**Automated Software Engineering manuscript No.**
(will be inserted by the editor)

# Assessment and cross-product prediction of SPL quality: accounting for reuse across products, over multiple releases

**Thomas Devine · Katerina Goseva-Popstajanova · Sandeep Krishnan · Robyn R. Lutz ·**

Received: date / Accepted: date

**Abstract** The goals of cross-product reuse in a software product line (SPL) are to mitigate production costs and improve the quality. In addition to reuse across products, due to the evolutionary development process, a SPL also exhibits reuse across releases. In this paper, we empirically explore how the two types of reuse - reuse across products and reuse across releases - affect the quality of a SPL and our ability to accurately predict fault proneness. We measure the quality in terms of post-release faults and consider different levels of reuse across products (i.e., common, high-reuse variation, low-reuse variation, and single-use packages), over multiple releases. Assessment results showed that quality improved for common, low-reuse variation, and single-use packages as they evolved across releases. Surprisingly, within each release, among preexisting ('old') packages, the cross-product reuse did not affect the change and fault proneness. Cross-product predictions based on pre-release data accurately ranked the packages according to their post-release faults and predicted the 20% most faulty packages. The predictions benefited from data available for other products in the product line, with models producing better results (1) when making predictions on smaller products (consisting mostly of common packages) rather than on larger products and (2) when trained on larger products rather than on smaller products.

**Keywords** software product lines · cross-product prediction · cross-product reuse · cross-release reuse · longitudinal study · assessment · fault proneness prediction

T.Devine
Lane Department of Computer Science and Electrical Engineering
West Virginia University
E-mail: tdevine4@mix.wvu.edu

K. Goseva-Popstajanova
Lane Department of Computer Science and Electrical Engineering
West Virginia University
E-mail: Katerina.Goseva@mail.wvu.edu

S. Krishnan
Department of Computer Science
Iowa State University
E-mail: sandeepk@iastate.edu

R. Lutz
Department of Computer Science
Iowa State University
E-mail: rlutz@iastate.edu

## 1 Introduction

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 (Gomaa, 2004). Weiss and Lai (1999) define a SPL as, "a family of products designed to take advantage of [their] common aspects and predicted variabilities". The goal of SPL engineering is to develop diverse families of software products and software-intensive systems in shorter time, at lower cost, and with higher quality (Pohl et al, 2005). While there have been several studies showing the benefits of systematic reuse across applications (Lim, 1994; Thomas et al, 1997; Frakes and Succi, 2001; Selby, 2005), empirical studies exploring these benefits in the context of SPLs are lacking.

In this paper we present a longitudinal study of the quality of products in a SPL. The study is based on a subset of the Eclipse family of products, which has been previously studied by Chastek et al (2007) and van der Linden (2009) as a SPL. The central concepts of SPL engineering are: (1) the use of a collection of reusable artifacts and (2) in order to provide mass customization (Pohl et al, 2005). Eclipse demonstrates the first concept in its management and reuse of both common and variable components across products. Eclipse demonstrates the second concept in its introduction of new products customized to the needs of its various user communities.

Specifically, our study examines four products from the Eclipse project, Classic, C/C++, Java, and JavaEE, through multiple releases. In the early releases Eclipse was developed and used as a single platform for providing tools to aid software developers. The platform could be customized to an individual developer's interests by incorporating specific tool suites as plugins. Starting with the release codenamed Europa, Eclipse evolved from a single, all-encompassing product into a true product line by providing separate products which were already customized to specific user requirements. These products contain shared code reused across multiple products. It appears that reuse across products in both open source and industrial SPLs is not 'all or nothing'. Rather, while some components are reused across all products (so called commonalities), other components are reused only in a subset of products referred to as high-reuse variation and low-reuse variation components in this paper. Products in a SPL also have single-use components that are used only in one product.

It is important to note that a SPL exhibits two types of reuse, as illustrated in Figure 1. *Reuse across releases*, represented on the x-axis in Figure 1, exists in all software systems developed in a release-oriented fashion and represents the evolution of an individual product across releases, as new functionality is added or improvements are made over time. *Reuse across products*, represented by the y-axis in Figure 1, is typical for SPLs and represents the reuse of individual packages in two or more products within the same release.

The first part of our study focuses on assessing how reuse, both across releases and across products, affects the quality of a SPL. For this purpose, we focus on four distinct Eclipse products longitudinally across seven releases and measure quality in terms of the number of post-release faults[1]. We specifically explore different levels of reuse across products, over multiple releases, which provides SPL developers with insights into the utility of product lines and how the two dimensions of reuse affect the quality of SPL products. With respect to the *quality assessment* of the SPL we address the following research questions:

RQ1: Does quality, measured by the number of post-release faults for the packages in each release, consistently improve as the SPL evolves across releases?

RQ2: Do packages at different levels of reuse across products mature differently across releases? Does the quality of products benefit from reusing packages in multiple members of the SPL?

[1] A fault is defined as an accidental condition, which if encountered, may cause the system or system component to fail to perform as required. We avoid using the term defect, which is used inconsistently in the literature to refer in some cases to both faults and failures and in other cases only to faults or perhaps, faults detected pre-release.

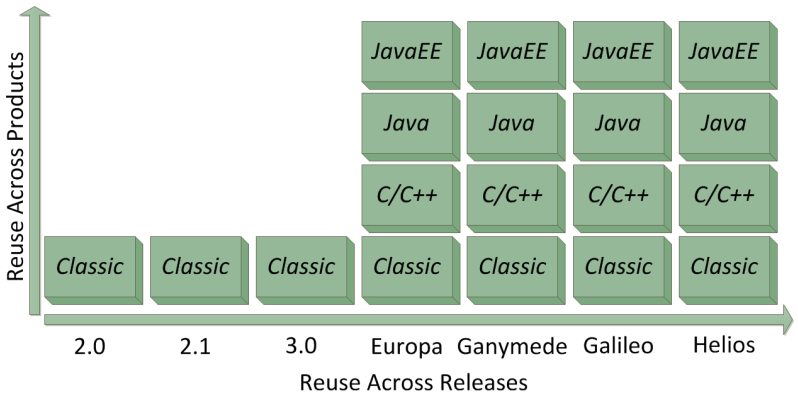



Fig. 1: An illustration of the two types of reuse (i.e., reuse across releases and reuse across products) in Eclipse

The second part of our study focuses on prediction of post-release faults in members of the product line using models built on previous releases. Specifically, we built generalized linear regression models from each individual member of the product line family, then used them to rank the packages in each product in the subsequent release by the number of post-release faults they are likely to contain. This cross-product prediction approach allowed us to explore whether the predictions for individual products benefit from available data for other members of the product line. This is a unique aspect of our work, which is especially important because it allows predicting the post-release fault proneness of new emerging products in the SPL, before they are released. The following research questions are devoted to the *prediction* of post-release faults from pre-release data:

RQ3: Can we accurately predict which packages will contain a high percentage of the total post-release faults from pre-release data using models built on the previous release?

RQ4: Do the predictions of the most fault prone packages benefit from the data available for other products? In other words, do cross-product predictions improve the accuracy?

RQ1 can be viewed as conceptual replications of similar research question explored in non-SPL context by several works (Fenton and Ohlsson, 2000; Ostrand and Weyuker, 2002; Ostrand et al, 2004; Khoshgoftaar and Seliya, 2004), and in only one study of a much smaller SPL (Mohagheghi and Conradi, 2008). Replicated studies, both exact and conceptual, are vital to empirical software engineering because they enable the software engineering community to explore the conditions required to obtain specific results and to determine the external validity of results (Shull et al, 2008). RQ2 is relevant only in a SPL context. The first part of this question furthers the preliminary investigation of our previous work (Krishnan et al, 2011a) by explicitly exploring different levels of cross-product reuse over multiple releases, including statistical tests of significance. The second part of RQ2 was not explored previously, including our previous work (Krishnan et al, 2011a). RQ3 and RQ4, which are focused on the effects of the reuse across products on cross-product predictions of fault proneness, are explored for the first time in this paper.

The main contributions of this study are:

– In addition to *reuse across releases*, which is typical for any evolving software system, we explored *reuse across members of a SPL* and how these two different types of reuse affect products' quality and our ability to make accurate predictions. *A unique characteristic of this work is our focus on the way packages are grouped in individual products, which allows us to make predictions for emerging products, even before they are released for the first time. Thus, we were able to predict the fault proneness of the products C/C++, Java, and Java EE, which were introduced in the Europa release of Eclipse (see Figure 1), based on the model built on the product Classic from the previous release, 3.0.*

- The *assessment results* showed that as the product line continued to evolve through releases, previously existing (i.e., 'old') common packages, high-reuse variation packages and low-reuse variation packages remained prone to changes; only single-use packages experienced decreasing change proneness as they evolved, perhaps because they were not affected by changes made to other products. *'Old' common packages, low-reuse variation packages, and single-use packages improved in quality (measured by their post-release fault density) as they evolved across releases, despite exhibiting significant change proneness.* Previously existing high-reuse variation packages, however, did not exhibit an improvement in quality. *Surprisingly, the level of cross-product reuse (i.e., common, high-reuse variation, low-reuse variation and single-use) did not affect the change and fault proneness of 'old' packages within each release.* Newly developed low-reuse variation packages tended to have higher post-release fault densities than either single-use packages or the common package, but the sample size of newly developed packages was too small to draw strong conclusions.
- *A novel aspect of this research is the exploration of the benefit of using data from other products in the SPL in support of fault proneness predictions.* Specifically, we built models from the individual products in each release and then used these models to make predictions for each product in the subsequent release. Overall, we built 15 models, which were used to make cross-product predictions for a total of 54 combinations of products and releases.
- The predictions were based on a *generalized linear regression model* with an ordered multinomial distribution and cumulative negative log-log linking function, which is specifically appropriate for skewed distributions characterized by higher probabilities of lower or zero values, as is typical for the distributions of post-release faults across software units (i.e., files, components, or packages). Models built on the previous release were used to predict the post-release fault proneness of the following release. This approach mimics the actual data collection process and thus has more practical value than using cross-validation or bootstrapping. These models were used to *predict the 20% most faulty packages*, as well as to *rank software packages based on the number of post-release faults they contain.* Compared to binary classification of packages as fault-prone or not, this type of prediction conveys more information that is useful for determining effort required to repair faulty packages, which in turn may allow for more efficient allocation of verification and validation resources.
- The prediction results showed that models built from the data of one release could accurately predict the most fault prone packages in a subsequent release from that release's pre-release data. Furthermore, rankings of fault prone packages created by our models were positively correlated to the actual rankings. The most interesting finding from the product line perspective is that *the best predictive models for each product were built from pre-release data that included other products. This means that the predictions benefited from the use of data available for other products.* 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.
- We synthesized the findings of this study with the observations made in our previous work based on a smaller industrial SPL with goal of identifying the trends, both in assessment and prediction of SPL quality, that are invariant across multiple product lines.

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 describes the Eclipse product line case study. Section 4 defines our metrics and discusses the process of their extraction, while Section 5 details our machine learning approach for creating and evaluating predictive models. The results on assessment of product line quality are presented in Section 6, followed by the results of the predictive analysis in Section 7. Section 8 offers a synthesis of the results from this study and our previous work based on an industrial SPL. Section 9 describes the threats to validity, and Section 10 provides a summary of the main results and concluding remarks.

## 2 Related Work

We first summarize the related work on numerical fault prediction not in SPLs. Then, we present prior work on assessment and prediction (including classification and numerical prediction) in the context of SPLs. We end the section with a summary of the main contributions of this paper.

### 2.1 Numerical prediction of post-release software faults

Several works in the literature have constructed and tested numerical models for fault prediction, with the aim of predicting 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[2].

Of these papers, four have used Eclipse as a case study (D'Ambros et al, 2009, 2010; Kamei et al, 2010; Zimmermann et al, 2007). It should be noted that none of them considered the SPL aspects of Eclipse. Rather, they analyzed collections of files and/or packages in several releases of Eclipse. In particular, D'Ambros et al (2009) used generalized linear models to explore the utility of change coupling metrics for predicting post-release faults and to compare predictive techniques on four different Eclipse components (D'Ambros et al, 2010). Both D'Ambros et al (2009) and D'Ambros et al (2010) used $n$-fold cross validation within a single data set to arrive at their final results. Kamei et al (2010) used linear regression, regression tree, and random forest models to predict post-release faults in three components of Eclipse. Experimental data showed that fusion performed after 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 of the data and the model was tested on the other half, and by building models on one release and predicting on the next. Zimmermann et al (2007) 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 all three releases.

Numerical, post-release fault prediction studies of other software products not related to Eclipse include (Bibi et al, 2006; Kastro and Bener, 2008; Khoshgoftaar and Munson, 1990; Li et al, 2006; Nagappan et al, 2006; Ostrand et al, 2004, 2005, 2010; Bell et al, 2006; Weyuker et al, 2008; Shin et al, 2009).

Bibi et al (2006) compared twelve different models to determine the benefits of regression via classification. Results were validated using $n$-fold cross validation. Static code metrics were combined with change metrics by Kastro and Bener (2008) to create neural network prediction models for Linux. Khoshgoftaar and Munson (1990) used complexity metric features selected by stepwise regression or factor analysis to compare linear regression models which predicted fault densities. Li et al (2006) 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. The models were constructed from source code, change, deployment, and usage metrics. Nagappan et al (2006) 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 model 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 (2004) and Ostrand et al (2005) built models from file level information on LOC, number of previous faults, and change metrics. Bell et al (2006) compared the predictive ability of several negative binomial models built using different combinations of LOC and change metrics on the data extracted from an automated

[2] For a comprehensive survey of binary classification studies the reader is referred to the recent paper by Hall et al (2012).

voice response system. A primary focus of the work by Weyuker et al (2008) was to investigate the impact of the number of developers to the accuracy of predictions. The results showed that the metrics related to the number of developers led to "no more than a modest improvement in the predictive accuracy". Ostrand et al (2010) also used negative binomial regression model and explored whether including information about individual developers can be used to improve the predictions. The results showed that the individual developer's past performance was not an effective predictor of future bug locations. Shin et al (2009) used different combinations of LOC, static code metrics, change metrics, faults from previous releases, and calling structure information to construct negative binomial regression models. It appeared that the addition of calling structure information to a model based solely on non-calling structure code attributes provided noticeable improvement in prediction accuracy, but only marginally improved the best model based on history (i.e., change) and non-calling structure code attributes.

### 2.2 Assessment and prediction in a SPL context

Large, industrial product lines rarely provide data for academic research. To bypass this problem, Zhang and Jarzabek (2005) developed four members of a mobile gaming product line, both simultaneously using a SPL architecture and developing each product independently. 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.

Mohagheghi et al. examined data from two products of a large telecom product line (Mohagheghi et al, 2004; Mohagheghi and Conradi, 2008). Subsystems and blocks (each consisting of multiple source code files) were considered as components in these studies. The components reused in the two distinct products were developed in-house and reused as-is. The data used in these studies covered three years of development of the reused (i.e., common) components in the two products and the variable components of product 1. (Variable components of product 2 were not available to the research team.) The empirical analysis showed that within each release the fault density of reused subsystems was lower than the fault density of non-reused ones, but the sample size was too small to perform statistical tests. The results at block level were consistent – within each release reused blocks had lower fault density than non-reused ones, and the result was statistically significant. In this case study, neither the number of faults nor the fault density experienced reduction over time (that is, across the three releases considered in the paper).

In our previous work (Devine et al, 2012) we presented an empirical study of pre-release faults in a medium-size, industrial product line. Specifically, we studied four products of the SPL PolyFlow. These four products collectively consisted of 42 components totaling approximately 65,000 LOC. The study characterized the fault and change proneness at various levels of reuse and explored the benefits of SPL development to newly developed products, both in improved quality and in the ability to predict pre-release faults. The results showed that single-use components had the highest fault density and were the most prone to change. We also found that the number of pre-release faults in variation components of new products could be accurately predicted using stepwise linear regression models built on data from the previous products. It should be noted that post-release faults were not available for this industrial case study (Devine et al, 2012).

Some of our earlier works were also based on Eclipse viewed as a SPL. First, we analyzed change metrics (e.g., new files per component) and post-release fault data from four releases (i.e., Europa, Ganymede, Galileo, and Helios) of the Eclipse project (Krishnan et al, 2011a). In that work, the analysis was done at the component level (i.e., at a coarser level of granularity than package level) for severe faults (i.e., Eclipse's blocker, critical, and major categories). Components were grouped based on the level of reuse across the product line into common components (a total of six components), high-reuse variation components (a total of seven components), and low-reuse variation components (a total of three components). Single-use variation components were

not considered. The results showed that commonalities followed a decreasing trend in file churn through subsequent releases and exhibited fewer severe post-release faults. Conversely, both high-reuse and low-reuse variation components exhibited a high degree of change as the SPL evolved through releases and had mixed behavior with respect to the severe post-release faults. Note that components at different levels of reuse were not compared within each release in (Krishnan et al, 2011a). Also, the small sample sizes at component level prevented us from using statistical tests.

Then, in (Krishnan et al, 2011b) we classified Eclipse files as fault-prone or not fault-prone using change metrics as features and the J48 decision tree algorithm. 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, suggesting that classification in a SPL might be useful.

In the follow-up work (Krishnan et al, 2012) we studied multiple learners for classification and compared 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. It should be noted these papers (Krishnan et al, 2011b, 2012) conducted classification (rather than numerical predictions) on collection of Eclipse files and did not investigate cross-product predictions as is done in this paper.

### 2.3 Contributions of this paper

The empirical results presented in this paper are based on 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. This information totaled 125,118 files containing over 20 million lines of code.

A major difference of this study from the related work on analysis and prediction of fault proneness is the fact that in addition to reuse across releases we considered reuse across products in the SPL. For this purpose our analysis considered *multiple products* from the Eclipse product line rather than *a collection of individual files and/or packages. This allowed us to determine the benefits of building models and making predictions across different products in a SPL, including newly introduced products before they have been released.* Related work that used regression to make numerical predictions of post-release faults across software applications (Nagappan et al, 2006) was based on unrelated software applications (i.e., five Microsoft software systems: Internet Explorer 6, IIS W3 Server Core, Process Messaging Component, DirectX, and NetMeeting), rather than products that are members of a SPL. These cross-application predictions in a non-SPL context (often referred to as cross-project predictions) did not show promising results.

This paper extends our previous work (Devine et al, 2012) which analyzed pre-release faults in a much smaller industrial product line and predicted the number of *pre-release faults* in newly developed components using a model built on the existing components. In this paper our focus is on *post-release faults* observed over multiple releases in four much larger products from an open-source, evolving product line. Furthermore, in this paper we apply a specific generalized linear model to the problem of software fault prediction for the first time. While generalized linear models were used by others in (D'Ambros et al, 2009, 2010), they did not specify the distribution and linking function. In this paper, we model post-release faults as an ordered multinomial distribution and use the cumulative negative log-log linking function. This combination is specifically

appropriate for skewed distributions characterized by higher probabilities of lower or zero values (Norušis, 2012), as is typical for the distributions of post-release faults across software units (i.e., files, components, or packages).

The assessment part of this paper clearly distinguishes between the two dimensions of reuse (i.e., reuse across releases and reuse across products) for the first time and explores how they affect the SPL change proneness and fault proneness, including the statistical significance of the results. The effect of reuse across releases on software quality has been studied both in non-SPL context (Fenton and Ohlsson, 2000; Ostrand and Weyuker, 2002; Ostrand et al, 2004; Khoshgoftaar and Seliya, 2004) and in SPL context (Mohagheghi and Conradi, 2008) and, therefore, this part of our study is a conceptual replication. On the other side, the reuse across products is relevant only in a SPL context. In this paper, we distinguish among different levels of reuse across products (i.e., commonalities, high-reuse variation, low-reuse variation and single-use variation packages) and explore two aspects: (1) each of the specific levels of reuse across products over multiple releases and (2) different levels of reuse across products within individual releases. While the former aspect furthers our initial work (Krishnan et al, 2011a), the latter is explored for the first time in this paper.

Compared to our previous prediction studies based on Eclipse (Krishnan et al, 2011b, 2012), this study incorporates both change and static code metrics, and performs numerical prediction rather than binary classification. We also present a new machine learning approach, which has not been used previously either for numerical prediction or for classification of fault-prone units. Specifically, we construct a model from the data of one member of the product line family and use it to rank the packages of other members in the subsequent release by the amount of post-release faults they are likely to exhibit. This approach allowed us to explore for the first time whether predictions benefit from data available for other products in the SPL, that is, to explore the usefulness and accuracy of cross-product predictions, as well as to make predictions for new products before they were released.

## 3 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. The Eclipse wiki refers to the emergence of Eclipse from a long product line of development environments (Laffra and Veys, 2013). Originally created by IBM in November 2001, it is currently maintained by the Eclipse Foundation, a not-for-profit member supported corporation that hosts 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 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.

Eclipse has been studied as a product line (Chastek et al, 2007; van der Linden, 2009) and in the context of open source product family engineering (van Gurp et al, 2010). Pohl et al (2005) identify two key differences between SPL engineering and single-system development: the need for two distinct development processes, namely, the domain engineering process and the application engineering process; and the need to explicitly define and manage variability. In both these regards, the Eclipse project demonstrates SPL engineering practices. With regard to the first criterion, the Eclipse project employs both domain engineering and application engineering, as described below, to plan for and implement reuse. With regard to the second criterion, Eclipse explicitly defines and manages variability across products through its customization to specific user-community requirements.

Eclipse has also been referred to as an ecosystem, both in its own wiki (Laffra and Veys, 2013), and in a recent keynote by Taylor (2013). Taylor draws a distinction between a product line, which he considers to have a single agency developing it, and an ecosystem, in which there may be multiple development agencies. He considers the open-source nature of Eclipse's development to move it beyond a product line. Our view is closer to that of van der Linden (2013), who considers the benefits of the heterogeneous, distributed development of product lines. Similarly, Pohl et al (2005) describe a wide variety of organizational structures that are used to develop product lines. They indicate that, for an organization with strong project groups and with a need for a strong customer focus, the distributed domain engineering organization (as with Eclipse) is best suited. In such an organization, domain engineering is distributed among business units, with each producing products for a specific customer group or market segment. The development of open-source product lines such as Eclipse is a natural outgrowth of industry's investments and interest in distributed development and open source systems. The Eclipse project development practices and suite of products are well within the scope of current product line engineering.

Our study encompasses a total of seven Eclipse releases, whose names and dates are given in Table 1. As mentioned in the Introduction, starting with the Europa release, Eclipse evolved from a single, all-encompassing product into several specialized products (see Figure 1). In the early releases 2.0, 2.1, and 3.0, only one product existed, namely Classic. These early releases have been analyzed before based on the data set provided by Zimmermann et al (2007), and are included in this study for several reasons. First, it allows us to compare our results with related work. Second, it allows us to explore the evolution of Eclipse from a single product (in releases 2.0, 2.1, and 3.0) to a SPL with multiple products (starting from the release 3.3 codenamed Europa). Finally, it allows us to explore whether accurate predictions can be made for newly emerging products. We examined four products from the Eclipse project: Classic, C/C++, Java, and JavaEE. These products were chosen due to their persistence throughout the examined releases. The Europa release contained only one additional product, RCP, which is not considered in this study because it differs from the product Java by only one package. The sizes and the number of faulty packages[3] in each product, for each release are given in Table 1.

Table 1: A timeline of the products examined in this study with the sizes (in thousands of lines of code) and number of packages. and number of faulty packages

| *Release (codename)* | *Date* | *Classic* | | *C/C++* | | *Java* | | *JavaEE* | | ***Total*** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *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 2 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. We first introduced the Venn diagram representation[4] in our earlier work (Devine et al, 2012) to illustrate the amount of code shared between the products. In each Venn diagram, the 61 to 62 packages in the central region, which are shared by all four products, are *common packages*. For example, the package

[3] For this study, a package is considered faulty if any file contained in that package exhibited one or more post-release faults.

[4] Thompson and Heimdahl (2003) proposed a set-theoretic approach to represent requirements reuse in product line engineering, which described the boundaries of sets as commonalities and the members within the sets as products. The approach taken in our previous work (Devine et al, 2012) and used here is complementary to (Thompson and Heimdahl, 2003). Specifically, it is used to illustrate the amount of shared code at different levels of cross-product reuse; the elements within the sets are packages of the SPL, and the boundaries of sets define the products.

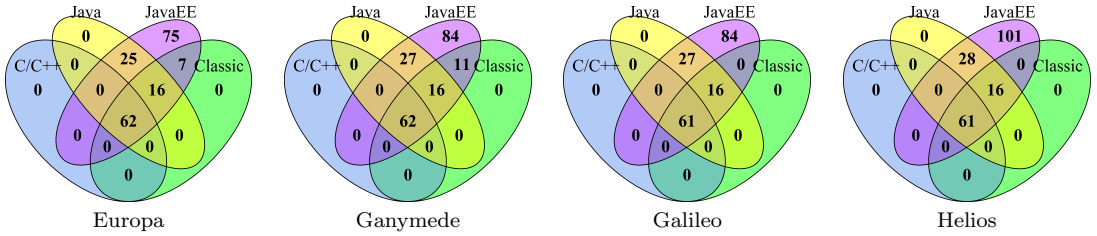



Fig. 2: Venn diagrams showing the distribution of packages among the four products for the four multiple-product releases studied

*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.

As noticed in our previous work based on an industrial product line (Devine et al, 2012) and confirmed on Eclipse, there are different levels of reuse across the products of the SPL. The four regions directly adjacent to the center in Figure 2 contain packages that are reused in three out of the four products. We label these packages used in all but one product *high-reuse variation packages*. In this 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++.

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

The remaining 75 to 101 packages are used in only one product, and can be found in the perimeter regions of Figure 2. We call these packages *single-use packages*. For example, the Web Standard Tools package *org.eclipse.wst.wsi*, which is used only in JavaEE, belongs to this group.

Figure 2 shows that C/C++ is made entirely of common packages, 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 the single-use packages. It is important to note that, while several other products exist in addition to the four products examined in this study for the Galileo and Helios releases of Eclipse, the packages marked as single-use and low-reuse in this study are not used in any additional products. In other words, throughout the entire Eclipse product line family the single-use and low-reuse packages analyzed in this study are used in only one and two products, respectively.

Ensuring data quality is a very important aspect of empirical research. Therefore, in the process of collecting metrics, we made great efforts to assure that the data we collected was as complete and accurate as possible. We also explicitly documented the inclusion / exclusion criteria. Data quality and the potential threats to validity are discussed in detail in Section 9. Here we briefly describe several points.

As expected, because 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, despite our best efforts. Not all source code files for which CVS logs were recorded were available to anonymous developers at the Eclipse CVS repository server[5]. 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 package for the C/C++ product. C/C++ thus became a product composed of only common packages in this study. We retained it as relevant to the study because it provides important insights into how the most reused packages, those common to all products, behave in predictive models. Additionally, the source code for the Galileo and Helios releases of PDE (Plug-

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

in Development Environment) was unavailable, as shown in Figure 2 by the eleven packages shared between Classic and JavaEE which are present in the Ganymede release, but do not appear in the Galileo or Helios releases. 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 from the Ganymede release.

Since the analysis and predictions in this paper were carried on at the package level, whenever a part of the data was not available for a package, that package and the associated number of post-release faults were not included in the dataset. The total number of packages for which data were complete and accurate, and the number of those complete packages that exhibited post-release faults for each release, are given in the two rightmost columns in Table 1 and shown in the bar graph in Figure 3.

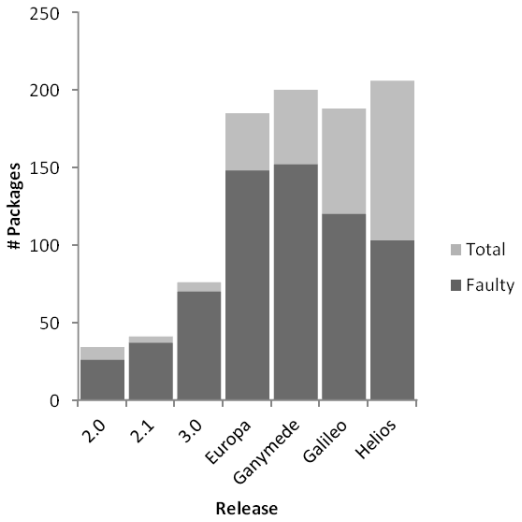



Fig. 3: A histogram showing the total number of packages and the number of packages which contain at least one post-release fault, for each of the releases

## 4 Description of metrics and their extraction

We use the post-release faults as a measure of software quality. Specifically, we considered the top five severity categories of Eclipse faults: blocker, critical, major, normal and minor. This study uses two types of software metrics — change metrics and static code metrics. These metrics were collected at the 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 was then treated as a feature, in the machine learning sense, when building and evaluating the predictive models.

### 4.1 Change metrics

The alterations made to a source code file over the course of its existence are captured by the change metrics. Change metrics used in this study were collected previously and used for classification of fault-proneness at a file level in our previous work (Krishnan et al, 2012). In particular,

Table 2: 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* | $\frac{\sum_{i=1}^{n} Age(i) \times LOC\ Added(i)}{\sum_{i=1}^{n} LOC\ Added(i)}$ |

for each file we extracted the same set of seventeen change metrics as in the work by Moser et al (2008). These metrics are defined in Table 2, while detailed descriptions can be found in (Moser et al, 2008).

Next, we provide a brief description of the change metrics extraction process. Eclipse uses CVS as a version control system, which maintains timestamped log files detailing the history of changes made to any given source code file. The first step in the process of extracting the change metrics was to map the CVS log entries to the bug database at a file level. For this purpose, we followed the approach of Zimmermann et al (2007). Specifically, for each entry in the bug database with one of the top five severity categories (i.e., blocker, critical, major, normal and minor), we matched the bug ID with the commit messages from the CVS logs. (The trivial and enhancement categories were not considered.) We used strings such as 'bug' and 'fix' followed by numbers that match the IDs from the bug repository. If a match was found, we mapped the bug entry to the corresponding source code file. In case of releases 2.0, 2.1, and 3.0, the bug IDs were either four or five digit strings. while for Europa and later releases the bug IDs were six digits strings. A manual review showed no entries containing the word 'bug' that were not caught by this pattern matching.

For each file the *Revisions* metric denotes the number of times that file was involved in changes, including bug fixes, improvements to existing features, and addition of new features. The *Refactorings* metric (i.e., the number of times a file was refactored) was extracted following the approach of Moser et al (2008), which consists of tagging all log entries with the word 'refactor' in their commit comments. Change metrics also include *Bugfixes* metric, which represents the number of times a file was involved in pre-release bug fixes. The pre-release bugs were distinguished from post-release bugs using the bug creation date from the bug repository. If the bug was created before the release date, we denoted it as pre-release bug. In this work we accounted for five categories of bugs (i.e., faults): blocker, critical, major, normal and minor (that is, only the trivial faults were not considered). The *Authors* metric is the total number of developers who contributed to a given file. *LOC Added* and *LOC Deleted* denote the number of lines added and deleted in all the revisions. *Codechurn* denotes the effective lines added and was calculated by subtracting the *LOC Added* from the *LOC Deleted*. As in (Moser et al, 2008) we also calculated the maximum and average values for the lines added, lines deleted and codechurn. To determine the *Changeset*

size (i.e., the files that were committed together with the file of interest) we used the CVSPS tool (Mansfield, 2012) after making slight modifications to it. For extracting the *Age* metric we reviewed all CVS log data from 2001 onward to identify the timestamp of the first occurrence of each file name. The *Weighted Age* metric, in addition to when the change was made, takes into account the size of change and is computed using the formula given in Table 2.

To ensure the data quality we made a few modifications to the log script so that the data collected from various input sources were compatible and mapped accurately. For example, files marked as 'dead' in the Eclipse project are often moved to the Attic in CVS, which results in different file path. To handle this, we excluded from our study all instances that had the pattern '/Attic/' in their file paths. Another modification was needed to handle the fact that when using the CVS rlog tool with date, files that were not changed during the specific filter period were listed as having no revision-specific information such as date, author, etc. This is true even if the file was previously marked 'dead' on a branch. In order to determine which files were 'alive' and which revisions applied to each release, rather than examining only the date of each specific release we examined the rlog for the entire file history.

### 4.2 Static code metrics

Static code metrics capture information pertaining to the source code. 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 easily be gathered via a variety of software tools available online. In part for this reason, static code metrics have been frequently used in fault prediction studies. For this study, we used the freeware code analysis tool SourceMonitor[6] (SourceMonitor, 2011) to extract the static code metrics.

A list of the gathered static code metrics and their brief descriptions are given in Table 3. 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.

Table 3: A list of static code metrics and their descriptions

| **Static Code Metrics** | **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* $x$ where $x = 0, 1, 2, \ldots, 9$ | Total number of statements in all methods contained at block level $x$. |

[6] The complexity measure used by SourceMonitor approximately follows the definition by McConnell (2004).

### 4.3 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 contained no post-release faults, making the file level data more suited to binary classification than ordered multinomial fault prediction. On the other hand, when viewed from the package level, the skewness of the distribution of post-release faults towards zero is much less pronounced. Such aggregations have been performed and supported in the literature, such as (Zimmermann et al, 2007; D'Ambros et al, 2012).

Before aggregation could occur, the change metrics and source code metrics had to be combined for each file. Code was written to accomplish this utilizing the detailed file names of the Eclipse project as combination indices. Selected matchings were manually validated to ensure the accuracy of the resulting data sets.

Change and static code metrics were then aggregated using the naming conventions of the Eclipse project, as in (Zimmermann et al, 2007). For example, the metrics for the file named *org.eclipse.gef.Command.java* were combined with those for all the class files named *org.eclipse.gef.[Class name].java*. Faults, initially mapped at file level, were attributed to the package to which these files belong. The total number of faults in a product includes the total number of faults for each package in that product. The aggregations were performed using the *Aggregate* function in IBM SPSS (v. 20.0), and the mean, median, maximum, and total (i.e., sum) 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. (For the specific aggregations for each static code metric and change metric the reader is referred to the Appendix.) The aggregation process resulted in a total of 112 metrics (i.e., features) for each package. (The total number of packages for each product, by release are listed in Table 1.) Thus, each package was characterized by a vector $\mathbf{m}$ of 112 metrics (i.e., features), where $\mathbf{m}[i], i = 1, \ldots, 73$ were static code metrics, while $\mathbf{m}[i], i = 74, \ldots, 112$ were change metrics.

## 5 Machine learning approach

This section describes the data preprocessing steps, some 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.

### 5.1 Data preprocessing

Before creating regression models, we normalized the aggregated data. Normalization is a common practice in machine learning, see for example (Bell et al, 2006; Kitchenham and Mendes, 2009; Ostrand et al, 2004, 2005). For this study, we performed a logarithmic transformation of the following metrics which had very skewed distributions: *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

$$x' = \frac{max_{user} - min_{user}}{max_i - min_i}(x - min_i) + min_{user}, \tag{1}$$

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 $max_{user}$.

### 5.2 Background on Generalized Linear Regression models

To determine the packages likely to exhibit post-release faults we used the generalized linear models (GLMs) introduced by Nelder and Wedderburn (1972) and later extended by McCullagh and Nelder (1983), which are implemented in IBM SPSS (v. 20.0). These models are of the general form given by

$$link(\gamma_{ij}) = \theta_j - [\beta_1 x_{i_1} + \beta_2 x_{i_2} + \ldots + \beta_p x_{i_p}], \tag{2}$$

where $\gamma_{ij}$ is the cumulative probability of the $j$th category for the $i$th case, $\theta_j$ is the threshold for the $j$th 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 $i$th case.

The generality of GLMs comes from their flexibility. Model builders are able to specify the type of distribution to model as well as the transformative linking function to use. The linking function is a transformation of the cumulative probabilities that allows estimation of the model. To determine which regression model to apply to our data, we considered both theory and implementation. The best ranking results were achieved when treating post-release faults as an ordered multinomial distribution (treating each recorded value as an ordinal category) and using the cumulative negative log log linking function,

$$link(x) = -\log(-\log(x)). \tag{3}$$

This is due to the fact that this combination is theoretically well-suited for data with many lower or zero values and fewer higher values (Norušis, 2012). Since post-release fault distributions are typically skewed (i.e., have many packages with a few or no faults and a few packages with many faults), GLMs are more suitable to the task of software fault prediction than standard linear regression models. It should be noted that we also tried negative binomial model and logistic regression model. Both models performed comparably, but the results tended to be more extreme than the cumulative negative log log GLM model, i.e., they performed much better in some cases and much worse in others. Due to its consistently good predictions, we chose to present the prediction results of the cumulative negative log log GLM model.

### 5.3 Cross-product prediction of fault proneness

Software development and testing teams want to use models constructed from data for the previous release of their product to predict which packages are likely to exhibit the most post-release faults in the next release. Our approach seeks to mimic this actionable procedure for the development community in order to investigate the merit of such prediction.

Specifically, we built regression models for each product in each release of Eclipse from the aggregated and normalized data, consisting of 112 features (i.e., metrics) as described in Section 4. 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 product's release, referred to as post-release faults throughout the paper.

We built a model for each of the products in release $n$ and used it to predict the degree of post-release faults for *each* product in the following release $n+1$. Thus, we built a total of fifteen predictive models – one model for each of the first three releases (which contained only one product each), and four for each of the next three releases (which contained four products each). Models built for each product in each release $n$ were then used to make predictions for all products in the subsequent release $n+1$, resulting in a total of fifty-four trials. We tested the models on all products in the subsequent release, rather than on only the same product, because one of our main research questions was to explore whether reuse across products in a product line and additional data from other product line members can provide benefits for post-release fault prediction. This

research question has not been explored previously in either the related work, or in our previous work.

It should be emphasized that building a model on the previous release (i.e., $n$-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, which mimics the software development process across releases and is motivated by practical needs. In addition, this prediction approach preserves the temporal order of the fault detection process, thus contributing to more realistic predictions[7].

### 5.4 Feature selection

In order to select the features that have the best predictive capability out of the (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 usually speeds up machine learning algorithms and improves their performance.

In particular, we performed feature selection via stepwise regression (Kleinbaum et al, 1988) to determine the best features to use in the generalized linear models used for prediction. In stepwise regression, the features in the final model are selected in one of three ways: forward selection, backward elimination, or bidirectional elimination. The forward selection process starts with the single feature model with the best fit and adds features that most improve the model, one at a time, until no significant improvement can be achieved. Backward elimination begins with all features in the model and eliminates as many as possible while the model improves. Bidirectional elimination is a combination of the other two techniques, that adds and deletes features from the model until the best fit is achieved. For feature selection we used bidirectional elimination, which only includes features that improve the fit of the model as a whole and gives each feature an equal chance to gain inclusion in the model. This method is also suggested as a way to handle the problem of multicollinearity[8] in linear regression models (Bingham and Fry, 2010). Our stepwise regression models were implemented in IBM SPSS (v. 20.0), using the default maximum number of steps (twice the number of features).

### 5.5 Performance metrics

We report three performance metrics, the *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 $\rho$ and Kendall's $\tau_b$. 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.

#### *5.5.1 nTop20%*

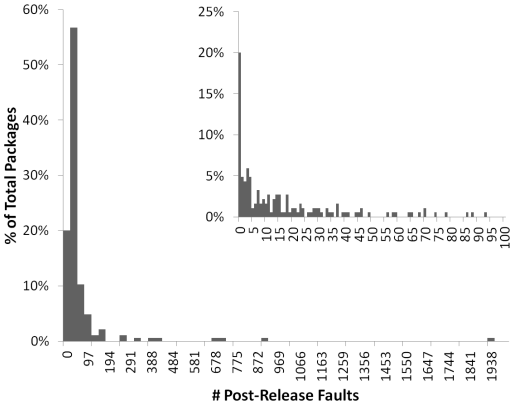
The percentage of actual faults found in the 20% most faulty packages calculated by the predictive model is a common performance metric reported in several other works (Ohlsson and Alberg, 1996;

[7] Some form of $k$-fold cross validation is commonly employed in machine learning in general and software engineering in particular. Cross validation is the process of splitting the data randomly into $k$ groups, and then predicting values for the $k$-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 (as was done in this paper) because averaging the results over $k$ repeated trials offers more consistent, flattened end results than one achieved via building models and predicting on disjoint sets.

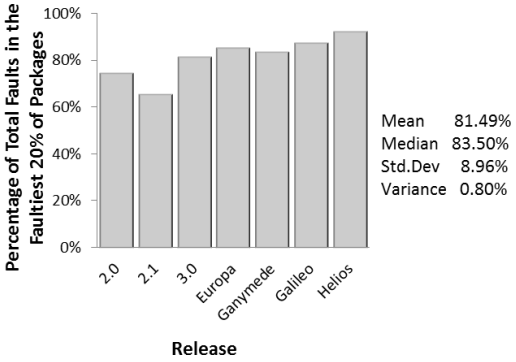
[8] Many software metrics are highly correlated to each other, which engenders a problem that is commonly referred to as multicollinearity. To quote Kutner et al (2004) "The fact that some or all predictor variables are correlated among themselves does not, in general, inhibit our ability to obtain a good fit nor does it tend to affect inferences about mean responses or predictions of new observations ..." However, multicollinearity may cause the estimated regression coefficients to have a large sampling variability and thus affect explanatory studies.

Fenton and Ohlsson, 2000; Ostrand et al, 2004, 2005; Bell et al, 2006; Shin et al, 2009). To justify the use of this metric for our data we explored how the post-release faults are distributed among Eclipse packages. Here, we only present the histogram and detailed analysis for the Europa release because the results for the other releases were very similar. As can be seen from the histogram in Figure 4a, the distribution of post-release faults across packages is skewed – many packages had no faults or a small number of faults, but some packages had a very large number of faults. Specifically, 20% of the packages did not have any post-release faults (i.e., were fault free) and 51% of the packages contained ten or fewer post-release faults. These packages together contained less than 3% of the total faults. On the other side, the faultiest packages (shown at the right tail of the histogram in Figure 4a) had as many as 1,920 faults each. Overall, the 20% most faulty packages contained over 85% of the total faults. (The cumulative negative log log linking function, given by equation (3), was also chosen based on these observations.)

The results were consistent for all releases of Eclipse considered in this paper, as it can be seen in Figure 4b, which shows the percentages of the total number of post-release faults located in the 20% most faulty packages for each of the seven releases, together with the descriptive statistics of the results. It follows that **from 66% to 93% of the post-release faults detected across all products in each release were located in approximately 20% of the packages, with the average and median around 81% and 84%, respectively**.




(a) A histogram showing the percentage of total packages for the Europa release which exhibit the number of post-release faults listed on the x-axis. Bin widths were assigned according to the Freedman-Diaconis rule. The insert shows an unbinned view of packages with 0 to 100 faults.

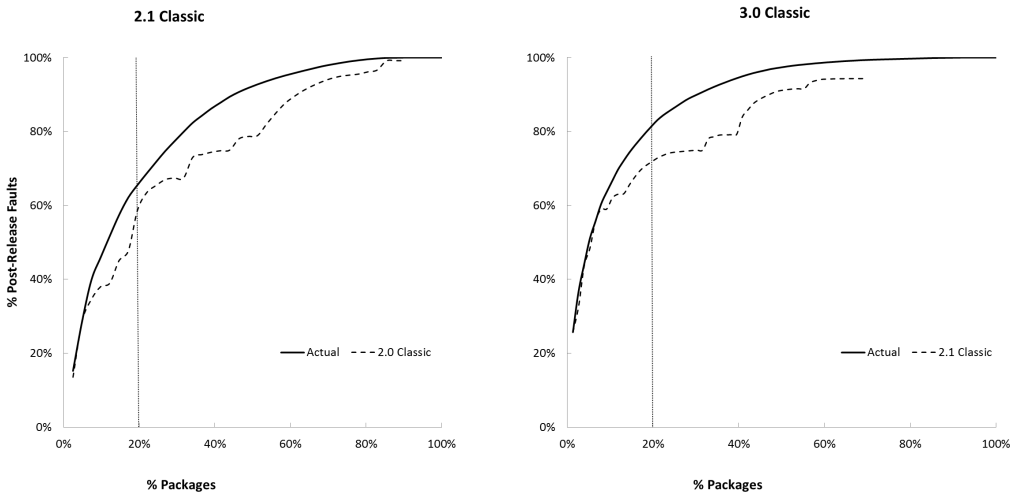



(b) A bar graph showing the percentage of total post-release faults detected in the top twenty percent of the total packages for each release, with accompanying descriptive statistics

Fig. 4: A histogram and bar graph addressing the location of the majority of post-release faults

These results generally agree with other works (Andersson and Runeson, 2007; Boehm and Basili, 2001; Fenton and Ohlsson, 2000; Hamill and Goseva-Popstojanova, 2009; Ostrand and Weyuker, 2002), which have consistently found that between 60 and 90% of faults normally reside in around 20% of the lines of code, files, or packages, depending on the unit. Furthermore, these results confirm that 20% is a good cut-off point for the most faulty packages used in the nTop20% performance metric.

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 (Ohlsson and Alberg, 1996), which provides a succinct manner of showing the ability of independent variables to rank a dependent variable (Fenton and Ohlsson, 2000).




(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

Fig. 5: 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 faultiest packages.

As an illustration we show two Alberg diagrams – in Figure 5(a) for release 2.1 and in Figure 5(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 faultiest 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(a) (Figure 5(b)) the dashed line shows the total number of actual faults located in the packages identified as the faultiest 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 faultiest by the model. The area between the dashed and solid lines shows how close the model's ranking comes to identifying the actual faultiest packages. It is important to note that we were conservative when plotting the predictive models' performances. That is, we only ranked and plotted in dashed lines in our Alberg diagrams the packages with nonzero predicted fault values. This is the reason why the dashed lines in Figures 5(a) and (b) terminate before convergence with the solid line. We chose this approach because packages with zero number of predicted faults cannot be ordered in a meaningful way and should not be contributing to the accuracy of the predicted ranked lists.

The vertical dotted line at 20% is a reference line used for measuring the effectiveness of a predictive model. For example, 65.5% of the total post-release faults (solid line) were located in the top 20% of actual faulty packages in the 2.1 release of Classic. The top 20% of faulty packages identified by our predictive regression model contained 58.7% 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 faultiest 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 20%*

*most faulty packages for that product.* For brevity, we refer to this normalized performance metric taken from Alberg diagrams as the *nTop20%* score for a model. For 2.1 Classic (see Figure 5) the metric normalized in this way is nTop20% = 58.7/65.5 and it shows that *the predicted faultiest 20% of packages account for approximately 90% of the faults occurring in the actual top 20% of the faultiest packages.* It should be noted that adding a new product would not require rescaling of the nTop20% scores for the existing products because the normalization is done per product, that is, with the actual number of faults in the faultiest 20% of packages in that specific product.

We prefer *nTop20%* over other performance metrics, such as the coefficient of determination $R^2$, due to its ease of comparison across software products combined with its robustness to outliers, as detailed by Ohlsson and Alberg (1996).

### *5.5.2 Rank correlation measured by the Spearman's $\rho$ and Kendall's $\tau_b$*

The Alberg diagram basically shows two ranked lists of packages – one based on the actual number of post-release faults in each package (shown as a solid line) and another based on the predicted number of post-release faults in 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 $\rho$ and Kendall's $\tau_b$ 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 $\rho$ is the nonparametric version of Pearson's $r$ correlation coefficient, and is used when the assumptions of normality and equal variance are not fulfilled or when the data are given in an 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. Spearman's $\rho$ is computed by:

$$\rho = 1 - \frac{6\sum_{i=1}^{n} d_i^2}{n(n^2-1)}. \tag{4}$$

Tied values among the ranks are handled by assigning them the average of their positions in the ascending order of the values. For a full description of Spearman's $\rho$ correlation coefficient see (Agresti, 2010).

Kendall's $\tau_b$, also described by Agresti (2010), is a variant of the Kendall's $\tau$ coefficient specifically designed to handle ties within the ranked lists. The Kendall $\tau_b$ coefficient is defined as:

$$\tau_b = \frac{n_c - n_d}{\sqrt{(n_0 - n_1)(n_0 - n_2)}} \tag{5}$$

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 $\tau_b$ has several advantages[9] over Spearman's $\rho$. Nevertheless, in order to be able to compare our results with the previous works which used the Spearman's $\rho$ metric we used both metrics to measure the association of the ranked lists based on the actual and predicted number of post-release faults. It should be noted that when Spearman's $\rho$ and Kendall's $\tau_b$ are both used on the same data, typically Spearman's $\rho$ tends to be larger than Kendall's $\tau_b$, in absolute value. However, as a test of significance both produce nearly identical results in most cases.

[9] Kendall's $\tau_b$ approaches the normal distribution quite rapidly so that the normal approximation is better for Kendall's $\tau_b$ than it is for Spearman's $\rho$. Another advantage of Kendall's $\tau_b$ 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).

## 6 Assessment of the SPL quality

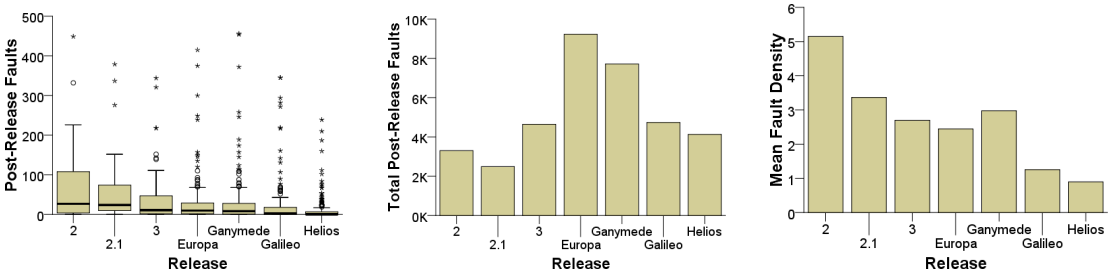
The research questions in this section assess the quality of Eclipse as a product line. First, we study the cross-release reuse by considering the distribution of post-release faults occurring in all products, by release. Then, we examine the change and fault proneness for packages at different levels of cross-product reuse as they evolve throughout multiple release and within each release.

### 6.1 RQ1: Does quality, measured by the number of post-release faults for the packages in each release, consistently improve as the SPL evolves across releases?

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 across releases we offer the three plots in Figure 6. The box plot in Figure 6(a) shows the median value and variance of post-release faults for the packages in each release. As the 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 6(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. Namely, Eclipse, which initially consisted of only one product Classic, starting from the Europa release began to resemble a true product line with multiple products. To accommodate this, many changes were made to the structure of the packages and to the source code itself. Compared to release 3.0 the Europa release had 128 new packages and the source code more than doubled in size.

To account for this size increase, Figure 6(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 the lowest 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.




(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 in thousands of faults

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

Fig. 6: Plots showing post-release fault data over all packages for each release

To statistically confirm that the SPL quality improves as it evolves across releases we performed a Kruskal-Wallis test on the distribution of the number of post-release faults shown in Figure 6(a), as well as on the post-release fault densities shown in Figure 6(c). In both cases, the Kruskal-Wallis test confirmed that the distributions were not equal for Eclipse releases 2.0, 2.1, 3.0, Europa,

Ganymede, Galileo, and Helios. Then we ran the post-hoc Jonckheere-Terpstra test, which rejected the null hypotheses in favor of the ordered alternative hypotheses that the number of post-release faults, as well as the post-release fault density for each release was less than or equal to those of the previous release ($p = 6.2 \times 10^{-22}$). (Note that even though there is a small visible spike of the average post-release density in the Ganymede release shown in Figure 6(c), the variation of the fault density was not statistically significant and did not lead to rejection of the null hypothesis.) **This strong evidence of overall decreasing trends in both the number of post-release faults and post-release fault densities show that quality did improve as the SPL evolved through releases.**

Quality improvement is intuitively expected in release-based software as it evolves over time, and several case studies have corroborated this expectation. In a non-SPL context, consistent with our results, faults have been shown to decrease over multiple releases in empirical software studies conducted by Ostrand and Weyuker (2002), Ostrand et al (2004), and Khoshgoftaar and Seliya (2004). However, another non-SPL study conducted by Fenton and Ohlsson (2000) produced conflicting results, that is, reported that fault densities remained roughly the same across two releases of telecommunications software. In a SPL context Mohagheghi and Conradi (2008) reported an increase in faults across three releases of a telecommunications product, which was a part of a SPL. The authors claimed that this result was due to improved testing and larger user base. Our finding that the SPL's quality improves as it evolves through releases is significant for the following two reasons. First, the improved quality was shown to be statistically significant, which was not the case with related works in non-SPL context (Ostrand and Weyuker, 2002; Ostrand et al, 2004; Khoshgoftaar and Seliya, 2004). More importantly, we showed that the improved quality over releases exists in a SPL context, which was not the case with the much smaller SPL study reported by Mohagheghi and Conradi (2008). It appears that, as pointed out by Lim (1994), "Because work products are used multiple times, the defect fixes from each reuse accumulate, resulting in higher quality."

### 6.2 RQ2: Do packages at different levels of reuse across products mature differently across releases? Does the quality of products benefit from reusing packages in multiple members of the SPL?

A key goal of product line engineering is the application of reuse across products to achieve higher quality, in a cost efficient way (Gomaa, 2004; Pohl et al, 2005). In addition, reuse is also expected to reduce the changes to software (Weiss and Lai, 1999), that is, reused components are expected to remain more stable. Therefore, metrics of software change were used as quality indicators of reused programs, both in a non-SPL context (Thomas et al, 1997; Selby, 2005) and in a SPL context (Mohagheghi and Conradi, 2008). RQ2 evaluates the change and fault proneness behaviors of different levels of reuse across products, over multiple releases of the Eclipse product line and within each release individually. The first part of this research question furthers the very preliminary observations made in our previous work (Krishnan et al, 2011a), while the second part is explored for the first time in this paper. It should be noted that RQ2 is relevant only in the SPL context.

To evaluate the benefits of reuse across products (i.e., from code shared among multiple products,) we grouped the packages in each release into four categories according to their level of cross-product reuse: single-use packages (used in only one product in this release), low-reuse variation packages (used in two products in this release), high-reuse variation packages (used in three products in this release), and common packages (used in all four products in this release). These categories are visualized by the Venn diagrams in Figure 2. In addition, for each release we explicitly distinguished between previously existing ('old') packages and newly developed ('new') packages. Note that the use of the term 'old' refers to packages which were reused across releases. Packages in any level of cross-product reuse can be 'old' (i.e., reused across releases) or newly

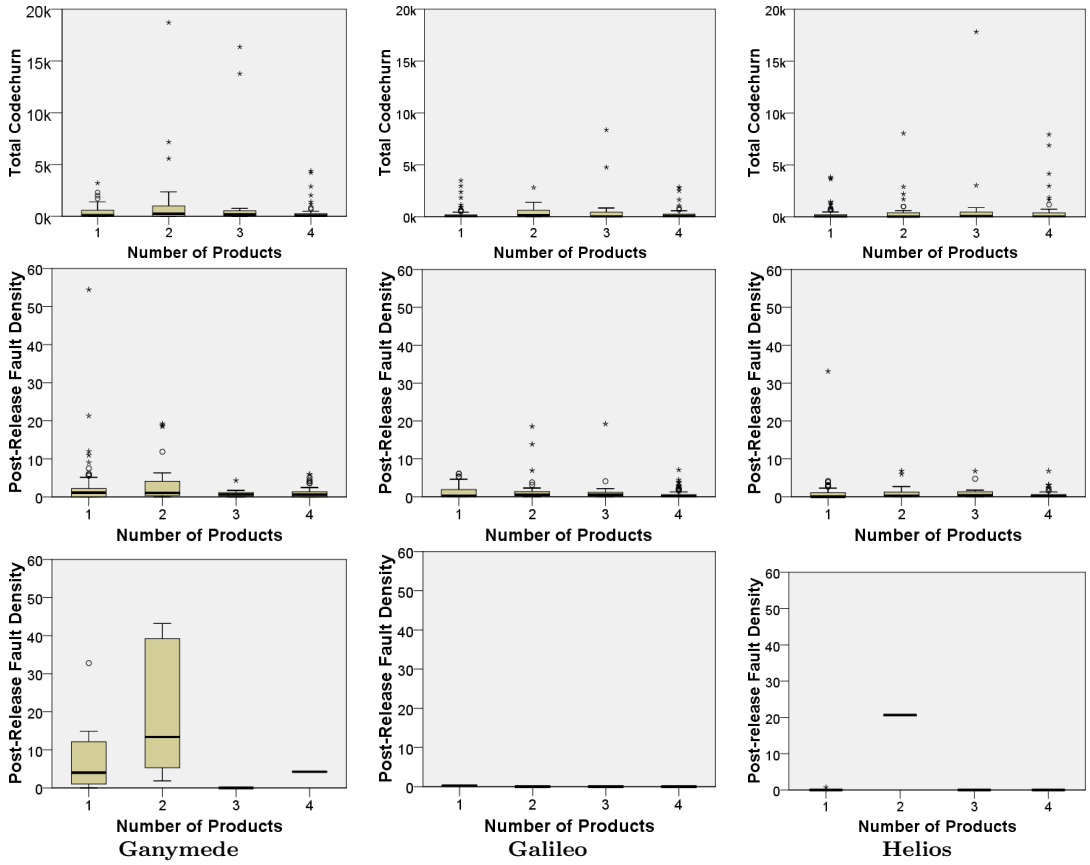



Fig. 7: Box plots showing the total Codechurn of previously existing ('old') packages (first row), the post-release fault densities of previously existing ('old') packages (second row) and post-release fault densities of newly developed ('new') packages (third row) grouped by their level of reuse across products, for releases Ganymede, Galileo, and Helios. The values 1, 2, 3, and 4 on x-axes note, for each release, the single-use, low-reuse variation, high-reuse variation, and common packages, respectively. In Ganymede, Galileo, and Helios releases the number of previously existing packages per level of reuse (single-use, low-reuse, high-reuse, and common) were (74, 32, 16, 61), (83, 27, 16, 61), and (83, 27, 16, 61), respectively. The sample sizes of newly developed packages were (10, 6, 0, 1), (1, 0, 0, 0), (18, 1, 0, 0)for Ganymede, Galileo, and Helios releases, respectively.

developed. For example, one newly developed common package (reused across all four products considered in this paper) was introduced in the Ganymede release.

Figure 7 depicts the total Codechurn of previously existing packages (the first row), the post-release fault densities of previously existing packages (the second row), and the post-release fault densities of newly developed packages (the third row) for the Ganymede, Galileo, and Helios releases. The Europa release is not considered in this section due to the three years time difference from the previously considered release, 3.0, which made the identification of newly developed packages impossible.

Exploring the total Codechurn, shown in the top row in Figure 7, 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 may acquire new functionality and must also adapt to coexist with newly introduced variation packages.** This evidence-based observation does not support the

traditional expectation that commonalities would remain stable (Weiss and Lai, 1999), that is, would not be change prone. Interestingly, even though the introduction of new functionality and adaptation also introduced new post-release faults, the fault density of the common packages remained fairly low, that is, the common packages incurred a low number of faults for their size. In general, the second row of Figure 7 shows that **even though pre-existing ('old') packages had a relatively high total Codechurn, they showed low post-release fault densities, clearly illustrating a benefit of reuse across releases.**

To examine how 'old' packages at different level of cross-product reuse (i.e., single-use, low-reuse, high-reuse, and common packages) evolve across releases, we ran two tests – the Friedman test and Page test for ordered alternatives – first for total Codechurn and then for post-release fault density. These tests are applicable here because for each level of reuse across products, we examined the values of the corresponding attribute (i.e., total Codechurn and post-release fault density) of the same packages through three releases (i.e., Ganymede, Galileo, and Helios), which resulted in a related, not independent, samples. The results of the tests are summarized in Table 4 and briefly described below.

For common packages, the Friedman test did not reject the null hypothesis that there is no difference between the total Codechurn across the three releases (i.e., Ganymede, Galileo, and Helios). The results were the same for high-reuse variation packages and low-reuse variation packages. In other words, at each of these three levels of cross-product reuse, packages continued to change, that is, the amount of changes did not decrease as they matured through the releases. On the other side, the Friedman test rejected the null hypothesis that there is no difference in total Codechurn for the single-use packages. The Page test for ordered alternatives[10] showed (with a significance of 0.05) that the total Codechurn of single-use packages decreased monotonically through releases, that is, single-use packages had less total Codechurn in the later releases. **The decreasing change proneness for single-use packages is likely due to the fact that they were not affected by changes made in other products.** (The results for the Codechurn density (i.e., total Codechurn divided by the size measured in thousand lines of code) were consistent with these results for the total Codechurn.)

The Friedman test rejected the null hypotheses that there is no difference in the quality (measured by post-release fault density) of the three subsequent releases for all levels of reuse across products, except for the high-reuse variation packages. Based on the Page test for ordered alternatives, the post-release fault densities showed a decreasing trend across the three releases under consideration for both common packages and single-use packages. The test for multiple comparisons between releases for low-reuse variation packages showed a statistically significant decrease in post-release fault densities between the Ganymede and Helios releases.

To summarize, for the old packages **only single-use packages showed a decreasing change proneness as they evolved across releases. Packages at other levels of cross-product reuse (i.e., low-reuse, high-reuse, and common packages) did not experience a statistically significant difference in change proneness.** When it comes to quality measured by the post-release fault density, **only high-reuse variation packages did not improve their quality as they evolved across releases. On the other hand, the common, low-reuse variation and single-use packages experienced statistically significant quality improvements**, which explains the improvement of SPL quality as it matured through releases shown to exist in RQ1.

Next, we focus on the 'old' packages within each of the three releases and test how change and fault proneness differ among different levels of cross-product reuse (i.e., singe-use, low-reuse, high-reuse, and common packages). This part of RQ2 was not explored in our previous work

[10] If the Friedman test results in rejection of the null hypothesis that there is no difference, a post hoc multiple comparison test is used to identify where the difference is. Alternatively, instead of the Friedman test, one can use the Page test which is used to test the null hypothesis that there is no statistically significant difference in several related samples (i.e., $H_0 : \mu_1 = \mu_2 = \mu_3$) against the ordered alternative that the samples differ in a specified direction, with at least one inequality (i.e., $H_1 : \mu_1 \geq \mu_2 \geq \mu_3$).

Table 4: Trends of change and fault proneness across three releases (Ganymede, Galileo, and Helios), for different levels of cross-product reuse (i.e., common packages, high-reuse packages, low-reuse packages, and single-use packages). The term 'No difference' is used for the cases in which the null hypothesis that there is no difference across the three releases cannot be rejected.

| Cross-product reuse level | Trend of Total Codechurn across releases | Trend of Post-release Fault Density across releases |
|---|---|---|
| Common packages | No difference | Decreasing trend |
| High-reuse variation packages | No difference | No difference |
| Low-reuse variation packages | No difference | Decreasing trend between Ganymede & Helios |
| single-use packages | Decreasing trend | Decreasing trend |

Table 5: Effects of cross-product reuse (i.e., common packages, high-reuse packages, low-reuse packages, and single-use packages) on change and fault proneness for each release (Ganymede, Galileo, and Helios). The term 'No difference' is used for the cases in which the null hypothesis that there is no difference across different levels of reuse cannot be rejected.

| Release | Trend of Total Codechurn within release | Trend of Post-release Fault Density within release |
|---|---|---|
| Ganymede | No difference | No difference |
| Galileo | No difference | No difference |
| Helios | No difference | No difference |

(Krishnan et al, 2011a). Interestingly, for each of the three releases, Kruskal Wallis tests retained the null hypothesis that there is no difference among total Codechurn in different levels of reuse across releases, as well as among post-release fault density (see Table 5). In other words, **change proneness (measured in total Codechurn or Codechurn density) and fault proneness (measured in post-release fault density) of 'old' packages, within each release did not experience a statistically significant difference among common, high-reuse, low-reuse and single-use packages.** This means that within one release, cross-product reuse did not affect the change and fault proneness of the 'old' packages (i.e., packages reused across releases). This unexpected result is not consistent with the results of related work based on a much smaller SPL (Mohagheghi et al, 2004; Mohagheghi and Conradi, 2008), which reported that reused components were less modified and had lower fault density than the non-reused components. Some reasons for these inconsistent results may be the different domain and development dynamics between Eclipse and the industrial product line from Ericsson, and the facts that the study by Mohagheghi and Conradi (2008) was based on only two products that shared 'as-is' 60% of the components and, furthermore, only included the variable components of one of these two products because the variable components of the second product were not available.

For the newly introduced ('new') packages, shown in the third row of Figure 7, no clear trend for post-release fault densities was observed. In the Ganymede release the ten newly developed single-use packages of JavaEE had noticeably lower post-release fault densities than the six newly developed low-reuse variation packages, which were shared between Classic and JavaEE. The one common package had very few post-release faults. In the Galileo release, only one newly developed single-use 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 packages and one was a low-reuse variation package. The eighteen new single-use packages, which belong to JavaEE, had nearly twice the total Codechurn as the one newly introduced low-reuse package, but together contained only two post-release faults. Basically, the new single-use packages had the highest total Codechurn and the lowest fault density. In contrast, the low-reuse package, shared by Java and JavaEE, had over a thousand post-release faults and the highest post-release fault density.

In summary, **newly developed low-reuse variation packages showed higher post-release fault densities than either single-use packages or the common package.** However,

we do not draw strong conclusions with respect to the post-release fault densities of the newly developed packages due to the small sample sizes, especially when compared to the sample sizes of the previously existing packages.

## 7 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. Basically, in each release we built a model from each product and then used it to make predictions for each product of the subsequent release from the corresponding pre-release data. This process resulted in building fifteen models which were then used for prediction in 54 trials.

For each of the fifteen models, we performed feature selection using stepwise regression with bidirectional elimination. Basically, the selection method incrementally determined a set of features which improved the fit of the model. The number of features selected for each of the fifteen models created in this study ranged from 4 to 16, with a mean value of 8.7. This indicates that a small number of features is sufficient to rank packages by the predicted number of post-release faults using the model built on the previous release and the pre-release data of the current release.

The Alberg diagrams for each of the 54 trials are shown in Figures 5 and 8. 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 previous 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 5.5.1.

The Alberg diagrams shown in Figures 5 and 8 were used to compute the nTop20% performance metric (i.e., the percentages of actual faults found in the top 20% of faulty packages identified by the predictive models), which are presented in a tabular form in Figure 9(a). Figures 9(b) and (c) show the values of the other two performance metrics – Spearman $\rho$ and Kendall's $\tau_b$ – each of which measures the association between two ranked lists of packages: the list ranked by the actual number of post-release faults and the list ranked by the predicted number of post-release faults. The tables in Figure 9 (a)-(c) are shown as heat maps where, for each performance metric, the better results are shaded darker.

Specifically, Figures 9 (a)-(c) show the results (i.e., the three performance metrics nTop20%, Spearman $\rho$ and Kendall's $\tau_b$) of the cross-product predictions (i.e., the 54 combinations of model-building and model-evaluation described in Section 5.3). In these figures, the releases 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. We use Figures 5, 8 and 9(a)-(c) to answer RQ3 and RQ4, which are focused on the cross-product predictions.

### 7.1 RQ3: Can we accurately predict which packages will contain a high percentage of the total post-release faults from pre-release data using the models built on the previous release?

To address this question, we first discuss the results for the nTop20% performance metric presented in Figure 9(a). The results show that the best nTop20% metric for each product across all releases is in the range of 76% to 97%, which indicates that **a high percentage of the faultiest packages post-release within a product can be consistently predicted from the packages' pre-release data**. As can be seen from the third column from the left in Figure 9(a), **the prediction model built on the Classic product in the release 3.0 was able to predict**

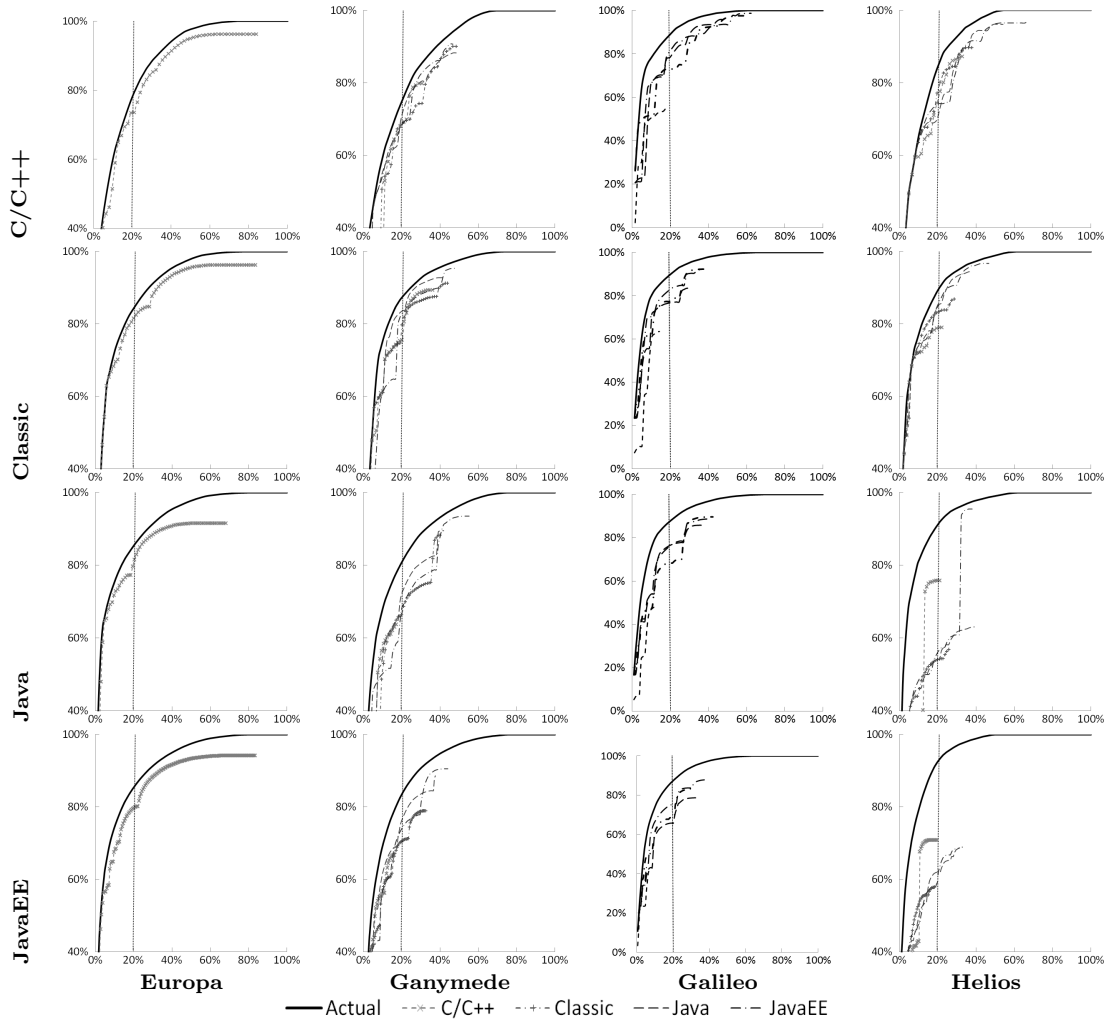
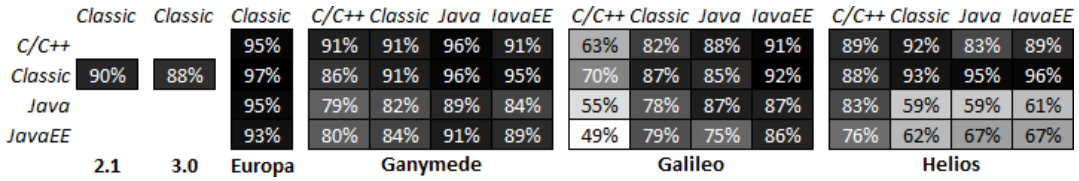



Fig. 8: Alberg diagrams for each product's predictive trials in the Europa, Ganymede, Galileo, and Helios releases. The releases on which the predictions were made are shown on the bottom. The predictive models were built on the previous release. The rows represent the products on which the predictive models were evaluated and are labeled on the left by the product name. In each Alberg diagram the full line represents the cumulative number of actually observed faults and each of the four different dashed lines represent the predicted values by the model built on one of the four products given in the legend.

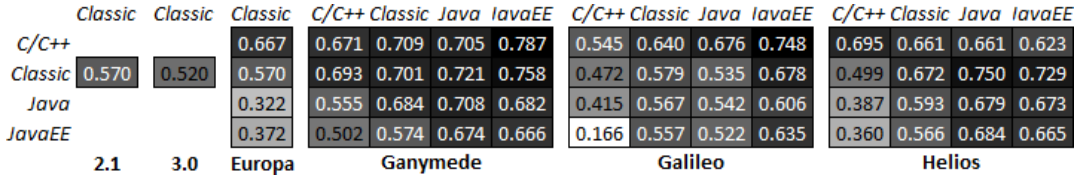
**very accurately the nTop20% scores of the three new products introduced in the Europa release (i.e., C/C++, Java, and JavaEE).**

Moreover, 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 8 showed that these two groups of particular models performed worse than others due to an outlier package in each case, as detailed next.

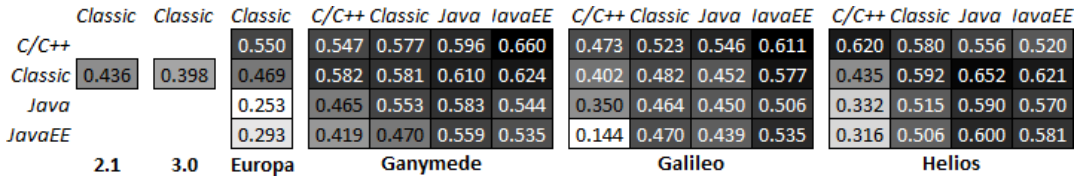
Models built from the Ganymede version of C/C++ provided uncharacteristically poor predictions for all products in the Galileo release (see the first column of the Galileo release in




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




(b) A heat map of the values for the performance metric **Spearman's** $\rho$. *Higher* values represent better results and are shaded darker.




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

Fig. 9: Heat maps showing the results of the predictive trials for each product, in each release. The releases on which the predictions were made are shown on the bottom. The predictive models were built on the previous release. The columns represent the products on which predictive models were *built*, and 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 by the product name. Each value in the table represents a performance metric score achieved by evaluating the predictions made for that release of the row-labeled product, using the model built on the previous release of the column labeled product.

Figure 9(a)). The model built on Ganymede's C/C++ data predicted fewer faulty packages for every product than other models. 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. These are shown respectively in the second, third, and fourth rows of the third column in Figure 8.

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 10, most products showed a similar distribution of faults, with one very faulty package followed by several packages with a relatively high number of faults, then a small spread of the majority of packages with very few faults. However, as the boxplot in Figure 10 shows, Ganymede's C/C++ had one very faulty package and a large difference with the next faulty package. This absence of packages between the faultiest packages and those with relatively few faults created an "all or nothing" effect in the predictive model.

Predictions for the Helios versions of Java and JavaEE made by the model built from the Galileo versions of Classic, Java, and JavaEE were also worse than others (see the last table in Figure 9(a)). The data revealed that the faultiest package in the Helios release had 1,021 total post-release faults and was a low-reuse variation package shared by Java and JavaEE. This package,

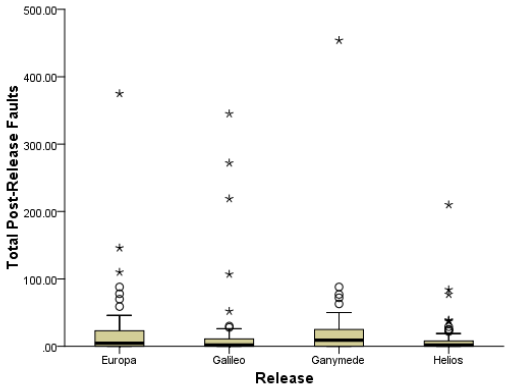


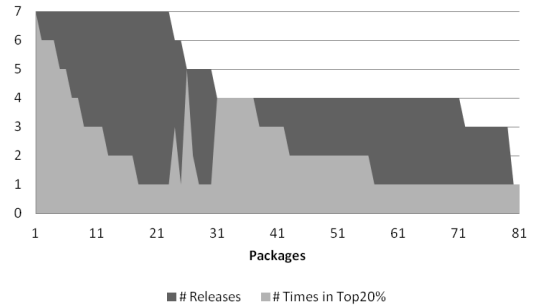

Fig. 10: A boxplot of the distributions of post-release faults of C/C++ product across releases. Notice the one extremely faulty package in the Ganymede release compared to the other packages in that release.

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 8), 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. Looking more closely, it appeared that this particular package was not identified among the faultiest 20% of packages in the Helios release because it had different values for 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).

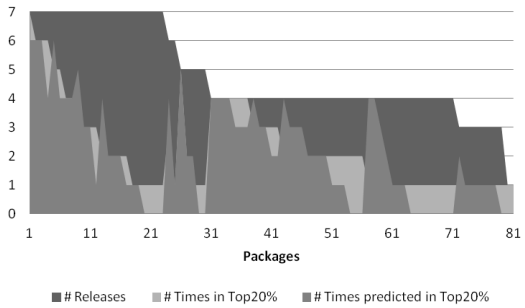
We also tracked the fault proneness of individual packages as they evolved through releases. As shown in Figure 11a, many packages did not appear among the 20% most faulty packages for every release in which they existed, meaning that their fault proneness did not persist across all releases. This shows that while some packages may be identified by developers as "the usual suspects", there are many other packages that are among the faultiest in only one or two releases. Identification of these intermittently faulty packages is not a trivial problem and is likely to be quite useful to development teams. Figure 11b shows that **our predictive models were able to identify not only the usual suspects (i.e., packages that were consistently among the most fault prone in the releases they existed in), but also to identify accurately the intermittently faulty packages, with an overall accuracy close to 80%.**

Next, we discuss the two additional performance metrics, Spearman's $\rho$ and Kendall's $\tau_b$, which are shown in Figures 9(b) and 9(c), respectively. Each of these two correlation coefficients measures the association between two ranked lists of packages: the list ranked based on the actual number of post-release faults and the list ranked based on the number of post-release faults predicted by our models. It should be noted that, in general, the values of Spearman's $\rho$ are higher than the values of Kendall's $\tau_b$. As discussed in Section 5.5.2, Kendall's $\tau_b$ has some advantages for measuring the association of two ranked lists. Nevertheless, we present the Spearman's $\rho$ values as well in order to be able to compare our results with the results presented in the related work.

Specifically, we compare our results shown in Figure 9(b) with the related works which used Eclipse as a case study (D'Ambros et al, 2009, 2010; Zimmermann et al, 2007). It should be noted that none of these three previous studies treated Eclipse as a product line, that is, they did not consider reuse across products and its effect on quality and predictions. D'Ambros et al (2009) examined two components of Eclipse from releases 3.1 and 3.3 using 50-fold cross validation of




(a) For each package shown on x-axis, y-axis shows the total number of releases it appeared in and the number of releases it was among the 20% of most faulty packages.




(b) For each package shown on x-axis, y-axis shows the total number of releases it appeared in, the number of releases it was among the 20% of most faulty packages, and the number of times our predictive models ranked it among the 20% of most faulty packages.

Fig. 11: Visualizations of the fault proneness for 81 packages that were among the 20% of the most faulty packages in at least one of the seven releases considered in this paper.

regression models built on different combinations of metrics. Spearman correlation $\rho$ values ranged from 0.4 to 0.81. D'Ambros et al (2010) evaluated many different prediction approaches on several components of Eclipse versions 3.1, 3.4, and 3.4.1. The Spearman's $\rho$ values attained when using the same change metrics as in this study ranged from 0.165 to 0.534 and, when using comparable static code metrics, ranged from 0.277 to 0.547. The values of Spearman's $\rho$ shown in Figure 9(b) are comparable with values reported by D'Ambros et al (2009, 2010). It should be noted, however, that exact comparison with D'Ambros et al (2009, 2010) is not possible because they used cross validation and studied different releases of Eclipse. More accurate comparison can be made with the results presented by Zimmermann et al (2007), which were based on using logistic regression models built from 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 $\rho$ were 0.420 for the release 2.1 predictions and 0.449 for the release 3.0 predictions. Our results for these same combinations of Classic releases, as shown in Figure 9(b), are better (i.e., 0.570 and 0.520, respectively), which could be due to the addition of change metrics to our features and/or to the different machine learning technique we employed.

In summary, results presented in Figures 9(a)-(c) show that **models built from the data of one release could accurately predict the most fault prone packages in the subsequent release from the pre-release data of that release. More importantly, the prediction model built on Classic, the only existing product in release 3.0, was able to very accurately predict the fault proneness of the three new products (i.e., C/C++, Java, and JavaEE) introduced for the first time in the Europa release. Furthermore, rankings of fault prone packages created by our models were positively correlated to the actual rankings, and the values are comparable or better than those in related works using Eclipse as a case study.**

7.2 RQ4: Do the predictions of the most fault prone packages benefit from the available data for other products? In other words, do cross-product predictions improve the accuracy?

This research question considers the benefit of the product line approach for the prediction of post-release fault proneness. This has not been previously explored in the related works, including

our previous work (Devine et al, 2012; Krishnan et al, 2011b,a, 2012), and is of significant interest to SPL developers.

As shown in Figure 9(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 data available for 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 9(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., regardless of the product from the previous release from which a model is constructed, it will consistently make accurate predictions for C/C++ and Classic.* As shown in Figure 2, 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 the 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 RQ4.

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 include 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 RQ4.

Based on these observations for the nTop20% metric, we conclude that models produced better results when built on larger products with more variation (non-shared packages) than when built on smaller products. Models also produced better predictions for smaller products (consisting mostly of common packages) than they did for larger products. This is evidence that prediction of the most fault prone packages benefited from data available from other products, that is, cross-product predictions provided accurate results.

Next, we explore whether these trends remain valid with respect to the performance metrics that measure how well packages are ranked based on predicted numbers of post-release faults. As shown in Figures 9(b) and 9(c), while some of the best values for Spearman's $\rho$ and Kendall's $\tau_b$ appeared on the main diagonals, at least three fourths of the best rankings for both performance metrics were made by models built from previous releases of other products. As in the case of the nTop20% metric, this suggests that post-release fault rankings for products in a product line may benefit from the data available from other members of the product line family. When examining the results for values of Spearman's $\rho$ and Kendall's $\tau_b$, row and column patterns also emerged. As 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 $\tau_b$ and Spearman's $\rho$, as they did with the nTop20% metric.

In summary, the results based on all three performance metrics are consistent and show that **models produced the best results when built from larger products with more variation (non-shared packages), and when making predictions on smaller products consisting mostly of common packages. That is, cross-product predictions produced more accurate results.**

It should be noted that in a non-SPL context the cross-project predictions were not successful. The closest to our work is a study which used regression with five principal components as independent variables to predict the post-release faults of five Microsoft products (Nagappan et al, 2006). The results of cross-project predictions were mixed – some product histories could serve as predictors for other products, while most could not. The authors concluded that predictions

were accurate only when obtained from the same or similar products. Another group of non-SPL related papers was focused on classification of software units into fault-prone and fault-free. Zimmermann et al (2009) used 12 real-world software applications (i.e., non-SPL products) and ran 622 cross-project predictions. That work showed "an alarmingly low success rate of 3.4%", where a prediction was considered successful if the precision, recall, and accuracy values were above 0.75. Interestingly, even using models from applications in the same domain or with the same process did not lead to accurate predictions. Attempts to improve the performance of cross-project fault classification were based on several approaches, such as data transformation, data selection, and transfer learning. The data transformation approach was based on adjusting the average values of each metric in the training and testing sets (Watanabe et al, 2008). The data selection approaches included a Nearest Neighbors filter (Turhan et al, 2009), brute force selection method based on combination of projects (He et al, 2012), and selection of similar projects combined with feature subset selection (He et al, 2013). Transfer learning methods, on the other side, tried to improve the classification algorithmically (Ma et al, 2012; Nam et al, 2013). In all these cases the improvements were in terms of the average performance metrics, with not all cross-project prediction being improved and, more importantly, resulted in cross-project predictions that were worse than within project predictions.

## 8 Generalizability of the findings and implications for software product lines development

In this section we synthesize the findings from this study with our findings based on a much smaller industrial software product line of software testing tools, called PolyFlow (Devine et al, 2012) with a goal to identify the results that generalize across different software product lines. It should be noted that the study of the industrial product line (Devine et al, 2012) was focused only on pre-release faults (data on post-release faults were not available) and the sample size of predictions was small (i.e., pre-release faults were predicted only for two components). Nevertheless, the following findings are consistent across both case studies and provide insights into characteristics of reuse across products in a software product line, and the benefit of such reuse to the quality of products and accuracy of predictions.

- The cross-product reuse in software product lines is not 'all or nothing', that is, packages are either reused in all products or they are implementing variabilities used in a single product only. On the contrary, there is *a wide spectrum of levels of cross-product reuse*, from common packages shared among all products, to high-reuse and low-reuse variation packages shared among some, but not all products, to single-use 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*, but tend to have *low fault densities*. This is not always true for newly developed components/packages.
- Predictions of pre-release faults in case of PolyFlow and post-release faults in the case of Eclipse can be done accurately from pre-release data. Moreover, *predictions benefit from data available from other products in the software product line.*

The consequences of the data-driven investigations described in this paper and in our previous work (Devine et al, 2012) for product line developers lie primarily in the intended future use of the empirical results to assist in decisions as to which techniques are best suited to product line change and quality management. We describe here three areas in which these works can inform and improve product line development.

- *Assessment of product line evolvability.* It is especially important in a product line to assess how readily its domain engineering assets will support change. For example, Breivold et al

(2012) in their recent review of software architecture evolution research, described several techniques that have been suggested to assess evolvability. Data-driven analyses such as those presented here can help in evaluating these techniques, in determining which of these are well suited to product lines, and in customizing the use of techniques to anticipate change to the attributes (size in number of products, legacy or greenfield, commercial or open-source, etc.) of individual product lines.

- *Accurate, early prediction of quality in new products.* One of the most promising results from the work reported here is related to cross-product predictions, that is, to the fact that using data from previous products enabled good predictions of the quality of new products. Being able to predict with some confidence which packages are likely to be among the most fault-prone leads to better decisions as to where to place scarce testing resources and when to deploy new products.
- *A need for better traceability and identification of the rationales for changes.* Some questions that product line organizations ask are currently stymied by a lack of data. Improving the traceability and identification of the rationales for changes, including bug fixes, is needed to support the improvement of product line practices. In the case of Eclipse, we discussed with an IBM manager additional traceability links that, if added, will improve the variability management of Eclipse. In the industrial product line PolyFlow, missing information about who had touched the code added uncertainty. Therefore, we collaborated with the lead developer and suggested additional fields to be added to the product line's bug-reporting database. In general, knowledge management tends to be an even bigger problem for successful product lines than for individual systems because of the change across both releases and products.

## 9 Threats to validity

In this section, we describe several threats to the validity of this study and the measures taken 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 paper 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 types 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 our 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 faultiest package in that release and therefore 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 might have led to better 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 total, for the named releases (i.e., Europa to Helios) we were able to retrieve archived source code for 91.6% of the 136,567 total files for which CVS data was retrieved. This is important because these are the four releases for which Eclipse followed the SPL approach, and thus are our main focus. For the source code of the older releases 2.0, 2.1, and 3.0 of Classic we were able to download the source code for only 18,111 of the 41,416 files. While this number seems low, it is comparable to the amount of data collected by Zimmermann et al (2007). In particular, based on the data for the 2.0, 2.1, and 3.0 releases available from the repository given by Zimmermann et al (2007), their dataset consisted of 15,395 files.

We built our dataset by combining static code metrics with change metrics collected by Krishnan et al (2012). As described briefly in Section 4.1, 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 Attic. These difficulties, as well as how they were overcome and how the data set was validated, are detailed by Krishnan et al (2012), 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, Friedman test and post-hoc multiple comparison test, and Page test) to explore the trend of 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 $\tau_b$ correlation coefficients, which have minimal assumptions. We also used appropriate versions of these statistics for datasets with many ties.

Validation of the results is an important and necessary part of any empirical study. One way to validate the results could be to present them to the software development team and use their feedback as validation. This feedback was unattainable despite our attempts to contact Eclipse developers. However, our empirical predictions were validated by the real, reported post-release faults in the bug tracking system. By building our models on one release and then using them to make predictions of the post-release faults in the following release, we were able to compare our predictions to the real number of reported post-release faults.

**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. Therefore, whenever possible, we compared the observations made in this paper with the relevant results in the previously published works that were based on Eclipse and other software systems. We also synthesized the findings of this study with the findings from our previous work based on an industrial case study (Devine et al, 2012) to contribute towards identification of characteristics that are invariant across multiple SPLs. In addition, in this paper we presented the 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 paper, which support objective comparison of our results with the results of future case studies.

## 10 Conclusion

Software product line engineering is a paradigm for reuse which is widely used to develop high-quality software product families faster and with less cost than traditional development methods. Real world case studies of SPLs are necessary both to empirically evaluate the benefit of product

line engineering and to improve the development and maintenance process by supplying actionable insights into how SPLs behave in practice.

The main goals of this paper are to explore how the two dimensions of software reuse, i.e., cross-product reuse and cross-release reuse, affect the quality of SPL and our ability to accurately predict fault proneness. The results 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 code 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, included over 135,000 files containing 20 million lines of code, aggregated into packages and described by 112 different features based on the static code and change metrics.

Our main findings are summarized in Table 6. The top part of the table summarizes the quality assessment results, and the bottom part summarizes the prediction results. The first column reports the *findings*; the second column provides the location in the paper of the *evidence* backing up each finding; and the final column shows how each finding *compares* to results reported in related works, if any.

Table 6: Summary of the main findings

| **Findings** | **Location of evidence** | **Related results** |
|---|---|---|
| **Findings related to quality assessment** | | |
| Quality, measured in number of post-release faults, improves as the SPL evolves across releases. | RQ1: Figures 6a & 6c | In non-SPL context consistent with (Khoshgoftaar and Seliya, 2004), (Ostrand et al, 2004), (Ostrand and Weyuker, 2002), and inconsistent with (Fenton and Ohlsson, 2000). Inconsistent with the result of a much smaller SPL (Mohagheghi and Conradi, 2008). |
| Across releases, for 'old' packages, only single-use packages showed decreasing change proneness; the common, low-reuse variation, and single-use packages experienced improvement in quality, but high-reuse variation packages did not. Within each release, for 'old' packages, change and fault proneness were not different for different levels of cross-product reuse. | RQ2: Figure 7 (top two rows) & Tables 4 and 5 | The across-releases results further the very preliminary study of different levels of cross-product reuse (Krishnan et al, 2011a). Within-release findings are reported for the first time in this paper. |
| **Findings related to post-release fault prediction** | | |
| Pre-release data consistently predicts the top 20% of faulty packages, post-release. Best nTop20% scores are from 76% to 97%, for each product across all releases. Pre-release data accurately ranks packages based on predicted numbers of post-release faults. | RQ3: Figures 9a, 9b & 9c | Not explored previously in a SPL context. |
| Fault proneness predictions benefit from data available for other products of the SPL. Models produce better results when built from larger products and when making predictions/rankings on smaller products. | RQ4: Figures 9a, 9b & 9c | Explored for the first time in this paper for cross-product predictions. In non-SPL context the cross-project predictions did not show promising results (Nagappan et al, 2006). |

Briefly, the assessment results showed that the SPL quality improved as it evolved across the seven releases considered in this paper. Furthermore, as the product line continued to evolve across releases, previously existing (i.e., 'old') common packages, high-reuse variation packages and low-reuse variation packages continued to change; only single-use packages experienced decreasing change proneness as they matured across releases. While the previously existing high-reuse variation packages did not exhibit improvement in quality (measured by the post-release fault density), common packages, low-reuse variation packages, and single-use packages improved in quality. The most surprising assessment result is that common, high-reuse variation, low-reuse variation, and single-use packages, among 'old' packages within each release, did not experience a statistically significant difference in change proneness and fault proneness (measured in post-release fault density).

Ranked predictions of post-release faults were made using generalized linear regression models. These models treated the distribution of post-release faults as an ordered multinomial distribution and used the cumulative negative log log linking function, which is particularly well-suited for modeling skewed distributions, such as post-release fault data. The results showed that models built from the data of one release could accurately predict the most fault prone packages in the

subsequent release from the pre-release data of that release. Furthermore, rankings of fault prone packages created by our models were positively correlated to the actual rankings.

The most interesting finding, from a product line perspective, is that the best predictive models for each product were built from pre-release data that included other products. This means that the predictions benefited from the use of data from other products. 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.

Finally, as described in Section 8, several of the main findings reported in this paper are consistent with findings from our earlier work on an industrial SPL. These data-driven investigations clearly show that the interrelationship between cross-product reuse and cross-release reuse is more complicated and less static than SPL techniques routinely support. The results therefore suggest that SPL development can benefit from better ways to characterize and support ongoing change across the entire spectrum of common, high-reuse, and low-reuse components to sustain and predict quality across the lifetime of the products and the product line.

## Acknowledgments

This work was supported in part by the National Science Foundation grants 0916275 and 0916284 with funds from the American Recovery and Reinvestment Act of 2009 and by the WVU ADVANCE Sponsorship Program funded by the National Science Foundation ADVANCE IT Program award HRD-100797. Part of this work was performed while Robyn Lutz was visiting the California Institute of Technology.

## References

Agresti A (2010) Analysis of Ordinal Categorical Data. John Wiley and Sons, Inc., Hoboken, NJ, USA

Andersson C, Runeson P (2007) A replicated quantitative analysis of fault distributions in complex software systems. IEEE Transactions on Software Engineering 33:273–286

Bell RM, Ostrand TJ, Weyuker EJ (2006) Looking for bugs in all the right places. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis, ISSTA '06, pp 61–72

Bibi S, Tsoumakas G, Stamelos I, Vlahvas I (2006) Software defect prediction using regression via classification. In: Proceedings of the IEEE International Conference on Computer Systems and Applications, AICCSA '06, pp 330–336

Bingham NH, Fry JM (2010) Regression: Linear Models in Statistics, 1st edn. Springer-Verlag, London, UK

Boehm B, Basili VR (2001) Software defect reduction top 10 list. Computer 34:135–137

Breivold HP, Crnkovic I, Larsson M (2012) A systematic review of software architecture evolution research. Information & Software Technology 54(1):16–40

Chastek G, McGregor J, Northrop L (2007) Observations from viewing Eclipse as a product line. In: Proceedings of the 3rd International Workshop on Open Source Software and Product Lines, pp 1–6

D'Ambros M, Lanza M, Robbes R (2009) On the relationship between change coupling and software defects. In: Proceedings of the 16th Working Conference on Reverse Engineering, WCRE '09, pp 135–144

D'Ambros M, Lanza M, Robbes R (2010) An extensive comparison of bug prediction approaches. In: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories, MSR '10, pp 31 –41

D'Ambros M, Lanza M, Robbes R (2012) Evaluating defect prediction approaches: A benchmark and an extensive comparison. Empirical Software Engineering 17:531–577

Devine T, Goseva-Popstajanova K, Krishnan S, Lutz R, Li J (2012) An empirical study of pre-release software faults in an industrial product line. In: Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation, ICST '12, pp 181 –190

Fenton NE, Ohlsson N (2000) Quantitative analysis of faults and failures in a complex software system. IEEE Transactions on Software Engineering 26:797–814

Frakes WB, Succi G (2001) An industrial study of reuse, quality, and productivity. Journal of Systems and Software 57:99–106

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

van Gurp J, Prehofer C, Bosch J (2010) Comparing practices for reuse in integration-oriented software product lines and large open source software projects. Software Practice & Experience 40(4):285–312

Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A systematic review of fault prediction performance in software engineering. IEEE Transactions on Software Engineering 38(6):1276 – 1304

Hamill M, Goseva-Popstojanova K (2009) Common trends in software fault and failure data. IEEE Transactions on Software Engineering 35:484–496

He Z, Shu F, Yang Y, Li M, Wang Q (2012) An investigation on the feasibility of cross-project defect prediction. Automated Software Engineering 19(2):167–199

He Z, Peters F, Menzies T, Yang Y (2013) Learning from open-source projects: An empirical study on defect prediction. In: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM'13, pp 45–54

Kamei Y, Matsumoto S, Monden A, Matsumoto Ki, Adams B, Hassan AE (2010) Revisiting common bug prediction findings using effort-aware models. In: Proceedings of the 2010 IEEE International Conference on Software Maintenance, ICSM '10, pp 1–10

Kastro Y, Bener AB (2008) A defect prediction method for software versioning. Software Quality Control 16(4):543–562

Khoshgoftaar T, Munson J (1990) Predicting software development errors using software complexity metrics. IEEE Journal on Selected Areas in Communications 8(2):253 –261

Khoshgoftaar TM, Seliya N (2004) Comparative assessment of software quality classification techniques: An empirical case study. Empirical Software Engineering 9(3):229–257

Kitchenham B, Mendes E (2009) Why comparative effort prediction studies may be invalid. In: Proceedings of the 5th International Conference on Predictor Models in Software Engineering, PROMISE '09, pp 4:1–4:5

Kleinbaum DG, Kupper LL, Muller KE (eds) (1988) Applied regression analysis and other multivariable methods. PWS Publishing Co., Boston, MA, USA

Krishnan S, Lutz RR, Goseva-Popstojanova K (2011a) Empirical evaluation of reliability improvement in an evolving software product line. In: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, pp 103–112

Krishnan S, Strasburg C, Lutz RR, Goseva-Popstojanova K (2011b) Are change metrics good predictors for an evolving software product line? In: Proceedings of the 7th International Conference on Predictive Models in Software Engineering, PROMISE'11, pp 7:1–7:10

Krishnan S, Strasburg C, Lutz RR, Goseva-Popstojanova K, Dorman KS (2012) Predicting failure-proneness in an evolving software product line. Information and Software Technology 55(8):1479–1495

Kutner MH, Nachtsheim CJ, Neter J (2004) Applied Linear Regression Models, forth edn. McGraw-Hill/Irwin, New York, NY

Laffra C, Veys N (2013) Where did Eclipse come from? `http://wiki.eclipse.org/FAQ_Where_did_Eclipse_come_from%3F`

Li PL, Herbsleb J, Shaw M, Robinson B (2006) 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, ICSE '06, pp 413–422

Lim W (1994) Effects of reuse on quality, productivity, and economics. IEEE Transactions on Software Engineering 11(5):23 –30

van der Linden F (2009) Applying open source software principles in product lines. CEPSUS UPGRADE, The European Journal for the Informatics Professional 10:32–40

van der Linden F (2013) Open source practices in software product line engineering. In: Lucia A, Ferrucci F (eds) Software Engineering, Lecture Notes in Computer Science, vol 7171, Springer Berlin Heidelberg, pp 216–235

Ma Y, Luo G, Zeng X, Chen A (2012) Transfer learning for cross-company software defect prediction. Information and Software Technology 54(3):248–256

Mansfield D (2012) CVSps-patchsets for CVS. URL `http://www.cobite.com/cvsps`

McConnell S (2004) Code Complete, Second Edition. Microsoft Press, Redmond, WA, USA

McCullagh P, Nelder J (1983) Generalized Linear Models. Monographs on Statistics and Applied Probability, Chapman and Hall, New York, NY, USA

Mohagheghi P, Conradi R (2008) An empirical investigation of software reuse benefits in a large telecom product. ACM Transactions on Software Engineering Methodology 17:13:1–13:31

Mohagheghi P, Conradi R, Killi O, Schwarz H (2004) An empirical study of software reuse vs. defect-density and stability. In: 26th International Conference on Software Engineering, ICSE '04, pp 282 – 291

Moser R, Pedrycz W, Succi G (2008) 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, ICSE '08, pp 181 –190

Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pp 452–461

Nam J, Pan SJ, Kim S (2013) Transfer defect learning. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pp 382–391

Nelder JA, Wedderburn RWM (1972) Generalized linear models. Journal of the Royal Statistical Society Series A (General) 135(3):pp. 370–384

Norušis MJ (2012) IBM SPSS Statistics 19 Advanced Statistical Procedures Companion. Prentice Hall, A division of Pearson Education, Upper Saddle River, NJ, USA

Ohlsson N, Alberg H (1996) Predicting fault-prone software modules in telephone switches. IEEE Transactions on Software Engineering 22(12):886–894

Ostrand TJ, Weyuker EJ (2002) The distribution of faults in a large industrial software system. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '02, pp 55–64

Ostrand TJ, Weyuker EJ, Bell RM (2004) Where the bugs are. In: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'04, pp 86–96

Ostrand TJ, Weyuker EJ, Bell RM (2005) Predicting the location and number of faults in large software systems. IEEE Transactions on Software Engineering 31(4):340 – 355

Ostrand TJ, Weyuker EJ, Bell RM (2010) Programmer-based fault prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, PROMISE'10, pp 19:1–19:10

Pohl K, Böckle G, Linden FJvd (2005) Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA

Selby R (2005) Enabling reuse-based software development of large-scale systems. IEEE Transactions on Software Engineering 31(6):495 – 510

Shin Y, Bell R, Ostrand T, Weyuker E (2009) Does calling structure information improve the accuracy of fault prediction? In: Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories, MSR '09, pp 61 –70

Shull FJ, Carver JC, Vegas S, Juristo N (2008) The role of replications in empirical software engineering. Empirical Software Engineering 13(2):211–218

SourceMonitor (2011) Version 3.2. URL `http://www.campwoodsw.com/sourcemonitor.html`

Taylor RN (2013) The role of architectural styles in successful software ecosystems. In: Proceedings of the 17th International Software Product Line Conference, SPLC '13, pp 2–4

Thomas WM, Delis A, Basili VR (1997) An analysis of errors in a reuse-oriented development environment. Journal of Systems and Software 38:211–224

Thompson JM, Heimdahl MPE (2003) Structuring product family requirements for n-dimensional and hierarchical product lines. Requirements Engineering 8(1):42–54

Turhan B, Menzies T, Bener AB, Di Stefano J (2009) On the relative value of cross-company and within-company data for defect prediction. Empirical Software Engineering 14(5):540–578

Watanabe S, Kaiya H, Kaijiri K (2008) Adapting a fault prediction model to allow inter language reuse. In: Proceedings of the 4th International Workshop on Predictor Models in Software Engineering, PROMISE '08, pp 19–24

Weiss DM, Lai CTR (1999) Software product-line engineering: A family-based software development process. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Weyuker EJ, Ostrand TJ, Bell RM (2008) Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models. Empirical Software Engineering 13(5):539–559

Zhang W, Jarzabek S (2005) Reuse without compromising performance: Industrial experience from RPG software product line for mobile devices. In: Software Product Lines, LNCS vol. 3714, pp 57–69

Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for Eclipse. In: Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering, PROMISE'07, p 9

Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE '09, pp 91–100

## Appendix: Aggregation metrics

The static code and change metrics were collected at file-level and then were aggregated to the package level, as specified in Tables 7 and 8. As a result, each package was characterized by a vector $\mathbf{m}$ of 112 metrics (i.e., features), where $\mathbf{m}[i], i = 1, \ldots, 73$ are static code metrics, while $\mathbf{m}[i], i = 74, \ldots, 112$ are change metrics.

Table 7: Aggregations applied to each static code metric

| **Static code metric** | **Aggregation levels** |
|---|---|
| *LOC* | Mean, Median, Max, Sum |
| *Statements* | Mean, Median, Max, Sum |
| *Percent Branch Statements* | Mean, Median, Max |
| *Method Call Statements* | Mean, Median, Max, Sum |
| *Percent Lines with Comments* | Mean, Median, Max |
| *Classes and Interfaces* | Mean, Median, Max, Sum |
| *Methods per Class* | Mean, Median, Max |
| *Ave Statements per Method* | Mean, Median |
| *Max Complexity* | Max |
| *Ave Complexity* | Mean, Median |
| *Max Block Depth* | Max |
| *Ave Block Depth* | Mean, Median |
| *Statements at Block Level 0* | Mean, Median, Max, Sum |
| *Statements at Block Level 1* | Mean, Median, Max, Sum |
| *Statements at Block Level 2* | Mean, Median, Max, Sum |
| *Statements at Block Level 3* | Mean, Median, Max, Sum |
| *Statements at Block Level 4* | Mean, Median, Max, Sum |
| *Statements at Block Level 5* | Mean, Median, Max, Sum |
| *Statements at Block Level 6* | Mean, Median, Max, Sum |
| *Statements at Block Level 7* | Mean, Median, Max, Sum |
| *Statements at Block Level 8* | Mean, Median, Max, Sum |
| *Statements at Block Level 9* | Mean, Median, Max, Sum |

Table 8: Aggregations applied to each change metric

| **Change metric** | **Aggregation levels** |
|---|---|
| *Revisions* | Mean, Median, Max, Sum |
| *Refactorings* | Mean, Median, Max, Sum |
| *Bugfixes* | Mean, Median, Max, Sum |
| *Authors* | Mean, Median, Max, Sum |
| *LOC Added* | Sum |
| *Max LOC Added* | Max |
| *Ave LOC Added* | Mean, Median |
| *LOC Deleted* | Sum |
| *Max LOC Deleted* | Max |
| *Ave LOC Deleted* | Mean, Median |
| *Codechurn* | Sum |
| *Max Codechurn* | Max |
| *Ave Codechurn* | Mean, Median |
| *Max Changeset* | Max |
| *Ave Changeset* | Mean, Median |
| *Age* | Mean, Median, Max, Sum |
| *Weighted Age* | Mean, Median, Max, Sum |