Empirical Study of Bug History with JUnit Tests

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. ABSTRACT

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 that matched our requirements and performed 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. INTRODUCTION

Every software project when in development, can always run into problems. After development ends, newer problems may arise. The Project team keeps track of these issues using Bug Trackers. The bug here, refers to a software bug which can be an error or a fault in a software program that may cause it to produce a wrong or unexpected output. Most of these bugs arise because of coding errors by software developers or due to inconsistencies in the software packages or its' dependencies. These bugs are recorded in a document called a bug report which consists of a unique ID for the bug, a short description of the bug, a person who reported it, a person is also assigned to resolve the bug and a status to show if it has been fixed or resolved. The Unique ID makes it easier for developers to identify bugs. The short description will contain details about how the bug arose or other details on how to reproduce it. Bug trackers consist of a collection of bug reports, each pertaining to a particular software project.

Any software will always be tested and the most basic form of testing is called Unit test. In Unit testing, the smallest testable module of a software program is tested individually to see if the module works as it is expected. The most used framework for Unit testing is called JUnit. JUnit is a regression testing framework used to implement unit testing in Java projects thus ensuring quality of the software and ensure that it works as it is intended to. The software bugs in a software project are fixed and they undergo testing. In case of most fixes, simple unit testing is performed to check that the proposed resolution for a software bug allows the software to work as intended.

3. 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 dealt with over the course of this project, when we go through a project's bug history. The bugs identified here also are more applicable when it comes to Unit testing.

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.

Other bugs that we found while going through the report were categorized generally as there was not enough description about the type of the bug. The bugs included are

1) Code bugs
2) Documentation bugs
3) Formatting bugs
4) Compiler bugs (Eclipse)
5) Miscellaneous bugs.

There were also situations where due to improper documentation and use of very proprietary software terms, it was not possible to classify the bugs or find any test cases for the project.

3.1 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 encountered. 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.

3.2 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.

3.3 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.

3.4 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.

3.5 Interfacing Bugs

Interfacing bugs occurred due to incorrect usage of APIs, wrong implementation of certain protocols and in some cases, wrong usage or improper usage of other software plug ins or calls in the project. For example, The Apache Zookeeper project relied on other APIs and plug ins.

While going through the bug trackers for the projects we have chosen, we noticed that the most frequent bugs were data related issues, Coding 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.

We also generally classified the other bugs that we came across into broader categories. The bugs are as follows

3.6 Code Bugs

These bugs are related to any human error in the coding process or any changes made to the code. Examples include any change to be done to the code to adjust software dependency, any code changes to be included to perform any specific function or any code changes to existing testing conditions or code. This is a very wide category and a few bugs that showed code changes were included under this category if the report was very vague in description.

3.7 Documentation Bugs

These bugs are not too common but they did exist in all the projects we considered for this project. These bugs usually occurred as typographical errors in the software documentation, mistakes in Java Docs, errors in comments that were already included in the code. These bugs also include changes in error messages for the software.

3.8 Formatting Bugs

These bugs referred to any issues that described wrong formatting of code, any change in the User Interface for the software or any change in how the software displayed its workings. There were very few formatting bugs in the projects we considered for this study.

3.9 Compiler Bugs

These bugs were found especially in Eclipse JDT Core project where there is an actual compiler that allows for compilation of Java Source files. These bugs described issues related to improper working of compilers or any issues that arose from the compiler working wrongly or producing error messages.

3.10 Miscellaneous Bugs

There were few bugs which could not be classified into a proper category. They were a combination of two bugs, for example, a combination of a data bug and a compiler bug. There were also other bugs in the projects which referred to proprietary software in the project they belonged to. Without further details about the software, we could not classify the bugs and hence these bugs are marked as Nil in the bug datasheet.

4. 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. We studied eleven different java projects, their respective bugs and JUnit tests. The projects are listed below

1) Eclipse JDT/Core
2) Apache Cassandra
3) Apache Phoenix
4) Apache Commons IO
5) Apache Tomcat 7
6) Apache Zookeeper
7) Apache HBASE
8) Joda Time

9) MARC4j
10) Android CTS

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.

4.1 Lexical Analyzer

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.

4.2 Bug Report XML Parser

Once we checked if a project has valuable information, the bug report for that project was obtained in an XML format. We used 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.

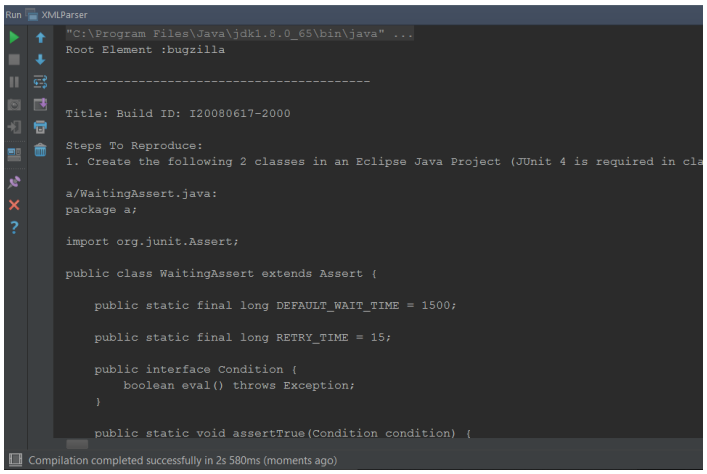


Figure 2: The output obtained from the XML parser detailing the description for one of the bugs. The description also contains details about the tests used in the resolution.

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.

5. OBSERVATIONS AND RESULTS

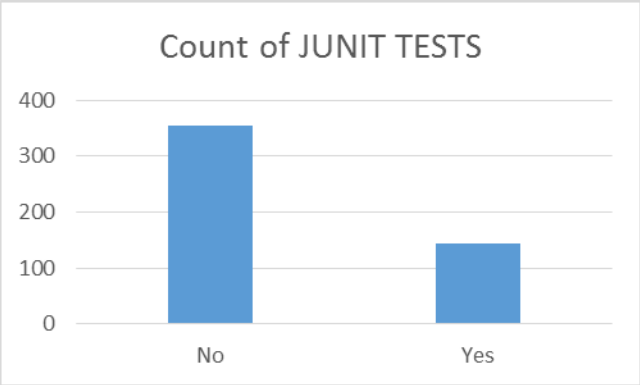
After running each of our code through the repositories and the Bug Reports, we got down to analyzing each of the bugs in each of the software projects that we chose. Please note that we have chosen bugs that have been historically reported and which have been considered resolved or fixed. This is because we can obtain details on when it was fixed and if it has any test cases attached to it. More consolidated data has been added in the spreadsheet attached along with the report. We have listed our observations and results for each project below

5.1 Eclipse JDT Core

Eclipse JDT Core is the Java Infrastructure of Eclipse. It includes an incremental Java compiler, a Java model that provides the API for navigating the Java element tree, a

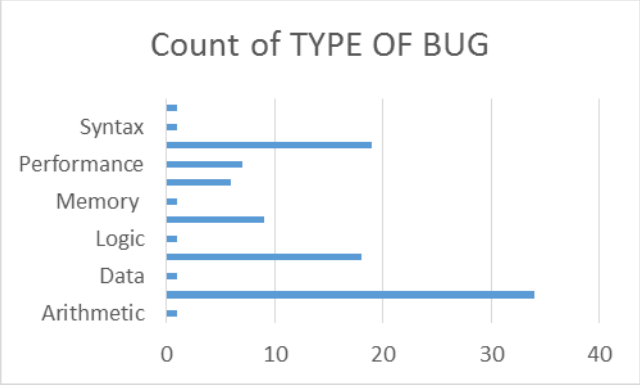
Java document model, and code assist and code support. This is essentially an open source Java project that conformed to our project requirements. We obtained the following details from the bug reports we collected.

We can see that the number of bug reports with JUnit tests were very less compared to the number of reports without JUnit tests. This was partially due to improper documentation of the bug reports and also due to some of the bugs reported having no need of a unit test at all.



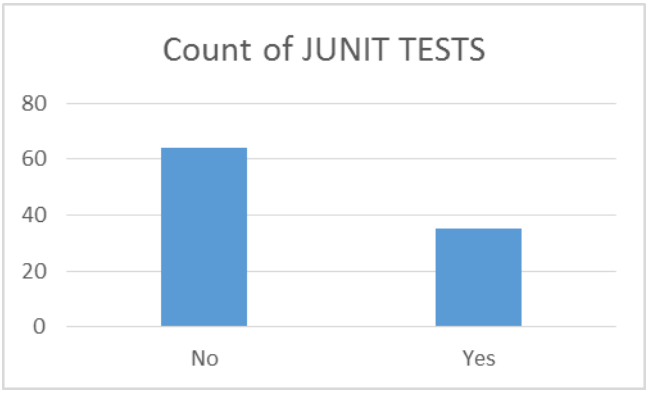
5.2 Apache Cassandra

Apache Cassandra is a database which provides high scalability and high availability without compromising on performance. It is an open source software project that was coded in the Java programming language. We observed that there was a lot of data bugs followed by syntactical bugs.



In the JUnit tests again, we could observe that the number of reports with JUnit tests was lower. Again, this was due to improper documentation of the bug reports.

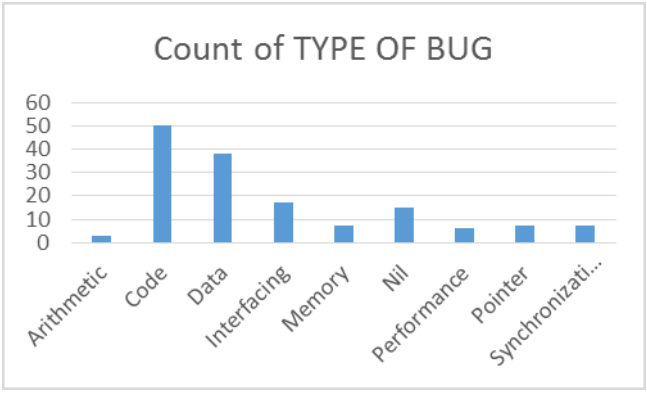
Another observation we found was that there were more number of internal tests that were performed using JUnit compared to the tests being performed after the software was released. In the below graph, you could see the number of JUnit tests that were used from the data set we collected.



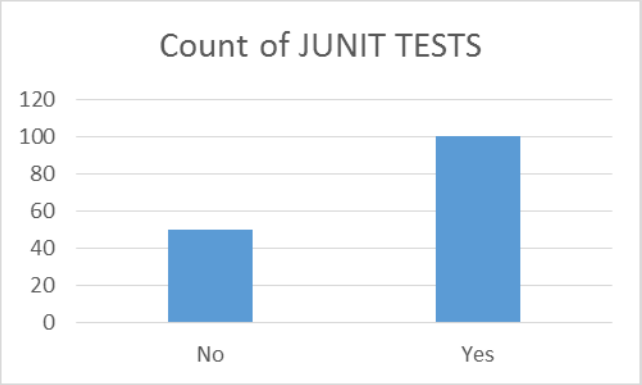
5.3 Apache Phoenix

Apache Phoenix enables Object Linking Protocols and Operational Analytics in Hadoop for low latency applications. This software is fully integrated with other Hadoop products such as Spark, Hive, Pig, etc.

We found that there were a lot of code related issues present in this project closely followed by data related issues.

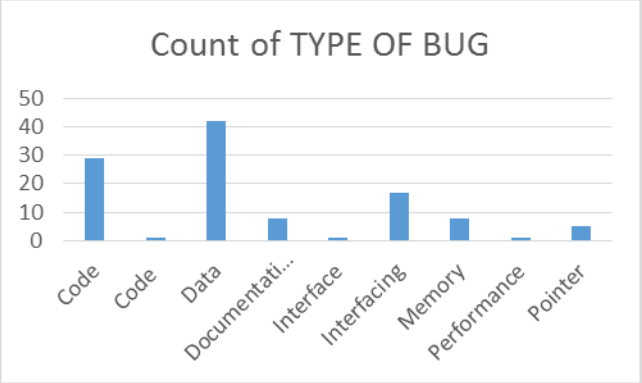


In the JUnit test cases side, we found that there were a lot of JUnit tests. This is because each of the bug report found in Phoenix (Phoenix relied on JIRA for bug tracking), had a proper patch file attached to the bug report. This patch file is supposed to consist of a JUnit test to be taken in for code review and thus be used for resolution of the bug. Another observation was that there was more number of Internal tests that used JUnit testing or unit testing in general to resolve the bugs. The persons who reported and were assigned to fix the bug in the bug reports were from the team of developers working on the project. Very few issues were fixed after the product was released and it was resolved by using softwar patches.

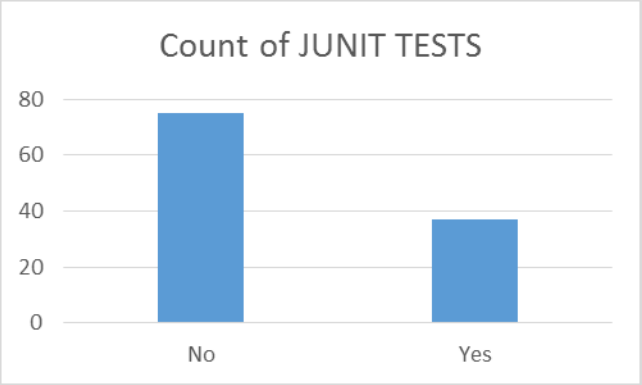


5.4 Apache Commons IO

Apache Commons is an Apache project focused on all aspects of reusable Java components. This project was coded in the Java language. We found that again, there was a higher number of data bugs compared to the other bugs and was closely followed by code bugs which dealt with human made errors.



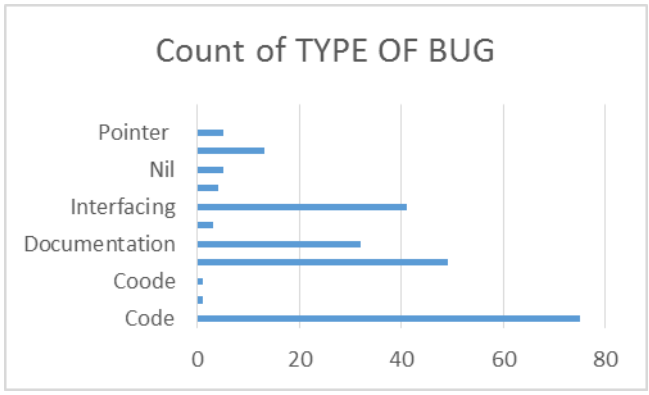
In commons IO, again, there were a fewer number of JUnit tests present and this was also followed by fewer number of reports using JUnit to fix bugs that were patched after the software release.



5.5 Apache Tomcat 7

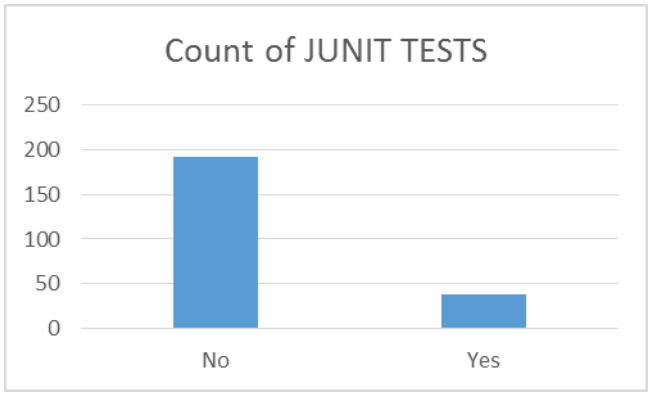
The Apache Tomcat software is an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. The Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket specifications are developed under the Java Community Process. This project was also coded in the Java language.

In this project, we observed that coding errors were far more than any other errors. Since Tomcat also had a lot of softwar dependencies, there was also a fair amount of interfacing bugs. There were a few bugs which were unidentifiable and hence they are marked as Nil on the charts



Again, the number of JUnit tests for Tomcat was very low. This was due to JUnit tests being referred in the bug reports but not being used at all in the reports. This was again due to improper documentation of bug reports.

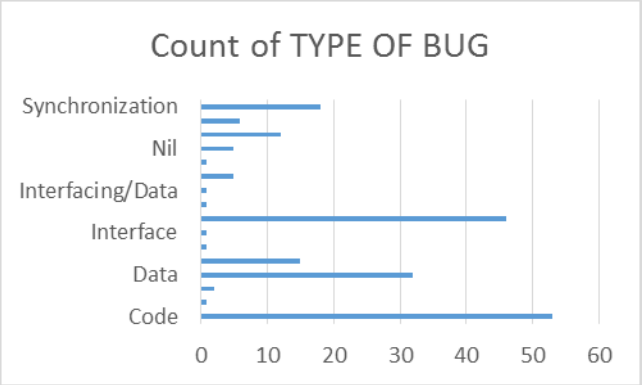
With respect to bugs being fixed before or after release of the software, the developer team had fixed more bugs internally compared to patching up a bug after release of the software.



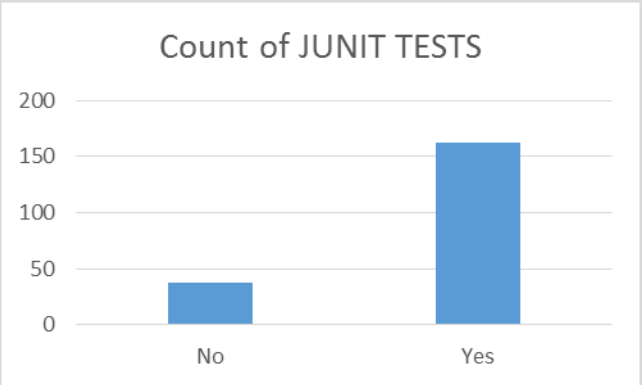
5.6 Apache Zookeeper

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. This project was also coded in the Java language.

In this project, we observed that there were again, more code bugs followed bu interfacing bugs. This was because of Zookeeper having a lot of software dependencies.



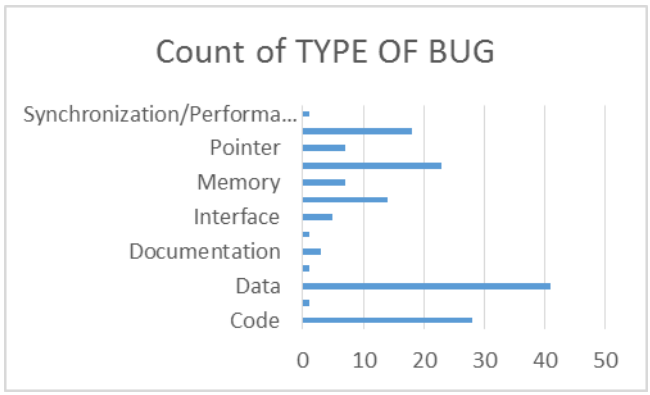
Zookeeper, unusually had a higher number of JUnit tests being used in internal tests and resolution compared to JUnit tests being used to resolve a patch for an after release fix. This was because the internal developer team has an automated bot called Hadoop QA that ran Unit tests for the modules in the bug reports that were to be resolved.

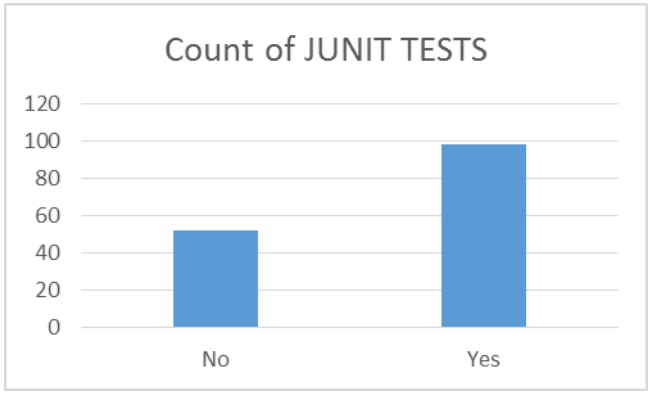


5.7 Apache HBASE

Apache HBase is the Hadoop database, a distributed, scalable, big data store. This project was coded entirely in the Java programming language.

In this project we obtained more data bugs closely followed by code bugs. Also since the project was also worked on and resolved by most of the developers working on the project, this project had more number of JUnit tests that were used for resolving internal tests. Very few JUnit tests were used for resolving the after release software fixes.





5.8 Joda Time

Joda-Org/ Joda-Time

Total JUnit Tests: 137 JUnit Tests / (Out of 469)

The problem solved:

The trouble with the older @Before and @After JUnit approach is that it is very easy to forget to do some tear-down in the @After. If there's a commonly used pattern in your tests requiring @Before and @After it's also harder to reuse because you have two separate methods to change.

A Rule is a great way of packaging the code for that commonly used procedure and easily reusing it elsewhere.

With JodaTime you can easily set a fixed date time for a unit test

Example:

DateTimeUtils.setCurrentMillisFixed(100000);

Usage:

DateTime aTimeToUseInThisTest = new DateTime().plusSeconds(10);

Currently, there are 2 problems with it:

First, you have to remember to call DateTimeUtils.setCurrentMillisSystem() in your tear down method to avoid potentially polluting other test cases, even in other classes. Second, it's not obvious in that line of code where you create a new DateTime that it will be using a fixed time. The reader has to have noticed the set-up of the fixed time, and remembered it.

We can fix both those problems with a JUnit rule.

Now, instead of set-up and tear-down in your @Before and @After, you simply declare your Rule:

@Rule

Public FixedTimeRule fixedTime = new FixedTimeRule(1000000);

It's very easy to read that line and immediately understand that we are making an offset from some fixed time, even if you haven't already spotted the Rule. The trick is make sure that your Junit Rule implements an appropriate interface so that you can use it as a stand-in for the original class. In this case, we implement the JodaTime ReadableDateTime by delegating to an underlying instance of DateTime, allowing you to change "new DateTime().plusSeconds(10)" into "fixedTime.plusSeconds(10)". You can quickly implement all the required delegation methods by asking your IDE…

That may be simple, but a good test is one that clearly expresses intent, and clean code is code that you can intuitively understand without having to hunt down distant lines of code that carry significant pre-requisite knowledge.

5.9 Marc4j

27 Test cases were only obtained. The documentation for this was very unclear.

The goal of MARC4J is to provide an easy to use Application Programming Interface (API) for working with MARC and MARCXML in Java. MARC stands for MAchine Readable Cataloging and is a widely used exchange format for bibliographic data. MARCXML provides a loss-less conversion between MARC (MARC21 but also other formats like UNIMARC) and XML.

The only good test we got for this was that there was integration for JUnit 4 to be included within the test cases.

5.10 Android CTS

The CTS is an automated testing harness that includes two major software components:

The CTS test harness runs on your desktop machine and manages test execution.

Individual test cases are executed on attached mobile devices or on an emulator. The test cases are written in Java as JUnit tests and packaged as Android .apk files to run on the actual device target.

10000 + JUnit Test Cases were found for this huge project.

6. 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, Bugzilla and JIRA 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.

Professor asked us to use JIRA to automate obtaining the Unit Tests but unfortunately, that was not possible without using JIRA as a platform for working on our own software projects so manual work was the only way to get details on the bugs and the bug reports.

7. RESOURCES USED

We used the following resources so far for the project

a) Apache Lucene

b) Apache Commons

c) Bugzilla for Eclipse

d) XML parsers

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

f) JIRA for the various projects to get bug reports.

g) Microsoft Excel for generating the statistical graphs for the study.

h) Google's Android Issue Tracker for the CTS study.

The Software code for the parser and lexical analyzer can be found in the following Github repository listed below.

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

8. WORK SPLIT UP

TEAM MEMBER	WORKED ON
Ishan	• Joda Time • Marc4j • Android CTS
Keerthi	• Cassandra • Tomcat • HBASE
Siddarth	• Eclipse JDT Core • Commons IO • ZooKeeper • Lexical Analyzer, XML Parser • Phoenix

9. FUTURE WORK

We hope to implement an easier way to look for bugs and if available, their respective unit test cases. 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 had to decide if it was going to be useful in furthering our research but, unfortunately, the reports we went through showed us it is not easy to create a properly trained classifier for classification of the various bugs. This is because of two reasons 1) The number of bugs is far too wide to be categorized correctly as we have done in the project and 2) There is quite a lot of incorrect or improper documentation of bug reports to be able to correctly identify the bug for a particular report. We have observed how difficult our task was over this study and we will look into a more streamlined and standardized method of filing bugs.

10. CONCLUSION

Thus we see how each of the projects we considered for this study differed in the type of bugs reported in each of their trackers and the effect of JUnit tests in resolving the bugs. More detailed statistics are provided in the Spreadsheet that was attached to this report. We learned a lot over the course of this project and would like to thank the professor for guiding us through this project.

11. REFERENCES

[1] Zhong, Hao, and Zhendong Su. "An empirical study on real bug fixes."*Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015.

[2] Mockus, Audris, Roy T. Fielding, and James D. Herbsleb. "Two case studies of open source software development: Apache and Mozilla." *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11.3 (2002): 309-346.

[3] Apache Common and Lucene libraries.

[4] https://eclipse.org/jdt/core/

[5] https://projects.apache.org/project.html?cassandra

[6] https://projects.apache.org/project.html?commons-io

[7] https://projects.apache.org/project.html?hbase

[8] https://projects.apache.org/project.html?phoenix

[9] https://projects.apache.org/project.html?tomcat

[10] https://projects.apache.org/project.html?zookeeper

[11] https://github.com/JodaOrg/joda-time

[12] https://github.com/marc4j/marc4j

[13] https://source.android.com/compatibility/cts/

[14]https://nemo.sonarqube.org/component_measures/metric/reliability_rating/list?id=junit%3Ajunit

[15] https://www.wikipedia.org/

12. ACKNOWLEDGEMENTS

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