day.

The IFFP metric calculation accounts for the first failure for each user across all field testing users. Subsequent failures for the same user are not considered. It is designed for the relative simplicity in calculating a score based on a line trend. By knowing if the y value is high earlier in the usage period or later, the equation can determine a score with little information. Figure 3-1 shows the sample data used as input for calculating the IFFP metric. The x-axis represents the days over the field testing period. The y-axis is the percentage of the users reporting the first failure on a specific day. P_i refers to the percentage of the users who report their first failure on the i-th day. A higher percentage of users reporting the first failure late in the testing period produce a higher score of the FI metrics. Low scores indicate that users of this application after release are likely to encounter problems early. Therefore, an application that commonly reports its first failures in the early days of the testing period will exhibit low scores. For example as depicted in Figure 3-1, the version B trend illustrates high values of P_i during the early days in the testing period. This creates a low Initial Failure Free Period score since the number of day, i, in Eq. (3-1) has a low value initially. However version A has a better distributed trend. The P_i of the version A series is lower than the version B series at the beginning but is consistently higher in the later stages of the testing period when the number of day progressively increases.

3.1.2 Failure Accumulation Ratings (FAR)

The Failure Accumulation Rating (FAR) metric analyzes the distribution of failure frequency to identify where the majority of users fall into in terms of total failures. Figure 3-2 depicts an example graph to show the distribution of the majority of the users. The x-axis represents

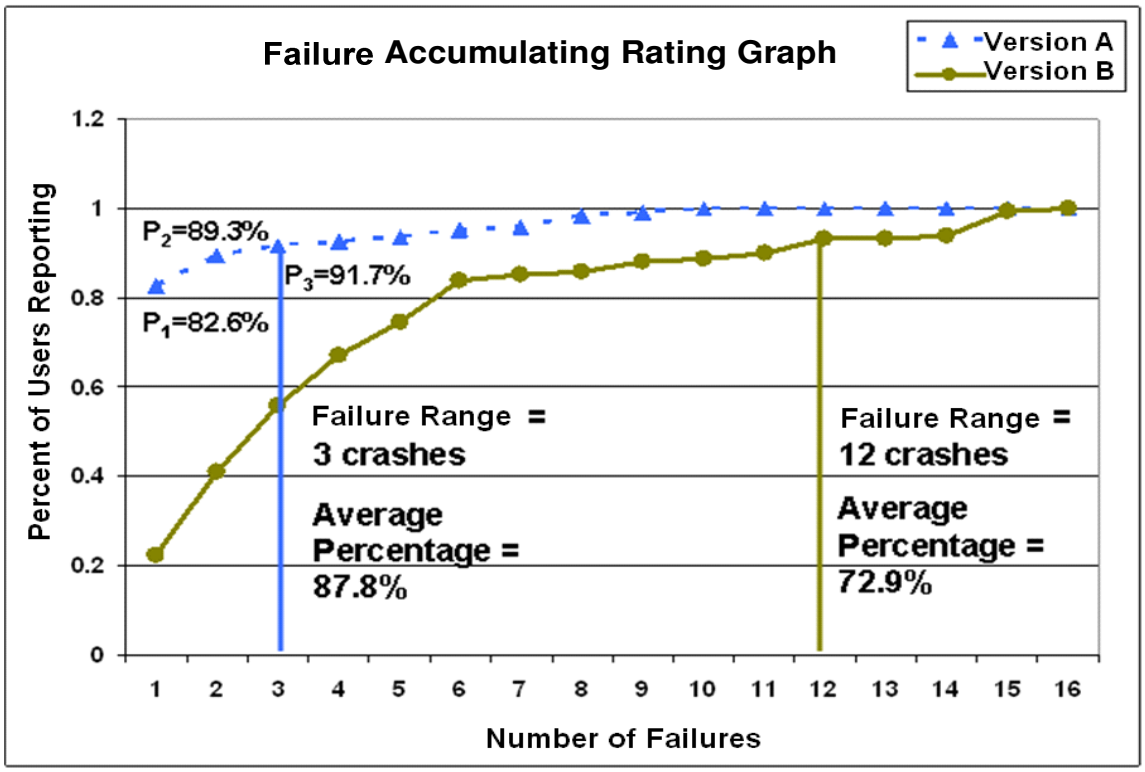


Figure 3-2. Sample data input for Failure Accumulation Rating metric

the number of failures. The y-axis shows the cumulative percentage of users that report a certain number of failures for the entire testing period. i refers to a certain number of failures. p_i corresponds to the accumulative percentage of users for i failures. For example, the version B trend shows approximately 20% of users report 1 failure during the entire testing duration (i.e., p_1=20%). Roughly 20% of users report 2 failures which might be the repetition of the same failure. Since the graph is cumulative, the percentage of users that report 2 failures or less is 40% (i.e., p_2 = 40%). We define a failure range as the number of failures reported by the majority of users. We consider 90% or more of users as the majority of users. For the example shown in Figure 3-2, 82.6% of users reports 1 failure (i.e., P_1= 82.6%). 89.3% of users report 1 or 2 failures

(i.e., P_2=89.3%). 91.7% of the users report 1, 2 or 3 failures (i.e. P_3 = 91.7%). Therefore, more than 91% of users (i.e., the majority of users) report no more than 3 failures. The failure range is 3 in this example. We use Eq. (3-2) to calculate the average percentage of users who report the number of failures within a failure range. For the example shown in Figure 3-2, the average percentage of users reporting no more than failures is 87.8% (i.e., $\frac{82.6\% + 89.3\% + 91.7\%}{3}$) in version A.

It is desirable that the majority of users report a small number of failures. It means that the bulk of the distribution is concentrated to the left of the distribution graph (e.g., Figure 3-2). The FAR metric measures the failures occurring to the majority of users over the failure range. The definition is specified in Eq. (3-3). FAR computes the inversion of the failure range value multiplied by the average percentage. FAR metric produces high scores to an application where the majority of users report a very low failure range. For the example shown in Figure 3-2, version A shows a desirable FAR trend because most of the users report failures in the left side of the graph. Conversely, if a large proportion of users report a high number of failures, the failure range will subsequently increase, resulting in a low FAR metric score. For the example shown in Figure 3-2, version B has more users tending to report a large number of failures. The trend of version B indicates a poor failure behavior among users, and therefore produces lower score of FAR metric score. The metric was designed in a way that would be easy to calculate if the failure distribution for a set of people was easily obtainable. The Average_Percentage variable was formulated because it was desirable for a majority of users to be reached as soon as possible (i.e. with a failure range as small as possible). Also having a metric that rates user perception inversely proportional to the number of failures is intuitive as well.

$$Average\_Percentage = \sum_{i=1}^{i=failure\_range} \frac{p_i}{failure\_range} \qquad \textbf{Eq. (3-2)}$$

i represents the number of failures. p_i is the cumulative percentage for the failure number i. Failure_Range refers to the number of faiures reported by a majority of users. Average_Percentage is the average cumulative percentage over the Failure_Range.

$$FAR = \left(\frac{1}{Failure\ \_\ Range} \times 100 \right)^{Average\ \_\ Percentage} \qquad \textbf{Eq. (3-3)}$$

Failure_Range refers to the number of failures reported by a majority of users. Average_Percentage is the average percentage of users who reports the failures no more than the value of Failure_Range.

3.1.3 Overall Failure Rating (OFR)

The percentage of users that report failures each day can portray the Overall Failure Rating from users of an application. A lower percentage of users reporting unique failures daily would indicate a generally positive perceived quality when released to market. Figure 3-3 depicts example results of analyzing the percentage of users reporting unique failures over the testing period. The x-axis tracks the days over a testing period. The y-axis represents the percentage of users that report unique failures in a given day. The y-axis can be equally divided into different ranges by incrementing a certain percentile (e.g., 10%). We use a percentile index (i.e., i) to annotate each percentile starting from the index 1. For example, i=1 represents the failure percentage in the range of 0-10%. Similarly, i=2 represents the failure percentage in the range of 11-20%. i has a maximum value of 10 in this example. This maximum value could change if the failure ranges are altered.

$$w_i = n - i \qquad \textbf{Eq. (3-4)}$$

n is the total number of percentiles plus 1. $i \in [1, n]$ is the percentile index.

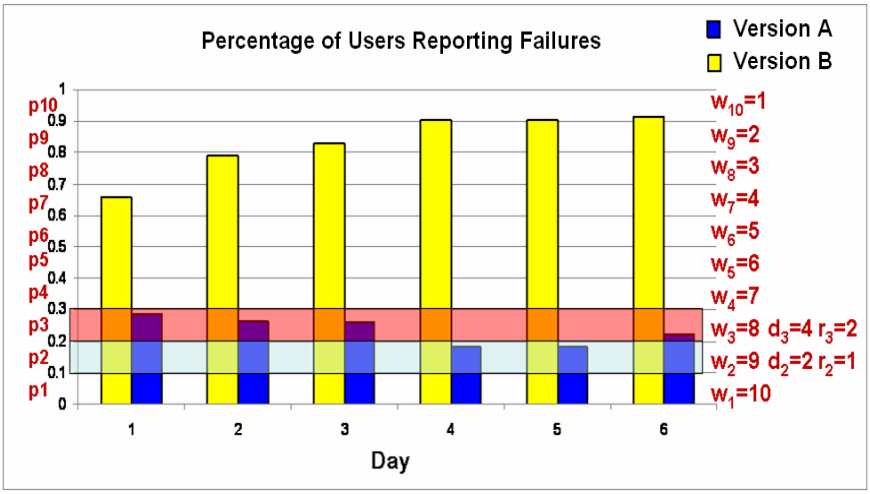


Figure 3-3. Sample data input for Overall Failure Rating metric

$$OFR = \sum_{i=1}^{i=n-1} \frac{w_i \cdot r_i \cdot d_i}{total\_days} \quad \textbf{Eq. (3-5)}$$

i represents percentile index. n is the maximum value of percentile index. total_days is the total number of days in the testing period. w_i *is the weight of a percentile,* d_i *is the count of days falling in a percentile range.* r_i *is the ranking of* d_i *in* $D=\{d_1, d_2, \ldots, d_i, \ldots, d_n\}$.

It is desirable to have a low percentage of failures each day. In other words, it is desirable to have the percentage located in the lower percentiles. To indicate a better user experience, we assign a weight (i.e., w_i) to each percentile as defined in Eq. (3-4). The highest weight is given to the percentile, p_1. The weight of a percentile decreases by one as it is further away from the percentile p_1. For example shown in Figure 3-3, the p_1 has a weight of 10, and p_2 has a weight of 9. The lowest weight is assigned to the percentile, p_{10}.

It is favorable to have more days with the low failure percentage. We count the number of days when a failure percentage falls into a percentile, p_i. We denote the corresponding count of days using d_i. For example as shown in Figure 3-3, there are two days that the failure percentages fall into the range of percentile, p_2, therefore, d_2 is equal to 2. Similarly, d_3 is equal to 4.

Given a set of counts of days, $D=\{d_1, d_2, \ldots, d_i, \ldots, d_n\}$, we rank d_i in D in an ascending order, and record the ranking of d_i in D, using the ranking set, $R=\{r_1, r_2, \ldots, r_i, \ldots, r_{max}\}$. r_i represents the position of d_i in the ordered sequence. A higher value of the day, d_i indicates a larger the rank value of r_i. Percentiles without days reporting failures (i.e., $d_i = 0$) are not ranked. For example as shown in Figure 3-3, the ranking of d_2 (i.e., r_2) is 1 and the ranking of d_3 (i.e., r_3) is 2.

An ideal case is when the majority of the daily failure percentages are distributed within the range of the lowest percentile. It represents fewer users reporting failures overall, and therefore, the application gains a better Overall Failure Rating. The corresponding count of days, d_i, the ranking, r_i, and weight, w_i, have high values. A worst case is when the majority of the daily failure percentages are distributed within the highest percentile. It indicates a poor Overall Failure Rating of the application. Although r_i and d_i may have the same values in the worse case as the corresponding variables in an ideal case, the weight, w_i turn out to be low in the worst case. For example as illustrated in Figure 3-3, the Overall Failure Rating for version B is poor because the weight, w_i, for each daily failure percentage would be small. The opposite is version A which shows a consistently low failure percentage, therefore a good Overall Failure Rating. As a result, the Overall Failure Rating is associated with three major factors: the weight of a percentile (i.e., w_i), the count of days with the failure percentages falling into each percentile (i.e., d_i), and the rank of the count of days (i.e., r_i). As defined in Eq. (3-5), we compute the average Overall Failure Rating each day by taking into account of all users. The Overall Failure Rating metric makes full use of all the information provided by a beta testing period and is therefore more complicated than other metrics. The most intuitive variable is w_i since days with lower failure rates are always desirable. The variables r_i and d_i are necessary because in the event failure rates fall into very distinct percentiles, it should be clear which percentile carries the most influence.

Normally this could be achieved with the ranking variable r_i, however if the beta testing period became very large (i.e. quarter of a year), r_i may not be able to convey that meaning with so many days. The variable d_i is therefore used to make the score more precise. The combination of these variables allows the metric score to evaluate user perception more accurately for a given beta testing period.

3.1.4 Failure Memory Rating (FMR)

A user's memory with an undesirable event (e.g., failure) will dissipate over time. However, the frequent reoccurrence of the event would constantly remind the user of the bad experience. Therefore, the user's memory of undesirable events would remain high. The Failure Memory Rating (FMR) metric measures the impact of undesirable events that an average user can remember. A user's Failure Memory Rating is computed based on three concepts: Failure memory value (EMV), Failure increase constant (FIC), and cooldown constant (CC). The failure memory value is initialized to 0, when a user starts using an application. When a user encounters a failure, the failure memory value increases a constant value (i.e., FIC). If the user go through a period without another failure incident, the failure memory value decreases by a constant (i.e., CC) per day until the failure memory value reaches 0. However if the failures frequently occur near the end of the testing period, the failure memory value may become a large positive value. The constant values for failure memory increase and cool down can be manually specified based on user studies. Such user studies would need to explore the different types of failures and their impact on the memory of failure for users. For example, a simple failure that would not lead to a data loss might lead to small increase in failure memory value over a failure that would cause the user to redo their work. Nevertheless, using such constant values could be used to compare new versions of an application against each other.

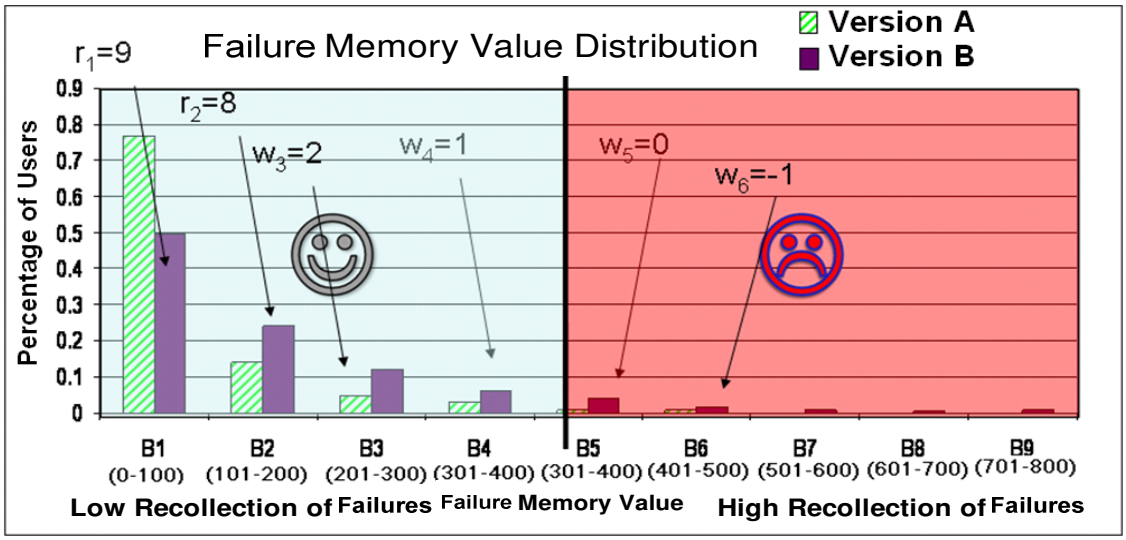


Figure 3-4. Sample data input for the Failure Memory Rating metric

$$CC = \alpha \times FIC \qquad \textbf{Eq. (3-6)}$$

CC is the cooldown constant, FIC is the failure memory increase constant and is always greater than 0. α *is the percentage constant and is specified as* $0 < \alpha < 1$.

CC is always specified as a percentage of FIC because it takes more time to forget an failure occurred than it does to encounter one. For example a user is more likely to remember a failure experience for a few days before the cooldown causes them to forget the event. Figure 3-4 depicts an example of failure memory values. The x-axis depicts the failure memory values. Smaller failure memory values (i.e., close to 0) indicates that a user hardly recalls any problems with an application. Users with a large failure memory value indicate they remember many problems.

To handle a large number of users who are associated with a failure memory value, we divide the failure memory values into a sequence of buckets (i.e., B_1, B_2, B_3, …, B_n) ordered from the left to right in the x-axis. The example buckets is shown in Figure 3-4. The y-axis shows the

percentage of users that fall into a specific bucket. It is worth to note that the failure memory value range in each bucket can be justified depending on the amount of users and ranges of failure memory values for a particular application. When most of the users report similar failure memory values, the failure memory value range for each bucket can be 1 in order to precisely capture the percentage of users falling into each buckets.

Buckets on the most left side of the x-axis (e.g., 0-100) show the most positive effect. The buckets on the most right side of x-axis show the most negative effect. To make our assessment of a user's failure memory value independent from the selected constants (i.e., failure increase constant and cool down), we distribute a weight w_i to bucket, B_i as shown in Eq. (3-7). We divide the buckets in the x-axis from the middle, and assign no weight to the bucket resident in the middle (e.g., w_5=0 shown in Figure 3-4). The weight of the buckets to the left is incrementally increased by one as the bucket is further away from the middle bucket. For example as shown in Figure 3-4, the weight for bucket B_1 is 4, and the weight for bucket B_4 is 1. The weight of the buckets to the right of the middle bucket is decreased by one as the bucket is further away from the middle. For example shown in Figure 3-4, the weight for bucket B_6 (i.e. w_6) is -1. Eq. (3-7) shows the definition of w_i.

$$w_i = \left\lceil \frac{n}{2} \right\rceil - i \qquad \textbf{Eq. (3-7)}$$

n is the total number of buckets. i is an index for a bucket, ranging from 1 to n. w_i represents the weight of bucket i.

Given a sequence of buckets, i.e., B_1, B_2, …, B_i, …, B_n, we rank them in an ascending order based on the percentage of users following in each bucket. We record the ranking of each bucket using a ranking sequence, R={r_1, r_2, …, r_i,…, r_n}, n is the number of buckets. The variable r_i represents the ranking of the bucket, B_j among all the buckets. For example as shown in

$$FMR = \sum_{i=1}^{i=n} w_i r_i \qquad \textbf{Eq. (3-8)}$$

n is the number of buckets; w_i is the weight for bucket; B_i, and r_i is the ranking for the bucket, B_i.

Figure 3-4, version B has users distributed in every bucket. r_1, the ranking of the bucket B_1, is 9. Similarly, r_2, the ranking of the bucket B_2, is 8.

It is desirable that most of the user percentages are distributed in the buckets with a low failure memory value. A higher percentage of users of the bucket create a strong ranking (for example in Figure 3-4, r_1=9). The bucket with a low failure memory value gives a high weight (i.e., w_1=4). On the contrary, the majority of users that are distributed in the buckets with negative weight would give strong rankings. However, high negative weight and strong ranking would reflect the dissatisfaction of the users. As a consequence, the weight (i.e, w_i) and the ranking (i.e., r_i) determines the user's satisfaction for the user. We compute the overall Failure Memory Rating for an application by considering the distribution of the users in all buckets. The Failure Memory Rating is defined in Eq. (3-8). The high value of the result is desirable indicating that the FMR is highly positive and the Failure Memory Rating is low. The Failure Memory Rating metric is designed with scalability in mind. A user's preference and inherent memory is largely dependent on the subject. That is why CC and FIC are largely dependent on the user's attitude, whether they are quick to anger or forget. Because the range of failure memory can be very flexible, the definition of the number of buckets, the CC, FIC and FMR are meant to reflect this. For example if the user review simply wanted a positive, neutral or negative report, the metric could simply set three buckets and calculate a very clear cut score. However if the users of the application are more nuanced and want a more accurate assessment of failure memory, the number of buckets could be increased.

3.2 Case Study for Evaluating Metrics

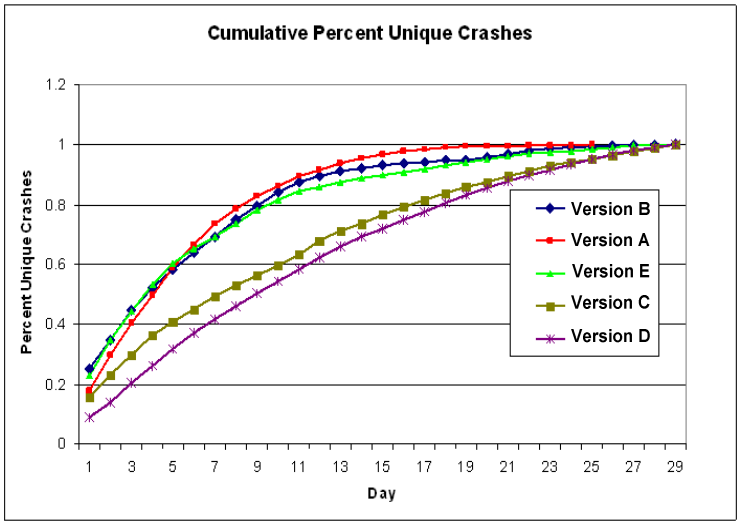


Figure 3-5. Growth Rate of Failures overtime for Each Version

Our case study sought to answer the following two questions: Whether the metrics were independent and whether the metrics were successful at predicting the user perceived quality for software. For our case study we used data derived from the field testing of over 21 versions of a mobile software application. 2,767 users participated in the field testing of these versions. As shown in Figure 3-5, we use data for the first 30 days of the field testing phase since our analysis of the growth rate of unique failures plateaus at around 30 days for most software versions. In the following subsections, we describe the challenges associated with the data collection, we detail the findings of our case study, and we discuss the threats to validity for our findings.

3.2.1 Challenges in analyzing field data

Due to the bulk of users actively sending application information, we face the challenges on processing the vast quantity of field data from various versions of the application. We discuss the challenges and our solutions as follows:

Crash duplication: Identical crashes can be submitted multiple times from an application within a short period of time. When an failure occurs, a crash report is sent. However, the user of an application may not be unaware of the causes of the crashes, and tends to try the same sequence of actions that trigger the previously reported crashes. Therefore, identical crash reports commonly exist in the repository. To filter the identical crash reports, we check the timestamp between each of the crashes to ensure that two crashes reporting the same failure are at least one hour apart and that the failures are distinct if two crashes are reported within one hour.

Hard to determine if a application is used every day: As aforementioned, we track the days of use for each application during the testing period. However, it is challenging to determine when the application is in use. The check-in reports submitted by an application indicate if the application is in use in the field. However, not every application submits daily check-in reports starting from the first day of their use of the application. There are many reasons to explain the absence of activity in using a particular model: 1) a user may block the functionality of submitting daily check-in reports. A crash could be reported suddenly without check-in reports for a few days; 2) a typical user may carry multiple instances of applications during their testing period. Only one application instance is used at a point of time. To determine if an application is used, we can simply apply a window rule whereby an application is assumed to be in use for a few days after a reported check-in or a crash report, even though no reports are generated. However, this assumption reduces the accuracy in the analysis. Instead we explore more extensive field data that could indicate when the application is active. We analyze other field data unrelated to crash reports, such as exception reports, battery logs and miscellaneous reports that would suggest the application was on even if a check-in report is not generated.

Application switching: To handle the presence of multiple application instances from a particular user, we track all field data from one application pertaining to a user. This technique provides a better indication if an application is removed from the testing period after a long period of absence or if a user simply switches among applications. This technique can also detect when a new application is placed in use by a user.

Date alignment: Each log is stamped with a calendar date when an event happens. When a new application is released for testing, not all the users start to use the new model in the same date. Aligning the crash dates to the corresponding day of use is of primary importance in order to capture the trend of usages. In particular, each user may have the tendency to generate similar log events during a certain period of use as they get more familiar with features in an application. Instead of using the absolute date when an application is released, we capture the testing period of all the field data for each user, and align the date of each crash into the actual day of use within the testing period. The time is considered relative to the starting point where the application is used by the user. More specifically, we put the starting dates of all users to a universally Day 1 and incrementing the Day with every new date when an application reports log events. Day 1 is started from the first time when an application submits any type of reports. The next unique date that the same application logs for a notable event will become their Day 2. For example, say that an application is on Day 4 of its testing period and the real date is 2008-09-08. Assuming no notable events happened until a battery log was reported on 2008-09-11, we must log this event on Day 5. We repeat this procedure for all users of a application model. As a result, all users effectively start on Day 1 and end on Day 30.

3.2.3 Evaluating the Independence of Metrics

To evaluate that proposed metrics are essentially different, we measure the correlations among the results of metrics. The strength of the correlation shows the similarity among the

metric results. A weak or no correlation among the results of metrics is desirable. It indicates less dependence among the results of the metrics, and in turn shows that the metrics convey different aspects of quality.

We use Spearman, a non-parametric correlation approach [17] and Pearson, a parametric correlation analysis [45] to evaluate the correlation. Both correlation analyses produce the similar results. In the rest of our discussion, we only report the results from Spearman analysis. We use the data from each version to compute the result for each metric. As a result, 21 results of each metrics are produced from all the versions. We analyze the independencies among the metrics.

Table 3-2. Spearman Correlation Results between Metrics

	MTBF	**IFFP**	**FAR**	**OFR**	**FMR**
MTBF	1	0.41	0.28	0.55	0.58
IFFP	0.41	1	0.02	-0.17	0.39
FAR	0.28	0.02	1	0.14	0.37
OFR	0.55	-0.17	0.14	1	0.29
FMR	0.58	0.39	0.37	0.29	1

MTBF: Mean Time Between Failures; FAR: Failure Accumulation Rating
IFFP: Initial Failure Free Period; OFR: Overall Failure Rating;
FMR: Failure Memory Rating

Table 3-2 shows the results of parametric correlation between the metrics. All metrics show a correlation less than 0.6. A weak correlation between the two metric values is determined if the correlation score is less than 0.6 [16]. We use MTBF, widely used to measure software reliability, as a baseline to evaluate if the proposed four metrics can asses different aspect of software quality other than the MTBF. As shown in Table 3-2, Failure Memory Rating and Overall Failure Rating metrics have a weak correlation with MTBF. However one metric with a weak correlation with

another metric is not indicative of the same pattern with other metrics. For example as shown in Table 3-2, the Failure Memory Rating has no correlation with other metrics except MTBF. All four metrics (Failure Memory Rating, Failure Accumulation Rating, first time impression and Overall Failure Rating) show no correlations. Overall, the correlation between metrics is considered to be low. As a conclusion, the metrics convey unique aspects of user perceived quality in addition to MTBF.

3.2.4 Evaluating the Predictive Powers of Metrics

It is infeasible to recruit the original users and ask them to report whether their perceived quality during beta testing accurately reflect the state of the current software. Instead, we examine two statistical correlations to determine if the metrics can predict the user perceived quality: 1) the correlation strength between the values of metrics and the number of the severe post-release bug reports and 2) The correlation strength between the average usage period of each software versions and their metric scores. The field data were collected over several years in the past for beta testing.

We start by showing the correlation strength between metrics and post-release bugs. Essentially, less bug reports inherently have more positive scores within the metrics. It is desirable to have a strong negative correlation between the results of a proposed metric and the number of severe bugs. For example a small number of severe bug reports after release would require high positive metric scores indicate could predictive ability whereas low correlation would mean the metrics could not predict the perceived quality software. To examine the predictive power of the metrics, we calculate metric results using the data collected from the first 30 days in the testing period. Furthermore, we correlate the results for each metric with the

number of severe bug reports after each version is released for the six months, one year and two years respectively.

Similar to the prior analysis for examining the independence of the metrics, we use Spearman, a non-parametric correlation approach and Pearson, a parametric correlation analysis to evaluate the correlation. Both correlation analyses produce the similar results. In the rest of our discussion, we only report the results from Spearman analysis.

Table 3-3. Spearman Correlation between Metric Results and Bug Reports

# of Bugs	MTBF	IFFP	FAR	OFR	FMR
6 months	-0.3	-0.5	-0.5	-0.8	-0.7
1 year	-0.6	-0.5	-0.3	-0.7	-0.7
2 years	-0.5	-0.4	-0.3	-0.8	-0.7
MTBF: Mean Time Between Failures; FAR: Failure Accumulation Rating IFFP: Initial Failure Free Period; OFR: Overall Failure Rating; FMR: Failure Memory Rating					

The correlation results are listed in Table 3-3. The strong correlations are highlighted in grey. A high number of severe bugs potentially have the negative impact on the software quality. Therefore, the number of severe bug reports is negatively correlated with the results of the proposed metrics. The correlation may be reduced because not all bugs reported are the direct result of failures and have visible impact on software quality (i.e. developers may simply want to improve certain functionality that never reported failures). Therefore, the relatively strong correlation between the number of bugs and the proposed metrics suggest good predictive power of the proposed metrics.

As shown in Table 3-3, the metric results correlated well with the number of the bugs in different periods after the release. It indicates the different metrics can predict the state of bugs and by extension software quality in these periods. For example, the Overall Failure Rating (i.e., OFR) and Failure Memory Rating (i.e., FMR) are strongly correlated with the bug reports in all three periods. A strong correlation is shown between the Failure Accumulation Rating (i.e., FAR) and the number of bugs within the first 6 months. It suggests that FAR can predict well the distribution of the failures experienced by the majority of users in the first 6 months. This also shows that the number of severe post release bugs reported is largely unaffected by the number of failures that users encountered after 6 months. MTBF shows a stronger correlation with the number of bugs after one year. It is probably caused by the severe bugs that cause failures which are fixed within the first six months. Therefore, the high score of MTBF predicts less severe bugs after 6 months. Similarity, the Initial Failure Free Period (IFFP) can predict well the quality of the application within one year of use.

Our second analysis solidifies our findings from table 3-3 by correlating the metric results to the amount of time a given user uses the same version of their software. We find the average number days that a given software version is used by the users and correlate it to the metric scores for the same version for an initial twenty day usage period. Again, we use Spearman and Pearson correlation to validate our results. Table 3-4 shows the correlation results between the metrics and the average usage period of users for each application version. In this case a high correlation would indicate that higher metric scores correlated to a longer usage period which would indicate higher user perceived quality in that aspect. The results show that OFR, FMR and IFFP are most suitable for indicating user perceived quality of users after 20 days. For example, their strong positive correlation indicates versions with higher scores in those metrics tend to use the software for a longer period of time before upgrading. This also agrees with the significant correlation

Table 3-4. Spearman Correlation Results between Metrics

	MTBF	**IFFP**	**FAR**	**OFR**	**FMR**
Correlation with days of use: 10 days	0.1	0.1	0.6	0.3	0.7
Correlation with days of use: 20 days	0.2	0.6	0.2	0.7	0.6
MTBF: Mean Time Between Failures; FAR: Failure Accumulation Rating IFFP: Initial Failure Free Period; OFR: Overall Failure Rating; FMR: Failure Memory Rating					

results those metrics have with severe bugs in table 3-3. The results also show that most users are unaffected by the average number of failures in the application as indicated by the relatively low correlation value for OFR.

3.2.5 Threats to Validity for User Experience Metrics

In the case study conducted on the user experience metrics as shown in section 3.5, each version has at least 50 users. However, users tended to fluctuate significantly between versions. Users may have different preferences that cause the user to behave significantly different from others. Some users were more casual and seldom use their application, while others use their user on a daily basis. Therefore, it may be more accurate if each version had roughly the same user base to ensure unbiased usage. The metrics focus on evaluating the average case with a large number of users. In the future, we will investigate the effect of the individual user behaviors on the frequency of the failures, and the impact on the metric results.

In the case study, we focus on the failures experienced by beta or alpha version users. However, the users understand that there will be failures and know what to do with the failures. In contrast, the real users experience failures differently: the real users may try harder to find a workaround to avoid using the functional features that crash often; or the real users give up immediately and switch to other application instead. In the future, we will conduct user study during the testing period.

It is a viable way to validate the predictive power of the proposed metrics by checking the correlations with the number of severe bugs reported after the release. However, some of the bugs that cause failures during the testing period could be fixed before the software release. Therefore, the correlation shown in Table 3-3 would be stronger when considering the bugs reported during the testing period.

In the evaluation of the Failure Memory Rating metric, we set the failure increase constant (i.e., FIC) to 100 and the cooldown constant (CC) to be 25 (i.e., α is set at 0.25 as defined in Eq. (3-6)). FIC and CC should be determined by performing user studies. Nevertheless a single FIC and CC could be used to compare different versions of the same application. The work presented for the metrics are still considered very preliminary in its development. Therefore, while the results have showed promise, more user studies need to be conducted in order to strengthen the validity of the data found so far.

3.3 Chapter 3 Summary

This chapter discussed the possibility of using user log data as input for user perceived quality metrics. In the next chapter, we focus on abstracting the same user log data into visualization graphs for easy identification of patterns between problems and users. We explore the feasibility of using these patterns to help resolve problems and automatically filter data.

Chapter 4
Visualizing Field Testing Results

In the current state of practice, it can be beneficial for developers to be able to analyze problems with a global view to establish potential relationships between different problems and the relation between the users reporting these problems. For instance, it may be the case that:

1. A particular set of problems co-occur frequently together; therefore studying and resolving any one of these problems might lead to the resolution of all these problems. With some problems easier to replicate than others, such information is likely to lead to faster resolution of these problems. We developed a *problem graph* to highlight such information.
2. Several users often report the same set of problems (i.e., share the same problem profile); therefore recruiting a few of these users to verify any fix is often a faster option instead of deploying the fix across the field blindly and awaiting problem reports. We developed a *user graph* to identify users reporting common problems. We can cross reference the problems reported and recruit users to replicate them.
3. A large number of problems are reported, however very few of these problems have a wide impact on many users. Fixing problems with high impact is usually a high priority effort. We developed an *interaction graph* to give a global view of the relation between the participants of field testing (i.e., users) and the outcome of the testing (i.e., problems).

Using our visualization, developers can analyze massive amounts of data reported during field testing. Our visualization helps identify patterns that demonstrate the user and problem interactions. The patterns allow developers to tackle problems with a global view instead of individually. In particular, such patterns help developers categorize, prioritize, and replicate problems, allowing developers to observe new relationships that may be overlooked in practice.

By visualizing the field testing results across multiple product releases, developers can compare the progress of field testing efforts for different releases.

4.1 Three Graphs for Visualizing Field Data

In this section, we introduce the three types of graphs that are produced from the field data. Our graphs could have as a node: a problem report, a user, or both. The edges in our graphs are based on different types of relations between the nodes. For example, an edge between two nodes in a problem graph indicates that the two nodes (i.e., problems) co-occur *frequently* together in a report. While an edge between two nodes in a user graph indicates that the two nodes (i.e., user) *frequently* report the same types of problems. Due to the large number of reported problems and users in field testing, we cannot simply show all the edges and nodes. Instead we must filter them. In the following subsections, we present our three graphs in more detail. We discuss the filtering process after the presentation of the three graph types.

4.1.1 Problem Graph: Visualizing the Interaction among Problems

In certain situations, developers tackle field problems on an individual basis. In these cases developers may overlook the possibility that certain problems may be interconnected not unlike what is described by Pantel *et al.* [44]. For instance, one problem reports that the disk is full, while another problem reports that the application fails to write to disk. Both problems can occur independently for different users or together for the same user who attempts to write to a full disk and triggers both problems. If the write-failure problem reports were sparse, developers might have a hard time to capture and fix the write-failure problem. However if developers can determine that this rare write-failure problem co-occurs frequently with a disk-full problem, developers can use the disk-full problem to better understand the write-failure problem

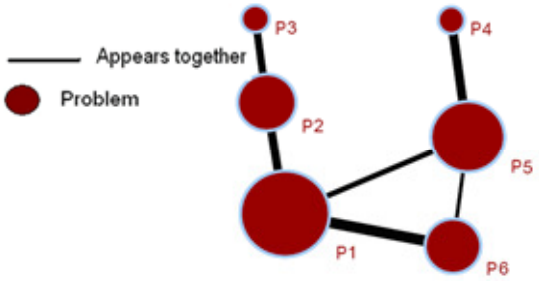


Figure 4-1. An example of a problem graph

and resolve it in a timely fashion. This situation may occur in large complex applications, given the different coding and reporting standards followed by various groups. By being able to show certain problems are related, we can overcome the fact that some reports might be sparse or that some reports are harder to reproduce or investigate.

In a more complex situation also described by Pantel, each occurrence of an error may trigger a different, yet often limited, number of problems. Such an error is primarily due to the use of an un-initialized field or timing/race conditions. When a user encounters the error several times, various problems associated with the same error can be reported by the same user in different occasions. Identifying the problems reported by the same user helps developers to capture the common cause for the problems, and fix all the problems at once.

Another type of problem co-occurrence happens when a single error leads to a cascade of related problems. For example, a problem, caused by an error in the code that reads data from the network, would often lead to multiple problems triggered in the subsequent execution of the application (due to the network-read-errors). In an ideal situation, such problems would have been reported as the same type. However, all too often the errors might manifest themselves as different problems. For example, the error to read from a network might manifest itself as the handling of a NullPointer problem, and the subsequent problems are reported as out of bound

processing. Identifying the problems that have these common origins could potentially help developers resolve multiple problems at once.

The aforementioned situations highlight the importance of studying field problems together instead of individually. By interlinking problems, developers might discover unexpected yet important relations between problems during the field testing process. The problem graph establishes such interlinking among problems.

The problem graph is an undirected graph, $G_{Problem} = (V_P, E_P)$. The set of nodes, V_P, of the graph contains the set of problems (i.e., P) reported in the repository. We consider problem reports with similar call traces as the same report. The set of edges, E_P, contains undirected edges, $e_{ab}=\{P_a, P_b\}$ if problem P_a and problem P_b are reported together by one or more user. If a large number of users report the two problems together, then we believe that such a relation needs to be highlighted to developers for further investigation. The edges are not transitive. For example as shown in Figure 4-1, problem P_1 is connected to P_2 and P_2 is connected to P_3. This signifies that P_1 and P_2 have been reported by the same user, and that P_2 and P_3 are reported by the same user, but that P_1 and P_3 are not interlinked. The graph is designed so that only nodes with a direct edge have a relationship. For example, the interconnected nodes P_1, P_5 and P_6 indicate that all three problems have been reported although not necessarily all together. For example there could have been three different users reporting each pair of problems (i.e., $\{P_1, P_5\}$, $\{P_5, P_6\}$ and $\{P_1, P_6\}$) or one user reporting the three problems together (i.e., $\{P_1, P_5, P_6\}$). A problem node appears in the problem graph only if it is reported together with at least one other problem node by a user.

We add weights to the nodes and edges. The weight of a node indicates the frequency of unique occurrences of that problem among users in the field testing repository. Visually, nodes

Table 4-1. Summary of the Three Types of Graphs

Graph	Entity	Entity type	Weight	Filtering
Problem Graph	Node	Problem	The number of different users who report the problem at least once in the repository	Must co-occur with at least one other problem after edge filtering
	Edge	Both problems co-occur	Statistical weight (see section 4.2)	Statistical filtering (see section 4.2)
User Graph	Node	User	The number of unique problems that a user has in common with other users	Must report a single problem after edge filtering.
	Edge	Both users report similar problems	Statistical weight (see section 4.2)	Statistical filtering (see section 4.2)
Interaction Graph	Node	Problem, User	**Problem:** The number of times a problem is reported by any user **User:** The number of unique problems reported by a user	Same as above for problem and user nodes
	Edge	User reports the Problem	Statistical weight (see section 4.2)	Statistical filtering (see section 4.2)

are shown as circles with the weight of a node depicted as the size of the circle. Looking at Figure 4-1, P_1 is reported by more users than P_3. Table 4-1 summarizes the definition of nodes and edges for the Problem Graph.

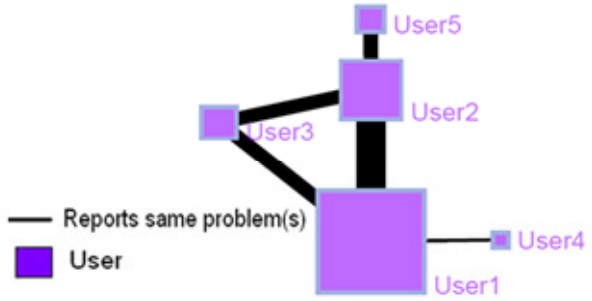


Figure 4-2. An example of a user graph

4.1.2 User Graph: Visualizing the Interaction among Users

Given the complexity of modern enterprise applications, the usage patterns between users tend to vary considerably with users forming clusters based on their usage of the application. For example, given an application for order taking and fulfillment, we would expect that at least two user clusters would arise along the two main uses of the application (i.e., order taking and fulfillment) with many sub-clusters forming based on the peculiarities of use within the two large clusters. Different clusters might also arise from commonalities across the user population. Similar characteristics such as similar hardware configuration or usage patterns (e.g., living in areas with bad wireless coverage or slow internet connections) are likely to lead users to report similar problems.

Flagging users with common problem profiles is of great value to developers. Using this knowledge about user clusters, developers could replicate specific problems and verify their fixes

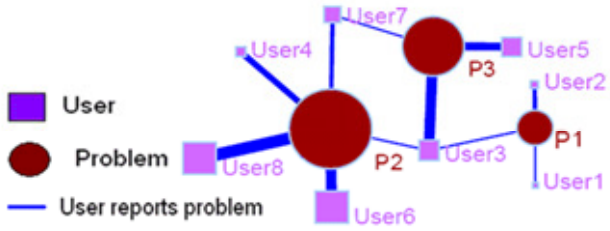


Figure 4-3. An example of an interaction graph

in an easier fashion. For instance, if most users in the order-taking department have the same problem profile, then a developer might be able to recruit a particular user to deploy additional monitoring on their installation. Developers can work closely with that user to solve a problem that affects the whole cluster of users in the order-taking department. Furthermore, using this knowledge about user clusters, managers can optimize their planning of future field testing efforts by picking at least one representative user from each cluster to participate in the field testing for a particular software version. Given the limited number of users that are willing to participate in field testing efforts and the large number of releases that are usually tested in parallel, a field-testing manager can distribute their user base more strategically. For example, if there are three versions of the software under testing and prior field testing shows the existence of two distinct user clusters, then the manager should ensure the participation of one user from each cluster to the three tests instead of randomly picking users.

We developed the user graph to help identify such clusters. The user graph is an undirected graph, i.e., $G_{User} = (V_U, E_U)$. The set of nodes, V_U, of the user graph contains a set of users (i.e., U) who report problems sent to the repository. The set of edges, E_U, contains an undirected edge $e_{ab}=\{u_a, u_b\}$ if users, u_a and u_b, have reported at least one common problem.

The weight of a user node represents the unique number of problems that a user reports with other users. For example a user that reports the same problem with four other users has a greater weight than a user who reports two common problems with one other user. Visually, nodes are shown as squares with the weight of a node depicted as the size of the square. Looking at Figure 4-2, $User_1$ reports more problems in common with other users than $User_3$. Table 4-1 summarizes the definition of nodes and edges for the User Graph.

4.1.3 Interaction Graph: Visualizing the Relations between Problems and Users

The interaction graph gives a global view of the results of field testing process. The graph cross references the relationship between problems and users. The interaction graph helps in analyzing the visualization produced by the other two graph types. For example, while the problems are not identified within the user graph, the interaction graph could be used to cross-reference the specific problems after locating a particular user base in the graph.

The interaction graph is an undirected, weighted graph, $G_{Inter} = (V_I, E_I)$. The set of nodes, V_I, is the union of the user set (i.e., U) and the problem set (i.e., P). Figure 4-3 illustrates an example of the interaction graph. The circular nodes represent problems and the square nodes represent users. The set of edges, E_I, contains an undirected edge, $e_{ab}=\{P_a, User_b\}$, if $User_b$ reports a problem, P_a. For the example shown in Figure 4-3, $User_3$ and $User_7$ both report problems P_2 and P_3. P_1 is reported by $User_1$, $User_2$ and $User_3$.

The weight of a user node captures the number of unique problems reported by the user. Visually, the weight of a user node is shown as the size of the square node. The weight of a problem node requires the frequency of the problem occurred during the field testing. Visually, the problems are depicted as circles with the weight of a node illustrated as the size of the circle. As depicted in Figure 4-3, User8 reports more problems than User1. Problem P2 occurs more

frequently than problem P1. Table 4-1 summarizes the definition of nodes and edges for the Interaction Graph.

4.2 Statistical filtering and weighing

Showing all the users and problems that exist in a field testing repository would lead to very complex graph with high over plotting. Therefore we chose to filter the edges and nodes in the created graph. Traditionally such filtering is performed by showing edges which are above a particular threshold. For example, we could show edges exhibiting a relation co-occurring 70% of the time or only 70 times. Instead we choose to perform statistical filtering given the large size of the data.

The goal of our **statistical filtering** is to only show edges that do not simply occur due to chance, instead we want to show edges that indicate statistically significant relations (i.e. association). We perform a chi-square (χ^2) test on each edge and filter edges in all graphs based on the results of the test [31]. Table 4-2 shows the 2x2 contingency table used to calculate the chi-square value. The same approach is used for all three graphs. For example, in the problem graph, the cell (P_a, P_b) counts the frequency (i.e., f_1) where problems P_a and P_b appear together for any users. The cell (P_a, $\neg P_b$) counts the number of times (i.e., f_2) that problem P_a occurred but not problem P_b for a user. The cell $(\neg P_a, P_b)$ counts the number of times (i.e., f_3) that problem P_b occurs but not problem P_a. The cell $(\neg P_a, \neg P_b)$ denotes the total number of times (i.e., f_4) that neither problem was reported. χ^2 is calculated using formula (1). Looking at the chi-square (χ^2) statistics distribution tables, we use a chi-square value of 5.99 to filter edges, indicating that we are 95% confident (i.e., p value<0.05) about the statistical validity of a shown edge.

Table 4-2. 2x2 Contingency Table

	P_b	$\neg P_b$
P_a	f_1	f_2
$\neg P_a$	f_3	f_4

$$\chi^2 = \frac{(f_1 f_4 - f_2 f_3)^2 (f_1 + f_2 + f_3 + f_4)}{(f_1 + f_2)(f_3 + f_4)(f_2 + f_4)(f_1 + f_3)} \qquad \textbf{Eq. (4-1)}$$

f_1, f_2, f_3, and f_4 are the frequencies of two independent events (e.g., problems) that appear for different cases listed in Table 4-2.

$$phi = \sqrt{\frac{(\chi^2)}{(f_1 + f_2 + f_3 + f_4)}} \qquad \textbf{Eq. (4-2)}$$

x^2 is the chi squared value calculated in formula (1). f_1,, f_2, f_3, and f_4 are the frequencies of the two independent events as described in Table 4-2.

Edges are given a **statistical weight** if it satisfies the conditions of statistical filtering (i.e., are statistically significant enough to be shown). The statistical weight of an edge is calculated using the *phi* measure [24]. The value of *phi* measures the strength of the relationship regardless of the sample size. It was chosen primarily to normalize the edge thickness in a given sample so that no edge would detract from the graph. It is based off the chi-squared test as shown in formula (1). The value of phi is computed using formula (2). A value of 0 for *phi* means that there is no association between two nodes (e.g., problems) and a value of 1 indicates a perfect association (i.e. both nodes always co-occur). Therefore, thick edges for any graph display phi values close to 1. The *phi* value is statistically significant when the χ^2 value is greater or equal to 5.99 (i.e., $p < 0.05$). We filter the edges if the weight of the edges is not statistically significant. Thin edges indicate low *phi* values meaning that the minimal statistical significance.

4.3 Case Study for Pattern Identification

To demonstrate the benefits of using our proposed graphs in section 4.3, we performed a case study using two beta versions of a large scale enterprise software application used by millions of users worldwide for communication. The application is written in the Java programming language. Table 4-3 shows descriptive statistics for each version. The data collected for this study was gathered over the course of 30 days within a field test of each version. To ease the comparison of problems of both versions, identical problems appearing in both versions are assigned the same identifier.

Table 4-3. Statistics for the Two Studied Field Tests

Version	# of Users	# of Reported Problems
A	367	342
B	1,302	883

The objective of this case study is to identify if there are any significant differences or trends among the two versions of the application. We analyze the three types of graphs to automatically identify patterns common in both versions.

4.3.1 Steps for Identifying Patterns from the Graphs

We generate the three graphs and automatically identify patterns. It should be noted that the following steps are automated in script form with no manual intervention until the graphs have been generated. The process is done automatically in the following steps:

1. For both versions, we analyze each problem report to determine the user who reported the problem and to determine similar problems using the reported call traces. We also identify similar problems across both studied versions.
2. We generate the edges between each user and problem.
3. We apply statistical filtering and weighing of the edges as described in section 4.2. We filtered nodes which no longer have edges attached to them.
4. We visualize the graphs using a spring-based layout algorithm that is built in the GUESS (Graph Exploration System) [23]. GUESS is a system and language for visualizing and manipulating graph structures. It can accept command scripts, which help highlight identified patterns from the data set. For example, Figures 4-4a), 4-6a), 4-7a) are the three types of graphs visualized for version A after statistical filtering.
5. We apply our pattern recognition scripts on the visualizations in GUESS to locate the patterns in the data set. The scripts were programmed to look for certain characteristics such as the number of edges attached to a node and the thickness of edges. For example, to find strongly interconnected problems in the problem graph, a GUESS command is used to isolate edges that are statistically significant. For example, Figures 4-4 b) and 4-4 c) shows instances of two patterns from the visualization of version A in Figure 4-4 a). Similarly instances of patterns in version B are depicted in Figure 4-5 b) and 4-5 c) based on Figure 4-5 a). Other patterns are observed directly from the GUESS visualization without applying our pattern recognition script. The use of a spring-based layout technique automatically clusters the data as shown in Figure 4-6.

4.3.2 Identified Patterns

We identified five patterns common in both versions using the three types of graphs. We summarize the patterns using the following template:

- **Symptom:** describes the characteristics of the pattern in the graphs which first caught our attention. For several of the patterns, the characteristics of the pattern are encoded in GUESS scripts which enable the automatic detection of instances of the patterns in the graphs.
- **Examples:** show examples from the studied application.
- **Significance:** explains the benefits of identifying the pattern.

4.3.2.1 Interconnected Problems

Symptom. This pattern is found in the problem graphs. The pattern is characterized by a thick (i.e., statistically significant) edge connecting adjacent problem nodes. The sizes of connected nodes can vary. This indicates that the two problems are reported frequently together, suggesting a chain reaction. More than two problem nodes can be interconnected with thick edges. This indicates a longer chain effect among the interconnected problems.

Examples. Figure 4-4 b) shows the identified interconnected problem patterns for version A. In the graph, we found that problem pairs $\{P_{2,} P_{16}\}$ and $\{P_{35}, P_{41}\}$ are separate examples of the interconnected problem pattern with their thick edges between each of the two node pairs.

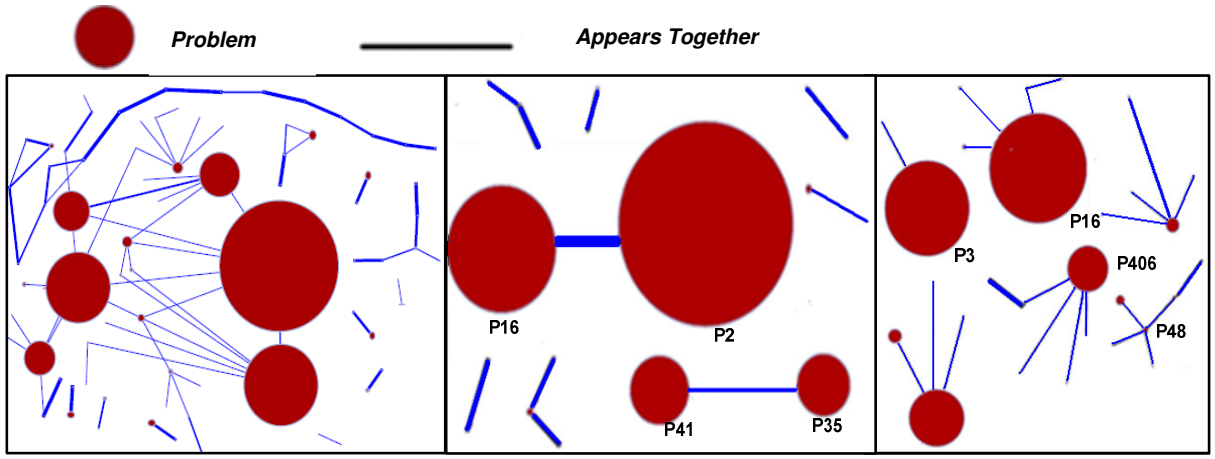


Figure 4-4. Interconnected and duplicate patterns extracted from version A

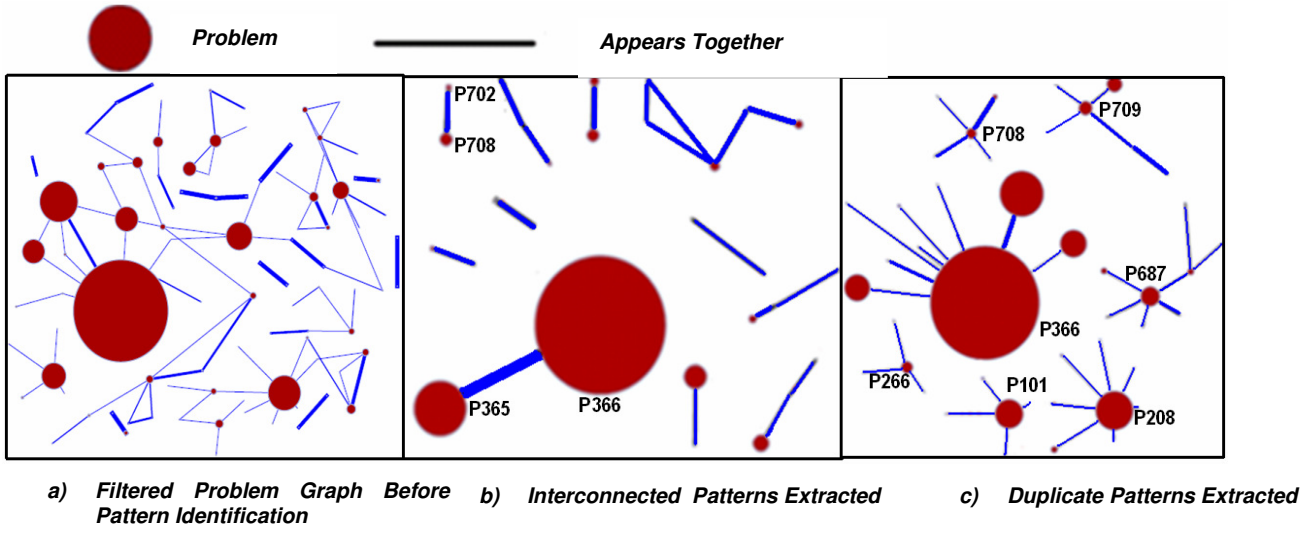


Figure 4-5. Interconnected and duplicate patterns extracted from version B

Similarly, Figure 4-5 b) visualizes the interconnected problem patterns recognized in version B. For example in Figure 4-5 b), the interconnected problem pattern is exhibited by the thick edges, such as $\{P_{365}, P_{366}\}$ and $\{P_{702}, P_{708}\}$. The node size indicates the frequency of the problem being

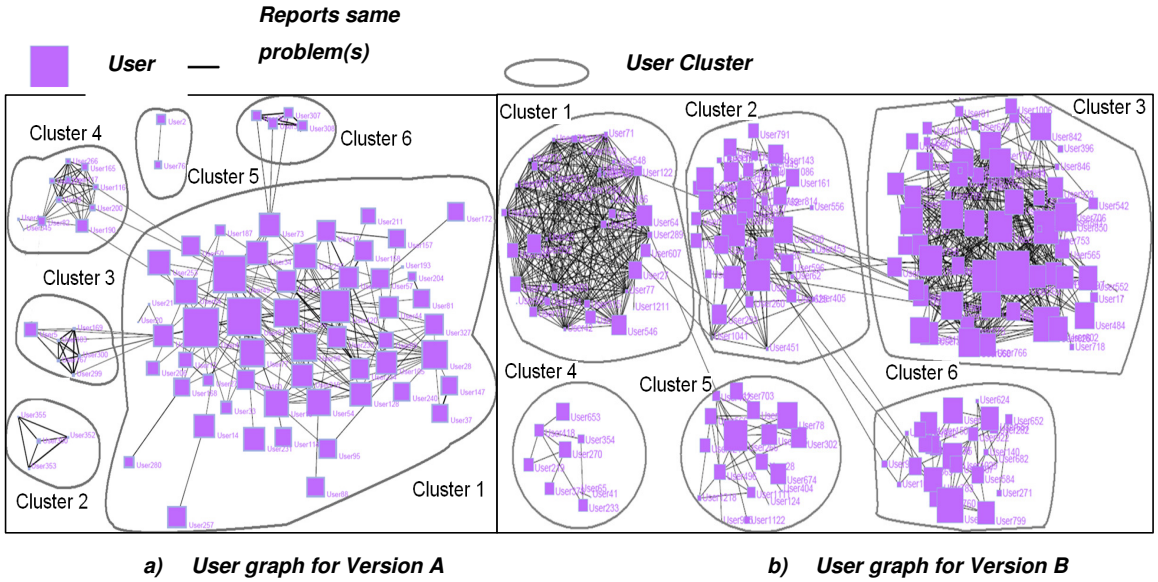


Figure 4-6. Visualization of the user graphs for both versions

reported by different users. If the problems are not frequently reported, fixing such interconnected problems might not be a high priority. For example illustrated in Figure 4-5 b), the node sizes of problems P_{365} and P_{366} are much bigger than problems P_{702} and P_{208}. Therefore, fixing problems P_{365} and P_{366} is a higher priority than fixing problems P_{702} and P_{208} because P_{365} and P_{366} are reported by a larger number of users.

Significance. Examining a particular problem might prove difficult to replicate or fix. However, that problem might co-occur frequently with another problem. For example, due to a chaining effect one problem might always be followed by another problem meaning one problem cannot occur without the other occurring first. Problems that are intertwined in this fashion indicate a series of chain effects that eventually lead to a final problem. It is desirable to find these patterns to eliminate the initial problem so it cannot propagate.

4.3.2.2 Near Duplicate Problems

Symptom. This pattern is recognized in the problem graphs. It is characterized by large problem nodes that have many smaller problem nodes attached to them. In effect the large node and its smaller neighboring nodes are essentially the same problem. However, when users alter their usage slightly, the two problem reports are recognized as different problems.

Examples. Figures 4-4 c) and 4-5 c) show the near duplicate problem patterns identified from both versions. For example, looking at Figure 4-4 c), $P_{3,}$ $P_{16,}$ P_{48} and P_{406} represent nodes with the duplicate problems. Such problem nodes have smaller nodes that branch off them. Similarly, the problem graph of version B, shown in Figure 4-5 c), has instances of this pattern which centers around the problem nodes, such as $P_{101,}$ P_{208} and P_{366} and P_{687}.

Significance. During field testing, a set of problems may be reported infrequently compared to other problems. Through close investigation, we might determine that the set of problems is a special case of a frequently-occurring problem (i.e., the near duplicates of each other). In this case, it is desirable to assign the investigation of both problems to the same developer.

4.3.2.3 Distinct User Cluster

Symptom. This pattern is found in the user graph. It is characterized with large clusters of user nodes that are tightly connected. The clusters are formed by applying the spring based layout algorithm which groups the user nodes with stronger association closer. The statistical filtering and weighing described in section 4.2 ensure that the clusters are statistical significant and not simply due to chance. The clusters are visually identified from the user graph.

Examples. Figure 4-6 shows the user graph for both versions. Both graphs are layed out using a spring-based algorithm. We note that each version manifests its own distinct clusters of user nodes. Version A has a large central cluster (i.e., cluster 1) with several splinter clusters. It indicates that the majority of users report the same set of problems and are indistinguishable by the problems they report. Version B has six more distinctive clusters. It shows that the problems among users are more distributed. Future field testing efforts are likely to benefit from picking a user from each cluster instead of randomly picking users which might all fall within a single cluster. Therefore it is more likely users can replicate a problem in a cluster of version B than they can for version A.

Significance. Identifying user clusters based on common problem profiles helps practitioners in the management of current field testing efforts and the planning of future efforts. Developers can work closely with representative users of clusters to gain a better understanding of their problems through interviews and additional instrumentation. Furthermore, the representative users could be used to replicate problems and verify fixes before they are deployed more widely during the field testing. The identification of user clusters could also help managers in planning future field testing efforts. Managers can ensure that each field test run has representative users from each cluster to guarantee a wider and more general testing of an application.

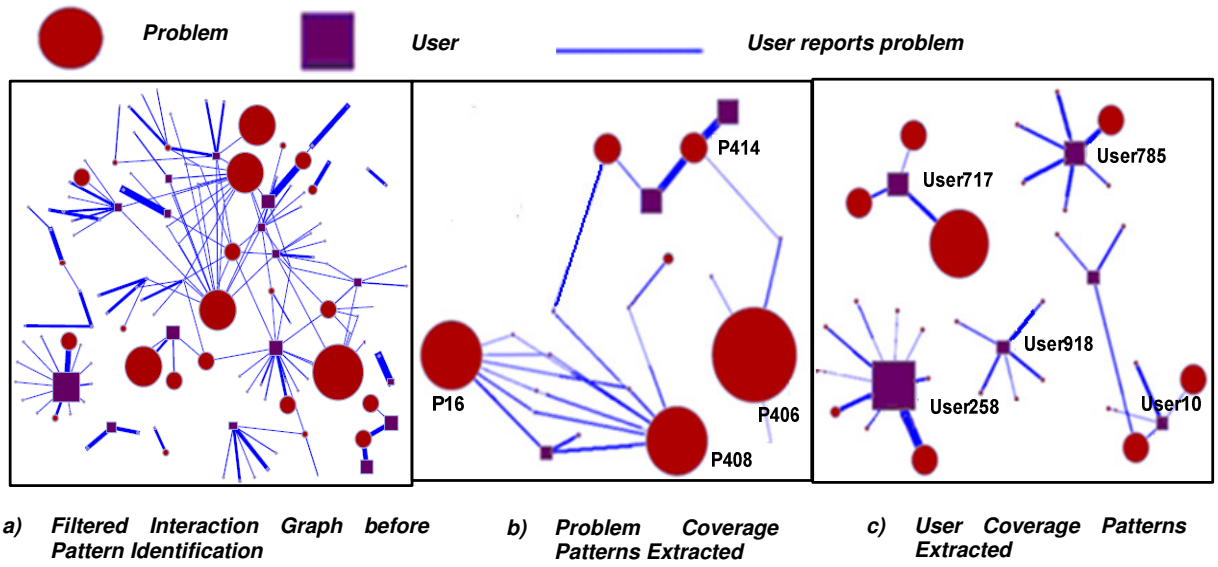


Figure 4-7. Visualization of the interaction graph for version A

4.3.2.4 Distributed Problem Coverage

Symptom. This pattern is uncovered by analyzing the interaction graph. It is characterized with a problem node having a large number of adjoining user nodes. To prioritize the fixing of problems, we identify the problems reported by a large number of users. If a problem node is attached to a large number of user nodes, it presents a wide spread problem. The thicknesses of the edges show the distribution of the problem occurrence across the users. If we can identify a very thick edge to a particular user, this indicates that this problem primarily occurs for that user. If a problem node is attached to a small number of user nodes, the problem is not widely spread. Part of the analysis also involves examining the users who report the problem to determine if they report a large number of problems.

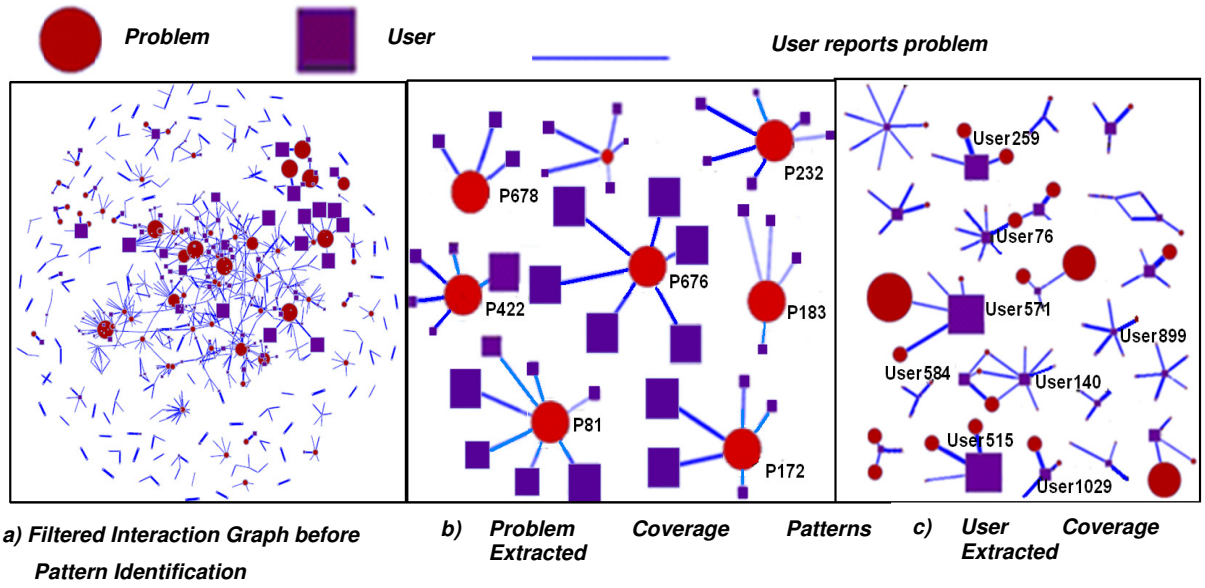


Figure 4-8. Visualization of the interaction graph for version B

Examples. Figures 4-7 b) and 4-8 b) show the distributed problem coverage patterns identified from the interaction graph for versions A and B respectively. Looking at the graph for version A depicted in 4-7 b), we note several instances of this pattern, such as the clusters around the problems, P_{16}, P_{406}, and P_{408}. For example, problem, P_{16} and P_{408} are reported by the same set of users. The interaction graph of version B illustrated in Figure 4-8 b) contains many instances of this pattern, such as the clusters around the problem nodes, P_{81}, P_{172}, P_{183}, P_{232}, P_{422}, P_{676}, and P_{678}. As a result, these problems are good candidates to fix first.

Significance. Prioritizing which problems to address first can be a complex decision involving the criticality of the problem and its widespread impact on the user base. Given two problems that occur frequently, it is often desirable to fix problems that impact a larger number of users evenly. Counting the number of users that encounter the problem is a good metric. However, the metric fails to capture the frequency of the occurrence of the problem. For example, given a problem

which occurs 100 times, it could be the case that the problem occurs once 80 times for a single user and very infrequently for another 20 users. Or it could be that the problem occurs evenly across the 21 users. A visualization that shows this distribution will help in the prioritization of problems. Moreover we must consider each user that has this problem in the context of the other problems. Referring back to our previous example of a user with a problem being reported 80 times, if we notice that this user is reporting a large number of other problems, it might be the case that this user's installation or environment has problems and that the problem is not as frequently occurring as we thought.

4.3.2.5 Distributed User Coverage

Symptom. This pattern shows up in the interaction graph with a large user node having many adjoining problem nodes. This pattern appears visually as a star-like shape with a center user node and problems radiating out of that user node.

Examples. Figures 4-7 c) and 4-8 c) show the distributed user coverage patterns extracted from both versions. As illustrated in Figure 4-7 c) for version A, the clusters around user nodes, $User_{10}$, $User_{258}$, $User_{717}$, $User_{785}$, and $User_{918}$, are example of this pattern. Similarity, Figure 4-8 c) for version B contains many instances, such as the clusters around user nodes, $User_{76,}$ $User_{140}$, $User_{259,}$ $User_{515}$ and $User_{571}$.

Significance. Users reporting a large number of problems are ones developers should examine closer. The user graph shows such users as large nodes. However, the mapping from the user to specific problems is not visible in the user graph. We want to better understand the distribution of problems generated by a particular user. In the same way we examine the distribution of a particular problem across different users. By exploring these users, we can then study if their

applications are misconfigured and whether their results should be ignored. Or if they are facing a large number of problems and are ideal users to work with closely. In essence, this pattern helps reduce the time of developers in finding good testing candidates.

4.3.3 Verification of Patterns Found in Case Study

To verify that the patterns identified in the graphs fulfill their respective purposes and are in fact statistically significant, we manually verify that the occurrence of each pattern is in fact a valid pattern. Each pattern has different criteria in order to be verified as real. For interconnected patterns, we manually identify all the problems that were reported as interconnected by this pattern. We then manually check the call stack associated with each problem that is actually logically interconnected and check the average time between their reports. Similar to the interconnected pattern, we manually verify the near duplicate pattern problem to see if they can be duplicates of each other. We analyze the call stack to see file similarity, and ending exceptions between the central problem and the identified duplicates. To verify the user clusters, we determine users in each cluster report the same problems. We randomly select 10 percent of users from each cluster and test if they all report a given problem that was known to be reported from that cluster. For the distributed problem coverage pattern, we manually verify each problem that was identified in that particular pattern using its node size. We compare the average number of user logs that will report a problem identified from the distributed problem coverage pattern to the average number of user logs that will be report a non-distributed problem. We use a similar approach to validate the distributed user coverage pattern. Since the pattern is used to highlight users useful for replication of unique problems we manually verify the average number of unique problems reported by users identified by the distributed user coverage pattern. We compare this number to the average number of unique problems reported by users not identified by this pattern.

$$Precision = \frac{(verifiedPatterns)}{(numberOfIdentifiedPatterns)} \quad \textbf{Eq. (4-3)}$$

For the case study, we use table 4-4 to summarize the statistical significance of the patterns identified. To determine the validity of our pattern identification scheme we calculated the precision of each pattern type identified. The precision for pattern type identification is defined in Eq. 4-3 as the number of patterns verified to be a certain type over the number of patterns identified as a certain type. Table 4-5 shows the number of patterns identified for each version.

Table 4-4. Statistics Validation of Patterns from Problem and User Graphs

Version	Precision of interconnected problem identification	Precision of near duplicate problem identification	Average percentage chance of finding correct user in cluster to report specific problem	Precision of distributed problem coverage pattern	Precision of distributed user coverage pattern
A	0.70	0.7	0.92	1.0	1.0
B	0.68	0.60	0.89	1.0	1.0

Table 4-5 Number of Identified Patterns for each version

Version	Number of interconnected problems identified	Number of interconnected problems identified	Number of user clusters identified	Number of distributed problem coverage identified	Number of distributed user coverage identified
A	12	10	6	6	7
B	22	15	6	14	10

Table 4-4 shows that around 70% of the interconnected problems identified by the pattern were manually verified to be related at the source code level. The reason that the precision is not

perfect is because certain interconnected problems were found not to be related after manual verification. For example, the file traces showed the executions were from two different unrelated modules, but coincidentally, the problems occurred in the user logs with comparative frequency and time. In the future, problems should have further criteria imposed before they are classified as interconnected problems such as module similarity and variable dependency.

We found that around 64% of identified duplicate problems were manually verified to be actual duplicates. During manual verification, it was found that the supposedly duplicate problems that were connected to the central problem did not have sufficient similarity. For example, the duplicate problems had similar file accesses but not method accesses. In the future, stricter matching criteria should be imposed that includes method similarity as well as call trace length when determining if a problem may be a duplicate.

Table 4-4 shows the average percentage of users that were found to report a given problem for all the clusters in each version. For this pattern, the precision remains high for both versions. It is unlikely to find perfect precision for this pattern because no matter how similar users are in a cluster, there will be some deviation for the set of problems reported between users. To maximize the precision, the sorting algorithm can have stricter criteria, such as only grouping two users together if their reported problem set is exactly the same.

We determined that the average problem identified by the distributed problem coverage pattern has higher coverage than the average problem that is not part of the pattern. We found that the problems identified as a distributed problem pattern had a consistently higher number of users reporting it. As with the distributed problem pattern, we found that the users identified in the distributed user pattern consistently reported a higher number of problems than their counterparts.

4.3.4 Visualizing of Patterns after Problem Resolution

The visualization technique is capable of showing the impact after a problem has been fixed. For example, if a specific problem that falls under the distributed problem coverage pattern is resolved, then scripts can be told to remove that specific problem from the Interaction Graph. In this sense, developers can be sure they have an up to date set of problem patterns visualized in their graphs.

4.4 Chapter 4 Summary

This chapter explored different visualization graphs for identifying patterns between problems and users in order to filter user log data and help resolve problems. The next chapter uses visualization techniques to map the file call trace of the entire user population in order to identify files of interest in bug isolation.

Chapter 5
Structure of Problems in Field Testing

Extensive field testing efforts produce a large number of reported problems. It would therefore be desirable for developers to eliminate problems with common causes without having to analyze each situation specifically. Basic techniques are used to group similar problems. For example, problems leading to a crash are grouped together while problems relating to excessive CPU usages are grouped together. Summary statistics are often collected and reported to track the progress of field testing efforts. When attempting to resolve a particular problem, developers work on problems in isolation. A global view of the reported problems and their interconnection is needed.

A global view helps developers in identifying patterns of problems and usage behaviors. In chapter 4, which used different visualization software, we proposed the use of a spring-based layout visualization technique to highlight the relation between different problems. In those visualizations, each problem is shown as a node and an edge is drawn between two nodes if the same set of users encounter them frequently enough. The spring-based layout ensures that problems that co-occur frequently are placed close to each other such that they form a cluster. Using our prior visualization, we can identify several patterns that highlight the similarity between different problems. However, it may be hard to fix high-battery-usage problems when such problems are examined in isolation. However, if some high-CPU-usage problems are clustered close to other high-battery-usage problems in our visualization, these problems are strongly associated. A closer examination might indicate that the high battery drain is due to suboptimal CPU usage. The main audiences for our prior visualization were the managers of field testing efforts.

We propose a new visualization that is of value to the developers who must repair the field problems. In contrast to testing managers that looked at problem priorities, developers would like a visualization which focuses on the code structure of reported problems and the relation between the structures of problems. By identifying the commonalities between the structures of different problems, developers can pinpoint troublesome parts of the code. Using this knowledge, development mangers can also rapidly triage reported problem and assign them to the most appropriate developer.

5.1 Visualizing the Structure of Problems

When examining a particular problem report, developers may start their investigation by studying the method at which the problem was reported. For example, a method might have reported an out-of-range exception when it tried to access an array element. However, the out-of-range exception may not necessarily be caused by that method, but higher up the call path. For example, the exception could be due to the method which initialized the array with a pre-defined length then passed the array down the call path. The process of investigating a problem often consists of going up the call trace and investigating each method along the path of that trace. We name that call trace the structure of a problem. The process of going up the call trace is time consuming, given the usual complexity of reported problems and their large numbers. However, one can make use of the call traces (i.e. structure) of reported problems to quickly identify commonality between problem structures and to focus their efforts.

Consider the case of API methods which have undocumented requirements. The misuse of these methods might cause problems down their call path. However all these problems are linked to the API method which is higher-up in the call trace. A visualization that would identify this situation would be of great value to developers.

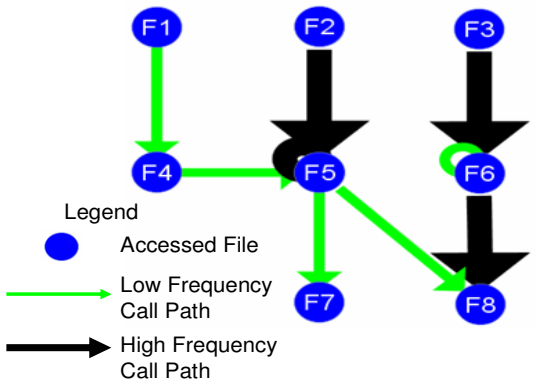


Figure 5-1. Problem structure visualization

To help developers understand the inter-dependencies among the problems, we developed a technique to visualize the structure of all reported problems in a single visualization. Given the large number of reported problems and methods, we present our visualization at the file level instead of method level to isolate files of interest to developers. The visualization highlights the commonly occurring call paths among reported problems. Each call path represents a reported problem.

Figure 5-1 shows an example of our visualization of the problem structure. The visualization shows a directed graph, i.e., $G= (V_F, E_F)$. The set of nodes, V_F, contains the complete set of unique files (i.e., F) appearing in the call traces of all reported problems. The set of edges, E_F, contains directed edges, $e_{ab}=\{F_a, F_b\}$, if file F_a subsequently accesses file F_b. A node, in Figure 5-1, represents files such as F_1 and F_4. An edge denotes a call between two files. Nodes that are being pointed to are designated as the callee while nodes with edges originating from them are the callers. For example as depicted in Figure 5-1, a method in node F_1 accesses a method in node F_4. A node can be both a callee and caller. The edges are not transitive. For example as shown in Figure 5-1, F_2 has a call edge to F_5 and F_5 has a call edge to F_7. This indicates F_2 accesses F_5 and F_5 can access F_7, but F_2 cannot access F_7. An edge may also refer back to the same node such is

the case for F_6. This means that the same file was accessed, and the call might be made to a different method in the same file.

The position of a node in all call paths is not fixed. For example, a node, F_5 is in the second position of the call path of F_3, F_5, F_7. F_5, and is in the third position of the call path of F_1, F_4, F_5, F_7. The call edges that connect from F_5 indicate the possible files that can access after it. Leaf nodes indicate the last file accessed before the reporting of a problem. The first node in a call path represents the first file initially accessed.

Node weights are constant and do not change regardless of its accesses. We add weights to the edges. Edge weight is determined by the frequency in which the call pair is accessed. If the frequency of a call pair is below a certain threshold T, it is given a thin edge. All call pairs that are above the threshold T are given a thick black edge. For example in Figure 5-1, F_2 is frequently reported to access F_5, while F_1 rarely accesses F_4.

5.2 Case Study for Identifying Patterns in Problem Structures

To demonstrate the applicability of our proposed visualizations for determining the structure of problems and their interdependencies, we performed a case study on the reported problems during field testing efforts for two releases of a large scale enterprise Java application used by millions of users worldwide. Table 5-1 shows the descriptive statistics for each used version. The data collected for this study was gathered from users over the course of thirty days.

Table 5-1. Statistics for Studied Field Tests

Version	Users	Reported Problems	Files in Call Trace
Version_A	367	1,621	1,096
Version_B	1,302	5,564	2,018

For the case study, we set a threshold for showing a thick edge if a call pair is accessed more than 80 times. Otherwise, the call pair is depicted in thin edge. The threshold could be increased when a developer wants to focus on the edges with high frequencies. On the other hand, the threshold could be lowered when the call pairs with low frequencies are inspected. The graphs are generated using AiSee [2], a graphical layout software whose customers include IBM and Intel.

5.2.1 Steps for Identifying Patterns from the Graphs

We generate the three graphs and automatically identify patterns. It should be noted that the following steps are automated in script form with no manual intervention until the graphs have been generated. The process is done automatically in the following steps:

1. For both versions, we analyze each problem report to determine the sequence of files accessed using the reported call traces. We also identify similar problems across both studied versions.
2. We generate the edges between files of all unique call paths found for both versions based on the frequency of each call.
3. We visualize the graphs with their edge weight using the layout functionality that is built in the AiSee.

5.2.2 Identified Patterns

AiSee reads textual specifications and generates the visual equivalent. The software is desirable for its ability to quickly process large amounts of data and simple input format. It also supports easy navigability, and customizable layout such as manipulating nodes and edges onscreen in order to get the optimum organization. In our case study we identified two patterns prominent in both visualizations depicted in Figures 5-2 and 5-3. These patterns help identify the problematic files and could be used to compare different versions. We summarize the patterns

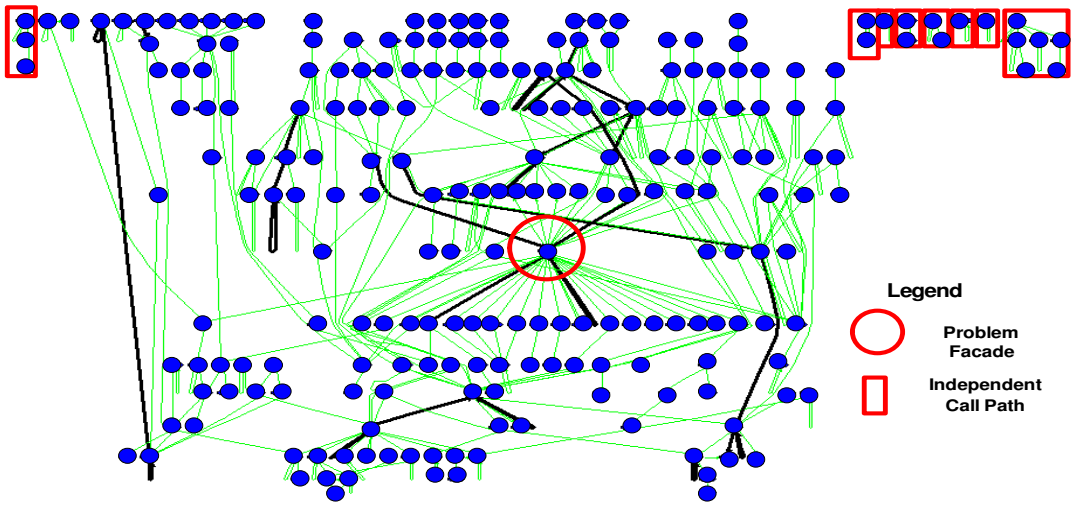


Figure 5-2. The structure of problems for version A

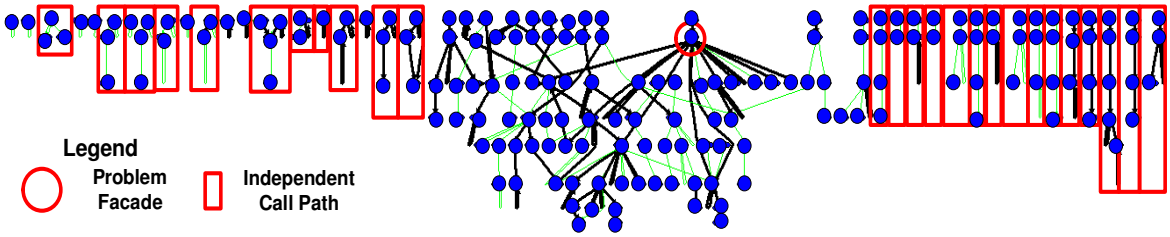


Figure 5-3. The structure of problems for version B

using the following template:

- **Symptom:** describes the characteristics of the pattern in the graphs which first caught our attention.
- **Examples:** show examples from the studied application.

5.2.2.1 Problem Façade

Motivation: A Problem Façade occurs when a large number of problems have a particular method in common. It is desirable to identify this method and put additional error checking in it or improve its documentation to prevent the occurrence of problems much lower in the call trace.

Symptom: This pattern is characterized by one node with many edges originating from it. This indicates that the file VF is commonly referenced in problem reports. The edges in this pattern can be thick or thin, however thick edges are good indicators that the pattern is more consistent.

Examples: Figure 5-2 shows several problem façade patterns identified in version A. The pattern is annotated as a circle in Figure 5-2. A similar pattern occurs for version B, shown in Figure 5-3. For version B, the Problem Façade method has a major file located at the top of the call paths indicating the file is accessed early in the call paths.

5.2.2.2 Independent Call Path

Motivation: When exploring the call trace (i.e., structure) of a particular problem relative to other problems, developers are faced with two types of situations either the call trace is significantly different than other call traces or it shares similarities with a large number of call traces (subsequently problems). The independent call path pattern occurs when a particular call trace is unique and different enough from the rest of the reported problems. Using this knowledge, developers have a better understanding of the impact of the problem and the potential benefits of fixing it. For example, the benefits of fixing a problem that is part of an independent call path pattern are low due to the small number of similar problems. Nevertheless, fixing such a problem might be easy due to the small number of interacting problems.

Symptom: The independent call path is shown as a series of nodes with a limited number of incoming edges and outgoing edges. Edges can be thick or thin, however all files in the call path access a small number of subsequent files.

Examples: As shown in Figures 5-2 and 5-3, both version A and B show several independent call path patterns. Specifically, version B has a higher number of independent call paths patterns. This indicates that a large number of reported problems in version B are singular cases while problems in version A are more likely to be inter-related.

5.2.3 Comparing Two Versions

While both versions show similar identifiable patterns, they have subtle differences. The defining characteristic of version A is the long call traces versus shorter calls traces for version B. In version B, we note that the independent call path pattern occurs a larger number of times relative to version A – indicating that the problems for version B might be easier to replicate. In version A, there are a larger number of problems with a similar structure as indicated with the large connected component in the center of Figure 5-2.

5.3 Chapter 5 Summary

This chapter used visualizations to identify patterns in the user log file traces. The next chapter analyzes discusses how the priority of problems should include its distribution as well as frequency. We further this analysis by using user log data to find the entropy and frequency of distinct problems. We determine how we can use these two characteristics of problems to find their relative priority to each other.

Chapter 6
Prioritizing Problems Using Pre-Deployment Field Data

Developers experience delays in resolving problems due to the uncertainty regarding their priorities. Many problems may also seem similar but due to underlying method invocations, may ultimately have different origins. They are also affected by the fact that the users recruited may not be able to replicate the problem they want specifically. Entropy can help developers determine priority of problems by measuring the distribution of a problem among users and its extent. Using our entropy analysis technique, developers can assess the priority of different problems more effectively than simply measuring the frequency and number of users reporting a problem. To help developers correctly identify the users reporting the problems with high priority, we visualize the users in clusters to specify which users report the most similar problems and highlight the affected users. We include a problem signature definition to remove uncertainty for developers referring to a specific problem.

6.1 Problem Signatures and Entropy

6.1.1 Problem Signature

Problems refer to the undesirable output that results from execution of a certain call path in software. For example, a problem can occur when the execution of a method encounters an unexpected situation and throws a sequence of Java exceptions in a call trace. When developers need to fix the cause of a problem, they cannot not only rely on the problem reported. They need to know the specific steps leading to the problem itself. While there are a limited number of unique exceptions that can be accessed in an application, the unique call traces leading up to the

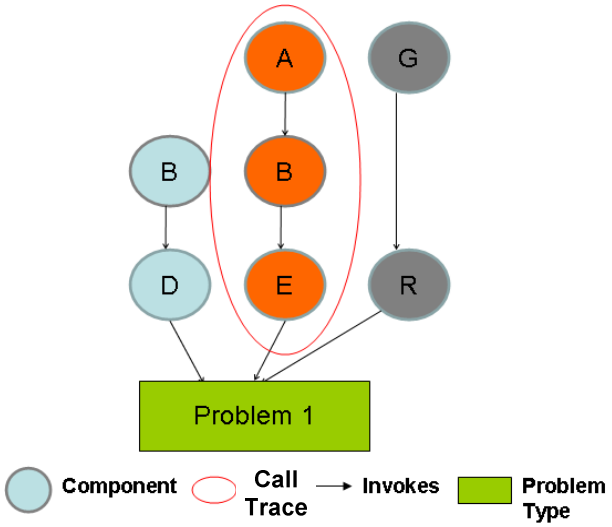


Figure 6-1. Example of problem signature

exception can be numerous. It is also possible for different call traces to result in the same exception invocation.

We define the problem signature as the sequence of methods accessed (i.e. call trace) leading up to the exception. Problem signatures reflect on the behavior of the user before a problem is accessed. Even if two call traces reported mostly contain the same sequence of methods, having one method that is exclusive to one call trace would classify them as two different problem signatures. Problem signatures always describe the history from the earliest method to the most recent. As shown in figure 6-1, method A is accessed followed by B then E. It is method E that eventually reports the problem. As in all call traces, there can be a varying amount of methods accessed before an exception is accessed. For example, the call trace A->B->E in figure 6-1 is one method longer that B->D. Therefore problem signatures can have varying call trace lengths.

6.1.2 Entropy of a Problem Signature

The entropy is a measure of the uniformity of a problem signature among its users. It is formally defined using Shannon's entropy equation [51].

$$entropy = \frac{\sum_{i=1}^{i=n} P_i \log_2(P_i)}{\log_2(n)} \quad \textbf{Eq. (6-1)}$$

n is the total number of problems that can occur in the system. Pi is the probability of problem i happening.

The entropy of problem signatures is calculated using the eq. (6-1) where P_i is the probability of a specific user i reporting a given problem and n is the total number of unique users. The entropy is normalized by $\log_2(n)$ from 0 to 1. The ideal value for entropy of a system is 1. If the entropy of the problem signatures is high (i.e. 1) it signifies that every user that reports the problem signature has an equal chance of reporting it. High entropy for a problem signature is the most desirable outcome since it means developers can select any of the users that report it with equal confidence they can replicate it. When the entropy of the problem signature is low, it indicates that there is a gross propensity for a certain subset of users to report it while other users rarely do. Signatures that have low entropy values may indicate an anomaly in the specific user and not in the application model themselves.

$$P_{userB\_P} = \frac{f_{userB\_P}}{\sum_{r=1}^{r=n} f_a} \quad \textbf{Eq. (6-2)}$$

f_{userB_P} is the number of times user b has reported problem signature P and f_a is the frequency of the same problem signature by any user. P_{userB_P} is the probability of f_{userB_P} over all the users.

Eq. (6-2) produces the input for Eq. (6-1) which in turn is used to determine if a problem signature is evenly distributed among its users. It is designed to show the probability of a user reporting the problem among all the users that report it. If the probability for each user is comparable, it will lead to a high entropy in Eq. (6-1) meaning the problem has a good chance of being replicated among those users. A high probability for a few of the users that report the problem signature would lead to low entropy in Eq. (6-1) meaning developers would need to rely on certain individual which makes replication harder. f_{userB_P} is the number of times userB report problem and f_a is the total number of time any user reported problem P. P_{userB_P} is the probability that a user b in the population set will report that specific problem signature P.

6.1.3 Entropy Distribution of Problem Signatures

The entropy distribution of a problem signature shows whether a single problem signature is evenly distributed among a set of people (i.e. high entropy) as shown in the previous section. However only using the entropy values does not reveal the extent of the problem signature's coverage. Combining the entropy distribution result with the total frequency determines the overall effect a problem signature has on the population in general. For example if problem signature 1 was reported by 3 out of a possible 100 users, each with a frequency of 4 reports (i.e. total frequency of 12 for all users), it would have high entropy but poor frequency, hence poor population coverage due to the insignificant percentage of users reporting. However if the same signature is reported by all 100 users, with 4 reports (i.e. frequency of 400), the overall population coverage is much more significant.

Problem signatures are categorized to determine the ease in which the majority of them can be replicated. Each problem signature for the product is categorized depending on how strong its

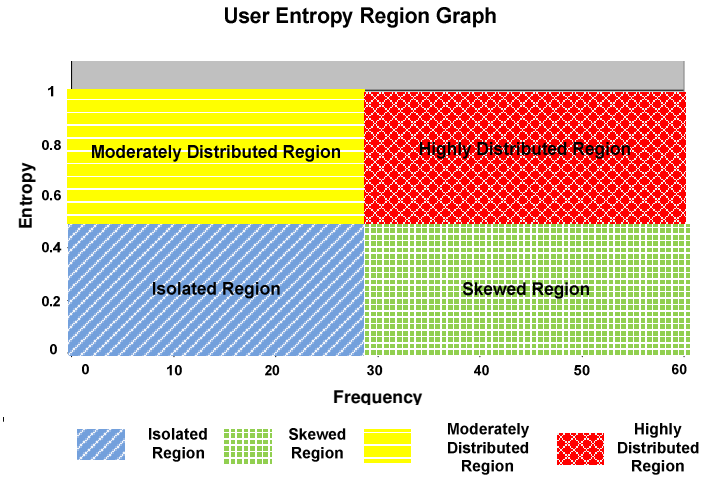


Figure 6-2. PSD entropy graph regions

entropy distribution (i.e. above a certain threshold) is as well as the total frequency of its occurrence. Because the number of unique problem signature reported can potentially be numerous, we visualize all the problem signatures as points according to their entropy distribution value and frequency. This makes it easier to see the general disposition of problem signatures among all the users. Entropy distribution points are plotted on a region graph as shown in figure 6-2. Each point on the graph describes how good their entropy is versus how frequently it is reported among the user population. The x-axis represents the total number of times a problem signature is reported regardless of the user. The y-axis represents the entropy distribution of the problem signature. As the frequency and entropy values increase across both axis, the probability that the problem signature covers a larger population increases. Having all the points on the graph can tell if an application is prone to isolated users reporting infrequent issues or whether it contains issues that are seen by everyone in significant frequency.

As shown in figure 6-2, the graph is split into regions in order to display the different priorities of the entropy distribution problem signatures. Table 6-1 summarizes the possible regions a problem signature can land and their different characteristics. The size of the region on

Table 6-1. Summary of Entropy Regions

Region	Entropy	Frequency	Result
Highly Distributed	High	High	Problem signature is reported with high frequency and is well distributed among users indicating more users can report it consistently. Developers will be able to easily replicate by selecting from testing base.
Skewed	Low	High	Problem signature is reported frequently however the distribution is skewed towards certain individuals. Developers may have to rely on certain beta users.
Moderately Distributed	High	Low	The problem signature is only reported by a select number of individuals but they report it evenly. This indicates a specific user cluster that the developer has to recruit from in order to replicate.
Isolated	Low	Low	Least desirable situation from developers point of view. Problem is obscure and hard to replicate on any application. Developers will have the hardest time resolving this problem.

the x-axis is determined by the size of the user population and the maximum frequency of reports for a problem signature. The size of the two regions for the x-axis are divided in half. The size of the region on the y-axis where the maximum value is 1 is divided in half. We use the previous example of problem signature 1 with 100 users to highlight its usage. Since every user reports the problem signature 4 times, f_{userB_1} is equal to 4. The total number of times that the problem signature is reported by any user is 4*100=400 times. Therefore, the probability that any user in that set will report the problem is 0.01. The entropy distribution is then calculated using Eq. (6-1) with all values of P_{userB_1} as input. Since there is perfect distribution the entropy will be 1. The problem signature would be mapped onto figure 6-2 with an x value of 400 and y value of 1. Table 6-1 summarizes the regions of figure 6-2 and their significance.

The unique aspect of entropy distribution analysis is its ability to categorize individual problem signatures in terms of its severity among the population. It also gives a sense of how reliable the user population is in general. For example if a region graph had most of its individual problem signatures in the risk region, the population in general is fairly reliable. However if the majority of problem signatures were in the safe region, the testing population as a whole would be fairly useless. If the application was proven to be reliable (i.e. error free) then having problem signatures in the safe region is actually desirable but for beta testing, this shouldn't be the case.

Problem signatures are not considered to be high priority unless their frequency (i.e. coverage) and entropy distribution values are significant. For example if a problem signature had high frequency but low entropy, it means the problem signature is only seriously affecting a small proportion of users that report it and is more likely to be specific to the user's application itself. Conversely, a problem signature with high entropy but low frequency means it is well distributed among the users that report it, but does not extend to the majority of users and therefore is not significant overall. To determine whether a problem signature is high priority, we take the

product of its frequency and entropy. We then compare all the problem signature's products to see which have the highest product values and assign the priority accordingly. The entropy of these high priority problem signatures are also checked to see they are sufficiently large and is not influenced only by a large frequency value. Eq. (6-3) shows how to find the product entropy of a problem signature and by extension its priority.

$$product\_entropy = f_p \times entropy \qquad \textbf{Eq. (6-3)}$$

f_p is the total frequency of problem signature p reported by any user. Entropy is output calculated in Eq. (6-1) for problem signature p.

6.2 Identifying User Clusters

To ease the replication of a problem signature, it is desirable that a developer can have a group of alternative users to replay the problem signature. User clusters can be identified to show the group of the users who are most likely to report the same problem signatures. We also analyze the frequency of the problem signature to show if the majority of the cluster can replicate the problem signature.

Figure 6-3 lists the components in the user cluster graph. Each node represents a unique user. Users that commonly report the same problems as each other are clustered together. Nodes that are connected by edges indicate a relationship. In this case, it indicates that two users report the same problem at least once. It is not concerned with the frequency in which they report the same problem signature but the number of unique problem signatures they share. User nodes in a cluster are characterized by the total frequency of problem signatures shared with other users in the cluster. In figure 6-3, User1 shares 3 problem signatures in common within User2 and User 2

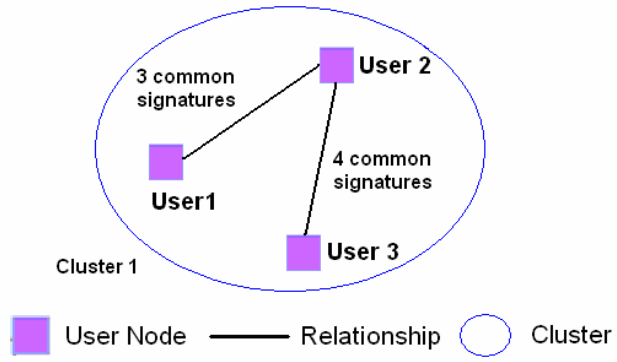


Figure 6-3. Example cluster filtering using GUESS

shares 4 problems in common with User3. Therefore the frequency is 3, 7 and 4 for User1, User2 and User3 respectively. We use a hierarchical clustering technique to determine the optimal clusters for each population set. This algorithm is used by a software clustering process which determines the users that are most similar in terms of call patterns while attempting to reduce the number of edges that crosses between clusters.

Many distinct user clusters indicate many subsets of users all use their application in their own way and problems are well distributed. This means developers can simply choose any from a specific cluster. If there are many clusters but only a few are large in size then it means only those few clusters really stand out as having problems. It would be harder in general for developers to replicate the problem.

A low number of clusters indicate users cannot really be differentiated. While users have all reported similar problem signatures, it is not really clear which ones they report with any real confidence. As a result, developers will have a harder time selecting the proper user for replication.

The most desirable scenario is for every user within a cluster to report the same number of distinctive problems with everyone else (i.e. perfect interconnectivity and therefore perfect confidence in similarity).

6.3 Case Study for Analyzing the Priority of Problems

To demonstrate that our analysis can assess the priority of problem signatures and automatically identify the users that report them, we performed a case study using four beta versions of a large scale enterprise software application used by millions of users worldwide for communication. Table 6-2 shows the descriptive statistics of each version. The data collected for this study was gathered over the course of 30 days within a field test of each version. To ease the comparison of problem signatures of all versions, identical problem signatures appearing in both versions are assigned the same identifier.

Table 6-2. Statistics of the versions for the case study

Application	Number of clusters reported	Number of Problem Signatures Reported	Number of users
Version A	6	161	367
Version B	2	342	182
Version C	3	883	588
Version D	10	284	1302

The objective of this case study is to

1) Demonstrate that the Product Analysis (based off of product entropy) is a better analysis technique to traditional prioritizing techniques.
2) Provide a visualization technique where developers can visualize the distribution of a certain problem signature among the user population. This allows developers to select users reporting specific problem signatures without error.

6.3.1 Steps for prioritizing problems using product analysis

The study makes use of an existing automatic logging system that captures log data from applications. The entropy analysis results are generated as follows:

1) For each version we record the problem signatures reported and the user that made the report. We accumulate the statistics of each problem signature including the total number of reports made which will help in statistics calculated such as frequency. We also record the unique users reporting it and the reports made by each user.
2) We calculate the entropy of each problem signature based on the statistical data for each version.
3) We rank problem signatures based on multiple criteria such as frequency and unique users to determine the priority of problem signatures. We use our product analysis to determine the priority of problems and determine the overlap of problems identified from other criteria. We select the top problem signatures based on product analysis criteria and automatically identify the users reporting them. Edge weight is calculated for users based on similarity of problem signatures reported.
4) We automatically calculated the user clusters based on the edge weights found in step 3 by using hierarchical clustering techniques implemented in the Bunch clustering tool [10] and visualize the results. User found to have reported the specific problem signature in step 3 are highlighted in the visualization.

6.3.2 Setup of the Experiment

To compare the performance of our analysis in identifying the priority of problem signatures, we conduct three types of analysis to prioritize problem signatures based on different characteristics: Frequency of reported problem signature, the number of unique users reporting the problem signature and the product entropy and frequency as described in section 6.1.

1) Unique User Analysis (UA):

This analysis determines the number of unique users that reported a given problem signature regardless of the number of times the user reported it. A higher number of users reported for the problem signature would give it higher priority in this analysis since more users reporting indicates the problem signature has a larger sphere of influence. If a problem signature is isolated to certain individuals, it is most likely a specific case, where a larger number of users affected indicates a fundamental problem in the software. This type of analysis can prioritize problem signature in terms of their affected user base.

2) Frequency Analysis (FA):

This analysis finds the total number of reports for a given problem signature regardless of the user that reported it. A high number of reports would give the problem signature a higher priority in this type of analysis. Frequency determines the overall prominence of a problem signature. Problems with high frequency are higher priority because they are higher recurring problems. This means users are more likely to experience these

problems specifically. Developers would be interested in frequency because it could potentially tell which problem signature is most often encountered by users.

3) Product Analysis (PA):

This analysis involves finding the priority of a problem signature by considering the 1) frequency of a problem signature and 3) the entropy of a problem signature. Product analysis checks to see if each of the characteristics stated above are high as shown in Eq. (6-3). Product analysis gives the largest values when the entropy is large (meaning the problem is well distributed) and the frequency that each user reports is high. This signifies the users that report the problem signature do so frequently which cumulatively leads to a large frequency value overall. Problem signatures that have high entropy but low frequency would yield a small product indicating only a small population reports it evenly. However a high entropy and frequency indicates not only is it well distributed but a large population reports it as well.

In general, problem signatures that ranked high in more than one analysis would have a higher priority but developers may consider some conditions trivial. We rank the problem signatures according to each separate analysis. We select the top 20 problem signatures from each analysis. Having high frequency and entropy instead of just high frequency will naturally increase its priority. We then show the corresponding visualization of users affected by the problem signatures found in the overlap.

The percentage of problem signatures common between product analysis and another (i.e. **UA**∩**PA** or **FA**∩**PA**) shows that the separate analysis is not as encompassing as **PA**. This is

because **PA** can find problem signature with high number of users or frequency or both while individually **UA** and **FA** can only focus on one criteria. We highlight the different combinations of priority that developers can use to assess their problem signatures:

- The percentage of problem signatures from the union of criteria between **PA** (i.e. **(UA∩PA) ∪ (FA∩PA))** shows the problem signatures that either are the most predominant with good entropy or are the best distributed in the user population.
- The percent of problem signature identified by **PA** but unaffected by intersecting criteria (i.e. **PA ∩ ^(UA ∪ FA))** gives an example of how developers can also filter criteria from certain analysis and focus purely on a single characteristic. It shows the percentage of problem signatures that are exclusively reported in product analysis but never reported in unique user analysis or frequency analysis. The exclusion is desirable among users if developers do not want the analysis influence by secondary factors.

6.3.3 Analysis of Results

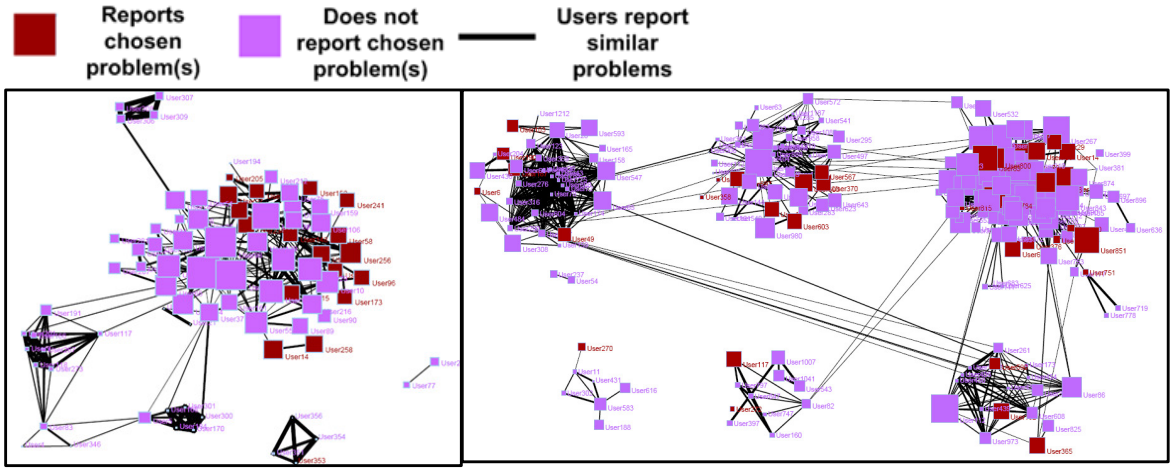
Table 6-3. Percentage of the top twenty problem signature common in different analysis

	Percentage of problems common to UA and PA	**Percentage of problems common to FA and PA**	**Percentage of problems from the union of (UA∩PA) and (FA∩PA)**	**Percentage of problems in PA but not in UA or FA**
Version A	0.3	0.75	0.75	0.25
Version B	0.9	0.85	0.95	0.05
Version C	0.35	0.45	0.55	0.45
Version D	0.4	0.65	0.65	0.35
UA: Unique User Analysis **FA: Frequency Analysis** **PA: Product Analysis**				

In comparison with the more traditional analyses (i.e., **UA** and **FA**), our analysis can identify the top priority problem signatures by considering the overall characteristics of the problems: entropy and frequency of the occurrences

Table 6-3 shows the percent overlap among problem signatures across the different types of analysis for all the applications analyzed. We show the percentage of problem common to **UA** and **PA** as well as **FA** and **PA** to highlight the fact only a fraction of the problem signature covered by the **PA** analysis will show up in **UA** or **FA** analysis. For example, in Version A, only 30% of problem signatures identified by **UA** are part of **PA.** With the exception of Version B, all versions have little overlap. The problem signatures identified by **UA** and **FA** but not **PA** are not as high priority because they would have a deficient characteristic (i.e. either low frequency for **UA** or low number of users for **FA**). This reinforces our point that individually UA and FA cannot identify as many high priority problem signatures as PA can.

The main purpose of showing multiple analysis is to emphasize how our analysis can find problem signatures that may not necessarily fall into certain criteria. For example the **(UA∩PA)** analysis included top priority signatures from **UA**. However if the developers only wanted to see problem signature unaffected by **UA** they could not use that analysis or just a PA analysis. Therefore we include the analysis **PA ∩ ^(UA ∪ FA)** to show the problem signature exclusive to PA only.

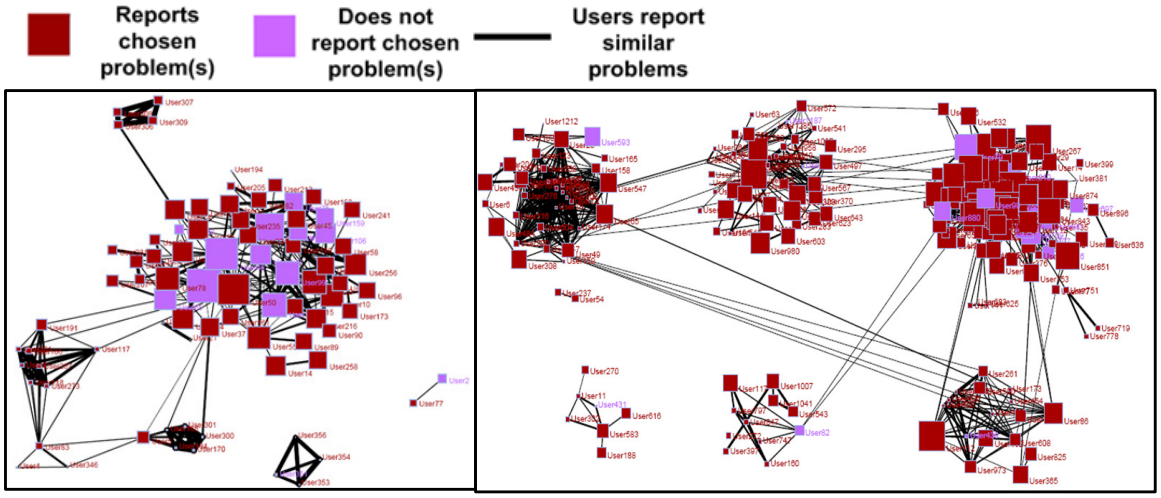


a) Affected users in Version A **b) Affected users in Version D**

Figure 6-4. Frequency analysis to show example exposure among users

6.3.4 Distribution of problems among user clusters

After developers have identified problem signatures based on their analysis in 6.2, they would want to see the distribution of the problem signatures among the user population. The visualization would help them identify candidates for replication as well as showing the different sets of users affected when choosing different problem signatures. We also show the visualizations to demonstrate that the problems identified by product analysis have higher user coverage and frequency than problems identified simply by frequency or unique user analysis. It would also tell developers if a problem signature is isolated to a certain user cluster. To show product analysis is a better technique we show the contrast of visualizations between **FA** and **PA**. We select versions A and D from table 6-3 for visualization to show these examples. For every figure, the top five problem signatures in their category are chosen to observe their distribution among the users. Dark highlighted user nodes indicate users that report at least one of the



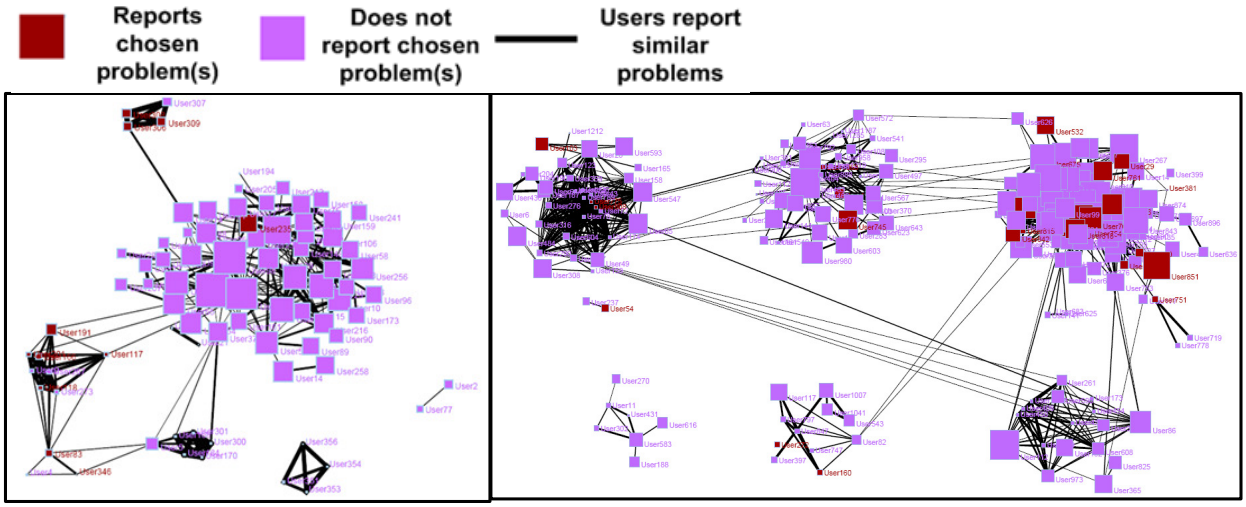
a) Affected users in Version A **b) Affected users in Version D**

Figure 6-5 Product analysis showing more users affected than frequency analysis

problem signatures chosen while light non-highlighted user nodes means those users report none of the problem signatures selected. Edges within the user graph indicate the adjoining users report similar problems but not necessarily the chosen problem signatures. The number of user nodes highlighted in each figure is dependent on the pervasiveness of the problem signatures (i.e. how much influence they exhibit). Table 6-4 summarizes the average frequency of top priority problems reported by affected users in these examples.

Table 6-4 Average frequency of reporting top priority problems by users affected

	Average reporting frequency per affected user in figure 6-4	**Average reporting frequency per affected user in figure 6-5**	**Average reporting frequency per affected user in 6-6**
Version A	26.0	14.5	134.3
Version D	41.6	27.6	95.8



a) Affected users in Version A **b) Affected users in Version D**

Figure 6-6. Overlap of users from problem signatures common to FA and PA.

Figure 6-4 a) and b) shows the user clusters for Versions A and D respectively where the highlighted sections are the users that report five problem signatures from **FA**. This is the subset of the user population that reports problem signatures frequently reported but not necessarily in a distributed fashion. While each of the users highlighted report the selected problem signatures with high frequency, it should be noted that the coverage of users is not significant to the total number of users present in the visualization. This indicates that the problem signatures identified are isolated to certain individuals, which means the problem signature may not be as high priority as the distributed case.

To contrast the previous results, we show the top five problem signature from the **PA** analysis. Figures 6-5 a) and b) shows the same user clusters where the highlighted sections are for users that report the top five problem signatures from **PA**. While the **FA** visualization from figures 6-4 a) and b) show users that report problem with high frequency, there are clearly more users affected by the identified problem signatures in figures 6-5 a) and b), making them higher priority. The users highlighted in this analysis will also have a higher probability of frequently

reporting the problem signatures. Visualizations from **UA** analysis may have a comparable number of users highlighted to **PA**, however the probability of each user reporting the problem signature frequently could be lower. This visualization reaffirms the better coverage of product analysis over frequency or unique user analysis.

Figure 6-6 a) and b) shows the same user clusters where the highlighted users are for those that report the top five problem signatures common in **FA** and **PA**. We show this visualization to demonstrate frequency analysis has insignificant overlap from users identified with product analysis (as shown in figure 6-5 a) and b)). It also demonstrates that developers cannot rely on the same users to replicate different problems regardless if they are considered high priority or not. While product analysis provides the most useful information both statistically and visually, the visualization technique does not show the advantage of using product analysis over unique user analysis. When visualizing top problems identified in both analysis, the number of users highlighted within the visualization would be comparable. However the users highlighted by the product analysis would report the problem signatures with greater frequency than unique user analysis but there is no way to visualize the frequency. Even if statistically, product analysis is more useful, the visualizations for product and unique user analysis would look the same.

6.4 Chapter 6 Summary

In this chapter we showed a technique that can combine the frequency and entropy of a problem to find its relative priority among users to other problems. We also devised a way to visualize the problem distribution in order to help developers quickly identity suitable users for replication. The next chapter summarizes the work done in previous chapters and the contributions of the thesis as a whole.

Chapter 7
Conclusion and Future Work

We present several techniques for mining log information in bulk in order to extract new information that otherwise may not be found individually. We defined IFFP, FMR, FAR and OFR metrics to indicate the quality of software after release, identified several useful patterns from an abstracted view of the log data and showed that product analysis is an effective way to prioritize problems. The effectiveness of our techniques is shown by performing various case studies on beta versions of software applications used by millions of people worldwide. In this section, we summarize the most significant contributions of our thesis and discuss the future work.

7.1 Thesis Contributions

In this section we discuss the major contributions of the thesis. The most significant was the providing techniques to 1) evaluate user perceived quality, 2) propose three graph types to highlight relationships between users and problems 3) create visualization schemes to provide insight into the nature of problems and 4) evaluate the priority of problems.

Establish metrics to predict the user perceived quality in software if released in current state: Determining the disposition of users for software before release into market is important to see if it is ready for production. We proposed four metrics to evaluate unique aspects of user experiences. Using the metrics, we quantify the failures reported in the testing period of thirty day's testing period. The results of metrics can be used to determine if the software is ready for release or is still premature by scoring the distribution of failures that would be encountered by the average user for the software application. The case study shows that proposed metrics are

sufficiently independent from each other and MTBF to convey unique information. It can predict the state of bugs in software after the use of 6 months, 1 year and 2 years respectively. We also show that the metrics can also predict the disposition of users by correlating the metric scores to the average usage period of each version.

Propose three types of graphs for easy identification of problem patterns: Many problems may seem like isolated or trivial cases until they can be linked to other factors to create patterns. Many of these patterns cannot be seen unless all the data is pooled together into a high level view. We proposed three types of graphs for visualizing the field testing results and testing users. The graphs help provide a high level view of the large amount of field testing results. We identify five patterns which help managers and developers make best use of the results of the field testing and improve future field testing efforts. For example, developers can identify problematic and representative users to help in replicating problems or planning future testing efforts. We automate the identification of patterns to reduce the complexity of the analysis for practitioners.

Create a visualization technique to identify the origins of problems: Field testing is an important phase of quality improvement processes in quality-driven software organization. We demonstrate the importance of having a high level overview of the structure of problems. We presented a visualization which developers can use to understand the interdependencies between reported problems. Using this knowledge, they can pinpoint areas of the code which require additional error checking or additional documentation. They can also prioritize their bug fixing efforts and determine which problems are easier to replicate due to the small number of interacting problems.

Propose an analysis technique that can determine the priority of problems: The distribution and priority of problems is largely unknown to developers when they are initially reported by users. We define problem signatures in order to eliminate uncertainty whether certain problems are similar or not. Our analysis technique provides flexibility to developers for determining the priority of problem signatures by allowing them to combine multiple criteria such as frequency and unique users. We also introduce product entropy as one of the possible criteria for determining the priority of problem signatures. The visualization technique allows developers to see the actual distribution of different problem signatures among their users and helps them to recruit users for replicating problem signatures with greater efficiency. We create an automated process that uses clustering algorithms to sort the users into clusters based on similar problem signatures to identify which users are most related.

7.2 Limitations and Future Work

There are several avenues in our work that have not been fully explored yet. In the future, the work can be extended in the following directions:

Determine if metrics reflect user opinions in other applications: In the future, we plan to conduct user studies to evaluate the accuracy of the metrics in predicting the user perceived quality of software. Developers would use the metric framework to generate data on their user's experience and determine if the results positively affect their decision when to release their software into the market.

Increase the repertoire of patterns identified: Our visualization approach is designed to help developers eliminate problems with greater efficiently. We plan to conduct an empirical study to

investigate the improvement in field testing effort using the proposed approach. We plan to examine more versions of software to extend our catalog of patterns.

Determine effectiveness of the product analysis in identifying high priority problems: The entropy analysis technique is used to help developers find the problems that deserve the greatest attention according to their criteria. The visualization technique allows developers to eliminate uncertainty for recruiting users by showing them the exact users who reported a specific problem. In the future we plan to conduct empirical studies to determine if the number of severe bug reports after deployment were fewer than predicted after applying our analysis technique. We also plan to conduct surveys as to whether developers found it easier to recruit users for replication using our visualization technique. In addition, we hope to increase the accuracy of the visualization technique by giving different color schemes to users who reported a problem with varying frequencies instead of using only one color to identify all users regardless of frequency.

References

[1] W. Aalst and M. Song. Discovering Social Networks from Logs. *In Computer Supported Cooperative Work*, Pages 549-593. May 2005.

[2] AiSee Graph Layout Software, http://www.aisee.com/. Last access on 2nd August 2009.

[3] S.V. Amari. Bounds on MTBF of Systems Subjected to Periodic Maintenance. *IEEE Transactions on Reliability*, SE-55(3):469-474, September 2006

[4] N. Anquetil and T. Lethbridge. File clustering using naming conventions for legacy systems. *Proceedings of CASCON 1997*, pages 184-195, Toronto, Ontario, Nov. 1997.

[5] J. Anvik, L. Hiew, Gail. C. Murphy. Coping with an open bug repository. *Proceedings of the 2005 OOPSLA worshop on Eclipse technology eXchange*, Pages 35-39, San Diego, California, Aug. 2005

[6] F.T. Au, S. Baker, I. Warren, G. Dobbie. Automated usability testing framework. *Proceedings of the 9th conference on Australasian user interface*, 76:55-64, Wollongong, Australia, Jan 2008.

[7] A. Bianchi, D. Caivano, F. Lanubile, and G. Visaggio. Evaluating software degradation through entropy. *Proceedings of the Eleventh International Software Metrics Symposium*, pages 210–219, London, UK, 2001.

[8] J. Bowring, A. Orso, M. J. Harrold. Monitoring deployed software using software tomography. *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program Analysis*, Pages 2-9, South Carolina, USA, Nov 2002.

[9] M. Buckley and R. Chillarege. Discovering relationships between service and customer satisfaction. *Proceedings of the International Conference on Software Maintenance*, pages 192 – 201, Opio, France, October 1995.

[10] Bunch Clustering Tool Software, http://serg.cs.drexel.edu/redmine/projects/show/bunch. Last access on 4th August 2009.

[11] N. Chapin. An entropy metric for software maintainability. *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, Software Track, pages 522–523, Jan. 1995.

[12] N. Chapin. Entropy-metric for systems with COTS software. *Proceedings of the Eighth IEEE Symposium on Software Metrics*, pages 173–181, Ottawa, Canada, June 2002.

[13] R. C. Cheung. A User-Oriented Software Reliability Model. *IEEE Transactions on Software Engineering,* SE-6(2):118-125, March 1980.

[14] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults based on field data. *Proceedings of the Twenty Sixth Annual International Symposium on Fault Tolerant Computing*, pages 304-313, Sendai, Japan, June 1996.

[15] M. Cinque, D. Cotroneo, and S. Russo. Collecting and Analyzing Failure Data of Bluetooth Personal Area Networks. *Proceedings of the 2006 International Conference on Dependable systems and Networks (DSN'06),* pages 313-322, Philadelphia, Pennsylvania, June 2006.

[16] J. Cohen. Statistical Power Analysis for the Behavioural Sciences. 1988.

[17] G.W. Corder, D.I. Foreman. Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach. Wiley. 2009

[18] W. Dickinson, The Application of Cluster Filtering to operational testing of Software. Doctoral dissertation. Case Western Reserve University. May 2001.

[19] W. Dickinson, D. Leon, and A. Podgurski, Finding failures by cluster analysis of execution profiles. *Proceedings of the 2001 International Conference on Software Engineering*, pages 339- 348, Toronto, Ontario, May 2001.

[20] W. Everett. Software Component Reliability Analysis. *Proceedings of the 1999 IEEE Symposium on Application,* pages 204-211, Richardson, Texas, March 1999.

[21] J. Froehlich, M. Chen, S. Consolvo, B. Harrison, J. Landay, "MyExperience: A System for In situ Tracing and Capturing of User Feedback on Mobile Phones", *International Conference in Mobile Systems, Applications, and Services,* 2007.

[22] M. P. Gonzalez, J. Lores, A. Granollers. Enhancing usability testing through datamining techniques: A novel approach to detecting usability problem patterns for a context of use. *Information and Software Technology,* SE-50(6):547-568, May 2008.

[23] GUESS: The Graph Exploration System. http://graphexploration.cond.org/. Last access on June 2009.

[24] J. P. Guilford. The phi coefficient and chi square as indices of item validity. Springer. 1941.

[25] S. Abd-El-Hafiz. Entropies as measures of software information. *Proceedings of the 2001 International Conference on Software Maintenance*, pages 110–117, Florence, Italy, 2001.

[26] G. Harald, J. Mehdi, R. Claudio. Visualizing Software Release Histories: The Use of Color and Third Dimension. *Proceedings of the 1999 IEEE International Conference on Software Maintenance.* Pages 99-108, Oxford, England, August 30th – September 3rd 1999.

[27] W. Harrison. An entropy-based measure of software complexity. *IEEE Transactions on Software Engineering, 18(11):1025–1029*, Nov. 1992.

[28] A. E. Hassan and R. C. Holt. The Chaos of Software Development. *In IEEE International Workshop on Principles of Software Evolution (IWPSE03)*, Pages 84-94, Helsinki, Finland, Sept. 2003.

[29] L. Hochstein, V. R. Basili, M. V. Zelkowitz, and J.K. Hollingsworth. Combining self-reported and automatic data to improve programming effort measurement. *Proceedings of the 10th European Software Engineering Conference on the Foundations of Software Engineering*, pages 356-365, Lisbon, Portugal, September 2005.

[30] D. Hovemeyer, and W. Pugh. Finding Bugs is Easy. *Proceedings of the 19th ACM SIGPLAN Conference*, Pages 132-136, October 2004.

[31] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, pages 947-757, Aug. 1985.

[32] J.A. Jones, M.J. Harrold and Stasko, J. Visualization of test information to assist fault localization. *Proceedings of the 24th International Conference on Software Engineering (ICSE'01)*, Pages 467-477, Orlando, Florida, 2001.

[33] K. Kim, Y. Shin and C. Wu, "Complexity measures for object-oriented program based on the entropy," *Proceedings of the Asia Pacific Conference on Software Engineering*, pp. 127-136, Brisbane, Australia, Dec. 1995.

[34] J. Lamping, R. Rao and P. Pirolli. A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies. *The ACM SIGCHI Conference on Human Factors in Computing Systems*, Pages 401-408, Denver Colorado, May 1995.

[35] H.O. Lancaster. Chi Squared Distribution (Probability & Mathematical Statistics). Wiley. 1969.

[36] D. Leon, A. Podgurski and White, L. J. 2000. Multivariate visualization in observation-based testing. *Proceedings of the 22th International Conference on Software Engineering (ICSE'00)*, Pages 116-125, Limerick, Ireland, 2000.

[37] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, and M.I. Jordan. Scalable Statistical Bug Isolation. *Proceedings of the 2005 SIGPLAN Conference on Programming Language Design and Implementation*, Pages 15-26, Chicago, Illinois, June. 2005.

[38] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, and M.I. Jordan. Bug isolation via Remote Program Sampling. *Proceedings of the 2003 SIGPLAN Conference on Programming Language Design and Implementation*, Pages 141-154, San Diego, California, June. 2003.

[39] J. D. Musa, Operational Profiles in Software-Reliability Engineering. *IEEE Transactions*, *10(2): 14-32*, March. 1993.

[40] A. Mockus. Empirical estimates of software availability of deployed systems. *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, Pages 222-231, Rio de Janeiro, Brazil, September 2006.

[41] M. G.H. Omran, Andries P. Engelbrecht, A. Salman. An overview of clustering methods. *Intelligent Data Analysis Vol 11 Issue 6*, December 2007, Pages 583-605.

[42] A. Orso, J. A. Jones, M. J. Harrold. GAMMATELLA: Visualizing program-execution data for deployed software. *Proceedings of the 26th International Conference on Software Engineering*, Pages 699-700, Scotland, UK, 2004.

[43] A. Orso, D. Liang, M.J. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02)*, Pages 65-69, Rome, Italy, 2002.

[44] P. Pantel, D. Lin. Document clustering with committees. *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and development in information retrieval*, Pages 199-206, Tampere, Finland, August 2002.

[45] J. L. Rodgers and W. A. Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42(1):59–66, Feb 1988.

[46] A. Rozinat, R. Mans and W. Aalst. Mining CPN Models: Discovering Process Models with Data from Logs. *Proceedings of the Seventh Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2006)*, Pages 57-76, Aarhus, Denmark, August 2006.

[47] M. Sarkar, M. H. Brown. Graphical Fisheye Views of Graphs. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Pages 83-91, Monterey, California, May 1992.

[48] Scalet. 2000: ISO/IEC 9126 and 14598 integration aspects: A Brazilian viewpoint. *The Second World Congress on Software Quality*, 2000.

[49] N. Schmidt. Software Usability: A Comparison Between Two Tree Structured Data Transformation Languages. *Proceedings of the 3rd Nordic Conference on Human-Computer Interaction*, pages 145-148, Tampere, Finland, October 2004.

[50] I. Schwab, A. Kobsa and I.Koychev. Learning about Users from Observation. *Proceedings of the AAAI 2000 Spring Symposium on Adaptive User Interface*, Pages 241-247, Stanford, California, March 2000.

[51] C. E. Shannon, A mathematical theory of communications," *Bell System Technical Journal, Vol. 27*, 1948, pp. 379-423.

[52] M. Terry, M. Kay, B. V. Vugt, O. Slack and T. Park. INGIMP: Introducing Instrumentation to an End-User Open Source Application. *Proceedings of the 26th IGHI conference on Human factors in computing systems*, Pages 607-616, Florence, Italy, April 2008.

[53] V. Tzerpos, R.C. Holt. Software Botryology: Automatic Clustering of Software Systems. *Proceedings of the 9th International Workshop on Database and Expert Systems Applications*, Pages 811-818, Vienna, Austria, Aug. 1998.

[54] E.A. Unger, L. Harn and V. Kumar, "Entropy as a measure of database information," *Proceedings of the Sixth Annual Computer Security Applications Conference*, pp. 80-87, Tuscon, Arizona, Dec. 1990.

[55] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 33-43, Amsterdam, Netherlands, IEEE Computer Society Press, Oct 1997.

[56] S. Zhong, J. Ghosh. A unified framework for model based clustering. *The Journal of Machine Learning Research Vol 4*, pages 1001-1037, 2003

[57] Y. Zou, Q. Zhang, X. Zhao, “Improving the Usability of E-Commerce Applications using Business Processes”, *IEEE Transactions on Software Engineering*, 33(12): 837-855, December 2007.