Asian Journal of Electrical Sciences
ISSN: 2249-6297 (P) Vol.11 No.2, 2022, pp.24-29

DOI: https://doi.org/10.51983/ajes-2022.11.2.3545

Bug Tracker: A Comprehensive Bug Tracking and Modification System

K. Vasumathi[1], S. Selvakani[2] and P. Sridhar[3]
[1]Assistant Professor, [2]Assistant Professor and Head, [3]Research Scholar,
PG Department of Computer Science, Government Arts and Science College, Arakkonam, Chennai, Tamil Nadu, India
E-mail: kulirmail@gmail.com, sselvakani@hotmail.com, sridharbagaveli@gmail.com
(Received 17 August 2022; Accepted 6 October 2022; Available online 12 October 2022)

***Abstract* - It is important that information provided in bug reports is relevant and complete in order to help resolve bugs quickly. However, often such information trickles to developers after several iterations of communication between Developers and reporters. Poorly designed bug tracking systems are partly to blame for this exchange of information being stretched over time. Our paper addresses the concerns of bug tracking systems by proposing four broad directions for enhancements. As a proof-of-concept, we also demonstrate a prototype interactive bug tracking system that gathers relevant information from the user and identifies files that need to be fixed to resolve the bug.**
Keywords: **Bug, Resolve, Developer, Reporter, Bug Tracking, Exchanging Information, Fixed Bug**

I. INTRODUCTION

The use of bug tracking systems as a tool to organize maintenance activities is widespread. The systems serve as a central repository for monitoring the progress of bug reports, requesting additional information from reporters, and discussing potential solutions for fixing the bug. Developers use the information provided in bug reports to identify the cause of the defect and narrow down plausible files that need fixing. A survey conducted amongst developers from the APACHE, ECLIPSE, and MOZILLA projects found out which information items are considered useful to help resolve bugs. Items such as stack traces, steps to reproduce, observed and expected behaviour, test cases, and screenshots ranked high on the list of preferred information by developers.

Previous research has shown that reporters often omit these important items. Developers are then forced to actively solicit information from reporters and, depending on their responsiveness, this may stall development. The effect of this delay is that bugs take longer to be fixed and more and more unresolved bugs accumulate in the project's bug tracking system. We believe that one reason for this problem is that current bug tracking systems are merely interfaces to relational databases that store the reported bugs. They provide little or no support to reporters to help them provide the information that developers need.

As researchers, we often rely on repositories of software project information as the main or only source of evidence to extract the histories of bugs and other work items. They are usually stored in the form of tickets or records in a bug database. They provide a convenient compartmentalization of work. We use project management systems' features such as audit trails and data fields that keep track of ownership and of the context of each work item. Sometimes we enrich the histories in ticketing systems with records of electronic communication among team members, and with organizational structure data extracted from human resources databases. However, to this point the use of these electronic repositories as reliable and sufficient accounts of the history of bugs or work items has not been properly validated, and we do not have a description of the common coordination dynamics underlying bug histories. This paper reports on a field study of coordination activities around bug fixing that used a combination of case study research and a survey of software professionals.

The study goes beyond the electronic repositories of software activity by talking directly to the key actors on the bugs to discover the patterns of group work that are commonly used to fix bugs.

It discusses the reliability of electronic repositories as the basis of research into the coordination of software projects and provides some implications for the design of coordination and awareness tools.

It is difficult to find and fix a software problem, and to verify the solution, without the ability to reproduce it. As an example, consider bug #30280 from the Eclipse bug database (Figure 1). A user found a crash and supplied a back-trace, but neither the developer nor the user could reproduce the problem. Two days after the bug report, the developer finally reproduced the problem; four minutes after reproducing the problem, the developer fixed it.

Software maintenance, users inform developers via bug reports which part of a software product needs corrective maintenance. For large projects with many users the amount of bug reports can be huge.

In open-source, bug tracking systems are an important part of how teams (such as the ECLIPSE and MOZILLA teams) interact with their user communities. As a consequence, users are more involved in the bug fixing process they not only submit the original bug reports but also participate in discussions of how to fix bugs. Thus, they help to make decisions about the future direction of a product. To a large extent, bug tracking systems serve as the medium through which developers and users interact and communicate.

However, friction arises when fixing bugs: developers get annoyed and impatient over incomplete bug reports and users are frustrated when their bugs are not immediately fixed.

II. REVIEW OF LITERATURE

Ralf Teusner, Christoph Matthies says that in any sufficiently complex software system there are experts, having a deeper understanding of parts of the system than others [7]. However, it is not always clear who these experts are and which particular parts of the system they can provide help with. Those a framework to elicit the expertise of developers and recommend experts by analysing complexity measures over time. Furthermore, teams can detect those parts of the software for which currently no, or only few experts exist and take preventive actions to keep the collective code knowledge and ownership high. In this employed the developed approach at a medium-sized company. The results were evaluated with a survey, comparing the perceived and the computed expertise of developers. This paper, show that aggregated code metrics can be used to identify experts for different software components. The identified experts were rated as acceptable candidates by developers in over 90% of all cases.

1. In this paper describes the idea of every programmer being able to improve any code anywhere in the system.
2. The Analyzer framework enables analyses on the expertise of developers for parts of the system based on proven code complexity measures.
3. Those are following three code complexity measurements were employed:
4. McCabe Complexity, Halstead Metrics, Coupling.
5. The Analyzer Framework these tools are extracted, transformed into a common data model, and saved in a typical Extract, Transform, Load (ETL) process, allowing analyses on well-defined data structures.
6. In order to evaluate the results of Analyzer and compare them to the expectations of the developers, a survey was devised. Survey participants were developers who volunteered. The survey consisted of two main parts:
7. Expert Selection Developers self-assessed whether they were free to name two other qualified developers as well.
8. Proposal Evaluation Developers were presented with the top three component experts identified by Analyzer and were asked to rate the accuracy of each result.

Gina Venolia, Jorge Aranda says that every bug has a story behind it [3]. The people that discover and resolve it need to coordinate, to get information from documents, tools, or other people, and to navigate through issues of accountability, ownership, and organizational structure. This paper reports on a field study of coordination activities around bug fixing that used a combination of case study research and a survey of software professionals. Results show that the histories of even simple bugs are strongly dependent on social, organizational, and technical knowledge that cannot be solely extracted through automation of electronic repositories, and that such automation provides incomplete and often erroneous accounts of coordination. The paper uses rich bug histories and survey results to identify common bug fixing coordination patterns and to provide implications for tool designers and researchers of coordination in software development.

1. A list of primary and secondary actors in the history and their contributions.
2. A list of relevant artifacts and tools.
3. A chronological list of the information flow and coordination events in the bug's history.
4. Pieces of evidence as required by the particularities of each case.
5. The history of the bug as reconstructed by its record in the bug database.

The history of the bug as reconstructed by the full collection of electronic traces we obtained. The history of the bug as reconstructed from making sense of all available evidence, including our interviews with participants. During our analysis we worked with several concepts that do not yet have a consistent definition in the literature. In particular, one could argue that our coordination patterns and goals are subjective and have blurry boundaries - we never specified, for instance, the difference between "rapid-fire" and "infrequent" emails. Although this is a valid criticism, our constructs are a first iteration given the data we collected. Additional data and further iterations should refine these constructs and add others that help convey the underlying concepts more clearly.

Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, Sunghun Kim says that in a survey we found that most developers have experienced duplicated bug reports, however, only few considered them as a serious problem [4]. This contradicts popular wisdom that considers bug duplicates as a serious problem for open-source projects. In the survey, developers also pointed out that the additional information provided by duplicates helps to resolve bugs quicker. In this paper, therefore, propose to merge bug duplicates, rather than treating them separately. To quantify the amount of information that is added for developers and show that automatic triaging can be improved as well. In addition, in this paper discuss the different reasons why users submit duplicate bug reports in the first place.

1. Often there are negative consequences for users who enter duplicates. As a result, they might err on the side of not entering a bug, even though it is not filed yet.
2. Triggers are more skilled in detecting duplicates than users and they also know the system better. While a user will need a considerable amount of time to browse through similar bugs, triggers can often decide within minutes whether a bug report is a duplicate.
3. Bug duplicates can provide valuable information that helps diagnose the actual problem.
4. Provide a feature to merge bug reports, so that all information is readily available to developers in one bug report and not spread across many.

5. Check for resubmission of identical bug reports. These duplicates are easy to catch and could be easily avoided by the bug tracking system.
6. Allow users to renew long-living bug reports that are still not fixed. Often the only way to remind developers of these bugs is to resubmit them (and thus creating a duplicate report).
7. Improve search for bug reports. Most users are willing to spend some time to search for duplicates, but not a lot. Here approaches for duplicate detection will be a valuable addition to bug tracking systems.

D. Nicolas Bettenburg, Sascha Just, Adrian Schroter says that the analysis of the 466 responses revealed an information mismatch between what developers need and what users supply [5]. Most developers consider steps to reproduce, stack traces, and test cases as helpful, which are at the same time most difficult to provide for users. Such insight is helpful to design new bug tracking tools that guide users at collecting and providing more helpful information. Our CUEZILLA prototype is such a tool and measures the quality of new bug reports; it also recommends which elements should be added to improve the quality. Those trained CUEZILLA on a sample of 289 bug reports, rated by developers as part of the survey. In this paper, CUEZILLA was able to predict the quality of 31–48% of bug reports accurately.

1. Each examined projects' bug database contains several hundred developers that are assigned to bug reports.
2. Keeping the five-minute rule in mind, we asked developers the following questions, which we grouped into three parts:
3. Contents of bug reports, Problems with bug reports, Contents of bug reports, Contents considered to be relevant.
4. Our CUEZILLA tool measures quality of bug reports on the basis of their contents. From the survey, we know the most desired features in bug reports by developers.
5. Endowed with this knowledge, CUEZILLA first detects the features listed below.
6. Terminations.
7. Keyword completeness.
8. Readability.
9. In addition to the description of the bug report, we analyse the attachments that were submitted by the reporter within 15 minutes after the creation of the bug report.
10. Code Samples.
11. Stack Traces.
12. Patches.
13. Screenshots.

Peter Fritzson, Tibor Gyimothy, Mariam Kamkar, Nahid Shahmehri says that this paper presents a version of generalized algorithmic debugging integrated with the category partition method for functional testing [6]. In this way the efficiency of the algorithmic debugging method for semi-automatic bug localization can be improved by using test specifications and test results. The long-range goal of this work is a semi-automatic debugging and testing system which can be used during large-scale program development of non-trivial programs. The method is generally applicable to procedural languages and is not dependent on any ad hoc assumptions regarding the subject program. The original form of algorithmic debugging is however limited to small programs without side-effects.

Another drawback of the original method is the large number of interactions with the user during bug localization. To our knowledge, this is the first method which uses category partition testing to improve the bug localization properties of algorithmic debugging. The method can avoid irrelevant questions to the programmer by categorizing input parameters, and match these against test cases in the test database. The algorithmic debugger traverses the execution tree and interacts with the user by asking about the expected behaviour of each procedure.

1. The user has the possibility to answer yes or no or to give an assertion about the intended behaviour of the procedure.
2. The search finally ends, and a bug is localized in a procedure p when one of the following holds:
3. Procedure p contains no procedure calls.
4. All procedure calls performed from the body of procedure p fulfil the user's expectations.

We divide our algorithmic debugging methodology into three major phases.

1. Transformation phase,
2. Tracing phase
3. Debugging phase.
4. The last phase consists of the three major components: pure algorithmic debugging, test case lookup and partitioning, and program slicing.
5. A prototype generalized algorithmic debugger for Pascal, and a test case generator for real size application programs in Pascal, C, dBase and LOTUS have been implemented.

Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, Michael I. Jordan says that in this way, present a statistical debugging algorithm that isolates bugs in programs containing multiple undiagnosed bugs [2]. Earlier statistical algorithms that focus solely on identifying predictors that correlate with program failure perform poorly when there are multiple bugs.

Our new technique separates the effects of different bugs and identifies predictors that are associated with individual bugs. These predictors reveal both the circumstances under which bugs occur as well as the frequencies of failure modes, making it easier to prioritize debugging efforts. Our algorithm is validated using several case studies, including examples in which the algorithm identified previously unknown, significant crashing bugs in widely used systems.

1. This survey, briefly report here on experiments with additional applications containing both known and unknown bugs. Complete analysis results for all experiments.
2. We analysed CCRYPT 1.2, which has a known input validation bug. Our algorithm reports two predictors, both of which point directly to the single bug.
3. For large applications the set P numbers in the hundreds of thousands of predicates, many of which are, or are very nearly, logically redundant.
4. A separate difficulty is the prevalence of predicates predicting multiple bugs.
5. Finally, different bugs occur at rates that differ by orders of magnitude. In reality, we do not know which failure is caused by which bug, so we are forced to lump all the bugs together and try to learn a binary classifier.
6. In this section we present the results of applying the algorithm described in Section 3 in five case studies. Statistics for each of the experiments. In each study we ran the programs on about 32,000 random inputs.
7. In this section we present the results of applying the algorithm described in Section 3 in five case studies. In each study we ran the programs on about 32,000 random inputs.
8. RHYTHMBOX 0.6.5, an interactive, graphical, open source music player.
9. RHYTHMBOX is a complex, multi-threaded, event-driven system, written using a library providing object-oriented primitives in C. Event-driven systems use event queues; each event performs some computation and possibly adds more events to some queues.

The Daikon project monitors instrumented applications to discover likely program invariants. It collects extensive trace information at run time and mines traces offline to accept or reject any of a wide variety of hypothesized candidate predicates.

Amy J. Ko and Brad A. Myers [1] says that in this section, software developers want to understand the reason for a program's behaviour, they must translate their questions about the behaviour into a series of questions about code, speculating about the causes in the process.

The Whyline is a new kind of debugging tool that avoids such speculation by instead enabling developers to select a question about program output from a set of why did and why didn't questions derived from the program's code and execution.

The tool then finds one or more possible explanations for the output in question, using a combination of static and dynamic slicing, precise call graphs, and new algorithms for determining potential sources of values and explanations for why a line of code was not reached. Evaluations of the tool on one task showed that novice programmers with the Why line were twice as fast as expert programmers without it. The tool has the potential to simplify debugging in many software development contexts. In this paper, we present a new kind of program understanding and debugging tool called a Whyline.

This work follows earlier prototypes. The Alice Whyline, supported a similar interaction technique, but for an extremely simple language with little need for procedures and a rigid definition of output.

1. These successes inspired us to extend these ideas to an implementation for Java, which removes many of the limitations of our earlier work.
2. In this way, present empirical evaluations of the technique, one of which found that novice programmers with the Whyline were nearly twice as fast as experts without it.

In a user study of this task, which we report on at the end of this paper, people using the Whyline took half the time that it took for participants to debug the problem with traditional techniques.

This was because participants did not have to guess a search term or speculate about the relevance of various matches of their search term, nor did they have to set any breakpoints. One notable approach is Cleve and Zeller's Delta Debugging, which, given a specification of success and failure, and successful and failing program inputs, can empirically deduce a small chain of failure-inducing events.

III. PROPOSED METHODOLOGY

In the Existing system the bugs are not properly maintained and they are simply relied on shared lists and email to monitor the bugs. In this type of system, it becomes difficult to track a bug. If a bug is overlooked then it may cause tremendous errors in the next phase. This also will improve the cost of project and whatever necessary effort spent on the bug maintenance may not be worthy. And there is no efficient search technique. One has to search the whole database for the details of particular bug which might have occurred sometime earlier. It is both time consuming and error prone.

A. Bug Tracking and Modification System

Bug Tracking and Modification System shows the data flow diagram developer between testers as shown in Figure 1.

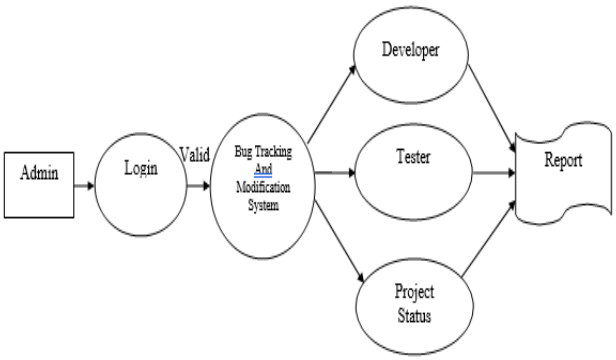

Fig. 1 Bug Tracking and Modification System

B. Experimental Results

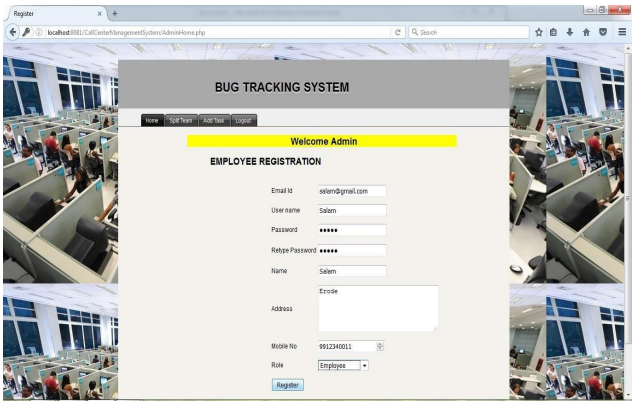

Fig. 2 Login

This is a Bug Tracking system. The user enter the correct username and password goes to the Next step, otherwise username and password error on this page.

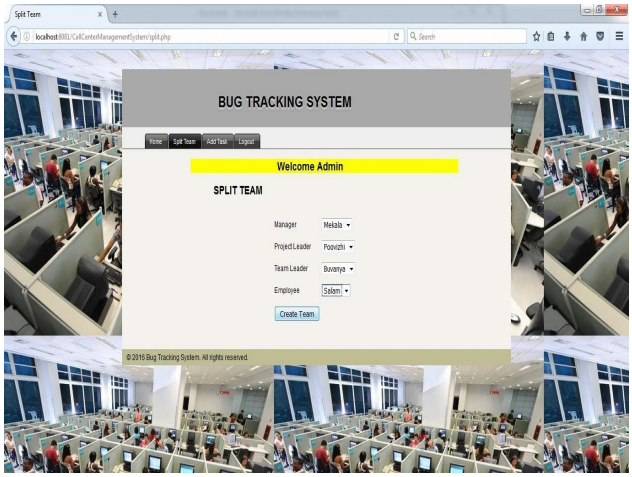

Fig. 3 Admin

After, Login your system Admin can control the overall your project. For example, add developer, add admin, add tester, etc.,

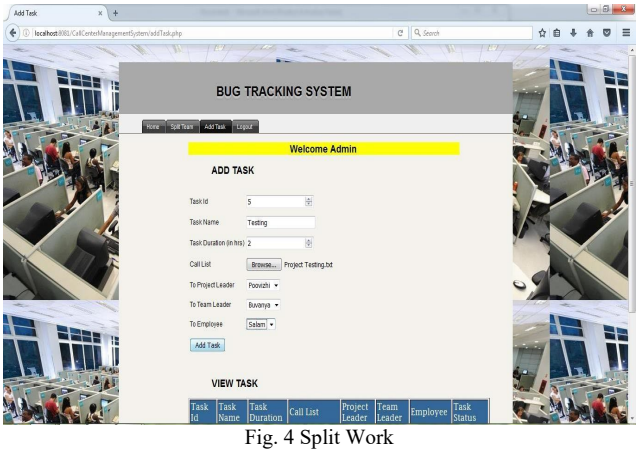

Fig. 4 Split Work

Admin can add a task, add a user. Admin can easy track the status on a within a second.

C. File Design

This system contains the menus for various kinds of operations. Menus and Files are created for displaying the information about Bug Tracking and Modification System. This system also contains the command buttons as part of the user interface. Menu driven programming is very easy to access the programs. In such a way the system is developed.

This system contains the following menus:

1. Create projects.
2. Configure projects.
3. Assign task.
4. View projects.
5. About.
6. Exit

D. Input Design

Input design is a process of converting a user-oriented description of the input to the computer-based system. This design is important to avoid errors in the input process and show the correct direction to the management for getting the correct information from the computerized system. Input design must be in such a way that it must control the amount of input, avoid delay, etc. It must be simple. The input design must ensure user-friendly screen, with simplicity, providing ease of viewing & entering the data. Every input data is validating. If the data is not valid, proper error message are displayed. The main objective of designing input focus on

1. Controlling the amount of input required.
2. Avoiding delayed response.
3. Controlling errors.
4. Keeping process simple.
5. Avoiding error.

E. Output Design

Computer output is the most important and direct source of information to the user. Efficient, intelligible output design should improve the systems relationship with the user and help in decision making. General characteristic of the output forms is as follows.

1. Each output is given a specific name or title.
2. State whether each output field is to include significant zeros, spaces between fields and alphabetic or any other data.
3. Provide a sample of the output including areas where printing may appear and the location of each field.

The output information is also displayed on the screen. The layout sheet for displayed output is similar to the layout chart for designing input. The major reports that are produced using the Systematic Granite Exports Transaction System are,

1. Bug status report.
2. User reports.
3. Project details report.

F. Database Design

Database design is the process of producing a detailed data model of a database. This logical data model contains all the needed logical and physical design choices and physical storage parameters needed to generate a design in a Data Definition Language, which can then be used to create a database. A fully attributed data model contains detailed attributes for each entity. The term database design can be used to describe many different parts of the design of an overall database system. Principally, and most correctly, it can be thought of as the logical design of the base data structures used to store the data. In the relational model these are the tables and views. In an object database the entities and relationships map directly to object classes and named relationships Usually, the designer must,

1. Determine the relationships between the different data elements.
2. Superimpose a logical structure upon the data on the basis of these relationship.

IV. CONCLUSION

All the objectives of this project are satisfied. The intermediate reports can be used for verification, if necessary, in future. The system has been tested with sample data, with original data and the system is found to run well. The concern in which the proposed system will be implemented will find it more efficient. The atmosphere has been made more efficient and interactive. The functioning of the system can be further enhanced in a number of ways, though an attempt has been made for security and high reliability. The newly developed system had simplified the operation for bug tracking. It is portable and flexible for further enhancement.

REFERENCES

[1] A. J. Ko and B. A. Myers, "Debugging reinvented: asking and answering why and why not questions about program behaviour," *In ICSE'08: Proceedings of the International Conference on Software Engineering,* pp. 301-310, 2008.

[2] B. Liblit, M. Naik, A. X. Zheng, A. Aiken and M. I. Jordan, "Scalable statistical bug isolation," *In PLDI'05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation,* pp. 15-26, 2005.

[3] G. Venolia and J. Aranda, "The secret life of bugs: Going past the errors and omissions in software repositories," *In ICSE'09: Proceedings of the 31st International Conference on Software Engineering (to appear),* 2009.

[4] N. Bettenburg, R. Premraj, T. Zimmermann and S. Kim, "Duplicate bug reports considered harmful… really?," *In ICSM'08: Proceedings of the 24th IEEE International Conference on Software Maintenance,* pp. 337-345, 2008.

[5] N. Bettenburg, S. Just, A. Schroter, C. Weiss, R. Premraj and T. Zimmermann, "What makes a good bug report?," *In FSE'08: Proceedings of the 16th International Symposium on Foundations of Software Engineering,* pp. 308-318, November 2008.

[6] P. Fritzson, T. Gyimothy, M. Kamkar and N. Shahmehri, "Generalized algorithmic debugging and testing," *In PLDI'91: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation,* pp. 317-326, 1991.

[7] Ralf Teusner and Christoph Matthies, "Who should fix this bug?," *In ICSE'06: Proceedings of the 28th International Conference on Software engineering,* pp. 361-370, 2006.