Empirical Study of Bug History with JUnit Tests
Progress Report

Siddarth Udayakumar
The University of Texas at Dallas
800 W Campbell Rd
Richardson
TX 75080
sxu140530@utdallas.edu

Keerthi Santhalingam
The University of Texas at Dallas
800 W Campbell Rd
Richardson
TX 75080
kxs142830@utdallas.edu

Ishan Dwivedi
The University of Texas at Dallas
800 W Campbell Rd
Richardson
TX 75080
ixd140630@utdallas.edu

1. INTRODUCTION

The Project we have chosen involved using a few open source java projects and studying about the bug history involved with each project. Each of the projects we have chosen contained more than 1000 lines of code and contain Junit tests of their own. We have chosen ten projects so far that match our requirements and will be performing our study on their bug history to find out which types of bugs could be revealed during testing and which bugs would be revealed after release of the project.

2. RESEARCH

Before we got into the project, we did some research about the common types of bugs that a software may run into. This process made it easier for us to match the bugs that we will be dealing with over the course of the project when we go through a project's bug history.

Some of the bugs that we identified in the bug history of the projects we considered included the following:

1) Bugs related to pointers and memory
2) Arithmetic bugs.
3) Data bugs
4) Synchronization bugs
5) Interfacing bugs.

All these bugs can be further classified under their own topic into categories based on how their sources.

2.1 Pointer and Memory bugs

Some of the projects we went through contained bugs from caused due to pointer issues. The issues raised in the bug trackers was either due to memory leaks, which, were more frequent. Other bugs were due to resources that were already freed being accessed to free them again. Other issues were marked as being caused due to null dereferencing caused because of improper initialization of variables. Pointer aliasing is another issue that can cause bugs if improperly used.

2.2 Arithmetic Bugs

Arithmetic bugs cause major or minor issues depending on the context where the bug occurs. The errors include unexpected results and also termination of the program. They are also one of the more frequently occurring bugs in projects and can easily solved and resolved by basic unit testing as and when required. Some of the reasons for arithmetic bugs include the off by one error where the loop variable has been initialized with one or a zero wrongly. Other issues include enumerated data types and wrong operator precedence. Another rare but common error is the division by zero error.

2.3 Data Bugs

While going through the projects, we noticed that most of the bugs were data related bugs. These bugs may be caused due to uninitialized variables. Other causes include if values are outside the domain and remain uninitialized. One of the more frequent source of data bugs is due to buffer overflow or underflow where the data is written either beyond the acceptable limit of storage. Buffer overflows can also create security bugs.

2.4 Synchronization Bugs

Synchronization bugs occur in cases where there multiple threads in the program that try to access a common resources. Some of these bugs are due to cases of deadlock where multiple threads in the program mutually lock each other. Another reason for a synchronization bug is when two threads lock each other over a common resource because the execution of the threads depends on the order in which they execute.

2.5 Interfacing Bugs

Interfacing bugs occurred due to incorrect usage of APIs, wrong implementation of certain protocols and in some cases,

While going through the bug trackers for the projects we have chosen, we noticed that the most frequent bugs were

data related issues, pointer issues and interfacing issues. Taking into context the Junit tests, it is easier to use junit tests in cases of simple pointer, data and arithmetic issues.

3. IMPLEMENTATION:

For the implementation purposes, we had to choose between a large set of java projects presented to us in the project description and based on the data available to us, we ended up choosing a set of projects that are a little different than the ones listed in the project proposal that was initially submitted. Out of the ten different java projects we managed to study the details of five different java projects and their respective bugs and Junit tests. The projects are listed below

1) Eclipse JDT/Core
2) Apache Lucene
3) Apache Tomcat
4) Evernote SDK
5) Pushy Master

We took each of these projects and downloaded their respective sources from their GitHub repositories. Next, we ran the data source code through a Lexical analyzer we created to look for any traces of Junit tests or any information pertaining to a Junit test added in the code.

We also obtained the bug reports for each of the respective projects in the form of an XML and parsed the XML report to for the title and description of the bug making it easier for us to identify the type of bugs or the issue description associated with the bug.

We obtained the lexicon for the project by creating a parser that uses the Apache Lucene library and Apache Commons Library. Apache Lucene is a high performance full featured text search engine library written entirely in Java. It is suitable for any process that requires full text search functions. Apache commons consists of reusable java components which is used in parallel with the Lucene to allow for parsing the Java source code from the projects. The lexical parser is run on the source code we obtained from the repositories to look for any Junit test cases (We specify the parser to look for occurrences of the particular word or in this case, a string such as “Junit” or “test”). This analyzer passes through the files to deem if the project which we use has valuable data corresponding to bugs and test cases.

Figure 1 shows one of the results we obtained for Eclipse JDT/Core using the Lexical Parser.

Type = 160834
String = 120084
Java = 101539
Null = 74579
Node = 62118
Test = 60097
Name = 59507
Binding = 57425
Method = 56168
Source = 54904

Figure 1 : The output from the Lexical Parser

As per the results, we could immediately identify if the project contains sufficient information for us to use for our study or not. If we identified a project with less information, we move on to the next project.

Once we check if the project has valuable information, the bug report for that project was obtained in an XML format. We use an XML format since the bug reports for the projects run for quite a few pages and we parse it using a simple DOM parser. The DOM Parser can parse for both the title of the bug and its description. We get information about the bug and its related resolution or test cases in the description. The description of the bug contains information about what the bug was, why it was raised in the tracker. In case of closed bug issues, we could also parse the resolution tags in the XML to see if any tests were used in solving the issues. This also provides us with extra information about how the bug was resolved and in cases what the type of bug was. Figure 2 illustrates the data we receive from the parser.

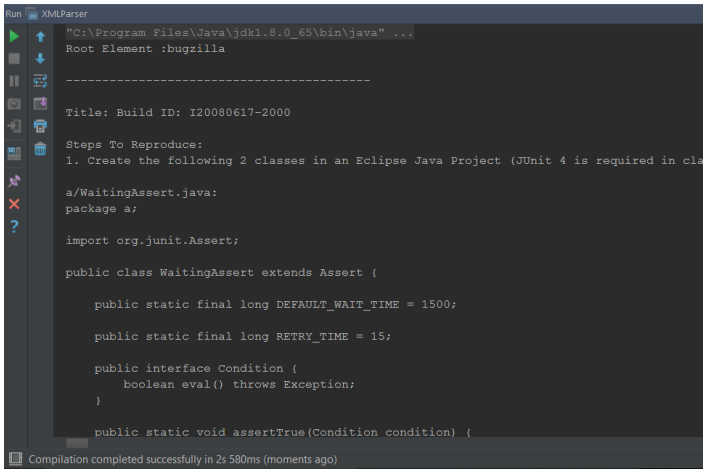

Figure 2 : The output obtained from the XML parser detailing the description for one of the bugs. The description also contains details about the tests used in the resolution.

From Figure 2, we could see that the information pertaining to a particular bug ID is displayed in the results. Using this information, it is much easier for us to identify all the bugs

in the tracker which would provide information about the Junit tests used and if it exists, the reports received from the users about the issue. Using this information, it was possible for us to get more information for this study.

4. LIMITATIONS AND ISSUES

Over the course of this project, the major limitation we encountered was the sheer amount of projects we had to go through to find the correct data and information we needed pertaining to the bug and its respective unit tests (if tests exist). Once we identified proper java projects we would be using over the course of this study, we had a tough time identifying the Junit tests and the type of bugs. The code we have written for parsing through the project source code and the XML parser for the bug reports could only identify the essential sources which could provide us the required information. This streamlines the process to an extent but does not reduce the manual time it took to identify the bugs. This is partially because of the information given in the bug reports. The reports in Github and Sourceforge do not contain all the required information for us to easily identify the type of bug or if any Junit test cases that have been used in testing the program.

We had to manually check the issues in Github to get the information we required. Eclipse, Tomcat used a better defined bug tracking system by employing Bugzilla (Apache Lucene relied on Jira). This allowed us to get the list of bugs easily through the search channels the respective trackers provided, get a list of bugs and consequently, obtain the same in an XML format. But again, not all the reports provided in the trackers contained a proper description about the type of bug and the unit tests. Manual work had to be employed again to correctly identify the bug and if any Junit tests have been employed in its resolution. In spite of our best efforts to find an alternative means to streamline this process, we could not find any means and hence chose to continue with the work manually. This manual work resulted in better accuracy in identifying the bugs and the unit test cases but at heavy cost of the time taken to do so.

5. RESOURCES USED

We used the following resources so far for the project

a) Apache Lucene

b) Apache Commons

c) Bugzilla for certain software projects

d) XML parsers

e) JUnit for custom test cases.

f) Github to store the Apache Analyzer and the DOMParser.

Any other resources we may use in the future will be detailed in the future reports.

The Software code for the parser and lexical analyzer can be found in the following Github repository which will be updated more frequently over the course of this project.

https://github.com/siddarthudayakumar/Software-Testing.git

6. FUTURE WORK

Going further in the project, we hope to complete the study on the rest of the projects we have chosen. We hope to implement an easier way to look for bugs and if available, their respective unit test cases. The remaining projects in the list we have chosen may also change depending on the amount of data we would be able to acquire from them. For now, the projects we have considered that are in the pipeline for investigation are listed here:

1) Bukkit
2) Calculon
3) Nodebox
4) Docker
5) Pushy

As stated in our limitations, we had trouble in getting a proper project with bugs which could be related to a proper Junit test case. This cost us a lot of time in proceeding with the project. We hope to find a more efficient way to identify the bugs and its respective tests. Another one of our plans as stated in the initial project proposal was to include NLTK (Natural Language ToolKit) and use Natural Language Processing to help identify bugs in the projects by using a parser in tandem with NLTK. Implementing a parser with NLTK should not cost us a lot of time but we will have to decide if it is going to be useful in furthering our research or if it is an unfeasible addition to the project. We will be recording the information we obtain from the research and presenting them in the final report.

7. ACKNOWLEDGEMENTS

Thanks to ACM SIGCHI for allowing us to modify the paper templates they had developed.