2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

An empirical study of pre-release software faults in an industrial product line

Thomas R. Devine[1], Katerina Goseva-Popstajanova[1], Sandeep Krishnan[2], Robyn R. Lutz[2,3] and J. Jenny Li[4]

[1]Lane Department of Computer Science and Electrical Engineering, West Virginia University
[2]Department of Computer Science, Iowa State University
[3]Jet Propulsion Laboratory, California Institute of Technology
[4]Avaya Labs, US

***Abstract*—There is a lack of published studies providing empirical support for the assumption at the heart of product line development, namely, that through structured reuse later products will be less fault-prone. This paper presents results from an empirical study of pre-release fault and change proneness from four products in an industrial software product line. The objectives of the study are (1) to determine the association between various software metrics, as well as their correlation with the number of faults at the component level; (2) to characterize the fault and change proneness at various degrees of reuse; and (3) to determine how existing products in the software product line affect the quality of subsequently developed products and our ability to make predictions. The research results confirm, in a software product line setting, the findings of others that faults are more highly correlated to change metrics than to static code metrics. Further, the results show that variation components unique to individual products have the highest fault density and are the most prone to change. The longitudinal aspect of our research indicates that new products in this software product line benefit from the development and testing of previous products. For this case study, the number of faults in variation components of new products is predicted accurately using a linear model built on data from the previous products.**

I. Introduction

There have been several studies in the software engineering community showing the benefits of systematic reuse. These studies put to the test the intuitive claims that structured reuse will increase productivity, lower fault density, lower modification rates of modules, reduce development and maintenance effort, and reduce the complexity of the source code [6], [12], [14], [20], [21]. Nowhere is systematic reuse more prevalent than in a Software Product Line (SPL). However, there is a lack of empirical data supporting reuse benefits in SPLs.

The reuse in SPL is not ad hoc; it is deliberately and systematically planned from the inception of the SPL. Weiss and Lai define a SPL as "a family of products designed to take advantage of [their] common aspects and predicted variabilities" [22]. A commonality is a software component that is shared, or reused, among all the products in a product line. Components that are not present in all members of the SPL are called variabilities [22]. It is these differences that define the individual members of the SPL.

In this paper we present an empirical study of fault- and change-proneness trends in an industrial SPL during its development and testing phases. Specifically, we examine pre-release software faults and changes made in the code of four members of the PolyFlow product line family of software testing tools developed by Avaya [23]. To evaluate this SPL, we extracted data from the source code repository and the modification request tracking system. Based on the nature of the measures taken and their method of acquisition, we break the metrics gathered into three general categories: code metrics, change metrics, and fault metrics.

The work presented here concerns the following three main areas, each with its own research questions: (1) association of software faults with other metrics at the component level, (2) characteristics of fault and change proneness depending on the degree of reuse, and (3) longitudinal study of software faults over the period of development and testing of the SPL.

We first study the correlation between each possible pair of metrics gathered at the component level. Specific attention is given to the correlations between the number of software faults and collected static code and change metrics. In particular, we address the following set of research questions:

- RQ1: Are faults correlated with any of the gathered metrics?
- RQ2: Are any of the gathered metrics correlated to each other?
- RQ3: Does a small set of components contain the majority of faults?

Secondly, we study the fault-proneness and change-proneness of components with different degrees of reuse and thus address issues central to SPLs and their structured reuse in the form of commonalities and variabilities. Specifically, we explore the following research questions:

- RQ4: Do the number of faults and/or fault density vary in components by degree of reuse?
- RQ5: Do the number of New Features and Improvements vary in components by degree of reuse?
- RQ6: Does the change-proneness of the code vary by degree of reuse?

The research questions in areas (1) and (2) assume a cross-

978-0-7695-4670-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICST.2012.30

sectional analysis, i.e., the data was gathered at the end of the SPL's development. Our third set of research questions address data over the entire period of development and testing, that is, form a longitudinal study. These questions take into account the genesis of new products in the SPL, as well as the changes occurring in existing products. In particular, we explore the following questions:

RQ7: Do products developed later benefit from the reuse inherent in the product line?

RQ8: Can the number of faults in a new product be predicted from previously existing products' data?

The main contributions of this paper are as follows. Our results related to the association of the number of faults and other collected metrics support the findings of [11], [15], [17] that change metrics are more highly correlated to the number of faults in software components than static code metrics. We also found, in agreement with [2], [5], [7], [19], [25] that most faults are found in about 20% of the components. Furthermore, we found that complexity metrics were poor predictors of pre-release faults, which is consistent with [5]. This part of our work can be considered as a literal replication [2] (i.e., is aimed at producing similar results as the previous studies) carried out on a different type of system, within a different development context (i.e., SPLs), which has been argued to increase the external validity [2]. Unlike more mature scientific areas, such as medicine, which rely on replication, in software engineering replicated studies are often disregarded. We believe that more replicated studies need to be published to establish trends that are valid across multiple case studies, thus addressing the external validity of results.

Our research related to fault and change-proneness within different degrees of reuse is specific to product lines, an area with few empirical studies to date. The only other work on this topic seems to be our previous study [10], which was based on analysis of post-release failures of an open-source SPL. Unlike [10], in this paper we study pre-release faults and change proneness of an industrial SPL. Results show that variation components used only in individual products have the highest fault density, and are the most change prone. As in [10], common components reused in all four products had similar fault densities to the high-reuse variation components, but higher average code churn.

The last set of research questions, which are focused on longitudinal analysis of the SPL, are investigated for the first time in this paper. Our results show that newer members of the software product line benefit significantly from the development and prior testing of the more mature members. We also found that, in this SPL, the number of faults in variable components of subsequent products can be successfully predicted by a linear model developed using the metrics extracted from previously developed, more mature products. We believe that numerical prediction of the number of faults would be more useful for developing SPLs than binary classification of components into fault-prone and not fault-prone, as it would allow for more efficient allocation of testing effort and planning of release time.

The remainder of this paper is organized as follows. Related work is presented in Section II. Section III describes the SPL used as a case study. Section IV defines our metrics and discusses the process of their extraction. The main results, as they pertain to the research questions, are presented in Section V. Section VI describes the threats to validity and Section VII provides our concluding remarks.

II. Related Work

Our first set of research questions addresses the correlation of various static code and change metrics to the number of faults in a component. This type of analysis has been performed before on software products that may or may not utilize reuse. Nagappan and Ball [17] performed a correlation analysis between change metrics gathered from code of Windows Server 2003 and the associated fault database. In [2], Andersson and Runeson presented an empirical study including correlational analysis of static code metrics and fault densities for three projects from a large telecommunications company. Zimmerman et al. computed the Spearman correlations between faults and fourteen different static code metrics collected from the Eclipse project [25].

Prediction of fault-proneness is another active area of research in this field. Complexity metrics were used in a discriminant analysis in [16], which successfully created models to classify program modules into broad, fault-prone groups. Also, in [8], Taghi and Munson used regression models based on developmental complexity metrics to predict fault densities. Ostrand et al. used a negative binomial regression model to predict the number of faults at a file level from one release to the next [19]. Although the individual predictions were often inaccurate, the models could accurately identify the 20% of files containing around 80% of the total faults. In [18], Nagappan et al. used static code metrics from five diverse Microsoft products to create regression models for estimation of post-release faults. They found that using Principle Component Analysis allowed them to obtain a good set of predictor variables, but that no single set of predictors worked well across all products, that is, predictors were accurate across products only when those products were similar.

Several empirical studies have been conducted on the general benefits of systematic reuse. Frakes and Succi's analysis of four different sets of industrial data [6] indicated that more reuse results in fewer faults, higher perceptions of quality, and lower fault density. The reuse in their study was ad-hoc, black box, code reuse. In a study of twenty-five different software systems developed by NASA [20], Selby found that modules reused verbatim had on average 98% less faults, while reused modules that were modified showed 55%

less faults than non-reused modules. Seven medium-scale projects with reused components, also developed for NASA, were analyzed by Thomas et al. [21]. They determined that verbatim reuse resulted in over a 90% reduction in fault density when compared to new code, while modified reuse resulted in a 59% decrease. Lim studied two products from HP that utilized reusable code [12] and found in both cases a reduction in fault density of around four times for the reused code over the newly developed code.

Case studies of reuse in SPLs are less common, due in part to the limited availability of subjects. Bypassing this, Zhang and Jarzebek concurrently developed four members of the mobile gaming product line family by using a product line architecture and by developing each game individually [24]. In their experiment, the product line approach led to improvements in development and maintenance, as well as to an increase in the actual performance of the applications. Mohaghegi et al. performed empirical studies on three large telecom product lines [13], [14] by mining trouble report repositories. Their results indicated that reused components had a fault density of 44 to 61% of the non-reused components. Furthermore, reused components required fewer modifications than non-reused components.

In earlier work [10], we analyzed post-release fault data from four recent releases of the Eclipse project, viewed as a SPL. In that work, as in this paper, components were grouped based on the degree of reuse across the product family. The examination showed that commonalities exhibited fewer post-release faults than any other degree of reuse and followed a decreasing trend in file churn through subsequent releases. We also found that as the SPL evolved through releases, the amount of change in the variable components remained high. More recently, in [11] we performed a further study of Eclipse as a SPL. In that work we used the J48 decision tree algorithm to analyze the effectiveness of change metrics at classifying Eclipse components as fault-prone or not. The classification results were very good (probability of detection 79 to 85%, probability of false alarm 2 to 4%), with the particular subset of change metrics performing well throughout all releases of the SPL. In addition, the results showed that the learner's performance increased as the product line matured.

The work presented in this paper is set apart from this body of research by some key differences. In contrast to papers that examine reuse in general, we take a SPL as our subject. PolyFlow is an industrial SPL from design to implementation. Our work examines pre-release faults detected during development and testing. While others [25] have examined correlations between static code metrics and pre-release faults, we also consider change metrics. In several of the related works [11], [15], [17], change metrics were found to be better predictors of fault-proneness than static code metrics, which is consistent with our findings. Finally, throughout the literature most predictions are aimed at binary classification, i.e., classifying components as faulty or not. We use numerical prediction to gain more information from the data on the degree of fault-proneness of components. In particular, based on the data collected from more mature products we build and test a linear model to predict the number of pre-release faults in subsequent products. While [18] used regression models to make numerical estimations, they used only static code metrics and models created from independent, stand-alone products that did not share code as the PolyFlow SPL does.

III. Case Study Description

PolyFlow, formerly known as eXVantage, is a suite of software testing tools developed by Avaya Corporation that allows developers to generate and execute test cases and calculate associated coverage metrics [23], in addition to other tasks. Variabilities across the product line include support for various operating systems, target programming languages, and user interfaces. The entire suite was developed in Java, with a modular architecture (i.e., related classes were grouped into packages that serve as components).

The PolyFlow product line currently has eleven defined products in different stages of development. The results presented in this paper are based on the first four fully implemented products, which are also the only products that have sufficient number of entries in the Modification Request (MR) database to allow statistical analysis. Each of these four products, P_1, P_2, P_3, and P_4, is a different subset of 42 components totaling approximately 65,000 LoC. The number of components and LoC that comprise each product are shown in Table I. (The total number of components in Table I is greater than 42 due to components being used in more than one product.) Figure 1 shows the distribution of the components in a Venn diagram, to help visualize their organization by degree of reuse. In the diagram, the 13 components in the central region, which are reused in all products, are *common components*. The four regions directly adjacent to the center contain components that are common to three of the four products. We label them *high-reuse variation components*. *Low-reuse variation components* are in the regions where only two products overlap. The perimeter regions contain the *single-use variation components*, i.e., components currently used in only one product. Of these products, the members of the sets $\{P_1, P_3\}$ and $\{P_2, P_4\}$ are similar to each other in composition and function, and share many common components.

Table I. Number of components and LoC for the four products from the PolyFlow SPL examined in this paper

Product	*Components*	*LoC*
P_1	23	47,138
P_2	29	35,238
P_3	37	49,676
P_4	22	36,852

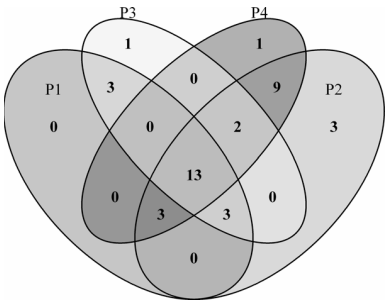

Figure 1. The distribution of components among four products

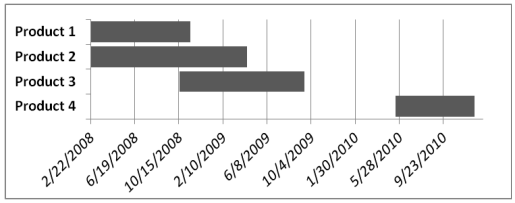

Figure 2. Timeline of products development

Figure 2 presents a Gantt chart depicting the development effort in the PolyFlow product line chronologically, with a horizontal axis interval of four months. As the chart shows, development of P_1 and P_2 began simultaneously and continued concurrently throughout the completion of the development of P_1. Development of P_3 began upon the completion of P_1, and was concurrent with the final months of P_2's development. The completion of P_3 marked the beginning of an eight month period during which other products not covered by this study were the main focus of development efforts. Following this time, development of P_4 began and covered the final span of seven months.

The MR database from which some of our metrics were mined consists of three MR types. We performed the classification in consultation with the lead software developer who had been with the project from the beginning. MRs of the type *Fixes* were made to fix software faults. A fault is defined as an accidental condition, which if encountered, may cause the system or system component to fail to perform as required [1][1]. All faults analyzed in this paper are pre-release faults detected during testing, as data from field usage had not yet been collected. *Improvements* are any requests for modifications to improve the quality, efficiency, or output of existing code. An example of an improvement is refactoring the code in a component. *New Features* represent requests for entirely new code to produce previously unimplemented functionality. These are commonly associated with the introduction of new products, but sometimes refer to new functionalities being implemented in existing products.

The data from the MR database had to be preprocessed before they could be used in our analysis. We found that individual MRs were mapped to multiple components. In particular, 10.3% of the MRs labeled Fixes were mapped to two or three components, which is consistent with our earlier results [7]. In these cases the fix was associated with two or three faults, one for each affected component. Similarly, 11.5% of Improvements and 20% of New Features resulted in changes to two or three components. Again, we replicated the Improvements/New Features that affected more than one component so that one entry exists for each affected component.

The MR database, post-processing, consists of 258 individual entries, distributed by type as shown in Table II. The third column shows how many of the overall MRs were code-related, and the last column shows how many of those code-related MRs were closed. 70 out of 258 MRs were excluded from consideration because they were either related to other products currently under development or were not related to changes in the code of products P_1, P_2, P_3, and P_4. Of the remaining 188 MRs, five Improvements and 20 New Features are still open and unresolved, which means that the additional functionality requested by these MRs has not yet been implemented in code. Since our study is directed toward the analysis of implemented code, we removed those MRs from our consideration.

Table II. DISTRIBUTION OF MRS BY TYPE

MR Type	*Overall*	*Code Related*	*Closed*
Faults	117	92	92
Improvements	52	35	30
NewFeatures	83	61	41
Other	6	0	0
Total	258	188	163

IV. METRIC DEFINITION AND EXTRACTION

We had access to several different sources of data from the PolyFlow product family. The gathered metrics are divided into three categories: source code metrics, change metrics, and fault metrics. First, we considered the following static source code metrics at the component level:

- *Lines of Code (LoC)* - the number of non-comment lines of Java code in a component.
- *Number of Files* - the number of files comprising the component.
- *Maximum Complexity* - the maximum complexity exhibited by any method in a given component.
- *Average Complexity* - the average complexity of the methods in a given component.

These metrics were gathered at the class level using the freeware code analysis tool SourceMonitor and then aggregated into component level metrics.

Change metrics were gathered from two places, the MR database and the logs in the subversion (SVN) repository in which the code was maintained. The following metrics

[1]Bug is often used as a synonym to software fault. We avoid using the term defect because in the past it has been used to refer to both software faults and failures, as well as anomalies.

quantify the amount of modification to a component during development and testing:

- *Improvements* - the number of MRs requesting changes for improvements of the code.
- *New Features* - the number of MRs requesting new code to be written to implement new features.
- *CodeChurn* - the sum of the LoC added to and deleted from a component over the course of its existence in the repository.
- *Average CodeChurn* - the CodeChurn of a component divided by the total LoC for that component.
- *FileChurn* - the number of times a component's files were added to or deleted from the repository.
- *Average FileChurn* - the FileChurn of a component divided by the number of files in that component.

The data for Improvements and New Features were collected from the MR database and mapped to the components they affected as described in Section III. The four churn metrics, which reflect changes made to fix faults and implement improvements and new features, were extracted in two stages. First, we used the freeware application StatSVN to analyze the SVN logs kept by the code repository and acquire general folder level metrics in an html format. Then, we wrote code to extract individual metrics for each file in the repository, which were subsequently aggregated to compute the churn metrics at component level.

Finally, the number of faults per component metric was computed based on the data from the MR database, as explained in Section III.

V. Case Study Observations

This Section is organized into three subsections, each addressing one set of research questions.

A. Association of software faults with other metrics

Table III shows the results of the correlation analysis between pairs of metrics. In this study we used the Spearman correlation, because the data did not conform to the normal distribution necessary to apply the Pearson correlation. The cells of the table contain the value of the Spearman correlation coefficient ρ, with the p-value below in parenthesis, for all non-trivial combinations which resulted in statistically significant results at the $\alpha = 0.05$ significance level. The blank cells in the upper triangle represent either statistically not significant correlations (e.g., Faults and Average Complexity) or high correlations by definition, which therefore are trivial (e.g., Maximum Complexity and Average Complexity). Since the number of faults per component is of special interest, Figure 3 presents the scatter plots for each metric versus the number of faults in a component.

RQ1: Are faults correlated with any of the gathered metrics?

The table clearly shows that the number of faults is positively correlated with almost every metric gathered in this study. The only change metric that showed no correlation was Average FileChurn, whereas the only static code metric not correlated to the number of faults was Average Complexity. We make the following observation based on the correlation coefficients presented in the first row in Table III: *The correlation of the number of faults at the component level with each of the change metrics except FileChurn is higher than the correlation with any static code metric.* Specifically, the strongest correlation is with New Features and CodeChurn (0.760 and 0.702, respectively), followed by the correlation with the Average CodeChurn, Improvements, and FileChurn (0.612, 0.597, and 0.435 respectively). On the other side, the highest correlation among the static code metrics is with LoC, followed by Number of Files and Maximum Complexity (with values of 0.490, 0.469, and 0.321 respectively). These results are in agreement with Nagappan and Ball, who found in [17] that Average CodeChurn had statistically significant, very high positive correlation to fault density (p-value < 0.01, $\rho = 0.883$). We are led by these results to agree with their general conclusion, which, in the terms of this study, is that an increase in change metrics in a component is often accompanied by an increase in faults in that component.

We also compared our results to the works of Andersson et al. [2] and Zimmerman et al. [25] regarding the use of size metrics as predictors for the number of faults in a component. When considering LoC versus number of faults, Andersson et al.'s results showed correlation coefficient values (they used the Pearson test, as opposed to the Spearman) of 0.37 and 0.6 in the most closely correlated projects and 0.05 in the least correlated. Zimmerman et al. calculated $\rho = 0.487$, significant at the 0.01 level. Our values for LoC versus number of faults are consistent with these results, as the components in our study showed a positive correlation $\rho = 0.490$ with a p-value $= 0.0015$. Further, we found that the number of files in a component was also positively correlated with the number of faults with $\rho = 0.469$ and p-value $= 0.0026$. This result is also supported by Zimmerman et al., who computed a ρ value of 0.406, significant at the 0.01 level, for the same relation.

In [5], Fenton and Ohlssen determined that cyclomatic complexity was not a good predictor of pre-release faults. Our results for the average cyclomatic complexity are in agreement. However, we did note a positive correlation between Maximum Complexity and number of faults. The higher correlation of Maximum Complexity to number of faults than Average Complexity was indicated in [25], with $\rho = 0.475$ for Maximum Complexity and $\rho = 0.300$ for Average Complexity, both significant at the 0.01 level. Compared to our results Zimmerman et al.'s results for Maximum Complexity show a slightly higher level of correlation, while we are unable to support the correlation between the Average Complexity and number of faults.

Table III. SPEARMAN CORRELATION ρ VALUES FOR NON-TRIVIAL ASSOCIATIONS, ACCOMPANIED BY THE P-VALUE IN PARENTHESES

	Faults	*Improvements*	*NewFeatures*	*CodeChurn*	*AvgCodeChurn*	*FileChurn*	*AvgFileChurn*	*LoC*	*NumFiles*	*MaxComplexity*	*AvgComplexity*
Faults	–	0.597 (0.0001)	0.760 (0.0000)	0.702 (0.0000)	0.612 (0.0000)	0.435 (0.0056)		0.490 (0.0015)	0.469 (0.0026)	0.321 (0.0461)	
Improvements		–	0.676 (0.0000)	0.586 (0.0001)	0.597 (0.0001)		-0.388 (0.0146)	0.418 (0.0082)	0.359 (0.0247)	0.299 (0.0645)	0.352 (0.0281)
NewFeatures			–	0.734 (0.0000)	0.674 (0.0000)	0.359 (0.0247)				0.398 (0.0122)	
CodeChurn				–				0.548 (0.0003)		0.497 (0.0013)	
AvgCodeChurn					–	0.398 (0.0121)				0.417 (0.0083)	
FileChurn						–				0.344 (0.0320)	
AvgFileChurn							–	-0.343 (0.0327)			-0.344 (0.0319)
LoC								–		0.599 (0.0001)	
NumFiles									–	0.374 (0.0192)	
Max Complexity										–	
Avg. Complexity											–

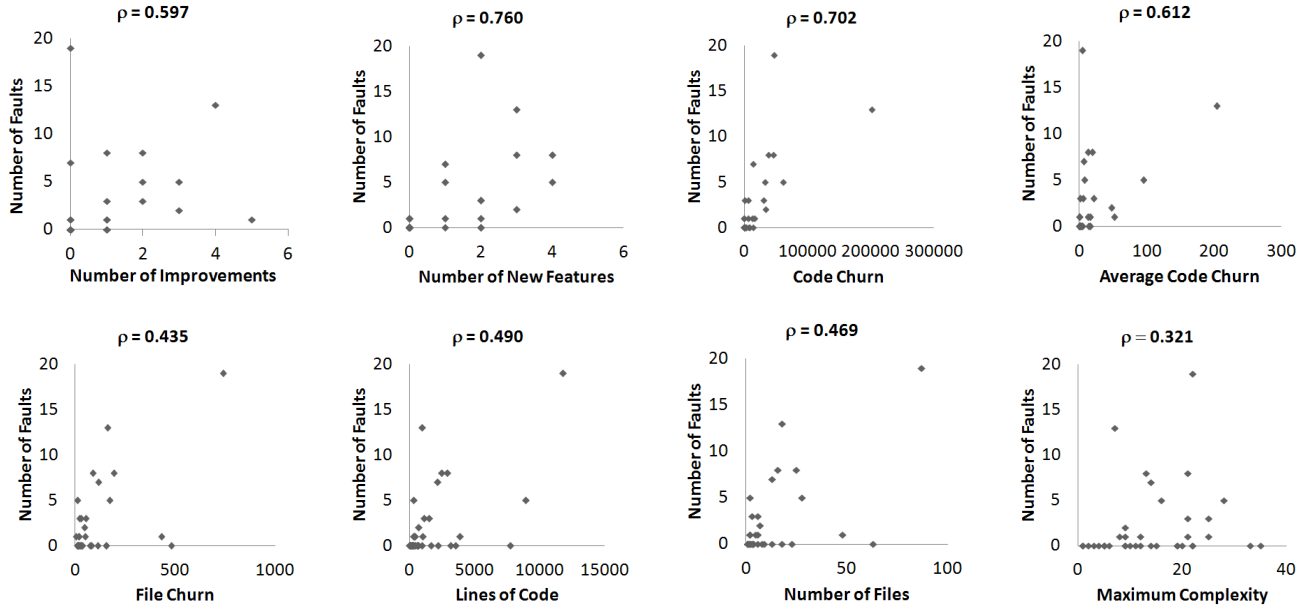

Figure 3. Scatter plots of the number of faults and each metric for which the correlation was statistically significant

RQ2: Are any of the gathered metrics correlated to each other?

Based on the results given in Table III the following observations are made. Improvements and New Features are highly correlated to each other, and both are moderately to highly correlated to CodeChurn and Average CodeChurn. This is expected, as both Improvements and New Features lead to changes in the code. Similarly, New Features are correlated to FileChurn because New Features often result in the addition of new files. Improvements, however, are not correlated with FileChurn.

Improvements have small to moderate correlations with all static code metrics, which indicates that larger and more complex components tend to undergo more Improvements. Of the static code metrics, New Features is only correlated with Maximum Complexity. Maximum Complexity is actually correlated with all metrics except Average FileChurn. However, Average Complexity is correlated with only Average FileChurn and Improvements. These observations indicate that a higher Maximum Complexity (i.e., having at least

one complex method in a component) has much more impact on change and fault metrics than the Average Complexity of all methods in that component.

RQ3: Does a small set of components contain the majority of faults?

Our results show that approximately 85% of the faults detected across all of the products are located in approximately 20% of the components. This result agrees with other works [2], [3], [5], [7], [19], which have consistently found that between 60 and 90% of faults normally reside in around 20% of the components. However, when we examined the data from a LoC perspective, we found that 80% of the faults were in 50% of the code. This still shows a skewed distribution of the number of faults across code, but with a greater spread.

B. Fault and change proneness for different degrees of reuse

This section studies the effects and benefits of the systematic code reuse in the PolyFlow SPL. To account for different degrees of reuse, we organized the component data into four groups: (1) Common component shared by all four products (2) High-reuse variation components reused in three products (3) Low-reuse variation components reused in two products and (4) Single-use variation components used in only one product. The organizational structure and accompanying data are shown in Table IV. Some of the metrics shown in Table IV are plotted on bar graphs in Figure 4 for ease of visual comparison.

RQ4: Do the number of faults and/or fault density vary in components by degree of reuse?

The groups of low-reuse variation components and single-use variation components share the fewest number of total faults (see Figure 4a). However, the low-reuse variation components have the lowest fault density after normalization by LoC (see Figure 4b). Due to the considerably smaller LoC in the single-use variation components (nearly one-third the size of the next smallest), they have the highest fault density even though they have the least number of total faults. If we consider single-use variation cox mponents as non-reused components to conform to the context of [14], then this result is in agreement with their finding that components that are reused have lower fault densities than those that are not. This may be due to the fact that single-use variation components have high Maximum Complexities. The most complex component in the product family also resides in this area. The high fault density for the single-use variation components can then, at least partially, be explained by the correlation between faults and maximum complexity ($\rho = 0.321, p < 0.05$) expressed in section V-A.

RQ5: Do the number of New Features and Improvements vary in components by degree of reuse?

Table IV. COMPONENT LEVEL DATA ORGANIZED BY DEGREE OF REUSE

	Common comp	*High-reuse variation comp*	*Low-reuse variation comp*	*Single-use variation comp*
NumComps	13	8	12	5
Faults	26	22	15	15
Faults/KLoC	1.201	1.295	0.799	2.555
Improvements	7	3	11	7
NewFeatures	5	6	12	10
CodeChurn	300,293	62,154	139,783	93,096
AvgCodeChurn	13.873	3.660	7.442	15.857
FileChurn	1229	1016	971	241
AvgFileChurn	8.361	8.397	7.301	6.025
LoC	21,646	16,984	18,783	5,871
NumFiles	147	121	133	40
MaxComplex	33	25	25	35
AvgComplex	1.761	2.278	2.550	3.492

As shown in Figure 4a, the low-reuse variation components exhibit the greatest number of New Features and Improvements. This is due to the fact that some low-reuse variation components, which are reused in two products were not originally designed to be reused. Instead, when a new product was added, it was concluded that some components could be reused, which in turn resulted in the increased number of New Feature and Improvement MRs in these components. This result agrees with our earlier finding in [10] that variation components evolve rapidly.

When values for the low-reuse variation components are combined with the values for the single-use variation components, these numbers far surpass the combined amounts contained by the high-reuse variation components and common components (1.8 times more Improvements and exactly twice the New Features). This is consistent with the fact that New Features introduce new components. Since the high-reuse variation components and common components are designed to be used in almost all products in a SPL, the introduction of new components into this area after the creation of several products is unlikely. Furthermore, the fact that the newly introduced variation components are less mature than the highly reused components may contribute to the higher number of improvements they require. Lending support to this claim, the high-reuse variation components, which exist in three of the four products, show the least number of Improvements. This could be interpreted as showing the benefit of their planned reuse.

RQ6: Does the change-proneness of the code vary by degree of reuse?

Before normalization, the common components show the highest amount of CodeChurn. However, when considering the amount of code contained in each reuse group (see Figure 4b), the single-use variation components have the most change-prone code (i.e., have the highest Average CodeChurn). This finding lends support to the results of Mohaghehi et al. in [14] that non-reused components have higher code modification rates.

Interestingly, the most reused components, i.e., the group of common components, still exhibit a relatively high Average CodeChurn. One reason for this is that this study looks at code as it is being developed. As the development

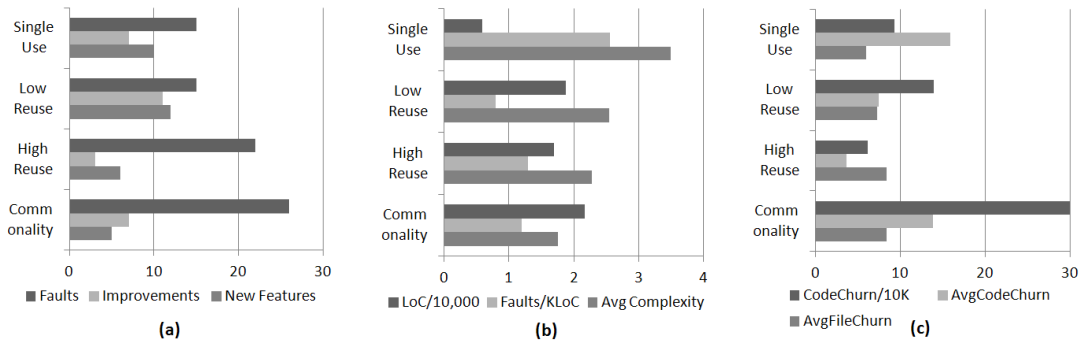

Figure 4. Comparisons of multiple metrics by degree of reuse (Units for the x-axes are given in the legends below figures.)

of the SPL progressed, and new products were introduced, common components had to be changed to accommodate unanticipated requirements from new products. Another reason for the large change proneness was the known phenomenon of evolving requirements (not always related to reuse) throughout the development of some components. In particular, 67% of the CodeChurn in the group of common components was due to a single component which had only 4.57% of the code, but was responsible for 67% of the New Features, 57% of Improvements, and 50% of the Faults. It would be interesting to explore, once the data is available, whether this particular component continues to be faulty and change-prone post release. We note that in an earlier study of post-release failures of an open source product line, common components also experienced more churn than expected [10].

C. Longitudinal study of software faults over the whole span of development and testing of the SPL

RQ7 and RQ8 investigate the evolution of the SPL through its development. As depicted in Figure 2, P_1 and P_2 were the first two products to exist, followed by subsequently developed products P_3 and P_4.

RQ7: Do products developed later benefit from the reuse inherent in the product line?

To explore this question we considered the frequency of faults occurring in each of the newer products P_3 and P_4. We distinguished between those faults that occurred in relevant components before a product's creation and those that occurred afterward, when later products reusing those components were created. We examined the MR data in this light and found that of the 37 faults that affected components in P_3, eight were found and fixed before P_3 was created. These eight faults were all located in the common components, which illustrates how P_3 benefited from the structured, planned reuse across all products. Each of the remaining 29 faults was located in a component that was shared between P_3 and another previously or concurrently developed product (i.e., 11 faults were in low-reuse variation components shared between P_1 and P_3 and 18 in high-reuse variation components shared across P_1, P_2, and P_3). That is, not a single fault was unique to P_3. There are two main reasons for these faults in low-reuse and high-reuse variation components: (1) new faults were introduced or existing faults were detected in the process of accommodating requirements due to the introduction of the new product P_3 and (2) the concurrent development of P_3 and P_2 (see Figure 2).

The benefit of reuse is more prominent in the case of P_4, which was developed after the completion of the other three products in the SPL. Of the 69 faults affecting components included in P_4, 67 were detected and fixed before P_4 existed (out of which 26 were in the common components, 21 were in the high-reuse variation components, and 20 were in the low-reuse variation components). That is, only two faults were detected during the actual development of P_4, one in a single-use variation component and another in a high-reuse variation component shared with P_1 and P_2. Since P_4 was under development for the least amount of time, and therefore had less time for testing to expose faults within the scope of our study, it is possible that future testing and field usage may expose additional faults. Clearly, however, the fact that 67 faults were fixed in code subsequently reused in P_4 shows that P_4 benefited from the development and testing of earlier products.

RQ8: Can the number of faults in a new product be predicted from previously existing products' data?

In addition to common components, P_3 and P_4 share high-reuse and low-reuse variation components with products P_1 and P_2, and as a result each one has only one single-use variation component. These variation components, however, are non-trivial, and together they contain over 2,000 lines of code. The fact that there are only two new components made the traditional classification into fault-prone and not fault-prone components infeasible. Instead, we used the data from P_1 and P_2 to create a linear model via stepwise regression [9], which we then used to predict the number of faults in the components introduced by P_3 and P_4.

Stepwise regression is an iterative feature selection method that builds a linear model by selecting predictors from the feature set having high correlations with the dependent variable. Each step in the creation of the model eliminates the least significant feature, resulting in a smaller, more highly correlated feature set. Our final model consisted of the following static code and change metrics: LoC, Number of Files, New Features, CodeChurn, Average CodeChurn,

FileChurn, and Average FileChurn. We used this model to predict the number of faults in the two components new to P_3 and P_4, which resulted in absolute errors of 0.18 for P_3 and 0.08 for P_4 (i.e., the model predicted 0.18 and 1.08 faults for components which had zero and one fault, respectively). These results indicate that, in this SPL, a linear model of code and change metrics gathered from previously developed products can be used to accurately predict the number of faults in variation components of subsequently developed products. Testing the external validity of this result on another, preferably larger, case study is a topic of our future research.

VI. Threats to Validity

In this section, we describe several threats to the validity of this study and what measures were taken to mitigate them.

Construct validity addresses whether we are testing what we intended to test. One obvious threat to construct validity is having insufficiently defined constructs before their translation to metrics. Using inconsistent and/or insufficiently precise terminology in the area of software quality assurance is a serious threat to validity, often making meaningful comparisons of results difficult. Therefore, we provided the definitions of the terms and metrics used in this paper and avoided using terms, such as defects, that lack rigorous definitions or are used inconsistently across related works.

So called mono-operation bias to construct validity is related to under-representation of the cause-construct. Many empirical studies experience lack of some types of data that, if available, may improve the interpretation of the results or help explain the cause-effect relationships. Unlike many earlier studies on prediction of fault-proneness that considered only static code metrics, we also use change metrics, which, as our results and some of the related work results show, are more correlated with the number of faults than the static code metrics.

Internal validity threats are concerned with influences that can affect the independent variables and measurements without researchers' knowledge. Data quality is one of the biggest threats to internal validity. To ensure the quality of the MR data, we studied each individual record with the lead developer. MRs related to New Features and Improvements which were not closed were excluded from the analysis, as well as MRs which were not related to the actual code. Furthermore, as described in Section III, the remaining MRs were preprocessed to reflect that around 10% of all MRs were mapped to more than one component. The missing information in some MRs (e.g., whether the MR is directly related to implemented code) was acquired through an iterative and painstaking process of review with each step receiving validation from the lead developer of the products, who is also a co-author of this paper.

In cases where the components in the source code were organized differently than the original design documents, preference was given to the source code over design documentation. As a result, each component consists of files that belong to only one of the defined degrees of reuse.

Conclusion validity is concerned with the ability to draw correct conclusions. The most obvious threat to conclusion validity is using statistical tests in cases where their underlaying assumptions are violated. Since none of our data conformed to the normal distribution we used the less powerful Spearman correlation test to avoid violating the normality assumption of the Pearson test. Additionally, the similarity of product pairs $\{P_1, P_3\}$ and $\{P_2, P_4\}$ in this SPL could play a role in the prediction of number of faults in variation components of P_3 and P_4. This threat to the conclusion validity is based on the nature of our case study and cannot be lessened in this paper.

External validity is related to the ability to generalize the results. Obviously, research based on one case study cannot claim that the results would be valid across other studies. The external validity of our study is, to some extent, supported by the fact that whenever possible we compared our results with related works. Dissemination of our results and results based on other SPLs in time will provide support for identifying observations that apply across multiple studies.

VII. Concluding remarks

This paper described an empirical study of pre-release software faults in an industrial software product line. The main research goals and findings are summarized as follows:

(1) Are there any correlations between the number of faults at the component level and code and change metrics? We found that change metrics have higher correlations to the number of pre-release faults than static code metrics. Furthermore, our data revealed that Maximum Complexity was correlated with pre-release faults but the correlation was low, while Average Complexity was not correlated at all. For the products in this SPL, the majority of pre-release faults were contained in around one-fifth of the components.

(2) Does the degree of reuse play a role in a component's fault-proneness or change-proneness? Our research indicates that components used in only one SPL product are the most likely to change and have the highest fault densities of any degree of reuse. Common components reused in all four products had fault densities close to those used in three products and higher Average CodeChurn.

(3) Do products developed later benefit from the development and testing of earlier, more mature members of the SPL? Furthermore, can the number of faults in the variation components of subsequent products be predicted using data collected from earlier products? Our results suggest that later products do benefit from the faults fixed in the components they share with other concurrently or previously developed products. This provides some empirical support for the assumption that is at the heart of product

line development, namely, that through structured reuse of core, common components, the subsequent products will be less fault-prone and require less time and effort to develop and test. However, reused components also experienced an increased number of New Features and Improvements, which were introduced to accommodate requirements due to the introduction of new products. We also showed that, in the SPL studied, the data from more mature products could be used to build a model that predicts the number of faults in subsequent products. Models developed in this manner might provide developers with fault prediction tools that are specialized for their specific SPL and could be useful in pinpointing those components that will likely exhibit the highest number of faults.

These results suggest three lessons learned that may affect other product lines. First, the finding that change metrics are more highly correlated to faults than are static code metrics helps make the case that *rigorous change control* is central to the quality of product line products. Second, the finding that there is a spectrum of component reuse (ranging from commonalities through high-reuse, low-reuse, and single-use components, see Figure 1), with *significant, measurable differences among their fault profiles*, tends to confirm that the high degree of planned reuse in product line development enhances the quality of products. Lastly, even in the context of the systematic reuse in a software product line, existing products are changed to accommodate new requirements, sometimes due to products that are gradually introduced into the product line. The sustainability of a product line over time seems to depend on *consistent, ongoing reuse* with a few, cohesive variations.

We conclude the paper with a remark that a *centralized repository with product-line data* would help overcome the notable lack of empirical data for product lines that hampers product line research. It is encouraging that the plan for the product line used here is to release at least some of its products as open-source in the future, which could benefit both product-line research and teaching.

ACKNOWLEDGEMENTS

This work was supported by National Science Foundation grants 0916275 and 0916284 with funds from the American Recovery and Reinvestment Act of 2009. We thank David Weiss for his help in accessing the PolyFlow data.

REFERENCES

[1] www.computer.org/portal/pages/seportal/subpages/sedefinitions.html
[2] C. Andersson and P. Runeson, "A replicated quantitative analysis of fault distributions in complex software systems," *IEEE Trans. on Softw. Eng.*, vol. 33, pp. 273–286, May 2007.
[3] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Computer*, vol. 34, pp. 135–137, January 2001.
[4] G. Chastek, J. McGregor, and L. Northrop, "Observations from viewing Eclipse as a product line," in *3rd Int'l Workshop on Open Source Software and Product Lines*, pp. 1–6, 2007.
[5] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. on Softw. Eng.*, vol. 26, pp. 797–814, 2000.
[6] W. B. Frakes and G. Succi, "An industrial study of reuse, quality, and productivity," *J. Syst. Softw.*, vol. 57, pp. 99–106, 2001.
[7] M. Hamill and K. Goseva-Popstojanova, "Common trends in software fault and failure data," *IEEE Trans. on Softw. Eng.*, vol. 35, pp. 484–496, 2009.
[8] T. Khoshgoftaar and J. Munson, "Predicting software development errors using software complexity metrics," *IEEE J. Selected Areas in Communications*, vol. 8, no. 2, pp. 253–261, 1990.
[9] D. G. Kleinbaum, L. L. Kupper, and K. E. Muller, Eds., *Applied regression analysis and other multivariable methods*. PWS Publishing Co., Boston, MA 1988.
[10] S. Krishnan, R. R. Lutz, and K. Goševa-Popstojanova, "Empirical evaluation of reliability improvement in an evolving software product line," in *8th Working Conf. on Mining Software Repositories (MSR'11)*, pp. 103–112, 2011.
[11] S. Krishnan, C. Strasburg, R. R. Lutz, and K. Goševa-Popstojanova, "Are change metrics good predictors for an evolving software product line?" in *7th Int'l Conf. on Predictive Models in Softw. Eng.*, pp. 7:1–7:10, 2011.
[12] W. Lim, "Effects of reuse on quality, productivity, and economics," *IEEE Software*, vol. 11, no. 5, pp. 23 –30, 1994.
[13] P. Mohagheghi, R. Conradi, O. Killi, and H. Schwarz, "An empirical study of software reuse vs. defect-density and stability," in *26th Int'l Conf. on Softw. Eng. (ICSE '04)*, pp. 282 – 291, May 2004.
[14] P. Mohagheghi and R. Conradi, "An empirical investigation of software reuse benefits in a large telecom product," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, pp. 13:1–13:31, 2008.
[15] R. Moser, W. Pedrycz, and S. G., "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *30th Int'l Conf. on Softw. Eng. (ICSE'08)*, pp. 181–190, May 2008.
[16] J. Munson and T. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Trans. on Soft. Eng.*, vol. 18, no. 5, pp. 423–433, 1992.
[17] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *27th Int'l Conf. on Softw. Eng.(ICSE '05)*, pp. 284–292, 2005.
[18] N. Nagappan, T. Ball and A. Zeller, "Mining metrics to predict component failures," in *28th Int'l Conf. on Softw. Eng. (ICSE '06)*, pp. 452–461, 2006.
[19] T. J. Ostrand, E. J. Weyuker and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. on Soft. Eng.*, vol. 31, no. 4, pp. 340–355, 2005.
[20] R. Selby, "Enabling reuse-based software development of large-scale systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 495–510, 2005.
[21] W. M. Thomas, A. Delis, and V. R. Basili, "An analysis of errors in a reuse-oriented development environment," *J. Syst. Softw.*, vol. 38, pp. 211–224, 1997.
[22] D. M. Weiss and C. T. R. Lai, *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1999.
[23] D. M. Weiss, J. J. Li, H. Slye, T. Dinh-Trong, and H. Sun, "Decision-model-based code generation for SPLE," *Int'l Software Product Line Conf.*, pp. 129–138, 2008.
[24] W. Zhang and S. Jarzabek, "Reuse without compromising performance: Industrial experience from RPG software product line for mobile devices," *9th Int'l Conf. Software Product Lines*, pp. 57–69, 2005.
[25] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for Eclipse," in *3rd Int'l Workshop on Predictor Models in Softw. Eng. (PROMISE'07)*, pp. 9–, 2007.