

2013

Evidence-based defect assessment and prediction for software product lines

Sandeep Krishnan
Iowa State University

Follow this and additional works at: <https://lib.dr.iastate.edu/etd>

 Part of the [Computer Sciences Commons](#)

Recommended Citation

Krishnan, Sandeep, "Evidence-based defect assessment and prediction for software product lines" (2013). *Graduate Theses and Dissertations*. 12993.

<https://lib.dr.iastate.edu/etd/12993>

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

**Evidence-based defect assessment and prediction
for software product lines**

by

Sandeep Krishnan

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:

Robyn R. Lutz, Major Professor

David M. Weiss

Samik Basu

Hridayesh Rajan

Karin S. Dorman

Katerina Goseva-Popstojanova

Iowa State University

Ames, Iowa

2013

Copyright © Sandeep Krishnan, 2013. All rights reserved.

DEDICATION

To my teacher.

TABLE OF CONTENTS

LIST OF TABLES	vi
LIST OF FIGURES	viii
ACKNOWLEDGEMENTS	x
ABSTRACT	xii
CHAPTER 1. INTRODUCTION	1
1.1 Problem Statement	1
1.2 Research Questions	4
1.3 Approach	5
1.4 Results	6
1.5 Contributions	7
1.6 Dissertation Outline	8
CHAPTER 2. BACKGROUND AND RELATED WORK	9
2.1 Software Product Lines	9
2.2 Eclipse Product Line	10
2.2.1 Products	11
2.2.2 Components	12
2.3 Software Defect Assessment	13
2.4 Software Defect Prediction	16
CHAPTER 3. APPROACH	19
3.1 Defect Assessment	19

3.1.1	Failure Trends in an Evolving SPL	20
3.1.2	Change Trends in an Evolving SPL	21
3.2	Defect Prediction	23
3.2.1	Data Collection, Integration and Validation	23
3.2.2	Types of Datasets	27
3.2.3	Data Analysis	30
CHAPTER 4. DEFECT ASSESSMENT RESULTS		35
4.1	Research Questions	35
4.2	Results	36
4.2.1	Failure Trends	36
4.2.2	Change Trends	40
4.2.3	Failure/Change Trends	43
4.3	Discussion of the results	45
CHAPTER 5. DEFECT PREDICTION RESULTS		47
5.1	Research Questions	47
5.2	Classifier Selection	48
5.3	Single Product Evolution	50
5.3.1	How do our results related to learner’s performance compare with previously published results?	51
5.3.2	Does learner performance improve as a single product evolves? . . .	53
5.3.3	Is the set of prominent predictors consistent across releases of a single product?	57
5.4	Product Line Evolution	60
5.4.1	Does learner performance improve as the product line evolves? . . .	60
5.4.2	Is the set of prominent predictors consistent across products as the product line evolves?	64
5.5	Evolution of Components at Different Levels Of Reuse	66

5.5.1	Does the learner performance improve for components in each category of reuse and across categories of reuse?	66
5.5.2	Is there a common set of best predictors across all categories of reuse?	69
5.6	Prediction with Incrementally Increasing Data Collection Periods	70
5.7	Summary of the Results	72
CHAPTER 6. THREATS TO VALIDITY		74
CHAPTER 7. CONTRIBUTIONS AND FUTURE WORK		78
7.1	Contributions	78
7.2	Future Work	84

LIST OF TABLES

2.1	List of Components	13
3.1	Number of Total and Severe Failures	20
3.2	List of Change Metrics [Moser et al., 2008b]	25
3.3	Base Probability for All Releases for Multiple Products of Eclipse	31
3.4	Confusion Matrix	32
4.1	Percentage of New Files for Commonalities and Variabil- ities	42
4.2	Failures/New-File for Commonalities and Variabilities .	43
4.3	Failures/Kchanges for Commonalities and Variabilities .	43
5.1	List of Classifiers	49
5.2	Comparison of Classification Performance for 2.0, 2.1, and 3.0 Releases of Eclipse Classic for UsePre_PredictPost Dataset	52
5.3	Comparison of Results for Newer Releases (3.3-3.6) With Older Releases (2.0, 2.1, 3.0) of Eclipse Classic	55
5.4	Comparison of Prominent Predictors for Older Releases of Eclipse Classic	59
5.5	Prominent Predictors for Newer Releases of Eclipse Classic	60
5.6	Performance Trends for All Products	63

5.7	Prominent Predictors at Product Level	65
5.8	Performance Trends for Components at Different Levels of Reuse	68
5.9	Prominent Predictors for Components at Different Levels of Reuse	69
7.1	Severe Failures in Reused Code for Component Categories	81
7.2	Comparing Cross-release Classification Performance Us- ing Static Code Metrics and Change Metrics	82

LIST OF FIGURES

1.1	Product Line Evolution across two dimensions	2
2.1	Eclipse Product Line for year 2010	12
3.1	Data Timeline of Eclipse Classic	23
3.2	Data Collection Process	24
4.1	Number of Severe Failures in Common Components . . .	37
4.2	Percentage of Severe Failures in Common Components .	38
4.3	Number of Severe Failures in Variation Components with High Reuse	39
4.4	Percentage of Severe Failures in Variation Components with High Reuse	39
4.5	Number of Severe Failures in Variation Components with Low Reuse	40
4.6	Percentage of Severe Failures in Variation Components with Low Reuse	41
5.1	CD Diagram for AUC and TPR Ranks of UseAll_PredictAll Dataset	51
5.2	CD Diagram for AUC and TPR Ranks of UseAll_PredictPost Dataset	52

5.3	CD Diagram for AUC and TPR Ranks of UsePre_PredictPost Dataset	53
5.4	PC, TPR and FPR Comparison of Eclipse Products Across Releases for UseAll_PredictAll Dataset	62
5.5	PC, TPR and FPR Comparison of Eclipse Products Across Releases for UseAll_PredictPost Dataset	62
5.6	PC, TPR and FPR Comparison of Eclipse Products Across Releases for UsePre_PredictPost Dataset	62
5.7	Incremental Prediction for Four Eclipse Products	71
5.8	Incremental Prediction for Three Reuse Categories	72

ACKNOWLEDGEMENTS

I would like to thank Dr. Robyn Lutz for her encouragement and support throughout my Ph.D. She gave me the confidence to pursue this research and provided valuable feedback on the different research directions. I attribute all the research skills I have learned to her. She has taught me how to not be let down by failures in research. I am grateful to Dr. Katerina Goseva-Popstojanova for her insightful inputs and directions in this research. I cannot sufficiently thank her for her contributions to the research ideas and to the writing of the papers.

I am thankful to Dr. Karin Dorman for her valuable guidance in this research. I have certainly learned a lot from her inputs and from the statistical analysis she conducted in this research. However, I am more grateful to her for teaching me how to be enthusiastic about different areas of research and to come up with thought provoking questions. I thank my other committee members, Dr. David Weiss, Dr. Hriday Rajan and Dr. Samik Basu for their valuable time and inputs throughout this research. I thank my colleague Chris Strasburg for his valuable time and efforts in this research. His insightful comments made me think more deeply about my research questions.

I am deeply indebted to Dr. P. V. Krishnan for giving me the right direction in life when I needed it the most. He has taught me that education is meant to bring about good character. I can never forget the principles he has inculcated in me and I am very grateful to him for his relentless support throughout this journey of mine. I aim to use this degree for the right cause, as he has always taught me.

I am grateful to my parents, Mr. and Mrs. Krishnan and my brother Mr. Satish Krishnan for encouraging me to pursue this program. I thank my friends Dr. Kasthuri-

rangan Gopalakrishnan, Dr. Siddhartha Khaitan, Dr. Venkat Krishnan, Dr. Ganesh Ram Santhanam, Dr. Amit Pande, Dr. Ankit Agrawal, Dr. Abhisek Mudgal, Dr. Siva Swaminathan, Sparsh Mittal and other friends for their association and for giving me the necessary moral support in the most crucial situations of life. Their support never made me feel I was away from home.

I would like to acknowledge the help of all my lab members, Dr. Jing (Janet) Liu, Dr. Hongyu Sun and Jingwei Yang. They provided me valuable suggestions to my research and also a wonderful work atmosphere in the lab. I would also like to thank Tom Devine for his help in this work. I acknowledge the help of all the Computer Science department staff.

I acknowledge the support I have received from National Science Foundation grant 0916275 and funds from the American Recovery and Reinvestment Act of 2009.

ABSTRACT

The systematic reuse provided by software product lines provides opportunities to achieve increased quality and reliability as a product line matures. This has led to a widely accepted assumption that as a product line evolves, its reliability improves. However, evidence in terms of empirical investigation of the relationship among change, reuse and reliability in evolving software product lines is lacking.

To address the problem this work investigates: 1) whether reliability as measured by post-deployment failures improves as the products and components in a software product line change over time, and 2) whether the stabilizing effect of shared artifacts enables accurate prediction of failure-prone files in the product line.

The first part of this work performs defect assessment and investigates defect trends in Eclipse, an open-source software product line. It analyzes the evolution of the product line over time in terms of the total number of defects, the percentage of severe defects and the relationship between defects and changes.

The second part of this work explores prediction of failure-prone files in the Eclipse product line to determine whether prediction improves as the product line evolves over time. In addition, this part investigates the effect of defect and data collection periods on the prediction performance.

The main contributions of this work include findings that the majority of files with severe defects are reused files rather than new files, but that common components experience less change than variation components. The work also found that there is a consistent set of metrics which serve as prominent predictors across multiple products and reuse categories over time. Classification of post-release, failure-prone files using

change data for the Eclipse product line gives better recall and false positive rates as compared to classification using static code metrics. The work also found that on-going change in product lines hinders the ability to predict failure-prone files, and that predicting post-release defects using pre-release change data for the Eclipse case study is difficult. For example, using more data from the past to predict future failure-prone files does not necessarily give better results than using data only from the recent past. The empirical investigation of product line change and defect data leads to an improved understanding of the interplay among change, reuse and reliability as a product line evolves.

CHAPTER 1. INTRODUCTION

The systematic reuse provided by software product lines (SPLs) provides opportunities to achieve increased quality and reliability as it matures. We follow Weiss and Lai in defining a product line as “a family of products designed to take advantage of their common aspects and predicted variabilities [Weiss and Lai, 1999].” The planned reuse of artifacts allows more rapid development of new products and lower-cost maintenance of existing products [SEI, 1984], [Gomaa, 2004], [Pohl et al., 2005], [Weiss and Lai, 1999], [Clements and Northrop, 2001]. Since the product line artifacts such as specifications, design, code and test cases are reused and maintained via a centralized repository, there is reason to anticipate that the quality and reliability of both the existing products and the new products may improve over time. This has led to a widely accepted assumption that as a product line evolves, its reliability improves.

1.1 Problem Statement

The goal of this research is to investigate: 1) whether reliability as measured by post-deployment failures improves as the products and components in a software product line change over time, and 2) whether the stabilizing effect of shared artifacts as the product line evolves enables accurate prediction of failure-prone files in the product line.

We define *reliability* as continuity of correct service [Avizienis et al., 2001]. A *failure* is a departure of the system or system component behavior from its required behavior. A failure occurs when a system or component is unable to perform its required functions

within specified performance requirements [IEEE, 1999]. Post-deployment failures are indicators of the reliability of the system or components. In this work we study how reliability, measured by the number of post-deployment failures, changes as Eclipse, a large, open-source software product line, evolves over time. In accordance with the literature, we use the broader term *defect* to describe occurrences in which a software system does not behave as desired or specified [Pohl et al., 2005]. A defect is also often referred to as a fault or bug [Clark and Zubrow, 2001].

Ongoing change is typical in product lines and proceeds along two main dimensions. The first dimension is evolution of the product line in which, as the product line matures, more products are built. Some of these additional products may include new features (i.e., units of functionality [Batory et al., 2006]). The changes also may propagate to other, previously built products [Stephenson, 2002]. If the changes are incorporated into the product line, the product line asset repository is updated so that future products can reuse them.

The second dimension of product line evolution involves changes to an individual product from one of its releases to another. This is similar to the evolution and maintenance of a single system, except that it may happen to each system in the product line. The two dimensions of product line evolution are shown in Figure 1.1.

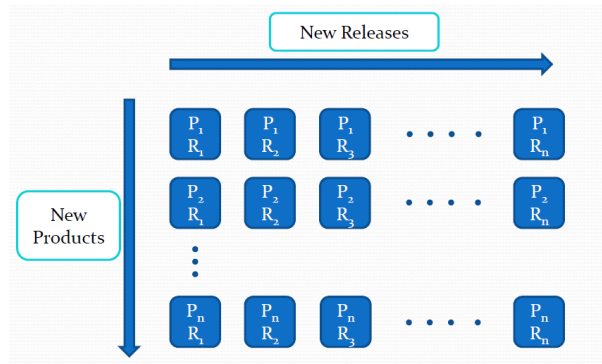


Figure 1.1: **Product Line Evolution across two dimensions**

As each product in the product line evolves across releases, failure and change data for each release may become available and can be used to estimate and predict reliability for individual products. We wish to discover whether the knowledge acquired from older products in a product line helps in estimating the quality and reliability of newer products.

The work reported here has two parts. The first part performs defect assessment and investigates defect trends in Eclipse, an open-source software product line. This part characterizes the evolution of Eclipse in terms of the amount of change it experiences over a period of 7 years and the corresponding bug trends for three different levels of reuse: *common* components reused in all products; *high-reuse variation* components, used in many but not all products; and *low-reuse variation* components, used in only one or two products.

Results from the first part of this work show that even common components experience change on an on-going basis. However, common components have less change than the variation components over time. As the product line evolves, fewer serious failures occur in common components than in variation components.

The second part of this work is motivated by these results. It explores prediction of failure-prone files in the Eclipse product line at the product level and component level to determine whether prediction improves as the Eclipse product line evolves over time. In addition, this part investigates the effect of defect and data collection periods on the prediction performance.

Results from the second part of this study show improvement in prediction performance for some but not all defect and data collection periods. It also identifies consistent predictors for individual products and components as well as for the entire product line.

1.2 Research Questions

This work investigates the following research questions.

1. Defect Assessment

- (a) *Failure trends.* Do serious failures decrease over time as the common/high-reuse variation/low-reuse variation components evolve over releases?
- (b) *Change trends.* Does the amount of change to the source code decrease across releases for the common/high-reuse variation/low-reuse variation components?
- (c) *Failure/Change relationships.* Does the number of serious failures normalized for source-code changes decrease over time for the common/high-reuse variation/low-reuse variation components?

2. Defect Prediction

- (a) *Prediction trends.*
 - i. Does our ability to predict the failure-prone files improve over time across products as the product line matures?
 - ii. Does the ability to predict failure-prone files differ across components belonging to different categories of reuse?
 - iii. How do datasets with different data collection and prediction periods affect prediction performance?
 - iv. Do datasets with incrementally increasing data collection periods yield better results?
- (b) *Product/Component evolution.* Are there any change metrics that serve as good predictors for which files are failure-prone as a product or component matures over releases?

- (c) *Product line evolution.* Do any of these change metrics also serve as good predictors across all the products and components in a product line over time?

1.3 Approach

We study Eclipse from a product line perspective, following the examples in the previous literature of [Chastek et al., 2007] and [van der Linden, 2009] in treating Eclipse as a product line. Since Eclipse is an open source project, the evolution of its products is documented in public failure reports, change reports and source code available across its component releases. We collect information from the defect and change repository to study how reliability changes as Eclipse evolves over time.

The work uses data extracted from the Eclipse Bugzilla database and the CVS change repositories for the individual Eclipse projects. Data is collected for five of the six defect severity categories, namely *blocker*, *critical*, *major*, *normal* and *minor*, for Eclipse products from 2002-2010, with the *trivial* category excluded. For the defect assessment analysis, the focus is on the severe post-deployment defects, i.e., the top three severity categories. The investigation of the relationship between defects and change uses information about the number of changes made to existing files and the number of new files created in each release for components belonging to all three categories of reuse (commonalities, high-reuse variations and low-reuse variations). The defect assessment considers the total number of severe failures, the percentage of the total failures that are severe and the defect trends for the different reuse categories.

The defect prediction work maps the bug data to the change data from the CVS repository using standard data collection techniques [Zimmermann et al., 2007]. Performance of defect classification by several well-known classifiers is compared using the collected data in order to identify whether any particular learner performs better than others. The evaluation of results uses defect classification performance metrics accu-

racy, recall, false positive rates, precision, and area under ROC curve (AUC) values, as described in Chapter 3, to compare our results with results obtained in other studies.

1.4 Results

The findings resulting from the investigation are briefly described here and in more detail in Chapters 4 and 5.

- **Defect Assessment**

1. *Failure trends.* As the product line evolves, *fewer serious failures occur in components implementing commonality.* The occurrence of failures in variation components shows no uniform pattern of decrease as the product line evolves. Although the number of failures in some variation components decreases as the product line matures, the percentage of severe failures in those components holds steady or even increases.
2. *Change trends.* *Common components exhibit less change over time* as compared to variation components, but also observe on-going change. Components implementing variations, even when reused in five or more products, *continue to evolve fairly rapidly.* In common components, the percentage of new files shows a decreasing trend as the product line evolves. In variation components that are lightly reused, the percentage of new files generally shows a decreasing trend, comparable in values to one of the common components. Heavily reused variation components have a very low percentage of new files, much lower than either common components or lightly reused variation components.
3. *Failure/Change relationships.* The occurrence of failures in variation components shows no uniform pattern of decrease as the product line evolves, even when normalized for the occurrence of change.

- **Defect Prediction**

1. *Prediction trends.* As the product line matures, prediction of post-release failure-prone files using pre-release change data for four products in the Eclipse product line shows statistically significant improvement in accuracy across releases, but not in recall. Similarly, components in the three categories of reuse show significant improvement in accuracy and false-positive rate but not in recall. Further, there is no statistically significant difference in performance improvement across releases among the three categories of reuse.
2. *Product/Component evolution.* As each product evolves, there is a set of change metrics that are consistently prominent predictors of failure-prone files across its releases. Looking at the evolution of components in the different categories of reuse in the product line (i.e., commonalities, high-reuse variations and low-reuse variations), we find that there is consistency among the prominent predictors for some categories, but not among all of them.
3. *Product line evolution.* There is some consistency among the prominent predictors for early vs. late releases for all the considered products in the product line.

1.5 Contributions

The contributions, discussed in more detail in Chapter 7, indicate the following interplays among change, reuse and reliability in the evolving software product line:

- The majority of files with severe defects are reused files rather than new files.
- On-going change in product lines hinders the ability to predict failure-prone files.
- Classification of post-release, failure-prone files using change data for the Eclipse product line gives better recall and false positive rates as compared to classification

using static code metrics.

- Predicting post-release defects using pre-release change data for the Eclipse case study is difficult.
- Using more data from the past to predict future failure-prone files does not necessarily give better results than using data only from the recent past.
- Common components experience less change than variation components.
- There is a consistent set of metrics which serve as prominent predictors across multiple products and reuse categories over time.

1.6 Dissertation Outline

The rest of the dissertation is organized as follows. Chapter 2 discusses related work and the Eclipse product line case study. Chapter 3 describes the approach. Chapter 4 presents results from the defect assessment for the Eclipse product line. Results from the defect prediction for the Eclipse product line are detailed in Chapter 5. Chapter 6 presents the threats to validity of this study. Chapter 7 summarizes the contributions and suggests future work.

CHAPTER 2. BACKGROUND AND RELATED WORK

This chapter defines the terminology used in the rest of this work and presents the related work.¹

2.1 Software Product Lines

We first introduce the terminology used to describe product line development. Weiss and Lai [Weiss and Lai, 1999] define a product line as “a family of products designed to take advantage of their common aspects and predicted variabilities [Weiss and Lai, 1999].” The Software Engineering Institute [SEI, 1984] defines a software product line as “a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.”

Core Assets: Core assets are the reusable artifacts and resources that form the basis for the software product line [SEI, 1984].

Commonalities: In a product line, some requirements are shared by all the products in the product line. These requirements are called commonalities [Weiss and Lai, 1999].

Variabilities: Variabilities describe how the products differ from each other. Each product has a particular combination of variability requirements which makes it unique in the product line [Weiss and Lai, 1999].

Commonality Variability Analysis: Commonality Variability Analysis (CVA) is a document which describes the commonalities, variabilities and dependencies in the prod-

¹Parts of this chapter are adapted from [Krishnan et al., 2011a] and [Krishnan et al., 2012a].

uct line. Dependencies are the constraints amongst the variabilities [Weiss and Lai, 1999].

Domain Engineering: This is the first step of product line engineering where the core assets of the product line are defined and implemented. Domain Engineering is defined as “A process for creating the production facilities for a family” [Weiss and Lai, 1999] or as “The process of software product line engineering in which the commonality and the variability of the product line are defined and realized” [Pohl et al., 2005].

Application Engineering: This is the second stage of product line engineering where the products in the product line are implemented by reusing the assets developed in the domain engineering phase. Application engineering is defined as “A process for rapidly creating members of a family (applications) using the production facilities for the family” [Weiss and Lai, 1999] or as “the process of software product line engineering in which the applications of the product line are built by reusing domain artifacts and exploiting the product line variability” [Pohl et al., 2005].

2.2 Eclipse Product Line

As the common and variation code files are reused across products, they go through iterative cycles of testing, operation and maintenance that over time identify and remove many of the bugs that can lead to failures. There is thus some reason to anticipate that the quality and reliability of both the existing products and the new products may improve over time.

The lack of available product line data, however, makes it hard to investigate such and similar claims. The availability of Eclipse data is a noteworthy exception. The Eclipse project, described on its website as an ecosystem, documents and makes available bug reports, change reports, and source code that span the evolution of the Eclipse products.

Chastek, McGregor and Northrop [Chastek et al., 2007] were the first that we know of to consider Eclipse as a product line. Eclipse provides a set of different products

tailored to the needs of different user communities. Each product has a set of common features, yet each product differs from other products based on some variation features. The features are developed in a systematic manner with planned reuse for the future. The features are implemented in Eclipse as plug-ins and integrated to form products. The products in the Eclipse product line are thus the multiple package distributions provided by Eclipse for different user communities.

2.2.1 Products

Each year Eclipse provides more products based on the needs of its user communities. For Java developers, the Eclipse Java package is available; for C/C++ developers, Eclipse provides the C/C++ distribution package, etc. In 2007, five package distributions were available: Eclipse Classic, Eclipse Java, Eclipse JavaEE, Eclipse C/C++, and Eclipse RCP. In 2008, two more products became available: Eclipse Modeling and Eclipse Reporting. Year 2009 saw the introduction of Eclipse PHP and Eclipse Pulsar. In 2010, Eclipse had twelve products, including three new ones: Eclipse C/C++ Linux, Eclipse SOA and Eclipse Javascript. The columns in Fig. 2.1 list the 2010 products. New products reuse and sometimes modify the common components and existing variation components and implement any required new variations in new component files.

In this study we observe four products (Eclipse-Classic, Eclipse-C/C++, Eclipse-Java, and Eclipse-JavaEE). Each product has a release during the years 2007-2010, with Eclipse-Classic also having releases for years 2002-2004. The yearly releases of Eclipse products are given release names in addition to the release numbers: *Europa* for year 2007, *Ganymede* for 2008, *Galileo* for 2009 and *Helios* for 2010. The release numbers corresponding to each release are 3.3 for Europa, 3.4 for Ganymede, 3.5 for Galileo, and 3.6 for Helios. In the rest of the paper, to refer to a particular release of a product, we mention the release name along with the release number, i.e., Classic-3.3 (Europa), Java-3.4 (Ganymede), etc. For the older releases from 2002-2004 we refer to

them using their release numbers, namely 2.0, 2.1 and 3.0, respectively.













	 Java	 Java EE	 C/C++	 C/C++ Linux	 RCP/Plugin	 Modeling	 Reporting	 PHP	 Pulsar	 SOA	 Javascript	 Classic
RCP/Platform	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CVS	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓
EMF	✓	✓				✓	✓	✓				
GEF	✓	✓			✓	✓	✓					
JDT	✓	✓			✓	✓	✓		✓			✓
Mylyn	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
Web Tools		✓					✓	✓		✓	✓	
Linux Tools				✓								
Java EE Tools		✓					✓					
XML Tools	✓	✓			✓		✓	✓	✓			
RSE		✓					✓					
EclipseLink		✓					✓					
PDE		✓			✓	✓	✓					✓
Datatools		✓					✓					
CDT			✓	✓								
BIRT							✓					
ECF					✓							
GMF						✓						
PDT								✓				
MDT						✓						
MTJ									✓			
Swordfish										✓		

Figure 2.1: **Eclipse Product Line for the year 2010** [<http://www.eclipse.org/downloads/compare.php>]

2.2.2 Components

The products are composed of components which are implemented as plugins. For the 2010 release, the components in the Eclipse product line are shown in the first column in Figure 2.1. The individual cells indicate which components are used/reused in each product.

Based on the level of reuse we observe three categories of components that implement: *commonalities*, *high-reuse variations* and *low-reuse variations*. Table 2.1 lists the components studied in this paper, grouped by level of reuse.

The first category contains the common components reused in all products. The large component RCP/Platform is the only common component reused across all products. Henceforth in the paper, we abbreviate the RCP/Platform component to Platform.

The second category is the set of variation components with high reuse, which are reused in more than two products but not in all products. The number of products in which these components are reused increases with each subsequent release from 2007 to 2010. The components in this category are EMF, GEF, JDT, Mylyn, Webtools, XMLtools, and PDE.

The third category is the set of variation components with low reuse. This category includes components that are reused only in two products, and the number of products in which they are reused does not increase with each release. The components in this category are CDT, Datatools and Java EE Tools (called JEEtools here).

Table 2.1: **List of Components**

Category	Component
<i>Common</i>	Platform
<i>High-reuse variation</i>	EMF GEF JDT Mylyn Webtools XMLtools PDE
<i>Low-reuse variation</i>	CDT Datatools JEEtools

2.3 Software Defect Assessment

There have been many studies of failure trends or profiles for commercial as well as open source systems. Work to date has been done to identify causes of failures, distribution of different types of failures, classification of the consequences of failures, comparison of the failure density of open-source systems to commercial systems, and defect/failure prediction for individual systems. However, studies investigating the effects of software product line engineering and their benefits by mining the failure databases

are rare.

The work most closely related to ours is that of Mohagheghi and Conradi [Mohagheghi and Conradi, 2008], [Mohagheghi et al., 2004], who reported on a system developed using a product family approach. They compared the fault density and the stability (amount of change) of the reused and non-reused components. They observed that reused components have lower fault density and less modified code as compared to non-reused components.

Recently we have also studied pre-release software faults in an industrial software product line [Devine et al., 2012]. We examined the pre-release software faults and code changes made to four products of the PolyFlow product line, a family of software testing tools developed by Avaya [Weiss et al., 2008]. Results showed that in that software product line setting, faults are more highly correlated to change metrics than to static code metrics. Also, variation components unique to individual products had the highest fault density and were most prone to change. Results also showed that development and testing of previous products benefited the new products in the software product line. This work is not discussed further in this dissertation as it will appear in Devine's master's thesis [Devine et al., 2012].

For open-source systems, Mockus, Fielding and Herbsleb [Mockus et al., 2000] investigated the effectiveness of open-source software development methods on Apache in terms of defect density, developer participation and other factors. They showed that for some measures of defects and changes, open-source systems appear to perform better while for other measures, the commercial systems perform better. In our study we use one of the measures they recommend, i.e., *KChanges*. Kim, Cai and Kim recently found that the number of bug fixes in three large open-source systems, one of them Eclipse JDT, increases after refactorings [Kim et al., 2011]. Eaddy et al. found a moderate to strong correlation between scattering (where the implementation of a cross-cutting concern is scattered across files) and defects for three case studies, one of which was an

Eclipse component [Eaddy et al., 2008].

Fenton and Ohlsson [Fenton and Ohlsson, 2000] analyzed the faults and failures in a commercial system and tested several hypotheses related to failure profiles. Their results related the distribution of faults to failures and the predictive accuracy of some widely used metrics. They found that pre-release faults are an order of magnitude greater than the operational failures in the first twelve months. Lutz and Mikulski [Lutz and Mikulski, 2004] analyzed serious failures/anomalies in safety-critical spacecraft during operations. Hamill and Goševa-Popstojanova conducted a study of two large systems to identify the distribution of different types of software faults and whether they are localized or distributed across the system [Hamill and Goševa-Popstojanova, 2009]. They analyzed different categories of faults and their contribution to the total number of faults in the system. Børretzen and Conradi [Borretzen and Conradi, 2006] performed a study of four business-critical systems to investigate their fault profiles. They classified the faults into multiple categories and analyzed the distribution of different types of faults.

Paulson, Succi and Eberlein [Paulson et al., 2004] investigated the growth pattern of open-source systems and compared them with that for commercial systems. They found no significant difference between the two in terms of software growth, simplicity and modularity of code. They found, however, that in terms of defect fixes, open-source systems have more frequent fixes to defects.

A comparative study of software reliability modeling for open source software was performed recently by Rahmani, Azadmanesh and Najjar [Rahmani et al., 2010]. They analyzed five open source software systems, collected the failure reports for them and compared the prediction capability of three reliability models on this data. One of their results was that the failure patterns for open-source software follow a Weibull distribution.

2.4 Software Defect Prediction

There has been extensive research in defect/failure prediction based on a wide variety of metrics data. Chidamber-Kemerer (CK) metrics have been widely used to predict the reliability of system or the fault-proneness of modules [Basili et al., 1996] [Zhou and Leung, 2006], [Tang et al., 1999], [Denaro et al., 2003]. Menzies et al. [Menzies et al., 2010] recently did a comprehensive study of the different approaches towards defect prediction from static code features and suggest that not only are the choice of metrics used for prediction important but also the machine learning techniques used for prediction. They explain the advantages of static code metrics over other process metrics.

Nagappan, Ball and Zeller used static code metrics to predict failures in five Microsoft software systems [Nagappan et al., 2006b]. They find that although code complexity metrics act as good predictors of failure-prone files, there is no single set of metrics applicable to all the projects. They also found that the predictors built from one Microsoft project using a particular set of metrics could be generally applied to other similar projects, but not to dissimilar projects. Since products in a product line are similar (i.e., share commonalities), this helps motivate our efforts to understand under what circumstances predictors from one product in a product line are relevant to other products in the product line.

[Nagappan and Ball, 2005, Menzies et al., 2007, Zimmermann et al., 2008] are some of the other works that have used static code metrics for classification and prediction.

Work described by Moser et al. in [Moser et al., 2008b, Moser et al., 2008a], and Nagappan et al. in [Nagappan et al., 2006a, Nagappan et al., 2010] used process metrics such as change data from the version repositories to predict the failure-proneness of modules. In our work, we have investigated the applicability of existing prediction techniques using process metrics to predict failure-proneness of files for product lines.

Our work builds on previous work by Zimmermann, Premraj and Zeller [Zimmermann et al., 2007] and by Moser, Pedrycz and Succi [Moser et al., 2008b]. The authors in [Zimmermann et al., 2007] studied defects from the bug database of three early releases of an Eclipse product at both the file and package level. They built logistic regression models to predict post-release defects. At the file level, the models had mixed results, with low recall values less than 0.4 and precision values mostly above 0.6. The authors in [Moser et al., 2008b] found that change metrics performed better than code metrics on a selected subset of the same Eclipse dataset, and that the performance of the J48 decision tree learner surpassed the performance of logistic regression and Naïve Bayes learners.

Besides the work of [Zimmermann et al., 2007] and [Moser et al., 2008b], several different approaches for defect prediction also have used Eclipse as the case study, giving additional insights into the role of various product and process metrics in defect prediction for Eclipse. D’Ambros, Lanza and Robbes analyzed three large Java software systems, including Eclipse JDT Core 3.3, using regression modeling, and found correlations between change coupling (files that change together) and defects [D’Ambros et al., 2009]. They found that Eclipse classes have, on average, many more changes and more shared transactions than classes in the other two systems studied.

Schroter, Zimmerman and Zeller [Schröter et al., 2006] found that their predictive models (regression models and support vector machines) trained in one version could be used to predict failure-prone components in later versions (here, from version 2.0 to 2.1 of Eclipse) . Shihab et al. reported work to minimize the number of metrics in their multivariate logistic regression model [Shihab et al., 2010]. In a case study on the Eclipse dataset, Zimmermann et al. [Zimmermann et al., 2008] identified four code and change metrics. One change metric, total prior changes in the 6 months before the release, was in their short list.

Studies reported in [Zimmermann et al., 2007], [Nagappan et al., 2006c],

[Nagappan et al., 2010], [Zimmermann et al., 2008], [Guo et al., 2010] used bug reports and bug repositories such as Bugzilla for predicting defects and failures. Jiang, Menzies, Cukic and others [Jiang et al., 2008], [Menzies et al., 2010] used machine learning algorithms successfully to perform defect prediction. Ostrand, Weyuker and Bell were able with high accuracy to predict the number of faults in files in two large industrial systems [Ostrand et al., 2005]. Menzies et al. found that a lower number of training instances provided as much information as a higher number for predicting faulty code modules [Menzies et al., 2008]. Zhang predicted the number of future component-level defects reasonably well using a polynomial regression-based model built from historical defect data [Zhang, 2008].

The work reported here is similar to the work by Mohagheghi et al. in [Mohagheghi and Conradi, 2008] in that both consider the effect of reuse on the quality of product lines. However, we focus on post-deployment failures, rather than fault densities, since post-deployment failures are experienced by end users and affects reliability more directly. Further, we consider the effects of code change specifically on the severe failures rather than failures in general. Eclipse also involves components reused across more products than in [Mohagheghi and Conradi, 2008].

To our knowledge this is the first work to perform defect prediction in a product line setting, investigating the ability to predict failure-prone files in the different product line reuse categories as they evolve over time. This work also contributes to the software engineering community by replicating earlier studies on prediction of failure-prone files and comparing results with updated datasets. While previous research focused on identifying better defect prediction approaches, our goal has been to investigate trends in the ability to predict post-release failure-prone files as a product line grows. We investigate the influence of the different reuse categories on several factors related to prediction of failure-prone files such as classifier selection, dataset selection and identifying prominent predictors.

CHAPTER 3. APPROACH

This chapter¹ describes the approach used for the defect assessment and the defect prediction parts of the study. For both parts we collected data from the defect as well as the version control repositories of Eclipse and used existing techniques to form mappings among entries from both repositories. The chapter explains the approach used for each part of the study separately, highlighting the data collection and integration methods used in each part.

3.1 Defect Assessment

The data studied is from four recent releases of the Eclipse Product Line, 2007 to 2010. The individual components that form the products were available before 2007, but the integration of components into products began from 2007, leading us to select these four releases for investigation.

The focus is on the effect of evolution for each component across these four releases on the component's post-release failures. Post-release failures are those that occur after the software is operational. For a user-community, the number of post-release failures encountered strongly affects their opinion of the quality of the software. The analysis also considers change in the source code of these components across the four releases.

¹Parts of this chapter are adapted from [\[Krishnan et al., 2011a\]](#) and [\[Krishnan et al., 2012a\]](#).

3.1.1 Failure Trends in an Evolving SPL

3.1.1.1 Data Sources and Severity Categories

The failure reports come from the public Eclipse Bugzilla database [Eclipse, a]. Users can query the database through a web interface and retrieve the results in graphical or textual format. We collect data for five of the six [Eclipse, b] severity categories: *blocker*, *critical*, *major*, *normal* and *minor*. We exclude the *trivial* failure category as these do not contribute significantly toward reliability.

We consider the failures in the top five severity categories to be the *total* failures for a given component. We aggregate the failures in the most serious three categories (*blocker*, *critical* and *major*) into a single category called *severe failures*. These failures all have serious consequences for the user, such as a major loss of functionality, crash, or blockage without a workaround.

A total of 9,266 failures are identified for the 11 components considered in this study across the four releases. Of these, approximately 1,542 are severe failures. The number of total failures and severe failures for each release is shown in Table 3.1.

Table 3.1: **Number of Total and Severe Failures**

Year	2007	2008	2009	2010	Total
All Failures	2928	2781	2089	1468	9266
Severe Failures	496	497	303	246	1542

3.1.1.2 Data Collection, Integration and Analysis

For each of the components listed in Table 2.1, we query the Eclipse Bugzilla database and retrieve the number of total failures and the number of severe failures. In the Bugzilla database, data can be retrieved for each component or for each of the individual sub-components of the component. We first map the plugins to the components for each distribution of every product. We collect the post-release failure data for only those sub-components that have corresponding plugins in the product distributions. This is to ensure that we consider failures for only those sub-components present in the distribution

and not for other sub-components that may not have been present during that particular release. The numbers for these sub-components are aggregated to calculate the number of failures for each component. Although this approach is more time consuming than directly finding the numbers for the components from Bugzilla, it allows us to get more accurate numbers for the components based on the product distributions.

We analyze the collected data in two ways. First we calculate the raw number of severe failures for each of the three categories of components (common, high-reuse variation and low-reuse variation). This is to investigate whether there are interesting patterns such as decreasing or increasing trends in the number of severe failures. The second way that we look at the data is by the percentage of severe failures. To detect, when a failure occurs, how often its effects are judged to be severe and how this factor changes over time, we determine the percentage of the total failures that are severe. If the percentage of severe failures increases or remains stable over time, it may indicate that the impact of failure on users is not necessarily decreasing, even if the number of severe failures is decreasing.

3.1.2 Change Trends in an Evolving SPL

3.1.2.1 Data Sources and Type of Changes

In order to measure the amount of change to the source-code, we mine the CVS release repository of Eclipse, which is our source for change data. There are two ways in Eclipse to readily observe software change. This study uses both these types of change to try to characterize the SPL evolution. The first kind of observable change is changes to existing files. We measure change to existing files in terms of modifications to existing code. Since the number of modifications is large, we calculate the number of Kchanges to the source code in each release of a component. Kchanges is the number of modifications to existing files for that component, divided by 1000. The second kind of observable change is change via new files. Since the number of new files is not as large, we calculate the percentage of files that are new for each release of each component.

3.1.2.2 Data Collection, Integration, Validation and Analysis

We use the tool *cvschangelogbuilder-2.5* [cvs, 2004] to query the CVS repository. The plugins for each component are available in the different Eclipse product distributions/packages. The plugins associated with these components are also annotated with the corresponding release numbers. Using this information of plugin name and associated release number, we query the CVS repository. The commit information also is annotated with whether files are changed or added. Using this information, we retrieve the number of changes and the number of additions made to the source-code per release using textual pattern matching. We match patterns like *changed* and *added* and find the number of times files are changed and number of new files added. We aggregate the number of changes for all plugins of a component to calculate the number of additions and modifications for each component. Since data is not available for all releases of some components, we exclude these components (PDE, Mylyn, EMF and Datatools) from the change analysis. However, we collect data for the majority of components in each of the three categories and analyze them. Of the 11 components examined in this study, data is available and retrieved for 7 components (Platform, JDT, GEF, Webtools, XMLtools, CDT, and JEEtools).

The Eclipse community maintains active forums for development of its products and components. In order to validate whether the collected data represents the real picture, we sought responses from the developers through the forums. The data entered in the bug repository is public, and both users and developers can initiate the bug. Developers actively use the bug database as well as the CVS/SVN/Git source control repositories. When a user opens a bug, s/he may not have the correct knowledge of which product/component the bug belongs to or what severity level the bug may have. However, the fields in the bug database are reassigned by the person who commits the fix for the bug, if it is incorrectly assigned by the user. We can therefore place confidence in the data in the bug database as it is maintained under the supervision of the developers

who actually commit the fixes for the bugs. Thus, the information collected about the component/sub-component that the bug is assigned to as well as the severity and priority level of the bug can be considered to be representative of the actual situation.

3.2 Defect Prediction

3.2.1 Data Collection, Integration and Validation

As mentioned in Chapter 2, our work in defect prediction builds on the previous work by [Zimmermann et al., 2007] and [Moser et al., 2008b]. In order to both replicate and extend the work conducted by Moser et al. [Moser et al., 2008b], we collected CVS log data and bug tracking database entries from May 2001 to May 2011 for the Eclipse Classic product. This data was partitioned into time periods corresponding to 6 months before and after the release of Eclipse 2.0, Eclipse 2.1, Eclipse 3.0, Eclipse 3.3 (Europa), Eclipse 3.4 (Ganymede), Eclipse 3.5 (Galileo), and Eclipse 3.6 (Helios). Figure 3.1 shows the time periods for each release.

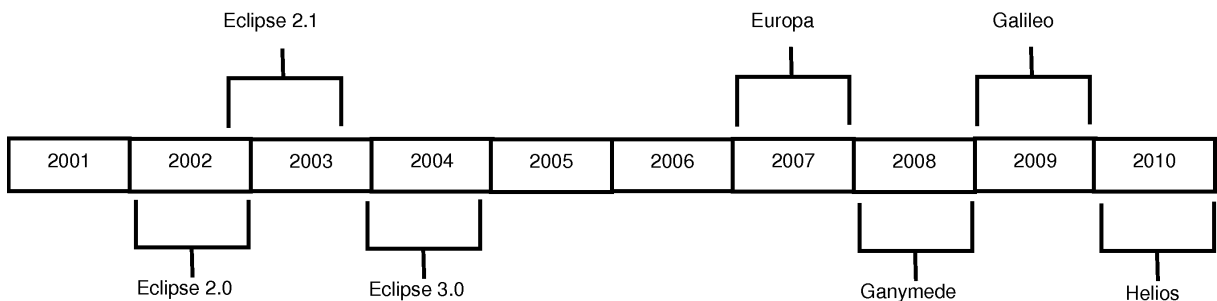


Figure 3.1: Data Timeline of Eclipse Classic

We extracted the same set of 17 change metrics as in [Moser et al., 2008b], including identifying bug-fixes, refactorings, and changeset size as listed in Table 3.2. A detailed description of these metrics is given in [Moser et al., 2008b]. For pre-Europa releases, i.e., releases 2.0, 2.1, and 3.0, as in [Zimmermann et al., 2007], we mined the CVS log

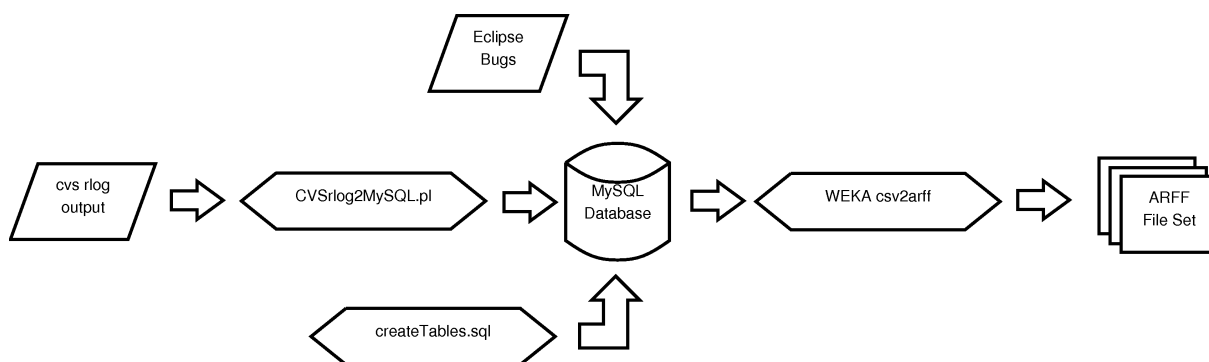


Figure 3.2: Data Collection Process

data by looking for four and five digit strings matching the bug IDs. For Europa and later releases, we matched six-digit strings to bug IDs. A manual review of data instances showed that no entries containing the word “bug” existed which were not caught by this pattern match. Extracting the metric *Refactorings* followed Moser’s approach, namely tagging all log entries with the word “refactor” in them. Refactoring the code involves restructuring parts of the code to improve code quality while preserving its internal structure. The *Age* metric was calculated by reviewing all CVS log data from 2001 onward and noting the timestamp of the first occurrence of each file name.

To determine changeset size, we used the CVSPS tool [Mansfield, 2001]. This tool identifies files which were committed together and presents them as a changeset. Slight modifications to the tool were required to ensure that the file names produced in the changesets included the path information to match the file names produced by our rlog processing script.

We wrote custom scripts to parse the CVS logs, converting the log entries into an SQL database. This data, along with changesets, bugs, and refactorings, were used to compute the metric values for each file. Finally, Weka-formatted files (ARFF) were produced. We also found and corrected an error in the script we had used to extract the change data from the database into ARFF files in [Krishnan et al., 2011b]. This error

had caused the data to be extracted beyond the stated end date (beyond six months pre-release) for 13 of the 17 metrics. Figure 3.2 provides an overview of this process.

Table 3.2: List of Change Metrics [Moser et al., 2008b]

Metric name	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 that 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 to its first appearance in the code repository)
WEIGHTED_AGE	$\frac{\sum_{i=1}^N Age(i) \times LOC_ADDED(i)}{\sum_{i=1}^N LOC_ADDED(i)}$ where $Age(i)$ is the number of weeks starting from the release date for revision i and $LOC_ADDED(i)$ is the number of lines of code added at revision i

To determine changeset size, we used the CVSPS tool [[Mansfield, 2001](#)]. This tool identifies files which were committed together and presents them as a changeset. Slight modifications to the tool were required to ensure that the file names produced in the changesets included the path information to match the file names produced by our rlog processing script.

We wrote custom scripts to parse the CVS logs, converting the log entries into an SQL database. This data, along with changesets, bugs, and refactorings, were used to compute the metric values for each file. Finally, Weka-formatted files (ARFF) were produced. We also found and corrected an error in the script we had used to extract the change data from the database into ARFF files in [[Krishnan et al., 2011b](#)]. This error had caused the data to be extracted beyond the stated end date (beyond six months pre-release) for 13 of the 17 metrics. [Figure 3.2](#) provides an overview of this process.

To ensure that the data resulting from the various input sources all contained matching filenames (the key by which the data were combined), and covered the same time periods, a few on-the-fly modifications were necessary. In cases where a file has been marked “dead”, it is often moved to the Attic in CVS. This results in an alteration of the file path, which we adjusted by removing all instances of the pattern “/Attic/” from all file paths.

An artifact of using the CVS rlog tool with date filtering is that files which contain no changes during the filter period will be listed as having zero revisions, with no date, author, or other revision-specific information. This is true even if the file was previously marked “dead” on a branch. Thus, rather than examining only the date range required for each specific release, we obtained the rlog for the entire file history and determined the files which were alive and the revisions which applied to each release.

To validate our approach, we compared our resulting file set for the pre-Europa releases (2.0, 2.1 and 3.0) with the file sets available from Zimmermann’s work [[Zimmermann et al., 2007](#)]. We found that there were a few differences in the two

datasets due to the independent data collection processes. While most of the files were common to both datasets, there was a small subset of files which were unique to each of them. For the three components, Platform, JD-T and PDE, in the 2.0 release, we included 6893 files as compared to their 6730 files. In the 2.1 release, we had 7942 files while they had 7888, and in the 3.0 release, we had 10822 files as compared to 10593 in theirs. Further inspection showed that there were some differences in the list of plugins included in both studies. We also observed that some files which were not present in the earlier dataset did have revisions during the development and production lifetime of the respective releases, and hence should have been included in the analysis. We thus included those in our dataset.

Moser et al. [Moser et al., 2008b] use a subset of the dataset used in [Zimmermann et al., 2007] (57% of Classic-2.0 files, 68% of Classic-2.1 files and 81% of Classic-3.0 files) and annotate it with change metrics. Since this dataset is not publicly available, we cannot compare our dataset with theirs. As discussed earlier, our dataset is comparable in size with the Zimmermann dataset in [Zimmermann et al., 2007] and hence larger than the Moser dataset in [Moser et al., 2008b].

3.2.2 Types of Datasets

Previous classification studies use several different types of datasets. Some previous defect prediction studies use datasets that divide the time period into pre-release and post-release [Zimmermann et al., 2007], [Moser et al., 2008b], [Zimmermann et al., 2009], [Nagappan et al., 2006a], [D'Ambros et al., 2012]. In these studies, metrics are collected for a specified period before the release of the software (typically six months), and these metrics are used to predict the post-release defects six months after release. Other studies use datasets that do not have such division of data into time periods. These include datasets from the NASA MDP repository [MDP, 2004] and the PROMISE repository [Menzies et al., 2012]. MDP and PROMISE datasets provide static metrics at

file (or class) level but do not distinguish between pre-release and post-release defects [Hall et al., 2012], [Gray et al., 2011].

Studies using the NASA MDP and PROMISE datasets have shown good prediction performance (e.g., [Menzies et al., 2007], [Menzies et al., 2011], [Bettenburg et al., 2012]), applying cross-validation to predict the defective files. However, the high recall rates in experiments carried out on these datasets may not be achievable for our goal of a product line project predicting future failure-prone files from past data.

Studies which have divided their data into pre-release and post-release periods observed mixed results in terms of prediction performance. For studies on open-source systems, Zimmermann et al. [Zimmermann et al., 2007] report that for three releases of the Eclipse system, classifying files as failure-prone or not gave very low recall rates (the best being only 37.9% for Eclipse 3.0) when static metrics were used. Moser et al. [Moser et al., 2008b] reported much better results for the same releases of Eclipse when change metrics were used with recall rates greater than 60%. However, this dataset is not publicly available and hence reproducibility is not possible. Recently, D’Ambros et al. performed a study to provide a benchmark for existing defect prediction strategies [D’Ambros et al., 2012]. They report high AUC values (greater than 0.85) for five open-source systems, when change metrics were used. Studies from Microsoft by Nagappan et al. [Nagappan et al., 2010] report very high recall and precision rates (both greater than 90%) when using change burst metrics for predicting defect-prone binaries. However, they also report that the same change burst metrics perform poorly for some open-source projects like Eclipse (recall rate of only 51%).

To check the consistency of results across datasets with different data collection and prediction periods, we experiment with three existing approaches to classifying our datasets, each involving a different time period for collecting change and defect data. For every release of the Eclipse products (i.e., 2.0, 2.1, 3.0, 3.3, 3.4, 3.5 and 3.6), we collected change and defect data for 6 months before and after release. Except for release 2.1,

which was released in March, 2003, the other releases were in June of their respective years. We partition this collected change and defect data in three different ways to form the three types of datasets. We then compare results among the three types of datasets as we investigate the research questions.

- *UseAll_PredictAll*: This dataset uses the same approach as the NASA MDP and PROMISE datasets [MDP, 2004, Menzies et al., 2012]. For this type of dataset, change data is collected for the entire twelve months (Jan-Dec) of each release. Pre-release and post-release defects are grouped into a single field. If a file has any defects associated with it, we tag the file as defective; otherwise, the file is tagged as non-defective. In this type of datasets we do not distinguish between pre-release and post-release defects. Therefore, the metric BUGFIXES is not included in the feature set, i.e., only the other 16 change metrics are included.
- *UseAll_PredictPost*: This dataset is a variant of the approach used in our earlier work [Krishnan et al., 2011b]. As with the previous dataset, the change data is collected for the twelve months (Jan-Dec) of each release. Pre-release defects are distinguished from post-release defects. The number of pre-release defects (defects in Jan-June) are counted and recorded in the BUGFIXES metric. If a file has any post-release defects (defects in Jul-Dec), it is tagged as defective; otherwise, the file is tagged as non-defective.
- *UsePre_PredictPost*: This dataset uses the same approach as that used by Zimmermann et al. [Zimmermann et al., 2007] and others [Moser et al., 2008b], [Zimmermann et al., 2009], [Nagappan et al., 2006a], [D'Ambros et al., 2012]. For this dataset, change data is collected for six months (Jan-Jun) pre-release, including the BUGFIXES metric. Again, pre-release defects are distinguished from post-release defects. If a file has any post-release defects (defects in Jul-Dec), it is tagged as defective; otherwise, the file is tagged as non-defective.

The main reason for wanting to distinguish pre and post-release defects is that, since post-release defects are encountered and reported by customers, they may have a higher impact on the quality of the software as perceived by the customer. Additionally, in terms of the practical utility of prediction, projects may seek to use metrics collected from the pre-release period to predict post-release defects. Using pre-release data to predict pre-release defects, or post-release data to predict post-release defects may have limited practical value.

3.2.3 Data Analysis

The base probabilities (proportion of defective files) for all releases of all four products for the three datasets are given in Table 3.3. The total number of files for each release of each product is given in the third column. For the *UseAll_PredictAll* datasets, the percentage of defective files is shown in the fourth column. For both the *UseAll_PredictPost* and *UsePre_PredictPost* datasets, the percentage of defective files is the same as shown in the last column. The percentage of defective files in the *UseAll_PredictAll* dataset, which includes both pre-release and post-release defects, is two to three times larger than in *UseAll_PredictPost* and *UsePre_PredictPost* datasets, for all products and releases.

In our previous work [Krishnan et al., 2011b], the prediction was done at the product level, for each product in the product line. In this work, we perform prediction and analysis at the component level as well. Data at the product level is an aggregation of data at the component level, i.e., the total number of files in a product is an aggregation of the files of all the components that belong to that particular product. For example, Eclipse-Classic is composed of three components, Platform, JDT and PDE. As such, the total files for any release of Eclipse-Classic is an aggregation of all the files of Platform, JDT and PDE for that release.

We perform an initial exploration using seventeen different learners including Bayesian methods, decision tree methods, support vector techniques, neural network techniques

Table 3.3: Base Probability for All Releases for Multiple Products of Eclipse

Product	Release	Total Files	<i>UseAll_PredictAll</i>	<i>UseAll_PredictPost</i> and <i>UsePre_PredictPost</i>
Classic	2.0	6893	54.6%	26.2%
	2.1	7942	45.9%	23.3%
	3.0	10822	47.6%	23.5%
	3.3	15661	32.1%	16.7%
	3.4	17066	32.1%	16.6%
	3.5	16663	24.0%	11.9%
	3.6	17035	18.6%	8.3%
C_C++	3.3	14303	36.7%	18.3%
	3.4	15689	37.6%	21.3%
	3.5	16489	32.6%	16.6%
	3.6	16992	30.4%	10.5%
Java	3.3	18972	40.4%	18.1%
	3.4	20492	32.4%	17.8%
	3.5	20836	25.8%	13.7%
	3.6	21178	21.2%	8.6%
JavaEE	3.3	35311	48.7%	24.2%
	3.4	39033	34.8%	16.5%
	3.5	39980	26.3%	11.5%
	3.6	41274	19.1%	6.6%

and nearest neighbor methods. Based on the results we choose the J48 decision tree learner for the subsequent work. The prediction results are obtained using 10-fold cross validation (CV). We divide the dataset into 10 folds and use 9 folds for training and 1 fold for testing. This is done for each fold and the results of the 10 folds are averaged. For some statistical tests, we repeat the 10-fold CV multiple times as indicated in the text.

Based on the confusion matrix shown in Table 3.4, we use the following metrics of learner performance, consistent with [Zimmermann et al., 2007] and [Moser et al., 2008b].

$$PC = \frac{(n_{11} + n_{22})}{(n_{11} + n_{12} + n_{21} + n_{22})} * 100\% \quad (3.1)$$

$$TPR = \frac{n_{22}}{(n_{21} + n_{22})} * 100\% \quad (3.2)$$

Table 3.4: **Confusion Matrix**

		Predicted Class	
		Not Failure-prone	Failure-prone
True Class	Not Failure-prone	$n_{11}(TN)$	$n_{12}(FP)$
	Failure-prone	$n_{21}(FN)$	$n_{22}(TP)$

$$FPR = \frac{n_{12}}{(n_{11} + n_{12})} * 100\% \quad (3.3)$$

$$Precision = \frac{n_{22}}{(n_{12} + n_{22})} * 100\% \quad (3.4)$$

The metric *PC*, also known as *Accuracy*, relates the number of correct classifications to the total number of files. The metric *TPR*, also known as *Recall*, relates the number of files predicted and observed to be failure-prone to the total number of failure-prone files. It is also known as the probability of detection. The metric *Precision* gives the number of files that are actually failure-prone within the files that are predicted as failure-prone. The measure *False Positive Rate (FPR)* relates the files incorrectly classified as failure-prone to the total number of non-failure-prone files. We use these metrics to compare our results with those by Moser, et al. [Moser et al., 2008b] and Zimmermann, et al. [Zimmermann et al., 2007]. In addition to these metrics, we also use the *Area Under the ROC Curve (AUC)* as a performance metric. The Receiver Operating Characteristic curve is a curve that gives the trade-off between the recall and the false positive rates. The curve can be used to find the optimal operating condition based on the desired recall and false positive rates. Such values may differ from one system to another, based on the domain and functionality required of the system. Area under the ROC curve (AUC) is a measure to identify the performance. AUC value lies between 0 and 1. The

larger the AUC value, the better. An ROC curve which lies towards the upper left corner of the graph (high recall and low false positive rate) is the desirable position [Heagerty and Zheng, 2004], [Wikipedia, 2003].

In addition to the prediction results obtained from 10-fold cross-validation, we identify the metrics which are most prominent. Prominent predictors are those which provide the highest information gain for classification of post-release defects. We find the Gain Ratio (GR) for each metric. GR has been found to be an effective method for feature selection [Shivaji et al., 2009]. Information Gain (IG) favors features with a larger number of values, although they actually have less information [Wang et al., 2011]. GR improves upon IG by normalizing it with the actual intrinsic value of the feature. Gain Ratio is calculated as

$$GR(C, a) = (H(C) - H(C|a))/H(a) \quad (3.5)$$

where H is the entropy function, C is the dependent variable (CLASS) and a is the feature being evaluated. We modified the J48 code in Weka to output the gain ratio weights assigned to the nodes of the tree based on the number of correctly classified files from the total number of files.

Based on the GR of the features we perform a step-wise greedy feature selection approach. We first select the feature with the highest GR to perform classification. We then add the feature with second-highest GR to the dataset and repeat the classification. If there is significant improvement in classification performance, this feature is added to the prominent predictor list. Features are added in decreasing GR order until no additional feature significantly improves classification performance. We repeat the procedure for each release of each product (or component). Note that the significance levels reported by this procedure are not literal (since predictors are pre-screened by GR and the t-test is not valid because the 10-fold CV values are not independent). As a result, this feature selection procedure neither guarantees the best set of predictors nor that each predictor actually significantly improves prediction, but it is a reasonable procedure to identify

likely important predictors in a standard way.

Finally, we investigate an incremental prediction approach that uses increasing amount of change data (instead of the usual 6 months) to predict the failure-prone files in the remaining post-release months. We increment the change data period from 6 months to 11 months, in steps of 1 month, while simultaneously reducing the post-release failure-prone file data from 6 months to 1 month.

Note that in order to control the family-wise error rate (FWER) at the 0.05 level due to multiple statistical tests performed in this work, we use a cut-off significance value of $0.05/36 = 0.001$.

CHAPTER 4. DEFECT ASSESSMENT RESULTS

In this chapter, we present the results from the defect assessment study. Recall that in Chapter 1 we briefly stated the research questions for our defect assessment study. The research questions are stated in detail below. ¹

4.1 Research Questions

1. *Failure trends*

- (a) Failure trends for common components
 - (i) Do the number of severe failures (*blocker, critical and major*) decrease with time as the common component is being reused across multiple releases?
 - (ii) Does the percentage of severe failures decrease with time as the common component is being reused across multiple releases?
- (b) Failure trends for high-reuse variation components
 - (i) Do the number of severe failures decrease with time as the high-reuse variation components are being reused across multiple releases?
 - (ii) Does the percentage of severe failures decrease with time as the high-reuse variation components are being reused across multiple releases?

¹The work presented in this chapter is adapted from [Krishnan et al., 2011a]. Empirical Evaluation of Reliability Improvement in an Evolving Software Product Line. *Proceedings of the 8th Working Conference on Mining Software Repositories*, Waikiki, Honolulu, HI, USA, May 2011, pp. 103-112.

- (c) Failure trends for low-reuse variation components
 - (i) Same as b(i) but for low-reuse variation components.
 - (ii) Same as b(ii) but for low-reuse variation components.

2. *Change trends*

- (a) Does the percentage of new files and/or modifications to the source code for the common components decrease across releases?
- (b) Does the percentage of new files and/or modifications to the source code for the variation components decrease across releases?

3. *Failures/Change relationship*

- (a) Is there a decrease in the number of failures with respect to changes (new file creation/code modifications) for the common components across releases?
- (b) Is there a decrease in the number of failures with respect to changes (new file creation/code modifications) for the variation components across releases?

4.2 Results

The following discussion describes the results obtained from the analysis of the data for each of the above research questions.

4.2.1 Failure Trends

4.2.1.1 Failure Trend for Common Components

The common components are those that have been reused in all products.

- (i) *Number of severe failures decreases over time, as expected.* Fig. 4.1 shows the decreasing number of severe failures for the five individual sub-components of Platform and for the aggregate Platform-combined which is the sum of the failures

for its fourteen sub-components. Note that the 2007 Release is labeled as 1, 2008 Release as 2 and so on.

- (ii) *Percentage of severe failures tends to stabilize and even shows a gradual increase over time, contrary to our expectations.* As shown in Fig. 4.2, the percentage of severe failures for Platform-combined tends to remain in the range of 14.5% to 17%, rather than continuing to drop.

In fact, for Platform-combined, the percentage of severe failures increases over the last three releases. SWT, UI, Resources and Platform-combined show an increase in the percentage of severe failures from release 3 to release 4. Of the remaining two sub-components, Runtime and Debug, Debug shows a decrease of approximately 1% only from release 3 to 4. Only Runtime subcomponent shows a significantly decreasing trend over the last three releases. This shows that from release 3 to 4 the percentage of severe failures does not exhibit a significant decrease for the common sub-components.

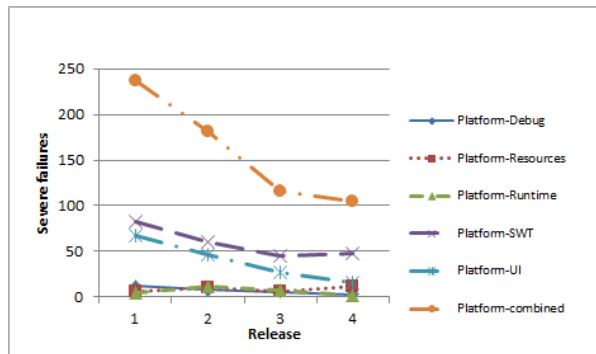


Figure 4.1: Number of Severe Failures in Common Components

4.2.1.2 Failure Trend for High-reuse Variation Components

These are the components which are reused increasingly in multiple products across the four releases.

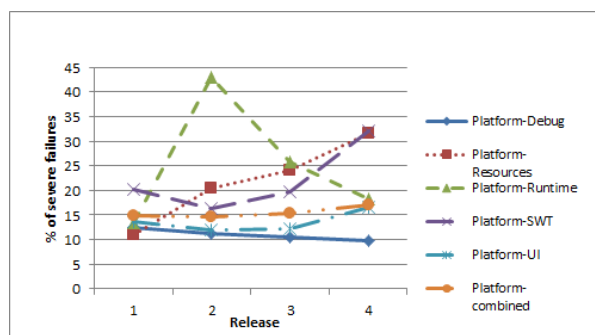


Figure 4.2: **Percentage of Severe Failures in Common Components**

- (i) *Number of severe failures does not monotonically decrease over time and shows mixed behavior, contrary to our expectations.* Fig. 4.3 shows that this monotonic decrease occurs for some components (JDT), but not for others. For example, PDE, Mylyn and GEF show an increase in the number of severe failures from release 1 to 2. EMF shows an increase in the fourth release. Other components show uneven behavior. For example, Webtools and XMLtools show an increase in severe failures from release 1 to 2, then a decrease from release 2 to 3 and again an increase from release 3 to 4.

The data suggest that for variation components with high reuse, the trend for number of severe failures over time is highly mixed and dependent on the component. While we would expect that highly reused variation components tend to behave like common components, only a few of them do.

- (ii) *Percentage of severe failures also shows a mixed trend contrary to our expectations.* Fig. 4.4 shows that the values for percentage of severe failures also show similar uneven trends. PDE and the last three releases of JDT show trends similar to the common Platform-combined component, with the percentage of severe failures tending to stabilize at 9.5% to 13%. However, the values for Webtools and XMLtools fluctuate in a large range of 4% to 27% with alternating increases and

decreases in the percentage values. Six of the seven components have a higher percentage of severe failures in release 4 than in release 3. Interestingly, although the number of severe failures for JDT steadily decreases from release 2 to 4, the percentage of severe failures steadily increases for these releases.

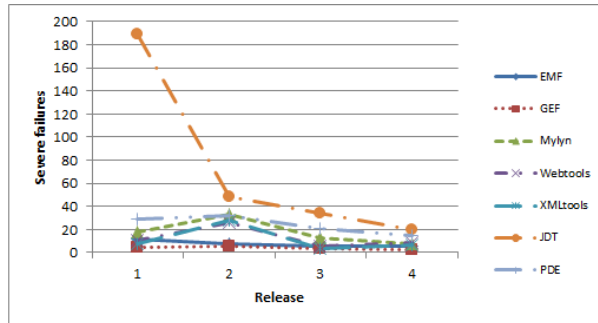


Figure 4.3: Number of Severe Failures in Variation Components with High Reuse

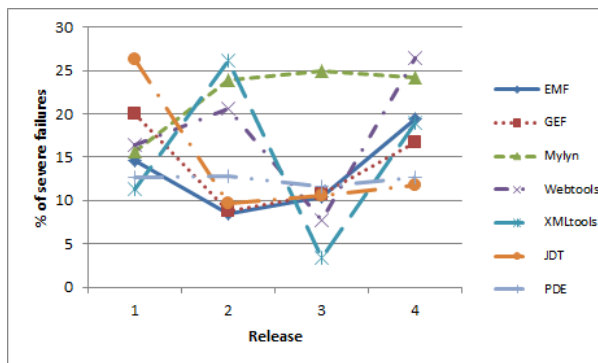


Figure 4.4: Percentage of Severe Failures in Variation Components with High Reuse

4.2.1.3 Failure Trend for Low-reuse Variation Components

Variation components with low reuse display tendencies similar to variation components with high reuse.

(i) *Number of severe failures show mixed trends and do not monotonically decrease.*

The low-reuse variation components have a higher number of severe failures than

most of the high-reuse variation components, which matches our expectations. However, we also observe mixed trends. Fig. 4.5 shows an overall decrease in the number of severe failures from release 1 to 4 for CDT. The number of severe failures for JEEtools increases from release 1 to 2, then decreases from release 2 to 3 and then remains stable from release 3 to 4. Datatools shows an alternate rise and drop in the number of severe failures. The number of severe failures do not monotonically decrease for all components; rather there is a mixed behavior.

- (ii) *Percentage of severe failures shows mixed trends and not a decreasing trend.* For low-reuse variation components, the percentage of severe failures fluctuates less than for high-reuse variation components. Only CDT shows a steady decrease in percentage values as seen in Fig. 4.6. JEEtools shows an increase from release 1 to 2, then a decrease from 2 to 3 and the remains stable from 3 to 4, whereas Datatools first decreases, then increases and remains stable. Thus, the percentage of severe failures shows mixed results similar to the trends for number of severe failures.

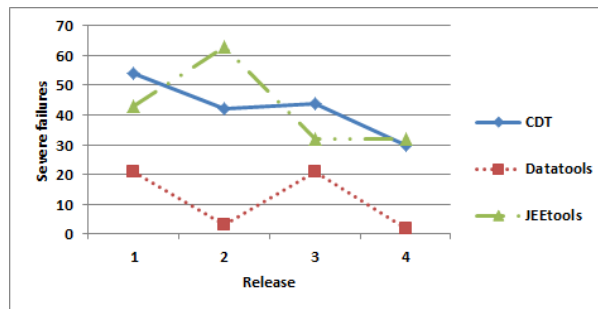


Figure 4.5: **Number of Severe Failures in Variation Components with Low Reuse**

4.2.2 Change Trends

Chapter 3, Section 3.1.2 described two kinds of observable changes as Eclipse evolved: creation of new files and modifications to existing files. We first discuss trends related to

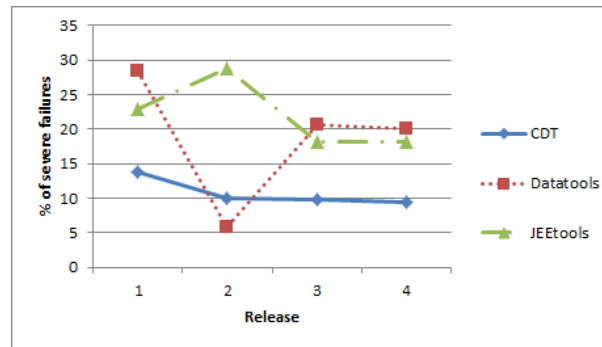


Figure 4.6: **Percentage of Severe Failures in Variation Components with Low Reuse**

new files and the occurrence of failures in the next subsection, and then trends related to modifications to existing files and the occurrence of failures in the following subsection.

4.2.2.1 SPL Evolution With Respect to New Files

We investigate the amount of change in the common components by calculating the percentage of new files for each component in each release. Table 4.1 gives the percentage of new files per release for each component we analyze. Cells with “No-info” indicate that the data for that particular time period were not available in the Eclipse repository. Cells with “-” indicate that there were no new files added in that time period. For the common components, the percentage of new files gradually decreases across the four releases. This is consistent with the SPL expectation that the common components, since they contain features shared by all products, will be relatively stable. We also see that in the initial release, the percentage of new files is very high.

For the high-reuse variation components, the percentages of new files are less than for the common components/ sub-components. The percentages of the files that are new for a given component tend to be stable across releases, rather than showing a decreasing trend (as the common components do).

For the low-reuse variation components, the percentages of new files are comparable to

the common components and sub-components. These percentages are also much higher than for the high-reuse variation components. They show an overall decreasing trend as per our expectations with the last release of JEEtools being an exception.

4.2.2.2 SPL Modification of Existing Code

Modification to existing code is observed by calculating the number of code changes normalized to the number of files over the sequential releases of the components. There is no overarching trend, except that a single component, SWT, has a significantly larger modification rate over time than any other component. The two smallest common components in this study, in terms of number of files (Resource and Runtime), have the lowest modification rate among the common components. However, even the common components do not show a decreasing modification rate across releases. To summarize, existing as well as new files show significant change, and even the common components were not reused intact, but were modified on an ongoing basis.

Table 4.1: Percentage of New Files for Commonalities and Variabilities

Category	Component	Percentage of new files			
		2007	2008	2009	2010
Common	Debug	14.87	4.32	4.26	3.16
	UI	9.10	6.57	4.80	No-info
	SWT	28.82	15.77	7.70	4.74
	Resources	42.76	1.32	-	5.15
	Runtime	-	-	-	-
High-reuse	JDT	2.28	6.43	1.91	1.03
	Webtools	1.44	13.30	1.26	1.23
	XMLtools	3.78	7.41	2.21	4.18
	GEF	0.53	1.90	0.68	2.18
Low-reuse	CDT	22.97	12.05	3.53	3.51
	JEEtools	29.92	9.44	2.13	7.31

Table 4.2: Failures/New-File for Commonalities and Variabilities

Category	Component	Failures/new-file			
		2007	2008	2009	2010
Common	Debug	0.09	0.21	0.15	0.10
	UI	0.27	0.25	0.19	No-info
	SWT	0.44	0.58	0.83	1.37
	Resources	0.05	2.50	-	0.86
	Runtime	-	-	-	-
High-reuse	JDT	1.95	0.17	0.40	0.43
	Webtools	0.16	0.03	0.07	0.11
	XMLtools	0.09	0.15	0.05	0.06
	GEF	1.33	0.55	0.75	0.15
Low-reuse	CDT	0.05	0.07	0.25	0.17
	JEEtools	0.03	0.11	0.24	0.06

Table 4.3: Failures/Kchanges for Commonalities and Variabilities

Category	Component	Failures/Kchanges			
		2007	2008	2009	2010
Common	Debug	6.92	7.47	14.81	4.71
	UI	18.02	14.79	12.72	No-info
	SWT	26.64	6.90	7.83	10.86
	Resources	44.44	50.51	34.88	35.82
	Runtime	200.00	857.14	235.29	250.00
High-reuse	JDT	59.94	5.95	3.84	8.51
	Webtools	4.47	2.31	3.95	3.24
	XMLtools	3.02	17.29	2.65	5.91
	GEF	90.91	39.74	51.72	1.62
Low-reuse	CDT	6.65	5.83	10.98	9.69
	JEEtools	6.54	17.51	14.00	6.56

4.2.3 Failure/Change Trends

One reason for increased failures might be large amounts of new/changed code that introduced faults. To analyze the failure/evolution relationship, for each component we extracted the number of new files added in each release and the number of times existing files were changed. Table 4.2 shows the number of severe failures per new file and Table 4.3 the number of severe failures per 1000 changes.

4.2.3.1 Failure/Evolution Relationship for New Files

Interestingly, the rise in the percentage of new files for three of the four high-reuse variation components (as seen in Table 4.1) is accompanied by an increase in the number of severe failures (as seen in Fig. 4.3). For the second release, Webtools, XMLtools and GEF show an increase in the percentage of new files and also a corresponding increase in the number of severe failures from the first release. Similarly in the fourth release, XMLtools shows an increase in the percentage of new files and also an increase in the number of severe failures. This may indicate a relationship between the failures and the number of new files.

Table 4.2 shows a non-uniform increase or decrease in the ratio of failures over new files. However, comparing the Failures/new file values of release 1 to release 4, there is one observation that distinguishes the common components from the variation components. With the exception of the Debug sub-component, for the other two *common* sub-components for which we have valid data (SWT and Resources), release 1 has a lower Failures/new-file value than release 4, with the difference being more than 0.8. This means that the addition of new files in later releases of the evolution led to more failures. For the *high-reuse variation* components, however, we observe that the values in release 1 are always higher than in release 4, which may mean that the addition of new files in later stages did not lead to as many failures as in the early stages. *Low-reuse variation* components also show trends similar to the *common* sub-components.

4.2.3.2 Failure/Modification Relationship for Existing Files

As a reminder, Kchanges is the number of modifications to existing files divided by 1000. Table 4.3 shows that most components do not have a steadily decreasing rate for Failures/Kchanges. Even for *common* components, the Failures/Kchanges decreases over releases for only one of the five sub-components (UI). For Debug, Failures/Kchanges

increases in the first three releases, and for SWT it increases from release 2 to 4. Resources and Runtime first show an increase, then a decrease and then tend to remain stable. The *high-reuse* and *low-reuse* variation components show similarly mixed trends in the Failures/Kchanges. With respect to changes, failures fluctuate a great deal with no distinguishing upward or downward trend.

4.3 Discussion of the results

The highlights of the empirical observations about post-deployment failures and stability of changes in the open-source, evolving product line Eclipse are summarized as follows:

1. Components/sub-components implementing commonality reused in every product exhibit fewer serious post-deployment failures across releases.
2. Variable components, both heavily and lightly reused, do not show a monotonically decreasing trend for post-deployment failures across releases. No obvious trend is observed even when failures are normalized for the number of changes made to existing files or for the number of new files.
3. Although the number of failures in some variation components decreases as the product line matures, the percentage of severe failures in those components holds steady or even increases.
4. The percentages of new files in common components show a decreasing trend as the product line evolves through releases. The values of these percentages are roughly comparable to lightly reused variation components, but higher than for heavily reused variation components.

As expected, common components experience fewer severe post-deployment failures and less change as the product line matures through releases. Conversely, contrary to

typical expectations, variable components, even if reused in multiple products, do not show a decreasing pattern either in post-deployment failures or in the changes made/new files added across subsequent releases. These findings clearly indicate that the improvement of post-deployment quality and the stability of source code do not depend solely on how often components are reused.

None of the Eclipse components considered in our study was reused intact (“as-is”). “As-is” reuse without change to existing components might have led to more straightforward conclusions about the benefits of reuse in software product lines. The extent of enhancements/new features added with each release is one of the factors that may help explain the mixed results for the variation components and that may determine the benefit of reuse for software product lines such as Eclipse that undergo rapid evolution. This finding of on-going change in reused elements merits further investigation that should take into account the amount of change measured at a finer granularity (e.g., blocks or lines of source code), and possibly include metrics such as the size and complexity of the components.

The mixed results also suggest that there may be other factors associated with these reuse components, such as the amount of pre-deployment testing and the extent of field usage, that are not accounted for in this analysis. Unfortunately, as pointed out in [Fenton and Ohlsson, 2000], this information is typically unavailable to allow more in-depth study in this direction.

CHAPTER 5. DEFECT PREDICTION RESULTS

In this chapter, we present results from the defect prediction study.¹

5.1 Research Questions

Chapter 3, Section 3.2.2, gave a description of the three types of datasets studied in this chapter: *UseAll_PredictAll*, *UseAll_PredictPost* and *UsePre_PredictPost*. This chapter explores the following research questions for each of the three types of datasets mentioned above:

RQ1. *Classifier Selection*

- (i) Is there a specific machine learner that is significantly better than other learners for classifying failure-prone files using change data?

RQ2. *Single Product Evolution*

- (i) How do our results related to learner performance compare with previously published results?
- (ii) Does learner performance improve as a single product evolves?
- (iii) Is the set of prominent predictors consistent across releases of a single product?

RQ3. *Product Line Evolution*

¹The work presented in this chapter is adapted from [Krishnan et al., 2012a]. Predicting failure-proneness in an evolving software product line. *Journal of Information and Software Technology*, 2012.

- (i) Does learner performance improve as the product line evolves?
- (ii) Is the set of prominent predictors consistent across products as the product line evolves?

RQ4. *Evolution of Components at Different Levels of Reuse*

- (i) Does the learner performance improve for components in each category of reuse (commonalities, high-reuse variation and low-reuse variation)? Does performance differ across categories of reuse?
- (ii) Is there a common set of best predictors across all categories of reuse?

RQ5. *Incremental Prediction*

- (i) Does performing incremental prediction (increasing the period of change data collection) improve the prediction performance?

The next five sections address these five sets of research questions in turn.

5.2 Classifier Selection

In this section, we explore RQ1 from the list of research questions. In our previous work [Krishnan et al., 2011b], we used the J48 machine learner to perform classification of failure-prone files. In the past, researchers have shown that prediction performance is not crucially dependent on type of classification technique used. Menzies, et al. [Menzies et al., 2007, Menzies et al., 2008] and Lessmann, et al. [Lessmann et al., 2008], observed that there is no statistical difference between the performance of most learners. However, there were a few learners that performed significantly worse than others.

We wanted to check whether J48 performs well enough when compared to other learners. Hence, we performed analysis similar to that of Lessmann, et al.

[Lessmann et al., 2008]. We evaluated a total of 17 classifiers, over the 11 distinct component datasets identified in Table 2.1. All the 17 chosen classifiers are implemented in the Weka machine learning software [Hall et al., 2009] and listed in Table 5.1.

Table 5.1: **List of Classifiers**

Type	Classifier
<i>Statistical</i>	Naive Bayes Bayesian Networks Logistic Regression Bayesian Logistic Regression
<i>Decision Tree methods</i>	J48 ADTree LADTree RandomForest
<i>Support Vector methods</i>	Voted Perceptron SPegasos SMO
<i>Neural Network methods</i>	RBF Network
<i>Nearest Neighbor methods</i>	IBk
<i>Others</i>	DecisionTable OneR Bagging with J48 RandomSubSpace with J48

We evaluated the performance of the 17 classifiers over the 11 components for the 2007 Europa release. As this was part of a pilot study and as we were interested in observing the general trends, we did not consider all the releases. We measured the AUC and the recall (TPR) values for each learner-component combination. To test whether the differences in AUC or TPR are significant, we carried out the Friedman test. A p-value $< 2.2 \times 10^{-16}$ suggested that the hypothesis of equal performances among the classifiers was unlikely to be true. This shows that there is a statistically significant difference between some pairs of learners. This was true when comparing AUC as well as TPR values. We then conducted post-hoc Nemenyi tests to find where was the difference, and represented the results with Demsar’s Critical Difference (CD) diagram [Demšar, 2006]. For 11 datasets and 17 classifiers the CD value was 7.45 at a significance level of 0.05.

The results of Nemenyi tests for the AUC and TPR values are shown in Figure 5.1. When using AUC as the performance measure, we find that there is no statistical difference between the top 10 classification algorithms. Furthermore, we observe that there is no significant difference between the performance of J48 learner and the observed best performer, RandomForest, both in terms of AUC and TPR. Since our focus is not on analysis of classifier performances, we do not present the details of the ranking of the different classifiers.

Figure 5.1 shows the results for the *UseAll_PredictAll* dataset (i.e., for each component, the change metrics and defect data encompass the entire 12 months). Similar results are observed for the *UseAll_PredictPost* and *UsePre_PredictPost* data. These results are shown in Figures 5.2 and 5.3, respectively. In all, there are 6 cases (3 types of dataset and 2 performance metrics, AUC and TPR). Although the individual rankings differ for each case, there is no statistical difference between the performance of J48 and the best learner in 5 out of 6 cases. Only for one case is there a statistically significant difference, i.e., (AUC ranking for *UsePre_PredictPost* dataset). For the *UsePre_PredictPost* dataset, we also see that there is no statistical difference between performance of the first 12 classifiers. These results are similar to the results observed by Lessmann et al. [Lessmann et al., 2008] who also found no statistical difference between the performance of most learners. Their results were for the MDP datasets which are based on static code metrics while ours are based on change metrics. Since J48’s performance was good overall, we continued our analysis in this paper with J48.

5.3 Single Product Evolution

In this section we discuss the performance of the J48 machine learner and the sets of prominent predictors for a single product, Classic, in the Eclipse product line. We look at each of the questions listed in RQ2 in Section 1.2.

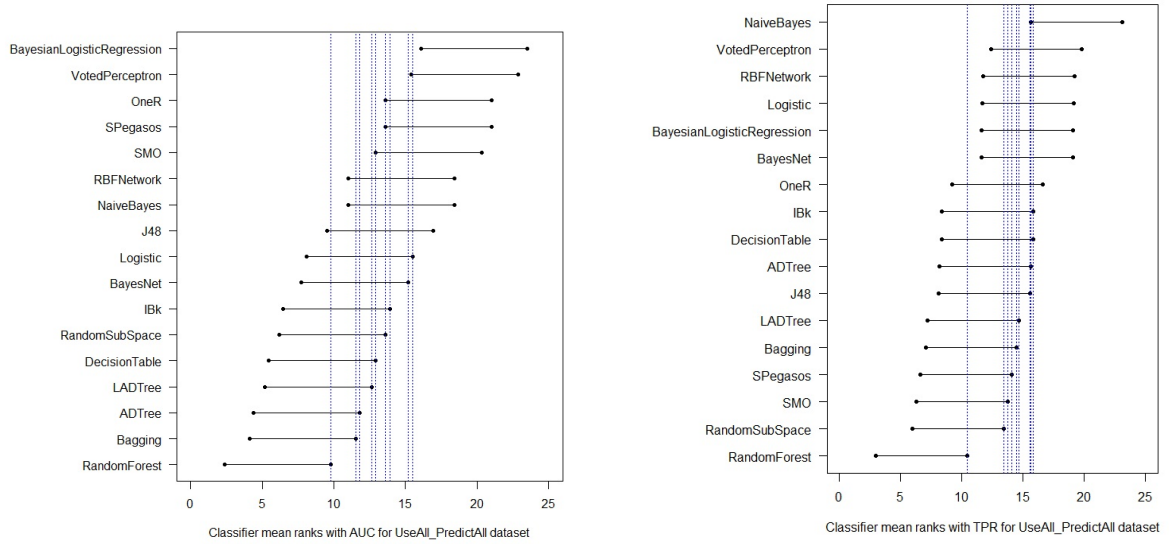


Figure 5.1: CD Diagram for AUC and TPR Ranks of UseAll_PredictAll Dataset

5.3.1 How do our results related to learner’s performance compare with previously published results?

Older releases of Eclipse did not have many components. Platform, JDT and PDE were the important components, and the combination of these three components was distributed as Eclipse SDK. This combination of components is now one product called Eclipse Classic in the Eclipse product line. Moser, Pedrycz and Succi in [Moser et al., 2008b] looked at three releases, 2.0, 2.1 and 3.0 of this product. We performed classification on the same three releases for this product using the J48 learner.

Table 5.2 compares our results with the results by Zimmermann et al. [Zimmermann et al., 2007] and Moser et al. [Moser et al., 2008b]. The authors in [Zimmermann et al., 2007] and [Moser et al., 2008b] used pre-release data to predict post-release defects. Hence we compare their results with our results for the *UsePre_PredictPost* dataset of Eclipse-Classic for releases 2.0, 2.1 and 3.0.

We see that our results using change data are better than the results of Zimmermann

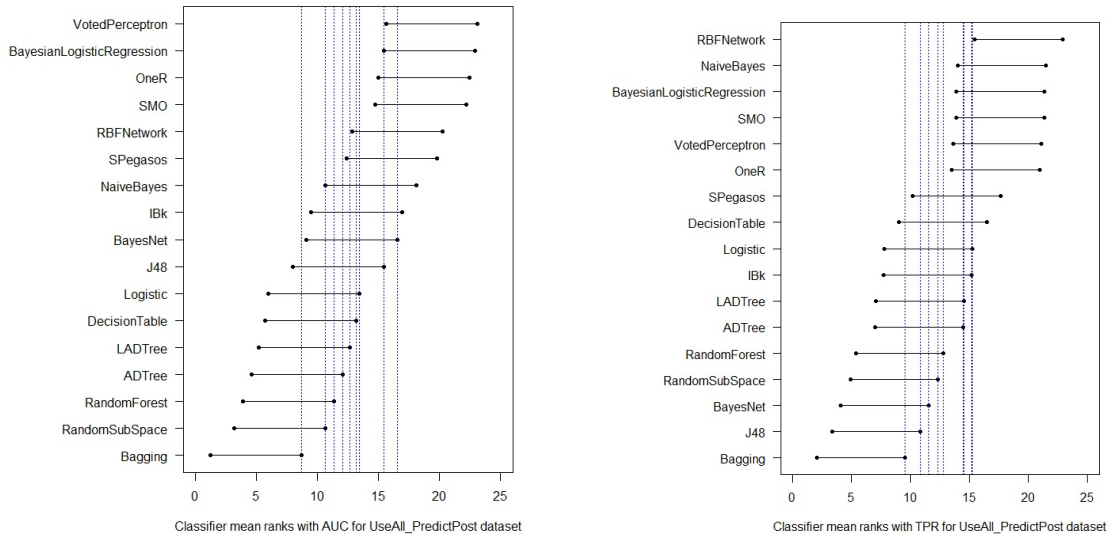


Figure 5.2: CD Diagram for AUC and TPR Ranks of UseAll_PredictPost Dataset

et al. [Zimmermann et al., 2007] which were based on using static data. The values of PC and Precision are similar to theirs, while the TPR and FPR values are much better than theirs. The TPR values reported by Moser et al. [Moser et al., 2008b] are higher than the TPR values we observed. It should be noted that the dataset used in [Moser et al., 2008b] is significantly smaller. Because that dataset is not publicly available, we are unable to further investigate the discrepancy of the results.

Table 5.2: Comparison of Classification Performance for 2.0, 2.1, and 3.0 Releases of Eclipse Classic for UsePre_PredictPost Dataset

Release	[Moser et al., 2008b]				[Zimmermann et al., 2007]				This study			
	PC	TPR	FPR	Precision	PC	TPR	FPR	Precision	PC	TPR	FPR	Precision
Classic-2.0	82	69	11	71	77	24	27	66	79	52	11	63
Classic-2.1	83	60	10	65	79	22	24	65	81	46	8	63
Classic-3.0	80	65	13	71	71	38	34	66	80	38	7	63

A reason for the difference in results may be the different number of files used by Moser et al. and us. The datasets used in [Moser et al., 2008b] consisted of significantly smaller subsets of the files in [Zimmermann et al., 2007], which was mentioned to be due to

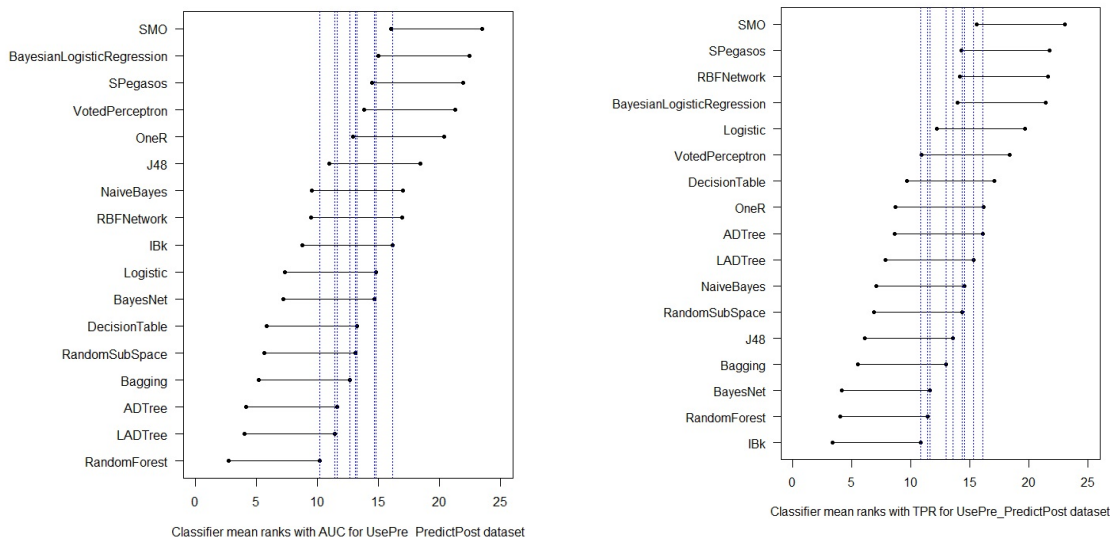


Figure 5.3: CD Diagram for AUC and TPR Ranks of UsePre_PredictPost Dataset

incomplete CVS history. Instead, we use the dataset used in [Zimmermann et al., 2007] as a reference point. As described in Section 1.3.1, our datasets are comparable in size to the datasets in [Zimmermann et al., 2007], with few differences between them.

5.3.2 Does learner performance improve as a single product evolves?

We next add to the analysis four additional releases of the same product, Eclipse Classic for the three types of datasets. The results in Table 5.3 show values for PC, TPR, FPR and AUC over the seven years for the three datasets, *UseAll_PredictAll*, *UseAll_PredictPost* and *UsePre_PredictPost*. The comparison over the three datasets reveals that results that may look promising when using a particular type of dataset need not hold for other type of datasets. In our case, the results are promising for the *UseAll_PredictAll* and *UseAll_PredictPost* datasets. However, when we look at more practical datasets like the *UsePre_PredictPost*, the results are much worse. PC, TPR, FPR and AUC values for *UseAll_PredictAll* and *UseAll_PredictPost* datasets are improv-

ing with time. For the later releases, the PC and TPR values are above 85% which is good. Similarly, the FPR values are as low as 2%. Quite opposite to the other two datasets, for *UsePre_PredictPost* the TPR values for the later releases of Eclipse-Classic are worse than for the older releases. The highest TPR value for the later releases is 40% for the Ganymede release.

We used statistical methods to test for differences in learner performance in time and then estimate the magnitude of the change in performance over time for each dataset. For each release of the Classic product, we computed the average PC, TPR, and FPR of the J48 learner over a ten-fold cross-validation. To reduce the variance in these estimated statistics, we repeated the ten-fold cross-validation 1,000 times.

First, we used one-way analysis of variance (ANOVA) to test for constant mean PC, TPR, FPR and AUC across all releases. For all three datasets, this hypothesis was resoundingly rejected (p-value $< 5 \times 10^{-16}$) for all three responses. The ANOVA assumption of normality was largely satisfied, except for response TPR on the Europa release (p-value 4×10^{-4}) for the *UseAll_PredictAll* dataset, for response PC on the Ganymede release (p-value 2×10^{-3}) for the *UsePre_PredictPost* dataset, and for response PC on the Galileo release (p-value 8×10^{-3}) for the *UseAll_PredictPost* dataset. The equal variance assumption was violated for all responses of all datasets (based on Figner-Killeen test p-values $< 5 \times 10^{-16}$). As a precaution against these violated assumptions, we carried out the non-parametric Kruskal-Wallis test which does not have assumptions about distributions. The hypothesis of equal distributions was resoundingly rejected (p-value $< 5 \times 10^{-16}$) for all four responses (PC, TPR, FPR and AUC) in all three datasets.

Table 5.3: Comparison of Results for Newer Releases (3.3-3.6) With Older Releases (2.0, 2.1, 3.0) of Eclipse Classic

Release	UseAll_PredictAll			UseAll_PredictPost			UsePre_PredictPost					
	PC	TPR	FPR	AUC	PC	TPR	FPR	AUC	PC	TPR	FPR	AUC
Classic-2.0	83.5	85.7	19.1	86.5	88.3	77.0	7.6	90.2	79.3	52.0	11.0	73.7
Classic-2.1	84.8	84.1	14.6	86.8	90.2	79.0	6.4	91.7	81.1	46.0	8.2	72.8
Classic-3.0	83.6	84.1	16.7	87.1	89.7	78.6	6.9	91.9	80.2	37.9	6.8	70.5
Classic-3.3	94.4	94.7	5.7	97.1	95.7	87.3	2.6	96.3	84.4	25.2	3.7	65.1
Classic-3.4	94.8	92.2	4.0	97	95.6	86.5	2.6	96.0	87.9	39.8	2.5	75.0
Classic-3.5	97.2	96.3	2.5	98.7	96.4	85.7	2.2	96.6	89.1	23.1	1.9	65.8
Classic-3.6	97.8	94.5	1.9	99	96.9	85.9	2.1	95.4	92.0	19.4	1.4	68.0
Estimated Slope of Improvement (in%)	2.0	1.7	-2.3	1.9	1.2	1.3	-0.8	0.8	1.5	-3.6	-1.1	-0.5
	**	**	**	**	**	**	**	**	**	**	**	**
p-value	0.0004	0.003	0.0001	0.0002	0.0001	0.007	0.0002	0.001	0.0002	0.004	8.6×10^{-05}	0.22

Given that there was change in PC, TPR, and FPR across releases, we next sought to characterize the size and direction of the trend over time. Our interest is in detecting possible trends in time and, since there are only seven releases (and only four in later sections), we restrict our attention to linear trends. If the temporal trend is in fact linear, then the estimated slopes are a more parsimonious and precise summary of the trend than pairwise post-hoc tests. To estimate the linear trend in PC, TPR, and FPR over time, we fit a linear mixed model to the 1,000 repeated measures for each release using *R* package nlme [Pineiro et al., 2001]. We estimated a separate variance for each release. The slopes and associated p-values for testing the null hypothesis of no temporal trend are shown in the last row of Table 5.3. Cells marked with ** denote values that are statistically significant at the 0.001 level. For the *UseAll_PredictAll* dataset, PC increased 2.0% per year (p-value 0.0004); TPR increased 1.7% per year (p-value 0.003); FPR decreased 2.3% per year (p-value 0.0001) and AUC increased 1.9% per year (p-value 0.0002). However, for the *UsePre_PredictPost* dataset, only PC and FPR have an improving trend, whereas TPR and AUC have a worsening, but not significant, trend. PC increased 1.5% per year (p-value 0.0002); TPR decreased 3.6% per year (p-value 0.004); FPR decreased 1.1% per year (p-value 8.6×10^{-05}) and AUC decreased 0.5% per year (p-value 0.22).

For the *UsePre_PredictPost* dataset it is difficult to assess whether performance is increasing or decreasing over time. However, there is a clear reduction in the TPR and AUC for the *UsePre_PredictPost* dataset as compared to the others. Thus, *training only on pre-release data makes it very difficult to successfully find post-release failures*. One likely reason for the high recall rates and improving performance for the *UseAll_PredictAll* and *UseAll_PredictPost* datasets is that the changes made to correct the post-release defects are included in the change data collected period. Another possible reason for the worse performance of the *UsePre_PredictPost* dataset is the lower percentage of defects, i.e., it is a less balanced dataset. Looking back at Table 3.3, we

can see that the percentage of defective files for *UsePre_PredictPost* dataset is between 6-27%, almost half of the percentage for the *UseAll_PredictAll* dataset. However, since the *UseAll_PredictAll* datasets also have high recall rates, class imbalance does not appear to be as important as the period of collection of change data and prediction data here. It appears that the continuous change observed [Krishnan et al., 2011a] even in the components that implement commonalities and high-reuse variabilities makes classification more difficult.

5.3.3 Is the set of prominent predictors consistent across releases of a single product?

We next explore whether the set of prominent predictors remains stable across releases for a single product in the product line, namely Eclipse Classic. To identify the prominent predictors, we order the 17 change metrics with decreasing gain ratio (GR) weights, and perform a step-wise feature selection approach followed by classification of each feature selected subset using the J48 machine learner. We run the following algorithm to perform the step-wise feature selection:

1. Let m be set of all metrics for the dataset.
2. Select $m' = \max GR(m)$
3. Add m' to the prominent predictor list.
4. Add m' to temporary dataset d'
5. Perform J48 classification on d' . Store result in R_1
6. Delete m' from m .
7. While $m \neq \phi$, repeat steps 8-12
8. Select $m'' = \max GR(m)$

9. Add m'' to d'
10. Perform J48 classification on d' . Store result in R_2
11. If (R_2 is statistically significantly better than R_1) then { Add m'' to prominent predictor list; $R_1 = R_2$ }
12. Delete m'' from m .
13. Output prominent predictors.

We performed the above steps for all releases of the Eclipse-Classic product. For each feature-selected dataset, we performed 10-fold cross validation. To test whether a metric should be included in prominent predictor list, we compared the performance when a new feature is added with the previous feature selected dataset (that resulted in a prominent predictor) using t-test. The feature with the highest GR is considered as prominent by default. To test if the feature with second highest GR should be included in the prominent predictor set, we do a t-test between 10 outputs of 10-fold CV for the second dataset (when the highest and second highest GR features are selected), and the 10 outputs of 10-fold CV of the dataset with only the highest GR feature. If the improvement is significant, we add the feature with second highest GR to the prominent predictor set. As multiple t-tests had to be performed, we applied a Bonferroni correction to the p-value. Since the number of t-tests to be performed was not known apriori (due to all metrics not contributing towards GR), we took a conservative approach for Bonferroni correction. A maximum of 16 t-tests would be performed if all features contribute towards GR and, each being a one-sided test to check for increase in the AUC value, we compared the p-value returned by t-tests with $0.05/16 = 0.003125$.

Results of the feature selection approach for the different releases of Eclipse Classic across the three types of datasets are shown in Tables 5.4 and 5.5. Table 5.4 gives the prominent predictors for the older releases of Eclipse-Classic, while Table 5.5 gives

the results for the newer releases. We find that in both tables the *UseAll_PredictAll* and *UseAll_PredictPost* datasets have a prominent predictor that is common across the respective sets of releases (*Revisions* for older releases of *UseAll_PredictAll*, *Authors* for newer releases of *UseAll_PredictAll* and *Revisions* for older and newer releases of *UseAll_PredictPost*). However the *UsePre_PredictPost* dataset does not have a prominent predictor that is common across all the considered releases. The previous study by Moser, Pedrycz and Succi [Moser et al., 2008b] identified *Bugfixes*, *Revisions* and *Max_Changeset* as the most common predictors. Although it did not mention using any statistical test to check for prominence, we find that there is some overlap between those results and our results for the *UsePre_PredictPost* dataset. We also find that *Bugfixes* and *Revisions* appear as prominent in more than one release. For the newer releases, in addition to *Bugfixes* and *Revisions*, we find that *Age* also appears in more than one release.

Table 5.4: **Comparison of Prominent Predictors for Older Releases of Eclipse Classic**

Release	Top 3 predictors from [Moser et al., 2008]	Top predictors from this study		
		<i>UseAll_PredictAll</i>	<i>UseAll_PredictPost</i>	<i>UsePre_PredictPost</i>
Classic-2.0	Max_Changeset, Revisions, Bugfixes	Revisions, Age, Authors	Revisions, Weighted_Age	Revisions, Loc_Deleted
Classic-2.1	Bugfixes, Max_Changeset, Revisions	Revisions, Ave_Changeset	Revisions, Weighted_Age	Bugfixes, Max_Changeset
Classic-3.0	Revisions, Max_Changeset, Bugfixes	Revisions, Max_Changeset, Age	Revisions, CodeChurn	Bugfixes, Revisions

Table 5.5: **Prominent Predictors for Newer Releases of Eclipse Classic**

Release	Top predictors		
	<i>UseAll_PredictAll</i>	<i>UseAll_PredictPost</i>	<i>UsePre_PredictPost</i>
Classic-3.3 (Europa)	Max_CodeChurn, Age, Loc_Added, Authors	Revisions , Max_Changeset, Max_Loc_Added	Revisions
Classic-3.4 (Ganymede)	Authors , Revisions, Age, Ave_Changeset	Revisions , Age, Ave_Changeset	Age, Bugfixes, Ave_Loc_Added
Classic-3.5 (Galileo)	Ave_CodeChurn, Age, Ave_Changeset, Authors	Revisions , Max_Changeset, Loc_Added, Authors	Revisions, Bugfixes
Classic-3.6 (Helios)	Authors , Ave_Changeset	Revisions , Authors, Bugfixes	Loc_Added, Age

5.4 Product Line Evolution

In this section we discuss how the performance of the machine learner and the sets of prominent predictors change as the product line evolves, looking at both of the questions in RQ3 given in Section 1.2. In addition to the Eclipse Classic product studied in Section 5.3, we applied the learning algorithm to three other products in the Eclipse product line, Eclipse Java, Eclipse JavaEE, and Eclipse C/C++.

5.4.1 Does learner performance improve as the product line evolves?

Figures 5.4, 5.5 and 5.6 show the results for PC, TPR and FPR across four years 2007-2010, for the four products in Eclipse’s product line, for the three types of datasets. The X-axis shows the four products and the Y-axis shows the PC, TPR and FPR values.

As in the case with the Eclipse-Classic product, we observe that across the product line, results show an improving trend for all products in the *UseAll_PredictAll* and *UseAll_PredictPost* datasets. In terms of correctly classified instances, all products have

PC rates above 94%. The true positive rates are almost all above 85% for both these datasets. False positives show very low values, less than 6% with the 2010 Helios release of JavaEE product having the lowest FPR for both datasets. For the *UsePre_PredictPost* dataset, we see similar results as in Section 5.3, i.e., although the PC and FPR values are improving with time, the recall values are low and do not show improvement. The highest recall value is of 60% for the 2007 Europa release of the JavaEE product.

The plots of Figures 5.4-5.6 appear to show some trends over time. Specifically, PC appears to increase; FPR appears to decrease; and TPR increases for two of the three datasets. To test whether this tendency is a global and significant trend across products, we regress each of these responses separately on time (release). We used a linear mixed model with random intercept to account for covariance due to repeated measures on the same product. The slope values along with the corresponding p-values are shown in Table 5.6. The estimated trends from these four years of data are similar to the results obtained from the Classic product over seven years (Table 5.3). However, none of the slopes estimated for *UsePre_PredictPost* dataset are significant, that is, the predictions do not show a recognizable trend as the product line evolves.

Similar to Section 5.3.2, there is no evidence to conclude any performance trend in time for the *UsePre_PredictPost* dataset, but there is an obvious reduced TPR for the *UsePre_PredictPost* dataset at *all* releases relative to the others. Why this is so is a topic of current research, but it seems that ongoing change [Krishnan et al., 2011a] is altering the patterns associated with failure as the products evolve in time. Products are made of both commonalities and variations, and it is reasonable to suspect that failure patterns are more stable in commonalities. In Section 5.5, we check to see if files from commonalities are easier to predict than files from variations.

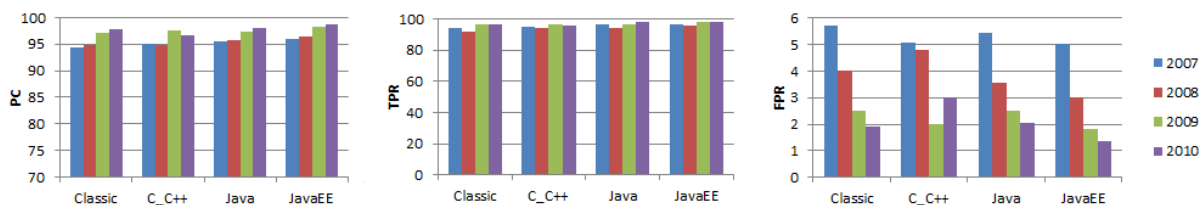


Figure 5.4: PC, TPR and FPR Comparison of Eclipse Products Across Releases for UseAll_PredictAll Dataset

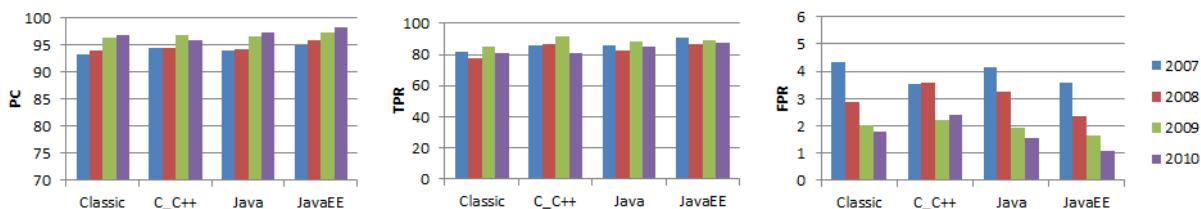


Figure 5.5: PC, TPR and FPR Comparison of Eclipse Products Across Releases for UseAll_PredictPost Dataset

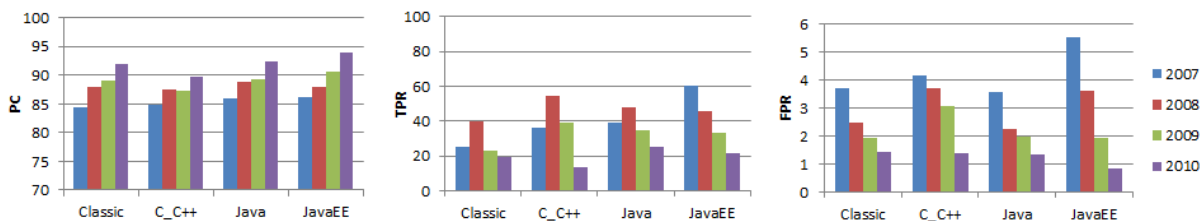


Figure 5.6: PC, TPR and FPR Comparison of Eclipse Products Across Releases for UsePre_PredictPost Dataset

Table 5.6: Performance Trends for All Products

Release	UseAll_PredictAll			UseAll_PredictPost			UsePre_PredictPost					
	PC	TPR	FPR	AUC	PC	TPR	FPR	AUC	PC	TPR	FPR	AUC
Estimated Slope of Improvement (in%)	0.9 **	0.7	-1.1 **	0.6 **	1.1 **	-0.2	-0.8 **	0.5	1.4	-2.3	-0.6	-0.4
p-value	3.9×10^{-05}	0.05	5.6×10^{-06}	9.6×10^{-05}	2.4×10^{-05}	0.81	3.1×10^{-06}	0.04	0.009	0.39	0.03	0.79

5.4.2 Is the set of prominent predictors consistent across products as the product line evolves?

In Section 5.3.3 we discussed the prominent predictors of failure-prone files over time for the three types of datasets for the Eclipse-Classic product. Here we investigate whether the set of prominent predictors differs for different products in the product line. We use the algorithm explained in Section 5.3.3 to identify the prominent predictors.

Table 5.7 compares multiple products across the 2007-2010 (Europa, Ganymede, Galileo and Helios) releases. Each cell gives a summary of the prominent predictors for that particular product and in how many of the four releases they appeared as prominent. We find that for the *UseAll_PredictAll* dataset, the *Authors* metric is common across all releases of all products, followed by *Ave_Changeset*, which is prominent in three releases of each product. For the *UseAll_PredictPost* dataset, *Revisions* is common, appearing in 15 of 16 releases across the four products. *Authors* and *Max_Changeset* are the next most common. For the *UsePre_PredictPost* dataset, however, there is no common predictor across products and releases. *Age* is the most common predictor, appearing in 11 of 16 releases across the four products. *Bugfixes* and *Revisions* are the next most common prominent predictors for the product line, appearing in 9 releases and 8 releases, respectively, across four products.

The observations suggest that while there are predictors which are common across all releases for the *UseAll_PredictAll* and *UseAll_PredictPost* datasets, for the *UsePre_PredictPost* dataset no common predictor exists across all releases.

Table 5.7: Prominent Predictors at Product Level

Dataset Type	Classic	Java	JavaEE	C/C++
UseAll_PredictAll	Authors:4 Ave_Changeset:3 Age:3 Loc_Added:1 Max_CodeChurn:1 Ave_CodeChurn:1	Authors:3 Revisions:3 Ave_Changeset:2 Loc_Added:1 Age:1 Weighted_Age:1 Max_Changeset:1 CodeChurn:1	Authors:4 Age:3 Ave_Changeset:3 Revisions:1 Loc_Deleted:1 Max_Changeset:1	Authors:4 Revisions:3 Ave_Changeset:3 Age:1 Max_Changeset:1
UseAll_PredictPost	Revisions:4 Max_Changeset:2 Authors:2 Ave_Changeset:1 Age:1 Max_Loc_Added:1 Loc_Added:1 Bugfixes:1	Revisions:3 Bugfixes:3 Authors:3 Max_Changeset:2 CodeChurn:1 Age:1 Ave_Loc_Added:1	Revisions:4 Authors:4 Max_Changeset:3 Age:1 Loc_Added:1 Refactorings:1 Max_CodeChurn:1	Revisions:4 Max_Changeset:3 Authors:2 Age:2 CodeChurn:1 Ave_Loc_Added:1 Max_CodeChurn:1 Max_Loc_Added:1 Ave_Changeset:1
UsePre_PredictPost	Bugfixes:2 Revisions:2 Age:2 Ave_Loc_Added:1 Loc_Added:1	Revisions:3 Age:3 Bugfixes:2 Max_Loc_Added:1	Bugfixes:3 Age:3 Authors:1 Revisions:1 Ave_Code_Churn:1	Age:3 Revisions:2 Bugfixes:2 Authors:1 Ave_Loc_Added:1

5.5 Evolution of Components at Different Levels Of Reuse

We explore the learner performance and consistency of predictors for components grouped by level of reuse (commonalities, high-reuse variations and low-reuse variations) considering both questions listed in RQ4 in Section 1.2.

5.5.1 Does the learner performance improve for components in each category of reuse? Does performance differ across categories of reuse?

Failure prediction at the product level showed that the prediction performance is improving across time only for PC and FPR, but not for recall. Products are an aggregation of components, so we wanted to observe whether there is an improvement in prediction for components in the different reuse categories. Intuitively, we expect that the learner performance would improve for each category of reuse. Since commonalities are reused in every product, change less and have fewer defects [Krishnan et al., 2011a], we expect the J48 learner to show better performance for higher reuse, i.e., performance improvement for commonalities to be better than high-reuse variations which in turn would be better than low-reuse variations. To explore this, we performed 10-fold cross validation using the J48 learner for the individual components.

We used a linear mixed effects model with random intercept to estimate the slope of improvement and considered the main and interaction effects of “time (year)” and “Type of reuse”. The overall increase/decrease rates for PC, TPR, FPR and AUC averaged across all components for the three types of datasets are shown in Table 5.8. The results are similar to the previous results obtained for products. For *UseAll_PredictAll* and *UseAll_PredictPost* datasets, we observe significant improvement trends for all the responses (with the exception of FPR and AUC for *UseAll_PredictAll*). For the *UsePre_PredictPost* dataset we see similar patterns as before, although PC is significantly improving for components.

We found that with time, there is an improvement in learner performance for each category of reuse for the *UseAll_PredictAll* and the *UseAll_PredictPost* datasets. Similar to the results in Table 5.6, most of the results for *UsePre_PredictPost* dataset are not statistically significant. For each dataset, when comparing the different categories of reuse, we found that no category has a performance increase that is significantly less (or more) than the overall improvement rate. Hence, the values in the Table 5.8 indicate the overall improvement rates for all three categories of reuse. In some cases, as expected, commonalities seem to be classified better than the other two categories, while for others commonalities are classified worse, which does not confirm our intuition.

It should be noted that except for three components (Platform, JDT and PDE), other components had change data for only four releases (2007-2010). Due to limited data we are not able to conclusively say whether one category of reuse performs better than the others. In addition, the components are much smaller in size compared to products and hence we expect more noise in the analysis at the component level.

Table 5.8: Performance Trends for Components at Different Levels of Reuse

Release	UseAll_PredictAll			UseAll_PredictPost			UsePre_PredictPost					
	PC	TPR	FPR	AUC	PC	TPR	FPR	AUC	PC	TPR	FPR	AUC
Estimated Slope of Improvement (in%)	1.9 **	2.8 **	-1.1	1.5	1.6 **	4.0 **	-0.9 **	2.1 **	1.4 **	-2.5	-1.0	-0.7
p-value	2.1×10^{-15}	2.0×10^{-08}	0.225	0.0017	5.5×10^{-12}	1.9×10^{-07}	2.5×10^{-07}	1.4×10^{-10}	5.9×10^{-05}	0.08	0.0055	0.275

5.5.2 Is there a common set of best predictors across all categories of reuse?

Next we explore whether the set of prominent predictors differs across component categories. We use the algorithm described in Section 5.3.3 for feature selection.

Table 5.9: **Prominent Predictors for Components at Different Levels of Reuse**

Dataset Type	Commonalities	High-Reuse Variations	Low-Reuse Variations
UseAll_PredictAll	Authors:3 Ave_Changeset:2 Max_Changeset:2 Revisions:2 Max_CodeChurn:1 Age:1 Max_Loc_Added:1	Revisions:3 Authors:2 Ave_Changeset:2 Age:2 Max_Changeset:1 Ave_CodeChurn:1 Max_CodeChurn:1 Weighted_Age:1	Authors:4 Age:3 Max_Changeset:2 Revisions:1 Ave_Changeset:1
UseAll_PredictPost	Revisions:3 Max_Changeset:3 Authors:3 Weighted_Age:1 Max_CodeChurn:1 Max_Loc_Added:1 Loc_Added:1	Revisions:4 Authors:3 Max_Changeset:2 Bugfixes:2 Age:2 Code_Churn:1 Ave_Changeset:1 Loc_Added:1	Max_Changeset:3 Loc_Added:3 Weighted_Age:2 Age:1 Authors:1 Bugfixes:1 Revisions:1 Max_CodeChurn:1
UsePre_PredictPost	Bugfixes:2 Authors:2 Loc_Added:2 Age:1 Max_Changeset:1	Age:4 Bugfixes:2 Ave_Changeset:1 Weighted_Age:1 Max_Code_Churn:1	Age:3 Bugfixes:2 Weighted_Age:1 Max_Loc_Added:1 Revisions:1

Table 5.9 lists the prominent predictors for the three reuse categories, for the three types of datasets. Each cell gives a summary of the prominent predictors for that particular reuse category and in how many of the four releases they appeared as prominent. We observe that there is some overlap among the prominent predictors for the three reuse categories. For the *UseAll_PredictAll* dataset, the metric *Authors* is prominent and common across all three reuse categories. Similarly for the *UseAll_PredictPost* dataset, *Bugfixes* and *Max_Changeset* are common and prominent across all reuse categories. For the

UsePre_PredictPost dataset, the metric *Bugfixes* is common across all reuse categories, although it appears as prominent in only two of the four releases (2007-2010). Additionally, the metric *Age* is also common between the two types of variations (high-reuse and low-reuse) and appears in three or more releases. *Age* is prominent for commonalities in only a single release. This indicates that while there are some metrics that are prominent across all reuse categories, there are also differences among the prominent predictors for the different reuse categories.

5.6 Prediction with Incrementally Increasing Data Collection Periods

In this section we explore RQ5. Results in sections 5.3, 5.4 and 5.5 showed that predicting post-release failure-prone files using pre-release change data gives low recall values. In this section we investigate whether increasing the period of collecting change data improves the prediction of failure-prone files. The *UsePre_PredictPost* type of datasets uses 6 months pre-release data to predict failure-prone files 6 months post-release. We would like to investigate whether using post-release change data in monthly increments, combined with pre-release change data helps to better classify post-release failure-prone files in the remaining months. In our incremental approach we begin from the *UsePre_PredictPost* dataset (i.e., using 6 months pre-release change data to predict 6 months post-release failure-prone files). We increment the change data period from 6 months to 11 months in increments of 1 month, while simultaneously reducing the post-release failure-prone file data from 6 months to 1 month, i.e., our final dataset will have 11 months of change data to predict failure-prone files in the 12th month.

Figure 5.7 shows the results of incremental prediction for the four products in the product line. We find that increasing the period of change data does not improve recall values. One possible reason is that as the period of change data increases (from 6 months

to 11 months), the number of files that are failure-prone in the remaining months reduces. As a result the J48 learner may not have a sufficient number of defective files from which to learn. We find that for the last two iterations the recall values drop as compared to the first four iterations.

Similar results are observed for the three reuse categories, as shown in Figure 5.8. Even commonalities, which should change less and hence have a good classification performance, show low recall values. In fact, the recall values for commonalities are in some cases lower than for the other two reuse categories. High-reuse variations have the highest recall values.

Results from Sections 5.3.2 and 5.4.1 indicated that using only pre-release change data to predict post-release failure-prone files is difficult. The results presented in this section indicate that even when post-release change data are added to pre-release change data, the predictions do not improve.

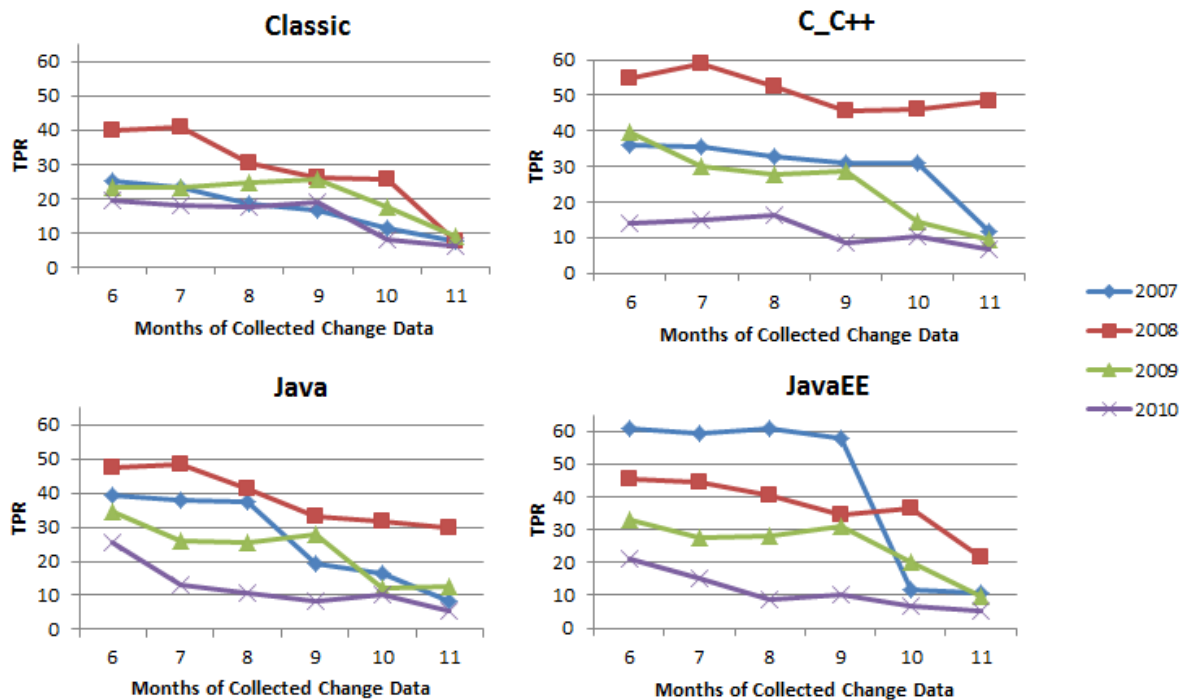


Figure 5.7: Incremental Prediction for Four Eclipse Products

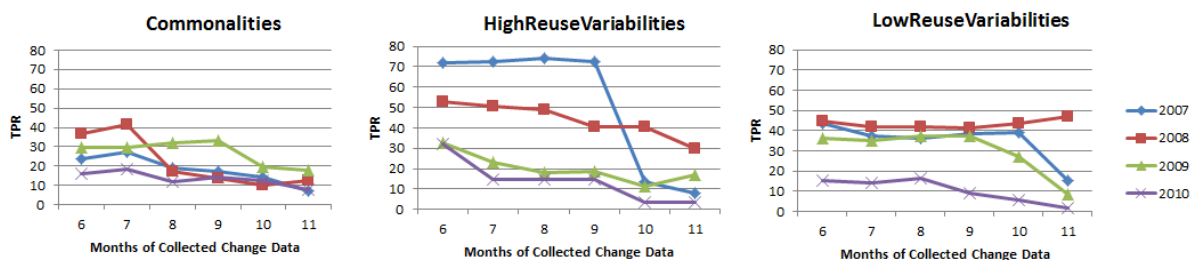


Figure 5.8: Incremental Prediction for Three Reuse Categories

5.7 Summary of the Results

The highlights of the observations from the study are summarized as follows:

We initially used the J48 decision tree learner to classify failure-prone files. In experiments with other learners, we found that there is no statistically significant difference between the performance of J48 and learners which perform slightly better (e.g., Random Forest).

A replication study, comparing our results with results from previous studies for the same releases of Eclipse-Classic, showed that while change metrics are better predictors than static metrics, predicting post-release failure-prone files using pre-release data led to low recall rates. Although accuracy and false-positive rates were impressive, the low recall rates suggest that it was difficult to classify failure-prone files effectively based on pre-release change data.

A comparison between different types of datasets distinguished by the data collection and prediction period showed that datasets that do not distinguish pre-release period with post-release period (similar to MDP) have better performance with respect to accuracy, recall and false-positive rate.

From the product line perspective, prediction of failure-prone files for four products in the Eclipse product line based on pre-release data did not show recognizable trends across releases (i.e., the estimated trends were not statistically significant).

When comparing the prediction trends among the three categories of reuse (i.e., commonalities, high-reuse variations and low-reuse variations), the results showed statistically significant improvement in accuracy, but not statistically significant trends for the other performance metrics.

As each product evolved, there was a set of change metrics that were consistently prominent predictors of failure-prone files across its releases. This set was different for the different types of datasets (with respect to change and defect data collection period) considered in this study.

There was some consistency among the prominent predictors for early vs. late releases for all the considered products in the product line. This set was different for the different types of datasets considered here. For predicting post-release failure-prone files using pre-release change data, the subset of change metrics, *Bugfixes*, *Revisions* and *Age* was among the prominent predictors for all the products across most of the releases.

Looking at the evolution of the different categories of components in the product line (i.e., commonalities, high-reuse variations and low-reuse variations), we found that there was consistency among the prominent predictors for some categories, but not among all categories. For predicting post-release failure-prone files using pre-release change data, the change metric *Bugfixes* appeared to be prominent in all three categories, although not across all releases. Metrics such as *Age* were prominent across more than one category but not across all three of them.

CHAPTER 6. THREATS TO VALIDITY

This section discusses the threats to validity of this study.

Construct Validity. For the defect assessment study, a possible threat to the construct validity is that the number and severity of failures may be affected by the expertise of the programmers who worked on the components. To alleviate this, we have normalized the failures with respect to the number of changes and additions to source code files, rather than normalizing it with the lines of code for each component. This is in accordance with Mockus, Fielding and Herbsleb who identify the programmer’s lack of expertise, leading to unnecessarily lengthy code, as one of the reasons for seemingly lower failure density [Mockus et al., 2000]. In addition, because each component in Eclipse is typically developed by multiple programmers, this threat may be alleviated.

Another threat to the construct validity is the limited number of releases in the study. While analyzing more releases might give additional insight into the trends, the 2007-2010 releases provide a representative picture of the current product line situation. We did not include the minor quarterly releases into our analysis because there were fewer users downloading them and because the entries in the bug database for these minor releases were missing data for several components. Furthermore, some of the minor releases reported higher numbers of failures while others did not report any.

As mentioned by Moser, Pedrycz and Succi in [Moser et al., 2008b], a possible threat to construct validity could be the choice of metrics used in this study. We followed [Moser et al., 2008b] in using a particular set of change metrics. In general, there could be other change metrics that give different results. We believe that our results are compara-

ble to results from previous studies which evaluate the performance of different metric sets in terms of classification of failure-prone files. Arisholm, et al. in [Arisholm et al., 2010] observe process metrics to be the best metric set. However, they also report low recall and precision values (in the range of 36% to 62%) when using process metrics.

Internal Validity. Inaccuracies in our data collection process at one or more steps could be one of the possible threats to internal validity of this study. We performed manual and automated inspections on our dataset to verify and validate its accuracy, including comparison with data provided by Zimmermann, Premraj and Zeller in [Zimmermann et al., 2007]. Chapter 3 Section 3.2.1 discusses these inspections. Iterative inspections on the dataset were conducted by 2 programmers to identify and correct duplicate and missing entries, and to ensure accurate mapping of the Eclipse Bugzilla database to the source code version control repositories.

Conclusion Validity. A possible threat to the conclusion validity of the defect assessment study is that we analyzed only one common component, namely Platform. To moderate this we investigated five sub-components of Platform. Platform is a large component, and each of these sub-components is comparable in size to other components in our study. Each of these Platform sub-components provides a specific common functionality. Also, each of these sub-components has a large number of severe failures.

For the defect prediction part, one threat to the conclusion validity may be that we performed analysis using only one machine learning algorithm, namely J48. Moser et al. [Moser et al., 2008b] additionally used Naïve Bayes and logistic regression learners but found J48 to give the best results. We also analyzed the performance of 17 machine learners, including J48, and found that there is no statistically significant difference between the performance of J48 and learners with higher mean rank (e.g., Random Forest).

Another possible threat to the conclusion validity is the class-imbalance problem. The datasets used in this study are imbalanced, i.e., the proportion of defective files is

smaller than the percentage of non-defective files. Several studies have found that the learner performance improves when trained on balanced data, using techniques such as over-sampling, under-sampling, etc. [Menzies et al., 2008, Drummond and Holte, 2003]. Our emphasis in this work is on the trends in prediction performance as the product line evolves. When we carried out the tests to check performance trends (Sections 5.3.3, 5.4.2 and 5.5.1) on both balanced and imbalanced datasets, we found that the slopes of the trends (either improving trend or worsening trend) hold for both datasets. While the performance might be improved using balancing methods, it appears that the trends in defect prediction as the product line evolves do not depend on the balancing of datasets.

Another possible threat to the conclusion validity, as pointed out by D. Weiss [Weiss, 2013], is the applicability of the underlying assumption that predictive models from one product or release can be used to predict defects in another product or release. Software changes between releases due to bug fixes and enhancements. Our research attempts to validate the existing predictive models in relation to this assumption. The difficulty in accurate prediction of defective files may indicate the dynamic nature of software and the resulting inapplicability of existing defect prediction models.

A typical threat to conclusion validity relates to the validity of the assumptions of the statistical tests and errors in statistical conclusions. As much as possible, we analyzed the validity of the statistical assumptions. Given the small number of releases, the linear mixed effects models parsimoniously account for some correlation among responses due to repeated temporal measures on the same product or component, but far more complex correlation is likely. Releases close in time are more likely to share common characteristics. Further, many files, especially high reuse files, are included in multiple products and hence contribute to multiple performance responses. Improper modeling of the covariance can have a large impact on estimated significance levels. The component datasets do not suffer from the potentially unaccounted covariance due to shared files because each file exists in only one component, so the component p-values are likely the

most reliable.

Another possible threat to the conclusion validity is related to the normalization of failure rates. For the defect assessment part of our work, we have normalized the failure rate using number of changes to each component. A desirable measure of normalization is the amount of usage for each component, such as product installation information. Such data are hard to collect. Eclipse keeps track of the number of downloads for each component. However, using this measure for normalization may not lead to accurate results. Other methods of normalization of failure-rates must be evaluated.

Finally, we included releases from products spanning 2002 through 2010, but only the Classic product and its components were available prior to 2005, and the years 2005 and 2006 were not sampled.

External Validity. An external validity threat to this study is the extent to which these observations can be generalized to other product lines. Eclipse was not intended to be developed as a product line, but has adopted some of the methodologies of product line engineering, such as a common repository for all products, reuse of components from previous products in new products, and selection and integration of needed features in a new product. Eclipse, however, does not keep track of variability constraints as with product line engineering. Further, Eclipse is a large product line with many developers in an open-source, geographically distributed effort. This may mean that the development of the Eclipse product line is more varied in terms of the people involved and the development techniques used than in commercial product lines. Chastek, McGregor and Northrop consider the open-source development to be largely beneficial in terms of quality [Chastek et al., 2007]. We hope to study other open-source software product lines and have studied an industrial software product line [Devine et al., 2012] to learn more about reuse, change and reliability in product lines. We have made our dataset [Krishnan et al., 2012b] public so that other researchers can validate the results of this study and/or use it to conduct other types of analysis.

CHAPTER 7. CONTRIBUTIONS AND FUTURE WORK

This chapter summarizes the consequences of our results for the Eclipse product line. It also describes more broadly the potential impact of these results for other open-source and industrial product lines. Finally, the chapter briefly proposes some directions for future work to further investigate defect assessment and prediction in product lines.

7.1 Contributions

The empirical study of changes and defects in the Eclipse product line identifies several important considerations both for Eclipse and for other product lines.

1. *The majority of files with severe defects are reused files rather than new files.* This finding is contrary to the widespread assumption that product line reuse of files improves the quality of those files. For example, [Card et al., 1986] report that 98% of the modules reused without modification in five flight dynamics software projects were fault free. Similarly, [Mohagheghi and Conradi, 2008] report that reused components exhibited lower fault density than non-reused components in a large commercial telecommunication product line.

Table 7.1 shows the relationship between reused files and severe defects for the four yearly releases (2007-2010) of the Eclipse product line for each of the three reuse categories, commonalities, high-reuse variations and low-reuse variations. Column 3 shows the total number of files in each reuse category. Column 4 gives the number of files reused in this release from the previous release. Reused

files are the files which were present in the previous release and are also present in the current release. Column 5 gives the number of files that have been reused as-is (i.e., without any modification). Columns 6, 7 and 8 give the total number of files with severe defects, the number of reused files with severe defects and the number of files reused as-is with severe defects, respectively. Columns 9 and 10 give the percentage of the files with severe defects that are reused and are reused as-is, respectively.

Table 7.1 shows that the majority of the severe defects found are in reused code. For the commonalities, 75% to 96% of the files with severe defects are reused files, and 13% to 52% of the files with severe defects are files that have been reused as-is. This is interesting because 78% to 95% of the files are reused from the previous release, and 48% to 81% are reused as-is. Most surprising is that *even common components experience ongoing change* as seen from columns 3 and 4. This suggests that the on-going evolution of common components negatively affects the quality of the reused components. For the Eclipse product line, reducing change in reused files and reducing change in common components would be likely to reduce severe defects.

Whether this is practical is an open question. However, in our discussions with IBM project managers, they identified the finding that a majority of reused files have severe defects as important information for the Eclipse project as well as other IBM projects. The potential benefit offered by reuse of code is hampered when reused code contributes the majority of the severe defects.

2. *On-going change in product lines hinders the ability to predict failure-prone files.* While the Eclipse components benefit from high reuse, the ongoing change makes it difficult to predict failure-prone files effectively, allowing only 40 to 60% of the defective files to be classified correctly. [Kim et al., 2011] found that the refactoring

process in Eclipse, although it facilitates bug fixing, also induces bugs. Moreover, Eclipse undergoes bug fixes on a daily basis with nightly builds that often may not be stable. Further investigation is needed to know the nature of these fixes and refactorings in Eclipse to get a better understanding of how to predict these bugs.

3. *Classification of post-release failure-prone files using change data for the Eclipse product line gives better recall and false positive rates as compared to classification using static code metrics.* Table 7.2 compares our results using change data to those of [Zimmermann et al., 2007] using static code metrics for Eclipse releases 2.0, 2.1 and 3.0 (2002-2004). Table 7.2 compares the accuracy, recall, false-positive rate and precision performance of the two studies. Chapter 3 defined these performance metrics.

The accuracy values obtained in our study are comparable or better than the Zimmermann study. Similarly, the recall values we obtained using change metrics are better than those previously obtained using code metrics. Most importantly, the false-positive rates we obtain using change metrics are very low. Low false-positive rates are essential for practical use of prediction by developers since high false-positive rates can be very costly due to wasteful inspection and testing of files that are not defective. The precision values we obtained using change metrics are lower than those previously achieved using code metrics. To put this in context, [Menzies et al., 2007] note that optimizing for one of precision and recall will often compromise the other. They advise that in many industrial situations, low precision and high recall detectors are useful especially for datasets, such as Eclipse, where the number of defective files is much smaller than the number of non-defective files.

Two other studies on classification of defective files for Eclipse have confirmed the value of change metrics. [Moser et al., 2008b] report even better recall values on a small earlier Eclipse dataset that is not publicly available. [D'Ambros et al., 2012]

Table 7.1: Severe Failures in Reused Code for Component Categories

(1) Reuse type	(2) Year	(3) Total number of files	(4) Number of reused files	(5) Number of as-is, reused files	(6) Total files with severe failures	(7) Reused files with severe failures	(8) As-is reused files with severe failures	(9) % of files with severe failures that are reused	(10) % of files with severe failures that are as-is reused
Commonality	2007	9084	7137	4396	205	161	28	78.5	13.7
	2008	9912	8495	6312	160	121	27	75.6	16.9
	2009	9668	9141	6249	225	217	117	96.4	52.0
	2010	9933	9453	8065	181	171	65	94.5	35.9
High-Reuse Variation	2007	17933	10942	5332	451	284	56	63.0	12.4
	2008	19377	12591	8076	440	274	61	62.3	13.9
	2009	19711	17107	13088	322	200	70	62.1	21.7
	2010	19926	18541	15988	238	208	101	87.4	42.4
Low-Reuse Variation	2007	12647	7156	4412	277	142	24	51.3	8.7
	2008	14471	9777	7259	421	240	78	57.0	18.5
	2009	16219	12684	9559	180	131	66	72.8	36.7
	2010	17357	14549	11212	123	101	39	82.1	31.7

report high AUC (Area Under ROC Curve, defined in Chapter 3) values of 85-92% for two Eclipse components for data collected from 2005-2008. More broadly, we observed high AUC values of 65-75% (as shown in Table 5.3 in Chapter 5) for three Eclipse components for data collected from 2002-2010.

Table 7.2: Comparing Cross-release Classification Performance Using Static Code Metrics and Change Metrics

Trained On	Tested On	Zimmermann et al.				This study			
		Acc.	Rec.	FPR	Pre.	Acc.	Rec.	FPR	Pre.
Classic-2.0	Classic-2.0	76.6	24.2	27.3	65.7	91.5	77.8	3.7	88.3
Classic-2.0	Classic-2.1	78.3	25.9	22.6	57.8	75.7	40.6	13.6	47.6
Classic-2.0	Classic-3.0	68.2	24.4	41.6	63.8	76.8	33.5	9.9	51.0
Classic-2.1	Classic-2.1	78.9	21.9	23.9	64.5	88.1	59.5	3.2	85.2
Classic-2.1	Classic-3.0	68.2	19.5	44.3	68.7	77.6	35.3	9.4	53.6
Classic-3.0	Classic-3.0	71.1	37.9	34.2	66.3	86.2	52.5	3.4	82.6

4. *Predicting post-release defects using pre-release change data for the Eclipse case study is difficult.* What interests project managers the most is the capability to predict post-release defects using pre-release data. However, it is more difficult than predicting post-release defects using pre and post-release change data and also more difficult than predicting total defects using pre and post-release change data.

We thus have investigated the effect of the period of data collection on prediction. Studies in the past have shown good results when no distinction is made between pre-release and post-release defects ([Menzies et al., 2007], [Menzies et al., 2011], [Bettenburg et al., 2012]). At least for the Eclipse product line, we found that predicting post-release failure-prone files using pre-release change data yields low recall rates. Previous defect prediction results on Eclipse using code metrics, [Zimmermann et al., 2007], also showed that at the file level, predicting post-release failures using pre-release code metrics data gives poor recall as well as false-positive rates. However, previous studies on Microsoft systems showed that effective pre-

diction of post-release failure-prone files can be performed using pre-release change data ([Nagappan et al., 2006a], [Nagappan et al., 2010]). Incorporating new prediction techniques such as clustering ([Menzies et al., 2011]), or topic based defect prediction ([Nguyen et al., 2011]) might improve recall rates for Eclipse similar to previous studies.

5. *Using more data from the past to predict future failure-prone files does not necessarily give better results than using data only from the recent past.* Data collected during the evolution of the Eclipse products does not improve prediction ability. We performed studies aggregating data over multiple years to predict defective files for a future release. For example, we collected the aggregated change and defect data from January, 2007 to December, 2008 and used it to predict the defective files in the 2009 release. The results were worse than using data from the immediate past, i.e., from using data only from 2008 to predict 2009.

In Chapter 5 Section 6, we describe increasing the period of change data in monthly increments and predicting the post-release failure-prone files in the remaining months. The results showed that even when we use a portion of the post-release change data in combination with pre-release change data, the predictions do not necessarily improve. This tends to indicate that defect prediction models may be more applicable for product lines that change less than Eclipse.

6. *Common components experience less change than variation components.* This is as expected for a product line. However, the finding that the amount of change in commonalities does not decrease over time, as seen in Table 7.1, is contrary to expectations. Whether this on-going change is typical of other product lines merits additional investigation.
7. *There is a consistent set of metrics which serve as prominent predictors across multiple products and reuse categories over time.* We found that *Age*, *Bugfixes* and *Re-*

visions were the prominent predictors at the product level. For the reuse-category level, *Age* and *Bugfixes* were prominent across the multiple reuse categories.

These results give additional indication of the benefit of change metrics over static metrics. For example, [Menzies et al., 2007] found that when static metrics are used, the set of their best defect predictors were different for different datasets. Having a consistent set of change metrics as good predictors can be especially useful in a product line for predicting failure-prone files across products as well as for future releases of the same product.

7.2 Future Work

The results from this empirical investigation open up the following future research directions.

- This empirical research is based on extensive study of a single open-source product line as well as on a defect assessment study of a commercial product-line. To test and strengthen the conclusions, the assessment and prediction methods used in this work should be applied to other product lines, both from the commercial as well as the open-source community.
- This study used change metrics, based on their good classification performance in previous research. The comparative classification of defective files in different reuse categories might be enhanced by incorporating information about the developers' and the projects' characteristics. Specific techniques to be explored are developer metrics [Bird et al., 2011] and socio-technical metrics [Bird et al., 2009].
- Discussions with Eclipse developers have sparked their interest in exploring how Eclipse can benefit more from product line engineering techniques. Managers from IBM are interested in examining whether the reuse of a component in different

products leads to different types of bugs. However, the Eclipse Bugzilla repository currently does not keep track of the product in which the bug is encountered, e.g., whether the bug was encountered while Eclipse JAVA or Eclipse JAVAEE was being used. The rationale for not logging the product information is that Eclipse follows a plugin-based development to give users more flexibility. For example, users can download a C++ plugin and use it in the JAVA product if they choose. However, our manual inspection of bugs found that some bugs occur only in the presence of plugins which are part of only certain products. Hence, changing the bug reporting process to keep track of product information will be essential to understanding product specific bugs.

- This work performed binary classification of files (defective vs. non-defective). A promising avenue to explore is whether results for predicting the true number of defects using regression techniques show similar or different results. We are supporting work at West Virginia University to investigate this using combinations of code and change metrics.

This work used product line change and defect data from Eclipse to empirically investigate the widely accepted assumption that as a product line evolves, its reliability improves. The work consisted of two parts, defect assessment and defect prediction. Defect assessment results showed that common components experienced less change than variation components. However, the majority of files with severe defects were reused files rather than new files, which challenged the assumption of increasing reliability over time. Defect prediction results showed that on-going change in product lines hindered prediction of failure-prone files and that the on-going change negatively affected the stabilizing behavior intended by planned reuse. More encouragingly, the study identified a set of change metrics that are consistently prominent in predicting failure-prone files across the multiple products and reuse categories in the product line. This empirical

investigation led to an improved understanding of the interplay among change, reuse and reliability as a product line evolves.

Bibliography

- [cvs, 2004] (2004). CVSChangeLogBuilder, tool for generating CVS log reports. <http://cvschangelogb.sourceforge.net/>.
- [MDP, 2004] (2004). Nasa IV&V Metrics Data Program. <http://nasa-softwaredefectdatasets.wikispaces.com/>.
- [Arisholm et al., 2010] Arisholm, E., Briand, L. C., and Johannessen, E. B. (2010). A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models. *Journal of Systems and Software*, 83(1):2–17.
- [Avizienis et al., 2001] Avizienis, A., Laprie, J.-C., and Randell, B. (2001). Fundamental Concepts of Dependability. *UCLA Report*.
- [Basili et al., 1996] Basili, V. R., Briand, L. C., and Melo, W. L. (1996). A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761.
- [Batory et al., 2006] Batory, D. S., Benavides, D., and Cortés, A. R. (2006). Automated Analysis of Feature Models: Challenges Ahead. *Communications of the ACM-Software Product Line*, 49(12):45–47.
- [Bettenburg et al., 2012] Bettenburg, N., Nagappan, M., and Hassan, A. E. (2012). Think Locally, Act Globally: Improving Defect and Effort Prediction Models. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, MSR '12*, pages 60–69, Zurich, Switzerland.

- [Bird et al., 2009] Bird, C., Nagappan, N., Gall, H., Murphy, B., and Devanbu, P. (2009). Putting it all Together: Using Socio-technical Networks to Predict Failures. *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering*, pages 109–119.
- [Bird et al., 2011] Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. (2011). Don't Touch My Code!: Examining the Effects of Ownership on Software Quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ESEC/FSE '11*, pages 4–14, Szeged, Hungary. ACM.
- [Borretzen and Conradi, 2006] Borretzen, J. A. and Conradi, R. (2006). Results and Experiences from an Empirical Study of Fault Reports in Industrial Projects. In *Proceedings of the 7th International Conference on Product-Focused Software Process Improvements, PROFES '06*, pages 389–394. Springer, LNCS.
- [Card et al., 1986] Card, D. N., Church, V. E., and Agresti, W. W. (1986). An Empirical Study of Software Design Practices. *IEEE Transactions on Software Engineering*, 12(2):264–271.
- [Chastek et al., 2007] Chastek, G., McGregor, J., and Northrop, L. (2007). Observations from Viewing Eclipse as a Product Line. In *Proceedings on the Third International Workshop on Open Source Software and Product Lines*, pages 1–6.
- [Clark and Zubrow, 2001] Clark, B. and Zubrow, D. (2001). How Good is Software: A Review of Defect Prediction Techniques. In *Software Engineering Symposium*, pages 1–35, Carnegie Mellon University, Pittsburgh, PA.
- [Clements and Northrop, 2001] Clements, P. C. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley.

- [D'Ambros et al., 2009] D'Ambros, M., Lanza, M., and Robbes, R. (2009). On the Relationship Between Change Coupling and Software Defects. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering, WCRE '09*, pages 135–144, Washington, DC, USA. IEEE Computer Society.
- [D'Ambros et al., 2012] D'Ambros, M., Lanza, M., and Robbes, R. (2012). Evaluating Defect Prediction Approaches: A Benchmark and an Extensive Comparison. *Journal of Empirical Software Engineering*, 17(4-5):531–577.
- [Demšar, 2006] Demšar, J. (2006). Statistical Comparisons of Classifiers Over Multiple Data Sets. *The Journal of Machine Learning Research*, 7:1–30.
- [Denaro et al., 2003] Denaro, G., Lavazza, L., and Pezze, M. (2003). An Empirical Evaluation of Object Oriented Metrics in Industrial Setting.
- [Devine et al., 2012] Devine, T. R., Goseva-Popstajanova, K., Krishnan, S., Lutz, R. R., and Li, J. J. (2012). An Empirical Study of Pre-release Software Faults in an Industrial Product Line. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 181–190, Washington, DC, USA. IEEE Computer Society.
- [Drummond and Holte, 2003] Drummond, C. and Holte, R. (2003). C4.5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling. In *In Proceedings of the ICML 2003 Workshop on Learning from Imbalanced Datasets*.
- [Eaddy et al., 2008] Eaddy, M., Zimmermann, T., Sherwood, K. D., Garg, V., Murphy, G. C., Nagappan, N., and Aho, A. V. (2008). Do Crosscutting Concerns Cause Defects? *IEEE Transactions on Software Engineering*, 34(4):497–515.
- [Eclipse, a] Eclipse. Eclipse Bugzilla homepage. <https://bugs.eclipse.org/bugs/>.

- [Eclipse, b] Eclipse. Eclipse Bugzilla Wiki homepage. http://wiki.eclipse.org/Eclipse/Bug_Tracking.
- [Fenton and Ohlsson, 2000] Fenton, N. E. and Ohlsson, N. (2000). Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering*, 26(8):797–814.
- [Gomaa, 2004] 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, USA.
- [Gray et al., 2011] Gray, D., Bowes, D., Davey, N., Sun, Y., and Christianson, B. (2011). The Misuse of NASA Metrics Data Program Data Sets for Automated Software Defect Prediction. In *Proceedings of Evaluation and Assessment in Software Engineering, EASE '11*, pages 96–103.
- [Guo et al., 2010] Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B. (2010). Characterizing and Predicting Which Bugs Get Fixed: An Empirical Study of Microsoft Windows. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE'10*, pages 495–504, New York, NY, USA. ACM.
- [Hall et al., 2009] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA Data Mining Software: An Update. *SIGKDD Explorer Newsletter*, 11:10–18.
- [Hall et al., 2012] Hall, T., Beecham, S., Bowes, D., Gray, D., and Counsell, S. (2012). A Systematic Review of Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering*, 38:1276–1304.
- [Hamill and Goševa-Popstojanova, 2009] Hamill, M. and Goševa-Popstojanova, K. (2009). Common Trends in Software Fault and Failure Data. *IEEE Transactions on Software Engineering*, 35:484–496.

- [Heagerty and Zheng, 2004] Heagerty, P. and Zheng, Y. (2004). Survival Model Predictive Accuracy and ROC Curves. UW Biostatistics Working Paper Series 1051, Berkeley Electronic Press.
- [IEEE, 1999] IEEE (1999). *Standard Glossary of Software Engineering Terminology 610.12-1990*. IEEE Press.
- [Jiang et al., 2008] Jiang, Y., Cukic, B., and Menzies, T. (2008). Can Data Transformation Help in the Detection of Fault-prone Modules? In *Proceedings of the 2008 Workshop on Defects in Large Software Systems, DEFECTS '08*, pages 16–20, Seattle, Washington. ACM.
- [Kim et al., 2011] Kim, M., Cai, D., and Kim, S. (2011). An Empirical Investigation into the Role of API-level Refactorings During Software Evolution. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 151–160, Waikiki, Honolulu, HI, USA.
- [Krishnan et al., 2011a] Krishnan, S., Lutz, R., and Goševa-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*, pages 103–112, Waikiki, Honolulu, HI, USA. ACM.
- [Krishnan et al., 2012a] Krishnan, S., Strasburg, C., Lutz, R., Goseva-Popstojanova, K., and Dorman, K. (2012a). Predicting Failure-Proneness in an Evolving Software Product Line. *Journal of Information and Software Technology*.
- [Krishnan et al., 2011b] Krishnan, S., Strasburg, C., Lutz, R. R., and 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*, pages 7:1–7:10, Banff, Alberta, Canada. ACM.

- [Krishnan et al., 2012b] Krishnan, S., Strasburg, C., Lutz, R. R., Goseva-Popstojanova, K., and Dorman, K. (2012b). Eclipse Product Line Prediction Data. <http://www.cs.iastate.edu/~lss/EclipsePLPredictionData.tar.gz>.
- [Lessmann et al., 2008] Lessmann, S., Baesens, B., Mues, C., and Pietsch, S. (2008). Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering*, 34(4):485–496.
- [Lutz and Mikulski, 2004] Lutz, R. R. and Mikulski, I. C. (2004). Empirical Analysis of Safety-Critical Anomalies During Operations. *IEEE Transactions on Software Engineering*, 30:172–180.
- [Mansfield, 2001] Mansfield, D. (2001). CVSPs-patchsets for CVS. <http://www.cobite.com/cvsps/>.
- [Menzies et al., 2011] Menzies, T., Butcher, A., Marcus, A., Zimmermann, T., and Cok, D. R. (2011). Local vs. Global Models for Effort Estimation and Defect Prediction. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11*, pages 343–351, Lawrence, Kansas, USA.
- [Menzies et al., 2012] Menzies, T., Caglayan, B., Kocaguneli, E., Krall, J., Peters, F., and Turhan, B. (2012). The PROMISE repository of empirical software engineering data. <http://promisedata.googlecode.com>.
- [Menzies et al., 2007] Menzies, T., Greenwald, J., and Frank, A. (2007). Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13.
- [Menzies et al., 2010] Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., and Bener, A. (2010). Defect Prediction from Static Code Features: Current Results, Limitations, New Approaches. *Journal of Automated Software Engineering*, 17.

- [Menzies et al., 2008] Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B., and Jiang, Y. (2008). Implications of Ceiling Effects in Defect Predictors. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, PROMISE '08, pages 47–54, New York, NY, USA. ACM.
- [Mockus et al., 2000] Mockus, A., Fielding, R. T., and Herbsleb, J. (2000). A Case Study of Open Source Software Development: The Apache Server. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, pages 263–272. ACM Press.
- [Mohagheghi and Conradi, 2008] Mohagheghi, P. and Conradi, R. (2008). An Empirical Investigation of Software Reuse Benefits in a Large Telecom Product. *ACM Transactions on Software Engineering and Methodology*, 17:13:1–13:31.
- [Mohagheghi et al., 2004] Mohagheghi, P., Conradi, R., Killi, O. M., and Schwarz, H. (2004). An Empirical Study of Software Reuse vs. Defect-Density and Stability. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 282–292, Washington, DC, USA. IEEE Computer Society.
- [Moser et al., 2008a] Moser, R., Pedrycz, W., and Succi, G. (2008a). Analysis of the Reliability of a Subset of Change Metrics for Defect Prediction. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 309–311, Kaiserslautern, Germany.
- [Moser et al., 2008b] Moser, R., Pedrycz, W., and Succi, G. (2008b). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, Leipzig, Germany. ACM.
- [Nagappan and Ball, 2005] Nagappan, N. and Ball, T. (2005). Static Analysis Tools as Early Indicators of Pre-release Defect Density. In *Proceedings of the 27th International*

Conference on Software Engineering, ICSE '05, pages 580–586, St. Louis, MO, USA. ACM.

[Nagappan et al., 2006a] Nagappan, N., Ball, T., and Murphy, B. (2006a). Using Historical In-Process and Product Metrics for Early Estimation of Software Failures. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, ISSRE '06, pages 62–74, Washington, DC, USA. IEEE Computer Society.

[Nagappan et al., 2006b] Nagappan, N., Ball, T., and Zeller, A. (2006b). Mining Metrics to Predict Component Failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 452–461, Shanghai, China. ACM.

[Nagappan et al., 2006c] Nagappan, N., Ball, T., and Zeller, A. (2006c). Mining Metrics to Predict Component Failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 452–461, Shanghai, China.

[Nagappan et al., 2010] Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., and Murphy, B. (2010). Change Bursts as Defect Predictors. In *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 309–318, San Jose, California, USA. IEEE Computer Society.

[Nguyen et al., 2011] Nguyen, T. T., Nguyen, T. N., and Phuong, T. M. (2011). Topic-based Defect Prediction (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 932–935, New York, NY, USA. ACM.

[Ostrand et al., 2005] Ostrand, T. J., Weyuker, E. J., and Bell, R. M. (2005). Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering*, 31:340–355.

- [Paulson et al., 2004] Paulson, J. W., Succi, G., and Eberlein, A. (2004). An Empirical Study of Open-Source and Closed-Source Software Products. *IEEE Transactions on Software Engineering*, 30:246–256.
- [Pinheiro et al., 2001] Pinheiro, J., Bates, D., DebRoy, S., Sarkar, D., and R Development Core Team (2001). *NLME: Linear and Nonlinear Mixed Effects Models*. R package version 3.1-103.
- [Pohl et al., 2005] Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlag, Berlin.
- [Rahmani et al., 2010] Rahmani, C., Azadmanesh, A., and Najjar, L. (2010). A Comparative Analysis of Open Source Software Reliability. *Journal of Software*, 5:1384–1394.
- [Schröter et al., 2006] Schröter, A., Zimmermann, T., and Zeller, A. (2006). Predicting Component Failures at Design Time. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, ISESE '06*, pages 18–27, New York, NY, USA. ACM.
- [SEI, 1984] SEI (1984). Software Engineering Institute, Software Product Lines. <http://www.sei.cmu.edu/productlines/>.
- [Shihab et al., 2010] Shihab, E., Jiang, Z. M., Ibrahim, W. M., Adams, B., and Hassan, A. E. (2010). Understanding the Impact of Code and Process Metrics on Post-release Defects: A Case Study on the Eclipse Project. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '10*, pages 4:1–4:10, Bolzano-Bozen, Italy. ACM.
- [Shivaji et al., 2009] Shivaji, S., Whitehead, J., Akella, R., and Kim, S. (2009). Reducing Features to Improve Bug Prediction. In *Proceedings of the 2009 IEEE/ACM Inter-*

national Conference on Automated Software Engineering, ASE '09, pages 600–604, Washington, DC, USA.

[Stephenson, 2002] Stephenson, Z. (2002). *Change Management in Families of Safety-Critical Embedded Systems*. PhD thesis, University of York.

[Tang et al., 1999] Tang, M.-H., Kao, M.-H., and Chen, M.-H. (1999). An Empirical Study on Object-Oriented Metrics. In *Proceedings of the 6th International Symposium on Software Metrics*, METRICS '09, pages 242–249. IEEE.

[van der Linden, 2009] van der Linden, F. (2009). Applying Open Source Software Principles in Product Lines. *UPGRADE, The European Journal for the Informatics Professional*.

[Wang et al., 2011] Wang, H., Khoshgoftaar, T. M., and Wald, R. (2011). Measuring Robustness of Feature Selection Techniques on Software Engineering Datasets. In *Proceedings of the IEEE International Conference on Information Reuse and Integration*, IRI '11, pages 309–314, Las Vegas, Nevada, USA. IEEE.

[Weiss, 2013] Weiss, D. (2013). Comments on thesis. Personal Conversation.

[Weiss and Lai, 1999] Weiss, D. M. and Lai, C. T. R. (1999). *Software Product-Line Engineering: A Family-based Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Weiss et al., 2008] Weiss, D. M., Li, J. J., Slye, J. H., Dinh-Trong, T. T., and Sun, H. (2008). Decision-Model-Based Code Generation for SPLE. In *Proceedings of the 12th International Conference on Software Product Lines*, SPLC '08, pages 129–138, Limerick, Ireland. IEEE Computer Society.

[Wikipedia, 2003] Wikipedia (2003). Receiver Operating Characteristic—Wikipedia, The Free Encyclopedia.

- [Zhang, 2008] Zhang, H. (2008). An Initial Study of the Growth of Eclipse Defects. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 141–144, New York, NY, USA. ACM.
- [Zhou and Leung, 2006] Zhou, Y. and Leung, H. (2006). Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults. *IEEE Transactions on Software Engineering*, 32:771–789.
- [Zimmermann et al., 2009] Zimmermann, T., Nagappan, N., Gall, H., Giger, E., and 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, pages 91–100, Amsterdam, The Netherlands. ACM.
- [Zimmermann et al., 2008] Zimmermann, T., Nagappan, N., and Zeller, A. (2008). *Predicting Bugs from History*, pages 69–88. Springer.
- [Zimmermann et al., 2007] Zimmermann, T., Premraj, R., and Zeller, A. (2007). Predicting Defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–15, Minneapolis, MN, USA.