Visualizing the Structure of Field Testing Problems

Brian Chan, Ying Zou
Dept. of Elec. and Comp. Engineering
Queen's University
Kingston, Ontario, Canada
{2byc, ying.zou}@queensu.ca

Ahmed E. Hassan
School of Computing
Queen's University
Kingston, Ontario, Canada
ahmed@cs.queensu.ca

Anand Sinha
Handheld Software
Research In Motion (RIM)
Waterloo, Ontario, Canada
asinha@rim.com

Abstract

Field testing of a software application prior to general release is an important and essential quality assurance step. Field testing helps identify unforeseen problems. Extensive field testing leads to the reporting of a large number of problems which often overwhelm the allocated resources. Prior efforts focus primarily on studying the reported problems in isolation. We believe that a global view of the interdependencies between these problems will help in rapid understanding and resolution of reported problems. We present a visualization that highlights the commonalities between reported problems. The visualization helps developers identify two patterns that they can use to prioritize and focus their efforts. We demonstrate the applicability of our visualization through a case study on problems reported during field testing efforts for two releases of a large scale enterprise application.

1. Introduction

Field testing of an application is often necessary to improve its quality before release. Alpha and beta testing are examples of field testing efforts. During a field testing phase, regular users are given an instrumented version of an application. The instrumented version reports any problems to a central repository. Crashes, exceptions, unresponsive user interface, and excessive resource (e.g., CPU, memory, and battery) usage are examples of problems. The field testing phase could last as short as a few days and as long as a few months. The central repository records for each reported problem: the state of the application (e.g., call stack), and other pertinent information (e.g., the time of occurrence and configuration and environment parameters).

Extensive field testing efforts produce a large number of reported problems. Traditional efforts often examine each problem in isolation or in groups of similar problems. 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 this paper, we propose a visualization that is of value to the developers who must repair the field problems. In contrast to testing managers who focus on prioritizing problems, 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.

The rest of the paper is organized as follows. Section 2 presents our approach to study the structure of field problems. Section 3 presents the results of a case study which is conducted to demonstrate the benefits of our proposed visualization on a large scale enterprise application. Section 4 provides a brief overview of related work. Section 5 concludes the paper and explores possible avenues for future work.

2. Visualizing the Structure of Problems

When examining a particular problem report, developers often start their investigation by studying the method that originally reported the problem. For example, a method might have reported an out-of-range exception when accessing an array element. However, the out-of-range exception is often due to a method higher-up in 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 stack and investigating each method along the path of that stack. We name that call stack the structure of a problem. The process of going up the call stack is time consuming, given the usual complexity of reported problems and their large numbers. However, one can make use of the call stacks (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

978-1-4244-4828-9/09/$25.00 2009 IEEE

might cause problems down their call path. However all these problems are linked to the other methods higher-up in the call stack. A visualization that would identify this situation would be of great value to developers.

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 lift our visualization to the file level instead of the method level. The visualization highlights the commonly occurring call paths among reported problems. Each call path represents a reported problem.

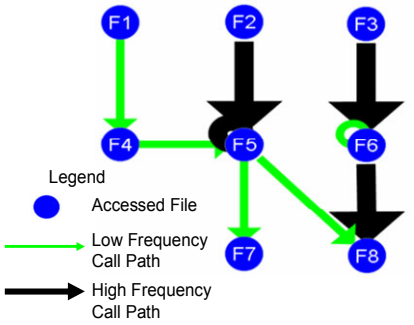

Figure 1 – Problem Structure Visualization

Figure 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 stacks of all reported problems. The set of edges, E_F, contains directed edges, $e_{ab}=\{F_a, F_b\}$, if file F_a subsequently calls file F_b. A node, in Figure 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 1, a method in node F_1 invokes 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 1, F_2 has a call edge to F_5 and F_5 has a call edge to F_7. This indicates F_2 invokes F_5 and F_5 can invoke F_7, but F_2 cannot invoke 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 are accessed 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 invoked.

Node weights are constant and do not change regardless of the number of times they are called. We add weights to the edges. Edge weight is determined by the frequency in which the call pair is invoked. 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 1, F_2 is frequently reported to invoke F_5, while F_1 rarely invokes F_4.

3. Case Study

To demonstrate the applicability of our proposed visualization, we performed a case study on the reported problems during field testing efforts for two releases of a large scale enterprise application. Table 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 1-Statistics for Studied Field Tests

Version	Users	Reported Problems	Files in Call Stack
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 invoked 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 [1], a graphical layout software. AiSee reads textual specifications and generates the visual equivalent.

3.1 Identified Patterns

In our case study, we identified two patterns prominent in both visualizations depicted in Figures 2 and 3. These patterns help identify problematic files and could be used to compare different versions.

3.1.1 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 stack.

Symptom: This pattern is characterized by one node with many edges originating from it. This indicates that the file V_F 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.

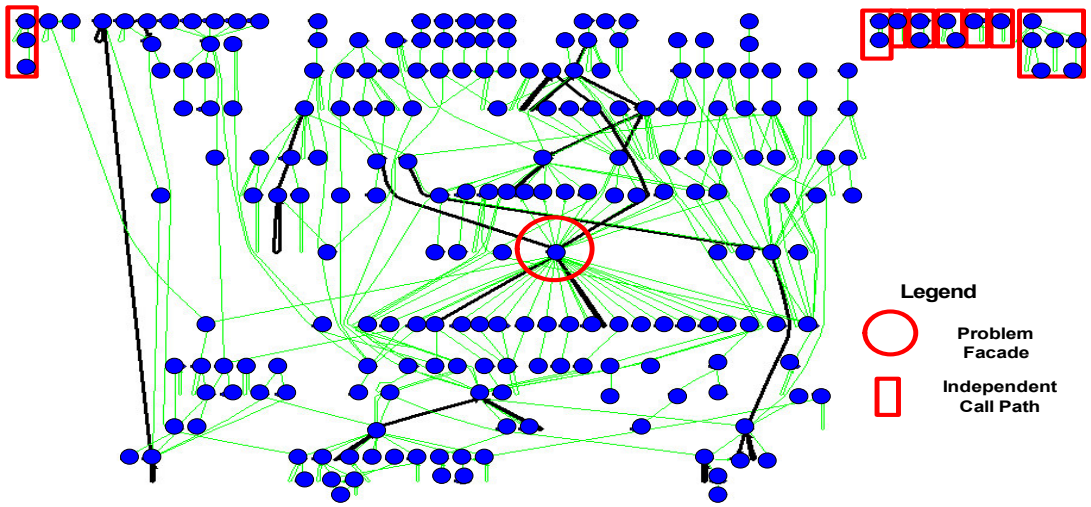

Figure 2 – The Structure of Problems for Version A

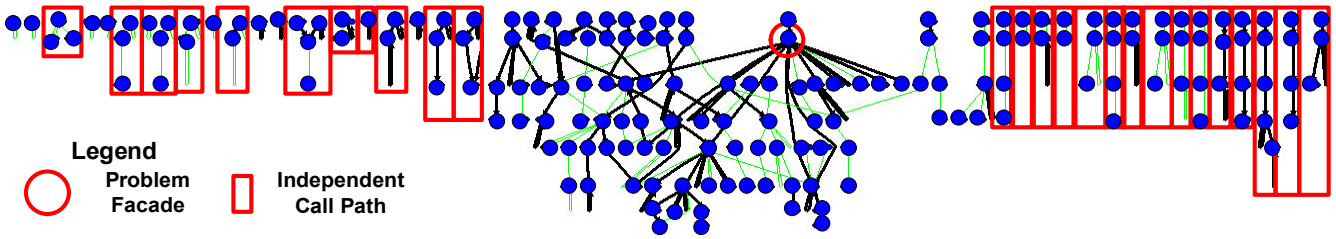

Figure 3 –The Structure of Problems for Version B

Examples: Figure 2 shows several instances of the problem façade patterns in version A. The instances are annotated using circles in Figure 2. A similar pattern occurs for version B, shown in Figure 3. For version B, the instance of the problem Façade pattern consists of a file located at the top of the call paths, indicating that the file is accessed early on in the call paths.

3.1.2 Independent Call Path

Motivation: When exploring the call stack (i.e., structure) of a particular problem relative to other problems, developers are faced with two types of situations either the call stack is significantly different than other call stacks or it shares similarities with a large number of call stacks (subsequently problems). The independent call path pattern occurs when a particular call stack is 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 invoke a small number of subsequent files.

Examples: As shown in Figures 2 and 3, both version A and B show several instances of the independent call path patterns. In version B, we note that there are a larger number of instances of that pattern. The variation between the two versions is a good indicator 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.

3.2 Comparing Two Versions

While both versions show similar identifiable patterns, they have subtle differences. The defining characteristic of version A is the long call stacks versus shorter calls stacks 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 2.

4. Related Work

There have been various attempts to visualize and study large scale systems to identify bugs. Sarkar *et al.* [9] provide fisheye visualization technique that magnifies the portion of a system. Lamping *et al.* [4] use a circular hyperbolic plane to visualize systems with a hierarchical structure. These techniques could be used in conjunction to our work to focus on certain sections of interest in the structure of field problems.

Various bug isolation and visualization techniques have been attempted. Orso *et al.* [7] have developed a tool called GAMATELLA that allows users to manipulate execution data in an interactive fashion. This tool also provides functions such as instrumentation and tree map nodes to show the most executed files. Liblit *et al.* [5] uses instrumented predictors to observe code segments and report failures if the program fails. Their approach determines the frequencies of failures and the modes of execution that cause the failure. They introduce an assertion statement framework [6] to track down faulty software. Hovemeyer *et al.* [3] isolate bugs by creating a bug pattern detector. The detector analyzes the code structure and determines the sections that match known faulty patterns (i.e. structures which are statistically prone to failure). These techniques could help us pinpoint the source of a bug based on files that are flagged by our visualization.

Bugs are often introduced due to different author contributions. Aalst *et al.* [2] apply workflow management and social network analysis on logs recording the execution of activities in business processes. Their study uses data produced from information systems similar to our work. However, their work focuses on the performer information instead of the execution activities. Rozinat *et al.* [8] use a visualization scheme to map relationships between components in a simulation model, called Colored Petri Net (CPN). Similar to our technique, their models can map a sequence of events (i.e., call stack) into a visual format. However, additional data, such as time and resources can be annotated in the model. The results of our technique could potentially be enhanced with CPN in order to gain more insight into call stack relationships.

5. Conclusion and Future Work

Field testing is an important phase of quality improvement processes in quality-driven software organization. The number of reported problems during field testing is often very large. Prior work primarily focuses on exploring each problem in isolation. In this paper, 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. In future work, we plan to conduct a user study to evaluate if the identified patterns improve the productivity of developers when investigating and fixing reported field problems.

References

[1] AiSee Graph Layout Software, http://www.aisee.com/

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

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

[4] 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. May 1995.

[5] 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, June. 2005.

[6] 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, June. 2003.

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

[8] A. Rozinat, R. Mans and W. Aalst. Mining CPN Models: Discovering Process Models with Data from Event Logs. In Workshop and Tutorial on Practical Use of Colored Petri Nets and the CPN. Pages 57-76. May 2006.

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