EVALUATING TESTING EFFORT AND ITS CORRELATION TO CYCLOMATIC
COMPLEXITY AND CODE COVERAGE

A Thesis
Presented to
The Graduate Faculty of The University of Akron

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

Pallavi Saxena.
August, 2015
EVALUATING TESTING EFFORT AND ITS CORRELATION TO CYCLOMATIC COMPLEXITY AND CODE COVERAGE

Pallavi Saxena

Thesis

Approved:  

Advisor  
Dr. Michael L. Collard

Accepted:  
Dean of the college  
Dr. Chand Midha

Faculty Reader  
Dr. En Cheng

Interim Dean of the Graduate School  
Dr. Chand Midha

Faculty Reader  
Dr. Chien-Chung Chan

Date

Interim Department Chair  
Dr. David N. Steer
Software testing is an important part of the development process, and therefore a measure of testing effort is needed to monitor the process. In addition, software metrics have been developed to measure the quality of a software system. These metrics tell us important facts about the software system, but to truly gauge the quality of a software system we specifically need to study a large number of projects, and compare some of them over multiple releases.

This thesis studies the testing effort on a large number of projects, taken from large, public repositories from the source code management site GitHub. For each project the testing effort, based on various criteria, is analyzed to try to understand the amount of testing effort performed. The testing effort is compared to complexity metric and code coverage. In addition, select systems were analyzed across their history to see if there were any changes in the testing effort over the entire time of development. The results show that for many projects, the testing effort is not where it needs to be.
ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Collard for his constant support and motivation during the research work. He not only showed me the right direction to move on but also bought me back whenever I wandered in wrong direction. Thank you for all his effort and valuable advises. I would like to thank all the faculty members of Computer Science department at The University of Akron specially Dr. Chan, Dr. Cheng, Dr. Liszka and Dr. Duan. The courses learned from them gave me insight to work on this research.

I would also like to thank my family for their moral support and encouragement during the research work.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>2</td>
</tr>
<tr>
<td>II. RELATED WORK</td>
<td>6</td>
</tr>
<tr>
<td>III. CORELATION OF TEST AND PRODUCTION CODE</td>
<td>8</td>
</tr>
<tr>
<td>3.1 Motivation</td>
<td>8</td>
</tr>
<tr>
<td>3.2 Repository Collection</td>
<td>9</td>
</tr>
<tr>
<td>3.3 Data Collection</td>
<td>12</td>
</tr>
<tr>
<td>3.4 Analysis</td>
<td>14</td>
</tr>
<tr>
<td>3.4.1 Test LOC vs Total LOC</td>
<td>14</td>
</tr>
<tr>
<td>3.4.2 Test LOC vs Production LOC</td>
<td>15</td>
</tr>
<tr>
<td>3.4.3 Percentage Of Test LOC and Date of First Commit</td>
<td>20</td>
</tr>
<tr>
<td>3.4.4 Percentage Number of Test Files and Date of First Commit</td>
<td>21</td>
</tr>
<tr>
<td>3.5. Results and Discussions</td>
<td>22</td>
</tr>
<tr>
<td>IV. PART II – DETAILED ANALYSIS OF THREE REPOSITORIES</td>
<td>24</td>
</tr>
<tr>
<td>4.1 Objective</td>
<td>24</td>
</tr>
<tr>
<td>4.2 Motivation</td>
<td>25</td>
</tr>
</tbody>
</table>
APPENDIX C. CODE TO CALCULATE TEST LOC AND PRODUCTION LOC PER REPOSITORY……………………………………………………………………..52

APPENDIX D. GIT COMMAND TO FORK 100 REPOSITORIES………………57
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hibernate detailed results</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>Hadoop detailed analysis</td>
<td>34</td>
</tr>
<tr>
<td>3</td>
<td>Spring detailed analysis</td>
<td>36</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Snapshot of project to fork repositories</td>
<td>11</td>
</tr>
<tr>
<td>2: Snapshot of batch file used to clone repositories</td>
<td>12</td>
</tr>
<tr>
<td>3: Snapshot of analysis</td>
<td>14</td>
</tr>
<tr>
<td>4: Test loc vs total loc</td>
<td>15</td>
</tr>
<tr>
<td>5: Production loc vs test loc</td>
<td>16</td>
</tr>
<tr>
<td>6: Total lines of code vs test lines of code</td>
<td>16</td>
</tr>
<tr>
<td>7: Test lines percentage vs total lines of code</td>
<td>17</td>
</tr>
<tr>
<td>8: TEST LOC - PROD LOC - TOTAL LOC over the entire system</td>
<td>18</td>
</tr>
<tr>
<td>9: Total number of files vs test number of files</td>
<td>19</td>
</tr>
<tr>
<td>10: Number of test files vs total number of files ratio</td>
<td>19</td>
</tr>
<tr>
<td>11: Percentage of test lines of code and date of first commit</td>
<td>21</td>
</tr>
<tr>
<td>12: Percentage number of test file and date of first commit</td>
<td>22</td>
</tr>
<tr>
<td>13: Hibernate test loc vs total loc</td>
<td>32</td>
</tr>
<tr>
<td>14: Hibernate detailed analysis</td>
<td>33</td>
</tr>
<tr>
<td>15: Hadoop test loc vs total loc</td>
<td>34</td>
</tr>
<tr>
<td>16: Hadoop detailed analysis</td>
<td>35</td>
</tr>
<tr>
<td>17: Spring test loc vs total loc</td>
<td>37</td>
</tr>
<tr>
<td>18: Spring detailed analysis</td>
<td>38</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

Source code is considered to be the most important artifact of a software system. However, a successful software system has many other related artifacts such as documentation, requirements, tests etc. A software system, including the source code and these other artifacts, should evolve and change over a period of time; otherwise the system becomes less useful. So a proper mechanism needs to be established to support change, enhancements, and bug fixes.

It has been well established that unit tests are important for success of any software system. In addition to showing that the system works correctly, unit tests are also important to understand the implication of any changes, the effort required to do any change, and also helps new developers to understand the software requirements of the system.

Software metrics have been developed to measure the quality of a software system. These metrics do tell us important facts about the software system, but to truly gauge the quality of a software system we need to study a large number of projects and compare them over multiple releases.

Code coverage is an important mechanism through which we can test if a system has sufficient testing. Low code coverage signifies lack of testing effort, while high code
coverage signifies acceptable levels of testing effort. This is an important metric for a healthy software system.

This thesis studies the testing effort on a large number of projects, taken from large, public repositories at the source-code management site GitHub. For each project the testing effort, based on various criteria, is analyzed to try to understand the amount of testing effort performed. The intent is to measure the testing effort to see what is typically done with these systems. In addition, select systems were analyzed across their history to see if there were any changes in the testing effort over time.

1.1 Background

Lehman states that a software system must undergo changes to maintain its usefulness; otherwise the users will grow dissatisfied [1]. Source code is considered the most important artifact of a software system. However, for a successful software system there are many components and all the components play an important role, with other important artifacts being documentation, tests, specifications etc. [2].

Software testing is an important part of software-development lifecycle. Programmers write unit tests during development. These unit tests are beneficial not only for ensuring the correctness of code for the current release, but also for future releases. The unit tests help us find issues early in the process – in the development cycle itself. As the software system continues to evolve the tests are run either periodically or are part of continuous build. These tests sometimes fail, typically due to issues in code and occasionally due to issues in tests itself.

Unit tests provide many practical benefits to the code. First, having good unit tests facilitates refactoring code, where refactoring is a change to the design of code that does
not change the behavior of the code. Over a period of time, due to enhancements or bug fixes, the code might need refactoring applied to keep up with best practices. One of the major issues with refactoring is developers are scared to touch working code as they might inadvertently introduce new bugs. However, for a software system with good unit tests these issues are mitigated to an extent. When unit tests fail after a refactoring is applied, developers immediately know there are issues with the refactoring. The developers then can fix the issue and can continue with the code refactoring. Second, unit tests also act as documentation of the system. By looking at the test code, developers can understand how the code is being utilized by the system. And finally, unit tests also encourage developers to write better and modular code. Unit tests are a whole lot easier to write and apply if the code is modular.

Objective measurement of test quality is one of the key issues in software testing, and has been a major research focus for the last two decades. Many test criteria have been proposed and studied for this purpose [4]. Testing should not just be performed at the end of the development cycle, but should be an integral part from the very beginning of the software development cycle. Therefore, test criteria should be applicable even during initial development.

Code coverage is a measure to describe the extent to which the code is tested by unit tests. A unit test will only cause specific lines of code to be executed. The code coverage of a system is a percentage of the total number of lines of code in the production software that were executed by the set of unit tests. Software systems with higher code coverage (i.e., approaching 100%) are usually more thoroughly tested and are assumed to have a lower probability of containing bugs [18]. Some platforms have
restrictions that do not allow the code to be deployed if the code coverage falls below a certain percentage. Code coverage can also be used for fault localization [3]. This could be beneficial as developers can prioritize and can examine the higher risks code first. There are various free and license-based tools available that measure the code coverage of a software system. Two of the code coverage tools used in this study are Eclemma and Clover.

One measure of the complexity of a program is cyclomatic complexity. It is a software metric developed by Thomas J. McCabe [4]. This metric measures the number of linearly independent paths through the code. The cyclomatic complexity can be measured for a method, class, package, or the entire system. The higher the complexity value, the more independent paths, therefore the complexity of code is higher. The idea behind the metric is to try to reduce the complexity of code during development. As a practical rule, McCabe suggested splitting the code into smaller modules or methods if the complexity exceeded 10.

One of the applications of cyclomatic complexity is in determining the minimum number of test cases required to achieve complete code coverage of a software unit, such as a method or class. The higher the complexity of code, the more test cases are required to cover the code. The number of execution paths through a method is directly related to the understandability, maintainability, and testability of the method. A general rule of thumb states that in order to ensure a high level of test coverage, the number of test cases for a method should be at least equal to the method's cyclomatic complexity. Also, higher complexity code leads to hidden bugs and issues, with developers required to put in extra effort to test and also understand the code.
Moonen et al. have shown that even while refactorings are behavior preserving, they potentially invalidate tests [12]. Elbaum et al. concluded that even minor changes in production code could have serious consequences on test coverage, or the fraction of production code tested by the test suite [13]. To mitigate these issues, it is understood that production code and test code need to evolve together. This leads to the almost paradoxical situation whereby tests are essential for the success of the software (and its evolution), while also being a serious burden during maintenance.

As the complexity of software increases, detecting all software bugs is practically impossible, thus making complete testing infeasible. In the past, several studies have explored different software testing strategies and techniques to address these challenges and propose methods to perform exhaustive testing of software.
CHAPTER II

RELATED WORK

Cyclomatic complexity and its effect on maintenance cost of software system is an active area of research. Menzies et. al, studied the effect of cyclomatic complexity density and software maintenance productivity [7]. Researchers have actively discussed McCabe’s Cyclomatic Complexity (CC) and lines of code (LOC) as a metric. It is believed that there is high-enough correlation between CC and LOC to justify adjusting CC by LOC or even substituting LOC for CC [15].

Testing is an indispensable part of software development efforts. Kochhar et.al, studied more than 20,000 non-trivial software projects and explore the correlation of test cases with various project development characteristics [5].

Testing should be a part of development and not something that should be scheduled towards end of project. Zaidman et.al, studied co-evolution of test and production code for open source projects [14].

Studies have suggested a correlation between the number of defects found in a method and cyclomatic complexity [19]. Some studies find a direct correlation between complexity and defects, as in higher number of defects are found in code with high
cyclomatic complexity. Other studies performed with controlled program size and varying complexities tried to find the relation between complexity and number of defects. The studies so far have been inconclusive; with some researchers have questioned the methodology of such studies [20].

Code coverage and Cyclomatic complexity is an active research area. Kim et.al, found efficient ways to analyze large industrial software systems using the results from defect distribution of previous releases and the release under current development [16].
This part of thesis aims to study the correlation between the test code and production code, and various other project development characteristics. Specifically, we are trying to answer following questions:

- RQ1 What is the relationship of testing effort and total effort, in terms of LOC?
- RQ2 What is the relationship of testing effort and total effort in terms of number of files.
- RQ3 Is there any correlation between testing effort and production code based on when the project was initiated?

3.1 Motivation:

Unit Tests are important for maintaining quality in software systems. An important measure of test code is test lines of code. This metric demonstrates the testing effort. We try to find out over a large number of systems the correlation between testing and production lines of code.

3.2 Repository Collection

In this section we describe the methodology and tools that were used for analysis. This includes how the systems were chosen, and how the analysis was performed.
GitHub is a web based hosting service, which offers all of the distributed revision control and source code management (SCM) functionality of the Git version control system as well as adding its own features. We used Git for the source of our projects because it is becoming the de facto version control system in the open-source development community. As of 2014, GitHub reports having over 3.4 million users [4] making it the largest code host in the world [5]. GitHub has rich API to filter and retrieve repositories, and there are numerous third party libraries that support the API.

We wanted to analyze a large number of systems, so an automated approach was used to download repositories. A third party Java library, Egit, was used to search, filter, fork, clone, download, and analyze the repositories. In this study Java code repositories of size larger than 30 MB were used.

For this investigation, a GitHub account was created and the GitHub GUI client downloaded. Included in the GUI client was the GitHub bash client. The GitHub API was used to list all public repositories based on the selection criteria.

The criteria for selecting projects was that they should be open-source Java projects with a repository size greater than 30 MB. Open-source projects were selected because the code base is publicly available. They also provide a history of the project, including multiple releases for further analysis. Projects in the Java language were selected for multiple reasons. Java is has been a popular language for open source projects for quite some time, so large projects can easily be found in Java in various domains or categories. In addition, Java has a well-defined unit-testing tool, JUnit that allowed for the identification of test code to be accurately automated.
This study analyzed repositories at least 30 MB in size. This was done to weed out trivial projects. This criterion ensured that most of the projects were large enough, and that there were multiple contributors in the project. Also, projects of this size are usually long-term projects with a history of development. This criteria helps to ensure that we get a proper subset for analysis, and increases the probability that selected repositories are indicative.

Over 100 systems were used in the study. At first, the GitHub bash client was used to determine how to select the projects for this study, specifically the search features of GitHub. After understanding the GitHub API, it was decided that a specific tool had to be built. To do this efficiently a GitHub third-party GitHub library was used. Various libraries were investigated including JCabi, GitHub Mylyn Connector, egit-github. It was decided to use the Java library egit-github (https://github.com/eclipse/egit-github), as it supported almost 100% of the GitHub v3 API.

A Java project was created in Eclipse to download all the systems. The library egit-github does not provide much documentation, so the whole source code was downloaded. The code in the Java project used this API to search according to the discussed criteria, fork 100 projects, and record all the repository names in a file. The decision was made to fork the repositories so that they would not change during the analysis. Figure 2 shows an Eclipse snapshot of the project with the GitHub and egit-github packages used, and the code to search for the repositories.
In addition to searching and forking the project, the above code wrote all the repositories names to a text file. The complete list of all forked repositories in the study can be found at https://github.com/scholarpallavi?tab=repositories. Even though the projects were public, user credentials had to be provided so that projects could be forked and cloned, as non-authorized users are not able to transfer large amounts of data.

While the search and fork of the repositories was performed with Java, repositories were locally cloned using a generated batch file with a git clone command for each repository. A snapshot of the batch file is given in Figure 3. On running this batch file, git cloned all the repositories to a local drive. A batch file was used so that the cloning of the project could be duplicated on another machine.
3.3 Data Collection

A total of 100 repositories were selected, forked, and cloned. A list of these can be found in Appendix A. For each of the 100 systems, the following data was extracted for each system:

- Test LOC
- Production LOC
- Number of test files
- Number of production files

In addition, the total sum of these measures was recorded.

The primary issue was to distinguish between test and production files. Since these were Java programs, the assumption was made that JUnit, a very-popular Java unit-testing framework, was used. That allowed us to tell the difference between production and testing files.

The first step is to determine, for all files, if they contain test or production code. The following bash command was used to find all source code files in a project:

```
find . -name '*.java'
```
To find the files that include the JUnit framework, the following was used to create a list of test files:

```
xargs grep --ignore-case --files-with-matches "junit"
```

The `xargs` command applies the following the `grep` command to each file. The `grep` command searches for the name “junit”, with the output being just the names of the files that were matched by the `grep`.

As a last step, the total number of lines of code was found using the `wc` command, which counts the number of lines, which in this case is the number of files.

```
find . -name '*_.java' | xargs grep --ignore-case --files-with-matches "junit" | wc -l
```

Across all the studied systems the total number of files that contained test code was 19,314 compared to 160,664 total source-code files and 179,978 production files.

For each repository, two files were generated. One file has a list of test code files, and the other had all source-code files. A Java program calculated the total lines of code and sum of test lines of code using these input files. The output was a report stating how many lines of code are in each repository and out of those how many of those lines belongs to test code. Figure 4 shows a snapshot from this analysis, with the complete reports in Appendix A.

<table>
<thead>
<tr>
<th>Repository</th>
<th>LOC</th>
<th>Test LOC</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>finnjava</td>
<td>83460.0</td>
<td>10600.0</td>
<td>0.4628970045535978</td>
</tr>
<tr>
<td>annj_seg</td>
<td>9200.0</td>
<td>424.0</td>
<td>0.046056656512173913</td>
</tr>
<tr>
<td>jersey</td>
<td>350028.0</td>
<td>130336.0</td>
<td>0.3723577189318599</td>
</tr>
<tr>
<td>spoon</td>
<td>7396.0</td>
<td>901.0</td>
<td>0.121826068149932</td>
</tr>
<tr>
<td>platform.framework</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ks_base</td>
<td>1387778.0</td>
<td>63281.0</td>
<td>0.031167392290458697</td>
</tr>
<tr>
<td>jna</td>
<td>117116.0</td>
<td>16674.0</td>
<td>0.14237166569696812</td>
</tr>
<tr>
<td>druid-l</td>
<td>271927.0</td>
<td>120063.0</td>
<td>0.441526586179379</td>
</tr>
</tbody>
</table>
Figure 3: Snapshot of analysis comparing total LOC to test LOC for the mass repositories

3.4 Analysis

We now take a look and the collected data, and try to determine the relationship between test LOC and production LOC. This is to try to determine the typical amount of test LOC in a system. The following sections try to find out relationship of testing effort and total effort, in terms of LOC (RQ1).

3.4.1 Test LOC vs Total LOC

The first direct analysis of this data is given in Figure 4. There, the size of the project (total LOC) is compared to the size of test LOC. The graph in the figure is ordered on the x-axis based on increasing test lines of code in a project. In general, the trend is what would be expected, that the larger the project the greater the amount of test code. The upward spikes are projects that have a large amount of code, but little test code. For instance the project XobotOS has 1,682,479 lines of code however it has only 28,095 test lines of code. Similarly the project intellij-community has 3,806,602 total lines of code and only 152,795 test lines of code. These projects are the spikes in the graph in Figure 4.
3.4.2 Test LOC vs Production LOC

We also compared test LOC to production LOC, where production LOC is the total LOC – test LOC, as shown in the graph of Figure 5. The test LOC seems to be following the general trend of production LOC. However, for the larger systems the test LOC actually goes down. Except for these large systems, the test LOC increases as production LOC increases.
In Figure 6 we plot the Total Lines of Code and test lines of code. We see that in general as total lines of code increases the test lines of code is also increasing. However for some larger projects do not have enough test cases yet.
In Figure 7 we plot test percentage of lines vs total lines of code. We see that test LOC is up to 30% of total LOC for most of the projects. Even for larger projects this ratio is maintained.

![Figure 7: Test Lines Percentage vs Total Lines of Code Ratio](image)

We calculated the overall percentage of test code and plotted in Figure 10 over the entire system. We found the test code was 15% of the total code.
18

Figure 8: Test LOC - PROD LOC - Total LOC Over the Entire System

It is said that ideally test LOC should have a 1:1 ratio with PROD LOC. This means in general test LOC is lagging behind PROD LOC.

3.4.3 Test Number of Files vs Production Number of Files

The following sections are for answering RQ2.

In Figure 9 we plot test number of classes and total number of classes. We see that test numbers of classes are smaller than total number of classes. The test classes increase but the number of test classes fall way behind the total number of classes.

In figure 10 we see that test percentage number of classes is mostly up to 30% of total number of classes. This is the same ratio obtained in the analysis of test LOC percentage against total LOC.
3.4.3 Percentage of Test LOC and date of first commit
The following sections try to investigate whether there is any correlation between testing effort and production code based on when the project was initiated (RQ3).

The studied systems have a variety of project starting times, so it was investigated whether there was any correlation between the data of the first commit and the amount of test code. The following bash command was used to retrieve the date of first commit:

```
ls | xargs -I^ bash -c 'echo -n ^; echo -en ";"; cd ^; git rev-list HEAD | tail -n 1 | xargs -I% git log --date=iso % | grep "Date:" | cut -d";" -f2- | cut -c4- | cut -d"+" -f1'
```

The command takes all of the repositories and uses the git rev-list command to list all the commits. The last commit in the list (the first commit made) is used to look up the data from the git log command. The rest of the script selects the proper field out of the data produced.

The following plot in Figure 11 shows the percentage test LOC and the date of first commit. The plot suggests that newer repositories have a higher percentage of test LOC. We see that the plot forms a bell curve suggesting older projects have more contents and so more tests. The newer projects have not built up enough tests content yet and as time progresses these project would add more test cases.
3.4.4 Percentage number of test files and date of first commit

We plotted number of test files of each repository and date of first commit as shown in Figure 12. This was done to see if the number of test files and number of production files have any correlation. We see that plot forms a bell curve. This suggests that older projects have over time build up test content where as newer projects are catching up and over time would add more test content.

Figure 11: Percentage of test lines of code and date of first commit
3.5 Results and Discussions

The analysis that were done were as follows:

- Test LOC and production LOC of 100 systems
- Number of test files and production files of 100 systems
- Test LOC and production LOC per system
- Number of test files and production files per system

Unix commands were written to do the analysis. After required data was gathered using the Unix commands, it was analyzed by writing custom code. We found out that some repositories had no test code. So those repositories were further analyzed and we
found that repositories did not have test code at all and a very few of them had python code. Our test analysis was based on test classes containing the word ‘junit’.

The research questions that we set out to answer are as follows:

**RQ1: What is the relationship of testing effort and total effort, in terms of LOC**
For answering RQ1 we plotted data as shown in Figures 4, 5, 6, 7, and 8. We saw that in general, Test LOC was quite behind the Production LOC in terms of Lines of Code. However, Test LOC mostly followed Production LOC, meaning that as Production LOC increased, Test LOC also increased. Most interestingly, we also saw that Test LOC was in the range 0 - 30% for most projects, regardless of the size of the project.

**RQ2: What is the relationship of testing effort and total effort, in terms of number of Files.**
For answering RQ2 we plotted Figures 9 and 10. We saw that in general Test number of files was quite behind the Production number of files. However, we saw that while Production number of files increased, Test number of files also increased. Test number of files were also in the range of 0 - 30% for most projects, regardless of the size of the project.

**RQ3: Is there any correlation between testing effort and production code based on when the project was initiated?** For answering RQ2 we plotted data as shown in Figures 11 and 12. We saw that newer repositories have a higher percentage of Test LOC. We also saw that the plot forms a bell curve, suggesting older projects have more content and so more tests. Newer projects have not built up enough test content yet, and as time progresses we expect that these projects would add more test cases.
4.1 Objective

This part of thesis aims to study correlation between the test code and production code, and various other project development characteristics, using three big open source projects. We are trying to answer following questions:

RQ1 Does the testing effort change over multiple releases? Does the Test LOC increase at the same rate as the overall LOC?

- RQ2: Is there any correlation between cyclomatic complexity, Test LOC and code coverage?

- RQ3: How does the complexity, Test LOC and code coverage changes over a period of time?

Based on the analysis, three repositories were chosen that had a good percentage of test code. For each chosen repository, three releases were analyzed to see how code coverage was changed in each release. A complexity analysis on the code base was also done.
4.2 Motivation:

Unit Tests are important for maintaining quality in software systems. An important measure of unit tests is code coverage. Another factor that affects quality is cyclomatic complexity. Cyclomatic complexity is a measure of complexity and when it increases that could imply that the code has more defects, and also could mean it is difficult to cover using unit tests. We study the testing effort on a number of large projects

4.3 Tools Used

The evaluation of code coverage and cyclomatic complexities require automated tools for application to a large number of systems. Other tools like IntelliJ IDEA, a tool for proprietary code for analyzing tool, was investigated but we used Clover, Sonargraph, and Eclemm. In addition, Cygwin was used to run the data analysis. We will now explain the use of each tool.

4.3.1 Sonargraph –

Sonargraph Architect is a static code analysis tool that can measure the complexity of various repositories and their packages. It builds an in-memory model of your software project. It is fast and can work on a user’s computer. Sonargraph provides visual tools for analyzing the software system. It alerts users concerning architectural issues. It also shows dependency structure and displays various software metrics. We used Sonargraph to calculate the cyclomatic complexity of software systems over multiple releases.
4.3.2 Clover-

Clover is a code-coverage tool that also provides other metrics including lines of code of application classes and test classes, and complexity of a project and individual classes. It also can determine the top most complex classes. It is a very good tool to find which parts of your project are error prone that need further examination. Clover presents information using easy-to-comprehend user interface.

4.3.3 EclEmma -

EclEmma is java code coverage tool for Eclipse, which is used to analyze code coverage and understand which particular portions of the classes, are not covered by a unit tests. It also calculates code coverage of classes and packages. This was used to calculate code coverage for multiple versions of a repository, to determine if there is any correlation between code coverage and complexity.

4.3.4 Cygwin –

Cygwin is a unix-like environment and command like interface for Windows. Cygwin was used to perform analysis on the cloned git repositories. Unix commands were used to differentiate between production files and test files. Specifically, the Unix commands grep and find were used to find the test files and for calculating the source and test lines of code.
4.4 Description of repositories

We start by describing the three repositories that were selected for detailed analysis. We then briefly describe some of the build issues with each of the repositories.

4.4.1 Hibernate

Hibernate ORM is an object-relational mapping library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database. Hibernate provides tools for mapping from Java classes to database tables and also provides data query tools. It also provides data query and retrieval functionality. It provides Hibernate Query language – HQL, which allows SQL like queries against hibernate data objects.

4.4.2 Hadoop

Hadoop is an open source software framework that stores large data sets. These datasets are used by users and/or contributors. Hadoop is an Apache product, and is framework for the processing of small datasets and large datasets. Hadoop is based on Google GFS and MapReduce. Hadoop common module contains features that support other modules of Hadoop.

4.4.3 Spring Framework

The Spring Framework is an open source application framework and inversion of control container for the Java platform. The framework's core features can be used by any Java application, but there are extensions for building web applications on top of the Java EE platform. Although the framework does not impose any specific programming model,
it has become popular in the Java community as an alternative to, replacement for, or even addition to the Enterprise JavaBean (EJB) model. The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform. A key element of Spring is infrastructural support at the application level. Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments.

4.5. Installing and building Repositories

   Each project required a different build, and in many cases different releases had different builds, with specific dependencies.

4.5.1 Hibernate

   Hibernate-orm-4.3.1 is built using the Gradle build-automation tool. Eclipse has a plugin for Gradle, which helps to import a Gradle project. The Hibernate project was imported using the plugin. However to run the Eclemma tool, all build errors in a project should be resolved, otherwise none of the tests would run. To resolve the errors we need to ensure all project references and libraries were set properly. Hibernate documentation also has commands that help us to build the project. In addition, the IDE Eclipse reported cyclic dependency errors on this project. The solution was quite simple, as all that was required was to configure Eclipse to ignore those errors. So to build a release various documentation and errors needs to be researched. After resolving those errors, the 3800 tests were able to be ran and I was able to get the accurate code coverage.
Hibernate-3.6.1 is built using the build tool Apache Maven. The Hibernate project documentation had instructions to build the 3.x release using Maven and also how to generate the Eclipse files needed to import the project into Eclipse. Also this version also required that we set libraries and project references correctly.

4.5.2 Hadoop

Hadoop was built using Maven. The repository contains the file Building.txt that has all the build instructions. Maven commands were executed from command prompt. The commands build the source code also got the required libraries from the required repositories. The requirements were JDK 1.6, Maven 3.1.1, ProtocolBuffer 2.4.1, and CMake 2.6.

4.5.3 Spring Framework

Firstly Eclipse STS (Spring Tool Suite) was downloaded and installed. This is a customized Eclipse IDE that makes Spring-based application development easy. The Spring project provides a batch file that does the task of building the project and provides instructions to import the project into Eclipse. This was the most convenient and error free build of all the projects. The unit tests also ran quickly and were no or very few failures. For this project, we analyzed Spring 4.2.0, Spring 3.2.8 and Spring 3.2.0.

4.6 Repository Analysis

In this chapter we describe the detailed analysis that was done on the three selected repositories.
4.6.1 Criteria

We selected three repositories for detailed analysis. The criteria for selecting repository is as follows:

1. Select repositories that have considerable test lines of code – 30% or more.
2. Select repositories that have been existing for some time, meaning they have multiple releases that can be analyzed.
3. Select repositories that have good amount of code base that have been added over years.
4. Select repositories that have multiple committers, meaning famous open source projects

The criteria above were used since it would give us a more realistic picture of open source projects.

We selected three releases of each repository. The criteria for selecting releases is as follows:

1. Calculate the range from the date of first release and latest release.
2. Select the first release
3. Select the last release
4. Select a release closest to the middle range.

The source code of each release was downloaded from GitHub. The study was done on the core component of each repository. This ensured that code in subsequent releases of a system was based on the previous release. This ensured that code reflected
changes from one release to another. Each project was build by instructions given in the release. This sometimes took hours to days based on project and release. After the release was build all tests were run to test coverage was measured using EclEmma. For each release complexity was calculated using Sonargraph.

4.6.2 Analysis

Below we present analysis for each of the three-selected repositories. Table 1 shows detailed analysis for Hibernate

<table>
<thead>
<tr>
<th>Repository</th>
<th>LOC</th>
<th>Test LOC</th>
<th>Test Loc %</th>
<th>Code Coverage</th>
<th>Logical Complexity</th>
<th>Avg method complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hibernate 4.3.0–Core</td>
<td>665081</td>
<td>138463</td>
<td>21</td>
<td>72.5</td>
<td>0.4</td>
<td>1.6</td>
</tr>
<tr>
<td>Jan 22, 2014</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hibernate 4.1.2–Core</td>
<td>569796</td>
<td>121000</td>
<td>21</td>
<td>74.1</td>
<td>0.4</td>
<td>1.6</td>
</tr>
<tr>
<td>April 5, 2012</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hibernate 3.6.1–Core</td>
<td>261631</td>
<td>88460</td>
<td>33</td>
<td>69.1</td>
<td>0.5</td>
<td>2.1</td>
</tr>
<tr>
<td>Feb 2, 2011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total LOC vs Test LOC

We observe that Test LOC is increasing with production LOC as shown in Figure 10. There was a larger increase in total LOC between releases one and two whereas slower increase in total LOC between release 3 and release 2.
Comparing the first release and second we see that the average method complexity, Complexity per LOC has decreased whereas Test LOC% has also decreased whereas code coverage has increased as shown in Figure 11.

Figure 13: Hibernate Test LOC vs Total LOC
In this section we analyze three releases of Hadoop repositories. Table 2 shows results for detailed analysis of Hadoop.
Table 2: Hadoop detailed analysis

<table>
<thead>
<tr>
<th></th>
<th>Total Lines of Code</th>
<th>Test Lines of Code</th>
<th>Test Loc %</th>
<th>Code Coverage</th>
<th>Logical Complexity</th>
<th>Complexity per Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop-core – 2.3.0</td>
<td>220659</td>
<td>69822</td>
<td>32</td>
<td>51</td>
<td>0.17</td>
<td>1.8</td>
</tr>
<tr>
<td>02/24/2014</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hadoop-core - 2.2.0</td>
<td>210138</td>
<td>64720</td>
<td>31</td>
<td>49</td>
<td>0.17</td>
<td>1.9</td>
</tr>
<tr>
<td>10/07/2012</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hadoop-core – 1.0.4</td>
<td>107682</td>
<td>30150</td>
<td>27</td>
<td>45</td>
<td>0.16</td>
<td>2.1</td>
</tr>
<tr>
<td>02/12/2011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Test LOC % and code coverage % over 3 releases

We see in Figure 12 that code was increased substantially from release one to release two, however code was not increased that substantially from release two to release three.

Also the Test LOC increase was similar to Total LOC however at a lower scale.
Code Coverage-Complexity-Test LOC

We see in Figure 13 that complexity per method has reduced over the releases and code coverage has increased. Test LOC% and complexity per line of code has also slightly increased.

Figure 16: Hadoop detailed analysis
In this section we analyze three releases of Spring repositories. Test LOC, complexity are calculated using Sonargraph and Clover and are calculated as shown in Table 3 and analyzed.

### Table 3: Spring detailed analysis

<table>
<thead>
<tr>
<th></th>
<th>Total LOC</th>
<th>Test LOC</th>
<th>Test Loc %</th>
<th>Code Coverage</th>
<th>Avg Complexity per method</th>
<th>Logical Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spring – core-402</td>
<td>80101</td>
<td>22173</td>
<td>.28</td>
<td>53%</td>
<td>1.9</td>
<td>0.2</td>
</tr>
<tr>
<td>02/18/20 14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spring-</td>
<td>61054</td>
<td>19459</td>
<td>.32</td>
<td>66%</td>
<td>1.6</td>
<td>0.1</td>
</tr>
<tr>
<td>core-328</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spring-</td>
<td>59986</td>
<td>18491</td>
<td>.30</td>
<td>66%</td>
<td>1.6</td>
<td>0.2</td>
</tr>
<tr>
<td>core-320</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spring-</td>
<td>55679</td>
<td>16488</td>
<td>.30</td>
<td>65%</td>
<td>1.6</td>
<td>0.2</td>
</tr>
<tr>
<td>core-311</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Spring Test LOC and Total LOC

We see in Figure 14 that both Total LOC and Test LOC are increasing throughout releases. However Total LOC increased substantially more than Test LOC.
Code Coverage-Complexity-Test LOC

Code coverage has reduced and Average Method Complexity has increased over the releases as shown in Figure 15. Test LOC % has reduced.
4.7 Results and Discussions

Three repositories were selected for detailed analysis. The criteria for selecting the repositories were big open source projects that have multiple developers and that have existed for some period of time. This was done to ensure that results reflect the practices in general and are not indicative of short term projects or projects of few people.
The projects selected for detailed analysis had more than 30% of test code. Code coverage and complexity of each repository was done. These results were calculated for three release of each repository. Custom code was written to calculate the production LOC and Test LOC. Complexity was calculated using Sonargraph and code coverage was calculated using EclEmma.

In all three repositories we found out that as production LOC increased over consecutive releases so did the Test LOC. This means testing effort was maintained over the releases and so the software systems had testing requirements and developers paid attention to writing test code.

In the second set of plots we plotted Average method complexity, Test LOC %, complexity per line of code and code coverage.

We see that in Spring framework when Test LOC% increases and Avg method complexity decreases the code coverage % increases.

Similarly for Hadoop when Test LOC% increases and avg method complexity decreases we see that code coverage % increases. However for Hibernate we see that coverage remains more or less constant.

We see that when average method complexity was reduced, the code coverage seems to have been increased even though there was a decrease in the test code percentage. However when Test LOC percentage and average method complexity changes only slightly then the code coverage changes are minimal.

So the questions that we set out to answer in the beginning of the chapter can be answered now:
RQ1: Does the testing effort change over releases? Does the Test LOC increase at the same rate as the overall LOC? The Test LOC was approximately around 30% of the total LOC. There were instances of release where Test LOC was reduced but in general we saw that the release maintained the Test LOC percentage over releases. This means testing effort was maintained over multiple releases and test code also increased as production LOC increased.

RQ2: Is there any correlation between complexity and Test LOC and code coverage? We saw that when average method complexity was reduced, the code coverage seems to have been increased even though there was a decrease in the test code percentage. However when Test LOC percentage and average method complexity changes only slightly then the code coverage changes are minimal.

RQ3: How does the complexity, Test LOC and code coverage changes over a period of time? We saw that total LOC and Test LOC increased in all subsequent released for all the three systems studied. However we did not see such a trend for code coverage. We however see that code coverage seems to be dependent on other factors like average method complexity. More extensive study needs to be done to research if any definitive correlation exists. From the preliminary studies done it appears there seems to be a correlation between Test LOC average method complexity and code coverage.
CHAPTER V

CONCLUSIONS AND FUTURE WORK

Testing is an important aspect of a stable and successful system. In this thesis we analyzed testing effort for a large number of repositories. We also analyzed testing effort in detail for three repositories over multiple releases. We also evaluated selected software metrics for the detailed analysis over multiple releases.

We analyzed 100 repositories and the code for these open source repositories was retrieved using GitHub. Using bash commands and custom Java code we analyzed the code. The files that had the word Junit were considered as test files and remaining files were considered production code. We found Test LOC and Production LOC and also found test number of files and production number of files. We found that Test LOC was quite behind the Production LOC. The Test LOC did increase as Production LOC increased. The Test LOC was in the range 0-30% irrespective of the size of the project. Similarly the Test numbers of files were quite behind the Production number of files. The test number of files increased as production number of files increased. The percentage numbers of test files were also in the 0-30% for most projects irrespective of the size of the project. We found out that testing effort in general has been lagging. We found out that the testing effort is roughly 15% of total LOC.
Code coverage is an important metric for confirming if a software system has sufficient unit tests or not. The goal is for a system to have 100% code coverage, meaning that executing the entire test suite we would be sure to test all of the production code. However, this is not easily attainable, and it is often recommended that a more practical 95% figure be used.

Cyclomatic complexity tells us complexity of code. Larger systems definitely have bigger complexity. So we use the Avg method complexity and Complexity per line of code in our analysis. We see that when complexity per LOC is increased and test LOC% is not considerably high then code coverage reduces. That means when complexity per LOC is increased then there has to be larger testing effort required to maintain the code coverage.

We did detailed analysis of three selected repositories. These three repositories were big open source projects, had good amount of test code and existed for quite some time. We analyzed three releases of each repository. For each release we found their test LOC, Production LOC, code coverage, average method cyclomatic complexity and complexity per line of code. We tried to investigate if there is any relation between Test LOC, code coverage and cyclomatic complexity. Test LOC and Production LOC was determined using the bash commands and custom Java code. Code coverage was found using the Eclemma plugin of Eclipse. In order for Eclemma to give correct result we had to build each release. We found the cyclomatic complexity using Clover.

We found that the subsequent releases of each system increased the Test LOC and also more or less maintained the Test LOC percentage. We also found out that Test LOC cannot by itself determine the code coverage. It is dependent on other metrics like
average method complexity. We saw that when average method complexity was reduced, the code coverage increased.

We analyzed the core packages of the Hibernate, Spring and Hadoop. We wanted to see how a project behavior changes over a period of time and over multiple releases. We see that the selected repositories are doing their diligence as far as testing effort is concerned.

Further analysis needs to be done with even more repositories, including repositories of projects written in other programming languages besides Java. The tools and approaches used here can be easily adapted. Of even more importance would be further analysis of the history of systems, as was done in the second analysis. This is more difficult to do, as it required each system to be built for each version, with dependencies that could be different for each version.

This further analysis would help to get a more generalized result. Hopefully, the tools and methodology used here can be used as a guide.
REFERENCES


APPENDICES
# APPENDIX A

## LIST OF REPOSITORIES

<table>
<thead>
<tr>
<th>Repository Name</th>
<th>First Commit</th>
<th>Total LOC</th>
<th>Test LOC</th>
<th>% Test LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android</td>
<td></td>
<td>26773</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>Androidannotations</td>
<td>12/22/10</td>
<td>42515</td>
<td>5541</td>
<td>13.033</td>
</tr>
<tr>
<td>AndroidTraining</td>
<td>4/5/13</td>
<td>47680</td>
<td>112</td>
<td>0.235</td>
</tr>
<tr>
<td>ansj_seg</td>
<td>9/4/12</td>
<td>9200</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>Astrid</td>
<td>12/23/08</td>
<td>225547</td>
<td>3866</td>
<td>1.714</td>
</tr>
<tr>
<td>Atmosphere</td>
<td>4/27/10</td>
<td>53441</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>bigbluebutton</td>
<td>5/21/08</td>
<td>105754</td>
<td>410</td>
<td>0.388</td>
</tr>
<tr>
<td>Buck</td>
<td>4/18/13</td>
<td>156977</td>
<td>43578</td>
<td>27.761</td>
</tr>
<tr>
<td>calabash-android</td>
<td>2/5/12</td>
<td>114111</td>
<td>437</td>
<td>0.383</td>
</tr>
<tr>
<td>Cassandra</td>
<td>3/2/09</td>
<td>290934</td>
<td>32707</td>
<td>11.242</td>
</tr>
<tr>
<td>Cgeo</td>
<td>7/11/11</td>
<td>290934</td>
<td>2234</td>
<td>0.768</td>
</tr>
<tr>
<td>ChatSecureAndroid</td>
<td>3/19/10</td>
<td>53715</td>
<td>1038</td>
<td>1.932</td>
</tr>
<tr>
<td>commons</td>
<td>6/25/09</td>
<td>67397</td>
<td>20769</td>
<td>30.816</td>
</tr>
<tr>
<td>cw-omnibus</td>
<td>3/12/12</td>
<td>103940</td>
<td>918.</td>
<td>0.883</td>
</tr>
<tr>
<td>databus</td>
<td>10/30/12</td>
<td>209348</td>
<td>19242</td>
<td>9.191</td>
</tr>
<tr>
<td>datafu</td>
<td>9/22/11</td>
<td>32489</td>
<td>4276</td>
<td>13.161</td>
</tr>
<tr>
<td>druid</td>
<td>10/23/12</td>
<td>130692</td>
<td>28442</td>
<td>21.763</td>
</tr>
<tr>
<td>druid-1</td>
<td>5/11/11</td>
<td>271927</td>
<td>120063</td>
<td>44.153</td>
</tr>
<tr>
<td>elclim</td>
<td>8/13/05</td>
<td>55840</td>
<td>9126</td>
<td>16.343</td>
</tr>
<tr>
<td>eclipse-themes</td>
<td>7/17/12</td>
<td>6862</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>egit-github</td>
<td>3/23/11</td>
<td>46127</td>
<td>14086</td>
<td>30.537</td>
</tr>
<tr>
<td>elasticsearch</td>
<td>2/8/10</td>
<td>564663</td>
<td>119362</td>
<td>21.139</td>
</tr>
<tr>
<td>elephant-bird</td>
<td>3/24/10</td>
<td>28504</td>
<td>6173</td>
<td>21.657</td>
</tr>
<tr>
<td>erjang</td>
<td>11/9/09</td>
<td>72761</td>
<td>1641</td>
<td>2.255</td>
</tr>
<tr>
<td>FBReaderJ</td>
<td>10/7/07</td>
<td>83797</td>
<td>1234</td>
<td>1.473</td>
</tr>
<tr>
<td>fitnesse</td>
<td>3/26/05</td>
<td>83460</td>
<td>38600</td>
<td>46.250</td>
</tr>
<tr>
<td>flume</td>
<td>6/15/10</td>
<td>89827</td>
<td>37045</td>
<td>41.240</td>
</tr>
<tr>
<td>gephi</td>
<td>3/2/09</td>
<td>225372</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>Project</td>
<td>Date</td>
<td>U</td>
<td>N</td>
<td>CR</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------</td>
<td>---------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>gradle</td>
<td>3/10/08</td>
<td>308888</td>
<td>41528</td>
<td>13.444</td>
</tr>
<tr>
<td>grails-core</td>
<td>7/2/05</td>
<td>127390</td>
<td>8680</td>
<td>6.814</td>
</tr>
<tr>
<td>gremlin</td>
<td>11/7/09</td>
<td>8075</td>
<td>2218</td>
<td>27.467</td>
</tr>
<tr>
<td>groovy-core</td>
<td>8/28/03</td>
<td>240902</td>
<td>6958</td>
<td>2.888</td>
</tr>
<tr>
<td>h2o</td>
<td>2/5/13</td>
<td>105760</td>
<td>12318</td>
<td>11.647</td>
</tr>
<tr>
<td>hadoop-20</td>
<td>12/1/10</td>
<td>775731</td>
<td>191919</td>
<td>24.740</td>
</tr>
<tr>
<td>hadoop-common</td>
<td>5/19/09</td>
<td>1126206</td>
<td>382789</td>
<td>33.989</td>
</tr>
<tr>
<td>hazelcast</td>
<td>12/15/08</td>
<td>233597</td>
<td>47519</td>
<td>20.342</td>
</tr>
<tr>
<td>hbase</td>
<td>4/3/07</td>
<td>830827</td>
<td>177395</td>
<td>21.352</td>
</tr>
<tr>
<td>hector</td>
<td>2/10/10</td>
<td>45438</td>
<td>11202</td>
<td>24.653</td>
</tr>
<tr>
<td>hibernate-orm</td>
<td>6/29/07</td>
<td>861576</td>
<td>217449</td>
<td>25.239</td>
</tr>
<tr>
<td>intellij-community</td>
<td>11/5/04</td>
<td>3806602</td>
<td>152795</td>
<td>4.014</td>
</tr>
<tr>
<td>jedis</td>
<td>6/11/10</td>
<td>27092</td>
<td>7575</td>
<td>27.960</td>
</tr>
<tr>
<td>jersey</td>
<td>2/14/12</td>
<td>350029</td>
<td>130336</td>
<td>37.236</td>
</tr>
<tr>
<td>Jna</td>
<td>10/1/98</td>
<td>117116</td>
<td>16674</td>
<td>14.237</td>
</tr>
<tr>
<td>jOOQ</td>
<td>7/24/11</td>
<td>453982</td>
<td>37777</td>
<td>8.321</td>
</tr>
<tr>
<td>jvm-serializers</td>
<td>11/16/08</td>
<td>20748</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>k-9</td>
<td>10/28/08</td>
<td>116731</td>
<td>1589</td>
<td>1.361</td>
</tr>
<tr>
<td>kotlin</td>
<td>11/8/10</td>
<td>341157</td>
<td>58414</td>
<td>17.122</td>
</tr>
<tr>
<td>legacy-jclouds</td>
<td>4/11/09</td>
<td>562652</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>libgdx</td>
<td>3/6/10</td>
<td>341234</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>liferay-portal</td>
<td>4/3/06</td>
<td>3255078</td>
<td>222942</td>
<td>6.849</td>
</tr>
<tr>
<td>liquibase</td>
<td>4/27/09</td>
<td>95873</td>
<td>19735</td>
<td>20.585</td>
</tr>
<tr>
<td>lombok</td>
<td>6/8/09</td>
<td>978794</td>
<td>178016</td>
<td>18.187</td>
</tr>
<tr>
<td>lucene-solr</td>
<td>3/17/10</td>
<td>978794</td>
<td>178016</td>
<td>18.187</td>
</tr>
<tr>
<td>mahout</td>
<td>1/15/08</td>
<td>186028</td>
<td>42331</td>
<td>22.755</td>
</tr>
<tr>
<td>MinecraftForge</td>
<td>7/12/11</td>
<td>20830</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>mongo-hadoop</td>
<td>8/23/10</td>
<td>10826</td>
<td>1053</td>
<td>9.727</td>
</tr>
<tr>
<td>neo4j</td>
<td>5/24/07</td>
<td>487062</td>
<td>176413</td>
<td>36.220</td>
</tr>
<tr>
<td>netty</td>
<td>8/8/08</td>
<td>172487</td>
<td>25157</td>
<td>14.585</td>
</tr>
<tr>
<td>Nightweb</td>
<td>9/4/12</td>
<td>198275</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>nutz</td>
<td>1/16/09</td>
<td>101828</td>
<td>16259</td>
<td>15.967</td>
</tr>
<tr>
<td>OpenRefine</td>
<td>4/26/10</td>
<td>82057</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>OpenTripPlanner</td>
<td>9/10/09</td>
<td>151045</td>
<td>15814</td>
<td>10.470</td>
</tr>
<tr>
<td>opentsdb</td>
<td>4/10/10</td>
<td>17395</td>
<td>1619</td>
<td>9.307</td>
</tr>
<tr>
<td>orientdb</td>
<td>3/29/10</td>
<td>286870</td>
<td>3383</td>
<td>1.179</td>
</tr>
<tr>
<td>parboiled</td>
<td>10/5/09</td>
<td>29046</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>phoenix</td>
<td>1/17/13</td>
<td>144955</td>
<td>48794</td>
<td>33.661</td>
</tr>
<tr>
<td>phonegap-plugins</td>
<td>2/3/10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PhotoView</td>
<td>7/17/12</td>
<td>2922</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>photup</td>
<td>6/28/12</td>
<td>13095</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>Platform Frameworks Base</td>
<td>Date</td>
<td>Downloads</td>
<td>Unique Users</td>
<td>Population</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------</td>
<td>-----------</td>
<td>--------------</td>
<td>------------</td>
</tr>
<tr>
<td>Presto</td>
<td>8/8/12</td>
<td>173333</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>Processing-JS</td>
<td>5/12/08</td>
<td>31877543281</td>
<td>3.119</td>
<td></td>
</tr>
<tr>
<td>Rest.li</td>
<td>9/18/12</td>
<td>244084316</td>
<td>0.129</td>
<td></td>
</tr>
<tr>
<td>Retrofit</td>
<td>9/6/10</td>
<td>119864382</td>
<td>36.559</td>
<td></td>
</tr>
<tr>
<td>Robospice</td>
<td>9/19/12</td>
<td>32136787</td>
<td>2.449</td>
<td></td>
</tr>
<tr>
<td>Robovm</td>
<td>8/1/10</td>
<td>1363517240219</td>
<td>17.618</td>
<td></td>
</tr>
<tr>
<td>Rootbeer1</td>
<td>8/11/12</td>
<td>388290</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>Rstudio</td>
<td>12/7/10</td>
<td>165546936</td>
<td>0.565</td>
<td></td>
</tr>
<tr>
<td>Rundeck</td>
<td>10/11/10</td>
<td>9669224000</td>
<td>24.821</td>
<td></td>
</tr>
<tr>
<td>RxJava</td>
<td>3/18/12</td>
<td>7672433901</td>
<td>44.186</td>
<td></td>
</tr>
<tr>
<td>Sensei</td>
<td>2/3/10</td>
<td>9417910214</td>
<td>10.845</td>
<td></td>
</tr>
<tr>
<td>SeriesGuide</td>
<td>7/3/11</td>
<td>4228336</td>
<td>0.085</td>
<td></td>
</tr>
<tr>
<td>Sikuli</td>
<td>1/25/10</td>
<td>5387626</td>
<td>0.048</td>
<td></td>
</tr>
<tr>
<td>Simplegit-Progit</td>
<td>3/15/08</td>
<td>115561810</td>
<td>15.663</td>
<td></td>
</tr>
<tr>
<td>Solandra</td>
<td>8/23/09</td>
<td>73967396</td>
<td>12.182</td>
<td></td>
</tr>
<tr>
<td>Spoon</td>
<td>7/10/12</td>
<td>1846138271</td>
<td>4.480</td>
<td></td>
</tr>
<tr>
<td>Spring-Framework</td>
<td>2/26/12</td>
<td>5969212277</td>
<td>20.567</td>
<td></td>
</tr>
<tr>
<td>Terasology</td>
<td>4/1/12</td>
<td>1168130</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>Twidere</td>
<td>10/2/11</td>
<td>13151333542</td>
<td>25.505</td>
<td></td>
</tr>
<tr>
<td>Usergrid-Stack</td>
<td>8/1/13</td>
<td>624859271</td>
<td>14.837</td>
<td></td>
</tr>
<tr>
<td>Vert.x</td>
<td>10/19/12</td>
<td>85510</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>Voldemort</td>
<td>4/17/11</td>
<td>21006047837</td>
<td>22.773</td>
<td></td>
</tr>
<tr>
<td>Weiciyuan</td>
<td>7/26/12</td>
<td>736190</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>Wildfly</td>
<td>5/28/10</td>
<td>1102779263253</td>
<td>23.872</td>
<td></td>
</tr>
<tr>
<td>XobotOS</td>
<td>4/30/12</td>
<td>168247928095</td>
<td>1.670</td>
<td></td>
</tr>
<tr>
<td>YCSB</td>
<td>4/19/10</td>
<td>116060</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>Zookeeper</td>
<td>5/19/08</td>
<td>9316230809</td>
<td>33.070</td>
<td></td>
</tr>
<tr>
<td>Zxing</td>
<td>10/22/07</td>
<td>646078655</td>
<td>#DIV/0!</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX B

CODE TO FORK GIT REPOSITORIES

```java
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import org.eclipse.egit.github.core.Repository;
import org.eclipse.egit.github.core.RepositoryId;
import org.eclipse.egit.github.core.SearchRepository;
import org.eclipse.egit.github.core.client.GitHubClient;
import org.eclipse.egit.github.core.service.CommitService;
import org.eclipse.egit.github.core.service.RepositoryService;

public class SearchGitHub {
    GitHubClient client = new GitHubClient();
    BufferedWriter bw;

    public void setUser(){
        client.setCredentials("scholarpallavi", "mayankpal16");
    }

    public void retrieveRepos() throws IOException{
        RepositoryService service = new RepositoryService(client);
        Map<String, String> searchQuery = new HashMap<String, String>();
        searchQuery.put("size", ">=30000");
        searchQuery.put("language","java");
        List<SearchRepository> searchRes = service.searchRepositories(searchQuery);
        for(SearchRepository r : searchRes){
            // bw.write( r.getName() + "  " + r.getSize() + r.getForks());
            // bw.newLine();
            service.forkRepository(r);
        }
        try {
            for (Repository repo : service.getRepositories("scholarpallavi")){
                CommitService cs = new CommitService(client);
                RepositoryId repoId = RepositoryId.createFromUrl(repo.getCloneUrl());
                //System.out.println(cs.getCommits(repo).get(0).getCommitter().getCreatedAt());
                //System.out.println(cs.getCommits(repo).get(0).getCommitter().getEmail());
                bw.write(repo.getCloneUrl());
                bw.newLine();
            }
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```
public void setUpFile()
{
    File file = new File("filename.txt");
    // if file doesn't exist, then create it
    if (!file.exists()) {
        try {
            file.createNewFile();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    FileWriter fw = null;
    try {
        fw = new FileWriter(file.getAbsoluteFile());
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    bw = new BufferedWriter(fw);
    System.out.println("Done");
}

public static void main(String args[])
{
    SearchGitHub api = new SearchGitHub();
    api.setUser();
    api.setUpFile();
    try {
        api.retrieveRepos();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}


import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.List;
import java.util.StringTokenizer;

public class ResultsAnalysis {
    public Map<String, ArrayList<Node>> repsTestClasses = new HashMap<String, ArrayList<Node>>();
    public Map<String, ArrayList<Node>> repsAllClasses = new HashMap<String, ArrayList<Node>>();
    public List<String> resultString = new ArrayList<String>();
    int totalLoc = 0;

    public void readLine(String fileName){
    }

    public class Node{
        int length = 0;
        String name="";
    }

    public List<String> getTotal(String FileName){
        BufferedReader br = null;
        try {
            String sCurrentLine;

            //br = new BufferedReader(new FileReader("C:\Users\Mayank\thesis\reps\results.txt"));
            br = new BufferedReader(new FileReader("results.txt"));
            while ((sCurrentLine = br.readLine()) != null) {
                if(sCurrentLine.endsWith("total")){
                    StringTokenizer st = new StringTokenizer(sCurrentLine);
                    totalLoc += Integer.parseInt(st.nextToken());
                    System.out.println("totalLoc "+totalLoc);
                }
            }
        }catch (IOException e) {
            e.printStackTrace();
        }finally {
            if (br != null)br.close();
        }
    }

    public void main(String[] args){
    }
}
```java
catch (IOException ex) {
    ex.printStackTrace();
}
return null;
}

public List<String> getTestReps(){
    BufferedReader br = null;
    try {
        String sCurrentLine;
        br = new BufferedReader(new FileReader("resulttake2test.txt");
        while ((sCurrentLine = br.readLine()) != null) {
            if(!sCurrentLine.endsWith("total")){
                StringTokenizer st = new StringTokenizer(sCurrentLine);
                Node n = new Node();
                n.length = Integer.parseInt(st.nextToken());
                String[] retval = st.nextToken().split("/", 2);
                n.name = retval[1];
                if(repsTestClasses.get(retval[0]) == null){
                    ArrayList<Node> nodeList = new ArrayList<Node>();
                    nodeList.add(n);
                    repsTestClasses.put(retval[0], nodeList);
                } else{
                    repsTestClasses.get(retval[0]).add(n);
                }
            }
        }
    } finally {
        try {
            if (br != null)br.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
    return null;
}

public List<String> getAllReps(){
    BufferedReader br = null;
    try {
        String sCurrentLine;
        br = new BufferedReader(new FileReader("resultake2.txt");
        int count = 0;
        while ((sCurrentLine = br.readLine()) != null) {
            if(!sCurrentLine.endsWith("total")){
                StringTokenizer st = new StringTokenizer(sCurrentLine);
                Node n = new Node();
                n.length = Integer.parseInt(st.nextToken());
                String[] retval = st.nextToken().split("/", 3);
                n.name = retval[2];
                if(repsAllClasses.get(retval[1]) == null){
                    ArrayList<Node> nodeList = new ArrayList<Node>();
                    nodeList.add(n);
                    System.out.println("0  " + retval[0]);
                    System.out.println("1  " + retval[1]);
                    System.out.println("2  " + retval[2]);
                    count++;
                } else{
                    repsAllClasses.get(retval[1]).add(n);
                }
            }
        }
    } finally {
        try {
            if (br != null)br.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
    return null;
}
```

System.out.println("count "+count);

}catch (IOException e) {
	e.printStackTrace();

}finally {
	ry {
		if (br != null)br.close();

}catch (IOException ex) {
		ex.printStackTrace();

}

return null;

public static void main (String [] args){
    ResultsAnalysis ra = new ResultsAnalysis();
    // ra.getTotal("ms");
    ra.getTestReps();
    ra.getAllReps();
    //displayResult
    ra.writeResult();
}

public void writeResult(){
    Iterator it = repsAllClasses.entrySet().iterator();
    double total = 0;
    int count = 0;
    int fileCount = 0;
    double maxLoc = 0;
    double srcTotalLOC,testTotalLOC;
    try {
        File file = new File("resultsperrepwithNumberResultTake2.txt");
        // if file doesnt exists, then create it
        if (!file.exists()) {
            file.createNewFile();
        }
        FileWriter fw = new FileWriter(file.getAbsoluteFile());
        BufferedWriter bw = new BufferedWriter(fw);
        srcTotalLOC = 0;
        testTotalLOC = 0;
        while (it.hasNext()) {
            bw.newLine();
            total = 0.0;
            fileCount = 0;
            Map.Entry pairs = (Map.Entry)it.next();
            List<Node> nodeList = (List<Node>)pairs.getValue();
            for(Node n : nodeList){
                total += n.length;
                fileCount++;
            }
            // bw.write(pairs.getKey() + " = " + total + " = " + fileCount);
            bw.write(pairs.getKey() + "   #   " + fileCount);
            srcTotalLOC += total;
            // avoids a ConcurrentModificationException
            double testtotal = 0.0;
            fileCount = 0;
            List<Node> testNodeList = repsTestClasses.get(pairs.getKey());
            while (it.hasNext()) {
                bw.newLine();
                total = 0.0;
                fileCount = 0;
                Map.Entry pairs = (Map.Entry)it.next();
                List<Node> nodeList = (List<Node>)pairs.getValue();
                for(Node n : nodeList){
                    total += n.length;
                    fileCount++;
                }
                // bw.write(pairs.getKey() + " = " + total + " = " + fileCount);
                bw.write(pairs.getKey() + "   #   " + fileCount);
                testTotalLOC += total;
                // avoids a ConcurrentModificationException
                double testtotal = 0.0;
                fileCount = 0;
                List<Node> testNodeList = repsTestClasses.get(pairs.getKey());
            }
        }
    }
    //write to file
    bw.close();
    fw.close();

}
if(testNodeList != null){
    for(Node n : testNodeList){
        testtotal += n.length;
        fileCount++;
    }
}
testTotalLOC += testtotal;
// bw.write(" + " Test LOC " + testtotal + " = " + fileCount);
bw.write(" + " # = " + fileCount);
double pc = testtotal/total;
if(pc > maxLoc) maxLoc = pc;
//bw.write(" --> " + pc);
count++;}

bw.write(" + " srcTotal " + srcTotalLOC);
bw.write(" + " testTotal " + testTotalLOC);
bw.close();
System.out.println("Done");
} catch (IOException e) {
    e.printStackTrace();
}
System.out.println("Number of reps " + count);
}
APPENDIX D

GIT COMMAND TO FORK 100 REPOSITORIES

"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/android.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/androidannotations.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/AndroidTraining.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/anaj seg.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/asturd.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/bigbluebutton.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/buck.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/calabash-android.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/cassandra.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/cgeo.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/ChatSecureAndroid.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/commons.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/cw-omnibus.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/databus.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/datafu.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/druid.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/druid-1.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/eclipse-themes.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/eclipse-theme-support.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/eclipse-theme-support-android.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/elasticsearch.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/elephant-bird.git"
"C:\Program Files (x86)\Git\bin\sh.exe." –login –i –c  "git clone https://github.com/scholarpallavi/elephant-bird-android.git"
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/mahout.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/MinecraftForge.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/mongo-hadoop.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/neo4j.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/netty.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/Nightweb.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/nutz.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/OpenRefine.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/OpenTripPlanner.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/opentsdb.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/orientdb.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/parboiled.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/phoenix.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/phonegap-plugins.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/PhotoView.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/photoup.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/platform_frameworks_base.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/presto.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/processing.js.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/processing.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/rest.li.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/robospice.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/robostorm.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/rootbeer-l.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/rstudio.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/rundeck.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/RxJava.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/sensei.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/SeriesGuide.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/sikuli.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/solandra.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/spoon.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/spring-framework.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/Terasology.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/titan.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/twidere.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/usergrid-stack.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/vitamioBundle.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/voldemort.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/weiciyuan.git
“C:\Program Files (x86)\Git\bin\sh.exe.” –login –i –c “git clone https://github.com/scholarpallavi/wildfly.git