Data, Methodology, and Study Design

Being an OSS project, the SVN commit history for Apache Ant was readily available. The raw file has 149,034 lines and 12,958 commits in total. The first and last commit took place on January 13, 2000 and January 27, 2014 respectively. For our analysis, we considered five releases, and the dates of release are shown in the following table.

|  |  |
| --- | --- |
| Version # | Month Year |
| 1.5 | July 10, 2002 |
| 1.6 | December 18, 2003 |
| 1.7 | December 19, 2006 |
| 1.8 | February 8, 2010 |
| 1.9 | March 7, 2013 |

*Fig 3.1: Apache Ant versions*

The SVN log is well structured, following an identical pattern for each commit and therefore from this history we were able to extract the relevant parts of each commit: revision ID, committer, date, path to modified file(s), and comment. We leveraged the date of the commit to associate to which version of Apache Ant a particular commit belongs.

The first four releases were used for analysis and the last one was used for validation. To facilitate the information retrieval we developed a Java project (\*add as footnote that the code is available in our Git?) aptly named LogParser which takes care of parsing, cleaning and filtering. During this process, we found that some of the commits were problematic in the sense that they were not related to actual code evolution but were instead performed for infrastructure, legal and cleanup reasons:

* changed the license/copyright (r637939, r439418, r276065, r276010 )
* removed the authors from code (r276208)
* cleanup imports / whitespaces (r273169, r276017)

Naturally, these commits were removed from our analysis since they typically modified all classes in the system, and inappropriately skewed the volatility results. As per our definition of volatility, we were only interested in commits which modified at least one Java class. We therefore also filtered out the changes to non-Java code (xml, html, etc.) and only considered classes which were modified (not added or removed).

In order to gather the data for our volatility buckets, we collected the information between one version to another and stored the unique set of classes impacted. The following table depicts the result of that analysis.

|  |  |  |
| --- | --- | --- |
| Version to Version | Number of Commit | Set of Java Classes |
| 15\_to\_16 | 434 | 788 |
| 16\_to\_17 | 630 | 1089 |
| 17\_to\_18 | 296 | 565 |
| 18\_to\_19 | 101 | <669> |

*Fig 3.2: Number of commit and impacted classes between versions*

From the three first collections of data, our aim was to match the modified classes into each of the three buckets: High Volatility (HV), Medium Volatility (MV) and Low Volatility (LV) (THIS MIGHT ALREADY BE THERE FROM THE INTRO). The HV bucket contains the classes which got changed in all three times, the MV buckets contains the classes which got changed two times in any of the collections, and the LV buckets contains the classes which got changed only once in any of the collections.

Since the target of our study was to compare the quality metrics between each bucket, we designed LogParser to create three output files: the set of the classes (including the path) in each volatility bucket. There were 294, 503, and 554 classes classified as HV, MV, and LV respectively.

The quality metrics chosen for comparison were a combination of class metrics (LCOM, LCOM Henderson, Cohesion, RFC, and LOC), as well as two system level metrics (MHF, and AHF). We adopted these well-defined metrics from the Chidamber and Kemerer suite, and MOOD respectively.

In order to develop the code which calculates these metrics, we leveraged JDeodorant, a well-designed and flexible framework which runs as an Eclipse plugin. [CITATION NEEDED] JDeodorant hides the complexity of obtaining the meta-model of the system under investigation (AST), and has a rich set of accessor methods in order to interrogate the system at any level.

In order to interact with the JDeodorant framework, we designed each metric as a separate class in the metrics package. We decided on a common usage pattern: each class would write its metric value to a map of “class name: metric value”. The design revealed that all the metric classes exhibited some common behavior so we designed an abstract parent class to take care of common tasks such as loading the volatility buckets and writing the results a separate, timestamped file. The abstract parent defined a template so that each child class would need to only implement one method, and write to the map defined in the parent. The abstract parent loaded the output of LogParser, and we therefore generated output files which contained the metric values for each class in each bucket. Here’s an example output for LCOM Henderson:

* 3, 0.3333333333333333
* 2, 0.8333333333333333
* 1, 0.5
* 1, 0.5
* 2, 0.8571428571428571
* 1, 0.0

The output file was a comma separated list where the first element indicated the volatility (1 is low, 2 is medium, and 3 is high), and the second value is the calculated metric. This format made it easy to import this data into R in order to visualize the boxplots.

Furthermore, LogParser was used to generate the output files which were used to calculate the association rules. We created a separate file for each collection (15\_to\_16, 16\_to\_17, 17\_to\_18, 18\_to\_19). The format for that output was as follows:

* RevisionID1, Java File 1, Java File 2[, .. , Java File n]
* RevisionID2, Java File 1, Java File 2[, .. , Java File n]
* …

Each line represented a single “transaction” (commit), and we only included lines where 2 or more Java classes were modified. The last file was used for validation.

Finally, in order to calculate the correlations between our metrics, we additionally designed some classes to store a summary of all metrics collected for each class. These classes are named “SummaryMetric” and “SummaryMetricCollector”.