Quality Assessment and Prediction in Software Product Lines

Thomas Ryan Devine
West Virginia University

Quality Assessment and Prediction in Software Product Lines

by

Thomas Ryan Devine

Thesis submitted to the
Benjamin M. Statler College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2013

Keywords: software product line, quality assessment, fault prediction, empirical studies

Abstract

Quality Assessment and Prediction in Software Product Lines

by

Thomas Ryan Devine
Master of Science in Computer Science

West Virginia University

Katerina Goseva-Popstojanova, Ph.D., Chair

At the heart of product line development is the assumption that through structured reuse later products will be of a higher quality and require less time and effort to develop and test. This thesis presents empirical results from two case studies aimed at assessing the quality aspect of this claim and exploring fault prediction in the context of software product lines. The first case study examines pre-release faults and change proneness of four products in PolyFlow, a medium-sized, industrial software product line; the second case study analyzes post-release faults using pre-release data over seven releases of four products in Eclipse, a very large, open source software product line.

The goals of our research are (1) to determine the association between various software metrics, as well as their correlation with the number of faults at the component/package level; (2) to characterize the fault and change proneness of components/packages at various levels of reuse; (3) to explore the benefits of the structured reuse found in software product lines; and (4) to evaluate the effectiveness of predictive models, built on a variety of products in a software product line, to make accurate predictions of pre-release software faults (in the case of PolyFlow) and post-release software faults (in the case of Eclipse).

The research results of both studies confirm, in a software product line setting, the findings of others that faults (both pre- and post-release) are more highly correlated to change metrics than to static code metrics, and are mostly contained in a small set of components/packages. The longitudinal aspect of our research indicates that new products do benefit from the development and testing of previous products. The results also indicate that pre-existing components/packages, including the common components/packages, undergo continuous change, but tend to sustain low fault densities. However, this is not always true for newly developed components/packages. Finally, the results also show that predictions of pre-release faults in the case of PolyFlow and post-release faults in the case of Eclipse can be done accurately from pre-release data, and furthermore, that these predictions benefit from information about additional products in the software product lines.

Acknowledgments

I would first like to thank my committee chair and adviser, Dr. Katerina Goseva-Popstojanova, for giving me the opportunity to work with her and her students. This thesis would not be possible without her constant guidance and support. Her many long nights of editing and inestimable assistance with making sense of this massive amount of real-world data are greatly appreciated!

I would also like to thank Dr. Robyn Lutz and Dr. Arun Ross for being on my committee. I have been fortunate to have worked directly with Dr. Lutz in collaboration on this research, and her editing skills and insights into the subject matter were invaluable. I have also had the opportunity to take courses with Dr. Ross. Both of their insights and teachings have been essential to my understanding of the subject.

Next, I would also like to thank Dr. Jenny Li with Avaya labs and Sandeep Krishnan at Iowa State University. These two people were invaluable in helping me gather, verify, validate, and understand the data for both case studies.

Finally, I would like to thank my family, especially my fiancé Lauren and my mother Mary Ann. Their enduring love, support, patience, and belief in me are my guiding light and give me the courage to attempt all things.

This work is funded in part by the National Science Foundation under the grant CCF-0916284 with funds from the American Recovery and Reinvestment Act of 2009.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The first chapter presents the motivation for this thesis. We include a description of software product lines (SPLs) in general followed by brief, high-level descriptions of the two case studies performed and the principal questions that guided our research.

In software engineering today, a widely used approach to reuse is through development of a software product line (SPL), which explicitly defines the common and variable components present in a family of systems [16]. Weiss and Lai define a software product line as, "a family of products designed to take advantage of [their] common aspects and predicted variabilities" [50]. A software product family[1] is a group of software products similar in purpose which share a basic platform of core components, yet are individualized by utilizing a set of variation components. These common, core components are called "commonalities" while the variation components are often generally referred to as "variabilities" [16]. For example, consider a vehicle manufacturer's use of cruise control software. The basic functions of the software are the same across all models of vehicles, so many core components in the software may be reused. However, each separate model has its own variations in hardware and user interface that require slightly different software components to be effective. Rather than rewrite the entire software system for every model of car, a software product line approach can take advantage of structured reuse to reduce production time and cost and increase overall product quality [46].

[1]With regards to terminology, the terms "software product family" and "software product line" are often used interchangeably, as noted by Pohl et al. in [46]. Throughout this thesis, we use both terms.

Developing software from reusable parts is not a new idea. However it has been demonstrated in the field of software product line engineering that structured and strategic reuse can result in considerable improvements in quality, time-to-market, scalability, and software engineering cost [28]. Software product lines are set apart from other forms of reuse by being *predictive* rather than *opportunistic*. By planning out the commonalities and effectively managing variabilities in a software family, software product line engineering (SPLE) allows for *mass customization*, the ability to efficiently create many variations of a product to suit the specific needs of users [28]. By combining mass customization with a common platform, SPLE allows for the simultaneous reuse of a common base of technology and delivery of products in close accordance with customers requirements [46].

There have been several empirical studies in the software engineering community which provide evidence of the benefits of systematic reuse. Reuse 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 [14], [30], [36], [47], [49]. Though structured reuse is prevalent in SPLE, few empirical case studies on quality assessment in SPLs exist in the literature. This provided a strong motivation for both of the empirical studies presented in this thesis.

Another open research area in the software engineering community is fault prediction. While many empirical studies attempt to develop consistent and accurate fault prediction methods, none of them take place in a product line setting. Fault prediction studies test intuitive claims related to learning from prior mistakes, i.e., by studying software development history, accurate models can be constructed to identify future modules which are likely to be faulty. The intertwined relations among members of a product line offer a unique test bed for examining fault prediction.

In this thesis we present two empirical studies of two very different software product lines. The first study examines pre-release software faults in four members of the PolyFlow product line of software testing tools developed by Avaya [51]. In the second study, we present an empirical quality assessment and results from experimental post-release fault prediction models of four products from Eclipse, an open source product line. We are motivated by the current lack of empirical results related to the benefits of reuse and to fault prediction in a

SPL context.

In this thesis, a “fault” is defined as an accidental condition, which if encountered, may cause the system or system component to fail to perform as required [3]. A “failure” occurs when the software system fails to produce the expected results. Thus, a single failure could be caused by one or more software faults. We use and explicitly define the term fault because other terms, such as “defect” or “error” are used inconsistently throughout the software engineering literature.

All predictive models used in the studies presented in this thesis are numerical. We predict the number of faults for a unit (i.e., component or package) of software, rather than performing binary classification of “fault-prone” or “not fault-prone” on units. When performing an empirical analysis of software faults, it is tempting to use the binary classification for units. However, some software units exhibit many more faults than others, a fact that is lost if a unit with twenty faults and one with a single fault are lumped together under the classification “fault-prone”. Furthermore, numerical prediction conveys more information useful for determining the effort required to repair faulty software units. This, in turn, may allow for more efficient allocation of verification and validation resources.

In the PolyFlow study, we focus on examining pre-release trends in an industrial SPL during its development and testing phases. To evaluate this SPL, we extracted data from the modification request tracking system and the source code repository for the four products. 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 PolyFlow study explores the following three main areas, each with its own research questions:

1. association of pre-release software faults with other metrics at the component level,
2. characteristics of pre-release fault and change proneness depending on the level of reuse, and
3. longitudinal study of pre-release faults over the period of development and testing of the SPL.

We first study the correlation between each possible pair of different metrics gathered at the component level. Specific attention is given to the correlations between the number of pre-release software faults and collected static code and change metrics. Secondly, we study the fault-proneness and change-proneness of components with different levels of reuse. This analysis addresses issues central to SPLs and their structured reuse in the form of commonalities and variabilities. The research questions (RQs) in areas (1) and (2) assume a cross-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. Furthermore, these research questions address the benefit newly generated products gain from product line reuse and focus on prediction of pre-release faults in new products.

In our second study, we present a longitudinal analysis of the quality of products in a large, open source software product line. The study is based on four products in the Eclipse software product family, which currently consists of fourteen different members that systematically share main components and are set apart by variable components. New products are introduced by combining parts of pre-existing products with new packages specifically developed for the new products. (A package, in this context, is a group of related Java class files.) Eclipse has a broad user base and maintains extensive collections of reported faults and release archives, enabling empirical studies of the quality of products developed in a SPL. Therefore, Eclipse is a fertile area for empirical research in a software product line context. For Eclipse, we follow trends in post-release faults across multiple releases. We examine static code metrics, gathered by analyzing downloaded source code for each of seven releases, combined with change metrics, which are taken from analysis of the CVS repository used to host all of the code for the Eclipse project.

The research questions which guided the Eclipse study are divided into two broad categories, each with its own research questions:

1. assessment of the overall *quality* of the SPL, and

2. *prediction* of post-release faults from pre-release data.

The first set of questions focuses on assessing whether the overall quality of the product line improves through releases over time. This is directed by a longitudinal examination of post-release fault trends, identifying the faultiest packages, and determining the benefit in quality, if any, that new products receive from reuse in the SPL. The second set of questions are devoted to the *prediction* of post-release faults from pre-release data. These questions attempt to determine if accurate predictions of post-release faults can be made from pre-release data, which features are the best predictors, and whether any benefit can be gained from predictions made across members of the product family.

It is our belief that by contributing these studies to the existing literature, software developers may find insight into the benefits of structured reuse, the process of making accurate fault predictions, and the quality of products developed in SPLs. This thesis offers directly actionable information that could be used to focus testing resources, determine release dates, facilitate update and patch creation, and guide efforts to refine and improve future predictive efforts.

1.1 Thesis Outline

Herein we present an outline of the content of the rest of this thesis.

In chapter 2, we give a thorough review of current work in the literature relating to our research. We review related work concerning the field of software reuse and its benefit, quality assessment in terms of the correlation of software metrics to pre- and post- release faults, and prediction of continuous values for software faults.

The contributions this thesis offers to the software engineering community are listed, in chapter 3. The contributions are grouped by case study.

Chapters 4 and 5 provide detailed accounts of the results from the industrial and open-source SPLs presented in this thesis, respectively. For each case study, the software product line used as the object of the study is first described. Then we detail the methods used to extract the data and what kind of data were extracted, followed by detailed results. Each chapter concludes with a discussion of study-specific threats to validity and the actions taken to mitigate them.

In chapter 6, we summarize the findings of the thesis, breaking the conclusions down for each product separately. We conclude with a summary of the main results that are consistent across both studies.

Chapter 2

Related Work

In this chapter we present a thorough review of all literature relating to systematic reuse in software development, as well as quality assessment and prediction in software engineering, both in general and within the specific context of software product lines.

2.1 Related work on reuse

Several empirical studies have been conducted on the general benefits of systematic reuse. Of the case studies that examine the beneficial aspects of reuse as it relates to code quality, Frakes and Succi's analysis of four different sets of industrial data [14] 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 [47], 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. [49]. 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 [30] and found in both cases a reduction in fault density of around four times for the reused code over the newly developed code.

2.2 Related work on quality assessment

In this thesis, we explore the correlation of various static code and change metrics to the number of faults (pre- or post-release) in a component/package. This type of analysis has been performed before on software products that may or may not utilize reuse. Nagappan and Ball performed a correlation analysis between change metrics gathered from the forty-five million lines of code in Windows Server 2003 and the associated fault database [38]. In [5], Andersson and Runeson presented an empirical study including correlational analysis of static code metrics and fault densities for three members of a large telecommunications SPL. Zimmerman et al. computed the Spearman correlations between faults and fourteen different static code metrics collected from the Eclipse project in [54]. Metrics were calculated based on both pre- and post-release data and the file level data were aggregated into packages.

2.3 Related work on numerical prediction

Prediction of fault-proneness is another active subject of research in software engineering.

Several works in the literature have constructed and tested numerical models for fault prediction, which aimed to predict the number of faults at a unit level (e.g., file, component, package) rather than providing a binary classification of whether the unit is fault-prone or not[1].

Of these papers, four have used Eclipse as a case study [10], [11], [20], [54]. It should be noted that none of [10], [11], [20], [54] considered the SPL aspects of Eclipse. Rather, they analyzed collections of files and/or packages in several subsequent releases of Eclipse. In particular, recently D'Ambros and Robbes used generalized linear models to explore the utility of change coupling metrics for predicting post-release faults in [10] and to compare predictive techniques on four different Eclipse components in [11]. Both [10] and [11] used n-fold cross validation within a single data set to arrive at their final results. Kamei et al. used linear regression, regression tree, and random forest models to predict post-release faults in three components of Eclipse in [20]. Experimental data showed that fusion performed after

[1]For a comprehensive survey of binary classification studies the reader is referred to the recent paper [17].

making file-level predictions provided slightly better results than aggregating file-level static code and process metrics to make predictions on the package-level. The results were validated by both a fifty-fifty split, where training was performed on half the data and the model was tested on the other half, and by building models on one release and predicting on the next. Zimmerman et al. in [54] used linear regression models on both file and package levels to perform a ranking from most to least faulty file and package, respectively. Models were built for each of three releases of Eclipse (2.0, 2.1, and 3.0) and tested on every other release.

Numerical, post-release fault prediction studies of other software products not related to Eclipse include [6], [7], [9], [21], [22], [29], [39], [43], [44], [45], [48], and [52]. Bell et al. examined a voice recognition system using Naive Bayesian models to compare the predictive ability of different combinations of LOC and change metrics in [6]. Models were built on data from all of the previous releases and then used to predict the fault-proneness of the next release. In [7], Bibi et al. compared twelve different models to determine the benefits of regression via classification. Results were validated using n-fold cross validation. Cruz et al. in [9] used logistic regression models built from object oriented metrics gathered from one project to predict post-release faults in different, unrelated projects. Static code metrics were combined with change metrics by Kastro and Bener in [21] to create neural network prediction models for Linux. In [22], Taghi and Munson used complexity metric features selected by stepwise regression or factor analysis to compare linear regression models which predicted fault densities. Li et al. also used linear regression and neural network models, as well as clustering, tree, and moving average models built from previous releases to predict the number of faults in the next release in [29]. The models were constructed from source code, change, deployment, and usage metrics. In [39] Nagappan et al. built logistic regression models from static code metrics alone on the module level and made predictions within a single project and across five different Microsoft projects (Internet Explorer 6, IIS W3 Server Core, Process Messaging Component, DirectX, and NetMeeting). The remaining studies all used negative binomial regression for binary classification on different software systems to predict fault-proneness and validated their results by building models on one or all previous releases, then making predictions on the next. Ostrand et al. built models from file level information on LOC, number of previous faults, and change metrics in [43] and [44], and used LOC and change metrics only in [45]; in [52], Weyuker, et al. used static code metrics combined with change metrics to build models at the module level; and in [48], Shin et al. used different combinations of LOC, static code metrics, change metrics, faults from previous releases, and calling structure information to construct negative binomial regression models.

2.4 Related work on quality assessment and prediction in SPLs

Large, industrial product lines rarely provide data for academic research. To bypass this problem, Zhang and Jarzebek developed four members of a mobile gaming product line, both simultaneously using a SPL architecture and developing each product independently [53]. The results showed that the products developed under the SPL architecture were easier to develop and maintain, consisted of less total code, and also showed a decrease in execution speed and memory usage.

Mohaghegi et al. examined data from three large telecom product lines in [35] and [36]. The empirical analysis concluded that components used in multiple products required fewer modifications and had fault densities 39% to 56% less than those of components used in only one product. The hypothesis that fault density was not related to component size in either type of component could not be confirmed or rejected by the data.

Some previous work from our collaborative research group funded by NSF was also based on Eclipse, viewed as a SPL. Specifically, in [25] we analyzed the change metrics and post-release fault data from four releases of the Eclipse project. In that work, components were grouped based on the level of reuse across the product line family. The results showed that commonalities followed a decreasing trend in file churn through subsequent releases and exhibited fewer post-release faults than any other level of reuse. Furthermore, variable components exhibited a high degree of change as the SPL evolved through releases.

In a follow up study [26] we used change metrics as features and the J48 decision tree algorithm to classify Eclipse components as fault-prone or not fault-prone. The classification results were very good (probability of detection 79% to 85%, probability of false positive 2%

to 4%), with the particular subset of change metrics {number of authors, Changeset (i.e., number of files committed along with this particular file), Number of revisions} performing well throughout all the studied releases of the SPL. Additionally, the results showed that as the product line evolved the learner's performance improved.

We further studied Eclipse in [27] by using multiple learners for classification and comparing three data collection approaches: using change and fault data from the entire release (i.e., no distinction between pre-release and post-release faults); using twelve months of change data and considering the file as faulty only if it displayed post-release faults; and using pre-release change and fault data to predict fault-prone files post-release. The best results were achieved via the first data collection technique, while using pre-release data to classify the files as fault-prone or not fault-prone post-release resulted in much lower true positive classification rates.

Chapter 3

Contributions

The findings of our research provide several contributions to the software engineering community and are set apart from the rest of the literature. In this section, we list the contributions of each empirical study individually.

3.1 Quality assessment and pre-release fault prediction in a medium sized, industrial software product line

In the PolyFlow study we examined four separate products which were each a subset of 42 total components. The entire set of components was comprised of approximately 65,000 lines of code. Our results related to the association of the number of faults and other collected metrics support the findings of other works [26], [38] 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 [5], [13], and [54] that most faults in pre-release testing are found in about 20% of the components. Furthermore, we found that complexity metrics were poor predictors of pre-release faults, which lends support to similar results in [13]. This part of our work can be considered as a literal replication carried out on a different type of system, within a different development context. Unlike more mature areas of scientific research, 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-proneness and change-proneness within different levels 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 previous work of our group in [25] working on the collaborative research project funded by the NSF, which was also based on analysis of post-release failures of Eclipse. Unlike [25], in the case study of PolyFlow we studied pre-release faults and change proneness of an industrial SPL. Results showed that variation components used only in individual products had the highest fault density, and were the most change prone. As in [25], common components reused in all four products had similar fault densities and higher average code churn than the high-reuse variation components. Closer exploration showed that this was mainly due to one very change prone common component having many New Features, Improvements, and detected and fixed faults.

The longitudinal analysis from the PolyFlow study presented new results related to the development and testing of SPLs. We found that newer members of the software product line benefited 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 could be successfully predicted by a linear model developed using code and change 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. This latter result is new to the community because while others, [39], have used models created from one project to predict faults in another project, this has not been done in the context of a SPL.

Throughout the literature many 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 component. 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 [39] 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.

3.2 Quality assessment and post-release fault prediction in a large, open-source software product line

The Eclipse study presented here also differs from the related work and offers its own contributions to the software engineering community. First, our analysis considers *multiple products* from the Eclipse product line rather than *a collection of individual files and/or packages*. This allows us to determine the benefits of building models and making predictions on different products in a software product line. Other studies that attempted to make predictions across software applications (such as [9] and [39]) were based on unrelated software applications (e.g., Internet Explorer 6, IIS W3 Server Core, Process Messaging Component, DirectX, and NetMeeting), rather than products that are members of a SPL. Second, we collected a large amount of data in both size, as measured in number of files and lines of code, and duration, as measured by number of releases and weeks in existence. We gathered both static code and change metrics and linked them to post-release faults for seven releases of Eclipse (i.e., seven years). This information totaled 125,118 files containing over 20 million lines of code.

For this case study we focused on post-release faults observed over seven releases in four large products of an open-source product line. Furthermore, in this study we apply a specific generalized linear model to the problem of numerical software fault prediction for the first time. While generalized linear models were used in [10] and [11], the linking functions were not specified. In this study, we use the cumulative negative log-log linking function, which is well suited for software fault prediction. This is because it is optimal for skewed distributions characterized by higher probabilities of lower or zero values that are typical for the way post-release faults are distributed across software units (i.e., files, components or packages).

Compared to previous studies of Eclipse in [26] and [27] performed by our collaborative research group under NSF funding, this study incorporates both change and static code metrics, and performs numerical prediction rather than binary classification. We also propose a new machine learning approach, which has not been used previously either for numerical prediction of post-release faults or for classification of fault-prone units. Specifically, we constructed a model from the data of one member of the product line family and used it to predict the number of post-release faults for packages of other members in the subsequent release. This approach allowed us to explore whether predictions for each product benefit from additional product line information.

The results of the Eclipse study extend related work in this area, including our study of PolyFlow, toward identifying actionable insights. The key aspects of the work presented in this case study that support these actionable insights are as follows:

- In addition to using traditional performance measures, such as Spearman's ρ and Kendall's τ, to assess the predictive ability of our models, we introduce a new normalized measure of the number of post-release faults in the top 20% of the most fault-prone packages. This new measure, which we call the *nTop20%*, attempts to establish a method of quantifying the predictive value of models that is commensurable between different projects.

- The *scope* of the empirical study is large, focused on four distinct Eclipse products longitudinally across seven releases.

- We perform extensive *quality* assessment of Eclipse as a product line by analyzing post-release fault and change trends and how systematic reuse improves the quality of products.

- We use a broader range of *metrics* (i.e., features) than the previous predictive studies, including static code metrics at the release date combined with change metrics and pre-release faults collected during the six month period prior to the release date. Our response variable is the number of user reported faults submitted during the six month period after the release of the software product.

- The predictions are based on a specific generalized linear regression *model* which is optimal for data that follow skewed distributions, typically observed when examining post-release fault data. Predictive models created via this theoretically appropriate regression model performed very well in this study.

- The number of post-release faults at package-level are predicted using the pre-release static code and change metrics from the model *built on the previous release.* Predictions made in this manner, as opposed to using cross-validation or bootstrapping, are of direct use to developers because they mimic the actual data collection process and thus have more practical value.

- Unlike previous works which made predictions for Eclipse and other software applications over multiple releases, in this study we build models from the individual products in each release and then use these models to predict the number of post-release faults for each product in the subsequent release. Overall, we build 19 models, which are used to make predictions for a total of 54 combinations of products and releases. It appears that this is the first research work that explores the benefit of *additional product line information* on predictions of post-release faults.

- The models are used to *predict the number of post-release faults* in the top 20% of the most fault-prone packages, as well as to rank software packages based on the number of post-release faults they contain. Compared to the typical binary classification of packages as fault-prone or not, this type of prediction conveys more information about the additional effort required to repair faulty packages, which in turn may allow for more efficient allocation of verification and validation resources.

Chapter 4

Quality assessment and pre-release fault prediction in a medium sized, industrial software product line

This chapter presents a description of an empirical case study of the software product line (SPL), PolyFlow[1]. It begins with a brief description of the SPL being studied, which is followed by the results of our analysis. This chapter concludes with a detailed account of different threats to the validity of this study and the steps that were taken to mitigate them.

4.1 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 measures [51], 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 and was designed with a modular architecture, i.e., related classes were grouped into packages that serve as components.

From this SPL, we were able to examine the Modification Request (MR) database and

[1]An earlier version of this chapter appeared as Thomas R. Devine, Katerina Goseva-Popstojanova, Sandeep Krishnan, Robyn R. Lutz, J. Jenny Li: An Empirical Study of Pre-release Software Faults in an Industrial Product Line. ICST 2012: 181-190

Table 4.1: Number of components and LoC for the four products from the PolyFlow SPL examined in this thesis

Product	*Components*	*LoC*
P_1	22	47,138
P_2	33	35,238
P_3	22	49,676
P_4	28	36,852

source code repository for four products P_1, P_2, P_3, and P_4. Each product is a different subset of 42 components totaling approximately 65,000 LoC. Table 4.1 shows the number of components and LoC that comprise each product. The total number of components in Table 4.1 is greater than 42 due to components being used in more than one product. Figure 4.1 shows the distribution of the components in a Venn diagram, to help visualize their organization by level 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.

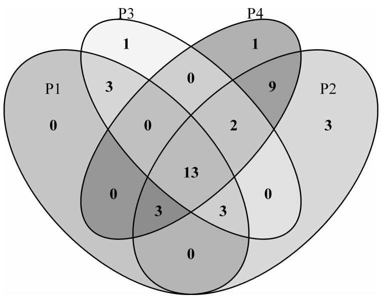


Figure 4.1: The distribution of components among four products

Figure 4.2 presents a Gantt chart depicting the development effort in the Polyflow product line chronologically. The period covered by the chart began with the initial development of P_1 and P_2 in January of 2008 and ended with the completion of P_4 in December of 2010, 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 was completed in the final span of seven months.

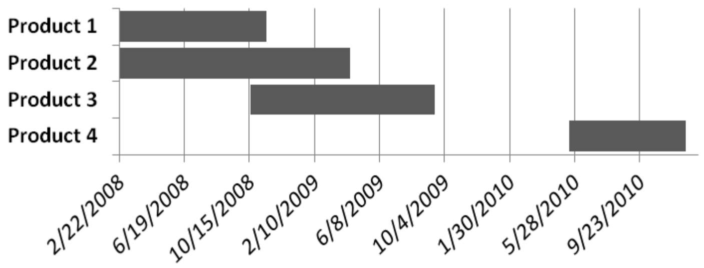


Figure 4.2: Timeline of products development

The MR database from which some of our metrics were mined consists of three MR types. The categorization of each MR in the database is dependent upon the nature of the change requested. 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 [3][2]. All faults analyzed in this study 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 it could be used in our analysis because we found that individual MRs were mapped to multiple components.

[2]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.

Table 4.2: 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

In particular, 10.3% of the MRs labeled *fixes* were mapped to two or three components. This percentage is somewhat lower than the 17% to 23% of fixes which affected two or more components in the two NASA data sets considered in our earlier work [18], but still shows a significant spread. Similarly to fixes, 11.5% of Improvements and 20% of New Features resulted in changes to two or three components. In order to perform analysis at the component level, we replicated fixes/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 4.2. 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 MRs were either not directly related to the products considered in this study or were not directly related to changes in the code of the products. For example, the database contained 26 MRs concerning changes to the Decision Model, which is defined as a specification of a partially-ordered sequence in which choices of values for parameters of variation must by made. While the Decision Model is used in the creation of each product, the code for it is not contained in any of the products and therefore contributes nothing to the functionality of the products. Of the remaining 188 MRs, 25 Improvements or New Features are still open and unresolved. This means that the additional functionality requested by those MR 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.

4.2 Description of metrics and the process of their extraction

For an empirical study to be effective, large amounts of data must be collected. For this study, we were fortunate to have access to several different sources of data from the PolyFlow product family, allowing a more diverse exploration of this SPL. We divided the metrics gathered into three categories: source code metrics, change metrics, and fault metrics.

4.2.1 Static code metrics

The static source code metrics considered for this study were gathered at the component level. They are listed in Table 4.3. These LoC and Complexity metrics were gathered at the class level using the freeware code analysis tool SourceMonitor [1]. We then aggregated these class level metrics into component level metrics. (The complexity metrics used by SourceMonitor approximately follows the definition by McConnell in [32]. It is a popular static code metric that represents the number of execution paths through a method.)

Table 4.3: A list of static code metrics and their descriptions

Static Code Metric	**Description**
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.
Average Complexity	The average complexity of the methods in a given component.
Maximum Complexity	The maximum complexity exhibited by any method in a given component

4.2.2 Change 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. Table 4.4 gives these metrics, which quantify the amount of modification to a component during development and testing. The data for Improvements and New Features were collected from the MR database and mapped to the components they affected as described in Section 4.1. The four churn metrics were extracted using the freeware application StatSVN and utility code that we wrote. The StatSVN application was used to analyze the SVN logs kept by the code repository and acquire general folder level metrics in an html format. We then wrote code to extract individual metrics for each file in the repository. These individual results were then aggregated to compute the change metrics at the component level.

Table 4.4: A list of change metrics and their descriptions

Change Metric	**Description**
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

Finally, the number of faults per component metric was computed based on the data from the MR database, as explained in Section 4.1.

4.3 Association of software faults with other metrics

The first two research questions in this section investigate the associations that exist between the gathered metrics and pre-release faults. The third research question examines whether the software engineering principal of a majority of faults existing in a small amount of the code holds true for this study.

Table 4.5 shows the results of the correlation analysis between all pairs of metrics. In this study we used the Spearman correlation, because the data did not conform to the normal distribution and thus violated the assumptions 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

Table 4.5: 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											-

significant results at the $\alpha = 0.05$ significance level. The blank cells represent either metrics with statistically not significant correlation or metrics which are highly correlated by definition and are therefore trivial (e.g., maximum complexity and average complexity).

Since the number of faults per component is of special interest, Figure 4.3 presents the scatter plots for each metric versus the number of faults in a component.

4.3.1 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 File Churn, 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 4.5: *The correlation of the number of faults at the component level with each of the change metrics except File Churn is higher than*

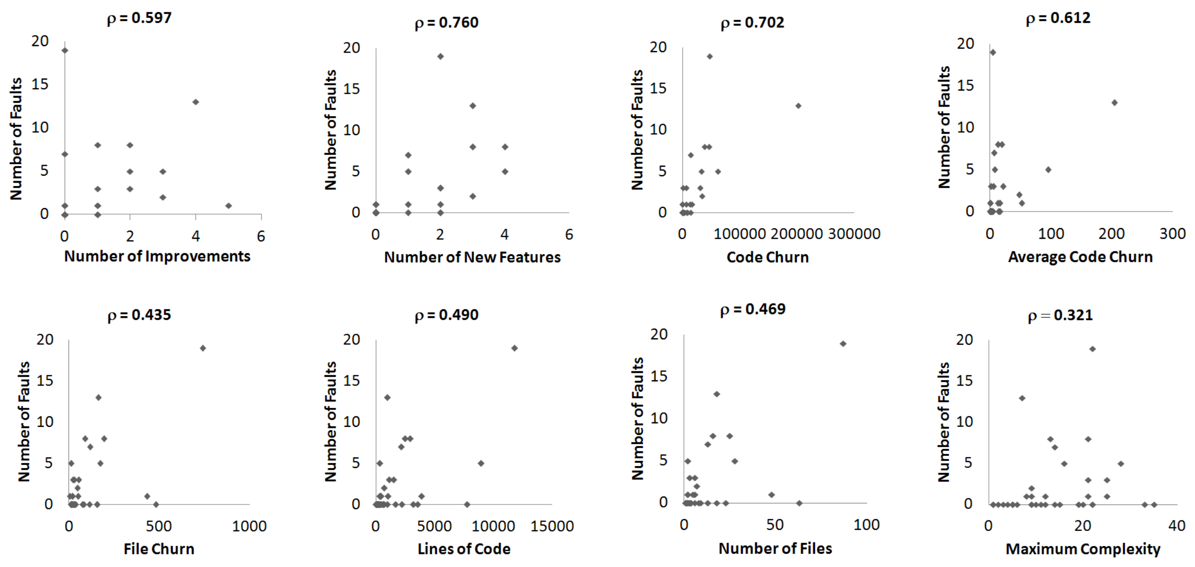


Figure 4.3: Scatter plots of the number of faults and each metric for which the correlation was statistically significant

the correlation with any static code metric. Specifically, the strongest correlation is with New Features and Code Churn (0.760 and 0.702, respectively), followed by the correlation with the Average Code Churn, Improvements, and File Churn (0.612, 0.587, and 0.439 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 [38] that average codechurn had a positive correlation to fault density that was both statistically significant and very high (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. [5] and Zimmerman et al. [54] 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 [13], Fenton and Ohlssen determined that cyclomatic complexity was not a good predictor of pre-release faults. Our results for the average cyclomatic complexity of a component 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 [54], 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 than ours, while we are unable to support the correlation between the average complexity and number of faults.

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

Our examination of the results in Table 4.5 led to the following observations about the association between metrics. Improvements and New Features are highly correlated to each other, and both are moderately to highly correlated to Code Churn and Average Code Churn. This is expected, as both Improvements and New Features lead to changes in the code. Similarly, New Features are correlated to File Churn because New Features often result in the addition of new files. Improvements, however, are not correlated with File Churn.

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 File Churn. However, Average Complexity is correlated with only Average File Churn 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.

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

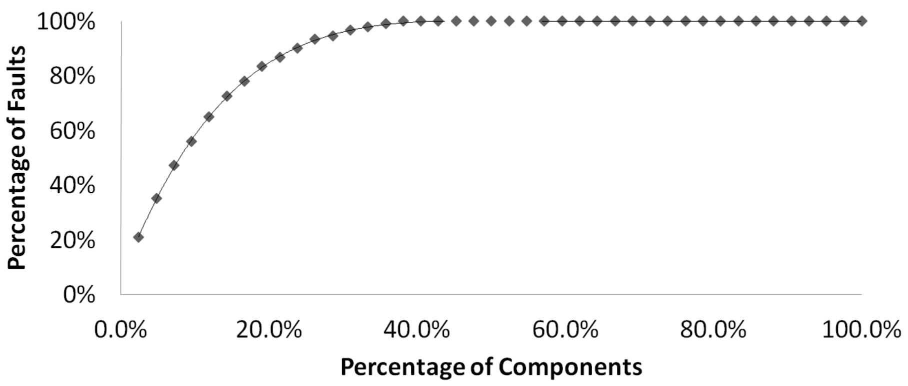


Figure 4.4: Percentage of components versus percentage of faults

To address RQ3, we computed the percentage of total faults for each component and plotted this against the percentage of the total components. The results of these calculations are ordered and plotted in Figure 4.4. The graph shows that approximately 85% of the bugs detected across all of the products are located in approximately 20% of the components. This result agrees with other works [5], [8], [13], [18], [42], which have consistently found that between 60 and 90% of bugs 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.

4.4 Fault and change proneness for different levels of reuse

This section studies the effects and benefits of the systematic code reuse in the PolyFlow SPL. To account for different levels 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 4.6. Some of the metrics shown in Table 4.6 are plotted on bar graphs for ease of visual comparison in Figure 4.5.

Table 4.6: Component level data organized by level 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

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

The groups of low-reuse variation components and single-use variation components share the fewest number of total faults (see Figure 4.5a). However, the low-reuse variation components have the lowest fault density after normalization by LoC (see Figure 4.5b). 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 components as non-reused components to conform to the context of [36], 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. (Four of the five single-use variation components have Maximum Complexities over fifteen, and three of those four have values over twenty. 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 4.3.

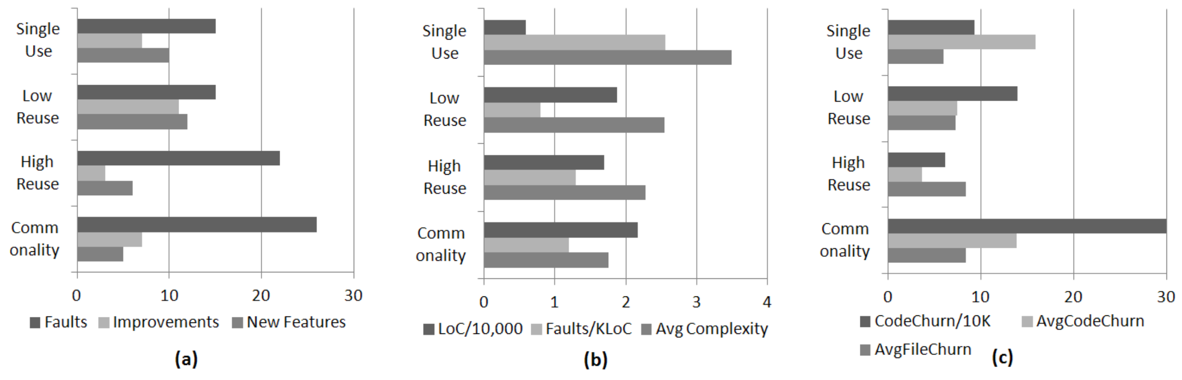


Figure 4.5: Comparisons of multiple metrics by level of reuse

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

As shown in Figure 4.5a, the low-reuse variation components exhibit the greatest number of New Features and Improvements. This is due to the fact that low-reuse variation components, which are reused in two products, often were not originally designed to be reused. Instead, when a new product was added, it was concluded that some components can 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 [25] 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.

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

Before normalization, the common components show the highest amount of Code Churn. However, when considering the amount of code contained in each reuse group (see Figure 4.5b), the single-use variation components have the most change-prone code (i.e., have the highest Average Code Churn). This finding lends support to the results of Mohaghehi et al. in [36] 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 Code Churn. One reason for this is that this study looks at code as it is being developed. As the development of the SPL progressed, and new products were introduced, common components had to be changed to accommodate different requirements from different 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 Code Churn 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 [25].

4.5 Longitudinal study of software faults over the span of development and testing

RQs 7 and 8 investigate the evolution of the SPL through its development. As depicted in Figure 4.2, development of P_1 and P_2 began simultaneously. Since these two products were the first to exist, in these questions we focus on the components of these two products, many of which are then shared by subsequently developed products P_3 and P_4.

4.5.1 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) i.e., 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 4.2).

The benefit of reuse is more prominent in the case of P_4, which is the only product in this study to be developed after the completion of the other three products in the SPL. It demonstrated the most benefit from prior fault fixes. Of the 69 faults affecting code components included in P_4, 67 were detected and fixed before P_4 existed (out of which 26 were in the common components, 22 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.

4.5.2 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 construct a linear model to predict the number of faults in the components introduced by P_3 and P_4. Numerical prediction in this manner can give a valuable insight into the degree of fault-proneness of a new component enabling more efficient estimation of testing effort and release date.

The data from P_1 and P_2 was used to create a linear model via stepwise regression [24]. 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,

Table 4.7: Results of numerical prediction of the number of faults in P_3 and P_4, using a predictive model created from the combined data of P_1 and P_2

Product	3	4
Actual # Faults	0	1
Predicted # Faults	0.18	1.08
Absolute Error	0.18	0.08

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, Code Churn, Average Code Churn, File Churn, and Average File Churn.

We used this model to predict the number of faults in the two components new to P_3 and P_4. As shown in Table 4.7, using the model created from the metrics of P_1 and P_2 resulted in absolute errors of 0.18 for P_3 and 0.08 for P_4. 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. We are hesitant to place too much emphasis on this result, however, as the sample size was regrettably small due to insufficient data.

4.6 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 wanted or 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 thesis 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. For example, many studies in prediction of fault-proneness consider only static code metrics. As our results and some of the related work results show, the number of faults is more correlated with change metrics than 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 domain expert. MRs related to New Features and Improvements which were not closed (i.e., implemented) were excluded from the analysis, as well as MRs which were not related to the actual code. Furthermore, as described in Section 4.1, 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.

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 any initial design documentation. As a result, each component consists of files that belong to only one of the defined levels of reuse. Our investigation also lead to the discovery of components that existed in multiple levels of reuse, e.g. one component had some files that were variabilities in P_2 and the rest were common to P_2 and P_4. In these cases the components in question were divided into multiple components, so that each one had files in only one area of reuse, a process which was also overseen and validated by the lead developer. There are a couple other notable threats to the internal validity of this study. First of all, there may be confounding factors at work throughout the development phase that we were unable to consider. Unmeasured, process-related factors such as the experience level of the programmers or the effort given to testing individual products or components could potentially affect the number of pre-release faults as much, or more than, the factors which we were able to measure. We again referred to the lead developer whenever possible to take any unmeasured factors into account when considering our results.

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. To avoid this, we were careful in our analysis of correlations to test the underlying assumptions of statistical tests before invoking them. For instance, none of our data conformed to the normal distribution. This forced us to use 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 study.

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. This threat is somewhat mitigated by our performance of similar analysis on a different, and much larger, SPL as the second case study presented in this thesis. This, together with dissemination of our results and results based on other SPLs in time will provide support for identifying observations that apply across multiple studies.

Chapter 5

Quality assessment and post-release fault prediction in a large, open-source software product line

This chapter presents a description of an empirical case study of the software product line (SPL) Eclipse[1]. It begins with a brief description of the SPL being studied, which is followed by a description of our machine learning approach and the results of our analysis. We conclude this chapter with a detailed account of several threats to the validity of this study and how we attempted to mitigate them.

5.1 Case study description

Eclipse is a set of products developed by an open-source collaboration to create integrated development environments (IDEs) to aid software development [2]. Originally created by IBM in November 2001, it is currently maintained by the Eclipse Foundation, a not-for-profit member supported corporation that hosts the various Eclipse projects. Eclipse is written in Java, and the original platform was a single product designed for Java and plug-in development. However, as support and ambition in the community grew, the scope of

[1]An earlier version of this chapter is submitted for publication as Thomas R. Devine, Katerina Goseva-Popstojanova, Sandeep Krishnan, Robyn R. Lutz: A longitudinal study of post-release faults in an evolving, open-source software product line.

the projects also expanded to encompass (currently) fourteen different products designed for development in many different languages and several different industries. Each of these products builds upon the common Eclipse platform shared by all. With well over a million downloads, Eclipse has an active user base. These qualities make Eclipse a fertile ground for testing research questions from the SPL community.

Our study examines four products from the Eclipse project, Classic, C/C++, Java, and JavaEE, through multiple releases. Table 5.1 gives the size of each product for each release. In the early releases, only one product existed in the Eclipse product line, namely Classic. Starting with the release codenamed Europa, Eclipse evolved from a single, all-encompassing product into several specialized products. These products all contain shared component code implementing the commonalities.

Table 5.1: A timeline of the products examined in this study with the sizes (in thousands of lines of code) and number of packages

Release	*Date*	*Classic*		*C/C++*		*Java*		*JavaEE*		***Total***		
(codename)		*KLoC*	*Pkgs*	*KLoC*	*Pkgs*	*KLoC*	*Pkgs*	*KLoC*	*Pkgs*	***KLoC***	***Pkgs***	***Faulty***
2.0	June 27, 2002	773	34							**773**	**34**	**26**
2.1	March 27, 2003	1,054	41							**1,054**	**41**	**37**
3.0	June 25, 2004	1,756	76							**1,756**	**76**	**70**
3.3 (Europa)	June 25, 2007	2,317	85	1,107	62	2,633	103	3,988	185	**3,988**	**185**	**148**
3.4 (Ganymede)	June 25, 2008	2,505	89	1,158	62	2,788	105	4,291	200	**4,291**	**200**	**152**
3.5 (Galileo)	June 24, 2009	2,125	77	1,117	61	2,748	104	3,913	188	**3,913**	**188**	**120**
3.6 (Helios)	June 23, 2010	2,208	77	1,184	61	2,921	105	4,262	206	**4,262**	**206**	**103**

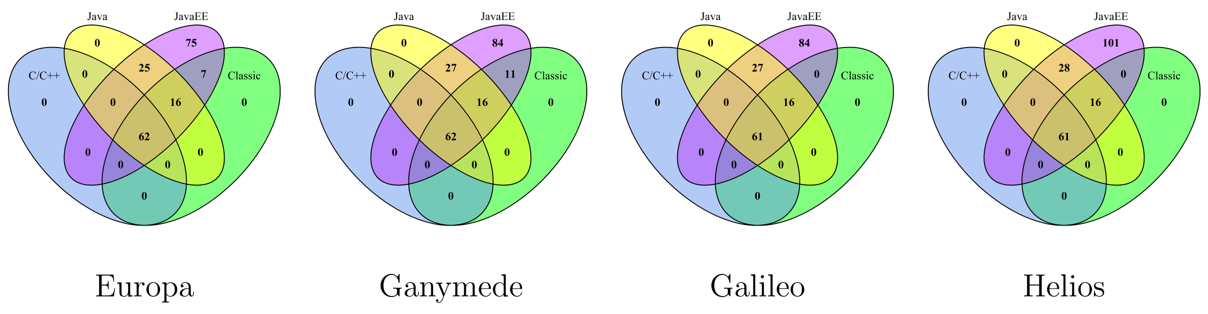


Figure 5.1: Venn Diagrams showing the distribution of packages among the four products for the final four releases considered in this study

Figure 5.1 provides a visual overview of the amount of code shared among the four products on the package level for the releases codenamed Europa through Helios. In each diagram, the 61 to 62 packages in the central region, which are shared by all four products, are *common packages*. For example, the package *org.eclipse.ui.ide*, which contains many of the classes involved in the user interface for the integrated development environments is included in every product in the Eclipse SPL.

The four regions directly adjacent to the center in Figure 5.1 contain packages that are used in three of the four products. In this case study, there are sixteen shared packages found in each release of Classic, Java, and JavaEE. For example, the package *org.eclipse.jdt.core* contains the classes that are the core of Eclipse's Java Development Tools, so it is naturally not included in the product C/C++. We label these packages used in all but one product *high-reuse variation packages*.

Low-reuse variation packages are in the regions where only two products overlap. In this study, are the 25 - 28 (depending on the release) packages shared by Java and JavaEE, such as Eclipse's Graphical Editing Framework (*org.eclipse.gef*).

The remaining 75 - 101 packages are currently used in only one product, and can be found in the perimeter regions of Figure 5.1. We call these packages *single-use variation packages*. The Web Standard Tools package *org.eclipse.wst.wsi*, used only in one product, belongs to this group.

Figure 5.1 shows that, following the terminology listed above, C/C++ is made entirely of common packages[2], and is thus a subset of the three remaining products. Java and Classic are specific subsets of JavaEE, which contains all the studied packages. Java and Classic contain a mixture of common, high-reuse variation, and low-reuse variation packages. JavaEE contains all of the single-use variation packages.

The total number of packages and the number of packages which exhibited post-release faults in each release are given in the right most column in Table 5.1 and shown in the bar graph in Figure 5.2.

[2]C/C++ contains one single-use variation package called CDT which handles development in the languages C and C++. However, we were unable to access source code for this package and therefore it was not considered in this study. For more details the reader is referred to Section 5.6.

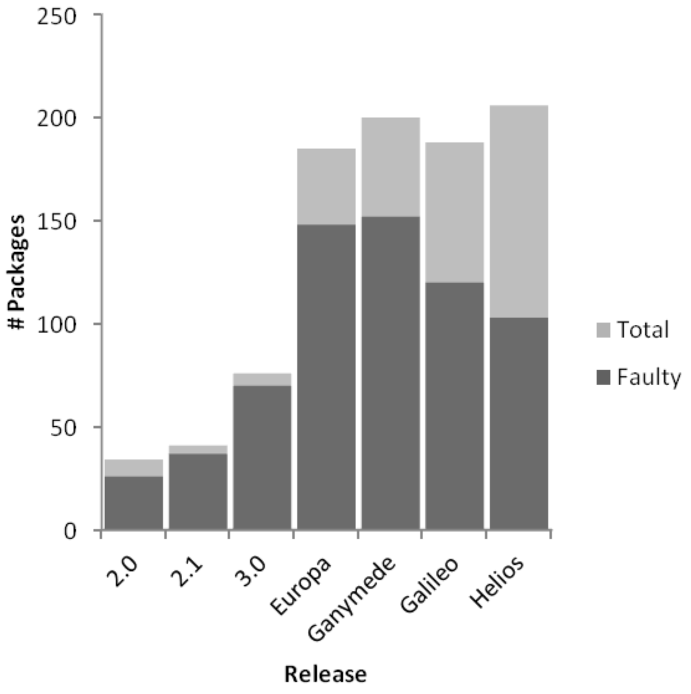


Figure 5.2: A histogram showing the total number of packages and the number of packages which contain at least one post-release fault, for each of the releases considered in this study

5.2 Description of metrics and the process of their extraction

In this study we use two types of software metrics - change metrics, which include the number of pre-release bug fixes, and static code metrics. These metrics were collected at file level and then aggregated to the package level for our analysis. This section details the techniques we used to gather, combine, and aggregate the metrics to achieve our final data set. Each of these metrics is then treated as a feature, in the machine learning sense, when building and evaluating the predictive models.

5.2.1 Static code metrics

Static code metrics capture the information pertaining to the source code itself. They range from simple metrics, such as lines of code (LOC), to metrics that measure structural intricacy, such as cyclomatic complexity. Static code metrics can be gathered via a variety of software tools available online, in part for this reason, they are often used in fault prediction studies. For this study, we used the freeware code analysis tool SourceMonitor[3] [1] to extract the static code metrics.

The list of the gathered static code metrics and their brief descriptions are given in Table 5.2. To gather these metrics, we downloaded the source code from the Eclipse CVS repository. First, batch files were generated to download the code for each set of packages for which we already had change metrics. In these batch files, the exact date of the release to be downloaded was specified in the CVS commands to ensure retrieval of the proper versions of the code. We then created XML files to automate and guide the code analysis performed by SourceMonitor. The result was a text file containing twenty-two static code metrics for every file in each release under consideration.

5.2.2 Change metrics

Change metrics quantify the alterations made to a source code file over the course of its existence. Change metrics used in this study were collected previously and used for classification of fault-proneness at the file level in [27]. Specifically, for each file we extracted the same set of seventeen change metrics as in [37]. Table 5.3 briefly describes these change metrics, while detailed descriptions can be found in [37].

Next, we provide a brief description of the change metrics extraction process. The version control system (CVS, in the case of Eclipse) maintains timestamped log files detailing the history of changes made to any given source code file, such as which developers made contributions and what changes they made. To extract the change metrics we mapped the CVS log entries to the bug database at a file level. For the releases 2.0, 2.1, and 3.0, as in [54], we searched the CVS log data for strings of four and five digits that matched the bug IDs. For the later releases, we used six-digit strings to match bug IDs. This process was manually validated to ensure that all metrics containing the word “bug” were captured by this pattern match.

[3]The complexity measure used by SourceMonitor approximately follows the definition by McConnell in [32].

Table 5.2: A list of static code metrics and their descriptions

Static Code Metric	**Description**
LOC	Total number of lines
Statements	Any *LOC* terminated by ‘;’
Percent Branch Statements	Percentage of statements causing a break in sequential execution, e.g., if, for, try, throw
Method Call Statements	All method calls, in statements and in logical expressions
Percent Lines with Comments	Percentage of comment lines
Classes and Interfaces	Total number of classes and interfaces, including anonymous inner classes
Methods per Class	Total method count divided by the total class count
Ave Statements per Method	Total number of statements found inside of methods divided by the number of methods
Max Complexity	Complexity value of the most complex method
Ave Complexity	Sum of all method complexity values divided by the number of methods
Max Block Depth	Maximum nested block depth level found within each method, starting at block level zero for each file. Depths up to 9 are recorded and all statements at deeper levels are counted as depth 9.
Ave Block Depth	Sum of all method block depths divided by the number of methods
Statements at Block Level x	Total number of statements in all methods contained at block level x, where $x \in (0, 1, 2, \dots, 9)$

It should be noted that change metrics include the metric *Bugfixes*, which represents the number of times a file was involved in pre-release bug fixes. Extracting the *Refactorings* metric followed Moser’s approach in [37], which entailed tagging all log entries containing the word “refactor”. We calculated the *Age* metric by noting the timestamp of the first occurrence of each file name in all CVS log data since 2001. We used the CVSPS tool [31] to determine changeset size, i.e., the files committed along with the file in question. Ensuring that the file names produced in the changesets included the path information matching the file names produced by our rlog processing script required slight modifications to the tool.

We did a few modifications to the log script to ensure that the data collected from various input sources were compatible and mapped accurately. For example, files marked as “dead” in Eclipse project are often moved to the Attic in CVS, which results in an alteration of the

Table 5.3: A list of change metrics and their descriptions

Change Metric	**Description**
Revisions	Number of revisions made to a file
Refactorings	Number of times a file has been refactored
Bugfixes	Number of times a file was involved in bug-fixing (pre-release bugs)
Authors	Number of distinct authors who made revisions to the file
LOC Added	Sum over all revisions of the number of lines of code added to the file
Max LOC Added	Maximum number of lines of code added for all revisions
Ave LOC Added	Average lines of code added per revision
LOC Deleted	Sum over all revisions of the number of lines of code deleted from the file
Max LOC Deleted	Maximum number of lines of code deleted for all revisions
Ave LOC Deleted	Average lines of code deleted per revision
Codechurn	Sum of (added lines of code - deleted lines of code) over all revisions
Max Codechurn	Maximum Codechurn for all revisions
Ave Codechurn	Average Codechurn per revision
Max Changeset	Maximum number of files committed together to the repository
Ave Changeset	Average number of files committed together to the repository
Age	Age of a file in weeks (counting backwards from a specific release)
Weighted Age	$\frac{\sum_{i=1}^{n} Age(i) \times LOC\ Added(i)}{\sum_{i=1}^{n} LOC\ Added(i)}$

file path. To account for this, we excluded from our study all instances that had the pattern "/Attic/" in their file paths. Another modification was due to the fact that when using the CVS rlog tool with date filtering, files that were unchanged during the filter period would be listed as having zero revisions, with no date, author, or other revision-specific information. This is true even if the file was previously marked "dead" on a branch. Therefore, the rlog for the entire file history was obtained and we determined the alive files and revisions which applied to each release, rather than examining only the date range required for each specific release.

5.2.3 Aggregation

We performed aggregation of all file-level metrics to the package level to achieve a coarser granularity, which offered a key advantage to our analysis. The vast majority of files in the various products we studied contained no post-release faults, making that data more suited to binary classification than numerical fault prediction. In addition, when viewed from the package level, the skewness of the distribution of post-release faults towards zero is much less pronounced. Such aggregations are not uncommon and have been performed and supported in the literature. For example, Zimmerman et al. found that it was better to classify Eclipse packages than files using regression models based on complexity metrics in [54]. Furthermore, in [12], D'Ambros et al. concluded that finer granularity has no effect on predictions made from change metrics.

Change and static code metrics were aggregated using the naming conventions of the Eclipse project, as in [54]. For example, the metrics for the file named:

org.eclipse.gef.Command.java

were combined with all of the class files named:

org.eclipse.gef.[Class name].java.

The aggregations were performed using the *Aggregate* function in IBM SPSS (v. 20.0), and the mean, median, maximum, and total were maintained for each metric, when appropriate. For instance, the file-level static code metric *LOC* after the aggregation became *Mean LOC*, *Median LOC*, *Max LOC*, and *Total LOC*, while the change metric *Ave Changeset* maintained only the mean value after the aggregation. The resulting data set contained a total of 112 metrics (i.e., features) for each package, with the total number of packages for each product by release listed in Table 5.1. Thus, each package was characterized by a vector $\mathbf{m}$ of 112 metrics, where $\mathbf{m}[i], i = 1, ..., 73$ are static code metrics, while $\mathbf{m}[i], i = 74, ..., 112$ are change metrics.

5.3 Machine learning approach

This section describes the data preprocessing steps, the background on the generalized linear regression models, our machine learning approach, including the feature selection method, and the performance metrics used to quantify the results.

5.3.1 Data preprocessing

In order to bring all attributes into equal ranges before creating regression models, we normalized the raw data prior to analysis. Normalization is a common practice in machine learning, see for example [6], [9], [23], [43], and [44]. For this study, we performed a logarithmic transformation of all metrics with very skewed distributions, i.e., *LOC*, *Statements*, *Method Call Statements*, *Classes and Interfaces*, *Statements at Block Level [0-9]*, *LOC Added*, and *LOC Deleted*. All other metrics were normalized using a min-max transformation, where each instance x of an attribute i is calculated according to

$$x' = \frac{max_{user} - min_{user}}{max_i - min_i}(x - min_i) + min_{user}, \tag{5.1}$$

where max_{user} and min_{user} are the user selected maxima and minima of the transformed values and max_i and min_i are the actual maximum and minimum values. For our analysis, we selected values of zero for min_{user} and one for max_{user}.

5.3.2 Background on Generalized Linear Regression models

For prediction of the number of post-release faults at the package level we used the generalized linear models (GLMs) introduced by Nelder and Wedderburn in [40] and later extended in [34], which are implemented in IBM SPSS (version 20.0). These models are of the general form given by

$$link(\gamma_{ij}) = \theta_j - [\beta_1 x_{i_1} + \beta_2 x_{i_2} + \ldots + \beta_p x_{i_p}], \tag{5.2}$$

where γ_{ij} is the cumulative probability of the jth category for the ith case, θ_j is the threshold for the jth category, p is the number of regression coefficients, $\beta_1, ..., \beta_p$ are regression coefficients, and $x_{i_1}, ..., x_{i_p}$ are the values of the predictors for the ith case.

The *link* function is a transformation of the cumulative probabilities that allows estimation of the model. For post-release fault prediction in software, we use the *link* function called the cumulative negative log log,

$$link(x) = -\log(-\log(x)), \tag{5.3}$$

because it is optimal for data sets in which lower values are more probable [33]. Since post-release fault distributions are typically skewed (i.e., have many packages with zero or small number of faults and a few packages with many faults), GLMs are more suitable to the task of software fault prediction than standard linear regression models. In GLMs two steps are required to attain a predicted value for a vector of metrics. First, the probabilities must be estimated for each possible value. Second, those probabilities must be used to select the most likely value for that vector. These steps are further described in [34].

5.3.3 Our machine learning approach

A software development or testing team can clearly benefit by using the models constructed from data acquired from the previous release of their product to predict which packages will likely exhibit the most post-release faults in the next release. Therefore, our approach mimics this actionable procedure for the development community.

Specifically, we built regression models for each product in each release of Eclipse from the aggregated and normalized data, which consists of 112 features as described in Section 5.2. The change metrics were gathered for a six month period before each release date, and static code metrics were extracted from the source code available on each release date. As a response variable for the predictive models, we used the number of faults in a package reported during a six months period after each products' release, referred to as post-release faults throughout the paper.

It should be emphasized that building a model on the n-th release to predict the post-release faults of the $n+1$-th release from the pre-release data of the $n+1$th release is an unbiased approach motivated by practical needs in software development. Many studies in the literature, whether limited by the size of their data sets or for other reasons, choose to employ some form of n-fold cross validation. Cross validation is the process of splitting the data randomly into k groups, and then predicting values for the i-th group by building a model on the other $k-1$ groups. This is repeated using each of the k groups as a testing group and the average value of the predicted variable is reported. Cross validation may provide better results than building models and predicting on disjoint data sets, however, the latter approach is more practical for the software engineering community. One possible reason is that during replication, there is a large overlap of data between the successive iterations, leading to training samples that are not independent. Furthermore, averaging out the results over many repeated trials can offer a more consistent and less extreme end result than one achieved via building models and predicting on disjoint sets.

We used the models built on each of the products in release n to predict the number of post-release faults in *every* product in release $n+1$. We built a total of nineteen predictive models – one model for each of the first three releases (which contained only one product each), and four for each of the next four releases (which contained four products each). Prediction models built for each product in each release (n) were then used to predict the number of post-release faults of all products in the subsequent release ($n+1$), which resulted in total of fifty-four trials. Testing the models on all products in the subsequent release, rather than on only the same product, is an approach taken for the first time. In this study, the goal is to explore whether the systematic reuse inherent in product lines and the addition of product line specific information can benefit the prediction of post-release faults.

5.3.4 Feature selection

In order to select the features that have the best predictive capability out of (typically) many available features we applied feature selection, a standard step used in machine learning. Reducing the number of features by removing the irrelevant and noisy features typically speeds up machine learning algorithms and improves their performance. The application of feature selection to reduce the set of features considered during model creation allowed us to use all available statistical measures of the aggregation, rather than a smaller subset as in [54] and [39].

In particular, we performed feature selection via stepwise regression [24] before creating the regression models. 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 response variable. Each step in the creation of the model eliminates the least significant feature, resulting in a smaller, more highly correlated feature set.

5.3.5 Performance metrics

We report four performance metrics, the *mean absolute error* (MAE), *nTop20%*, which is a normalized version of the percentage of total post-release faults found in the top 20% of packages predicted to be faulty, and the *rank correlation coefficients* Spearman's ρ and Kendall's τ-b. The latter measure the association between two ranked lists of packages – one based on the actual number of post-release faults and the other based on the predicted number of post-release faults.

Mean absolute error

MAE is a common evaluation metric which is calculated over the set of n predictions for each trial according to the equation:

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |f_i - y_i|, \tag{5.4}$$

where n is the number of predictions (the number of packages, in this case), f_i is the predicted number of post-release faults and y_i is the actual number of post-release faults in a package i.

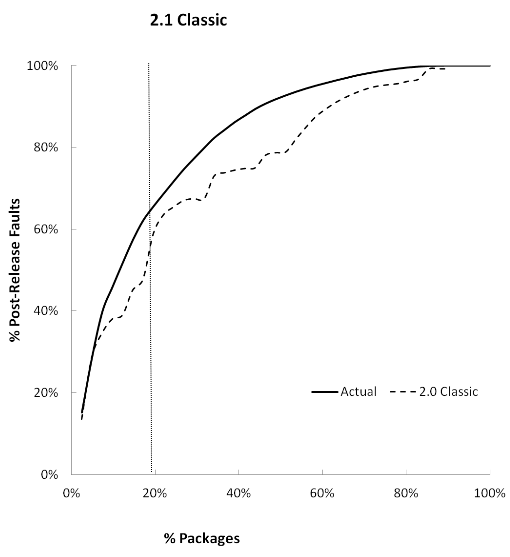
nTop20%

The percentage of actual faults found in the most faulty 20% of packages calculated by the predictive model is a common performance metric reported in several other works, e.g., [6], [13], [41], [43], [44], and [48]. To determine the nTop20% metric for our predictive models we used Alberg diagrams. The Alberg diagram is a standard way to show the relative accuracy of a set of predictions made by regression for software products [41], which provides a succinct manner of showing the ability of independent variables to rank a dependent variable [13].

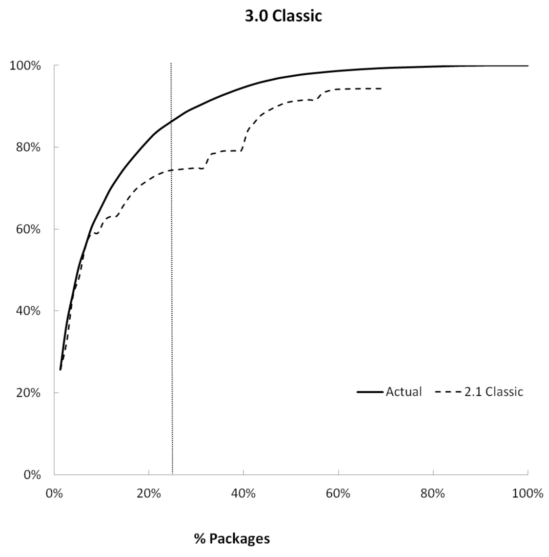
As an illustration we show two Alberg diagrams – in Figure 5.3(a) for release 2.1 and in Figure 5.3(b) for release 3.0 of Classic. The solid lines show the percentages of the cumulative number of post-release faults contained by packages, which are sorted on the horizontal axis in decreasing order by their *actual number of faults* in the corresponding release. (This means that the most faulty package is furthest to the left and the least faulty package in the release is furthest to the right). For the Alberg diagram in Figure 5.3(a) (Figure 5.3(b)) the dashed line shows the total number of actual faults located in the packages identified as the most faulty in release 2.1 (3.0) by the predictive model built using the data from the previous release 2.0 (2.1). Note that the *predicted number of faults* for each package of Classic 2.1 (3.0) based on the model built from the previous release of Classic 2.0 (2.1) is used to order the packages in decreasing order. Ordering the packages in this way provides a ranking of the packages which were predicted to be the most faulty by the model. The area between the dashed and solid lines shows how close the model's ranking comes to identifying the actual most faulty packages. It is important to note that we were conservative when plotting the predictive models' performances. Only packages with nonzero predicted fault values were ranked and plotted in dashed lines in our Alberg diagrams. This is the reason why the dashed lines in Figures 5.3(a) and (b) terminate before convergence with the solid line.

The vertical dotted line at 20% is a reference line used for measuring the effectiveness of a predictive model. From the figure, we see that for 2.1 Classic approximately 65% of the total post-release faults (solid line) were located in the top 20% of actual faulty packages. The top 20% of faulty packages identified by our predictive regression model contained approximately 60% of the total post-release faults.

Due to its dependence on the total number of faults residing in the top 20% of the software packages analyzed, and the fact that this number varies from product to product, the number of faults in the top 20% of the most faulty packages does not generalize well enough to allow comparisons between different products or even different releases. *To overcome this and support comparison, we used a normalized version of the Alberg measure consisting of the percentage of faults calculated by the predictive model found in the top 20% of packages, divided by the actual number of faults in the top 20% of most faulty packages.* For



(a) Predictions made for Classic release 2.1 using the model built on release 2.0



(b) Predictions made for Classic release 3.0 using the model built on release 2.1

Figure 5.3: Alberg diagrams showing the effectiveness of regression models (dashed lines) built on previous releases at predicting faults in the next release. The dotted vertical line marks the top 20% of the most faulty packages.

brevity, we refer to this normalized performance metric taken from Alberg diagrams as the *nTop20%* score for a model. In Figure 5.3, for 2.1 Classic the metric normalized in this way is nTop20% = 60/65 and it shows that the *predicted* top 20% of most faulty packages account for approximately 92% of the faults occurring in the *actual* top 20% of the most faulty packages.

We prefer *nTop20%* over other performance metrics, such as the coefficient of determination R^2, due to its ease of comparison across software projects combined with its robustness to outliers, as detailed in [41].

Rank correlation measured by the Spearman's ρ and Kendall's τ

The Alberg diagram basically shows two ranked lists of packages – one based on the actual number of post-release faults for each package (shown as a solid line) and another based on the predicted number of post-release faults for each package using the model built on the previous release (shown as a dashed line).

The association between two ranked lists is measured by Spearman's ρ and Kendall's τ correlation coefficients. The values returned by each metric range from -1 to 1, with lower values showing an indirect correlation, higher values indicating direct correlation, and values around zero representing no correlation between the two lists.

Spearman's ρ is the nonparametric version of Pearson's r correlation coefficient, which is used when the assumptions of normality and equal variance are not fulfilled or when the data are given in ordinal scale (i.e., data are comprised of ranks) as in this case. For a sample of size n of two variables X and Y, the differences in ranks on the two variables $d_i = X_i - Y_i$ are used as an indication of the disparity between the two sets of rankings. The Spearman's ρ is computed by:

$$\rho = 1 - \frac{6\sum_{i=1}^{n} d_i^2}{n(n^2-1)}. \tag{5.5}$$

Tied values among the ranks are handled by assigning them the average of their positions in the ascending order of the values. A full description of Spearman's ρ correlation coefficient is provided in [4].

Kendall's τ-b, also described in [4], is a variant of the Kendall's τ coefficient specifically designed to handle ties within the ranked lists. The Kendall τ-b coefficient is defined as:

$$\tau_b = \frac{n_c - n_d}{\sqrt{(n_0-n_1)(n_0-n_2)}} \tag{5.6}$$

where n_c is the number of concordant pairs, n_d is the number of discordant pairs, $n_0 = n(n-1)/2, n_1 = \sum_i t_i(t_i-1)/2, n_2 = \sum_i u_j(u_j-1)/2, t_i$ is the number of tied values in the i^{th} group of ties for the first quantity, and u_j is the number of tied values in the j^{th} group of ties for the second quantity.

Kendall's τ has several advantages[4] over the Spearman's ρ. Nevertheless, we used both metrics to measure the association of the ranked lists based on the actual and predicted number of post-release faults in order to be able to compare our results with the previous works which used the Spearman's ρ metric. It should be noted that when Spearman's ρ

[4]Kendall's τ approaches the normal distribution quite rapidly so that the normal approximation is better for Kendall's τ than it is for Spearman's ρ. Another advantage of Kendall's τ is its direct and simple interpretation in terms of probabilities of observing concordant pairs (both numbers of one observation are larger than their respective members of the other observation) and discordant pairs (the two numbers in one observation differ in opposite directions from the respective members in the other observation).

and Kendall's τ are both used on the same data, typically Spearman's ρ tends to be larger than Kendall's τ, in absolute value. However, as a test of significance both produce nearly identical results in most cases.

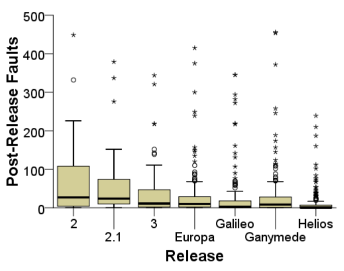
5.4 Assessment of the SPL quality

The research questions in this section assess the quality of Eclipse as a product line. First, we consider the distribution of post-release faults occurring in all products by release. Then, we consider the distribution of the post-release faults across the packages. Finally, we examine whether the quality of the products benefits from the reuse inherent in the product line.

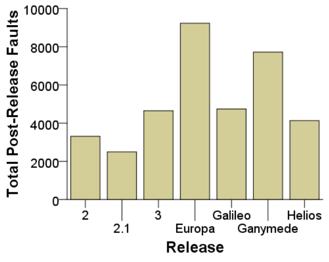
5.4.1 RQ 1: Does quality, measured by the number of post-release faults for the packages in each release, consistently improve as the SPL evolves?

Perhaps the most obvious measure of quality for any piece of software is the number of faults reported after the software's release. To discern trends in post-release faults we offer the three plots in Figure 5.4. The box plot in Figure 5.4(a) shows the median value and variance of post-release faults for the packages in each release. As Eclipse product line evolves through releases, there is a noticeable trend of decreasing median values, decreasing variances, and decreasing interquartile ranges in post-release faults.

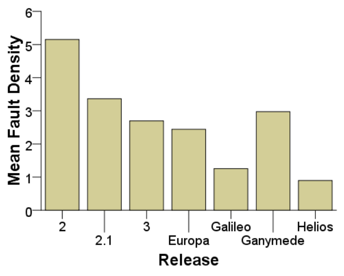
The bar graph in Figure 5.4(b) displays the total number of post-release faults for each release. As shown, the total number of post-release faults peaks in the Europa release, then follows a decreasing trend. This peak can be explained by the developmental changes that took place between release 3.0 and Europa. Between these two releases, the Classic standalone product split into multiple products as the Eclipse project first began to resemble a true product line. To accommodate this, many changes were made to the structure of the packages and to the source code itself. The shift brought about 128 new packages, and the source code more than doubled in size.



(a) A box plot of the distributions of post-release faults for each release



(b) A bar graph showing the total number of post-release faults for each release



(c) A bar graph of average post-release fault density per package, for each release

Figure 5.4: Plots showing total post-release fault data over all packages for each release

To account for this size increase, Figure 5.4(c) shows post-release faults normalized by size, i.e., the average number of faults for each package per one thousand lines of code. When the data is viewed in this light, the peak is seen in the first studied release and gradually declines to a lower value for the Helios release, with the exception of the Ganymede release. It is important to emphasize that despite the fact that three new products (i.e., C/C++, Java, and JavaEE) were introduced in the Europa release with 128 new packages, the post-release fault density of the Europa release still fits the decreasing trend.

To statistically confirm this observation we used the Kruskal-Wallis test, which determined that the distributions of the package post-release fault densities were not equal for Eclipse releases 2.0, 2.1, 3.0, Europa, Ganymede, Galileo, and Helios. This was followed by a post-hoc Jonckheere-Terpstra test, which rejected the null hypothesis in favor of the ordered alternative hypothesis that the fault density of each release was less than or equal to the fault density in the previous release ($p = 6.2 \times 10^{-22}$).

Additionally, the trend was confirmed by ranking all packages by their total number of post-release faults, numbering the releases chronologically from one to seven, and then computing Kendall's τ-b nonparametric rank correlation measure between the two ranked lists. The resulting value for τ-b was -0.238 with a p value of 6.2×10^{-22}. The negative correlation confirms the inverse relationship of post-release fault density and chronological releases, that is, **proves that the post-release fault density decreases as the product**

line evolves through releases. This strong evidence of an overall trend of decreasing post-release fault density shows that quality does improve as the SPL evolves.

5.4.2 RQ 2: Do the majority of faults reside in a small subset of packages?

To address this question, we examined what percentage of the total number of post-release faults is located in the top twenty percent of the most faulty packages, for each release of Eclipse considered in this study. Figure 5.5 shows a bar graph and descriptive statistics of the results. It follows that **from 66% to 93% of the post-release faults detected across all products in each release are located in approximately 20% of the packages, with average and median around 81% and 84%, respectively**.

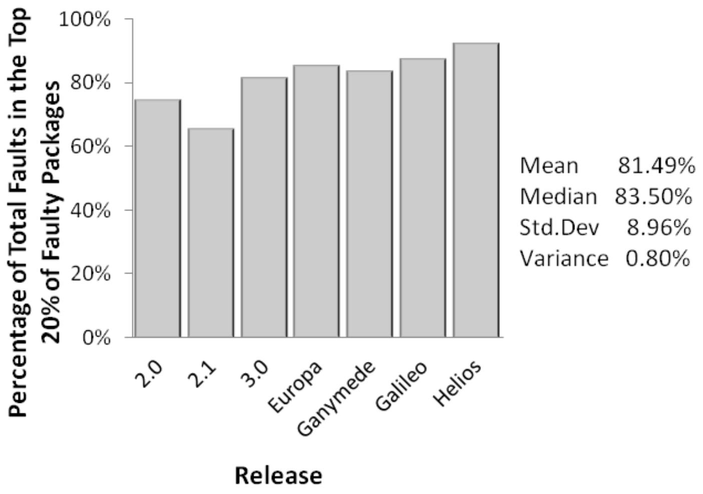


Figure 5.5: A bar graph depicting the percentage of total post-release faults detected in the top twenty percent of the total packages for each release, with accompanying descriptive statistics

This result generally agrees with other works [5], [8], [13], [18], [42], which have consistently found that between 60 and 90% of bugs normally reside in around 20% of the lines of code, files, or packages, depending on the unit. Furthermore, this result confirms that 20% is a good cut-off point for the most faulty packages used in the nTop20% performance metric.

5.4.3 RQ 3: Does the quality of products benefit from the reuse inherent in the product line?

A key goal of product line engineering is the application of structured reuse to achieve higher quality products [16], [46]. To evaluate how products benefit from this structured reuse, we examined the post-release fault densities of previously existing packages (third row in Figure 5.6) and newly developed packages (bottom row in Figure 5.6) in the Ganymede, Galileo, and Helios releases. (The Europa release was not considered because of the three years time difference from the previous considered release 3.0, which made the identification of newly developed packages impossible.) Note that the packages in each release are grouped by their level of reuse: single-use variation packages (used in only one product), low-reuse variation packages (used in two products), high-reuse variation packages (used in three products), and common packages (used in all four products).

Exploring the average Codechurn, which is shown in the first and second rows of Figure 5.6, showed that previously existing packages at all levels of reuse, including the common packages, continued to change. This suggests that rather than becoming stable over time in terms of lines of code, common packages acquire new functionality and must also adapt to coexist with newly introduced variation packages. Even though this introduction of new functionality and adaptation introduced new post-release faults, the fault density of the common packages remained fairly low, that is, they incurred a low number of faults for their size. In general, the third row of Figure 5.6 showed that **even though pre-existing packages had a relatively high average Codechurn, they sustained low post-release fault densities, which clearly illustrates the benefit of reuse.** This result is similar to our earlier findings reported for PolyFlow and those of Krishnan, et al. reported in [25].

For the newly introduced packages, shown in the last row of Figure 5.6, no clear trend of the post-release fault densities could be observed. In the Ganymede release the ten newly developed single-use variation packages of JavaEE had noticeably lower post-release fault densities than the six newly developed low-reuse variation packages, which were shared

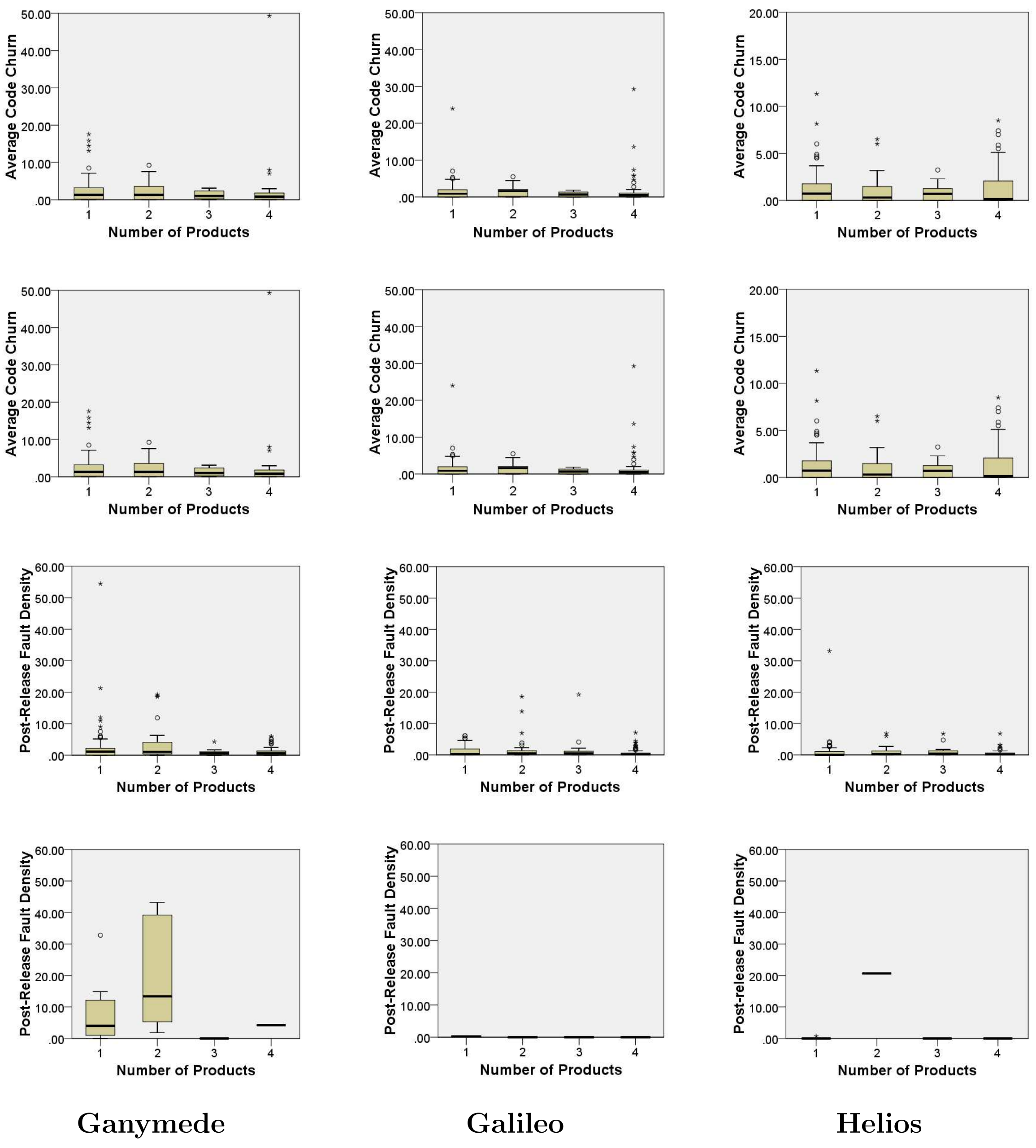


Figure 5.6: Bar graphs showing the average code churn and post-release fault densities of previously existing packages (first and third rows) and newly developed packages (second and last row) grouped by their level of reuse for releases Ganymede, Galileo, and Helios. 1, 2, 3, and 4 annotate single-use variation, low-reuse variation, high-reuse variation and common packages, respectively.

between Classic and JavaEE. The one common package had very few post-release faults. In the Galileo release, only one newly developed single-use variation package existed in our dataset; it had no post-release faults. Of the nineteen newly introduced packages in the Helios release, eighteen were single-use variation packages and one was a low-reuse variation package. The eighteen new single-use variation packages, which belong to JavaEE, received nearly twice the amount of average Codechurn as the newly introduced low-reuse package, but contained only two post-release faults in total. Basically, the new single-use variation packages had the highest average Codechurn and the lowest fault density. In contrast, the low-reuse package, which was shared by Java and JavaEE, had over a thousand post-release faults and the highest post-release fault density.

In summary, it appears that **newly developed low-reuse variation packages tend to have higher post-release fault densities than single-use variation packages and the common package.** However, we hesitate to make strong conclusions with respect to the post-release fault density of the newly developed packages due to the small sample sizes, especially when compared to the sample sizes of the previously existing packages[5]

5.5 Prediction of post-release faults from pre-release data

This section discusses findings related to the predictive ability of models generated according to the methods described in Section 5.3. Basically, in each release we built a model on each product and then used it to predict the number of post-release faults in each product of the subsequent release from the corresponding pre-release features (i.e., metrics). This process resulted in building nineteen models which were then used for prediction in 54 trials. The Alberg diagrams for each trial are shown in Figures 5.3 and 5.7. The solid lines in each diagram represent the actual percentage of post-release faults, while the dashed lines represent the performance of each predictive model constructed from a product of the pre-

[5]In the Ganymede, Galileo, and Helios releases the number of previously existing packages per level of reuse (single-use variation, low-reuse variation, high-reuse variation, and common) were (74, 32, 16, 61), (83, 27, 16, 61), and (83, 27, 16, 61), respectively.

vious release, as labeled in the legend at the bottom of the figures. The vertical dotted lines represent the 20% cut off point for ease of reference as described in Section 5.3.5.

The heat maps in Figure 5.8 show the resulting performance metrics recorded from the 54 combinations of model-building and model-evaluation products described in Section 5.3.3. In these heat maps, the release on which the predictions were made are labeled on the bottom. The models were built on the previous release. The columns represent the products on which predictive models were *built*. Columns are labeled at the top by product name. The rows represent the products on which the predictive models were *evaluated*, and are labeled on the left. Each value in the heat map tables represents a performance metric score achieved by evaluating the predictions made on the release of the row-labeled product by the model built on the previous release of the column labeled product. For each metric, the better results are shaded darker. Figures 5.3, 5.7 and 5.8 are used to illustrate the observations related to the predictions of the post-release faults based on pre-release data.

5.5.1 RQ 4: Can we accurately predict the packages which will contain a high percentage of the total post-release faults from pre-release data?

To address this question, we refer to the results presented in Figure 5.8(a). It can be observed that the best nTop20% prediction for each product across all releases is in the range of 76% to 97%, which indicates that **a high percentage of the most faulty packages post-release within a product can be consistently predicted from the packages' pre-release data**.

Even more, the nTop20% scores for all releases remain at high levels, not only for the best model, but in general. The only exceptions are two groups – the first full column of Galileo predictions (made by models built on C/C++ data in Ganymede release), and the last three cells of the bottom two rows in the Helios predictions (predictions made for Java and JavaEE). Examination of the raw data and Alberg diagrams shown in Figure 5.7 showed that these two groups of particular models performed worse than others due to an outlier package in each case, as detailed next.

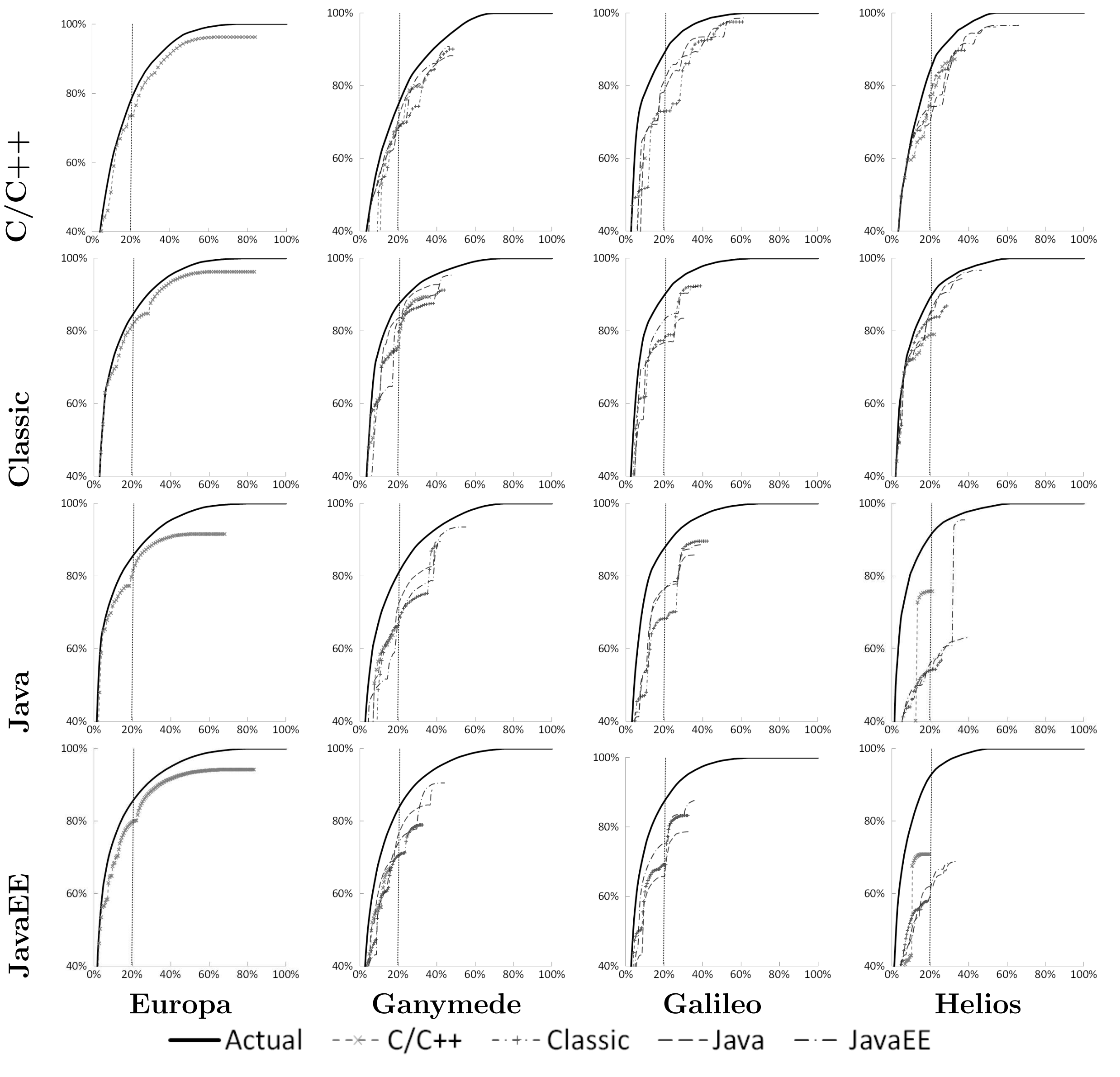
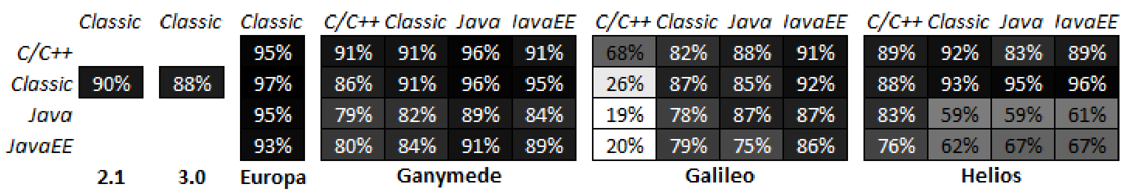
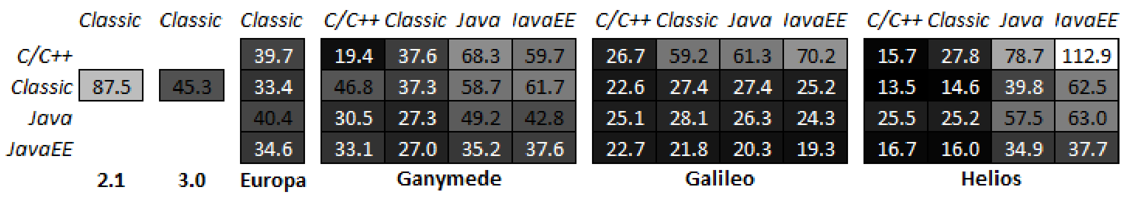


Figure 5.7: Alberg diagrams for each product's predictive trials in the final four studied releases



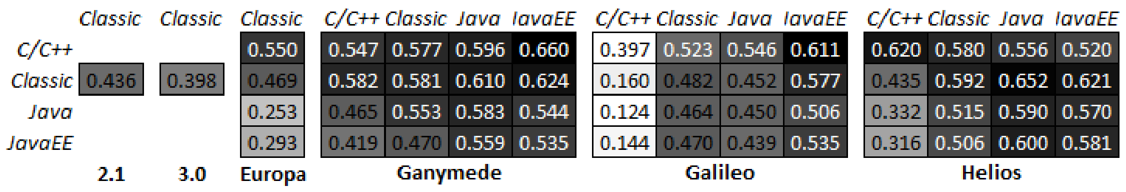
	2.1	3.0	Europa	Ganymede				Galileo				Helios			
	Classic	Classic	Classic	C/C++	Classic	Java	JavaEE	C/C++	Classic	Java	JavaEE	C/C++	Classic	Java	JavaEE
C/C++			95%	91%	91%	96%	91%	68%	82%	88%	91%	89%	92%	83%	89%
Classic	90%	88%	97%	86%	91%	96%	95%	26%	87%	85%	92%	88%	93%	95%	96%
Java			95%	79%	82%	89%	84%	19%	78%	87%	87%	83%	59%	59%	61%
JavaEE			93%	80%	84%	91%	89%	20%	79%	75%	86%	76%	62%	67%	67%

(a) A heat map of the values for the performance measure **nTop20%**. *Higher* values represent better results and are shaded darker.



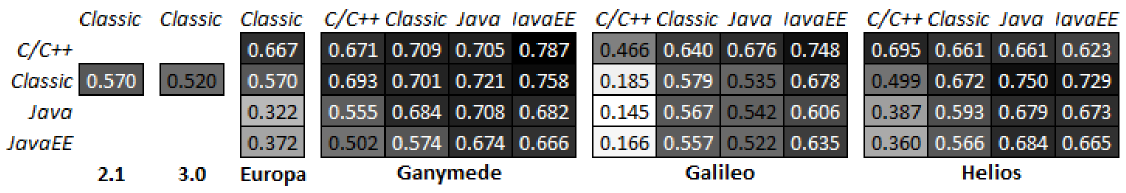
	2.1	3.0	Europa	Ganymede				Galileo				Helios			
	Classic	Classic	Classic	C/C++	Classic	Java	JavaEE	C/C++	Classic	Java	JavaEE	C/C++	Classic	Java	JavaEE
C/C++			39.7	19.4	37.6	68.3	59.7	26.7	59.2	61.3	70.2	15.7	27.8	78.7	112.9
Classic	87.5	45.3	33.4	46.8	37.3	58.7	61.7	22.6	27.4	27.4	25.2	13.5	14.6	39.8	62.5
Java			40.4	30.5	27.3	49.2	42.8	25.1	28.1	26.3	24.3	25.5	25.2	57.5	63.0
JavaEE			34.6	33.1	27.0	35.2	37.6	22.7	21.8	20.3	19.3	16.7	16.0	34.9	37.7

(b) A heat map of the values for the performance measure **MAE**. *Lower* values represent better results and are shaded darker.



	2.1	3.0	Europa	Ganymede				Galileo				Helios			
	Classic	Classic	Classic	C/C++	Classic	Java	JavaEE	C/C++	Classic	Java	JavaEE	C/C++	Classic	Java	JavaEE
C/C++			0.550	0.547	0.577	0.596	0.660	0.397	0.523	0.546	0.611	0.620	0.580	0.556	0.520
Classic	0.436	0.398	0.469	0.582	0.581	0.610	0.624	0.160	0.482	0.452	0.577	0.435	0.592	0.652	0.621
Java			0.253	0.465	0.553	0.583	0.544	0.124	0.464	0.450	0.506	0.332	0.515	0.590	0.570
JavaEE			0.293	0.419	0.470	0.559	0.535	0.144	0.470	0.439	0.535	0.316	0.506	0.600	0.581

(c) A heat map of the values for the performance measure **Kendall's** τ. *Higher* values represent better results and are shaded darker.



	2.1	3.0	Europa	Ganymede				Galileo				Helios			
	Classic	Classic	Classic	C/C++	Classic	Java	JavaEE	C/C++	Classic	Java	JavaEE	C/C++	Classic	Java	JavaEE
C/C++			0.667	0.671	0.709	0.705	0.787	0.466	0.640	0.676	0.748	0.695	0.661	0.661	0.623
Classic	0.570	0.520	0.570	0.693	0.701	0.721	0.758	0.185	0.579	0.535	0.678	0.499	0.672	0.750	0.729
Java			0.322	0.555	0.684	0.708	0.682	0.145	0.567	0.542	0.606	0.387	0.593	0.679	0.673
JavaEE			0.372	0.502	0.574	0.674	0.666	0.166	0.557	0.522	0.635	0.360	0.566	0.684	0.665

(d) A heat map of the values for the performance measure **Spearman's** ρ. *Higher* values represent better results and are shaded darker.

Figure 5.8: Heat maps showing the results of the predictive trials for each product, in each release. Releases on which the predictions were made are shown on the bottom. The rows represent the products on which the predictive models were *evaluated* and are labeled on the left by the product name. The columns represent the products on which predictive models were *built*, and are labeled at the top by product name. Each value in the table represents a performance metric score achieved by evaluating the predictions made on the release of the row-labeled product by the previous release of the column labeled product.

Models built from the Ganymede version of C/C++ provided uncharacteristically poor predictions for all products in the Galileo release (see the first column of Galileo release in Figure 5.8(a)). In fact, the model built on C/C++ data predicted only two faulty packages for every product other than itself, neither of which was among the top 20% of the most faulty packages in the Galileo release. This is represented by the low values of predictions from the models built on C/C++ (shown with a dashed line) at the vertical 20% line in the Alberg diagrams for Galileo Classic, Java, and JavaEE shown respectively in the second, third, and fourth row of the third column in Figure 5.7.

Upon closer examination, we saw that Ganymede's release of C/C++ had a much different distribution of faults than the other products in any release. As shown in Figure 5.9, most products showed a similar distribution of faults, with one very faulty package steadily followed by several more packages with a relatively high number of faults, then a small spread of the majority of packages having very few faults. However, as the boxplot in Figure 5.9 shows, Ganymede's C/C++ had one very faulty package and a large difference with the next most faulty package. This lack of packages between the most faulty packages and those with relatively few faults created an "all or nothing" effect in the predictive model.

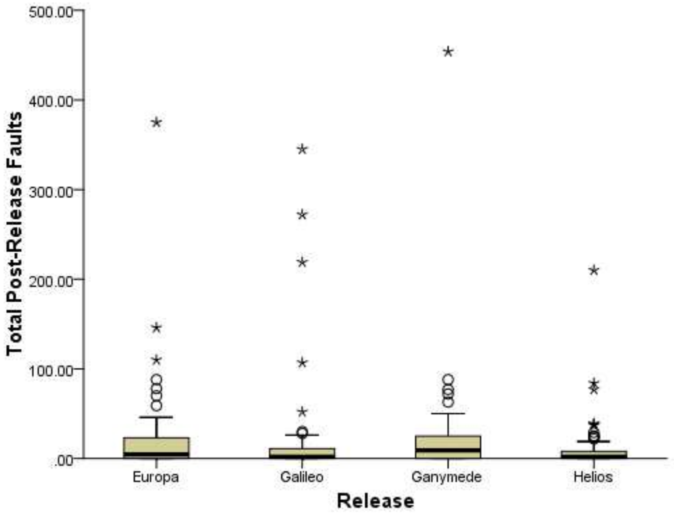


Figure 5.9: A boxplot of the distributions of post-release faults of C/C++ across releases. Notice the one extremely faulty package in the Ganymede release compared to the other packages in that release

Predictions for the Helios versions of Java and JavaEE made by the model built from the Galileo versions of Classic, Java, and JavaEE are also worse than others (see the last table in Figure 5.8(a)). The data revealed that the most faulty package in the Helios release had 1,021 total post-release faults and it was a low-reuse variation package shared by Java and JavaEE. This package, which comprised 36% and 25% of the total faults for the Helios release of Java and JavaEE, respectively, was not identified as one of the top 20% of faulty packages by the three specified models. In the Alberg diagrams for Java and JavaEE in the Helios release (shown in the last column of the third and fourth rows in Figure 5.7) this is indicated by the low values at the 20% vertical line followed by large vertical jumps for the predictive models built on Classic, Java, and JavaEE of the Galileo release. At a closer look it appeared that this particular package was not identified among the top 20% of the most faulty packages in the Helios release because it had different values of the two main features common to the predictive models of Classic, Java, and JavaEE (i.e., the values for both *Authors* and *Bugfixes* were much lower).

5.5.2 RQ 5: Do predictions of the number of faults for each package improve as each product evolves through releases?

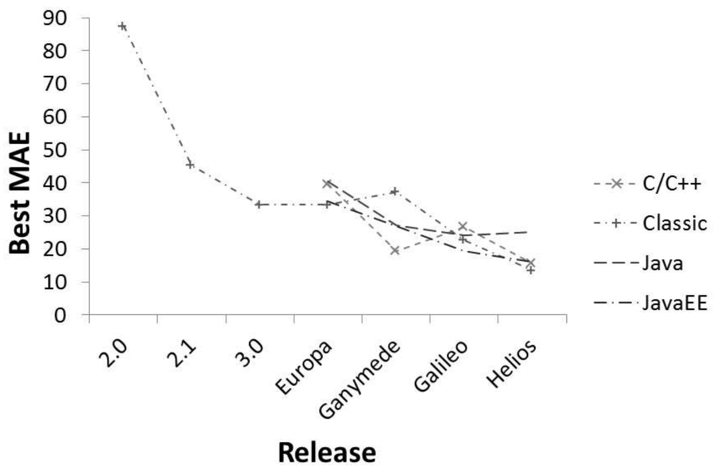


Figure 5.10: A graph of trend lines showing the best MAE score for each product across releases examined in this study.

We showed in RQ 1 of Section 5.4.1 that the quality, measured by the number of post-release faults for the packages in each release, improved as the product line evolved through releases. Here we consider the related question of whether predictive models built on the products also improve. For this purpose, we focused on the MAE results shown in Figure 5.8(b) and observed a **fairly steady reduction in the lowest mean absolute error for each product from the first release to the final release considered in this study**.

This trend is illuminated more fully in the line graph shown in Figure 5.10. The lines clearly show a decreasing trend of the mean absolute error for each product. This leads to the conclusion that the predictions of the number of post-release faults for each package do improve as each product evolves through releases.

5.5.3 RQ 6: Do the predictions of the most fault prone packages (i.e. nTop20%) benefit from additional product line information?

As shown in Figure 5.8(a), the best predicted values of nTop20% are typically not found on the main diagonal, i.e., where the predictive model for a product is constructed from the data of the same product in the previous release. This suggests that fault predictions for a member of a SPL can benefit from using information from other members of the family. We speculate that this benefit comes from the reuse inherent in the structure of SPLs. To test this, when examining the values within the tables in Figure 5.8(a), we focus on exploring row trends and column trends.

First, we note that *row patterns* show trends in which particular products have consistently good predictions made by models built from any product. Specifically, we make several interesting observations. *C/C++ and Classic show a strong row consistency, i.e., no matter on which product from the previous release a model is constructed, it will consistently make accurate predictions for C/C++ and Classic.* As shown in Figure 5.1, these two products are comprised mostly of common code and are the smallest of the four products. The only exception to this trend is the prediction made in Galileo release by the model built from the Ganymede release of C/C++, which resulted in poor predictions across the board due to an outlier package, as discussed in RQ 5.

Second, *column patterns* emerge when the models built by a particular product have consistent results regardless of the product on which they are evaluated. Thus, *Java and JavaEE show a strong column consistency, i.e., predictions made on any product by models built from these two products are consistently very good.* These two products are the largest in terms of source code and packages, and also express the most variability. The only exceptions are the predictions made on the Helios versions of Java and JavaEE, which suffered from a very large outlier, as discussed in RQ 5. Based on these observations, we conclude that **models produced the best results when built from larger products with more variation (non-shared packages), and when making predictions on smaller products with less variation (more shared pakages).** This is evidence that prediction of the most fault prone packages benefits from additional product line information.

5.5.4 RQ 7: Can we accurately rank the packages based on the predicted numbers of post-release faults? Does the accuracy of the fault-based rankings of the software packages benefit from additional product line information?

In addition to the nTop20% metric, we used the models to rank the packages based on the predicted value of the number of post-release faults. The associations of these ranked lists of packages with ranked lists based on the actual number of post-release faults were then used as performance metric. The values of the Kendall's τ and Spearman's ρ are shown in Figures 5.8(c) and 5.8(d), respectively.

In general, as discussed in Section 5.3.5, the associations measured by Spearman's ρ are higher than by Kendall's τ. Although Kendall's τ has some advantages for measuring the association of two ranked lists, we presented the Spearman's ρ values as well in order to be able to compare our results with the results presented in related work. Specifically, the Spearman's ρ values shown in Figure 5.8(d) compare favorably to values achieved in similar prediction attempts for Eclipse in related work. D'Ambros et al. examined two components of Eclipse from releases 3.1 and 3.3 in [10]. Using 50-fold cross validation of regression models based on the static code metric *Fanout* and change coupling metrics, Spearman correlation ρ values ranged from 0.4 to 0.810. The best values were achieved when correlating selected change coupling metrics with number of faults, however, the paper is not specific as to whether pre- or post-release faults are studied. In [11], D'Ambros and Robbes evaluated many different prediction approaches on several components of Eclipse versions 3.1, 3.4, and 3.4.1. The Spearman's ρ values attained when using the same change metrics as us ranged from 0.165 to 0.534 and, when using comparable static code metrics, ranged from 0.277 to 0.547.

More related results were presented by Zimmerman et al. in [54]. They used logistic regression models built upon static code metrics on Eclipse Classic releases 2.0 and 2.1 to predict the post-release faults for releases 2.1 and 3.0, respectively (among other combinations). The obtained values for Spearman's ρ were 0.420 for the 2.1 predictions and 0.449 for the 3.0 predictions. Our results for these same combinations of Classic releases are better (i.e., 0.570 and 0.520, respectively) as shown in Figure 5.8(d), which could be due to the addition of change metrics to our features and/or to the different machine learning technique we employed.

Next, we explore whether these rankings, which are based on predictions of the number of post-release faults for packages, benefit from additional SPL information. As shown in Figures 5.8(c) and 5.8(d), while some of the best values for Kendall's τ and Spearman's ρ appeared on the main diagonals, at least three fourths of best rankings for both performance metrics were made by models built from previous releases of other products. As in RQ 6, this suggests that post-release fault rankings for products in a product line may benefit from additional information from other members of the family.

When examining the results for values of Kendall's τ and Spearman's ρ, row and column patterns also emerged. As it was the case for the nTop20% performance metric, both Classic and C/C++ displayed a strong row consistency. Furthermore, Java and JavaEE showed a strong column consistency with the ranking performance metrics Kendall's τ and Spearman's ρ, as they did with the nTop20% metric. (The only exceptions are the slightly lower values by the model built on the Ganymede release of Java and evaluated on the Galileo versions of Classic, Java, and JavaEE.)

In summary, the finding for the nTop20% metric in RQ 6 is confirmed here by the accuracy of the fault-based ranking – **models produced the best results when built from larger products with more variation (non-shared packages), and when making predictions on smaller products with less variation (more shared pakages).**

5.5.5 RQ 8: Are some features better indicators of the number of post-release faults in a package than others?

Feature selection is a standard machine learning technique used to identify the features that are the best predictors for a response variable. As described in Section 5.3.4, we performed feature selection via stepwise regression. For each model, the selection method incrementally determined the set of features that was most strongly correlated to the response variable. The number of features selected for each of the 19 models created in this study ranged from 1 to 16 (see the histogram shown in Figure 5.11) with a mean value of 9.6, which indicates that **a small number of features are sufficient to predict the number of post-release faults from pre-release data.**

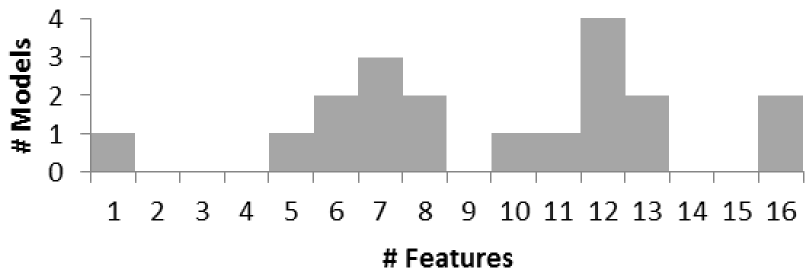


Figure 5.11: A histogram of the number of features selected for each of the 19 models used in this study

To determine the most often selected features across the set of all models, we created the histogram in Figure 5.12. The histogram shows the number of models in which each feature was used, in descending order. To further assess the ability of different features (i.e., change and source code metrics) we computed the Spearman correlation ρ value for each feature pairwise with the actual value of the response variable, in every release. (We used

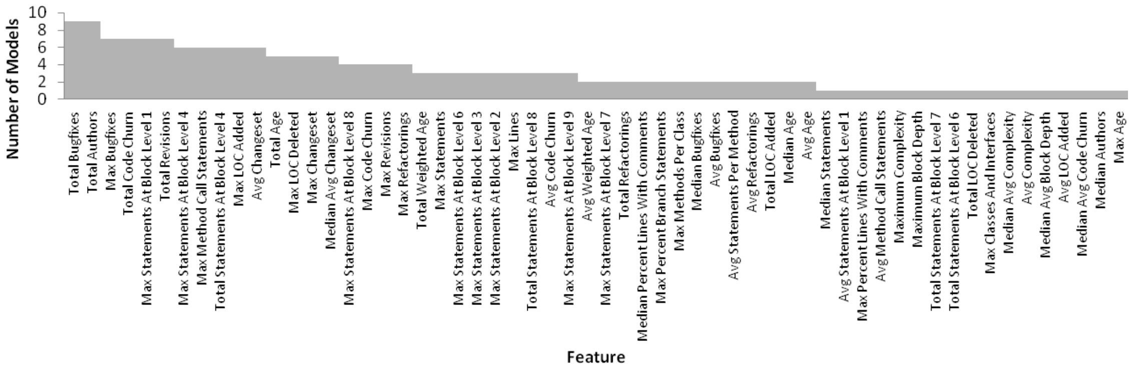


Figure 5.12: A histogram of the number of times each feature was selected by stepwise regression among the top predictors for the 19 models used in this study.

the nonparametric Spearman correlation because the data did not conform to the normal distribution.) The correlation table is not given due to space limitations, but certain values are discussed in support of the interpretations of the results.

The results showed that no single feature or set of features was consistently the best predictor for every model. The top two most commonly selected features in this study, which were selected in just under half (47%) of the 19 models, were both change metrics: total *Bugfixes* and total *Authors*. Furthermore, the Spearman correlation ρ values for these two features with the response variable were high for every release, with mean values over all releases of 0.768 and 0.726, respectively. These high positive correlation values indicate that *post-release faults are often located in packages that have both a high frequency of pre-release faults and a relatively high number of authors who made revisions.*

The high correlation of the number of pre-release faults (i.e., *Bugfixes*) with post-release faults contradicts the findings of Fenton and Ohlssen in [13], but it is consistent with the results of the replicated studies performed on different software systems by Andersson et al. in [5] and Grbac et al. in [19]. The fact that this result is consistently supported in several very large, complex software systems offers some measure of generality to the conclusion that **the most faulty code before release tends to be among most faulty code after release.** With respect to total *Authors*, consistent with our results, post-release faults have been shown to be related to more contributing authors by Giger et al. in [15]. However, no consistent relationship between number of authors and post-release faults was indicated in the work of Weyuker et al. in [52].

As Figure 5.12 displays, total *CodeChurn* and total *Revisions* were among other most often selected change metrics, with high mean correlations across all releases (i.e., Spearman ρ values of 0.741 and 0.751, respectively.) These results are in agreement with Nagappan and Ball, who found in [38] that *Average CodeChurn* had a statistically significant, very high positive correlation to fault density (p-value < 0.01, $\rho = 0.883$) and concluded that *an increase in change metrics for a package is often accompanied by an increase in post-release faults for that package.*

The fact that total *Bugfixes*, total *Authors*, maximum *Bugfixes*, total *CodeChurn* and total *Revisions* all had high frequencies of occurrence in predictive models and high correlations with the response variable, implies that **post-release faults are closely associated with the total number of times the source code in a package was fixed during pre-release testing, the number of authors contributing to a package, the amount of source code added to and deleted from a package before that package's release, and the total number of times the package was revised**. These results from a large and widely used family of software products may offer actionable suggestions for members of the software development industry interested in pinpointing packages likely to be faulty from historical data.

Of the fifteen features appearing in more than a quarter of models, only four were static code metrics: maximum *Statements at Block Level 1*, maximum and total *Statements at Block Level 4*, and the total *Method Call Statements.* These four metrics also have relatively high mean Spearman correlation ρ values, ranging from 0.610 for the total statements at block level four to 0.683 for the total number of method call statements. The results for these metrics indicate that **post-release faults are also associated with large amounts of source code in nested blocks at low and intermediate depths, and with many method calls**.

We also examined the metrics *LoC* and *Cyclomatic Complexity*, two static code metrics which are widely explored in the literature on fault prediction. We found that *LoC* was also correlated to total post-release faults with statistically significant ρ values ranging from 0.635 to 0.796 and an average ρ value of 0.692. However, it was selected among top predictors only for a small subset of the models. This correlation is slightly stronger than the results from our study of PolyFlow, and those reported by Andersson et al. in [5] and by Zimmerman et al. in [54]. With respect to *Cyclomatic Complexity*, our results from both this study and the study of PolyFlow agreed with Fenton and Ohlssen in [13] that it is a poor predictor of faults, as it was selected as an important predictor in only one of the 19 models. However, as in the case study of PolyFlow and [54], we did note a positive correlation between *Maximum Complexity* and number of post-release faults of $\rho = 0.453$. Compared to our results, Zimmerman et al. in [54] showed slightly higher level of correlation for this metric.

5.6 Threats to validity

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

Construct validity addresses whether we are testing what we intended to test. An obvious and prevalent construct validity threat is insufficiently defining constructs before translating them to metrics. Inconsistency and imprecision of terminology are significant threats to validity in software quality assurance which can complicate comparisons of results across studies. We were careful to provide the definitions of all terms and metrics used in this thesis and to avoid ambiguous or inconsistently used terms, such as defects, that are often used differently throughout the literature.

Mono-operation bias to construct validity occurs when the cause-construct is under-represented. Many empirical studies use limited data, that is, are missing type of data that could help explain the cause-effect relationships better. For example, many fault prediction studies were based on using only static code metrics. Throughout this study we used both static code and change metrics, allowing feature selection method to choose the features that are the best predictors.

Typically, a common step in creating regression models is to filter the data to remove outliers. We did not remove outliers in order to maintain the relevance of this study to actual software development. In software quality assurance, it is often the case that some files and packages have significantly more post-release faults than others, which was confirmed in this study as well. The distributions of many of the metrics we gathered are also skewed. For instance, in the Helios release the largest package, *org.eclipse.jdt.core*, has 431 KLOC, which is significantly higher than the mean value of 20 KLOC over the entire release. In addition, this is the second most faulty package in that release and therefore it is one of the main targets of our search. In general, for skewed distributions such as the distribution of the number of post-release faults across software packages, it is most important to identify the packages at the tail of the distribution. Therefore, even though excluding outliers may have led to more accurate predictions, no data were excluded from our datasets.

Internal validity threats concern influences that, without researchers' knowledge, can affect the independent variables and measurements. The biggest threat to internal validity is data quality.

In the process of collecting the static code metrics, we made great efforts to assure that the data we collected was as complete and accurate as possible. However, as we were dealing with massive downloads of source code files that have been archived for close to a decade, locating every single file was unrealistic. Not all source code files for which CVS logs were recorded were available to anonymous developers at the Eclipse CVS repository server[6]. In particular, no code was available for any files in the package CDT. This package includes C/C++ development tools and would have been the single use variation package for the C/C++ product. Additionally, the source code for the Ganymede and Galileo releases of PDE (Plug-in Development Environment) were unavailable. For consistency with respect to our predictive models, the unavailability of the PDE source code in the final two releases prompted us to omit the PDE packages when creating the predictive models for the Ganymede release. In total, for the named releases (i.e., Europa to Helios) we were able to locate archived source code for 91.6% of the 136,567 total files for which CVS data was retrieved.

While source code with metrics for the 2.0, 2.1, and 3.0 releases of Classic is available online from [54], we chose to collect and analyze the source code directly from the CVS repository in order to ensure repeatability. We were able to download source code for only 18,111 of the 41,416 files in these older releases. While this number seems low, it is compara-

[6]pserver:anonymous@dev.eclipse.org:2401/cvsroot

ble to amount of data collected by Zimmerman et al. in [54]. In particular, we downloaded the available data for 2.0, 2.1, and 3.0 releases from the repository given in [54] and found that their dataset consists of 15,395 files.

We built our dataset by combining static code metrics with change metrics collected by Krishnan et al. in [27]. This means that our data quality is dependent upon the quality of that data. As described briefly in Section 5.2.2, there were several obstacles to collecting a complete set of change logs for all files in every release of Eclipse, including "dead" files moved to the CVS repository's Attic. These difficulties, as well as how they were overcome and how the data set was validated, are fully detailed in [27], and we are confident in the quality of the data.

Conclusion validity concerns the ability to draw correct conclusions. Using statistical tests in cases where their assumptions are violated is the most obvious threat to conclusion validity. As our data did not conform to the normal distribution, we analyzed our results using nonparametric tests, such as the Kruskal-Wallis test and the post-hoc Jonckheere-Terpstra test for the trend of the post-release faults across releases. Due to the skewness of the feature distributions and the response variable distribution, we used the Spearman rank correlation coefficient to assess the correlation of individual features with the response variable (i.e., number of post-release faults). In addition, for the association of the ranked lists ordered by predicted and actual number of post-release faults we used the Spearman and Kendall's τ-b correlation coefficients, which have minimal assumptions. We also used appropriate versions of the statistics for datasets with many ties.

External validity concerns the generalizability of results. It is impossible for research based on one case study to claim that its results would be valid for other studies. However, some of the observations made in this thesis have some degree of external validity because they are consistent across both case studies in this thesis. Whenever possible, we also compared our results to the findings of other studies, both of Eclipse and other software systems. We have given complete details of how our study was performed so that it may be replicated in the future. Finally, we provided definitions of the features and performance metrics used in this study, which can be easily used to fairly compare our results with the results of future case studies.

Chapter 6

Conclusion

Software product line (SPL) engineering is a paradigm for systematic reuse that is widely used to develop high-quality, diverse software product families faster and with less cost than traditional development methods. Real world case studies of SPLs are necessary both to empirically validate the advantages of product line engineering and to facilitate the development process by supplying actionable insights into how SPLs behave in practice as well as ways to improve SPL engineering. This thesis described two separate empirical studies– the first of pre-release software faults in an industrial software product line (PolyFlow) and the second of post-release faults in a large, evolving, open source software product line (Eclipse). Conclusions drawn from these studies are presented separately in this chapter, followed by a list of unified results from both studies.

6.1 PolyFlow

This section describes the results of our empirical study of pre-release software faults in PolyFlow, a medium-sized industrial software product line. In this case study we collected and examined both change and static code metrics for 42 different components comprised of approximately 65,000 lines of code.

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.

Our research exploring the aspects of components with varying levels of reuse in PolyFlow indicates that components used in only one SPL product are the most likely to change and have the highest fault densities of any level of reuse. Common components reused in all four products had fault densities close to those used in three products and higher Average Code Churn.

The results of our longitudinal study of PolyFlow suggest that later products 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 gradual 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 two 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 unique components, see Figure 4.1), 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. We also saw, however, that even systematic reuse as in software product lines often results in introduction of changes to accommodate requirements from different products 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.

6.2 Eclipse

The results presented in this section are based on an empirical study of Eclipse, a mature and well documented open-source SPL with a wide and diverse user base. Our examination included both static metrics derived from the source code and change metrics extracted from the CVS repository logs. The data, collected over the course of seven releases for four products, include over 135,000 files containing 20 million lines of code, aggregated into packages and described by 112 different static code and change metrics.

The quality assessment results found that post-release faults are mostly located in a small percentage of the total packages. Furthermore, as the product line continued to evolve through releases, post-release fault densities at a package level showed a decreasing trend. Reused packages continued to exhibit a high degree of change throughout the releases, yet retained low fault densities.

For predicting the number of post-release faults we used generalized linear regression models, with cumulative negative log log linking functions. These are particularly well-suited for modeling skewed distributions, such as post-release fault data. The predicted values for the number of post-release faults were then used to select the top 20% of most fault-prone packages. The predicted values were also used to correlate the two ranked lists of packages – one based on the actual number of post-release faults, another based on the predicted number of post-release faults.

The results showed that models built from the data of one release could accurately predict the number of post-release faults and the most fault prone packages from the pre-release data of the subsequent release. Furthermore, rankings created by our models were positively correlated to the actual rankings. In addition to experiencing improved quality (i.e., a decreasing trend of the number of post-release faults), the accuracy of predictions measured by the mean absolute error steadily improved as the SPL evolved through releases. Perhaps the most surprising finding is that the best predictive models for each product were built from the pre-release data that included other products. This means that the predictions benefited from the use of additional product line information. Specifically, models built from larger products with more variability typically produced better predictions than models built on the smaller products, which mainly consisted of common packages. Furthermore, all models achieved their best results when making predictions on smaller products.

Last but not least, our results showed that a small number of features are sufficient to produce accurate predictions. In general, pre-release change metrics are better predictors of post-release faults than source code metrics. Specifically, post-release faults were closely associated with the total number of times the source code in a package was fixed during pre-release testing, the number of authors contributing to a package, the amount of source code added to and deleted from a package before it was released, and the total number of times the package was revised. Out of the fifteen most frequently selected features, only four were static code metrics. Specifically, the results indicated that post-release faults were associated with large amounts of source code in nested blocks at low and intermediate depths, and with many method calls.

6.3 Consistent observations across both empirical studies

Several important results were consistent across both of the empirical case studies presented in this thesis. While more replications and new studies of more SPLs are needed, having consistent results manifested across two different SPLs provides a certain measure of generality to the conclusions. The following results were found to be consistent between these two SPLs:

- There is a wide spectrum of different levels of reuse, from common packages shared among all products, to high-reuse variation and low-reuse variation packages shared among some, but not all products, to single-use variation packages used in only one product.
- Both pre-release faults and post-release faults have skewed distributions, that is, most of the faults are contained in a small set of components/packages.
- Pre-existing components/packages, including the common components/packages, continuously change (i.e., have high average code churn), but tend to sustain low fault

densities. This is not always true for newly developed components/packages.

- Both pre-release faults (Polyflow) and post-release faults (Eclipse) are more highly correlated with change metrics than with static code metrics.
- Predictions of pre-release faults (in the case of Polyflow) and post-release faults (in the case of Eclipse) can be done accurately from pre-release data. Even more, predictions benefit from additional product line information.

References

[1] (2011) Sourcemonitor version 3.2. [Online]. Available: campwoodsw.com/sourcemonitor.html

[2] (2013) About the Eclipse foundation. [Online]. Available: www.eclipse.org/org

[3] (2013) The IEEE website. [Online]. Available: www.computer.org/portal/pages/seportal/subpages/sedefinitions.html

[4] A. Agresti, *Analysis of Ordinal Categorical Data.* Hoboken, NJ, USA: Jon Wiley and Sons, Inc., 2010.

[5] C. Andersson and P. Runeson, "A replicated quantitative analysis of fault distributions in complex software systems," *IEEE Transactions on Software Engineering*, vol. 33, pp. 273–286, May 2007.

[6] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Looking for bugs in all the right places," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA '06. New York, NY, USA: ACM, 2006, pp. 61–72.

[7] S. Bibi, G. Tsoumakas, I. Stamelos, and I. Vlahvas, "Software defect prediction using regression via classification," in *Proceedings of the IEEE International Conference on Computer Systems and Applications*, ser. AICCSA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 330–336.

[8] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Computer*, vol. 34, pp. 135–137, January 2001.

[9] A. E. Camargo Cruz and K. Ochimizu, "Towards logistic regression models for predicting fault-prone code across software projects," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 460–463.

[10] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, ser. WCRE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 135–144.

[11] ——, “An extensive comparison of bug prediction approaches,” in *7th IEEE Working Conference on Mining Software Repositories*, ser. MSR ’10, may 2010, pp. 31 –41.

[12] ——, “Evaluating defect prediction approaches: a benchmark and an extensive comparison,” *Empirical Software Engineering*, vol. 17, pp. 531–577, 2012, 10.1007/s10664-011-9173-9.

[13] N. E. Fenton and N. Ohlsson, “Quantitative analysis of faults and failures in a complex software system,” *IEEE Transactions on Software Engineering*, vol. 26, pp. 797–814, August 2000.

[14] W. B. Frakes and G. Succi, “An industrial study of reuse, quality, and productivity,” *Journal of Systems and Software*, vol. 57, pp. 99–106, June 2001.

[15] E. Giger, M. Pinzger, and H. Gall, “Using the gini coefficient for bug prediction in eclipse,” in *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, ser. IWPSE-EVOL ’11. New York, NY, USA: ACM, 2011, pp. 51–55.

[16] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures.* Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[17] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, “A systematic review of fault prediction performance in software engineering,” *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, 2011.

[18] M. Hamill and K. Goseva-Popstojanova, “Common trends in software fault and failure data,” *IEEE Transactions on Software Engineering*, vol. 35, pp. 484–496, 2009.

[19] D. Huljenic, P. Runeson, and T. G. Grbac, “A second replicated quantitative analysis of fault distributions in complex software systems,” *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, pp. 1–1, 2012.

[20] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, “Revisiting common bug prediction findings using effort-aware models,” in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[21] Y. Kastro and A. B. Bener, “A defect prediction method for software versioning,” *Software Quality Control*, vol. 16, no. 4, pp. 543–562, Dec. 2008.

[22] T. Khoshgoftaar and J. Munson, “Predicting software development errors using software complexity metrics,” *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 2, pp. 253 –261, feb 1990.

[23] B. Kitchenham and E. Mendes, “Why comparative effort prediction studies may be invalid,” in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, ser. PROMISE ’09. New York, NY, USA: ACM, 2009, pp. 4:1–4:5.

[24] D. G. Kleinbaum, L. L. Kupper, and K. E. Muller, Eds., *Applied regression analysis and other multivariable methods.* Boston, MA, USA: PWS Publishing Co., 1988.

[25] S. Krishnan, R. R. Lutz, and K. Goseva-Popstojanova, "Empirical evaluation of reliability improvement in an evolving software product line," in *8th Working Conference on Mining Software Repositories*, ser. MSR '11, 2011, pp. 103–112.

[26] S. Krishnan, C. Strasburg, R. R. Lutz, and K. Goseva-Popstojanova, "Are change metrics good predictors for an evolving software product line?" in *7th International Conference on Predictive Models in Software Engineering*, 2011, pp. 7:1–7:10.

[27] S. Krishnan, C. Strasburg, R. R. Lutz, K. Goseva-Popstojanova, and K. S. Dorman, "Predicting failure-proneness in an evolving software product line," *Information and Software Technology*, no. 0, pp. –, 2012.

[28] C. W. Krueger. (2012) Introduction to software product lines. [Online]. Available: http://www.softwareproductlines.com/introduction/introduction.html

[29] P. L. Li, J. Herbsleb, M. Shaw, and B. Robinson, "Experiences and results from initiating field defect prediction and product test prioritization efforts at ABB Inc." in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 413–422.

[30] W. Lim, "Effects of reuse on quality, productivity, and economics," *IEEE Transactions on Software Engineering*, vol. 11, no. 5, pp. 23 –30, 1994.

[31] D. Mansfield. (2012) Cvsps-patchsets for cvs. [Online]. Available: http://www.cobite.com/cvsps

[32] S. McConnell, *Code Complete, Second Edition.* Redmond, WA, USA: Microsoft Press, 2004.

[33] P. McCullagh, "Regression models for ordinal data," *Journal of the royal statistical society. Series B (Methodological)*, pp. 109–142, 1980.

[34] P. McCullagh and J. Nelder, *Generalized Linear Models*, ser. Monographs on Statistics and Applied Probability. Chapman and Hall, 1983.

[35] P. Mohagheghi, R. Conradi, O. Killi, and H. Schwarz, "An empirical study of software reuse vs. defect-density and stability," in *26th International Conference on Software Engineering*, ser. ICSE '04, May 2004, pp. 282 – 291.

[36] P. Mohagheghi and R. Conradi, "An empirical investigation of software reuse benefits in a large telecom product," *ACM Transactions on Software Engineering Methodology*, vol. 17, pp. 13:1–13:31, June 2008.

[37] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *ACM/IEEE 30th International Conference on Software Engineering*, ser. ICSE '08, may 2008, pp. 181 –190.

[38] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *27th International Conference on Software Engineering*, ser. ICSE '05, 2005, pp. 284–292.

[39] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *28th International Conference on Software Engineering*, ser. ICSE '06, 2006, pp. 452–461.

[40] J. A. Nelder and R. W. M. Wedderburn, "Generalized linear models," *Journal of the Royal Statistical Society. Series A (General)*, vol. 135, no. 3, pp. pp. 370–384, 1972.

[41] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886–894, Dec. 1996.

[42] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02, 2002, pp. 55–64.

[43] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 86–96.

[44] ——, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340 – 355, April 2005.

[45] ——, "Programmer-based fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '10. New York, NY, USA: ACM, 2010, pp. 19:1–19:10.

[46] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[47] R. Selby, "Enabling reuse-based software development of large-scale systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 495 – 510, June 2005.

[48] Y. Shin, R. Bell, T. Ostrand, and E. Weyuker, "Does calling structure information improve the accuracy of fault prediction?" in *6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09, May 2009, pp. 61 –70.

[49] W. M. Thomas, A. Delis, and V. R. Basili, "An analysis of errors in a reuse-oriented development environment," *Journal of Systems and Software*, vol. 38, pp. 211–224, September 1997.

[50] D. M. Weiss and C. T. R. Lai, *Software product-line engineering: a family-based software development process.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[51] D. M. Weiss, J. J. Li, H. Slye, T. Dinh-Trong, and H. Sun, “Decision-model-based code generation for SPLE,” *International Software Product Line Conference*, pp. 129–138, 2008.

[52] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, “Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models,” *Empirical Software Engineering*, vol. 13, no. 5, pp. 539–559, 2008.

[53] W. Zhang and S. Jarzabek, “Reuse without compromising performance: Industrial experience from RPG software product line for mobile devices,” 2005.

[54] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for Eclipse,” in *3rd International Workshop on Predictor Models in Software Engineering*, ser. PROMISE ’07, Washington, DC, USA, 2007, pp. 9–.