# Empirical Study Comparing Quality Metrics with Code Volatility, and Investigating a Prediction Model

Amish Gala
Dept. of Software Engineering
Concordia University
Montreal, Canada
a_gala@encs.concordia.ca

Shams Azad
Dept. of Software Engineering
Concordia University
Montreal, Canada
sh_aza@encs.concordia.ca

Vinodkumar Raghu
Dept. of Software Engineering
Concordia University
Montreal, Canada
v_raguna@encs.concordia.ca

Md Tajmilur Rahman
Dept. of Software Engineering
Concordia University
Montreal, Canada
rupak.karmadhar@gmail.com

***Abstract*— It is often quoted that "without data, you only have opinions." Keeping that in mind, metrics can and are extensively used to measure the quality of software. In this study, we have analyzed the SVN history of five versions of Apache Ant in order to determine the code volatility of Java classes and to perform association rule mining. We consider a class to be volatile between two versions if there is at least one commit which modified that particular class. We built a tool named LogParser which grouped classes found in the commit logs into different volatility buckets. For our study, we considered three buckets: classes modified three times, two times, and once. Using JDeodorant, we then calculated a set of metrics on the classes in each bucket. We found that classes in the higher volatility bucket exhibited poorer quality metrics compared to classes in the lower volatility bucket. In parallel, two system level metrics were calculated for each bucket; however, it was observed that the lower volatility bucket actually had worse system level metrics compared to the other two. In the second part of our analysis we applied association rule mining on the historical data of Apache Ant. The goal is to guide the developer about related code changes. We received a precision of about 10% and a recall of about 2% for the predictor model.**

***Keywords*— *Code volatility, object oriented quality metrics, system quality metrics, association rule, Apriori, Precision, Recall***

## I. Introduction

How does one maintain the quality of software? This question is the source of much debate across organizations of different sizes and domains. Software companies aim for systems with longevity and as part of continuous improvement and feedback, many institution track metrics of these systems. Software metrics are designed to help developers maintain and check the quality of a software system. Some of the virtues of actively tracking metrics are improved risk analysis, cost analysis and planning [12]. However, metrics alone are unable to solve the problem of assessing the quality. According to Olague et al., there should be empirical validation of all the proposed OO metrics to ensure their appropriate use [1]. Ultimately we would like to minimize the cases where a metric is correct from a theoretical standpoint but has limited use in a practical sense.

In our approach we implemented six different known metrics: four of them based on class level calculation, and two of them based on system level. One metric (LCOM) is provided in JDeodorant as a starting point, thus overall we validated seven different metrics. The metrics we have chosen for our study are the following: LCOM [2], LCOM-Henderson [3], Cohesion [5], RFC, Lines of Code, MHF and AHF [4]. Details on each metric are available in section III.

In this paper, we introduce the notion of code volatility and compare it to quality metrics, which is a novel approach. Volatility describes the propensity for a given code fragment to change over different releases. We define volatility as the number of modifications that has been made to a Java class across a set of different releases. A high volatility (HV) class is one which is modified in most of the releases. A medium volatility (MV) class is one that has been modified in some of the releases, and finally a low volatility (LV) class in one that has been modified only a single time.

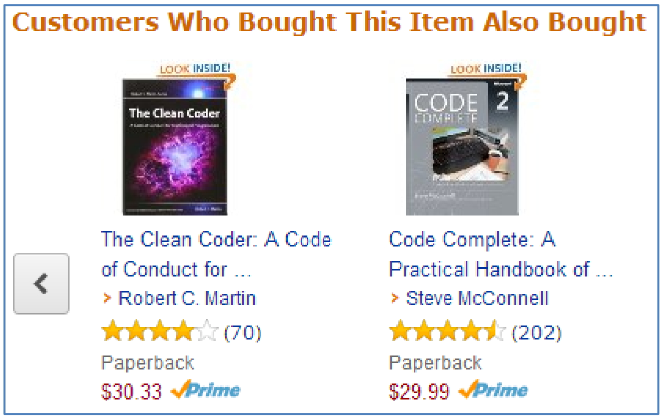
We inspected four consecutive versions of Apache Ant, a publically available OSS project used for software build and dependency management. We used the information contained in the SVN revision history in order to perform our calculations.

We were interested in observing the correlation (if any) between the different metrics and the class volatility. It is our belief that measuring code volatility may be used by organizations who wish to prioritize refactoring and unit testing activities – code volatility adds another dimension as to risk planning and mitigation.

Furthermore, we used the historical data to build a predictive model to forecast future changes based on current changes. Using Apache Ant versions 1.5 through 1.9 for our study, we have applied association rule mining for generating rules on how classes change together and used these rules to predict which classes will be changed as a result. Association rules are conditional statements that help in finding relationship between random objects in an information repository. Given a large dataset of transactions, the rules can give information on how strongly two objects are related to each other.

Association rules consist of two parts, an antecedent (if) and a consequent (then). An antecedent is an item found in the data. A consequent is an item that is found in combination with the antecedent. While there are different types of association rule mining, we leveraged the Apriori algorithm [6]. It is based on two widely accepted concepts: support and confidence. Support is an indication of how frequently the items appear in the dataset. Confidence indicates the number of times that the consequent appears with the antecedent.

Programmers use association rules to build applications capable of machine learning. Machine learning is a type of artificial intelligence that seeks to build programs with the ability to become more efficient without being explicitly programmed.


Customers Who Bought This Item Also Bought

The Clean Coder: A Code of Conduct for ...
› Robert C. Martin
★★★★☆ (70)
Paperback
$30.33 ✓Prime

Code Complete: A Practical Handbook of ...
› Steve McConnell
★★★★½ (202)
Paperback
$29.99 ✓Prime

**Figure 1. Association Rules in Action**

Association rules are widely used to find hidden trends and patterns. For example, Amazon's suggestions as shown in figure 1 is an implementation example of using association rules. Also, in the context of grocery shopping, given a dataset of store transactions, inferences can be made such as: "if a person buys milk then there is a 60% chance that he will buy eggs too."

The format of the paper proceeds as follows. Section two identifies and describes the related works for both code volatility and association rule mining. Here we present past works of other researches and show that the concept of code volatility is something novel. In section III, we introduce the metrics chosen for our analysis and provide a short primer on one of the more complicated implementations. Next in section IV, we describe the project which we analyzed and elaborate on the approach we've taken in order to prepare and calculate the desired metrics. A more in-depth description is provided regarding the LogParser tool we built, the decisions we made regarding the association rule mining calibration, and present the results of our investigation and calculations. In section V we discuss some of the implications of our results, and some of the threats to our approach. Finally, in section VI, we draw some conclusions and summarize our findings.

## II. Related Work

### A. *Code Volatility*

It proved difficult to find related works which also consider the correlation between quality metrics and code volatility. However, we mention the following papers which describe methods similar to which we used in our approach.

Munson and Elbaum looked at comparing the complexity of sequential builds in order to measure the impact of code change regarding fault proneness [7]. They aimed at determining a suitable fault surrogate (which they dubbed code churn), and furthermore established that code churn is associated with measures of program quality. This research has some similar themes compared to our research. Munson investigated how churn correlates with quality (faults); however, he didn't delve into the concept of occurrences of churn across multiple releases.

Graves et al. posit that "*In general, process measures based on the change history are more useful in predicting fault rates than product metrics of the code: For instance, the number of times code has been changed is a better indication of how many faults it will contain than is its length*" [8]. Here we clearly observe similar themes compared to our proposal of code volatility. Their model considers collections of files as an individual module, compared to our view that individual classes are the core building blocks. One of the arguments they make regarding standard complexity metrics is that they correlate too highly with lines of code: "*…numbers of lines of code in modules are not helpful in predicting numbers of future faults once one has taken into account numbers of times modules have been changed*." This work has similar findings to our research, most notably that using the sum of contributions from all changes in a module's history can effectively predict fault potential. They proceed one step further and also consider a module's age in the prediction calculation, which is something we did not do.

One other work which looks at code change and quality is Nagappan and Ball's research [9]. In it they describe two measures (M5 and M6) which take into account "the cumulative time that a file was opened for editing from the VCS". This concept was used to cross-check the other measures described in their research and it is interesting to our paper since it introduces the concept of quantifying volatility, albeit differently (total time open vs. # of changes across different releases). While this isn't exactly our definition of volatility, it does represent something similar.

### B. *Association Rule Mining*

A number of previous studies has been done on large OSS projects such as jEdit, Eclipse, Apache Ant, and Mozilla Firefox to detect the coupling between different Java files, classes and methods. The target of these studies was to build prediction models. Ying et al. developed an approach which is based on association rule mining to find recommendations on potential file changes [10]. For example if a developer changes a particular file, then her approach will recommend the other files that will need to be changed. Their findings were superior to ours mostly due to a richer set of data and by omitting confidence.

Similarly, Zimmermann et al. developed a tool named ROSE that works on association rule mining of CVS [11]. Their main concern was to find finer-grained entities, which was absent in Ying's work. They used association rule mining with minimum support and confidence. Their tool seems to be powerful in terms of suggesting and predicting the next likely code elements to change. It also prohibits and warns about incomplete changes. We took inspiration from both these works and built association rules in a similar fashion. We found that we were able to perform some predictions on future change; however, again their results were superior to ours for various reasons such as richer data set, and using a weighted set of rules (top 10).

Other researchers such as Ball et al. [13] have also used version history mining for determining a different perspective of software evaluation. Chen et al. [14] have also worked on CVS mining for fragments of source code but by using check-in comments. Although different researchers work on different areas of CVS mining, coupling between different modules was first defined by H. Gall [15] and associates and carry forward by many researchers later on. Most notably it was Zimmermann who first worked on code elements rather than whole modules or files.

## III. Metrics

### A. Metrics Definition

Our study involves seven well defined and well known metrics. A full definition of each one can be found in [2], [3], [4], and [5]. A summary is listed below.

TABLE I. Metrics analyzed

| Metric | Metric Description |
|---|---|
| LCOM (C&K) | Lack of Cohesion for a class. Two methods are cohesive if they access at least one common instance variable. |
| LCOM (Henderson) | Lack of Cohesion for a class. Measures how well a class uses its own attributes. The value is normalized compared to the C&K definition.<br>$LCOM = \frac{m - \frac{\sum_{i=1}^{a} p(A_i)}{a}}{m-1}$ |
| Cohesion (Briand) | Cohesion of a class. Measures how much a class depends on its own data, normalized to the product of the number of methods and the number of attributes.<br>$Coh = \frac{\sum_{i=1}^{a} p(A_i)}{m*a}$ |
| RFC (C&K) | Response for a class. Measures the cardinality of the number of methods within a class, along with the possible methods of other classes which may be called.<br>$RFC = \|RS\|$<br>$RS = M \cup R$<br>$R = \bigcup_{i=1}^{\|M\|} R_i$ |
| Size | Number of statements |
| MHF (MOOD) | Method hiding factor. The ratio of hidden methods compared to all methods for a given system. |
| AHF (MOOD) | Attribute hiding factor. Same as MHF, but looks at attributes instead of methods. |

### B. Metrics Implementation details

Most metrics above were calculated in a self-contained way. However, in order to calculate size, we needed to adjust the JDeodorant framework. Here we added an abstract method "count()" to the AbstractStatement class, and then leveraged the composite design pattern to have both subclasses (StatementObject, and CompositeStatementObject) implement this method appropriately.

Eric Raymond is often cited for his principle "smart data structures and dumb code works a lot better than the other way around." In the spirit of this, we leveraged some of the more flexible collection types in Guava (such as the Multimap[1]), as well as the improved I/O in order to better model and persist our data.

More details into our implementation are provided in section IV.B below.

## IV. Empirical Study

Given that quality metrics have been shown as a viable assessment of the defects found in a software system, we wish to determine if code volatility is correlated with a set of chosen class and system quality metrics. Furthermore, after determining a set of association rules for four versions of Apache Ant, we wish to determine how well these rules are able to predict changes in the fifth version.

### A. Examined hypotheses

**Code volatility hypothesis**: Classes which are deemed HV exhibit lower quality metrics compared to classes deemed LV. (Null hypothesis: Classes which are deemed HV have the same or better quality metrics compared to classes deemed LV).

**System volatility hypothesis:** The set of classes which constitute the HV bucket will have lower system level quality metrics as compared to the set of classes which constitute the MV and LV buckets. (Null hypothesis: the HV bucket will have the same or superior system level quality metrics compared to the MV and LV buckets).

[1] Storing a mapping of a given package to all of its classes in the implementation of MHF / AHF was one of the uses of Guava

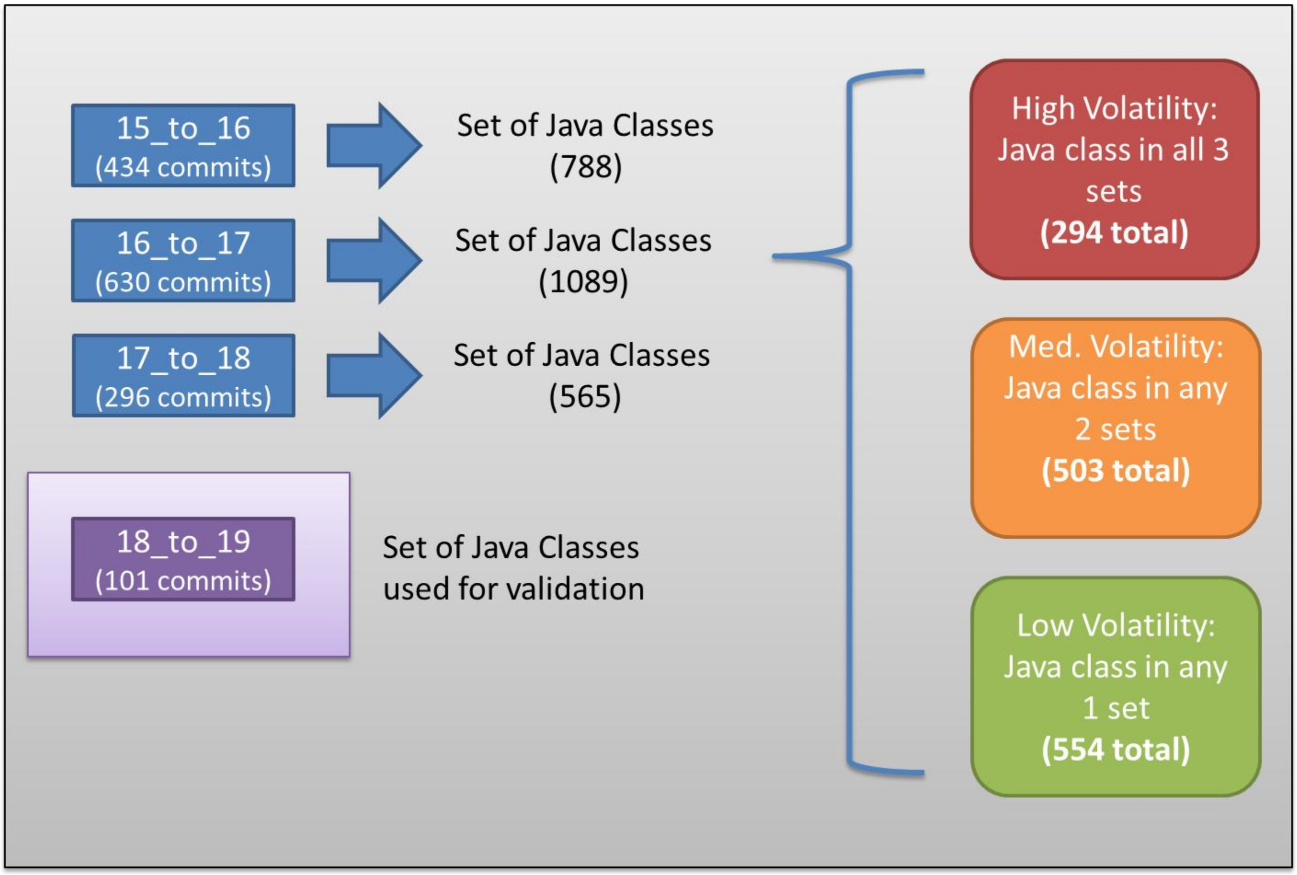


Figure 2. Distribution of Classes Into Volatility Buckets

**Association rule hypothesis**: Based on association rules and historical data, we can predict which classes may also change in the current/future version. (Null Hypothesis: association rules determined from historical data do not predict the classes which change in the current/future version).

### B. *Experiment design and Data Collection*

Being an OSS project, the SVN commit history for Apache Ant was readily available. Apache Ant is suitable for this analysis because it is widely used in the industry, is sufficiently large in terms of code and contributors, and the historical data is well formatted. The commit log contains 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 table II.

TABLE II. APACHE ANT RELEASES

| Version # | Release Date |
|---|---|
| 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 |

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.

Figure 3. Distribution of Classes Into Volatility Buckets

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[2] 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

[2] The code is available in the team Github repository: https://github.com/tajmilur-rahman/SOEN6611_Taj/tree/master/LogParser

but were instead performed for infrastructure, legal and cleanup reasons (revision ID in parentheses):

- 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.

TABLE III. NUMBER OF COMMITS AND IMPACTED CLASSES

| Version # | Number of Commits | Set of Java Classes |
|---|---|---|
| 15_to_16 | 434 | 788 |
| 16_to_17 | 630 | 1089 |
| 17_to_18 | 296 | 565 |
| 18_to_19 | 101 | 519 |

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). The HV bucket contains the classes which got changed all three times, the MV bucket 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. Figure 2 shows the visualization of the breakdown from collections of Java classes into volatility buckets.

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 [16]. 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".

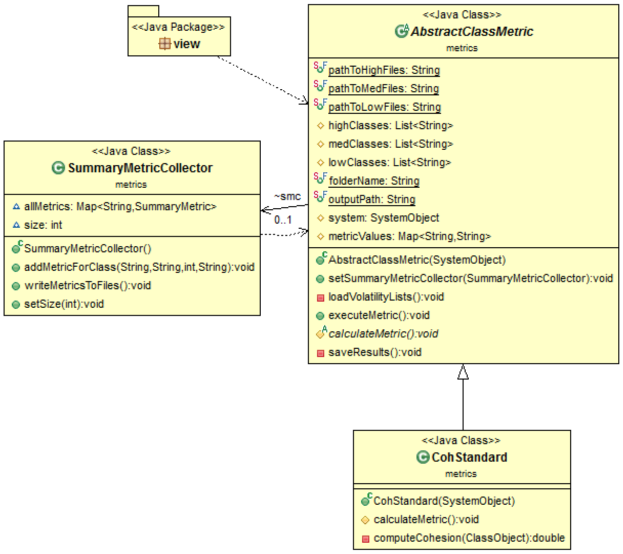
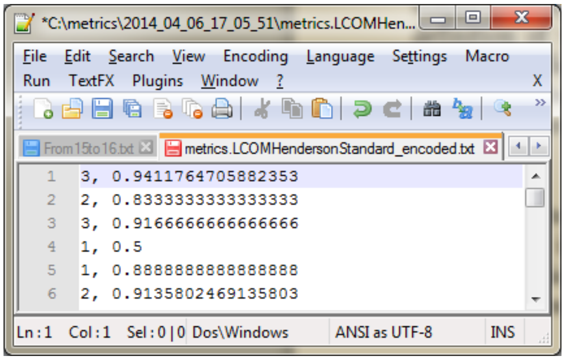


Figure 4. Sample Class Diagram for Developed Metrics

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 into separate, timestamped files. The abstract parent defines a template so that each child class would only need to implement one method (CalculateMetric), and write to the map defined in the parent. The abstract parent loads the output of LogParser. We designed the output of JDeodorant to be files containing the metric values for each class in each bucket. Here's an example output for LCOM Henderson:


*C:\metrics\2014_04_06_17_05_51\metrics.LCOMHen...
File
Edit
Search
View
Encoding
Language
Settings
Macro
Run
TextFX
Plugins
Window
?
X
From15to16.txt
metrics.LCOMHendersonStandard_encoded.txt

```
3, 0.9411764705882353
2, 0.8333333333333333
3, 0.9166666666666666
1, 0.5
1, 0.8888888888888888
2, 0.9135802469135803
```

```
Ln : 1   Col : 1   Sel : 0 | 0   Dos\Windows   ANSI as UTF-8   INS
```

Figure 5. Example Metric Output File

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 the data into R in order to visualize the results.

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 represents a single "transaction" (commit), and we only included lines where two or more Java classes were modified (as we were interested in seeing the relationship between the antecedent and the consequent). Since there were four collections of changes, we used the first three for training, and the last one for validation.

When determining the rules, we used a minimum support of 9% and minimum confidence of 50%. These values were not chosen arbitrarily – considerable effort was spent attempting to calibrate these values in order to obtain the right balance of valid rules / predictions. As the SVN log for a given collection (e.g. 15_to_16) is quite sizeable, choosing too high a support may prematurely filter out relevant rules. We calculated rules with 7 different supports ranging from 30% down to 9% and we decided that 9% gives better results than any other support. Additionally, going below 9% resulted in many rules, but this diluted the pool and many were irrelevant. Table IV below shows an example of commits between releases. The top row shows the names of classes which were modified together in a single commit. The bottom row indicates between which two releases the commit happened (e.g. classes A and B changed between versions 1 and 2 in a single commit).

TABLE IV. EXAMPLE TRANSACTIONS

| **TXs** | {A, B}<br>{D, G} | {**B**, C, **D**}<br>{C, G} | {**B**, **D**}<br>{E, K} | {A, **B**, **D**}<br>{**B**, **D**, G} | {D,C}<br>{F, H} |
|---|---|---|---|---|---|
| **Vers.** | 1-2 | 2-3 | 3-4 | 4-5 | 5-6 |

From the above table it can be determined that classes B and D are generally changing together. In terms of actionable information, this could lead us to understand that when a developer is changing class D, then he should pay much more attention to class B.

For calculating the rules we have only considered single dimension relationships: one antecedent and one consequent. The main reason for this was the processing power, memory, and time to perform the calculations when using two ore more.

Our rules are weighted based on lift: the higher the value of lift, the more important that rule will be. A lift is the measure of correlation between two entities. If the value of lift for the two entities is greater than 1 then those entities are positively correlated if the value is less than 1 then they are negatively correlated.

Finally, in order to calculate the correlations between the metrics themselves, we designed some classes to store a summary of all metrics collected. Figure 2 above shows how SummaryMetricsCollector interacts with AbstractClassMetric.

## C. *Statistical analysis*

We calculated five different metrics values for each of the classes present in the system. Each metric was charted using a series of boxplots to help visualize and verify our hypothesis. We have also calculated two system-based metrics for each of the volatility buckets.

The boxplot graphically depicts the groups of numerical data through their quartiles. It is one of the methods of comparing different data distributions. Each diagram depicts three swim lanes, each with its own plot. The first swim lane represents classes that exhibit LV, the second lane represents classes that exhibit MV and the third lane represents classes that exhibit HV (labelled 1, 2 and 3 on each diagram).

The figure below shows the boxplot for LCOM (C&K). It is clearly demonstrated that HV classes have poorer metrics as compared to MV and LV. The median value for LV is lower than the median value of MV, which in turn is lower than HV. Since LCOM represents lack of cohesion, a lower value represents a more desirable outcome.

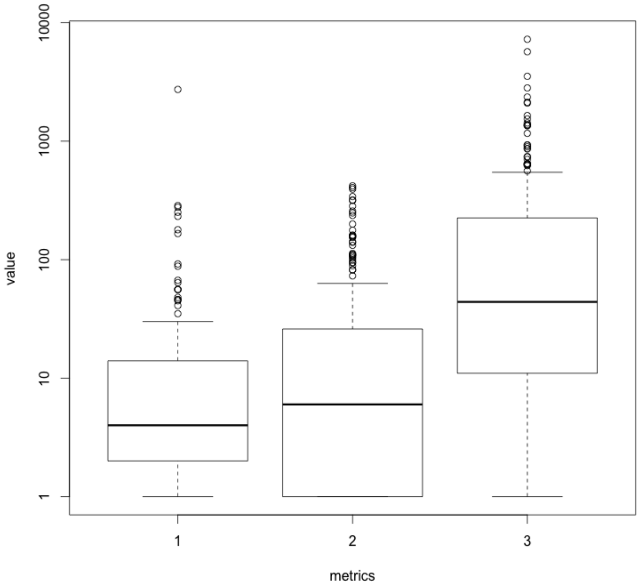
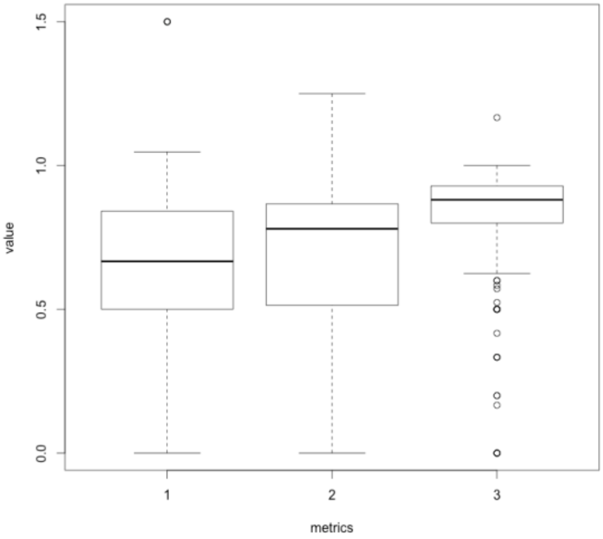


Figure 6. LCOM Boxplot

The next two boxplots display the results for LCOM (Henderson) and Cohesion (Briand). These metrics also depicted results that were expected in our hypothesis. Note that for Cohesion, a higher value is desirable, and thus the LV boxplot had the largest values



.

Figure 7. LCOM (Henderson) Boxplot

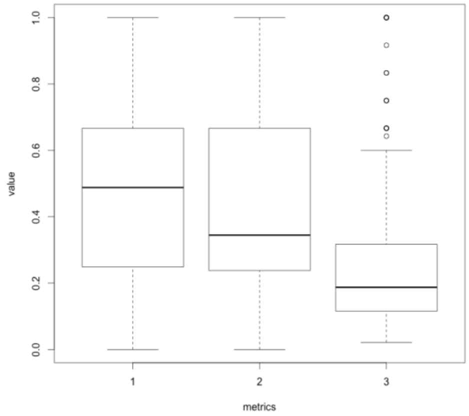


Figure 8. Cohesion (Briand) Boxplot

The last two boxplots below display the results for RFC and LOC. RFC and LOC are somehow related since they both measure the size of a class. As shown below, both these metrics increase as the volatility of the classes increases.

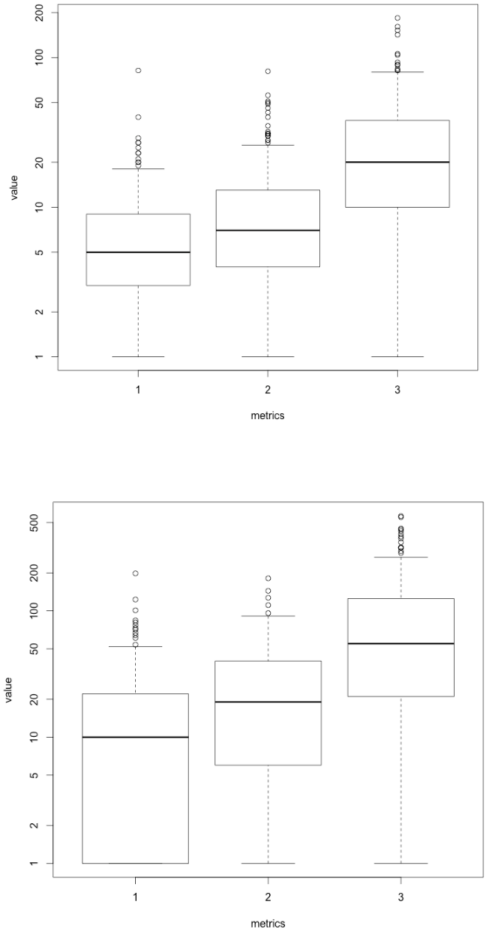


Figure 9. RFC and LOC Boxplots

The tables below show the Pearson's linear correlation between the different metrics with respect to volatility. We found that for each level of volatility, cohesion is negatively correlated to other metrics. This is expected since a higher cohesion value is desirable, whereas for the other metrics, a lower one is better. From the table (HV) it can be easily seen that LOC is strongly correlated with RFC. Correlation factor between these two metrics is 0.834, which shows that RFC and LOC change at a similar pace. We noticed some instances where the correlation became stronger as volatility increased (e.g. Loc vs Lcom), and some instance where the correlation went down then back up (e.g. RFC vs. Lcom_h). In general, the results below show that the metrics are not totally independent, and thus do cover redundant information.

TABLE V. LV CORRELATION

| | **Coh** | **Lcom** | **Lcom_h** | **Loc** | **RFC** |
|---|---|---|---|---|---|
| **Coh** | 1.00 | | | | |
| **Lcom** | -0.25 | 1.00 | | | |
| **Lcom_h** | -0.96 | 0.17 | 1.00 | | |
| **Loc** | -0.50 | 0.44 | 0.34 | 1.00 | |
| **RFC** | -0.52 | 0.83 | 0.36 | 0.71 | 1.00 |

TABLE VI. MV CORRELATION

| | **Coh** | **Lcom** | **Lcom_h** | **Loc** | **RFC** |
|---|---|---|---|---|---|
| **Coh** | 1.00 | | | | |
| **Lcom** | -0.50 | 1.00 | | | |
| **Lcom_h** | -0.97 | 0.39 | 1.00 | | |
| **Loc** | -0.51 | 0.57 | 0.40 | 1.00 | |
| **RFC** | -0.44 | 0.65 | 0.31 | 0.75 | 1.00 |

TABLE VII. HV CORRELATION

| | **Coh** | **Lcom** | **Lcom_h** | **Loc** | **RFC** |
|---|---|---|---|---|---|
| **Coh** | 1.00 | | | | |
| **Lcom** | -0.35 | 1.00 | | | |
| **Lcom_h** | -0.94 | 0.27 | 1.00 | | |
| **Loc** | -0.51 | 0.65 | 0.39 | 1.00 | |
| **RFC** | -0.56 | 0.80 | 0.42 | 0.83 | 1.00 |

To complement the class level metrics, two system level metrics were also measured. The results for the system level MOOD metrics MHF and AHF appear in the table below.

TABLE VIII. MOOD METRICS COMPARED AGAINST VOLATILITY

| **Metric** | **Volatility** | | |
|---|---|---|---|
| | ***High*** | ***Medium*** | ***Low*** |
| MHF | 0.1977 | 0.1508 | 0.1395 |
| AHF | 0.8326 | 0.7918 | 0.7760 |

It was observed that the classes in the system which constitute the HV bucket had better MHF and AHF than those in the MV and LV buckets. This appears counter to our original belief that highly volatile code will exhibit a lower MHF and AHF. Some possible explanations for this observation are that highly volatile classes tend to be bigger, and perhaps the ratio of private to public methods/attributes is also bigger. Additionally, Olague et al. showed that while CK and QMOOD metrics are good indicators of quality and fault proneness, MOOD metrics were not as useful as predictors [1].

We evaluated the association rules by using precision and recall. Our approach mimicked how Thomas Zimmermann performed his calculation. For each transaction, we split the contents into a query and an expected outcome (E). A query is a single entity of a transaction and the expected outcome for a query are the entities that got changed after that query. Table IX shows a subset of the association rules which were mined.

TABLE IX. ASSOCIATION RULES

| **LHS** | **RHS** | **Sup** | **Conf** | **Lift** |
|---|---|---|---|---|
| {/ant/.../JavaCC.java} | {/ant/.../WhichResource.java} | 0.11 | 1.00 | 98.7 |
| {/ant/.../ZipLong.java } | {/ant/.../ZipShort.java } | 0.19 | 1.00 | 98.1 |
| {/ant/.../Checksum.java} | {/ant/.../Pack.java} | 0.09 | 1.00 | 91.2 |
| {/ant/.../BlockFor.java } | {/ant/.../Funtest.java} | 0.09 | 0.81 | 79.6 |
| {/ant/.../Tokens.java} | {/ant/.../Sort.java} | 0.08 | 0.92 | 74.0 |
| {/ant/.../PlainMailer.java} | {/ant/.../MimeMailer.java} | 0.07 | 1.00 | 59.2 |

Starting with the query, we collected the association rules generated from Apriori. By taking the union of all the rules with the same antecedent, we determine how many consequents (C) are matches with the real outcome (E). This allows us to calculate the recall (R). Furthermore we counted what fraction of the rules was actually correct and this represents the precision (P).

- $R = |C \cap E|/E$
- $P = |C \cap E|/C$

The calculated precision was 10% and recall was 2%.

## V. DISCUSSSION AND THREATS TO VALIDITY

The relationship between code volatility and quality metrics is something that to our knowledge has not been explicitly investigated. While there are some similarities to code churn, there are some notable differences. First, while code churn looks at code added and deleted within a given module, volatility abstracts from that concept, and instead focuses on frequency and propensity for change. Next, volatility can be computed solely by looking at the version history of a given project, without looking at the code itself. Finally, it is simple to classify volatility according to different buckets in order to perform aggregate analysis.

The results from our analysis show that our code volatility hypothesis at the class level matches with our result – HV code exhibited lower quality metrics than LV code. On the other hand, the system level metrics did not share a similar trend, and in fact showed the opposite – HV code tended to have a superior MHF and AHF.

**Q: In the association rule mining section, why was the Apriori algorithm chosen?**

- Easy to implement and easy to understand
- Calibrating the support and count is possible
- The Apriori algorithm can be easily modified according to the needs of the analysis, for example, in giving weight to rules based on different criteria.
- While some other methods are faster, they often need lots of RAM. One example is FP-Growth. This necessity is one of the main reasons that almost all the previous researchers have chosen Apriori.

**Q: Why were some reasons to explain the low value for precision and recall?**

As we dealt with only five different releases and only three different sets of training data to calculate the association rules, the historical dataset from which to build the rules was somehow lacking depth. In order to compensate for the number of releases, we decided on a very low value for support and therefore this generated a large number of rules.

Another reason for the lower recall was the nature of how developers work. Often a single commit contained modifications to lots of Java files. Our decision to use one antecedent to one consequent was therefore not the most effective in cases where commits contained many Java files.

Our rules were generated using antecedent and consequent of ration 1:1. This may have caused us to miss some valuable information and thus affect our precision and recall.

**Q: What else we could have done to get better results?**

In order to get better precision, we could incorporate data from more releases and once again calibrate for the support and confidence. Another technique we could employ is to apply more comprehensive weighting rules.

To get better recall, instead of taking all the modifications in a single commit, we could have tried to use a different methodology. One such methodology is tf-idf, and using this allows the data to be pruned from noise.

A more involved way to improve recall is to reduce the expected outcome set by using the notion of time between changes as the criteria for splitting data into transactions. Thus for example within a single commit containing modifications to ten Java files over twenty minutes, we may use a time limit of ten minutes and therefore break this commit into two separate transactions.

**Q: What are some threats to validity?**

**Construct Validity:** We have done the association rule mining based on only 4 versions. Association rules might have variations if we would look at more versions.

**Internal Validity:** As a side effect of the low support, many of the rules that were mined were irrelevant and negatively affected our precision. Balancing both precision and recall is challenging, and when we tried to increase the support, we lost many of our rules, which in turn ruined our recall.

**External Validity:** We have made the study on a single project. Apache Ant has just over 1200 classes and approximately 227K lines of code. In order to better generalize our findings, we would need to apply a similar methodology on other OSS projects. We believe that this study can be replicated easily on other large software projects.

## VI. CONCLUSIONS

Over multiple versions of Apache Ant, it was clear that for the class-level quality metrics we chose, HV code exhibited less desirable values compared to LV code. We therefore accept the alternative hypothesis that increased volatility results in decreased quality metrics.

Furthermore, we also collected and analyzed two system-level metrics (MOOD). The values we obtained were counter to what we believed to be found – MHF and AHF did not decrease as volatility increased - and thus further investigation is necessary. We accept the null hypothesis.

Using the information on class volatility can be a powerful supplement to the existing tools and metrics available in order to help prioritize and plan maintenance and evolution activities (such as refactoring and targeted unit testing). This adds another dimension in the spectrum of code analysis and, as shown, given the SVN history for a project, can be automated using the LogParser tool we have built.

Next, using the SVN history information over five releases, we attempted to build a prediction model using association rules. We trained the model using the first four releases (3 sets of data), and tested it against the last release (the last set of data). We determined that the rules created resulted in a precision value of 10% and a recall value of 2%.

Measuring software metrics allows organizations and individuals to maintain and improve the quality of software. Analyzing the results helps us understand the meaning of the outcome, and ultimately helps guide decisions. Using volatility can thus add another dimension to that spectrum.

## REFERENCES

[1] H.M. Olague, L.H. Etzkorn, S. Gholston and S. Quattlebaum, "Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Process," *IEEE Trans. Software Eng.*, vol. 33, no. 6, pp. 402-419, June 2007

[2] S. Chidamber and C. Kemerer, “A Metrics Suite for Object-Oriented Design,” *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476-493, June 1994.

[3] B. Henderson-Sellers, L. Constantine, and I. Graham, "Coupling and Cohesion (Towards a Valid Metrics Suite for Object-Oriented Analysis and Design)," *Object-Oriented Systems*, vol. 3, no. 3, pp. 143-158, 1996.

[4] F. Brito e Abreu, "The MOOD Metrics Set," 9th European Conference on Object-Oriented Programming (ECOOP'95), Workshop on Metrics, 1995.

[5] L. Briand, J. Daly, and J. Wüst, “A Unified Framework for Cohesion Measurement in Object-Oriented Systems,” *Empirical Software Eng.: An Int’l J.*, vol. 3, no. 1, pp. 65-117, 1998.

[6] R. Agrawal, T. Imilienski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. *Proc. of the ACM SIGMOD Int’l Conf. on Management of Data*, pages 207-216, May 1993.

[7] Munson, J. C., Elbaum, S., "Code Chum: A Measure for Estimating the Impact of Code Change," *Proceedings of IEEE International Conference on Software Maintenance*, 1998, pp. 24-31.

[8] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy, “Predicting Fault Incidence Using Software Change History,” *IEEE Trans. Software Eng*., vol. 26, no. 7, July 2000.

[9] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density." in *Proceedings of the International Conference on Software Engineering (ICSE 2005)*, St. Louis, Missouri, USA, 2005, pp. 284-292.

[10] A.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll, “Predicting Source Code Changes by Mining Change History,” *IEEE Trans. Software Eng*., vol. 30, no. 9, pp. 574-586, Sept. 2004.

[11] T. Zimmermann, Weißgerber, P., Diehl, S., Zeller, A., “Mining Version Histories to Guide Software Changes”, *IEEE Transactions in Software Engineering*, 31(6), pp. 429-445, 2005.

[12] N. Schneidewind, ”Methodology for Validating Software Metrics,” IEEE Trans. Software Eng., vol. 18, no. 5, pp. 410-422, May 1992

[13] T. Ball, J. Kim, A.A. Porter, H.P. Siy, If your version control system could talk, in: Workshop on Process Modelling and Empirical Studies of Software Engineering, Boston, MA, 1997

[14] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: searching through source code using CVS comments. In ICSM 2001 [14], pages 364374.

[15] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In Proc. International Conference on Software Maintenance (ICSM 98), pages 190198, Washington D.C., USA, Nov. 1998. IEEE

[16] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. JDeodorant: Identification and removal of type-checking bad smells. In Proc. 12th CSMR, pages 329–331, Apr. 2008.