Craig Darmetko
cd2698@columbia.edu
COMS E6156 Project Report

Failing with Style:
Examining a Programmer's Error Handling Style

1 Intro

Software Provenance is the study of the origins of a software artifact, such as it's author, creation and compilation process. This information is valuable in many situations; it can help developers determine the origin of buggy or insecure code of unknown or third party origin, or detect source code plagiarism, or even aid law enforcement and intelligence officials who are attempting to determine the origin and intent of a digital attack. Unfortunately, very little provenance information is included in the distribution of most software, but there has been some promising progress to create techniques for reverse engineering this information from code or compiled artifacts after development. One promising technique is a style-metrics based approach: identifying quantifiable stylistic traits that are correlated with the developer's coding style and comparing this trait profile to profiles from samples with known authorship. These metrics ideally have a 'low within-programmer variability and high between-programmer variability' [2], which allows the technique to distinguish between authors. Similar to handwriting analysis and prose writing style, when writing code there is flexibility in the syntax that allow the developer to make decisions based on organization or ascetic habits, with little effect on the underlying functionality. It seems many developers are consistent with some of these decisions when crafting code, even across multiple applications and problem domains, which creates a unique style that can often be identified in their code [2].

One aspect of a code artifact that has been theorized to exhibit stylistic trends is how the author identifies and handle errors in their code [3,4]. Although this theory is not new, there has been little research on identifying and enumerating specific style features and evaluating their correlation with authorship. The aim of this project is to explore the area of Java Error Handling and evaluate the effectiveness of various hypothesized stylistic choices related to error handling. Understanding this topic better could allow metrics based

```
try {
        Scanner in = new Scanner(new BufferedReader(new FileRe
        PrintStream out = new PrintStream(new File("output.txt

        int testsNumber = in.nextInt();
        test:
        for (int testId = 1; testId <= testsNumber; testId++)
                int n = in.nextInt();
                int k = in.nextInt();
                String[] s = new String[n];

                 }
                 out.append("Case #" + testId + ": " + r
         }
         in.close();
         out.close();
}
catch (RuntimeException rte) {
         throw rte;
}
catch (Exception e) {
         System.err.println("Error:" + e.getMessage());
}
```

Figure 1. A condensed example of an error handling approach from the evaluation dataset. The author choses to include a large amount of code in the try block, and uses multiple catch blocks to handle different exception types.

provenance frameworks to expand and diversify their metric set, which would likely increase the accuracy and robustness of their tools.

2 System Overview

The system, as detailed in Figure 1, extracts metrics related to stylistic error handling traits and uses the information to either determine the relevance of different metrics or to evaluate a classification algorithm that uses the error handling metrics as features for determining authorship. The system takes as input a structured directory of Java projects with known ground truth and these projects are filtered down to only the Java source code files and the known authorship ground truth is inserted for training on the data. The system then spawns threads for each project and their source code files. For each file, the Parser component converts the code text file into a traversable Java object for the Feature Extraction component to examine. The Feature Extraction component next computes metrics on the source code object and updates them for each source code file in the project. After completion, the Feature Extraction computes any additional project wide metrics and reports the complete metric profile to the Controller. After all the metric profiles are completed for each input sample, the Controller next runs the classification evaluation or computes statistical relevance values for each metric that quantify its relationship with authorship.

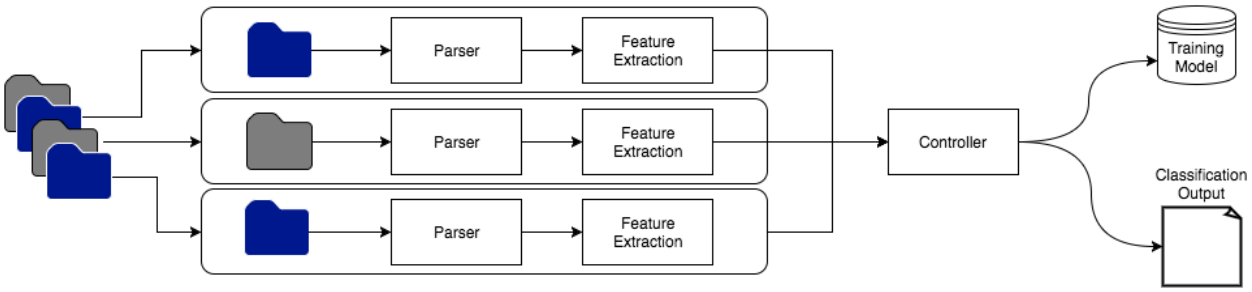

Figure 2. System Workflow Overview

2.1 Parser Component

The Parser component is responsible for parsing a Java source code file and returning a structured Java object for examining the code. Most of the heavy lifting is performed by the open source Java language parsing library JavaParser [5]. This library parses source code files or fragments and creates an Abstract Syntax Tree representation based on the code structure, javadoc and comments. From here, the library allows applications to traverse the Abstract Syntax Tree using the Visitor pattern [6] and can also modify the source code programmatically and reprint the code to file. It can iterate through specific components, such as methods, try statements or variable declarations, and also maps comments to the source code lines they likely reference.

2.2 Feature Extraction

The Feature Extraction component computes various metrics on the source code object created by the Parser Component. These metrics are related to error handling

and hypothesized to have a potential relationship with author coding style. It examines the code for multiple error and exception constructs, such as Try/Catch blocks, If blocks where the conditional is checking for 'null' and for any raising or throwing exceptions outside of these blocks.

2.2.1 Computed Metrics

The feature extractor computes 16 metrics on these constructs. These are metrics related to error handling that I hypothesize may have a correlation with authorship. The metrics fall into two categories: metrics related to how the author identifies errors and how the author handles the errors once identified. Some of the metrics related to error identification that the system computes include the ratio of Try/Catch blocks to lines of code (LoC), the average LoC in a Try block and the size of the set of unique exception classes used. Handling related metrics include the ratio of error handling LoC to the total LoC, the average number of Catches in a Try/Catch construct, the average number of comments in an error handling block and the ratio of specific error handling techniques, such as calling System.exit() to halt the program when an error is detected. A full list of metrics can be found in Figure 3.

Metric	Type	Relevance
Ratio Try/Catch constructs to LoC	Identification	Yes
Ratio of If-null constructs to LoC	Identification	No
Mean size of Try blocks	Identification	Yes
Ratio of Throw statements to LoC	Identification	Yes
# of methods that include a throwable exception in declaration	Identification	Yes
Size of the set of unique Exception classes used	Identification	Yes
Ratio of error handling code to LoC	Handling	No
Mean Catch blocks per Try/Catch construct	Handling	Yes
Mean comments per Catch block	Handling	Yes
Use of Finally blocks	Handling	No
Mean size of an error handling block	Handling	No
Ratio of handling blocks with Return statements to LoC	Handling	Yes
Ratio of handling blocks that raise exceptions to LoC	Handling	Yes
Ratio of handling blocks that print a message or stack trace to LoC	Handling	Yes
Ratio of handling blocks that immediately exit to LoC	Handling	Yes
Ratio of handling blocks that do nothing to LoC	Handling	Yes

Figure 3. A list of metrics computed by the system. Relevance will be further detailed in Section 3.

2.3 Controller Component

The controller is responsible for directing the flow of execution for computing error handling profiles on the input projects and after runs the evaluation on individual metrics or classification evaluation on the profiles when completed. This component utilizes Apache Spark, and it's MLlib machine learning library [7], for classification and training on the profiles. Apache Spark is a Java data processing engine for efficiently analyzing large datasets. This allows the system to analyze a potentially large number of input programs by efficiently parallelizing the computation using algorithms built off MapReduce[9]. The Controller component uses Linear Regression for classifying programs, due to it's ability to handle multi-dimensional data and multi-class classification and popularity in the field. The input feature vectors will be the Feature Extraction profiles and the component will support both training the classifier and classifying new input based on a trained model.

For determining the relevance of individual metrics, the system uses the One-Way ANOVA (Analysis of Variance) calculation from the Apache Commons Math library [10]. The calculation takes as input the values calculated for a metric on the dataset, grouped by author. It calculates the ratio of the within-group(author) variance to the between-group variance, which tells us how distinct the values are for each author. This approach is an adoption from the evaluation technique used in [2].

3 Evaluation

3.1 Dataset

To evaluate the system and determine metrics that correlate with software provenance, a dataset of programs was gathered with known authorship based on code samples submitted to the yearly Google Code Jam programming competition [8]. The dataset includes 25 authors, and 104 Java program samples, approximately 4 samples per author. This approach assures that there are multiple samples for most authors, as well as multiple solutions by different authors to the same problem, hopefully reducing the chance of the results being influenced by problem domain.

3.2 Evaluation Results

The system was first run to determine the metrics relevant to authorship. Metrics with an ANOVA value above 10.0 were characterized as relevant. Most of the metrics were found to have some degree of relevance, with the most relevant being the ratio of Try/Catch structures to LoC, the average number of Catch blocks per Try and the ratio of print message error handling to LoC. The presence of Finally blocks was examined and found that no author in the dataset used them, which is likely why it was not found to be related to authorship. The full breakdown can be found in Figure 3.

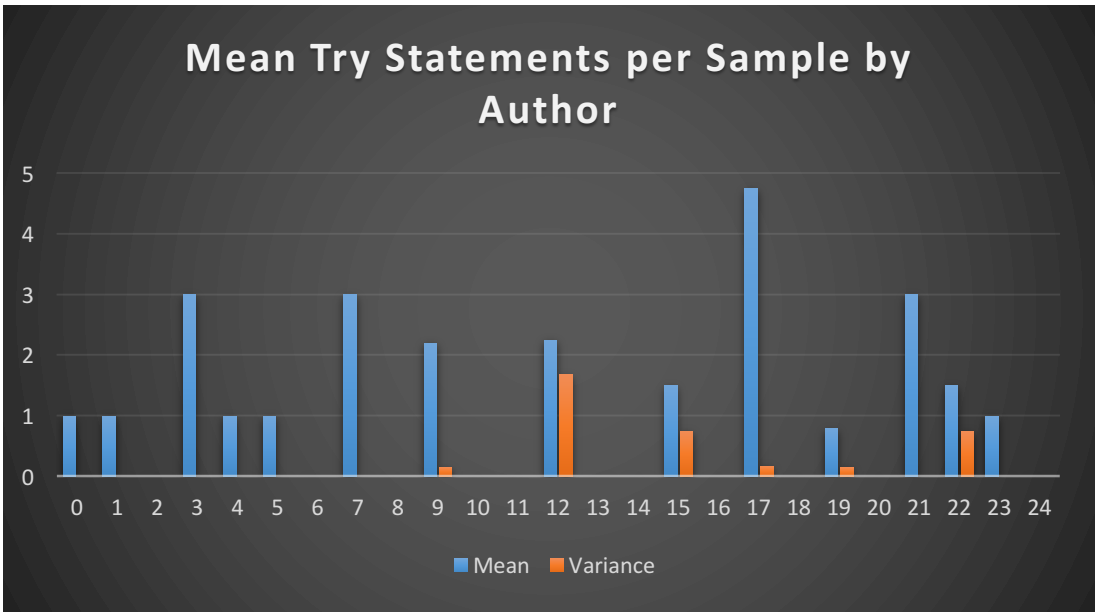

Figure 4. An examination of the Mean Try Statements metric on authors in the evaluation dataset. Note the zero or low variance for most authors.

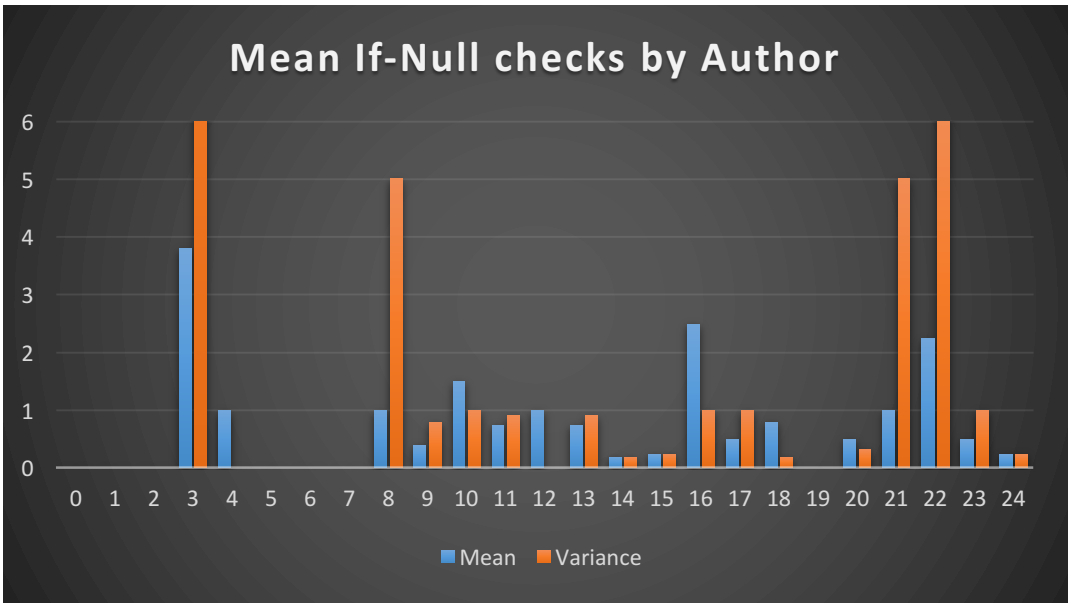

Figure 5. An examination of the Mean If-Null Statements metric on authors in the evaluation dataset. Note the much higher variance as compared to the previous metric.

The Classification algorithm was evaluated using a ten-fold cross-validation test on the Linear Regression classifier, similar to the evaluation in [11]. The test performed well, achieving 62.5% precision and 62.5% recall on the evaluation dataset. This result was better than expected, as the metrics were intended to augment an existing metrics-based profile, instead of this standalone approach. This result shows that adding the relevant metrics to existing models would likely improve results or at least increase coverage in these models.

4 Conclusion

Overall, the evidence from the evaluation dataset indicates that there is a correlation between the developer and the error handling techniques in the code. Many of the metrics were found to be relevant, and a classifier using these metrics achieved an acceptable amount of precision and recall on the dataset. Augmenting existing metrics based approaches with these new metrics would likely result in more coverage of the code artifacts and improved accuracy.

I learned a lot from this project. I had done little research into error handling, so examining and theorizing different stylistic approaches was very interesting. I learned more about the analysis of variance, including the ANOVA calculation to understand the relevance of each metric for this work. I had done very little work with Spark, and this helped further my understanding of the framework and its usefulness.

Some areas still need to be examined. A larger and more diverse dataset is needed to ensure the results are replicated in commercial code, open source code and malware, as well as to verify the classifier results were not influenced by an over-fit of the data. Gathering sizable and diverse datasets with authorship ground truth was difficult, as most large application datasets don't include authorship information due to the large documentation overhead that would be needed. Also, the impact of samples created by multiple authors was not examined, a common omission in metrics-based research. There are also likely additional metrics related to error handling that can be examined, and examining other languages besides Java would also be of interest. Finally, integrating these metrics with a general metrics-based approach would be helpful to understand the impact of these metrics on the accuracy of the system.

5 Source Code

The source code and dataset for the project can be found on my Github account at: https://github.com/sinflood/FailWithStyle

6 References

1. Gray, A., MacDonell, A.S., and Sallis, P. "Software forensics: Extending authorship analysis techniques to computer programs." (1997).
2. Ding, H., Samadzadeh, M., Extraction of Java program fingerprints for software authorship identification, Journal of Systems and Software, Volume 72, Issue 1, June 2004, Pages 49-57, ISSN 0164-1212, http://dx.doi.org/10.1016/S0164-1212(03)00049-9.
3. Frantzeskou, G., Gritzalis, S., Mac Donell, S. Source Code Authorship Analysis for supporting the cybercrime investigation process. In: Proc. 1st International Conference on e-business and Telecommunications Networks (ICETE04), vol. 2, pp. 85–92 (2004)
4. Krsul, I., and Spafford, E. H., 1996, *Authorship analysis: Identifying the author of a program*, Technical Report TR-96-052, 1996
5. “JavaParser”, http://javaparser.github.io/javaparser/
6. “Visitor Design Pattern”, https://sourcemaking.com/design_patterns/visitor
7. “Logistic Regression”, Apache Spark. http://spark.apache.org/docs/latest/mllib-linear-methods.html#logistic-regression

8. “Google Code Jam”, Google, 2016. https://code.google.com/codejam
9. Dean, J., Ghemawat, S., “MapReduce: Simplified Data Processing on Large Clusters”, OSDI, December 2004. http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf
10. “Class OneWayAnova”, Apache Commons. http://commons.apache.org/proper/commons-math/apidocs/org/apache/commons/math3/stat/inference/OneWayAnova.html
11. Rosenblum, N., Zhu, X., and Miller, B. 2011. Who wrote this code? identifying the authors of program binaries. In *Proceedings of the 16th European conference on Research in computer security* (ESORICS'11), Vijay Atluri and Claudia Diaz (Eds.). Springer-Verlag, Berlin, Heidelberg, 172-189. http://ftp.cs.wisc.edu/paradyn/papers/Rosenblum11Authorship.pdf