Which Code Construct Metrics are Symptoms of Post Release Failures?

Meiyappan Nagappan
North Carolina State University
Raleigh, NC, USA
mnagapp@ncsu.edu

Brendan Murphy
Microsoft Research
Cambridge, UK
bmurphy@microsoft.com

Mladen Vouk
North Carolina State University
Raleigh, NC, USA
vouk@ncsu.edu

ABSTRACT

Software metrics, such as code complexity metrics and code churn metrics, are used to predict failures. In this paper we study a specific set of metrics called code construct metrics and relate them to post release failures. We use the values of the code construct metrics for each file to characterize that file. We analyze the code construct metrics along with the post release failure data on the files (that splits the files into two classes: files with post release failures and files without post release failures). In our analysis we compare a file with post release failure to a set of files without post release failures, that have similar characteristics. In our comparison we identify which code construct metric, more often than the others, differs the most between these two classes of files. The goal of our research is to find out which code construct metrics can perhaps be used as symptoms of post release failures. In this paper we analyzed the code construct metrics of Eclipse 2.0, 2.1, and 3.0. Our results indicate that MethodInvocation, QualifiedName, and SimpleName, are the code constructs that differentiates the two classes of files the most and hence are the key symptoms/indicators of a file with post release failures in these versions of Eclipse.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Product Metrics*

General Terms

Algorithms, Measurement

Keywords

Code Construct Metrics, Post Release Failures, Empirical Analysis

1. INTRODUCTION

"Bad smells" are symptoms of possible problems in code that could lead to failures [3]. They indicate where to refactor in order to perform maintenance on a software system. Khomh et.al. [5] defines "bad smells" in code, as poor implementation choices. Fowler and Beck [3] define 22 different software structures as indicators of "bad smells". These smells are usually detected through metric based rules [6].

In our research we analyze code construct metrics that are derived from the Abstract Syntax Tree (AST) of the code. The value that a code construct metric has is the number of times a particular code construct appears as a node in the AST. Our research goal is to find out which code construct metrics can be used as symptoms to detect files with post release failures. Note that we do not study causation, but just identify the key symptoms/indicators of post release failures in files. For this we use an approach similar to identifying code smells. We analyze the code construct metrics for the files along with the post release failure data about the files. We define a file as having a 'post release failure' if it was edited right after the release of a software, and before work on the next release begins.

In our approach we first collect the code construct metrics for each file. Then we also find out which files have a post release failure and which ones don't. We then characterize each file with the code construct metrics. In our analysis we only compare files that have similar characteristics. By similar characteristics we mean that the files have similar values for the respective code construct metrics i.e. if we were to plot each file, with the code construct metrics as their coordinates, then files with similar characteristics would be, points in space that are close to each other. We compare each file with a post release failure, to a set of files that have no post release failures, but still have similar characteristics (similar values for most of the code construct metrics). We then identify which code construct metric more often than the others differs the most between the file a with post release failure and the files without post release failures.

Motivation : We are interested in identifying which code construct to examine first when preventively looking for post release failures. We know that large classes are a "bad smell" and hence we refactor them first. Similarly we want to identify which code constructs are symptoms for post release failures at a file level, so that they can be examined first.

Contributions : Our contributions in this research project are as follows

- To the best of our knowledge, we are the first to look at code construct metrics along with the post release failure data, to identify the code constructs that are the key symptoms/indicators of post release failures in files.
- We present a unique approach to determine the code

constructs that are the key symptoms/indicators of post release failures in files.
- We present initial results for the case study on the Eclipse metrics data set.

2. OUR APPROACH

Statistical techniques like correlation and regression are often applied on software metrics in order to describe and predict defects and failures. In statistical modeling, the first step is to identify the subset of metrics that are the best predictors. Usually a correlation technique or principal component analysis or stepwise regression are used. On the other hand a classification algorithm like Support Vector Machines (SVM), determines which metrics can predict the failures best, during the training phase. If the prediction is good then we can infer that the chosen metrics may be good indicators of future failures. We can effect a similar metric selection to identify which subset of the code construct metrics might best predict the post release failures. But the problem in using these techniques is that we are analyzing post release failures at file level. Since the difference in the number of files without post release failures and files with post release failures can be great, the prediction/classification algorithms have a high precision but poor recall rate [7], i.e. they are not able find files with post release failures accurately. Therefore we cannot use these methods to identify which code constructs might be symptomatic of post release failures.

In our approach, use a modified version of the SVM training approach. In the supervised learning step of SVM, we provide as input a set of marked examples with values for all metric dimensions. The marking signifies which of the two possible classes each example belongs to. The SVM training algorithm builds a SVM model that determines which class a new example (with values for all the dimensions) belongs to. In a SVM model, each example in the input (in our case, a file) is a point in a multidimensional space. The points in the multidimensional space of the model are mapped so that the examples of each category (failure and non-failure) are as far as possible. Then when we examine a new example, it is simply plotted in this space and depending on which group of points it is closest to, the prediction of which class it belongs to is made.

We use the same multidimensional space view. If we are collecting M code construct metrics about each file, then our space is an M-Dimensional space. In our space, each dimension is a particular code construct metric. Then each point in the space is a particular file. The mth code construct metric of a file is the mth coordinate of the point in the M-Dimensional space. For each of these points that represent a file with no post release failure, we determine the S closest points that represent files with no post release failures. This is where we differ from the SVM training algorithm. We don't consider all the points at the same time. Then we compare the metric value of the file with post release failure with each of the S closest files with no post release failures. We determine which metric differs the most and sort them in descending order of difference value to determine which metrics can help in differentiating a file with post release failure from a file without a post release failure.

Thus we examine each file with a post release failure and compare it only against the files with no post release failures that are the closest. We do this because we want to compare the metrics of similar files. Files that are close to each other in the M-dimensional space are files with similar coordinate values in their respective dimensions (in this case metric values). Thus we hypothesize that the metrics that differs the most are the likely symptoms of the post release failures since the other metrics are similar in value between the two classes of files. Since the absolute values of some metrics are much greater than the absolute values of others, this can influence the results when we are calculating difference in their values. Therefore we first need to normalize the metrics with respect to each file. Below is a formal description of how we determine and rank the code constructs that would indicate a post release failure.

1. Collect the M code construct metrics about the N files we are examining, and normalize them with respect to each file. Normalized code construct metric = (Actual code construct metric value for a file)/(Sum of all the code construct metric values for that file). Now plot the normalized values in an M-Dimensional space.

2. We mark each point as a file with post release failure or a file without post release failures. If we have N files, and k of them have post release failures, then let $l = N - k$ be the number of files without post release failures.

3. For each file k_i, of the k files with post release failures, do:

   (a) Calculate the distance from k_i to each of the l files with no post release failures. Distance between k_i, the i^{th} file with post release failures and l_j, j^{th} file with no post release failure is, $d_{ij} = \sqrt{(m_{j1} - m_{i1})^2 + ... + (m_{jM} - m_{iM})^2}$ where m_{ix} is the x^{th} code construct metric of the i^{th} file with post release failures, and m_{jx} is the x^{th} code construct metric of the j^{th} file with no post release failures.

   (b) Determine the S closest points of the set of points l, that represent files with no post release failures to the point k_i, that represents the i^{th} file with post release failures.

   (c) Calculate the difference in each of the M code construct metric values between each of the S files and the file k_i. This is the distance between each of the S points and the point k_i along each of the M-dimensions.

   (d) Determine the top R metric differences (distance along the corresponding axis), for each (file with no post release failure, file with post release failure) pair. Note that we calculate this only for the S closest files to the point k_i.

4. Thus the top R code construct metric differences for each of the k files with post release failures and its S neighbors are calculated.

5. Now count how many times each metric is in the ranks $1..R$

6. For each of the ranks, R_i , with $1 \leq i \leq R$, we determine the code construct metrics with the top 5 frequencies.

7. We then conjecture that these metrics are the most likely causes of the post release failures as more often than others they are the metrics that differ the most between files with post release failures and files with no post release failures.

3. RESULTS AND DISCUSSION

We analyzed the code construct metrics of Eclipse 2.0, 2.1, and 3.0 collected by Zimmermann et.al. [7]. In their dataset they had collected a total of 200 metrics about each file in the 3 versions of Eclipse IDE. The first 2 of those metrics were how many pre-release and post release failures were there for each file. They calculated post release by identifying if the file was edited right after the release before, work for the next release began. The assumption is that in this time window, any changes made to the code is to fix post release failure. Thus they can keep track of how many post release failures were fixed in each file. We just need the information if a file had a post release failure or not. The next 31 metrics were code complexity metrics. We did not analyze them. The next 82 metrics collected about each file is the code construct metrics. This was collected by mining the Abstract Syntax Tree (AST) of the code. Each construct is a node in the AST. Thus they increment the count of a specific code construct if the node is present in the AST of that file. The next metric has the total number of nodes in the AST of a file. The next and last 82 metrics are the normalized values of the 82 code construct metrics. The normalization was done with the total number of nodes in the AST of file as a reference.

We then applied our approach to the 3 Eclipse datasets. The values of the parameters are $M = 82$ (since there were 82 code construct metrics), $S = 12$ (we examined 12 files with no post release failures in the neighborhood of each file with a post release failure), and $R = 10$ (we extracted the top 10 metrics of each file with post release failures that were the most different when compared to the each of the S files with no post release failures in he neighborhood). The higher the value of S, the more files in the neighborhood we would compare. In our case there was no difference in the results when S was 12 or more. Hence we just report the results for $S = 12$. The total number of files in Eclipse versions 2.0, 2.1, and 3.0 are 6729, 7888, 10593 respectively. The number of files with post release failure in Eclipse versions 2.0, 2.1, and 3.0 are 975, 854, 1568 respectively. We also record the frequency of each metric in each of the R ranks and sort them in descending order.

In Table. 1, Table. 2, Table. 3, we present the results for Eclipse version 2.0, 2.1, and 3.0 respectively. Each table has 3 sets of results. One each for ranks 1, 2, and 3. By Rank 1, we mean that those metrics had the greatest difference when files with post release failures and the files without post release failures in the neighborhood were compared. Rank 2 are the metrics with the second most distance and so on. We report the metrics that have the top 5 frequencies descending order for each rank. We collect the frequencies of each metric in each rank. If we were to sum up the frequencies in each rank, the total would be equal to the product of the number of files with post release failures and the number of files without post release failures we examine in the neighborhood (S). In our case as we said earlier S was 12. Although we collect the information about the top 10 ranks ($R = 10$), we report the results only for the top 3 ranks due to space constraints in the paper.

From Table. 1 we can see that NORM_MethodInvocation was in rank 1, 1021 times. It was also the highest frequency in ranks 2 and 3 too. This is the code construct that involves a function call. From Table. 2 we can see that NORM_QualifiedName, and NORM_SimpleName are alternatingly the highest and second highest frequency metrics in ranks 1 and 2 respectively. In Table. 3 we can see similar results for Eclipse 3.0.

We also looked at what metrics were never in the top 10 ranks at all in order to find the code constructs that were least likely to be the symptoms a post release failure. We wanted to do this to see if these were indeed, code constructs which we don't expect to be symptoms of a failure. Some of these code constructs are: *NORM_LineComment*, *NORM_BlockComment*, *NORM_TagElement*, *NORM_TextElement*, *NORM_EnumDeclaration*, *NORM_NormalAnnotation*, *NORM_MarkerAnnotation*, and *NORM_SingleMemberAnnotation*. As we can see comments, Enums and annotations are all the least likely code constructs to be symptoms of post release failures. This is something we expect from these code constructs.

The actual explanation of what each of the code construct metrics (both the ones that are most likely and least likely to cause post release failures) in the three tables and the above discussion, stand for can be determined from the documentation for Eclipse AST.

4. RELATED WORK

There has been extensive research in finding out the exact reason for post release failure. These techniques include but are not limited to debugging using run time monitoring [1], statistical debugging via path profiling [2], or stack traces [4]. These are but a small subset of the techniques to determine the root cause of failures.

Fowler and Beck in the chapter on 'Code Smells' [3], describe the different code structures that are indicative of bad smells in code. A bad smell in the code does not require that it be refactored, but just that someone should take a look at that particular code structure. On similar lines we want to identify which code constructs should software engineers take an extra look at and use with caution. To the best of our knowledge, we are the first to look at code construct metrics and post release failure data to find which code construct will be a more likely symptom to files that have post release failure.

5. CONCLUSIONS AND FUTURE WORK

Metrics are often used to predict defects and post release failures in software. In our research we use the information about which files had post release failures and which files didn't have post release failures along with code construct metrics to determine which code constructs were symptoms to files with post release failures. Since the number of files with post release failures was much smaller than the number of files that did not have post release failures we couldn't use normal statistical techniques such as correlation or regression. We proposed a new approach which is a modification of the SVM training algorithm. We compare files that are similar by examining files with post release failures with the files with no post release failures in the neighborhood only.

Table 1: The Top 5 Highest Frequency Metrics and their Frequency in Rank 1, 2, and 3 for Eclipse 2.0

Rank 1		Rank 2		Rank 3	
Metric Name	Freq	Metric Name	Freq	Metric Name	Freq
NORM_MethodInvocation	1201	NORM_MethodInvocation	745	NORM_MethodInvocation	539
NORM_QualifiedName	1150	NORM_QualifiedName	678	NORM_ReturnStatement	532
NORM_SimpleName	1028	NORM_SimpleName	658	NORM_QualifiedName	520
NORM_ArrayType	675	NORM_InfixExpression	621	NORM_InfixExpression	513
NORM_Modifier	572	NORM_PrimitiveType	593	NORM_PrimitiveType	511

Table 2: The Top 5 Highest Frequency Metrics and their Frequency in Rank 1, 2, and 3 for Eclipse 2.1

Rank 1		Rank 2		Rank 3	
Metric Name	Freq	Metric Name	Freq	Metric Name	Freq
NORM_QualifiedName	1216	NORM_SimpleName	849	NORM_SimpleName	506
NORM_SimpleName	943	NORM_QualifiedName	667	NORM_ExpressionStatement	491
NORM_MethodInvocation	933	NORM_MethodInvocation	596	NORM_MethodInvocation	489
NORM_Modifier	653	NORM_SimpleType	516	NORM_SimpleType	467
NORM_InfixExpression	572	NORM_InfixExpression	501	NORM_InfixExpression	451

Table 3: The Top 5 Highest Frequency Metrics and their Frequency in Rank 1, 2, and 3 for Eclipse 3.0

Rank 1		Rank 2		Rank 3	
Metric Name	Freq	Metric Name	Freq	Metric Name	Freq
NORM_QualifiedName	1737	NORM_SimpleName	1159	NORM_MethodInvocation	881
NORM_MethodInvocation	1631	NORM_MethodInvocation	1125	NORM_Block	875
NORM_SimpleName	1558	NORM_QualifiedName	1005	NORM_Modifier	846
NORM_Modifier	1161	NORM_Modifier	931	NORM_Javadoc	831
NORM_Javadoc	1000	NORM_Javadoc	926	NORM_QualifiedName	830

We determine which metric most differentiates these two files in a small neighborhood. Such an analysis can provide developers and testers vital information on which code construct to inspect further. We did a study on the code construct metrics collected about Eclipse 2.0, 2.1, and 3.0 to determine the code constructs that were the key symptoms of the files with post release failures in these versions of Eclipse. We determined that *NORM_MethodInvocation*, *NORM_QualifiedName*, and *NORM_SimpleName* are the three constructs that are most different between the two classes of files. Hence they are the most identifiable symptom of a file with a post release failure. These results are not meant to discourage developers from using these code constructs but rather to take due precaution when using those code constructs.

As part of future work we want to inspect the source code repository for these versions of Eclipse to determine where the changes were made to fix post release failures. This will allow us to verify if the bug fixes were indeed in the code constructs that we identified as symptoms of files with post release failures. We also intend to subdivide these code construct metrics into groups that are similar and perform similar analysis in each sub group. For example we could group all the loop constructs to determine which among them is the key symptom in a file with a post release failure.

6. ACKNOWLEDGMENTS

We would like to thank Dr. Aditya Nori, and Dr. Thomas Zimmermann of Microsoft Research for their valuable inputs in this project. This research was also funded in part by the DOE grants DE-FC02-ER25809, and DE-AC02-05CH11231.

7. REFERENCES

[1] D.C. Arnold, D.H. Ahn, B.R. de Supinski, G.L. Lee, B.P. Miller, M. Schulz, "Stack Trace Analysis for Large Scale Debugging," Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International , vol., no., pp.1-10, 26-30 March 2007

[2] T.M. Chilimbi, B. Liblit, K. Mehra, A.V. Nori, K. Vaswani, "HOLMES: Effective statistical debugging via efficient path profiling," Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on , vol., no., pp.34-44, 16-24 May 2009

[3] M. Fowler, "Refactoring: Improving the Design of Existing Code," 1st ed. Addison-Wesley, June 1999.

[4] S. Hangal, M.S. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," 24th International Conference on Software Engineering, 2002, pp. 291-300

[5] F. Khomh, M.D. Penta, and Y.G. Gueheneuc, "An Exploratory Study of the Impact of Code Smells on Software Change-proneness," 16th Working Conference on Reverse Engineering, 2009, pp. 75-84.

[6] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in Proceedings of the 20th International Conference on Software Maintenance, 2004, pp. 350-359.

[7] T. Zimmermann, R. Premraj, A. Zeller, "Predicting Defects for Eclipse," Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007), pp.9-16, 2007.