Iterative Identification of Fault-Prone Binaries Using In-Process Metrics

Lucas Layman
North Carolina State University
Campus Box 8206
Raleigh, NC 27695
lucas.layman@ncsu.edu

Gunnar Kudrjavets, Nachiappan Nagappan
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
{gunnarku, nachin}@microsoft.com

ABSTRACT
Code churn, the amount of code change taking place within a software unit over time, has been correlated with fault-proneness in software systems. We investigate the use of code churn and static metrics collected at regular time intervals during the development cycle to predict faults in an iterative, in-process manner. We collected 159 churn and structure metrics from six, four-month snapshots of a 1 million LOC Microsoft product. The number of software faults fixed during each period is recorded per binary module. Using stepwise logistic regression, we create a prediction model to identify fault-prone binaries using three parameters: code churn (the number of new and changed blocks); class Fan In and class Fan Out (normalized by lines of code). The iteratively-built model is 80.0% accurate at predicting fault-prone and non-fault-prone binaries. These fault-prediction models have the advantage of allowing the engineers to observe how their fault-prediction profile evolves over time.

Categories and Subject Descriptors
D.2.8 **[Software Engineering]**: Metrics – *product metrics, process metrics.*

General Terms
Measurement, Reliability

Keywords
Code churn, fault prediction, regression, statistical models

1. INTRODUCTION
Code churn metrics, which capture the amount of code change taking place within a software unit over time, have been statistically correlated with fault-proneness in software systems. Prior research [10] has suggested that relative code churn metrics (ratios of code churn to module size) are correlated with fault density in Microsoft Windows XP and Server 2003, and that these metrics can be used in discriminant analysis to identify fault-prone binaries. We replicate and expand on this prior work using data collected from a medium-sized (1 million LoC) product at Microsoft Corporation. Fault information and software metrics were collected from six code snapshots that span two years of development.

The purpose of this research is twofold. First, our goal is *to demonstrate the feasibility and utility of creating an iteratively-built fault-prediction to make the model more reliable and less subject to process-related outliers in the metrics*. The model is "iterative" in the sense that new metrics data are taken during multiple phases of development and are used to refine the model periodically. Such a model can be used during the code development and testing phases to help guide risk assessment by identifying binaries that may require additional testing or further review. Also, an iteratively-built model is, in some respects, more *process aware* and tolerant of variations and trends in the metrics data that arise during different phases of the software lifecycle. For example, the faults and metrics collected during a stabilization/bug-fix phase of the software lifecycle may not be representative of the process as a whole and only account for a small development window (see Figure 1 for an illustration).

Our second goal is to expand prior research by examining fault-proneness with respect to measures of *software structure*, complexity and coupling in addition to code churn. Past studies [12] [13] [7] [2-4] have found correlations between fault density and software metrics such as lines of code, function points, cyclomatic complexity, and object-oriented metrics. Supplementing a fault-prediction model based on code churn with traditional software metrics may provide additional discriminatory power

The remainder of this paper is organized as follows: we provide related work in Section 2, provide information on the product in our study in Section 3 and discuss our research method in Section 4. We describe statistical analyses of fault-prediction data in Sections 5-7, and evaluate the model on iterative data in Section 8. We discuss limitations in section 9 and conclude in section 10.

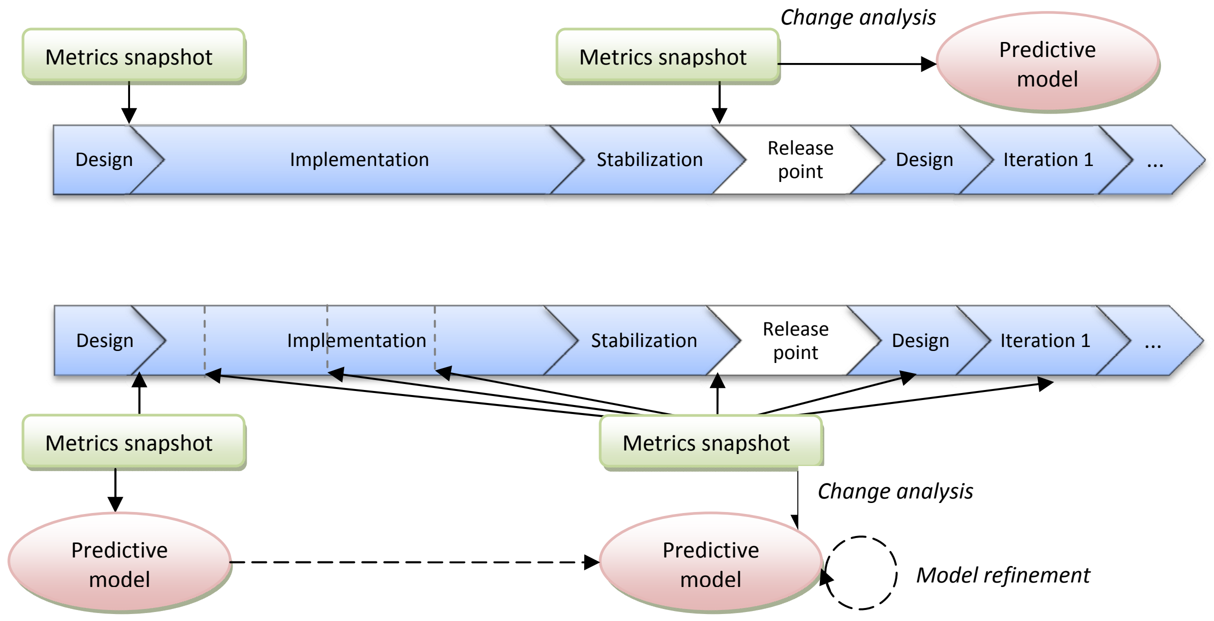

Figure 1. Single snapshot model (top) vs. iteratively-built model (bottom)

2. RELATED WORK ON FAULT-PREDICTION

Structural object-oriented (OO) measurements, such as those in the CK OO metric suite [5], have been used to evaluate and predict fault-proneness [2-4]. These metrics can be a useful early internal indicator of externally-visible product quality [2, 14, 15]. The CK metric suite consist of six metrics: weighted methods per class (WMC), coupling between objects (CBO), depth of inheritance (DIT), number of children (NOC), response for a class (RFC) and lack of cohesion among methods (LCOM).

Vouk and Tai [16] showed that in-process metrics have strong correlation with field quality of industrial software products. They demonstrated the use of software metric estimators, such as the number of failures, failure intensity (indicated by failures per test case), and drivers such as change level, component usage, and effort in order to identify fault-prone and failure-prone components. Basili et al. [2] studied the fault-proneness in software programs using eight student projects. They observed that the WMC, CBO, DIT, NOC and RFC were correlated with defects while the LCOM was not correlated with defects. Tang et al. [15] studied three real time systems for testing and maintenance defects. Higher WMC and RFC were found to be associated with fault-proneness. El Emam et al. [6] studied the effect of class size on fault-proneness by using a large telecommunications application. Class size was found to confound the effect of all the metrics on fault-proneness.

Mockus et al. [9] predicted with significant accuracy the customer perceived quality using logistic regression for a commercial telecommunications system by utilizing external factors like hardware configurations, software platforms, amount of usage and deployment issues. Finally Ostrand et al. [13] use information of file status such as new, changed, unchanged files along with other explanatory variables such as lines of code, age, prior faults etc. as predictors to predict the number of faults in a multiple release software system.

While this paper shares similar goals with the above work, our case study uses code snapshots from the same system at different intervals to built a fault-prediction model iteratively. Hence, we use the model to predict fault-proneness of binaries in subsequent periods rather than longer time periods as in the related work. We also draw on these prior research results as motivation for supplementing code churn metrics with structure metrics.

3. PRODUCT AND TEAM INFORMATION

The product under study is a commercial platform that performs a specialized server role and interfaces with numerous types of client devices – more specific information is withheld to preserve confidentiality. The entire platform is approximately 1 million LoC and is written primarily in C# with some C/C++. The product team is composed of 50+ personnel and 2 managers. The main product team is co-located, as are the many external teams that supply dependent code to the group. During the time period under study, most of the core functionality was revised, and many significant features were removed, rewritten, or introduced.

The product team is organized into feature crews that are responsible for one or more high-level features. A feature crew is typically comprised of a developer, a tester, and program manager. The developer writes the code, the tester writes and executes automated and manual functional tests, and the program manager is responsible for clarifying requirements and coordinating the feature crew with the rest of the product team. Testers work closely with developers throughout implementation, and and the ratio of testers to developers was approximately 1:1.

The product team uses several defect prevention and removal practices. Design documents are created and reviewed for each feature and any other major piece of functionality. Every piece of code that gets checked in is reviewed by peer developers, mainly using email threads and distribution lists. The team also uses Microsoft and third-party static analysis tools and others to check examine for potential defects. Additionally, the team has strong unit testing requirements. Unit testing considerations are required in the design document, and the developers are required to write comprehensive unit test suites and attain a high level of coverage. A dedicated test team also runs performance and stress tests. The entire test group combines to perform security testing as well.

4. RESEARCH METHOD

This section outlines the data collection methods, data processing, and statistical analysis methods used in this study.

4.1. Data Collection

The data used in this study were derived from code snapshots taken at four-month intervals over a two year timeframe beginning in April 2005. The timeframe encompassed most of the development of the Version 2 of the product. The four-month intervals corresponded roughly with product development milestones. At each milestone/four-month interval, the team would integrate separately developed components into the overarching product. Metrics collected at these milestones thus offered the most comprehensive view of the current state of the system.

Code snapshots were collected from the source code repository from the end of each four month period. All metrics were collected on a *per-binary* level. We used 15 binaries in our analysis, and each binary is compiled from one or more source files. This subset of binaries was picked for two reasons: 1) they contain the core code of the product; and 2) their history goes back for two or more years. The binaries in our analysis were comprised of approximately 160 KLoC in total.

The binaries from each code snapshot were analyzed using Microsoft's Tempest tool. Tempest calculates code churn metrics and uses another Microsoft tool, CodeMetrics, to calculate structure metrics such as cyclomatic complexity, coupling, counts of inter-procedural dependencies such as Fan-in and Fan-out, depth of inheritance trees, and many more.

The number of faults in the system was also counted on the binary level. In our data sets, the faults are called "hits" since some of the faults may not be actual failures, but rather users' requests for changes, such as changing a font color in a dialogue box. We use this looser definition of fault because the model will be used for risk analysis and all of these "hits" represent rework, requiring time and cost to correct. The faults recorded in our data sets may have surfaced at any point during the software development lifecycle, including after release. A distinction between pre-release and post-release faults was not available in our data set.

All hits were recorded in the team's Microsoft Product Studio project management tool as a bug. Every bug which was fixed has a version control change number associated with it. The change number was used to identify all the files that were modified during the bug fix. Bug fixes may touch multiple binaries. Thus, in our analysis, a single fault may be counted against multiple binaries.

4.2. Data Processing

All code metrics and hit counts were collected in Excel spreadsheets for the 15-binary subset in all periods. For each binary in Periods 2-6, we calculated the four code churn metrics described in Table 1. Churn metrics for Period *n* were calculated from size metrics on the current period *n* (new) and the previous period *n-1* (old). Churn metrics are calculated from code blocks. A basic block, as defined by Aho et al. [1] is "a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end."

The churn metrics in Table 1 are representative of different types of development activities. *Churn* represents new and changed functionality, *relative churn* is a ratio of change for comparison between components. *Deleted churn* suggests removal of functionality or moving functionality elsewhere, while *NCD churn* is meant to capture the proportion of churn that is due to new work (code being added or changed) to the portion that is due to removing old code (maintenance/refactoring). The churn metrics are, of course, very rough approximations of these concepts and should be interpreted with care.

Table 1. Churn metrics

Churn metrics	Definition
Churn	$New\ blocks(new) +\ Changed\ blocks(new)$
Relative churn	$\frac{New\ blocks(new) + Changed\ blocks(new)}{Total\ blocks(new)}$
Deleted churn	$\frac{Deleted\ blocks(old)}{Total\ blocks(new)}$
NCD churn	$\frac{New\ blocks(new) + Changed\ blocks(new)}{Deleted\ blocks(old)}$

We also calculated a number of transformations for every metric, including churn, structure metrics, and # of faults. We calculated the metric value *normalized* by total lines of code for that binary. We normalize the structure metrics by size because the metrics are themselves surrogates for size (with the exception of inheritance depth), and we are more concerned with the "Density" or "intensity" of the attributes these metrics represent. We normalize by LoC instead of the total number blocks of because the total number of blocks for a binary can be very large, which results in miniscule normalized ratios and statistical underflow problems. We also calculated the *delta change* for all metrics except churn, which is the value of the metric at Period *n* minus the metric value at Period *n-1*. We also calculated the absolute value of all delta metrics. Our resulting data set was composed of 159 metrics (including raw values, normalized, and deltas), and these data served as the basis for the experiments described in subsequent sections.

4.3. Classification of Fault-prone Binaries

We classify a binary as fault-prone if it contains a number of faults greater than the lower bound in the normal confidence interval calculation described in Equation 1. We are seeking a conservative classification of fault-proneness and hence use a normal confidence bound. This lower bound calculation used for fault-proneness is more conservative than the mean number of hits.

Equation 1. Lower bound for identifying fault-prone binaries

$$LB = \mu_x - \left(Z_{\alpha/2} * \frac{\sigma}{\sqrt{n}}\right)$$

LB is the lower bound on the number of faults across all binaries in all periods.

μ_x is the mean number of faults.

$Z_{\alpha/2}$ is the upper $\alpha/2$ quantile of the standard normal distribution.

σ is the standard deviation of number of faults

n is the number of observations.

5. CODE CHURN CORRELATION ANALYSIS

Spearman rank-order correlations were used to identify relationships between faults and the churn and software metrics. The data in our study almost exclusively have non-normal distributions, which require non-parametric correlation analysis. A correlation exists between code churn and the number of faults ($\rho = 0.628$, $p < 0.001$). As in prior research [10], relative code churn is positively correlated with the number of faults ($\rho = 0.357$, $p = 0.002$), though the correlation is weak.

The cross-correlations between the code churn metrics (see Table 2) differ from those in prior research. In prior research [10], relative churn, NCD churn, and deleted churn were all correlated, and NCD churn was considered a cross-check on relative churn and deleted churn. NCD churn is weakly correlated with total churn, suggesting that a majority of the overall development effort was new work. The absence of a linear association between relative churn and NCD churn is likely due to this spread in the type of work done among the binaries.

Table 2. Churn metric cross-correlation

	Churn	Relative churn	NCD churn
Relative churn			
NCD churn	0.255 $p = 0.030$*	-0.131 $p = 0.271$	
Deleted churn	0.024 $p = 0.840$	0.612 $p < 0.001$*	-0.787 $p < 0.001$*

* correlation is significant at the 0.05 level (2-tailed)

These analyses highlight an important point and support one of our earlier claims regarding fault prediction. Namely, *fault prediction models built for one product and process do not necessarily generalize outside the product context*. Prior research on code churn metrics were performed on Windows XP and Server 2003 [10, 11], which entailed the support and maintenance of a large existing code base with fewer new feature additions. The Windows XP and Server 2003 system is also significantly larger than the product in our study. Furthermore, the product in our study underwent a significant rewrite of the core functionality, including restructuring existing features as well as the implementation of many new features. The correlation analysis offers insight into these differences, and highlights the need for care when creating and applying fault-prediction models.

6. METRIC SUITE EXPLORATION

In this section, we describe how we explored and reduced our suite of 159 metrics in a systematic fashion to arrive at possible candidates for independent variables in our fault-prediction model.

6.1. Principal Component Analysis

The first method employed for exploring our large metric suite was Principal Component Analysis (PCA) [8]. PCA is a useful technique for analyzing data sets of significant size to identify interesting relationships that warrant further investigation. PCA examines cross correlations between all variables and looks for communalities, decomposing a set of data into linear components that can help identify clusters of related variables. Factors identified by PCA should always be examined further for statistical validity and conceptual viability.

We utilize PCA for two specific reasons: 1) PCA is useful for exploratory analysis of large data sets with a large number of variables; and 2) PCA factor extraction makes no assumptions on data distribution. We use PCA to find initial factors for further investigation, but not for model building or correlation analysis.

There are numerous requirements for ensuring that PCA analysis is accurate and reliable. The main complications are collinearity and sampling adequacy. We spent considerable effort (several person-days) to ensure that our analysis were reliable by examining many PCA reliability indicators, including the Kaiser-Meyer-Olkin test, anti-image matrices, residuals, and leverage values.

Our exploratory PCA analyses were performed in three batches. Our PCA analysis always included the number of faults and churn metrics. We then alternately incorporated the raw values of the software metrics, the *delta* metrics, and the *normalized* metrics. When performing this exploratory analysis, we used only the data from Periods 2, 3 and 4 since these were periods of "normal development." Our exploratory analysis yielded few findings. Much of the analysis did not pass reliability evaluations. A consistent finding among our PCA analyses was that the code churn metric (new and changed blocks) seemed to be consistently related to the number of faults.

6.2. Stepwise Logistic Regression

Logistic regression is a technique that uses a logit function to classify cases into two or more categories using a set of predictor variables. One of the strong points of logistic regression is that it makes no assumptions about the underlying data distribution. We cannot use linear regression to predict the actual number of faults in our data because of the non-normality of the predictor variables. Therefore, we use logistic regression to predict fault-proneness (a dichotomous variable) from our mostly non-normal data set. The general form of the logistic regression equation is as follows:

Equation 2. Logistic regression formula

$$P(Y) = \frac{1}{1 + e^{-X}}$$

$$\text{where } X = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

Stepwise regression techniques are a way of constructing models based on the relationship between predictor variables and the dependent variable (fault-proneness, in our case). The initial model consists of the predictor having the single largest correlation with the dependent variable. Subsequently, new predictors are selected for addition into the model based on their partial correlation with the predictors already in the model. With each new set of predictors, the model is evaluated and predictors that do not significantly contribute towards statistical significance in terms of the F-ratio are removed so that, in the end, the best set of predictors explaining the maximum possible variance is left.

As with PCA, we performed our stepwise logistic regression in three batches: predicting fault-proneness using standard software metrics (including churn), delta values, and normalized values. Again, we began by using data from Periods 2, 3 and 4 only. Stepwise regression yielded statistically-significant models comprised of raw software metrics and normalized metrics (delta metrics were never significant), but all models built had problematic reliability measures (high Cook's distance, residuals, or leverage values) that precluded them from further study. In most cases, these models were discarded because they were overly influenced by a few strong cases in the data that were causing the models to become statistically significant. In other cases, statistically significant and sound models were created, but the models had very low R^2 values (< 0.2) and poor classification plots, suggesting that these models poorly modeled the variability in the data. Models built using normalized values alone showed promise as they were statistically significant in some cases. However, an examination of the residuals and leverage values of the models built using normalized metrics alone suggested that these models were unduly influenced by a few outlier cases.

6.3. Summary of Metric Exploration

Neither PCA nor stepwise logistic regression yielded any permutation of structure metrics (raw, deltas, or normalized values) that was correlated with or predicted faults or fault-proneness. All factors produced by PCA or models produced by stepwise logistic regression that were statistically significant did not meet statistical reliability criteria.

When examining PCA factors regression models built from size metrics (such as lines of code), churn, and unmodified structure metrics, a common theme emerged. We observed that binary size appeared to be exerting some influence over all models, similar to [6]. Binary size was represented in the model either by a direct metric (e.g. lines of code), or by an indirect metric that itself is influenced by size, such as churn or total class cyclomatic complexity. Of all the size-influenced metrics, churn emerged as the most consistent, reliable, and significant predictor of fault proneness across all binaries. Consequently, we focused our-model building efforts on exploring and refining models based on code churn.

7. CODE CHURN-BASED LOGISTIC REGRESSION MODELS

7.1. Churn Alone

To replicate previous research, we attempted to build a model based on the *relative* churn metrics (Churn/KLoC, NCD churn, and Deleted churn). However, stepwise logistic regression rejected these three metrics as they did not contribute to the statistical significance of the model. We next built a model using the standard churn numbers alone (new + changed blocks). We combined the data from all binaries in periods 2, 3 and 4 as one set (representing "normal" development) and from periods 1, 5, and 6 as another set (representing ramp-up time, testing, and stabilization periods). This model correctly classified 75.6% of binaries in the Period 2+3+4 set as fault-prone or not, and 80.0% of binaries in the Period 1+5+6 set were correctly classified.

The model based on churn alone provides good prediction accuracy, correctly classifying approximately 78% of the cases overall. An examination of the model's residuals and leverage values suggest that the model is not unduly influenced by a few cases. The model based on churn alone seems best at predicting cases that *are not* fault prone, and its ability to identify cases which *are* fault prone is average at best. Despite the surface accuracy of this model, the model accounts for a relatively small amount of variance in the data (Nagelkerke $R^2 = 0.391$). Therefore, we sought to construct a model that improves on this low R^2 value that would be both more accurate and more reliable in the presence of varied data.

7.2. Augmented churn-based model

We created models with only one size-related metric (churn) and supplemented it with structure metrics (complexity, fan-in, fan-out, inheritance depth, and coupling) normalized by the size of the binary. Stepwise logistic regression using churn and normalized structure metrics yielded some statistically significant results, but churn still dominated the model.

To achieve a more discriminatory predictive model we looked at the interaction effect between churn and the structure metrics. In particular, we built regression models using the interaction of churn with fan-in, fan-out, cyclomatic complexity, class inheritance depth, and inter-class coupling. We selected these particular structure metrics because they often appeared in principal components as sharing covariance with the number of faults, but the relationships were not so strong as to produce statistically significant univariate correlations. We crossed churn by the actual values, deltas and normalized versions of these metrics. Stepwise regression of the interactions between churn and the delta metrics yielded models with only the churn term. Similarly, models of churn interacting with unmodified structure metrics also produced models with only the churn term. Thus, we discarded delta metrics and standard metrics from further analysis.

When we built models based on the interaction of churn with normalized structure metrics, we encountered interesting results. Stepwise logistic regression provided a statistically significant, reliable, and consistent model with three parameters: Churn, Churn by Normalized Fan-in, and Churn by Normalized Fan-out (see Equation 3).

Equation 3. Augmented churn-based model

$$FaultProne(Y) = \frac{1}{1+e^{-Z}}$$

$$Z = \beta_0 + \beta_1 Churn - \beta_2(Churn * NormalizedFanIn) + \beta_3(Churn * NormalizedFanOut)$$

$$Z = \beta_0 + Churn(\beta_1 - \beta_2 NormalizedFanIn + \beta_3 NormalizedFanOut)$$

$$Z = \beta_0 + \varphi Churn$$

The overall prediction accuracy similarly increases with each step (see Table 3). Most notably, we observe increases in the model's accuracy to identify fault-prone components as the new model terms are added. The iterations of the stepwise model are shown in Table 4. In the final model step, all model components are significant at the 95% confidence level with $R^2 = 0.730$. Examination of the residuals suggested that the model was not overly influenced by a few cases, and that there were very few outliers in the data that were poorly modeled.

Table 3. Model prediction accuracy - augmented churn

	Prediction accuracy	
	Periods 2,3,4	*Periods 1,5,6*
Step 1	75.6%	80.0%
Step 2	84.4%	83.3%
Step 3	86.7%	86.7%

Table 4. Augmented prediction model steps

Step	**Model components**	**Sig.**	**Nagelkerke R^2**
1	Churn	0.003	0.391
2	Churn Churn*NormalizedFanIn	0.006 0.023	0.523
3	Churn Churn*NormalizedFanIn Churn*NormalizedFanOut	0.007 0.010 0.011	0.730

Earlier, we stated that none of the normalized or raw structure metrics resulted in meaningful, reliable models on their own, and this includes Fan-in and Fan-out. It is only the interaction of churn with these terms that predictive models are accurate and reliable. We expect that multicollinearity exists among the three terms since churn appears in each of the terms. We treat the two additional terms, Normalized Fan-in and Normalized Fan-out, as adjustments to the Churn term (see Equation 3).

Our goal is to create a model for the product team to augment their risk analysis progress, and we are not trying to identify a model or a theory of why these interaction terms could or should generalize outside the team's specific context. Indeed, this task would be impossible given our analysis. However, it may be the variance captured by the structure metrics captures some other important facet of change that complements churn (i.e. considering both the size of the change and its shape). This is an avenue for future research. Our purpose in this paper is to demonstrate that such a model can be built and that, as the next section shows, the model can be constructed and iteratively refined *in-process* to identify fault-prone components.

8. MODEL EVALUATION

Once we identified model parameters that showed promise and built an accurate and robust regression equation, we simulated the creation and use of this model in-process by the development group. We used data from Period 2 only to create a regression model using Churn, Churn by Normalized Fan-in, and Churn by Normalized Fan-out and used the model to predict the fault-prone binaries in Period 3. We start with Period 2 because churn metrics were not available in Period 1. We then used a model based on the data from Periods 2 and 3 to predict fault-prone binaries in Period 4, and so forth. The results are summarized in Table 5.

Table 5. Application of iteratively built fault-proneness model

Predicted Period	**Seed periods**	**Accuracy**
3	2	73.3%
4	2, 3	73.3%
5	2, 3, 4	86.7%
6	2, 3, 4, 5	86.7%
All	2, 3, 4, 5, 6	85.3%

The overall accuracy of the iteratively-built model is 80%. Two false-positives occur in Period 6, which was a stabilization phase during development as the product was prepared for release; this period saw a significant decrease in the number of faults recorded. This may be due to the high false-negative rate, which indicates a tendency to classify fault-prone modules as non-fault-prone. However, the model still correctly classified 86.7% of the binaries as fault-prone or not during this period. The accuracy of the model could likely be improved by rebuilding the model using a more conservative lower-bound on the number of faults for classifying fault-prone binaries. Figure 2 depicts graphically the fault-proneness classification accuracy trend iteratively.

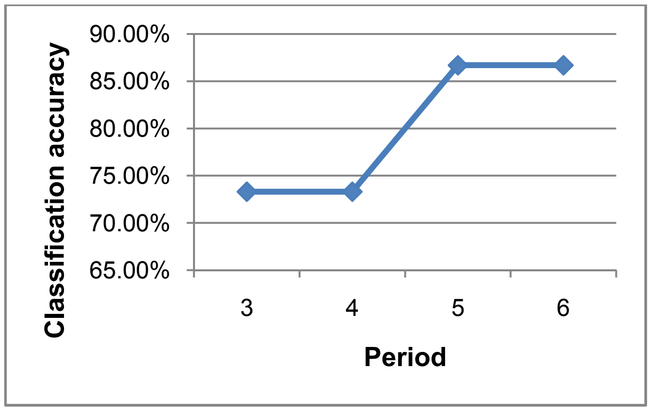

Figure 2. Fault-proneness classification accuracy

9. LIMITATIONS

The main limitation of this research regards the external validity of the fault-prediction model and raises an important point concerning fault-prediction models. The particular fault-prediction model created in this research is not necessarily transferable to other software projects. The source code information on which fault-prediction models are based contains attributes and features that directly result from the choice of development process, the individual developers, the product domain, and many other factors. Fault-prediction models are not necessarily transferable between products, teams, and processes, or may not be accurate within the same project and team if the product undergoes significant change.

An experimental limitation to this research is the manner of metric suite reduction and model building. When selecting metrics for model building, we sought to be systematic but also exploratory. We did not examine all possible combinations of the available metrics for analysis constraints (combinatorial explosion of various possible combinations). Therefore, we may have overlooked additional significant prediction models with parameters that may have run counter to our findings above.

10. CONCLUSION

In this paper, we expand on previous research using code churn and static code metrics collected at equal time intervals during development to iteratively predict fault-prone modules from a medium-sized (1 million LoC) product at Microsoft Corporation. From an initial set of 159 metrics, we use principal component analysis and stepwise logistic regression to identify code churn and its interaction with normalized fan-in and normalized fan-out as statistically significant predictors of fault-proneness. From these metrics, we *iteratively* create a fault-proneness prediction model using metric snapshots from four month intervals over a two year development period. The resulting model's average accuracy at predicting fault-prone models was 80.0%, and the accuracy of the model improved and stabilized over time. The created model will be implemented in the Microsoft Tempest tool for use by the product team to aid in risk assessment as part of their in-process development in the daily build labs. The model created in this research is not a replica of that in previous research: the model parameters are very different. One avenue of future research would be to identify the factors necessary to compare fault-prediction models and assess the applicability of existing models to new projects. This same research should also identify when the code and its generative process have changed substantially enough to warrant discarding an existing model and building a new one.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[2] V. Basili, Briand, L., Melo, W., "A Validation of Object Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering,* vol. Vol. 22, pp. 751 - 761, 1996.

[3] L. C. Briand, Wuest, J., Daly, J.W., Porter, D.V., "Exploring the Relationship between Design Measures and Software Quality in Object Oriented Systems," *Journal of Systems and Software,* vol. Vol. 51, pp. 245-273, 2000.

[4] L. C. Briand, Wuest, J., Ikonomovski, S., Lounis, H.,, "Investigating quality factors in object-oriented designs: an industrial case study," in *ICSE*, 1999, pp. 345-354.

[5] S. R. Chidamber, Kemerer, C.F., "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering,* vol. 20, pp. 476-493, 1994.

[6] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Transactions on Software Engineering,* vol. 27, pp. 630-650, July 2001.

[7] T. L. Graves, Karr, A.F., Marron, J.S., Siy, H., "Predicting Fault Incidence Using Software Change History," *IEEE Transactions on Software Engineering,* vol. 26, pp. 653-661, 2000.

[8] J. E. Jackson, *A User's Guide to Principal Components*. New York: Wiley, 1991.

[9] A. Mockus, Zhang, P., Li, P., "Drivers for customer perceived software quality," in *International Conference on Software Engineering (ICSE 05)*, St. Louis, MO, 2005, pp. 225-233.

[10] N. Nagappan, Ball, T.,, "Use of Relative Code Churn Measures to Predict System Defect Density," in *International Conference on Software Engineering (ICSE)*, St. Louis, MO, 2005, pp. 284-292.

[11] N. Nagappan, Ball, T., Murphy, B.,, "Using Historical In-Process and Product Metrics for Early Estimation of Software Failures," in *International Symposium on Software Reliability Engineering*, 2006, pp. 62-74.

[12] M. C. Ohlsson, von Mayrhauser, A., McGuire, B., Wohlin, C., "Code Decay Analysis of Legacy Software through Successive Releases," in *IEEE Aerospace Conference*, 1999, pp. 69-81.

[13] T. J. Ostrand, Weyuker, E.J, Bell, R.M., "Where the Bugs Are," in *the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2004, pp. 86-96.

[14] R. Subramanyam, Krishnan, M.S., "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Transactions on Software Engineering,* vol. Vol. 29, pp. 297 - 310, 2003.

[15] M.-H. Tang, Kao, M-H., Chen, M-H., "An empirical study on object-oriented metrics," in *Sixth International Software Metrics Symposium*, 1999, pp. 242-249.

[16] M. A. Vouk, Tai, K.C., "Multi-Phase Coverage- and Risk-Based Software Reliability Modeling," in *CASCON '93*, 1993, pp. 513-523.