AN EXPLORATION OF CHALLENGES LIMITING PRAGMATIC SOFTWARE DEFECT PREDICTION

by

EMAD SHIHAB

A thesis submitted to the

School of Computing

in conformity with the requirements for

the degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

August 2012

Abstract

Software systems continue to play an increasingly important role in our daily lives, making the quality of software systems an extremely important issue. Therefore, a significant amount of recent research focused on the prioritization of software quality assurance efforts. One line of work that has been receiving an increasing amount of attention is Software Defect Prediction (SDP), where predictions are made to determine where future defects might appear. Our survey showed that in the past decade, more than 100 papers were published on SDP. Nevertheless, the adoption of SDP in practice to date is limited.

In this thesis, we survey the state-of-the-art in SDP in order to identify the challenges that hinder the adoption of SDP in practice. These challenges include the fact that the majority of SDP research rarely considers the impact of defects when performing their predictions, seldom provides guidance on how to use the SDP results, and is too reactive and defect-centric in nature.

We propose approaches that tackle these challenges. First, we present approaches that predict high-impact defects. Our approaches illustrate how SDP research can be tailored to consider the impact of defects when making their predictions. Second, we present approaches that simplify SDP models so they can be easily understood and illustrates how these simple models can be used to assist practitioners in prioritizing the creation of unit tests in large software systems. These approaches illustrate how SDP research can provide guidance to practitioners using SDP.

Then, we argue that organizations are interested in proactive risk management, which covers more than just defects. For example, risky changes may not introduce defects but they could delay the release of projects. Therefore, we present an approach that predicts risky changes, illustrating how SDP can be more encompassing (i.e., by predicting risk, not only defects) and proactive (i.e., by predicting changes before they are incorporated into the code base).

The presented approaches are empirically validated using data from several large open source and commercial software systems. The presented research highlights how challenges of pragmatic SDP can be tackled, making SDP research more beneficial and applicable in practice.

Acknowledgments

First, I thank Allah for blessing me this wonderful opportunity to pursue my PhD.

I would like to thank my supervisor Dr. Ahmed E. Hassan for all his guidance and support throughout this journey. Ahmed, I cannot thank you enough for all you have done. You have been an advisor, a critic, a supporter, a friend and most importantly a wonderful human being. I am forever indebted to you. I would also like to thank Dr. Bram Adams who's patience and endless efforts have made me a much better researcher. A special thank you to my supervisory committee members, Prof. Hossam Hassanein and Prof. Patrick Martin, for their continued support and guidance. Many thanks to my examiners Prof. Lionel Briand and Dr. Thomas Dean for their fruitful feedback on my work.

I was very lucky to work and collaborate with some of the brightest researchers during my PhD. I would like to thank all of my lab mates and collaborators, Jack Jiang, Nicolas Bettenburg, Walid Ibrahim, Ian Shang, Thanh Nguyen, Dr. Yasutaka Kamei, Dr. Meiappan Nagappan, Dr. Audris Mockus, Dr. Christian Bird, Dr. Thomas Zimmermann and everyone else for the many fruitful discussions and collaborations. I learned so much from you all.

Through this entire journey I received so much love, guidance and support from many dear friends. I would like to thank Murad Hassan, Samer Fahmy, Marwan Dirani, Ramzi Atriss, Ziad Shahin, Yasser Abu Zaid, Amer Abdo, Khoder Shamy and Ahmad Dhaini for being such wonderful friends. You made my time during this PhD so enjoyable.

The school of computing staff and administration have made my time here at Queen's a very pleasant experience, thank you. Special thanks to Dr. Selim Akl, Dr. James Cordy, and Debby Robertson for helping and providing me with valuable advice throughout this journey.

To my mom, dad, and my brothers - thank you so much for your support through this journey. You have shown me what true sacrifice is. I would not have been here without your unconditional sacrifice, love, support, encouragement and guidance. I am so blessed to have such a wonderful family. To my wife and daughter, thank you for being there every step of the way. You made my days bright (even the really tough ones) and filled every second of my time with happiness and joy. I dedicate this thesis to you.

Dedication

To my mom, dad, Essam, Ayman, Rima and Aleena

Related Publications

The following publications are related to this thesis:

1. **Emad Shihab**. Pragmatic Prioritization of Software Quality Assurance Efforts. In *ICSE'11: International Conference on Software Engineering, Doctoral Symposium*, 4 pages, 2011. ($Acceptance\ Rate$: 22.6%) [Chapter 1]

2. **Emad Shihab**, Audris Mockus, Yasutaka Kamei, Bram Adams and Ahmed E. Hassan. High-Impact Defects: A Study of Breakage and Surprise Defects. In *ESEC/FSE'11: the 8th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 11 pages, 2011. ($Acceptance\ Rate$: 16.7%) [Chapter 3]

3. **Emad Shihab**, Akinori Ihara, Yasutaka Kamei, Walid Ibrahim, Masao Ohira, Bram Adams, Ahmed E. Hassan, and Kenichi Matsumoto. Predicting Re-opened Bugs: A Case Study on the Eclipse Project. In *WCRE'10: Working Conference on Reverse Engineering*, 10 pages, 2010. ($Acceptance\ Rate$: 31.3%) [Chapter 4]

4. **Emad Shihab**, Akinori Ihara, Yasutaka Kamei, Walid Ibrahim, Masao Ohira, Bram Adams, Ahmed E. Hassan, and Kenichi Matsumoto. Studying Re-opened Bugs in Open Source Software. *Empirical Software Engineering*, 23 pages, *Submitted May 2011* [Chapter 4]

5. **Emad Shihab**, ZhenMing Jiang, Walid Ibrahim, Bram Adams, and Ahmed E. Hassan. Understanding the Impact of Code and Process Metrics on Post-release Defects: A Case Study on the Eclipse Project. In *ESEM'10: International Symposium on Empirical Software Engineering and Measurement*, 10 pages, 2010. ($Acceptance\ Rate$: 29.4%) [Chapter 5]

6. **Emad Shihab**, ZhenMing Jiang, Bram Adams, Ahmed E. Hassan and Robert Bowerman. Prioritizing Unit Test Creation for Test-Driven Maintenance of Legacy Systems. In *QSIC'10: International Conference on Quality Software*, 10 pages, 2010. ($Acceptance\ Rate$: 16.5%) [Chapter 6]

7. **Emad Shihab**, ZhenMing Jiang, Bram Adams, Ahmed E. Hassan and Robert Bowerman. Prioritizing the Creation of Unit Tests in Legacy Software Systems. In *Software: Practice and Experience*, vol. 41:10, 22 pages, Aug. 2010 [Chapter 6]

8. **Emad Shihab**, Ahmed E. Hassan, ZhenMing Jiang and Bram Adams. An Industrial Study on the Risk of Software Changes. *FSE'12: International Symposium on the Foundations of Software Engineering*, 11 pages, *Accepted June 2012* [Chapter 7]

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software systems are becoming increasingly complex and are being used in everything from mobile devices to space shuttles. The increasing importance and complexity of software systems in our daily lives makes their quality a critical, yet extremely difficult issue to address. The US National Institute of Standards and Technology (NIST) estimated that software faults and failures cost the US economy $59.5 billion a year [3]. Other studies show that an average Fortune 100 company maintains 35 million lines of code and that this amount of maintained code is expected to double every 7 years [206]. Software Quality Assurance (SQA), i.e., the set of activities that ensure software meets a specific quality level, is one area that takes up a large amount of this maintenance effort [115].

A significant amount of recent research has focused on the prioritization of SQA efforts. One line of work that has been receiving increasing amounts of attention recently is Software Defect Prediction (SDP), where code and/or repository data (i.e., recorded data about the development process) is used to predict where defects might appear in the future (e.g., [204, 310]). In fact our literature review in Chapter 2 shows that in the past decade more than 100 papers were published on SDP alone.

Nevertheless, the adoption of software engineering research, especially SDP, in practice has been a challenge [102, 115]. For example, prior efforts in Adoption Centric Software Enigneering (ACSE) attempted to increase the adoption of software engineering tools and processes in practice [27, 28, 268]. Lethbridge [172] and Favre *et al.* [86] investigated the costs, benefits and risks of tool adoption in practice.

We hypothesize that the limited adoption of SDP is attributed to the fact that most SDP studies are not designed with a pragmatic view in mind [186]. This hypothesis is supported through numerous discussions with software engineering practitioners from a large software company where I spent 1.5 years as a SQA specialist and one year as an embedded SQA researcher. Based on an extensive literature review of SDP research in the past decade, we feel that the following challenges play a key role in the limited adoption of SDP in practice:

1. **Rarely consider the impact of defects**: SDP research rarely considers the impact of defects when providing recommendations of software locations that should be addressed [62, 202]. This makes the SDP approaches less effective since, for example, a documentation defects tend to have far less impact than security defects.

2. **Seldom provide guidance for use of results in practice**: Very few SDP studies focus on what to do once the predictions are made. Practitioners are left with no guidance on how to make use of SDP results [115, 186].

In addition, the majority of SDP approaches are reactive in nature and only focus on predicting defects, i.e., they assume that defects are already in the code and flag code that these defects might exist in. However, organizations are interested in managing risk, which covers more than just defects. For example, risky changes may not introduce defects but they could delay the release of projects, and/or negatively impact customer satisfaction. At the same time, it would be ideal to proactively flag risky code and address it before is injected into the code base. We believe that proactive approaches that predict risky changes are needed.

1.1 Research Hypothesis

Prior research and our informal industrial experience lead us to the formation of our research hypothesis. We believe that:

The adoption of SDP research in practice remains limited. We hypothesize that this low adoption is due to the fact that impact of defects is rarely considered and practitioners are seldom provided guidance as to how to make use of such research in practice. We believe that approaches that consider impact and provide guidance to practitioners are urgently needed in practice. Moreover, we believe that the focus on defect removal must be re-examined. SDP work should be more proactive and focus on risk prevention, rather than simply defect removal.

The goal of this thesis is to propose approaches that demonstrate how prior SDP research can be tailored to deal with the aforementioned challenges (i.e., considering the impact of defects and providing guidance on how to use the results), making SDP more pragmatic. The thesis is divided into three parts, two parts focus on each of the two aforementioned challenges. The third Part proposes an approach that demonstrates how SDP research can be more encompassing and proactive.

1.2 Thesis Organization

First, the thesis provides background on SDP and surveys the state-of-the-art in SDP (Chapter 2). Based on our survey, we highlight some of the challenges of SDP work. The remainder of the thesis is divided into three parts. Each Part focuses on tackling a specific challenge of SDP.

- **Part I:** Addresses the challenge of **impact of defects not being considered**. We present approaches that predict high impacting defects. We consider three possible definitions of high-impact defects: breakages (defects that occur in functionality that customers are used to), surprises (defects that occur in locations where practitioners did not expect) and re-opened defects (defects that have to be fixed more than once). Our work illustrates how SDP approaches can tailored to consider the impact of defects. [Chapters 3 and 4]

- **Part II:** Addresses the challenge of **how to make use of SDP results**. We present an approach that is used to simplify prediction models so they can be easily understood. In addition, we present an approach that uses the development history to prioritize the creation of unit tests in large software systems, i.e., which functions/methods should have unit tests created for them. Our work illustrates how to make SDP results more applicable in practical settings. [Chapters 5 and 6]

- **Part III:** Addresses the challenge of **making SDP approaches more encompassing and proactive**. We present an approach to identify risky code changes, i.e., changes that require additional attention through careful code/design review and possibly more testing. Our work illustrates how SDP approaches can be more encompassing and proactive. [Chapter 7]

To make each Part self contained, some repetition may exist between the various parts to facilitate their independent reading. Nevertheless, we tried to keep repetition to a minimum. The related work for each Chapter is examined and studied in the corresponding chapters of the thesis.

1.3 Thesis Overview

We now give an overview of the work presented in this thesis. When evaluating the different approaches presented in this thesis, we follow an empirical approach which requires access to historical data. Ideally, we would like to evaluate each approach on both, data from commercial and data from open source systems. However, this can be extremely difficult since some systems (e.g., commercial projects) have specific data that open source systems might not have. On the other hand, open source systems are more likely to share their data, whereas, data for commercial systems can be difficult to obtain (for confidentiality reasons). Therefore, we did our best to evaluate each approach presented in the thesis on as many projects as possible, however, some approaches are evaluated only on commercial systems while other approaches are evaluated only on open source systems.

1.3.1 Chapter 2: Background and State of the Art

Chapter 2 details the SDP process and presents a systematic review of SDP research from the year 2000 till the year 2011. The review characterizes more than 100 papers on SDP along four different dimensions. In particular, we discuss the data, factors, models, and performance evaluation criteria used to evaluate the SDP models. The Chapter concludes with a critical evaluation and discussion of the challenges of the surveyed SDP research.

1.3.2 Part I: Considering the Impact of Defects

A large body of prior work focuses on predicting post-release defects in open source and commercial systems [78,204,211,305,310]. One of the main reasons for the limited adoption of prior SDP research in practice is that even though they show promising accuracy results, all defects are considered to have the same negative impact. This is not realistic, because, for example, documentation defects tend to have far less impact than security defects. Therefore, we believe that there is a need for SDP approaches to consider impact when making their predictions [62, 202]. In this part, we present approaches that focuses on predicting the highest-impacting defects. Since impact has a different meaning for different stakeholders, we consider three possible definitions of high-impacting defects: *breakage defects*, *surprise defects* and *re-opened defects*.

Chapter 3: Studying and Predicting Breakage and Surprise Defects

The relationship between various software-related phenomena (e.g., code complexity) and post-release software defects has been thoroughly examined [78,204,211,305,310]. However, to date these predictions have a limited adoption in practice. The most commonly cited reason is that the prediction identifies too much code to review without distinguishing the impact of these defects. In this chapter, we aim to address this challenge by focusing on *high-impact defects* for customers and practitioners. *Customers* are highly impacted by defects that break pre-existing functionality (breakage defects), whereas *practitioners* are caught off-guard by defects in files that had relatively few pre-release changes (surprise defects).

We perform an empirical study on a large commercial software system to study and predict high-impact defects. We present models that can effectively identify files containing breakage and surprise defects. In addition, we perform analysis to identify and quantify the effect of the various factors on the likelihood of a file containing a breakage or surprise defect.
Systems evaluated on: Commercial telecommunication system.

Chapter 4: Studying and Predicting Re-opened Defects

Defect fixing accounts for a large amount of the software maintenance resources. Generally, defects are reported, fixed, verified and closed. However, in some cases defects have to be re-opened. Re-opened defects increase maintenance costs, degrade the overall user-perceived quality of the software and lead to unnecessary rework by practitioners.

We study and predict re-opened defects through a case study on three large open source projects – namely Eclipse, Apache and OpenOffice. We build prediction models that effectively predict re-opened defects. Then, we analyze the prediction models to determine which factors are the most important indicators of whether or not a defect will be re-opened.
Systems evaluated on: Eclipse, Apache and OpenOffice.

1.3.3 Part II: Providing Guidance on How to Use Results

Most SDP research today provides black-box type of models, i.e., a list of defect-prone software locations is given without any explanation as to why. This makes it difficult to understand why these models are making their predictions. To make prediction models easier to understand, we present an approach that simplifies prediction models. In addition, we present an approach to prioritize the creation of unit tests in large software systems (i.e., which parts of the code we should write unit tests for) to show how SDP results can be applied in practice (i.e., applying SDP to determine the most defect-prone functions so they can have unit tests created for them).

Chapter 5: Simplifying and Understanding SDP Models

Research studying the quality of software applications continues to grow rapidly with researchers building regression models that combine a large number of factors. However, these prediction models are hard to deploy in practice due to the cost associated with collecting all the needed factors, the complexity of the models and the black box nature of the models. For example, techniques such as Principle Component Analysis (PCA) are commonly used to merge a large number of factors into composite factors that are no longer easy to explain.

We use a statistical approach recently proposed by Cataldo *et al.* to create and operationalize explainable regression models. In addition, we show that our approach is able to quantify the impact of the used factors in a prediction model on the likelihood of finding post-release defects. Finally, we demonstrate that our simple models achieve comparable performance over more complex PCA-based models while providing practitioners with intuitive explanations on how to make use of the results.

Systems evaluated on: Eclipse.

Chapter 6: Prioritizing the Creation of Unit Tests

One major challenge of SDP research is that it does not provide any guidance on how to make use of their results. We believe that this challenge is due to the fact that this SDP work is not designed with a specific scenario in mind.

In this chapter, we use factors extracted from the development history of software projects to build simple SDP models that prioritize the creation of unit tests. Our approach is different from traditional SDP studies in that it performs its predictions at the function level and it takes into consideration the effort required to create the unit tests. This approach illustrates how software development and testing managers can leverage SDP to efficiently allocation their limited SQA resources.

Systems evaluated on: Commercial system and Eclipse.

1.3.4 Part III: Making SDP More Encompassing and Proactive

The majority of SDP research focuses on predicting defects and is reactive in nature, i.e., it assumes that the defects already exist in the code, and aim to identify the code that contains these defects [19, 30, 105, 109, 116, 122, 143, 171, 204, 209, 222, 263, 295, 310]. However, we believe that organizations are interested in more than just defects, they are interested in managing risk. Risk is more encompassing than defects. In this part, we present an approach that leverages historical data about software changes in order to predict the risk of a software change.

Chapter 7: Studying and Predicting the Risk of Software Changes

Modelling and understanding defects has been the focus of much of the Software Engineering research today. However, organizations are interested in more than just defects. In particular, they are more concerned about managing risk, i.e., the likelihood that a code or design change will cause a negative impact on their products and processes, regardless of whether or not it introduces a defect.

We conduct a study to predict and better understand risky changes, i.e., changes for which developers believe that additional attention is needed in the form of careful code/design reviewing and/or more testing. Our findings and models are being used today by an industrial partner to manage the risk of their software projects.
Systems evaluated on: Commercial mobile system

1.4 Thesis Contributions

The major contributions of this thesis are as follows:

- An extensive review of the state of the art in SDP. Such a review is of paramount importance at this time since a large amount (more than 100 papers according to our review) of research related to SDP has been done in the last decade, making it a good time to reflect on what has been addressed and what remains as an open issue in the field today. This survey provides an empirical foundation and motivation for our work in this thesis.

- The development of an approach to predict and better understand high-impact defects. This approach can be followed by other researchers to tailor their SDP approaches so they can focus on high-impact defects. We believe that such an approach is more applicable than the state-of-the-art in SDP today, since impact is taken into consideration when making predictions.

- The development of an approach to simplify SDP models by reducing the number of used factors. This approach shows how SDP research can be simplified, making it easier to understand and use. The presented results show that our approach can significantly simplify SDP models and that these simple models are able to achieve comparable performance to models that are much more complex.

- The development of an approach to guide the prioritization of unit test creation. This approach shows how simple SDP models can be used to prioritize the creation of unit tests. The presented results show that our approach outperforms ad-hoc methods used by practitioners today.
- The proposal of an approach that makes SDP more encompassing and proactive. Our approach identifies potentially risky code changes before they are incorporated into the code base.

Chapter 2

Background and State of the Art

Software Defect Prediction (SDP) involves the identification of software locations that quality assurance efforts should focus on. In the past decade, a plethora of research has focused on SDP. Each of these works used its own unique data, factors, modeling techniques and evaluated their models differently. In this chapter, we survey the state-of-the-art in SDP research in the past decade in order to understand the main findings and challenges. The Chapter provides background about the SDP process, summarizes the state-of-the-art in SDP and sheds light on some of the challenges in the area. Experts in the field, who are familiar with SDP research may wish to skip this chapter.

2.1 Introduction

Software Quality Assurance (SQA) is the set of activities that ensure that a software system meets a specific quality level. Organizations are always interested in ways to gauge the quality of their software before it is released [88]. To facilitate these organizational interests, researchers have proposed a slew of quality measures and built statistical models, that leverage these measures, to predict defect-prone areas of a software [22]. For example, some prior work focused on predicting files that contain one or more post-release [310] or security defects [309]. The line of research concerned with building such prediction models is called **Software Defect Prediction (SDP)** and will be the focus of this chapter.

More precisely, SDP can be defined as the identification of software locations (i.e., subsystems, files or functions) that quality assurance efforts should focus on (i.e., review or test). Incorporating SDP in the development process helps practitioners in the decision-making process by indicating which locations have a high chance (i.e., high predicted probability) of having a defect. Then, the limited validation and verification efforts can be focused on these locations.

It is important to note that there exist many different approaches to achieve high software quality. SDP is *one* approach, it is certainly not the only approach. For example, a large amount of research work uses model checking (e.g., [128]) and static analysis (e.g., Coverity [2]) to find defects in software systems. Other work focuses on fault localization, in which differences between the inputs of failing and passing tests are used to locate errors in the source code. The main difference between these other lines of research and SDP is that they identify defects in the current code base (i.e., the code base being analyzed). SDP warns about future defect-prone areas.

In this chapter, we survey papers related to SDP from the year 2000 to 2011. The papers are characterized and summarized along five dimensions: data, factors, models, performance evaluation and other considerations. That is, papers that perform research related to the used data in SDP are grouped together, papers related to factors in SDP studies are grouped together and so forth. To select the papers for this review, we searched for papers (search queries are listed in Table A.2) related to SDP in the top software engineering venues, listed in Table A.1, located in Appendix A at the end of the thesis. After searching for the SDP papers, we read through the titles and abstracts of each paper to narrow down the list of papers to read in more detail. In some cases, after reading the papers, we found that their focus is not on SDP, so we did not include them in the survey. The list of the rejected papers, along with the reason for rejecting them is provided in Appendix A as well.

2.1.1 Organization of Chapter

The rest of the Chapter is organized as follows: Section 2.2 presents a background on SDP and the current research trends of the surveyed SDP papers. Section 2.3 discusses the challenges of current SDP research and highlights the challenges that our thesis tackles. Section 2.4 concludes the chapter.

2.2 Background: Software Defect Prediction

Most SDP studies aim to correctly classify software artifacts (e.g., subsystems or files) as being fault-prone or not. Other SDP studies are interested in predicting the number of defects that may appear in a software artifact, so they can be ranked. Figure 2.1 shows an overview of the SDP process. First, project data is collected from software repositories (e.g., defect and source control repositories). Then, factors are calculated from the data. Statistical and machine learning models are built to predict the locations that have a high potential of containing defects. Finally, the prediction models are evaluated using various measures, such as precision, recall and explanative power.

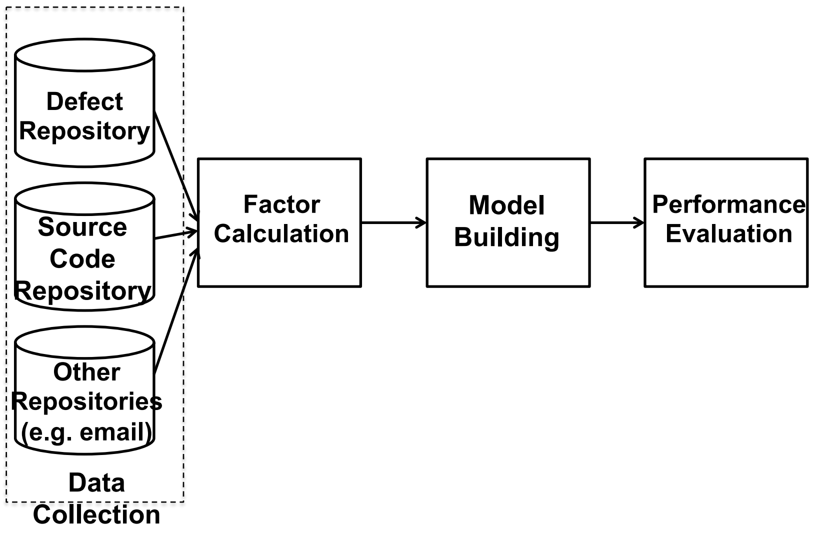


Figure 2.1: Overview of software defect prediction (SDP)

There has been an extensive body of work that focused on SDP. Each of these works used its own unique data, dependent and independent variables, modelling techniques and evaluated their models differently. Therefore, there is a need to compare and contrast the prior work in order to better understand the assumptions and implications of the work. In this chapter, we survey the prior research on SDP, and characterize and compare this literature along the following dimensions:

- **Data sources and granularity**: We report on the sources and the granularity of the data used in prior SDP reserch.
- **Factors**: We report on the factors used in SDP studies.
- **Models**: We report on the models used in SDP studies.
- **Performance evaluation**: We report on the different performance evaluation methods used to evaluate the SDP models.

Next, we provide a background on each of the aforementioned dimensions and report the state-of-the-art in SDP research today along these dimensions. Table 2.1 lists the criteria that make up each dimension and their explanation.

In addition to studying the research trends, we provide a summary of the surveyed papers in Appendix B at the end of the thesis.

Table 2.1: Dimensions and factors used to compare SDP studies

Dim.	Category	Factor	Description
Data sources and granularity	Repository	Source Code	Uses data from the source code repository
		Defect	Uses data from the defect repository
		Other	Uses data from other sources
	Project	Open Source	An open source project is used
		Commercial	A commercial project is used
		Project name	The name of the project used
		# releases	The number of releases used in the study
		Prog. languages	The programming language the project is written in
	Granularity	Subsystem	The predictions are made at the subsystem (e.g., directory) level
		File	The predictions are made at the file level
		Function	The predictions are made at the function level
		Other	The predictions are made at a level other than subsystem, file or function (e.g., project level).
Factors	Independent variables	Product	Product factors (e.g., code size) are used to predict
		Process	Process factors (e.g., churn) are used to predict
		Other	Factors other than product and process used to predict
		# of factors	The number of factors used in the prediction model
	Dependent Variable	Pre	Predictions are made to predict pre-release defects
		Post	Predictions are made to predict post-release defects
		Other	Predictions are made for a dependent variable other than pre- and post-release defects

Dim.	**Category**	**Factor**	**Description**
Types of models	Statistical	Naive Bayes	Naive Bays models are used
		MARS	A multivariate adaptive regression splines model is used
		Logistic regression	A logistic regression model is used
		Linear regression	A linear regression model is used
	Tree-based models	Decision trees	Decision tree models are used
		CART	A classification and regression trees model is used
		Random Forests	A Random Forests model is used
		Recursive Partitioning	A recursive partitioning model is used
		SVM	Support Vector Machines model is used
		Other	Other models are used
Performance evaluation	Cross validation	10-fold	10-fold cross validation is used
		Cross-project	Cross-project validation is performed
		Cross-release	Cross-release validation is performed
		Training and testing data	Data is plot into training and testing data
	Predictive power	Correlations	Correlations are used
		Precision	Precision is used
		Recall	Recall is used
		Accuracy	Accuracy is used
		F-Measure	F-measure is used
		AUC	Area under the ROC curve is used
		Other	Evaluation methods other than precision, recall, accuracy and f-measure are used
	Explanative power	R^2	R^2 is used
		Deviance Explained	Deviance explained is used

Dim.	Category	Factor	Description
Other considerations	Tools	WEKA	The WEKA tool is used
		R	The R tool is used
		Other	A tool other than R or WEKA is used
	Other statistical considerations	Consider collinearity	Whether collinearity is considered
		Control variables	Control variables are used
		Considers Effort	Effort is considered
		Effort to extract factors	Effort to extract factors is considered
		Considers Impact	The impact of the defect is considered
		Sought practitioner feedback	the work is discussed with practitioners

2.2.1 Data Sources and Granularity

Large software projects store their development history and other information, such as communication, in so called software repositories. The main motivation for using these repositories is to keep track of and record development history. However, researchers and practitioners realize that this repository data can be leveraged to uncover interesting and actionable information about software systems [115].

SDP work generally leverages various types of data from different repositories, such as:

- Source code/control repositories: stores and records the source code and development history of a project. The source code repository is used to rollback or extract older source code and meta-data about development changes.

- Defect report repositories: track the bug/defect reports or feature requests filed for a project and their resolution progress.

- Mailing list and other repositories (e.g., chat): track the communication and discussions between development teams. Mailing lists are most commonly used in open source software projects.

SDP generally uses repository data to extract factors and build prediction models. For example, prior work used the data stored in the source control repository to count the number of changes made to a file [310] and the complexity of changes [116], and used this data to predict files that are likely to have future defects. Other work uses data from the defect repositories to determine customer-reported (also referred to as post-release) defects.

SDP work uses repository data from a wide variety of projects (e.g., open source [116, 204, 310], commercial [42, 55, 124, 210] or both [215, 307]). This data is provided at different levels of granularity (e.g., at the subsystem [304], file [61] or function [155] level). In general, a subsystem is composed of a number of files and a file is made up of a number of functions.

Research trends: Table 2.2 shows the data sources and granularity used in the surveyed SDP papers. Each row of the table represents one of the surveyed papers. An '✓' means the criteria is applied by the paper, a '.' means the criteria was not applied by the paper and a '?' means we could not determine whether or not the criteria was applied. An 'NA' means the criteria is not applicable for that paper. We had to use our understanding and judgement to determine the difference between when a paper did not apply a criteria (i.e., '.') or we could not determine whether a criteria was used (i.e., '?'). To illustrate the difference, we provide the following example. If a paper mentions that one project is used and it is a commercial project, then we would determine that no open source project is used (and mark the open source column as '.' for that project). However, if the programming language of the project is not mentioned, then we would mark the programming language column as could not determine ('?'), since we know that every project must be written in some sort of programming language, but we could not determine what it was.

From Table 2.2, it is clear that repository data is important for SDP studies since 75% of the papers use source code repositories and 69% use a defect repository. A few studies use data from other repositories such as Mylyn interaction data and vulnerabilities database (e.g., [169, 188].

In terms of the type of projects used, the majority of SDP papers use commercial projects to perform their case studies. 69% of studies use data from commercial projects, whereas 37% of studies use data from Open Source Software (OSS) projects. Figure 2.2 plots the number of papers per year that use commercial and open source data. It is clear from the figure that most studies use commercial data, however, the number of paper using open source data is growing over time. We also find that 43% of studies use projects written in Java and 46% use C and C++ projects.

Another important consideration for SDP studies is the granularity at which the study is performed. We observe that 49% of studies perform their predictions at the subsystem level, 51% perform their prediction at the file level and 5% at the function level.

The percentages presented here do not add up to 100% since some studies may use multiple repositories, use both commercial and open source projects, and perform predictions at multiple levels of granularity. In such cases, we count the paper in multiple criteria (e.g., if multiple granularities are used we count the paper in all granularity levels).

Table 2.2: Data sources and granularities used in SDP studies. An '✓' means applied, a '.' means not applied and '?' means could not determine.

	Repository			**Project**					**Granularity**			
Paper	**Source Code**	**Defect**	**Other**	**Open Source**	**Commercial**	**Project Name**	**# releases**	**Prog. Language**	**Subsystem**	**File**	**Function**	**Other**
Yuan; 2000 [296]	✓	✓	.	.	✓	Telecom	?	? High level	✓	.	.	.
Cartwright; 2000 [53]	✓	✓	.	.	✓	Telecom	1	C++	✓	.	.	.
Neufelder; 2000 [216]	✓	✓	.	.	✓	17 commercial organizations	?	C++; 2 unkown	?	?	?	.

	Repository			Project					Granularity			
Paper	**Source Code**	**Defect**	**Other**	**Open Source**	**Commercial**	**Project Name**	**# releases**	**Prog. Language**	**Subsystem**	**File**	**Function**	**Other**
Khoshgoftaar; 2000 [147]	✓	✓	.	.	✓	Telecom	4	Protel	✓	.	.	.
Khoshgoftaar; 2000 [148]	✓	✓	.	.	✓	Telecom	4	Protel	✓	.	.	.
Morasca; 2000 [203]	✓	✓	.	.	✓	DATATRIEVE	2	BLISS	✓	.	.	.
Wong; 2000 [291]	✓	✓	.	.	✓	Telecordia; client-server system	2	C with embedded SQL	.	✓	✓	.
Fenton; 2000 [89]	✓	✓	.	.	✓	Telecom	2	?	✓	.	.	.
Graves; 2000 [105]	✓	.	.	.	✓	Telephone switching system	1	C	✓	.	.	.
Khoshgoftaar; 2001 [142]	✓	✓	.	.	✓	?	?	C++	.	✓	.	.
El Emam; 2001 [78]	?	✓	.	.	✓	Commercial word processor	2	Java	.	✓	.	.
Denaro; 2002 [66]	✓	✓	.	✓	.	Apache Web	2	C	.	✓	.	.
Briand; 2002 [50]	✓	✓	.	.	✓	Oracle Xpose and Jwriter	?	Java	.	✓	.	.
Khoshgoftaar;2002 [146]	✓	✓	.	.	✓	Telecom	2	Protel	✓	.	.	.
Quah; 2003 [232]	.	✓	Code	.	✓	QUES	?	?	.	✓	.	.
Khoshgoftaar; 2003 [144]	✓	✓	.	.	✓	Telecom	4	Protel	✓	.	.	.
Guo; 2003 [106]	?	?	Nasa data	.	✓	Nasa	?	C	✓	.	.	.
Amasaki; 2003 [12]	✓	✓	?	.	✓	Retail system	?	?	.	.	.	project
Succi; 2003 [264]	✓	.	.	.	✓	?	?	C++	.	✓	.	.
Guo; 2004 [107]	?	?	Nasa data	.	✓	Nasa	?	C; C++	✓	.	.	.
Li; 2004 [175]	?	✓	?	✓	✓	IBM OS and middleware; OpenBSD; Tomcat	22	?	?	?	?	.
Ostrand; 2004 [225]	✓	.	.	.	✓	AT&T inventory system	17	Java	.	✓	.	.
Koru;2005 [163]	?	?	Nasa data	.	✓	Nasa	?	C; C++	.	✓	.	.
Gyimothy; 2005 [109]	✓	✓	.	✓	.	Mozilla	7	C++	.	✓	.	.
Mockus; 2005 [202]	✓	✓	Customer info	.	✓	Telecom	?	C; C++	✓	.	.	.
Nagappan; 2005 [209]	✓	✓	.	.	✓	Windows Server 2003	?	C; C++	✓	.	.	.

Paper	Repository			Project					Granularity			
	Source Code	**Defect**	**Other**	**Open Source**	**Commercial**	**Project Name**	**# releases**	**Prog. Language**	**Subsystem**	**File**	**Function**	**Other**
Hassan; 2005 [117]	✓	✓	.	✓	.	NetBSD; FreeBSD; OpenBSD; KDE; Koffice; Postgres	?	C; C++	✓	.	.	.
Nagappan; 2005 [210]	✓	✓	.	.	✓	Windows Server 2003	?	C; C++	✓	.	.	.
Tomaszewski;2006 [270]	?	?	?	.	✓	Telecom	3	?; OO language	✓	✓	.	.
Pan; 2006 [227]	✓	.	.	✓	.	Apache HTTP and Latex2rtf	?	C; C++	.	✓	✓	.
Nagappan; 2006 [212]	✓	✓	.	.	✓	IE6; IIS W3 Server core; Process Messaging Component; DirectX; NetMeeting	?	C#; C++	✓	.	.	.
Zhou; 2006 [302]	?	?	?	.	✓	Nasa	?	C; C++	.	✓	.	.
Li; 2006 [173]	✓	✓	.	.	✓	ABB monitoring system and controller management	13(MS) and 15(CM)	? C++	✓	.	.	.
Knab; 2006 [159]	✓	✓	.	✓	.	Mozilla web browser	7	C; C++	.	✓	.	.
Arisholm; 2006 [19]	✓	✓	.	.	✓	Telecom	?	Java	.	✓	.	.
Tomaszewski;2007 [271]	?	?	?	.	✓	Telecom	3	?; OO language	✓	✓	.	.
Ma; 2007 [178]	?	?	?	.	✓	Nasa	?	C; C++	✓	.	.	.
Menzies; 2007 [190]	?	?	?	.	✓	Nasa	?	C; Java	✓	.	.	.
Olague; 2007 [223]	✓	✓	.	✓	.	Mozilla Rhino	6	Java	.	✓	.	.
Bernstein; 2007 [34]	✓	✓	.	✓	.	Eclipse	vary	Java	.	✓	.	.
Aversano; 2007 [23]	✓	.	.	✓	.	JHotDraw and DNSJava	?	Java	.	✓	.	.
Kim; 2007 [155]	✓	?	.	✓	.	Apache HTTP; Subversion; PostgreSQL; Mozilla; Jedit; Columba; Eclipse	?	C; C++; Java	.	✓	✓	.
Ratzinger; 2007 [238]	✓	✓	.	✓	✓	Health care; ArgoUML and Spring (OSS)	?	Java	.	✓	.	.

	Repository			**Project**					**Granularity**			
Paper	**Source Code**	**Defect**	**Other**	**Open Source**	**Commercial**	**Project Name**	**# releases**	**Prog. Language**	**Subsystem**	**File**	**Function**	**Other**
Mizuno; 2007 [195]	✓	✓	.	✓	.	ArgoUML and Eclipse BIRT	?	Java	.	✓	.	.
Weyuker; 2007 [283]	✓	✓	.	.	✓	AT&T inventory; provisioning and voice response	17 inv; 9 pro-visioning	?	.	✓	.	.
Zimmermann; 2007 [310]	✓	✓	.	✓	.	Eclipse	3	Java	✓	✓	.	.
Moser; 2008 [204]	✓	✓	.	✓	.	Eclipse	3	Java	.	✓	.	.
Kamei; 2008 [140]	✓	✓	.	✓	.	Eclipse	2	Java	?	?	?	.
Zimmermann; 2008 [304]	✓	✓	.	.	✓	Windows Server 2003	?	?	✓	.	.	.
Zimmermann; 2008 [305]	✓	✓	.	.	✓	Windows Server 2003	?	?	✓	.	.	.
Nagappan; 2008 [214]	✓	✓	.	.	✓	Windows Vista	?	?	✓	.	.	.
Zhang; 2008 [298]	.	✓	.	✓	.	Eclipse	?	Java	✓	.	.	.
Pinzger; 2008 [230]	✓	✓	.	.	✓	Windows Vista	?	?	✓	.	.	.
Lessmann; 2008 [170]	?	?	?	.	✓	Nasa	?	?	✓	.	.	.
Watanabe; 2008 [280]	✓	.	.	✓	.	jEdit and Sakura	6	C++; Java	.	✓	.	.
Kim; 2008 [153]	✓	✓	.	✓	.	Apache; Bugzilla; Columba; Gaim; Gforge; Jedit; Mozilla; Eclipse JDT; Plone; Post-greSQL; Scarab; Subversion	?	C; C++; Java; Perl; Python; JavaScript; PHP and XML	.	.	✓	.
Jiang; 2008 [131]	?	?	?	.	✓	Nasa	?	C; C++; Java; Perl	✓	.	.	.
Weyuker; 2008 [284]	✓	✓	.	.	✓	Business mainte-nance system	61	?	.	✓	.	.
Jiang; 2008 [133]	?	?	?	.	✓	Nasa	?	C; C++; Java; Perl	✓	.	.	.
Tosun; 2008 [273]	?	?	?	.	✓	Nasa	?	?	✓	.	.	.
Koru; 2008 [160]	✓	.	.	✓	✓	Koffice; ACE; IBM-DB	?	C++	.	✓	.	.
Menzies; 2008 [192]	?	?	?	.	✓	Nasa	?	C; C++; Java	✓	.	.	.
Layman; 2008 [168]	✓	✓	.	.	✓	?	?	C#; C++	✓	.	.	.

	Repository			Project					Granularity			
Paper	**Source Code**	**Defect**	**Other**	**Open Source**	**Commercial**	**Project Name**	**# releases**	**Prog. Language**	**Subsystem**	**File**	**Function**	**Other**
Goronda; 2008 [103]	?	?	?	.	✓	Nasa	?	C	✓	.	.	.
Vandecruys; 2008 [275]	?	?	?	.	✓	Nasa	?	?	✓	.	.	.
Elish; 2008 [77]	?	?	?	.	✓	Nasa	?	C; C++; Java	✓	.	.	.
Ratzinger; 2008 [239]	✓	?	.	✓	.	ArgoUML; Jboss Cache; Liferay Portal; Spring; Xdoclet	?	Java	.	✓	.	.
Meneely; 2008 [189]	?	?	.	.	✓	Nortel networking system	3	?	.	✓	.	.
Wu; 2008 [293]	✓	✓	.	.	✓	SoftPM	4	Java	✓	.	.	.
Tarvo; 2008 [266]	✓	✓	.	.	✓	Microsoft	?	?	.	.	.	Changes
Holschuh; 2009 [124]	✓	✓	.	.	✓	SAP	6 projects; 3 releases each	Java	✓	✓	.	.
Jia; 2009 [129]	✓	✓	.	✓	✓	Eclipse and QMP	6	C; Java	✓	✓	.	.
Shin; 2009 [257]	✓	✓	.	.	✓	Business maintenance system	35	C; C++	.	✓	.	.
Mende; 2009 [185]	✓	✓	.	.	✓	Alcatel-Lucent LambdaUnite	?	C;C++	.	✓	.	.
Binkley; 2009 [40]	✓	✓	.	✓	✓	Mozilla web browser and MP	1	C; C++; Java	✓	.	.	.
D'Ambros; 2009 [62]	✓	✓	✓	✓	.	ArgoUML; JDT Core; Mylyn	?	Java	.	✓	.	.
Hassan; 2009 [116]	✓	✓	.	✓	.	NetBSD; FreeBSD; OpenBSD; KDE; Koffice; Postgres	?	C; C++	✓	.	.	.
Bird; 2009 [43]	✓	✓	.	✓	✓	Vista and Eclipse	1 Vista; 6 Eclipse	C; C++; C#; Java	✓	.	.	.
Ferzund; 2009 [92]	✓	✓	.	✓	.	Apache; Epiphany; Evolution; Nautilus; PostgreSQL; Eclipse; Mozilla	?	C; C++; Java	.	.	.	Hunks

	Repository			**Project**					**Granularity**			
Paper	**Source Code**	**Defect**	**Other**	**Open Source**	**Commercial**	**Project Name**	**# releases**	**Prog. Language**	**Subsystem**	**File**	**Function**	**Other**
Liu; 2010 [176]	?	?	?	.	✓	Nasa	?	C; C++; Java	✓	.	.	.
Erika; 2010 [80]	?	?	?	?	?	ECS; BNS; CRS student projects	?	Java	.	✓	.	.
Zhou; 2010 [303]	✓	✓	.	✓	.	Eclipse	3	Java	.	✓	.	.
Mockus; 2010 [197]	✓	✓	.	.	✓	Avaya switching system	?	C; C++	.	✓	.	.
Kamei; 2010 [138]	✓	✓	.	✓	.	Eclipse Platform; JDT; PDE	9	Java	✓	✓	.	.
Meneely; 2010 [188]	✓	✓	Vulnerability DB	✓	.	Linux Kernal; PHP; Wireshark	?	C; C++	.	✓	.	.
Nguyen; 2010 [218]	✓	✓	.	✓	.	Eclipse	3	Java	✓	✓	.	.
Shihab; 2010 [254]	✓	✓	.	✓	.	Eclipse	3	Java	.	✓	.	.
Menzies; 2010 [191]	?	?	?	.	✓	NASA and Turkish commercial systems	?	C; C++; Java	✓	.	.	.
Weyuker; 2010 [286]	✓	✓	.	.	✓	AT&T inventory; provisioning and voice response	?	?	.	✓	.	.
Mende; 2010 [184]	✓	✓	.	✓	✓	NASA and Eclipse	3 for Eclipse	? Java	?o	.	.	.
Nugroho; 2010 [221]	✓	✓	.	.	✓	Healthcare system	?	Java	.	✓	.	.
Song; 2011 [261]	?	?	?	.	✓	Nasa	.	?	✓	.	.	.
Nguyen; 2011 [219]	✓	✓	.	✓	.	Eclipse JDT	1	Java	.	✓	.	.
Mende; 2011 [186]	✓	✓	.	.	✓	Avionics and Nasa	?	C; C++; Java; Perl	.	✓	✓	.
Lee; 2011 [169]	✓	✓	Mylyn data	✓	.	Eclipse Bugzilla	?	?	.	✓	.	Tasks
Shihab; 2011 [255]	✓	✓	.	.	✓	Avaya telephony project	5	C; C++	.	✓	.	.
D'Ambros; 2011 [61, 63]	✓	✓	.	✓	.	Eclipse JDT; PDE; Equinox framework; Mylyn; Apache Lucene	5	Java	✓	✓	.	.

	Repository			Project					Granularity			
Paper	**Source Code**	**Defect**	**Other**	**Open Source**	**Commercial**	**Project Name**	**# releases**	**Prog. Language**	**Subsystem**	**File**	**Function**	**Other**
Kpodjedo; 2011 [166]	✓	✓	.	✓	.	Rhino; ArgoUML; Eclipse	7 Rhino; 9 ArgoUML; 3 Eclipse	Java	.	✓	.	.
Bird; 2011 [44]	✓	✓	.	.	✓	Windows Vista and 7	?	?	✓	.	.	.
Meneely; 2011 [187]	✓	✓	.	.	✓	Cisco	.	C; C++; Java	.	✓	.	.
Giger; 2011 [101]	✓	✓	.	✓	.	Eclipse	?	Java	.	✓	.	.
Percentage of papers	**75**	**69**	**-**	**37**	**69**	**18 Nasa; 19 Eclipse**	**-**	**43 Java; 46 C;C++**	**49**	**51**	**5**	**-**

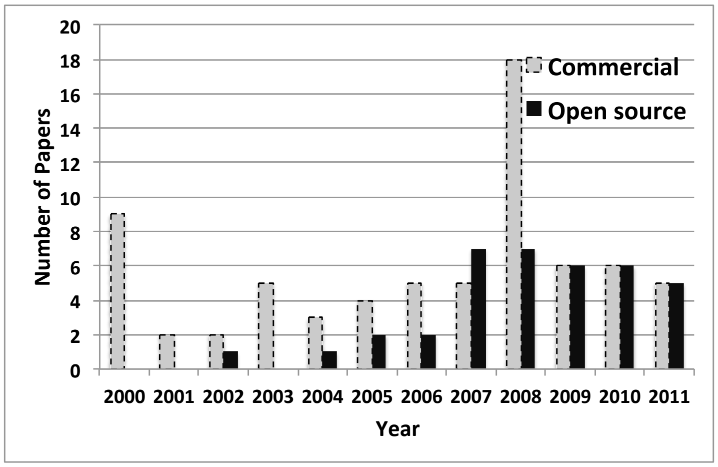


Figure 2.2: Percentage of SDP studies performed on commercial and open source projects

2.2.2 Factors

Generally speaking, factors are extracted using repository data. When used in SDP research, factors are considered to be independent variables, which means that they are used to perform the prediction (i.e., the predictors). Also, factors can represent the dependent variables, which means they are the factors being predicted (i.e., these can be pre- or post-release defects). Previous SDP work used a wide variety of independent variables (e.g., process [116], organizational [55,214] or code factors [162,190,211,263,310]) to perform their predictions. It also used different dependent variables. For example, previous work predicted for different types of defects (e.g., pre-release [210], post-release [301,310] or both [255,257]).

Product Factors

Product factors are factors that are directly related or derived from the source code (e.g., complexity or size). A large body of SDP work uses product factors to predict defects. The main idea behind using product factors is that, for example, complex code is more likely to have defects. Some of the most common complexity factors used in the literature are [310]:

- **Lines of Code (LOC)**: Measures the number of lines of code of a software artifact (i.e., file of package).
- **McCabe Cyclomatic Complexity [182]**: Measures the complexity of a program as the number of linearly independent paths through a program's source code.
- **Number of Methods**: Measures the number of methods in a software artifact.
- **Fan Out**: Measures the number of other software artifacts (e.g., class or method) referenced by a software artifact.
- **Fan In**: Measures the number of other software artifacts (e.g., class or method) referencing a software artifact.

- **Number of Parameters**: Measures the number of parameters passed in to a software artifact.
- **Number of Interfaces**: Measure the number of interfaces in a software artifact.
- **Number of Classes**: Measures the number of classes in a file or package.
- **Program Slicing [227]**: A function may contain multiple behavioural aspects that are captured by program slicing, such as statements that change a global variable or statements that compute the return value. This behavioural information is used to calculate program slicing factors such as the number of slices.
- **Dependency [266, 304]**: Factors that measure the directed relation between two pieces of code. These can be data dependencies or call dependencies.
- **Calling Structure [257]**: Factors that represent the invocation relationship between functions/files/subsystems of a software system.
- **Natural Language [40]**: Factors based on the occurrence of natural language, extracted from source code and comments.
- **Topic Models [219]**: Factors that build topics from source code.

There are other code factors that are used in defect prediction work such as, factors based on Halstead factors which are based on operators and operands, and the Chidamber and Kemerer [59] factors.

Process Factors

The idea behind using process factors in defect prediction is that the process used to develop the code may lead to defects. For example, if a piece of code is changed many times or by many people, this may indicate that it is more likely to be defect prone. Some of the most commonly used process factors in SDP research are [204, 210]:

- **Number of Changes**: Measures the total number of changes to a software artifact.
- **Number of Defect Fixing Changes**: Measures the number of changes to a software artifact to fix a defect.
- **Number of Pre-release Defects**: Measures the number of defects found in a software artifact prior to release.
- **Age**: Measures the number of days a file existed for.
- **Number of Developers**: Measures the number of developers who made changes to a software artifact.
- **Code Churn**: Measures the amount of churn (i.e., lines added, deleted and changed) of a software artifact.
- **Relative Churn [210]**: Measures the churn of a file, in LOC or files churned, normalized by the total LOC or total file count.
- **Change Complexity [116]**: Factors that are derived from the complexity of the changes performed to a source code artifact.
- **Social [43]**: Factors that combine dependency data from the code and from the contributions of developers.
- **Organizational [197]**: Factors that capture the geographic distribution of the development organization (e.g., the number of sites that modified a software artifact).
- **Ownership [44]**: Factors that measure the level of ownership (i.e., code addition and changes) of a developer of a software artifact.

The aforementioned process factors are the some of the most commonly used process factors in SDP studies, however, we note that other process factors have been used in SDP studies [40, 266].

Other Factors

Although product and process factors are the most commonly used in SDP work, other factors such as execution, social and geography factors have been used.

The type and number of factors used in each paper are listed in Table 2.3. We refer the reader to the individual papers for explanations of the specialized factors used in each paper, however, we explain some of the more popular factors next:

- **Execution [144, 147]**: Captures the execution characteristics of a software system. For example, execution factors can be the deployment percentage of a module and the average transaction time on a system serving a typical user.

- **Prog. Language [225]**: The programming language in which the software is written. For example, Java, C, C++ or Perl.

- **Module Knowledge [203]**: A subjective measure which captures the team's knowledge of a module.

- **Design/UML [50, 78, 80, 221, 291]**: Are factors that capture the design of the software system. These factors can be derived from the definition of the class interfaces at the design stage (e.g., from UML diagrams). These factors may include class factors, parameter types, class attributes and inheritance relationships.

- **Platform and Hardware Configuration [173, 202]**: Factors that capture the platform and HW configurations that software system runs on. For example, whether the software system runs on a Windows or Linux based platform and whether it runs on a single- or multi-core system.

- **Requirement Changes [19]**: Factors related to changes in the requirements of a software system.

2.2.3 Dependent Variables

In addition to using a number of independent variables (i.e., predictors), the surveyed SDP papers used a number of dependent variables as well. We highlight below several of the commonly used dependent variables:

- **Post-release Defects [210,310]**: Is the number of defects that appear after the software is released. Generally, post-release defects is the number of defects within six months of the software release date.

- **Defect Density [210, 216]**: Is generally measured as the number of defects per LOC or KLOC.

- **Defect-introducing Change [23, 153]**: Is a dependent variable that specifies whether a change introduced a defect.

- **Vulnerabilities [188]**: Is a dependent variable which accounts for a security vulnerability that exists in a software artifact.

Research trends: Table 2.3 shows the dependent and independent variables used prior SDP studies. In term of independent variables, we observe that 76% of studies use product factors and 45% use process factors. The large number of studies using product factors maybe explained by the large number of studies that used OO factors to predict factors in the early 2000s. On the other hand, recent studies seem to be trending towards using process factors, especially since recent studies showed that process factors are as good or better predictors than product factors [204].

Figure 2.3 shows the number of papers per year that use process and product factors. We see that, in general, more papers use product factors. However, the number of papers using both product and process factors is increasing over time. Figure 2.4 shows the median and average number of factors used over time. We see that the number of factors is high in 2002 and dips till 2006, when it starts to increase again. We see another increase in 2008, when most SDP papers were published and a slow decline since.

In terms of the dependent variables, we find that 7% of studies predict pre-release defects, whereas 65% predict post-release defects. Generally speaking, predicting for post-release defects is more desirable since post-release defects are the most important for organizations since they reflect the customer facing defects.

Table 2.3: Factors used in SDP studies. An '✓' means applied, a '.' means not applied and '?' means could not determine.

	Independent Variables				Dependent Variables		
Paper	**Product**	**Process**	**# of metrics**	**Other**	**Pre**	**Post**	**Other**
Yuan; 2000 [296]	✓	✓	10	.	.	✓	.
Cartwright; 2000 [53]	✓	.	12	.	.	✓	.
Neufelder; 2000 [216]	.	✓	14	.	.	.	Defect density
Khoshgoftaar; 2000 [147]	✓	✓	42	Execution	.	✓	.
Khoshgoftaar; 2000 [148]	✓	✓	42	Execution	.	✓	.
Morasca; 2000 [203]	✓	✓	8	ModuleKnowlege	.	✓	.
Wong; 2000 [291]	.	.	5	Design	.	✓	.
Fenton; 2000 [89]	✓	.	3	Design	✓	✓	.
Graves; 2000 [105]	✓	✓	9	.	.	✓	.
Khoshgoftaar; 2001 [142]	✓	✓	5	.	.	✓	.
El Emam; 2001 [78]	.	.	26	OO design	.	✓	.
Denaro; 2002 [66]	✓	.	38	.	.	✓	.
Briand; 2002 [50]	✓	.	22	Polymorphism	.	✓	.
Khoshgoftaar;2002 [146]	✓	✓	42	Execution	.	✓	.
Quah; 2003 [232]	✓	.	14	OO desgin	✓	✓	.
Khoshgoftaar; 2003 [144]	✓	✓	42	Execution	.	✓	.
Guo; 2003 [106]	✓	.	21	.	.	✓	.

Paper	**Independent Variables**				**Dependent Variables**		
	Product	**Process**	**# of factors**	**Other**	**Pre**	**Post**	**Other**
Amasaki; 2003 [12]	✓	✓	17	Effort; test items	.	.	Faults in development phase
Succi; 2003 [264]	✓	.	7	.	✓	.	.
Guo; 2004 [107]	✓	.	21	.	.	✓	.
Li; 2004 [175]	?	?	?	?	.	✓	.
Ostrand; 2004 [225]	✓	✓	6	Prog. Language	.	.	Pre+Post
Koru;2005 [163]	✓	.	31	.	.	✓	.
Gyimothy; 2005 [109]	✓	.	8	.	.	.	Pre+Post
Mockus; 2005 [202]	.	.	8	Deployment; usage; platform; HW configuration	.	✓	.
Nagappan; 2005 [209]	.	.	? 2	PREfix and PREfast	.	.	✓(pre-release defect density)
Hassan; 2005 [117]	.	✓	4	.	✓	.	.
Nagappan; 2005 [210]	.	.	8	Relative churn	.	.	✓(pre-release defect density)
Tomaszewski;2006 [270]	.	✓	3	.	.	✓	Fault density
Pan; 2006 [227]	✓	.	31	Program slicing	✓	.	.
Nagappan; 2006 [212]	✓	.	18	.	.	✓	.
Zhou; 2006 [302]	✓	.	6	Design	.	✓	.
Li; 2006 [173]	✓	✓	47	Deployment; usage; platform; HW configuration	.	✓	.
Knab; 2006 [159]	✓	✓	16	.	.	.	Defect density
Arisholm; 2006 [19]	✓	✓	32	Requirement changes	.	.	Pre+Post
Tomaszewski;2007 [271]	✓	.	9	.	.	✓	Fault density
Ma; 2007 [178]	✓	.	6	.	.	✓	.
Menzies; 2007 [190]	✓	.	38	.	.	✓	
Olague; 2007 [223]	✓	.	18	.	?	?	?
Bernstein; 2007 [34]	✓	✓	22	Temporal	.	✓	.

Paper	Independent Variables				Dependent Variables		
	Product	**Process**	**# of factors**	**Other**	**Pre**	**Post**	**Other**
Aversano; 2007 [23]	.	.	NA	Weighted term vector	.	.	Bug introducing change
Kim; 2007 [155]	.	✓	4	Least recently used (LRU); LRU change; LRU bug	✓	.	.
Ratzinger; 2007 [238]	✓	✓	17	Evolution	?	?	Time based
Mizuno; 2007 [195]	.	.	NA	Code	.	.	Pre+Post
Weyuker; 2007 [283]	✓	✓	8	Developer	.	✓	.
Zimmermann; 2007 [310]	✓	✓	73	.	.	✓	.
Moser; 2008 [204]	✓	✓	49	.	.	✓	.
Kamei; 2008 [140]	✓	.	15	.	?	?	.
Zimmermann; 2008 [304]	.	.	22	Dependency	.	✓	.
Zimmermann; 2008 [305]	✓	.	46	.	.	✓	.
Nagappan; 2008 [214]	✓	✓	28	Churn; dependencies; code coverage	.	✓	.
Zhang; 2008 [298]	.	.	1	# of defects	.	.	Pre+Post
Pinzger; 2008 [230]	.	✓	7	Developer networks	.	✓	.
Lessmann; 2008 [170]	✓	.	13-37	.	.	✓	.
Watanabe; 2008 [280]	✓	.	63	.	?	?	?
Kim; 2008 [153]	✓	.	63+	Terms from changes	.	.	Bug introducing change
Jiang; 2008 [131]	✓	.	40	Design	.	✓	.
Weyuker; 2008 [284]	?	?	?	?	.	.	Pre+Post
Jiang; 2008 [133]	?	?	21-40	.	.	✓	.
Tosun; 2008 [273]	✓	.	?	.	.	✓	.
Koru; 2008 [160]	✓	.	.	.	.	.	Pre+Post
Menzies; 2008 [192]	✓	.	?	.	.	✓	.
Layman; 2008 [168]	✓	✓	159	.	.	.	Pre+Post
Goronda; 2008 [103]	✓	.	21	.	.	✓	.
Vandecruys; 2008 [275]	✓	.	23	.	.	✓	.
Elish; 2008 [77]	✓	.	21	.	.	✓	.
Ratzinger; 2008 [239]	.	.	110	Refactoring and non-refactoring features	.	.	Pre+Post (time-based)

	Independent Variables				**Dependent Variables**		
Paper	**Product**	**Process**	**# of factors**	**Other**	**Pre**	**Post**	**Other**
Meneely; 2008 [189]	✓	✓	13	.	.	✓	.
Wu; 2008 [293]	✓	.	6	.	.	✓	.
Tarvo; 2008 [266]	✓	✓	29+	Dependency	.	.	regression
Holschuh; 2009 [124]	✓	✓	78	Dependency; Code smell	.	✓	.
Jia; 2009 [129]	✓	✓	30-40	.	.	✓	.
Shin; 2009 [257]	✓	✓	22	Calling structure	.	✓	.
Mende; 2009 [185]	✓	✓	?	.	.	✓	.
Binkley; 2009 [40]	✓	.	3	Natural language	.	✓	.
D'Ambros; 2009 [62]	✓	✓	16	Change coupling	.	.	Pre+Post
Hassan; 2009 [116]	.	✓	5	Change complexity	.	.	Pre+Post (time-based)
Bird; 2009 [43]	.	✓	24	Socio-technical	.	✓	.
Ferzund; 2009 [92]	.	.	27	Hunk metrics	.	.	Bug introducing hunk
Liu; 2010 [176]	✓	.	15	.	.	✓	.
Erika; 2010 [80]	✓	.	?	UML	?	?	?
Zhou; 2010 [303]	✓	✓	10	.	.	✓	.
Mockus; 2010 [197]	✓	✓	13	Geography	.	✓	.
Kamei; 2010 [138]	✓	✓	22	.	.	✓	.
Meneely; 2010 [188]	.	✓	4	Developer	.	.	Vulnerabilities
Nguyen; 2010 [218]	✓	.	33	Dependency	.	✓	.
Shihab; 2010 [254]	✓	✓	34	.	.	✓	.
Menzies; 2010 [191]	✓	.	21-39	.	.	✓	.
Weyuker; 2010 [286]	✓	✓	7	Prog. Language	.	.	Pre+Post
Mende; 2010 [184]	✓	?	?	.	.	✓	.
Nugroho; 2010 [221]	✓	.	3	UML	.	✓	.
Song; 2011 [261]	✓	.	21-40	.	.	✓	.
Nguyen; 2011 [219]	✓	✓	?	Topic models	.	✓	.
Mende; 2011 [186]	✓	.	17	.	.	.	Pre+Post
Lee; 2011 [169]	✓	✓	81	.	.	.	Post (time based)

Paper	Independent Variables				Dependent Variables		
	Product	Process	# of factors	Other	Pre	Post	Other
Shihab; 2011 [255]	✓	✓	16	Co-change; time	.	✓	Breakages and Surprise
D'Ambros; 2011 [61,63]	✓	✓	44	entropy of changes; churn of source code and entropy of source code	.	✓	.
Kpodjedo; 2011 [166]	✓	.	11-32	Design evolution	.	✓	.
Bird; 2011 [44]	✓	.	7	Ownership	✓	✓	.
Meneely; 2011 [187]	.	.	4	Team expansion	.	.	Failures per hour
Giger; 2011 [101]	.	✓	2	Fine-grained code changes	.	.	Pre+Post
Percentage of papers	**76**	**45**	**-**	**-**	**7**	**65**	**-**

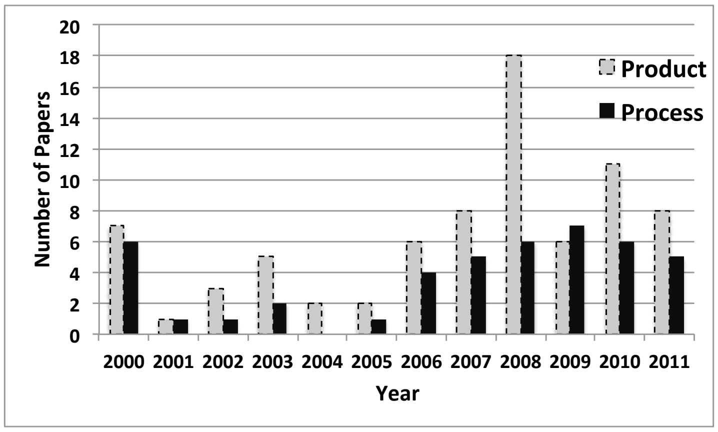


Figure 2.3: Number of papers using product and process factors per year

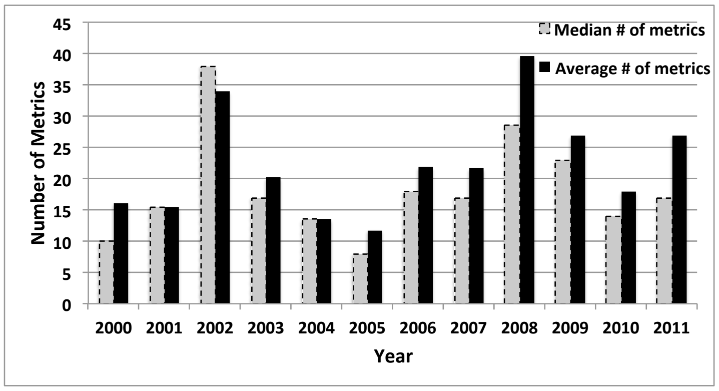


Figure 2.4: Average and median number of factors per year

2.2.4 Model Building

SDP work uses a variety of modelling techniques to perform their prediction. To categorize the different prediction models, we use the same categorization proposed by Lessmann *et al.* [170]. In this chapter, we categorize and compare previous work based on the types of models used to perform the prediction.

Statistical Models

Statistical models formalize the relationship between the independent variables and the dependent variable(s) in the form of a mathematical equation. Below, we describe some of the most commonly used statical models in SDP studies.

Naive Bayes Classifier: is a simple probabilistic classifier, based on Bayes theorem. The classifier calculates a future probability for a class as a product of a factor value times a prior probability of that class [190]. The simplicity of the Naive Bayes classifier makes it an attractive model to use in SDP work [190].

Linear Regression: is used to reveal relationships between one or more independent variables (e.g., product or process factors) and the dependent variable, which in most cases is the *number* of defects.

In addition to serving as a prediction tool, the models can be used to better understand the relationship between the dependent and independent variables. From the linear regression model, it is possible to infer which independent variables have an effect, the magnitude of the effect and the direction (i.e., positive or negative) of the effect.

Linear regression makes some assumptions that must be met to ensure the reliability of the models. First, the independent variables must not be highly correlated. Second, the residuals must be normally distributed. Therefore, to satisfy such assumption, a log transformation is often performed on any highly skewed variables used in the model.

Multivariate Adaptive Regression Splines (MARS): is a statistical method that attempts to approximate complex relationships by a series of linear regressions on different intervals of the independent variable range [50].

Logistic Regression: similar to linear regression, logistic regression also correlates the independent variables with the dependent variable. The main difference however is that the dependent variable in this case is a probability of the software artifact containing a defect (i.e., belongs to a yes or no class). In most cases, a cutoff probability is used to determine whether a software artifact belongs to the yes or the no class. For example, a cutoff of 0.5 would mean that any software artifact with a predicted probability of 0.5 or above is classified as being fault-prone. The same assumptions about collinearity and normal distribution of the data holds for logistic regression models as well.

Variants of Logistic regression models: Other variants of logistic regression models also exist and have been employed in prior SDP studies. Weyuker *et al.* [284] used Negative Binomial Regression (NBR). NBR assumes a linear relationship between the dependent and independent variables and models the logarithm of the expected number of faults.

Tree-based Models

Here, we describe some of the most commonly used decision tree-based models in SDP work. Decision tree-based approaches recursively partition the training data by means of attribute splits. The different decision tree-based approaches differ mainly in the splitting criterion which determine the attribute used in a given iteration to separate the data [170].

C4.5 Decision Tree Classifiers: are used to predict whether or not the software artifact is defect-prone or not. A number of algorithms are used to build decision trees, the most common is the C4.5 algorithm [233]. The algorithm starts with an empty tree and adds decision nodes or leafs at each level. The information gain using a particular attribute is calculated and the attribute with the highest information gain is chosen. Further analysis is done to determine the cut-off value at which to split the attribute. This process is repeated at each level until the number of instances classified at the lowest level reaches a specified minimum.

Classification and regression trees (CART): is a statistical tool that automatically searches large complex databases, searching for, discovering and isolating significant patterns and relationships in data [145]. The CART algorithm searches for questions that split nodes into relatively homogenous child nodes, such as a group consisting largely of responders, or high risk components. As the tree evolves, the nodes become increasingly more homogenous, identifying important segments. The predictor variables are used to split the nodes into segments, read directly off the tree and summarized in the variable importance tables, which are the key drivers of the response variable. The CART methodology solves a number of performance, accuracy and operational problems that plague many current decision-tree methods.

Random Forests (RF) Classifiers: consist of a number of tree-structured classifiers [107]. New objects are classified from an input vector that is composed of input vectors on each tree in the forest. Each tree casts a vote at the input vector by providing a classification. The forest selects the classification that has the most votes over all trees in the forest.

The main advantages of RF is that they generally outperform simple decision trees algorithms in terms of prediction accuracy. Also, RF is more resistant to noise in data. In addition, the RF algorithm deals well with correlated attributes [180].

Recursive Partitioning (RP): constructs a binary decision tree to partition training observations with the goal of producing leaf nodes that are each as homogeneous as possible with respect to the value of the dependent variable [284]. The RP algorithm starts by splitting the entire set of observations into two nodes, based on a binary cut of a single predictor variable. The predictor variable and cut are chosen to maximize the reduction in total sum of squared errors for the two resulting nodes. Additional steps are conducted where a single node is based on cutting a single predictor variable until a stopping criterion is satisfied. The prediction for any dependent variable is the mean of the dependent variable in the training data for the corresponding node [284].

The main advantage of tree classifiers is that they offer explainable models. This is very advantageous because they can be used to understand what attributes affect whether or not a defect will appear. Managers can use this information about the attributes to drive process changes in order to improve their software quality.

Other Models

In addition to the aforementioned models, a number of other models have been used in prior SDP studies. We list the model used in each paper in Table 2.4. We list some of the other models used for SDP below:

- **Support Vector Machines (SVM)**: is designed for binary classification. SVM utilizes mathematical programming to directly model the decision boundary between classes. An SVM tries to find the maximum margin hyperplane, a linear decision boundary with the maximum margin between it and the training examples in class 1 and the training examples in class 2 [153].

- **Case Based Reasoning [145]**: is a technique that aims to find solutions to a new problem based on past experiences, which are represented by cases in a case library. In the context of classification, each case in the case library has known attributes and class memberships.

- **K-Nearest Neighbor (KNN) [23]**: is a type of instance based learning for classifying objects based on closest training examples in the feature space. The training examples are mapped into multidimensional feature space which is partitioned into regions by class labels. A point the space is assigned to the class if it is the most frequent class label among the k nearest training samples. A Euclidean distance is used to compute the closeness to the samples. The number of neighbors can be set as a parameter or be selected considering the mean squared error for the training set.

Other more complex models used for SDP include Artificial Neural Networks [103, 232], AntMiner+ (based on Ant Colony Optimization) [275], Capture-recapture models [48], Dempster Shafer belief networks [106], Cox proportional Hazards Model for recurrent events [160, 164], Genetic programming [238], IB1 and Bagging [178], Multi-boosting [23]. Table 2.4 gives a list of the used models in each paper.

Research trends: Table 2.4 show the models used in prior SDP studies. In terms of tree-based models, we find that 26% of studies use decision trees, 15% use random forests, 3% use recursive partitioning and 2% use CART models.

In terms of statistical models, we find that 47% of prior studies use logistic regression, 22% use linear regression, 16% use Naive Bayes, and only 1% use MARS models. The fact that logistic regression is more commonly used is not surprising since traditionally, SDP studies are always interested in determining which software artifacts have one or more post-release defects. Predicting the number of defects is another way of prioritizing defect-prone locations, however, predicting the number of defects is harder to validate (i.e., no easy measure of precision and recall can be applied). SVM is used in 8% of the SDP studies.

Table 2.4: Types of models used in SDP studies. An '✓' means applied, a '.' means not applied and '?' means could not determine.

	Statistical				Decision Tree-based					
Paper	**Naive Bayes**	**MARS**	**Linear Regression**	**Logistic Regression**	**Decision Trees**	**CART**	**Random Forests**	**Recursive Partitioning**	**SVM**	**Other**
Yuan; 2000 [296]	.	.	.	.	.	.	.	.	.	Fuzzy subtractive clustering
Cartwright; 2000 [53]	.	.	✓	.	.	.	.	.	.	.
Neufelder; 2000 [216]	.	.	.	.	.	.	.	.	.	.
Khoshgoftaar; 2000 [147]	.	.	.	.	✓	.	.	.	.	.
Khoshgoftaar; 2000 [148]	.	.	.	.	.	✓	.	.	.	.
Morasca; 2000 [203]	.	.	.	✓	.	.	.	.	.	Rough Sets
Wong; 2000 [291]	.	.	.	.	.	.	.	.	.	?
Fenton; 2000 [89]	.	.	.	.	.	.	.	.	.	.
Graves; 2000 [105]	.	.	.	✓	.	.	.	.	.	GLM
Khoshgoftaar; 2001 [142]	.	.	✓	.	.	.	..	.	.	Zero-Inflated Poission
El Emam; 2001 [78]	.	.	.	✓	.	.	.	.	.	.
Denaro; 2002 [66]	.	.	.	✓	.	.	.	.	.	.
Briand; 2002 [50]	.	✓	.	✓	.	.	.	.	.	.
Khoshgoftaar;2002 [146]	.	.	.	.	✓	.	.	.	.	.
Quah; 2003 [232]	.	.	.	.	.	.	.	.	.	Neural Networks
Khoshgoftaar; 2003 [144]	.	.	✓	.	✓	✓	.	.	.	CART-LS; CART-LAD; S-PLUS; MLR; ANN and CBR
Guo; 2003 [106]	.	.	.	✓	.	.	.	.	.	Dempster Shafer belief networks; Discriminant analysis
Amasaki; 2003 [12]	.	.	.	.	.	.	.	.	.	BBN
Succi; 2003 [264]	.	.	✓	.	.	.	.	.	.	NBR; zero-inflated NBR and Poisson regression
Guo; 2004 [107]	.	.	.	✓	✓	.	✓	.	.	Discriminant analysis
Li; 2004 [175]	.	.	.	.	.	.	.	.	.	Exponential; Gamma; Power; Logarithmic and Weibull
Ostrand; 2004 [225]	.	.	.	.	.	.	.	.	.	.
Koru;2005 [163]	.	.	.	.	✓	.	.	.	.	J48 and Kstar
Gyimothy; 2005 [109]	.	.	✓	✓	✓	.	.	.	.	Neural Networks
Mockus; 2005 [202]	.	.	.	✓	.	.	.	.	.	.
Nagappan; 2005 [209]	.	.	✓	.	.	.	.	.	.	Discriminant Analysis
Hassan; 2005 [117]	.	.	.	.	.	.	.	.	.	Ranking

	Statistical				**Decision Tree-based**					
Paper	**Naive Bayes**	**MARS**	**Linear Regression**	**Logistic Regression**	**Decision Trees**	**CART**	**Random Forests**	**Recursive Partitioning**	**SVM**	**Other**
Nagappan; 2005 [210]	.	.	✓	.	.	.	.	.	.	Discriminant Analysis
Tomaszewski;2006 [270]	.	.	.	.	.	.	.	.	.	Random vs. best model
Pan; 2006 [227]	.	.	.	.	.	.	.	.	.	Bayesian Network
Nagappan; 2006 [212]	.	.	.	✓	.	.	.	.	.	.
Zhou; 2006 [302]	✓	.	.	✓	.	.	✓	.	.	Nnge
Li; 2006 [173]	.	.	✓	✓	.	.	.	.	.	.
Knab; 2006 [159]	.	.	.	.	✓	.	.	.	.	.
Arisholm; 2006 [19]	.	.	.	✓	.	.	.	.	.	.
Tomaszewski;2007 [271]	.	.	.	.	.	.	.	.	.	Random vs. best model
Ma; 2007 [178]	✓	.	.	✓	✓	.	.	.	.	IB1 and Bagging
Menzies; 2007 [190]	✓	.	.	.	✓	.	.	.	.	OneR
Olague; 2007 [223]	.	.	.	✓	.	.	.	.	.	.
Bernstein; 2007 [34]	.	.	✓	.	✓	.	.	.	.	.
Aversano; 2007 [23]	.	.	.	✓	✓	.	.	.	✓	Multi-boosting and KNN
Kim; 2007 [155]	.	.	.	.	.	.	.	.	.	.
Ratzinger; 2007 [238]	.	.	✓	.	.	.	.	.	.	Genetic programming
Mizuno; 2007 [195]	.	.	.	.	.	.	.	.	.	Spam filter
Weyuker; 2007 [283]	.	.	.	✓	.	.	.	.	.	NBR
Zimmermann; 2007 [310]	.	.	✓	✓	.	.	.	.	.	.
Moser; 2008 [204]	✓	.	.	✓	✓	.	.	.	.	.
Kamei; 2008 [140]	.	.	.	✓	✓	.	.	.	.	Linear discriminant
Zimmermann; 2008 [304]	.	.	✓	✓	.	.	.	.	.	.
Zimmermann; 2008 [305]	.	.	✓	✓	.	.	.	.	.	.
Nagappan; 2008 [214]	.	.	.	✓	.	.	.	.	.	.
Zhang; 2008 [298]	.	.	.	.	.	.	.	.	.	Polynomial
Pinzger; 2008 [230]	.	.	✓	✓	.	.	.	.	.	.
Lessmann; 2008 [170]	✓	.	.	✓	✓	✓	✓	.	✓	LDA; QDA; BayesNet; LARS; RVM; K-NN; K*; MLP; RBF net; L-SVM; LS-SVM; LP; VP; ADT; LMT
Watanabe; 2008 [280]	.	.	.	.	✓	.	.	.	.	.
Kim; 2008 [153]	.	.	.	.	.	.	.	.	✓	.
Jiang; 2008 [131]	✓	.	.	✓	.	.	✓	.	.	Bagging; Boosting
Weyuker; 2008 [284]	.	.	.	.	.	.	.	✓	.	NBR
Jiang; 2008 [133]	✓	.	.	✓	.	.	✓	.	.	Boosting; bagging

Paper	Statistical				Decision Tree-based					
	Naive Bayes	**MARS**	**Linear Regression**	**Logistic Regression**	**Decision Trees**	**CART**	**Random Forests**	**Recursive Partitioning**	**SVM**	**Other**
Tosun; 2008 [273]	.	.	.	.	.	.	.	.	.	Ensemble of naive bayes; neural networks and voting feature interval
Koru; 2008 [160]	.	.	.	.	.	.	.	.	.	Cox
Menzies; 2008 [192]	✓	.	.	.	✓	.	.	.	.	.
Layman; 2008 [168]	.	.	.	✓	.	.	.	.	.	.
Goronda; 2008 [103]	.	.	.	.	.	.	.	.	✓	ANN
Vandecruys; 2008 [275]	.	.	.	✓	✓	.	.	.	✓	RIPPER; 1-NN; majority vote
Elish; 2008 [77]	✓	.	.	✓	✓	.	✓	.	✓	KNN; Multi-layer perceptrons; radial basis function; BBN;
Ratzinger; 2008 [239]	.	.	.	.	✓	.	.	.	.	LMT; Repeated Incremental Pruning; Nnge
Meneely; 2008 [189]	.	.	✓	✓	.	.	.	.	.	.
Wu; 2008 [293]	.	.	.	.	.	.	.	.	.	?
Tarvo; 2008 [266]	.	.	.	✓	.	✓	.	.	.	Multilayer perceptron
Holschuh; 2009 [124]	.	.	✓	.	.	.	.	.	✓	
Jia; 2009 [129]	✓	.	.	.	✓	.	✓	.	.	IB1
Shin; 2009 [257]	.	.	.	✓	.	.	.	.	.	NBR
Mende; 2009 [185]	.	.	.	.	.	.	✓	.	.	.
Binkley; 2009 [40]	.	.	.	.	.	.	.	.	.	Linear mixed-effects
D'Ambros; 2009 [62]	.	.	✓	.	.	.	.	.	.	.
Hassan; 2009 [116]	.	.	✓	.	.	.	.	.	.	.
Bird; 2009 [43]	.	.	.	✓	.	.	.	.	.	.
Ferzund; 2009 [92]	.	.	.	✓	.	.	✓	.	.	.
Liu; 2010 [176]	✓	.	.	✓	✓	.	✓	.	.	Jrip; DecisionTable; OneR; PART; Ibk; IB1; ADTree; Ridor; LWLStump; SM; Bagging; LOC; TD
Erika; 2010 [80]	?	.	.	✓	?	.	.	.	?	?
Zhou; 2010 [303]	✓	.	.	✓	.	.	.	.	.	Neural Networks; Kstar; Adtree; No learner
Mockus; 2010 [197]	.	.	.	✓	.	.	.	.	.	.
Kamei; 2010 [138]	.	.	.	.	.	.	✓	✓	.	MASS
Meneely; 2010 [188]	.	.	.	.	.	.	.	.	.	Bayesian Network
Nguyen; 2010 [218]	.	.	.	✓	.	.	.	.	.	.
Shihab; 2010 [254]	.	.	.	✓	.	.	.	.	.	.

	Statistical				**Decision Tree-based**					
Paper	**Naive Bayes**	**MARS**	**Linear Regression**	**Logistic Regression**	**Decision Trees**	**CART**	**Random Forests**	**Recursive Partitioning**	**SVM**	**Other**
Menzies; 2010 [191]	✓	.	.	.	✓	.	.	.	.	RIPPER
Weyuker; 2010 [286]	.	.	.	.	.	.	✓	✓	.	Bayesian additive regression trees; NBR
Mende; 2010 [184]	.	.	.	.	.	.	✓	.	.	.
Nugroho; 2010 [221]	.	.	.	✓	.	.	.	.	.	.
Song; 2011 [261]	✓	.	.	.	✓	.	.	.	.	OneR
Nguyen; 2011 [219]	.	.	✓	.	.	.	.	.	.	.
Mende; 2011 [186]	.	.	.	.	.	.	✓	.	.	.
Lee; 2011 [169]	.	.	.	✓	✓	.	.	.	.	Bayesian Network
Shihab; 2011 [255]	.	.	.	✓	.	.	.	.	.	.
D'Ambros; 2011 [61, 63]	✓	.	.	✓	✓	.	.	.	.	.
Kpodjedo; 2011 [166]	.	.	✓	✓	.	.	.	.	.	.
Bird; 2011 [44]	.	.	✓	.	.	.	.	.	.	.
Meneely; 2011 [187]	.	.	✓	.	.	.	.	.	.	.
Giger; 2011 [101]	✓	.	.	✓	✓	.	✓	.	✓	Exhaustive CHAID; Neural Nets
Percentage of papers	**16**	**1**	**22**	**47**	**26**	**2**	**15**	**3**	**8**	**-**

2.2.5 Performance Evaluation

Once a prediction model is built, it must be evaluated. To evaluate their models, SDP studies use different validation approaches. The validation approaches determines how the data is split to perform the validation. Generally speaking, most SDP studies divide the data into two sets: a training set and a test set. The training set is used to train the prediction model. Then, the accuracy of the predictions are measured using the test set. We list the different validation approaches used in prior SDP research:

Validation Approach

- **10-fold Cross Validation**: in a 10-fold cross validation, the data set is divided into 10 sets. One set is used as testing data and the remaining 9 sets are used for training. We indicate papers that employ 10-fold cross validation.
- **Cross-release Validation**: some SDP papers train their models on data from one release and use data from another release to validate the performance of their models.
- **Cross-project Validation**: similar to cross-release validation, in cross-project validation the prediction model built from one project (i.e., training data) is applied to a different project (i.e., testing data).

Performance Measures of Classification Models

SDP studies use a number of ways to evaluate their prediction models. The performance evaluation methods can be divided into two main categories: evaluation measures for predictors that predict the number of an instance (e.g., linear regression models) and evaluation measures for models that output a classification (or a probability that is converted into a classification), e.g., logistic regression or decision trees.

For models that perform classification, performance is measured in two ways: predictive power measures and explanative power measures. Here, we list the most common used measures. Table 2.6 lists the measures used in each paper.

Predictive Power

Measures the accuracy of the model in predicting the software artifacts that have one or more defects. The accuracy measures are based on the classification results in the confusion matrix (shown in Table 3.3). Each cell in the Table 3.3 compares the overlap (or lack of) between the predicted and actual set. A defect can be classified as defective when it truly is defective (true positive, TP); it can be classified as defective when actually it is not defective (false positive, FP); it can be classified as not defective when it is actually defective (false negative, FN); or it can be classified as not defective and it truly is not defective (true negative, TN). Using the values stored in the confusion matrix, the performance of SDP approaches is measured as:

Table 2.5: Confusion msatrix

	True class	
Classified as	Defective	Not Defective
Defective	TP	FP
Not Defective	FN	TN

- **Precision/Correctness**: is the number of instances correctly classified as defective, divided by the total number of instances classified as defective. Precision is calculated as Pr $= \frac{TP}{TP+FP}$. A precision value of 100% would indicate that every instance was classified as (not) defective was actually (not) defective.

- **Recall/Sensitivity/Probability of Detection (PD)**: is the number of instances correctly classified over all of the actually defective instances. Recall is calculated as Re $= \frac{TP}{TP+FN}$. A recall value of 100% would indicate that every actual (not) defective instance was classified as (not) defective.

- **F-measure**: is a composite measure that measures the weighted harmonic mean of precision and recall. It is measured as F-measure $= \frac{2*Pr*Re}{Pr+Re}$.

- **Accuracy**: measures the number of correctly classified instances (both the defective and the not defective) over the total number of instances. Accuracy = $\frac{TP+TN}{TP+FP+TN+FN}$.

- **Completeness**: is the number of faults in instances classified as fault-prone, divided by the total number of faults in the system.

- **Specificity**: is the proportion of correctly identified non-defective instances. It is related to the probability of false alarm (PF) as PF = 1-specificity.

- **Percentage of Faults**: Accounts for the percentage of faults found in the top X files. Generally, studies look for the % of faults that can be discovered in the top 20% most fault-prone file.
- **Receiver Operating Characteristic (ROC) curve**: Most classification algorithms depend on a threshold. The performance of the predictor will depend on this threshold, achieving a tradeoff between PD and PF. The (PF,PD) pair generated by adjusting the threshold form an ROC curve. The Area Under the ROC Curve (AUC) is used to measure the performance of different classification methods.

Since there are generally fewer defective instances than not defective instances, using the accuracy measure alone may be misleading (e.g., if 90% of the data does not have a defect, then a classifier that predicts all instances will be 90% accurate).

Explanatory Power

In addition to measuring the predictive power, explanatory power is also used in SDP studies. Explanatory power, often measured in R^2 or deviance explained ranges between 0-100%, and quantifies the variability in the data explained by the model. Some studies (e.g., [254, 255]) also report and compare the variability explained by each of the independent variables in the model. Examining the explained variability of each independent variable enables researchers to quantify the relative importance of each of the independent variables in the model.

Performance Measures of ‘Number of Defects’ Models

Some SDP studies aim to predict the number of defects that may appear in a software artifact in the future. These prior studies used different measures to evaluate the performance of their models. Some of the most common performance measures are:

- **Correlation**: Measures the correlation between the ranked list of artifacts based on the number of predicted defects and the ranked list of artifacts based on the actual number of defects.

- **Average Absolute Error (AAE)**: measures the average of the absolute error between the actual number of defects in an artifact and the predicted number of defects. $AAE = \frac{1}{n}\sum_{i=1}^{n}|\bar{y}_i - y_i|$, where n is the number of artifacts, y_i is the actual value and $\bar{y}_i$ is the predicted value [142].

- **Average Relative Error (ARE)**: measures the average relative error between the actual number of defects in an artifact and the predicted number of defects. $ARE = \frac{1}{n}\sum_{i=1}^{n}\frac{|\bar{y}_i - y_i|}{y_i+1}$, where n is the number of artifacts, y_i is the actual value and $\bar{y}_i$ is the predicted value [142].

Research trends: Table 2.6 shows the performance evaluation methods used in prior SDP studies. In terms of how SDP studies are validated, we find that 30% of studies use 10-fold cross validation, 29% perform cross-release validation and only 5% perform cross-project validation. The practice of splitting data into training and testing data is very common, and is performed in 77% of the surveyed papers.

To evaluate predictive power, we find that 39% of studies use correlation, 34% use precision, 31% use recall, 23% use accuracy and 15% use AUC. A number of studies also use other evaluation methods such as the percentage of faults in the top 20% most defect-prone files.

In terms of explanative power, 17% of studies use R^2 and 12% use deviance explained. The main difference between R^2 and deviance explained is that deviance explained is used when logistic regression models are used, whereas R^2 is used when linear regression models are used. Both are measures for the quality of fit of the model to the data.

Table 2.6: Performance evaluation measures used in SDP studies. An '✓' means applied, a '.' means not applied and '?' means could not determine.

	Cross Validation				Predictive Power							Explanative Power	
Paper	**10-fold**	**Cross-project**	**Cross-release**	**Training and testing data**	**Correlation**	**Precision**	**Recall**	**Accuracy**	**F-measure**	**AUC**	**Other**	R^2	**Deviance Explained**
Yuan; 2000 [296]	.	.	?	✓	.	.	.	.	.	.	MCR; effectiveness; efficiency	.	.
Cartwright; 2000 [53]	.	.	.	?	✓	.	.	.	.	.	.	✓	.
Neufelder; 2000 [216]	.	.	.	.	✓	.	.	.	.	.	.	.	.
Khoshgoftaar; 2000 [147]	✓	.	✓	✓	.	.	.	.	.	.	MCR	.	.
Khoshgoftaar; 2000 [148]	✓	.	✓	✓	.	.	.	.	.	.	MCR	.	.
Morasca; 2000 [203]	?	.	✓	.	.	.	.	.	.	.	Overall completeness; faulty module completeness and correctness	.	✓
Wong; 2000 [291]	?	.	✓	✓	.	✓	✓	.	.	.	.	.	.
Fenton; 2000 [89]	.	.	✓	.	✓	.	.	.	.	.	.	.	.
Graves; 2000 [105]	.	.	.	✓	.	.	.	.	.	.	?	.	.
Khoshgoftaar; 2001 [142]	.	.	.	✓	.	.	.	.	.	.	AAE; ARE	.	.
El Emam; 2001 [78]	.	.	✓	✓	.	✓	✓	✓	.	✓	.	.	✓
Denaro; 2002 [66]	.	.	✓	✓	.	✓	✓	✓	.	.	.	.	✓
Briand; 2002 [50]	✓	✓	.	✓	.	.	.	.	.	.	Completeness; correctness; cost-effectiveness	.	.
Khoshgoftaar;2002 [146]	.	.	✓	✓	.	.	.	.	.	.	MCR	.	.
Quah; 2003 [232]	?	.	.	✓	.	.	.	.	.	.	Min/MSE; Min/AAE	✓	.
Khoshgoftaar; 2003 [144]	✓	.	✓	✓	.	.	.	.	.	.	AAE; ARE and ANOVA	.	.
Guo; 2003 [106]	✓	.	.	✓	.	.	.	✓	.	.	Specificity; Sensitivity; PFA; Effort	.	.
Amasaki; 2003 [12]	?	.	.	✓	.	.	.	✓	.	.	Error rate	.	.
Succi; 2003 [264]	?	?	?	?	.	..	.	.	.	.	Applicability; effectiveness and predictive ability	.	.
Guo; 2004 [107]	?	.	.	✓	.	.	.	✓	.	.	Specificity; Sensitivity; PFA; Effort	.	.
Li; 2004 [175]	?	?	?	?	.	.	.	.	.	.	AIC	.	.
Ostrand; 2004 [225]	.	.	✓	.	.	.	.	.	.	.	% faults	.	.
Koru;2005 [163]	✓	.	.	✓	.	✓	✓	.	✓	.	.	.	.

	Cross Validation				**Predictive Power**							**Explanative Power**	
Paper	**10-fold**	**Cross-project**	**Cross-release**	**Training and testing data**	**Correlation**	**Precision**	**Recall**	**Accuracy**	**F-measure**	**AUC**	**Other**	R^2	**Deviance Explained**
Gyimothy; 2005 [109]	.	.	.	✓	.	.	.	.	.	.	Completeness and correctness	✓	✓
Mockus; 2005 [202]	?	.	✓	✓	.	.	.	.	.	.	.	.	✓
Nagappan; 2005 [209]	.	.	.	✓	✓	.	.	.	.	.	.	✓	.
Hassan; 2005 [117]	.	.	.	.	.	.	.	.	.	.	Hit rate and avg. prediction age	.	.
Nagappan; 2005 [210]	.	.	.	✓	✓	.	.	.	.	.	.	✓	.
Tomaszewski;2006 [270]	.	.	.	.	.	.	.	.	.	.	Compare to random and best models	.	.
Pan; 2006 [227]	✓	.	.	✓	.	✓	✓	✓	.	.	.	.	.
Nagappan; 2006 [212]	.	✓	.	✓	✓	.	.	.	.	.	.	.	✓
Zhou; 2006 [302]	?	.	.	?	.	✓	.	.	.	.	Correctness; completeness	✓	✓
Li; 2006 [173]	.	.	✓	.	✓	.	.	.	.	.	ARE		.
Knab; 2006 [159]	?	?	?	✓	✓	.	.	✓	.	.	Classification rates	.	.
Arisholm; 2006 [19]	✓	.	✓	✓	.	.	.	.	.	.	False positives and false negatives	.	.
Tomaszewski;2007 [271]	.	.	.	.	✓	.	.	.	.	.	Compare to random and best models	.	.
Ma; 2007 [178]	✓	.	.	✓	✓	✓	.	✓	✓	.	Specificity; Sensitivity; PFA; G-mean	.	.
Menzies; 2007 [190]	✓	.	.	✓							POD; PFA and ROC		.
Olague; 2007 [223]	?	.	.	✓	✓	.	.	✓	.	.	.	.	.
Bernstein; 2007 [34]	.	.	✓	✓	✓	.	.	✓	.	✓	ROC; RMSE; AAE	.	.
Aversano; 2007 [23]	✓	.	.	✓	.	✓	✓	.	✓	.	.	.	.
Kim; 2007 [155]	?	.	.	?	.	.	.	.	.	.	Hit rate	.	.
Ratzinger; 2007 [238]	✓	.	.	✓	✓	.	.	.	.	.	AAE and MSE	.	.
Mizuno; 2007 [195]	.	.	.	✓	.	✓	✓	✓	.	.	.	.	.
Weyuker; 2007 [283]	.	.	✓	?	.	.	.	.	.	.	% faults	.	.
Zimmermann; 2007 [310]	?	.	✓	✓	✓	✓	✓	✓	.	.	.	.	.
Moser; 2008 [204]	✓	.	.	✓	.	✓	✓	✓	.	.	Cost	.	.
Kamei; 2008 [140]	.	.	✓	✓	.	✓	✓	.	✓	.	.	.	.
Zimmermann; 2008 [304]	.	.	.	✓	✓	.	.	.	.	.	.	✓(on training models)	.
Zimmermann; 2008 [305]	.	.	.	✓	✓	✓	✓	.	.	.	.	✓(on training models)	.

Paper	Cross Validation				Predictive Power							Explanative Power	
	10-fold	**Cross-project**	**Cross-release**	**Training and testing data**	**Correlation**	**Precision**	**Recall**	**Accuracy**	**F-measure**	**AUC**	**Other**	R^2	**Deviance Explained**
Nagappan; 2008 [214]	.	.	.	✓	✓	✓	✓	.	.	.	.	.	.
Zhang; 2008 [298]	.	.	.	.	.	.	.	.	.	.	ARE	.	.
Pinzger; 2008 [230]	.	.	.	✓	✓	✓	✓	.	.	✓	ROC	✓	.
Lessmann; 2008 [170]	.	.	.	✓	.	.	.	.	.	✓	.	.	.
Watanabe; 2008 [280]	✓	✓	✓	✓	✓	✓	✓	.	.	.	.	.	.
Kim; 2008 [153]	✓	.	.	✓	✓	✓	✓	✓	✓	.	.	.	.
Jiang; 2008 [131]	✓	.	.	✓	.	.	.	.	.	✓	ROC	.	.
Weyuker; 2008 [284]	.	.	✓	✓	.	.	.	.	.	.	% faults	.	.
Jiang; 2008 [133]	.	.	.	?	.	.	.	.	.	.	Cost curves	.	.
Tosun; 2008 [273]	?	.	.	✓	.	.	.	.	.	.	POD; PFA and balance	.	.
Koru; 2008 [160]	.	.	.	.	✓	.	.	.	.	.	% faults	.	.
Menzies; 2008 [192]	?	.	.	✓	.	.	.	.	.	.	POD; PFA and balance	.	.
Layman; 2008 [168]	.	.	.	✓	✓	.	.	✓	.	.	.	.	✓
Goronda; 2008 [103]	?	.	.	✓	.	.	.	✓	.	.	.	.	.
Vandecruys; 2008 [275]	?	.	.	✓	.	.	.	✓	.	.	Specificity; Sensitivity	.	.
Elish; 2008 [77]	✓	.	.	✓	.	✓	✓	✓	✓	.	.	.	.
Ratzinger; 2008 [239]	?	.	.	✓	.	✓	.	.	.	.	.	.	.
Meneely; 2008 [189]	.	.	✓	✓	✓	.	.	.	.	.	% faults	.	.
Wu; 2008 [293]	.	.	✓	✓	✓	.	.	.	.	.	% faults	.	.
Tarvo; 2008 [266]	?	.	.	✓	.	✓	✓	.	.	✓	False positive rate; ROC	.	.
Holschuh; 2009 [124]	.	.	✓	✓	✓	✓	✓	.	.	.	.	✓	.
Jia; 2009 [129]	?	.	?	?	.	.	.	.	.	✓	.	.	.
Shin; 2009 [257]	.	.	✓	✓	✓	.	.	✓	.	.	.	.	.
Mende; 2009 [185]	.	.	✓	✓	.	✓	✓	✓	.	✓	False positives and false negatives	.	.
Binkley; 2009 [40]	?	.	.	✓	.	.	.	.	.	.	.	✓	.
D'Ambros; 2009 [62]	.	.	.	✓	✓	.	.	.	.	.	.	✓	.
Hassan; 2009 [116]	.	.	.	✓	.	.	.	.	.	.	Error	✓	.
Bird; 2009 [43]	.	.	✓	✓	.	✓	✓	.	✓	✓	Nagelkerke coef	.	.
Ferzund; 2009 [92]	✓	.	.	✓	.	✓	✓	✓	✓	.	.	.	.
Liu; 2010 [176]	?	.	.	✓	.	.	.	.	.	.	Type I; TypeII errors	.	.
Erika; 2010 [80]	.	.	.	?	.	.	.	.	.	.	Specificity; Sensitivity; Correctness	.	.

	Cross Validation				Predictive Power							Explanative Power	
Paper	**10-fold**	**Cross-project**	**Cross-release**	**Training and testing data**	**Correlation**	**Precision**	**Recall**	**Accuracy**	**F-measure**	**AUC**	**Other**	R^2	**Deviance Explained**
Zhou; 2010 [303]	✓	.	✓	✓	✓	.	.	.	.	✓	.	✓	.
Mockus; 2010 [197]	.	.	.	.	.	.	.	.	.	.	Effect sizes	.	.
Kamei; 2010 [138]	✓	.	✓	✓	.	.	.	.	.	.	% faults per LOC	.	.
Meneely; 2010 [188]	✓	✓	.	✓	✓	✓	✓	.	✓	.	Inspection rate	.	.
Nguyen; 2010 [218]	✓	.	.	✓	✓	✓	✓	.	.	.	.	✓	.
Shihab; 2010 [254]	✓	.	.	✓	✓	✓	✓	.	✓	.	Odds ratios	.	✓
Menzies; 2010 [191]	?	.	?	?	.	✓	.	✓	.	✓	POD; POA	.	.
Weyuker; 2010 [286]	?	.	.	✓	✓	.	.	.	.	.	% Faults in 20% files; FPA	.	.
Mende; 2010 [184]	✓	.	.	?	.	✓	✓	.	.	.	% faults	.	.
Nugroho; 2010 [221]	✓	.	.	✓	.	✓	✓	✓	.	.	Specificity; FP rate; FN rate; odds-ratios	.	.
Song; 2011 [261]	✓	.	.	✓	.	.	.	.	.	✓	ROC; PD; PF	.	.
Nguyen; 2011 [219]	✓	.	.	✓	✓	.	.	.	.	.	.	✓	.
Mende; 2011 [186]	✓	✓	✓	✓	.	.	✓	.	.	✓	Cost Effectiveness; DDR; avg. % faults	.	.
Lee; 2011 [169]	✓	.	.	✓	✓	✓	✓	.	✓	.	AAE; root mean squared error	.	.
Shihab; 2011 [255]	✓	.	.	✓	✓	✓	✓	.	.	.	.	.	✓
D'Ambros; 2011 [61,63]	?	.	?	✓	✓	.	.	.	.	✓	.	.	.
Kpodjedo; 2011 [166]	.	.	✓	✓	.	.	.	.	✓	.	% faults	✓	✓
Bird; 2011 [44]	?	.	.	✓	✓	✓	✓	.	✓	.	.	✓	.
Meneely; 2011 [187]	?	.	.	.	✓	.	.	.	.	.	.	✓	.
Giger; 2011 [101]	?	.	.	✓	✓	✓	✓	.	.	✓		.	✓
Percentage of papers	**30**	**5**	**29**	**77**	**39**	**34**	**31**	**23**	**13**	**15**	**-**	**17**	**12**

2.2.6 Other Considerations

We also set a few of our own criteria for each SDP paper such considering collinearity and considering effort. Table 2.7 lists the criteria for each paper. We explain these characteristics below:

- **Consider Collinearity**: Collinearity is caused when many independent variables are included in a prediction model. Having highly correlated factors in a single model, can make it difficult to determine which factors are actually causing the effect being observed and introducing high variance to the corresponding coefficients [55]. Therefore, collinear (i.e., highly correlated) independent variables need to be removed. We record papers that consider collinearity.

- **Control Variables**: When proposing new factors for SDP, often one has to control for things like size or age. If control variables are not used, it can be difficult to determine if the observed outcome is true. For example, if we want to know whether files that frequently change have lower quality. In addition to considering the number of changes to a file, we need to control for size of the file since it is expected that larger files will have more changes. We record whether or not the SDP work uses control variables.

- **Considers Effort**: More recently, there has been a push for SDP work to consider the effort required to address the predicted software artifacts. We record whether or not a SDP paper considers effort in their prediction models.

- **Effort to Extract Factors**: Practically speaking, there is a cost associated with extracting factors. Therefore, we record whether or not SDP papers consider the effort required to extract the factors used in the SDP models.

- **Considers Impact/Severity**: One of the main criticism of SDP work today is the fact that it does not consider impact or severity of the defects. Therefore, we record papers the consider impact or severity when making predictions.

- **Sought Practitioner Feedback**: Since the ultimate goal of SDP work is to help practitioner, we record which SDP papers actually show or discuss their implementation with practitioners to get their feedback.

Tools

We also record the tools used to build and validate the prediction models in SDP papers. The main reason for doing so is to better understand what tools are most prominently used for SDP and to know which default settings are being used. Many of the prediction models used in SDP work require parameters to be passed in, which are often set as default. Knowing which tool is used helps readers know which parameters values were possibly used.

We find that the majority of SDP work uses WEKA or R. WEKA [113] is a tool developed by the Machine Learning Group at University of Waikato. Weka is a collection of machine learning algorithms for data mining tasks that can be applied directly to a dataset. Weka is used in SDP research since it contains functionality for (amongst others) data pre-processing, classification, regression, and visualization. R is an open source language and environment for statistical analysis [5]. R provides a wide variety of statistical (linear and nonlinear modelling, classical statistical tests and classification). It also provides support for graphical techniques, and is highly extensible. A few SDP papers use SAS/STAT [6], which is another commercial statistical tool that is similar to R.

Research trends: Table 2.7 shows other considerations that affect SDP. In terms of the tools used to perform the prediction, we find that 18% of studies use WEKA and 7% use R. Other commercial tools such as SAS are less commonly used. We find that the majority of the papers do not report the tools used in their studies. It is important to mention the used tools, since generally many of the prediction algorithms have parameters that need to be set. Knowing the used tools can help with determine these parameters.

We also look for other challenges that need to be considered in SDP studies. We find that 37% of studies consider collinearity, 21% control for factors such as size, 16% consider effort when making their predictions, only 5% consider or mention the effort required to extract the factors used, 4% consider the severity of the defects and 18% of studies sought practitioner feedback regarding their study (i.e., either mention or discuss feedback from practitioners).

Table 2.7: Other considerations of SDP studies. An '✓' means applied, a '.' means not applied and '?' means could not determine.

	Tools			Other Considerations					
Paper	**WEKA**	**R**	**Other**	**Consider Collinearity**	**Control Variables**	**Considers Effort**	**Effort to Extract Factors**	**Considers Impact/Severity**	**Considers Practicality**
Yuan; 2000 [296]	.	.	?	.	.	.	.	.	.
Cartwright; 2000 [53]	?	?	.	✓	.	.	✓	.	.
Neufelder; 2000 [216]	.	.	?	.	.	.	.	.	✓
Khoshgoftaar; 2000 [147]	.	.	?	✓	.	.	.	.	.
Khoshgoftaar; 2000 [148]	.	.	?	✓	.	.	.	.	.
Morasca; 2000 [203]	.	.	?	.	.	.	.	.	.
Wong; 2000 [291]	.	.	?	.	.	.	.	.	.
Fenton; 2000 [89]	.	.	?	?	.	.	.	.	.
Graves; 2000 [105]	.	.	?	?	?	.	.	.	.
Khoshgoftaar; 2001 [142]	.	.	?	.	.	.	.	.	.
El Emam; 2001 [78]	.	.	?	✓	✓	.	.	.	.
Denaro; 2002 [66]	.	.	?	✓	.	.	✓	.	.
Briand; 2002 [50]	.	.	?	✓	✓	.	.	.	✓
Khoshgoftaar;2002 [146]	.	.	.	✓	.	.	.	.	.
Quah; 2003 [232]	.	.	?	✓	.	.	.	.	.
Khoshgoftaar; 2003 [144]	.	.	?	✓	.	.	.	.	.
Guo; 2003 [106]	.	.	SAS	✓	.	✓	.	.	.
Amasaki; 2003 [12]	?	?	?	?	.	.	.	.	.
Succi; 2003 [264]	.	.	?	?	.	.	.	.	✓
Guo; 2004 [107]	✓	✓	See5/C5	?	.	.	.	.	.
Li; 2004 [175]	.	.	?	.	.	.	.	.	.
Ostrand; 2004 [225]	.	.	SAS/STAT	.	✓	.	.	.	.
Koru;2005 [163]	✓	✓	.	.	.	.	.	.	.
Gyimothy; 2005 [109]	.	.	?	✓	.	.	✓	.	.
Mockus; 2005 [202]	.	.	?	?	.	.	.	✓	✓
Nagappan; 2005 [209]	.	.	.	.	.	.	.	.	✓
Hassan; 2005 [117]	.	.	.	.	.	.	.	.	✓
Nagappan; 2005 [210]	.	.	?	✓	✓	.	.	.	✓
Tomaszewski;2006 [270]	.	.	.	.	✓	.	.		.
Pan; 2006 [227]	.	.	?	.	.	.	.	.	.
Nagappan; 2006 [212]	.	.	?	✓	.	.	.	.	✓
Zhou; 2006 [302]	✓	.	.	✓	.	.	..	✓	.
Li; 2006 [173]	.	.	?	?	.	.	.	.	✓

	Tools			**Other Considerations**					
Paper	**WEKA**	**R**	**Other**	**Consider Collinearity**	**Control Variables**	**Considers Effort**	**Effort to Extract Metrics**	**Considers Impact/Severity**	**Considers Practicality**
Knab; 2006 [159]	✓	.	.	.	.	.	.	.	.
Arisholm; 2006 [19]	.	.	?	✓	.	✓	.	.	✓
Tomaszewski;2007 [271]	.	.	.	✓	✓	✓	.	.	✓
Ma; 2007 [178]	✓	.	.	✓	.	.	.	.	.
Menzies; 2007 [190]	✓	.	.	?	.	.	.	.	.
Olague; 2007 [223]	.	.	?	✓	.	.	.	.	.
Bernstein; 2007 [34]	✓	.	.	✓	✓	.	✓	.	.
Aversano; 2007 [23]	✓	.	.	NA	.	.	.	.	.
Kim; 2007 [155]	.	.	.	.	.	.	.	.	.
Ratzinger; 2007 [238]	.	.	?	?	✓	.	.	.	.
Mizuno; 2007 [195]	.	.	.	NA	.	.	.	.	.
Weyuker; 2007 [283]	.	.	.	?	✓	.	.	.	.
Zimmermann; 2007 [310]	.	✓	.	.	.	.	.	.	.
Moser; 2008 [204]	?	?	?	.	.	✓	.	.	.
Kamei; 2008 [140]	.	.	.	.	.	.	.	.	.
Zimmermann; 2008 [304]	.	.	?	✓	.	.	.	.	.
Zimmermann; 2008 [305]	?	?	?	✓	.	.	.	.	.
Nagappan; 2008 [214]	.	.	?	✓	?	.	.	.	.
Zhang; 2008 [298]	.	.	?	.	.	.	.	.	.
Pinzger; 2008 [230]	.	.	?	✓	.	.	.	.	.
Lessmann; 2008 [170]	.	.	.	?	.	.	.	.	.
Watanabe; 2008 [280]	✓	.	.	.	.	.	.	.	.
Kim; 2008 [153]	?	?	?	.	.	.	.	.	✓
Jiang; 2008 [131]	✓	.	.	?	.	.	.	.	.
Weyuker; 2008 [284]	.	.	?	?	?	.	.	.	.
Jiang; 2008 [133]	✓	.	.	.	.	✓	.	.	.
Tosun; 2008 [273]	?	?	?	?	.	✓	NA	.	.
Koru; 2008 [160]	.	.	.	NA	.	✓	NA	.	.
Menzies; 2008 [192]	✓	.	.	?	.	.	.	.	.
Layman; 2008 [168]	.	.	?	✓	.	.	.	.	.
Goronda; 2008 [103]	.	.	?	✓	.	.	.	.	.
Vandecruys; 2008 [275]	.	.	?	?	.	.	.	.	✓
Elish; 2008 [77]	✓	.	.	✓	.	.	.	.	.
Ratzinger; 2008 [239]	✓	.	.	?	.	.	.	.	.
Meneely; 2008 [189]	?	?	?	✓	✓	.	.	.	.
Wu; 2008 [293]	?	?	?	?	✓	.	.	.	.

	Tools			Other Considerations					
Paper	**WEKA**	**R**	**Other**	**Consider Collinearity**	**Control Variables**	**Considers Effort**	**Effort to Extract Metrics**	**Considers Impact/Severity**	**Considers Practicality**
Tarvo; 2008 [266]	.	?	?	✓	.	.		.	.
Holschuh; 2009 [124]	?	?	.	✓	.	.	.	.	✓
Jia; 2009 [129]	?	?	?	?	.	.	.	.	.
Shin; 2009 [257]	?	?	?	✓	✓	.	.	.	.
Mende; 2009 [185]	.	.	.	?	?	✓	.	.	.
Binkley; 2009 [40]	.	.	?	?	✓	.	.	.	.
D'Ambros; 2009 [62]	.	.	?	✓	?	.	.	✓	.
Hassan; 2009 [116]	?	?	?	?	.	.	.	.	.
Bird; 2009 [43]	.	.	?	✓	?	.	.	.	.
Ferzund; 2009 [92]	?	?	?	?	.	.	.	.	.
Liu; 2010 [176]	✓	.	.	?	.	.	.	.	.
Erika; 2010 [80]	?	?	?	?	.	.	.	.	.
Zhou; 2010 [303]	✓	.	.	?	✓	.	.	.	.
Mockus; 2010 [197]	.	.	?	?	✓	.	.	.	.
Kamei; 2010 [138]	.	✓	.	?	NA	✓	.	.	✓
Meneely; 2010 [188]	.	.	SAS	.	.	.	.	.	.
Nguyen; 2010 [218]	?	?	?	.	✓	✓	.	.	✓
Shihab; 2010 [254]	.	✓	.	✓	✓	.	✓	.	.
Menzies; 2010 [191]	?	?	?	?	?	✓	.	.	.
Weyuker; 2010 [286]	?	?	?	?	✓	.	.	.	.
Mende; 2010 [184]	?	?	?	?	✓	✓	.	.	.
Nugroho; 2010 [221]	.	.	SPSS	?	.	✓	.	.	.
Song; 2011 [261]	?	?	?	?	?	.	.	.	.
Nguyen; 2011 [219]	?	?	?	NA	?	.	.	.	.
Mende; 2011 [186]	.	✓	.	?	?	✓	.	.	✓
Lee; 2011 [169]	✓	.	.	?	.	.	.	.	.
Shihab; 2011 [255]	.	✓	.	✓	✓	✓	.	✓	✓
D'Ambros; 2011 [61,63]	?	?	?	✓	.	✓	.	?	.
Kpodjedo; 2011 [166]	?	?	?	?	.	.	.	.	.
Bird; 2011 [44]	?	?	?	✓	✓	.	.	.	.
Meneely; 2011 [187]	?	?	?	.	.	.	.	.	.
Giger; 2011 [101]	✓	.	SPSS	✓	.	.	.	.	.
Percentage of papers	**18**	**7**	**-**	**37**	**21**	**16**	**5**	**4**	**18**

2.3 Critical Evaluation and Open Issues

Thus far, we have characterized the state-of-the-art in SDP research. In this section, we evaluate the surveyed SDP work from a pragmatic point of view (i.e., how easily the work can be applied in practice, with special focus on whether the work considers impact and provides guidance on how to use the results) and point out the challenges of current SDP work.

2.3.1 Data

With respect to data used in SDP studies, we find that most studies use commercial data, however the number of SDP studies using OSS data is steadily increasing. The most popular sources for SDP studies are source code and defect repositories. The majority of the SDP studies today are done on Java and C/C++ systems. In addition, most of these studies are done at the subsystem or file level. These findings raise a few challenges that we believe are worth further investigation:

Challenge 1. Commercial vs. OSS Data: 69% of the surveyed SDP studies use commercial data. However, recently, the widespread of OSS data has enabled SDP researchers to perform their studies using OSS data as well. One challenge with using commercial data is that researchers are often not able to release the data to the public. This hinders the chances of reproducing the work and building on it. Recently, there have been initiatives to have the research community share their data through the PROMISE repository [4] and the MSR challenge [7]. Moving forward, we believe that researchers need to investigate ways in which commercial data can be shared, while maintaining privacy so this commercial data can be shared.

Challenge 2. Replicability: The replicability of SDP studies is a challenge that is commonly discussed within the SDP research community. Therefore, researchers are being encouraged to share their data in order to facilitate the replicability of their results. For example, the PROMISE repository has been setup for researchers to share their data and analysis scripts. We believe that future SDP studies should do their best to share their data sets in order to further facilitate the replicability of their SDP research and advance the field as a whole.

Challenge 3. Data Sources: Since the large majority of SDP studies use repository data (i.e., 75% from source code and 69% from defect repositories), it is worth investigating the quality of the data stored in software repositories. As mentioned earlier, researchers have have already begun looking in this direction (e.g., [25, 41, 154, 217]). We believe that this line of research is very important for future SDP work and encourage researchers to further investigate what other types of data can be used to mitigate challenges related to noise in repository data.

Challenge 4. Level of Granularity: The majority of studies today are performed at the subsystem (i.e., 49% of papers) or file (i.e., 51% of papers) levels. The main reason for this is that repository data is often given at the file level and can be easily abstracted to the subsystem level. Although performing predictions at the subsystem and file levels may lead to better results [138, 218], the usefulness of the SDP work becomes less significant (i.e., since more code would need to be inspected at high levels of abstraction). We believe that future SDP work should focus more on performing predictions at a finer level of granularity, e.g., at the function level. We believe that designers of future software repositories should enable these repositories with features that allow for data to be recorded at a fine granularity. Furthermore, granularity should be taken into account when evaluating the performance of SDP methods.

Challenge 5. Programming Languages: The majority of the surveyed SDP studies are performed on projects written in object oriented programming languages such as Java (i.e., 43% of papers) and C/C++ (i.e., 46% of papers). The main reason for doing so is the availability of data and wide use of object oriented programming languages. However, very few SDP studies are done using projects that use other programming languages such as scripting languages (e.g., Perl) or markup languages (e.g., HTML). In the future, we would like to see SDP research investigate differences or similarities that may arise when non-traditional programming languages are used.

2.3.2 Factors

From our survey, we see that the majority of SDP work today focuses on investigating factors that best predict defects. Such works investigated the use of product, process, deployment and social factors for the purpose of SDP. The work aims to predict pre- and post-release defects and defect density. Although there has been a rich body of work in this area, we see room for future research in:

Challenge 6. Actionable Factors: A plethora of SDP work investigated the usefulness of using various product (i.e., 76% of papers) and process (i.e., 45% of papers) factors in predicting pre- and post-release defects. Although this work is very important, the main questions being asked now is not what independent variables best predict defects, but how useful and applicable are these factors/independent variables in practice. Moving forward, we believe that the SDP research needs to focus on finding actionable factors. For example, studies showed that size and churn are good predictors, however, it is very difficult to reduce size or churn since software systems need to continually evolve.

Challenge 7. Considering the Impact of Defects: The majority of the work today (i.e., 65% of papers) focuses on predicting post-release defects. Although post-release defects are extremely important and measure the quality of the released software, they are not the only criteria for software quality. Recent work, for example,the work by Shin *et al.* [256] and Zimmermann *et al.* [309] focused on predicting software vulnerabilities since they have high priority. We believe that future research needs to further build on this type of work and take into account the impact or severity of these post-release defects when making the predictions since a documentation defect is not as impacting as a security defect.

2.3.3 Models

Prior research compared the use of a wide variety of models for use in SDP. The used models are based on machine learning techniques such as decision trees and statistical techniques such as logistic regression. Moving forward, we believe that SDP work should:

Challenge 8. Providing Guidance on How to Use Results: Today, the majority of the models used in SDP are highly advanced models that are optimized to achieve high precision and recall. Although precision and recall are very important, the ability of a model to explain why the predictions are being made needs to be considered as well [173]. Some prior work focused on building tools and techniques to increase the adoption of software engineering research in practice [27, 28, 268]. We believe that practitioners are willing to sacrifice a small drop in precision and recall for better understandability and guidance on how to use the results. Making sense of why the models are making their predictions helps get buy-in from practitioners who will use these SDP models.

Challenge 9. Implementing Models in Practice: Another major challenge of current SDP work is its lack to address how such models can be implemented in practice. For example, some models (e.g., models that analyze text) may take long periods of time to produce their predictions. This makes them very difficult to implement in practice since practitioners want quick results that they can act on. Moving forward, we believe that future SDP work should consider and evaluate how easy it is to implement the models being used in practice, not just precision and recall values.

Challenge 10. Feedback Loop: Another area that needs to be better studied in SDP research is how these SDP models are maintained in practice. For example, once a model is implemented, it needs to have a feedback loop that updates it with the changes being made to the software [71]. Moving forward, we believe that more SDP research needs to focus on how the maintenance and updating of SDP approaches once they are implemented in practice.

2.3.4 Performance Evaluation

Our survey of SDP work shows that standard statistical measures such as correlation, precision, recall and model fit (measured in R^2 or deviance explained). However, we believe that moving forward the performance of SDP work should consider more than just statistical measures, it should:

Challenge 11. Considering Effort: The amount of effort required to address the software artifacts flagged by SDP work is an important factor that needs to be added in the performance evaluation criteria. In fact, recent work by Arisholm and Briand [19] use OO and historical metrics to predict fault-prone classes in a large legacy telecommunication system. They show that their models that use historical data can save verification efforts by 20% over a random predictor. Kamei *et al.* [138] evaluate common defect prediction findings when effort is considered. In particular, they investigate whether process factors outperform product factors and whether package-level prediction outperform file-level prediction. They perform their case study on three open source projects Eclipse Platform, JDT and PDE and find that process factors still outperform product factors when effort is considered, however, package-level prediction do not outperform file-level predictions when effort is considered. Menzies *et al.* [191] argue that recent studies have not been able to improve defect prediction results since their performance is measured as a tradeoff between the probability of false alarms and probability of detection. Therefore, they suggest changing the standard goal to consider effort, i.e., to finding the smallest set of modules that contain most of the errors. Using static code factors and propose the use of the WHICH meta-learner framework, the authors show that WHICH outperforms all data mining methods studied by them, including Naive Bayes, decision trees and RIPPER. Mende and Koschke [184] compare two strategies to include the effort treatment into defect prediction models. One strategy is applicable to any probabilistic classifier and the other applicable only for regression algorithms. They perform a case study using 15 projects using random forests and show that both strategies improve the predictive performance.

Challenge 12. Identifying Practical Performance Measures: Till now, there has been very little research to examine what practitioners consider useful when evaluating SDP work. We believe that practitioners care about more than just precision and recall [173]. Therefore, moving forward we believe that studies (e.g., such as the study by Mende *et al.* [186]) should be conducted to find out what practitioners actually care about when evaluating SDP work and use these measures to re-evaluate prior SDP work.

2.3.5 Focus of This Thesis

In this thesis, we address two of the aforementioned challenges and argue that SDP needs to be more encompassing and proactive. In particular, we focus on addressing **challenge 7**,which calls for SDP research to consider impact when making predictions, and **challenge 8**, which deals with the lack of current research to provide guidance on how to make use of SDP results.

We chose to address these two challenges in our thesis since we believe they are two of the most important challenges facing pragmatic SDP today. This believe is driven by:

1. Our industrial experience, working as a SQA specialist and an embedded SQA researcher with Research In Motion.

2. Our numerous discussions with practitioners and researchers from both, industry and the Software Engineering research community.

To deal with these challenges, we propose approaches that predict high-impact defects. These approaches show how current SDP approaches can be tailored consider impact in their predictions. In addition, we propose approaches that simplify SDP models so they can be easily understood and applied in practice.

In addition, the majority of the current SDP research only focuses on defects and is reactive in nature (i.e., it assumes that defects exist in the code and aims to identify this defective code). We believe that organizations are interested in more than just defects, they are interested in risk, which is more encompassing than defects. Therefore, we propose an approach that shows how SDP research can be more encompassing and proactive.

2.4 Concluding Remarks

In this Chapter we surveyed and characterized SDP papers from the year 2000 - 2011 to determine trends and avenues for future work. The papers are characterized along five dimensions: 1) the data sources and granularity, 2) factors, 3) models, 4) performance evaluation, and 5) other considerations. We also provided brief summaries of each paper. The main findings of the survey are:

- The majority of SDP papers rely on source code and defect repositories. Approximately 70% of SDP studies use commercial data and 37% use OSS data. NASA and Eclipse are the two most commonly used datasets in SDP studies.

- The majority of SDP papers use (76%) product and (45%) process factors as their independent variables. The vast majority (65%) predict post-release defects.
- The majority of SDP studies use logistic regression (47%), linear regression (22%) and decision trees (26%) to build their prediction models. Other techniques such as naive bays and random forests are also used, but less common.
- To evaluate the performance of their built models, 39% used correlations, 34% used precision and 31% used recall.

The findings of this survey help us identify the state-of-the-art and the current challenges of SDP research. We present approaches that aim to tackle two of the challenges that hinder the adoption of SDP research in practice. Then, we propose an approach that demonstrates how SDP research can be more encompassing, by focusing on risk and not only defects, and proactive (i.e., by predicting risky changes as they are commited). The thesis is divided into three parts:

- In Part I we present approaches that study and predicts high-impacting defects. Our work illustrates how SDP approaches can be tailored to consider the impact of defects.
- In Part II we present an approach that simplifies prediction models so they can be easily understood and acted upon by practitioners. Furthermore, we present an approach that uses the development history to prioritize the creation of unit tests in large software systems. Our work illustrates how to make SDP results more applicable in practical settings.
- In Part III we present an approach to identify risky code changes. Our work illustrates how SDP approaches can be more encompassing (i.e., by considering risk and not only defects) and proactive (i.e., by identifying risky code changes before they are incorporated into the code base).

Part I

Considering the Impact of Defects

A plethora of work on SDP focuses on predicting pre-release and post-release defects. However, the adoption of this work in practice is limited [102, 115]. One of the reasons for this adoption is the fact that prior work does not take into account the impact of the defects when making predictions.

This Part of the thesis presents two approaches that show how SDP work can be tailored to predict high-impact defects:

- **Studying Breakage and Surprise Defects [Chapter 3]:** We present an approach that focuses on predicting two types of high-impact defects, breakages and surprises. Breakages are defects that break functionality that customers are used to, highly impact customers. On the other hand, surprises are defects that appear in unexpected locations. We show how factors extracted from a project's history, stored in commonly used software repositories, can be used to effectively predict these high-impact defects. We compare our approach to state-of-the-art approaches (that rarely consider impact) and show that focusing our predictions on breakage and surprise defects can reduce the amount of code to be reviewed by up to 30%.

  The main recommendations based on the findings of this Chapter are:

  - Practitioners need to consider both, breakage and surprise defects, separately since they are rare, unique and different. Surprise defects have high severity and appear to indicate problems in the requirements.
  - Using specialized defect prediction models can effectively predict breakage and surprise defects, yielding sizeable effort savings over using simple post-release defect prediction models.
  - Traditional defect prediction factors (i.e., the number of pre-release defects and file size) are good predictors of breakage defects. However, the number of co-changed

files, the size of recently co-changed files and the time since the last change should be used to predict surprise defects.

- **Studying Re-opened Defects [Chapter 4]:** We present an approach that studies and predicts re-opened defects, i.e., defects that are more likely to be re-opened after they are addressed. We use factors related to: the work habits of the developers, the defect report, the fix of the defect and the personnel who fixed the defect to perform our prediction. We find that using a small number of factors related to the defect report can achieve a precision between 49.9-78.3% and a recall in the range of 72.6-93.5% when predicting whether a defect will be re-opened.

  The main recommendations based on the findings of this Chapter are:

  – The occurrence of re-opened defects should be minimized since they take considerably longer to resolve.

  – Practitioners can leverage decision tree prediction models to accurately predict re-opened defects. Predicting re-opened defects in three different projects, we were able to achieve a precision between 49.9-78.3% and a recall in the range of 72.6-93.5%.

  – The factors that best indicate re-opened bugs vary based on the project. The comment text is the most important factor for the Eclipse and OpenOffice projects, while the last status is the most important one for Apache. All of these factors can be extracted from the bug reports.

This Part of the thesis is likely to be of interest to software practitioners and researchers. Practitioners can see how simple approaches, using data that is widely available to them today, can be used to help them prioritize their SQA efforts. Researchers can use our approach as an example to tailor their approaches to consider impact when predicting defects. Furthermore, practitioners and researchers can use our approaches to better understand what factors are the most important in predicting high-impact defects.

Chapter 3

Studying Breakage and Surprise Defects

The relationship between various software-related phenomena (e.g., code complexity) and post-release software defects has been thoroughly examined. However, to date SDP has limited adoption in practice. The most commonly cited reason is that the prediction identifies too much code to review without distinguishing the impact of these defects. Our aim is to address this limitation by focusing on two types of high-impact defects for customers and practitioners. Customers are highly impacted by defects that break pre-existing functionality (breakage defects), whereas practitioners are caught off-guard by defects in files that had relatively few pre-release changes (surprise defects). The large commercial software system that we study already had an established concept of breakages as the highest-impact defects, however, the concept of surprises is novel and not as well established. We find that surprise defects are related to incomplete requirements and that the common assumption that a fix is caused by a previous change does not hold in this project. We then fit prediction models that are effective at identifying files containing breakages and surprises. The number of pre-release defects and file size are good indicators of breakages, whereas the number of co-changed files and the amount of time between the latest pre-release change and the release date are good indicators of surprises. Although our prediction models are effective at identifying files that have breakages and surprises, we learn that the prediction should also identify the nature or type of defects, with each type being specific enough to be easily identified and repaired.

3.1 Introduction

Work on defect prediction aims to assist practitioners in prioritizing software quality assurance efforts [61]. Most of this work uses code measures (e.g., [212, 310]), process measures (e.g., [105]), and social structure measures (e.g., [55]) to predict source code areas (i.e., files) where the post-release defects are most likely to be found. The number of pre-release changes or defects and the size of an artifact are commonly found to be the best indicators of a file's post-release defect potential.

Even though some studies showed promising performance results in terms of predictive power, the adoption of defect prediction models in practice remains low [102, 115, 253]. One of the main reasons is that the amount of code predicted to be defect-prone far exceeds the resources of the development teams considering inspection of that code. For example, Ostrand *et al.* [225] found that 80% of the defects were in 20% of the files. However, these 20% of the files accounted for 50% of the source code lines. At the same time, all defects are considered to have the same negative impact, which is not realistic, because, for example, documentation defects tend to be far less impacting than security defects. At the same time, model-based prediction tends to indicate the largest and the most changed files as defect-prone: areas already known to practitioners to be the most problematic.

In this chapter, we address the problem of predicting too many defects by focusing the prediction on the limited subset of *high-impact defects*. Since impact has a different meaning for different stakeholders, we consider two of the possible definitions in this chapter: *breakages* and *surprises*. Breakages are a commonly accepted concept in industry that refer to defects that break functionality delivered in earlier releases of the product on which customers heavily rely in their daily operations. Such defects are more disruptive because 1) customers are more sensitive to defects that occur in functionality that they are used to than to defects in a new feature and 2) breakages are more likely to hurt the quality image of a producer, thus directly affecting its business. To ensure that we focus only on the highest impact defects, we only consider breakages that have severities “critical” (the product is rendered non-operational for a period of time) and “high” (the product operation has significant limitations negatively impacting the customer’s business).

Whereas breakages have a high impact on customers, surprise defects are a novel concept representing a kind of defects that highly impact practitioners. Surprises are defects that appear in unexpected locations or locations that have a high ratio of post-to-pre release defects, catching practitioners off-guard, disrupting their already-tight quality assurance schedules. For example, post-release defects in files that are heavily changed or have many defects prior to the release are expected and scheduled for. However, when a defect appears in an unexpected file, the workflow of developers is disrupted, causing them to shelf their current work and shift focus to addressing these surprises.

The investigated project has used the concept of breakages for several decades but surprise defects are a new concept. Hence, we start by comparing the properties of breakage and surprise defects and qualifying the impact of surprises. We then build prediction models for breakages and surprise defects (RQ1), identify which factors are the best predictors for each type of defect (RQ2) and quantify the relative effect of each factor on the breakage and surprise-proneness (RQ3). Finally, we calculate the effort savings of using specialized defect prediction models and perform a qualitative evaluation of the usability of the prediction models in practice and propose ways to make such prediction more relevant.

In this chapter, we make the following contributions:

- **Identify and explore breakage and surprise defects.** We find that breakage and surprise defects represent approximately one-fifth of the post-release defects. Only 6% of the files have both types of defects. For example, breakages tend to occur in locations that have experienced more defect fixing changes in the past and contain functionality that was implemented less recently than the functionality in locations with surprise defects.

- **Develop effective prediction models for breakage and surprise defects.** Our models can identify future breakage and surprise files with more than 69% recall and a two to three fold increase of precision over random prediction.

- **Identify and quantify the major factors predicting breakage and surprise defects.** Traditional defect prediction factors (i.e., pre-release defects and size) have a strong positive effect on the likelihood of a file containing a breakage, whereas the co-changed files and time-related factors have a negative effect on the likelihood of a file containing a surprise defect.

- **Measure the effort savings of specialized prediction models.** Our custom models reduce the amount of inspected files by 3-30%, which represents a 21-24% reduction in the number of inspected lines of code.

- **Propose areas and methods to make defect prediction more practical.** A qualitative study suggests that an important barrier to the use of prediction in practice is lack of indications about the nature of the problem or the ways to solve it. The method to detect surprise defects may be able to highlight areas of the code that have incorrect requirements. We propose that an essential part of defect prediction should include prediction of the nature of the defect or ways to fix it.

3.1.1 Organization of Chapter

Section 3.2 highlights the related work. Section 3.3 compares the properties of breakages and surprise defects. Section 3.4 outlines the case study setup and Section 3.5 presents our case study results. Section 3.6 discusses the effort savings provided by the specialized models built in our study. Section 3.7 discusses the threats to validity of our study. Section 3.8 reflects on the lessons learned about the practicality of defect prediction. We conclude the Chapter in Section 3.9.

3.2 Related Work

We survey the state-of-the-art in SDP in Chapter 2. In this section, we discuss the work that is most closely related to this Chapter. Previous work typically builds multivariate logistic regression models to predict defect-prone locations (e.g., files or directories). A large number of previous studies use complexity factors (e.g., McCabe's cyclomatic complexity factor [181] and Chidamber and Kemerer (CK) factors suite [59]) to predict defect-prone locations [30, 109, 209, 222, 263, 310]. However, Graves *et al.* [105], Leszak *et al.* [171] and Herraiz *et al.* [122] showed that complexity factors highly correlate with the much simpler lines of code (LOC) measure. Graves *et al.* [105] argued that change data is a better predictor of defects than code factors in general and showed that the number of prior changes to a file is a good predictor of defects. A number of other studies supported the finding that prior changes are a good defect predictor and additionally showed that prior defect is also a good predictor of post release defects [19, 116, 143, 171, 204, 295]. To sum up, this previous work showed that complexity factors and hence size measured in LOC, prior change and prior defects are a good predictor of defect-proneness. The focus of the aforementioned work was to improve the prediction performance by enriching the set of factors used in the prediction model. In this work, we incorporate the findings of previous work by using traditional defect prediction metrics/factors, in addition to other more specialized factors related to co-change and time properties to predict the location of highly impacting defects. However, we would like to note that our focus here is capturing high-impact defects rather than adding factors to improve the performance of defect prediction models in general.

Although it has been shown that defect prediction can yield benefit in practice, its adoption remains low [102, 115]. As Ostrand *et al.* [225, 226] showed, 20% of the files with the highest number of predicted defects contain between 71-92% of the defects, however these 20% of the files make up 50% of the code. To make defect prediction more appealing to practice, recent work examined the performance of software prediction models when the effort required to address the identified defect is considered (i.e., effort-aware defect prediction) [22, 138, 183]. Although effort is taken into account, these models still predict the entire set of post-release defects, giving each defect equal impact potential.

Other work focused on reducing the set of predicted defects based on the semantics of the defect. For example, Shin *et al.* [256] and Zimmermann *et al.* [309] focused on predicting software vulnerabilities since they have high priority. Instead of narrowing down the set of defects vertically based on the semantics of the defects, we narrow down the defects horizontally across domains. For example, our models can predict high impact defects across many domains, whereas a model focused on vulnerability defects is only useful for one domain.

3.3 Breakage and Surprise Defects

In this section, we provide background of the software project under study, and define breakages and surprise defects. We then characterize and compare breakage and surprise defects.

3.3.1 Background

Software Project: The data used in our study comes from a well-established telephony system with many tens of thousands of customers that was in active development for almost 30 years and, thus, has highly mature development and quality assurance procedures. The present size of the system is approximately seven million non-comment LOC, primarily in C and C++. The data used in our study covers five different releases of the software system.

Change Data: There are two primary sources of data used. Sablime, a configuration management system, is used to track Modification Requests (MR), which we use to identify and measure the size of a software release and the number of defects (pre-release and post-release). When a change to the software is needed, a work item (MR) is created. MRs are created for any modification to the code: new features, enhancements, and fixes. The project uses internally developed tools on top of the Source Code Control System (SCCS) to keep track of changes to the code. Every change to the code has to have an associated MR and a separate MR is created for different tasks. We call an individual modification to a single file a delta. Each MR may have zero or more deltas associated with it. Since the development culture is very mature, these norms are strictly enforced by peers.

For each MR, we extracted a number of attributes from Sablime and the SCCS: the files the MR touches, the release in which the MR was discovered, the date the MR was reported, the software build where the code was submitted, the resolution date (i.e., when the MR was fixed/implemented), resolution status for each release the MR was submitted to, the severity and priority of the MR, the MR type (e.g., enhancement or problem) and the general availability date of the release that includes the MR.

For each release, we classify all MRs into two types: pre-release changes (or defects if they are type problem), and post-release defects. MRs that are submitted to a release before the General Availability date (we refer to it as GA), are considered to be pre-release defects. Fixes reported for a particular release after the GA date, are considered to be post-release defects.

3.3.2 Defining Breakages and Surprise Defects

Breakage Defects: Defects are introduced into the product because the source code is modified to add new features or to fix existing defects. When such defects break, i.e., cause a fault or change existing functionality that has been introduced in prior releases, we call these defects breakages. The concept of breakages typically is familiar in most companies. For example, in the project studied in this chapter, all severity one and two defects of established functionality are carefully investigated by a small team of experts. In addition to a root-cause analysis and suggestions to improve quality assurance efforts, the team also determines the originating MR that introduced the breakage. While other companies may use a different terminology than "breakages", most investigate such high-impact problems just as carefully, therefore similar data is likely to be available in other projects with mature quality practices.

Surprise Defects: Previous research has shown that the number of pre-release defects is a good predictor of post-release defects (e.g., [204, 310]). Therefore, it is a common practice for software practitioners to thoroughly test files with a large number of pre-release defects. However, in some cases, files that rarely change also have post-release defects. Such defects catch the software practitioners off-guard, disrupting their already-tight schedules. To the best of our knowledge, surprise defects have not been studied yet, and therefore are not recorded in issue tracking systems. Hence, for the purpose of our study, we use one possible definition of surprise. We define the degree to which a file contains surprise defects as $Surprise(file) = \frac{No.\ of\ post\ release\ defects(file)}{No.\ of\ pre\ release\ defects(file)}$, whereas in files without pre-release defects we define it as $Surprise(file) = No.\ of\ post\ release\ defects(file) * \beta$. The choice of β depends on how severe having one or more post-release defects and no pre-release defects is. For example, if such a situation is very severe, then β should be given a very high value (e.g., 100). On the other hand, if such a situation is not very severe then β can be equal to 1. In this study, we set β to equal 2. Our choice was based on our prior experience with defect prediction work. A formal empirical study is needed to determine the optimal value of β.

Because our definition of surprise is given in terms of the ratio of post-to-pre release defects, we need to determine from what threshold the ratio should be considered significant. For example, if a file has one post-release defect and 10 pre-release defects, i.e., the defined surprise value is $\frac{1}{10} = 0.1$, should this file be flagged as being a surprise?

To calculate the surprise threshold, we examine all the files that had a defect reported within one month of the GA date of the previous release. The intuition behind this rule is that high impact defects are likely to be reported immediately after the software is released. Hence, we calculate the median of the ratio of all post-release defects and all pre-release defects for all files changed within one month of the previous GA date, then use this value as the surprise threshold for the next release. For example, the surprise threshold for release 2.1 is determined by the median ratio of post-to-pre-release defects of all files that had a post-release defect reported against them within a month after release 1.1.

3.3.3 Occurrence of Breakage and Surprise Defects

Before analyzing the characteristics of breakage and surprise defects, we examine the percentage of files that have breakages and surprise defects. To put things in context, we also show the percentage of files with one or more post-release defects. Table 3.1 shows that on average, only 2% of the files have breakages or surprise defects. That is approximately one fifth of the files that have post-release defects.

Having a small percentage of files does not necessarily mean less code, since some files are larger than others. Therefore, we also examine the amount of LOC that these files represent. Table 3.1 shows that on average, the amount of LOC that these breakage and surprise files make up is 3.2% and 3.8%, respectively. This is approximately one fourth the LOC of files with the post-release defects. The reduction is both promising, because it narrows down the set of files to be flagged for review, and challenging, because predicting such unusual files is much harder.

Table 3.1 also compares the percentages of MRs representing post-release, breakage and surprise defects. On average, the percentage of breakage and surprise MRs is much smaller than that of post-release MRs. Since we use the surprise threshold from the previous release to determine the surprise threshold, we are not able to calculate surprise defects for the first release (R1.1).

Table 3.1: Percentage of files, LOC and MRs containing post-release, breakage and surprise defects.

Release	**Post-Release**			**Breakage**			**Surprise**		
	Files	**LOC**	**MRs**	**Files**	**LOC**	**MRs**	**Files**	**LOC**	**MRs**
R1.1	21.8	27	78.8	1.6	2.6	29.4	-	-	-
R2.1	6.5	8.4	48.6	2.1	2.6	27.1	0.2	0.4	1.5
R3.0	7.8	12.7	54.5	2.1	3.8	28.4	3.3	6.4	13.6
R4.0	11	18	79.7	1.4	1.3	25.6	3.0	5.2	11.5
R4.1	5.0	6.8	46.1	2.5	2.8	22.4	1.5	3.2	6.1
Average	**10.4**	**14.6**	**61.5**	**2.0**	**3.2**	**26.6**	**2.0**	**3.8**	**8.2**

Breakage and surprise defects are difficult to pinpoint, since they only appear in 2% of the files.

3.3.4 Breakages vs Surprise Defects

As mentioned earlier, breakage defects are high severity defects that break existing functionality. They are a common concept in industry, and their impact is understood to be quite high. However, surprise defects are a concept that we defined based on our own industrial experience. In this section, we would like to learn more about the characteristics of surprise defects and verify our assumption that surprise defects highly impact the development organization, by addressing the following questions:

- Are surprise defects different than breakage defects? If so, what are the differences?
- Do surprise defects have high impact?

Such verification helps us appreciate the value of studying surprise defects and to understand the implications of our findings for future research in the field.

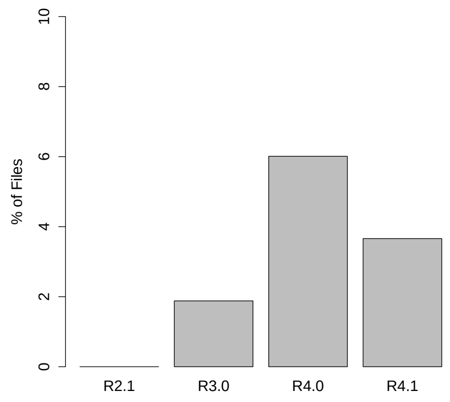


Figure 3.1: Percentage of files containing both, surprise and breakage defects

Are surprise defects different than breakage defects? If so, what are the differences?

First, we look at the locations where breakage and surprise defects occur. If we, for example, find that breakage and surprise defects occur in the same location, then we can assume that the prediction models for breakage-prone files will suffice for surprise defect prediction.

To quantify the percentage of files that contain both types of defects, we divide the files into two sets: files with breakages and files with surprise defects. Then, we measure the intersection of the files in the two sets divided by the union of the files in the two sets:

$$\frac{Breakages \bigcap Surprise}{Breakages \bigcup Surprise}.$$

Figure 3.1 shows the percentage of files that have both breakages and surprise defects. At most 6% of the breakage and surprise files overlap. This low percentage of overlap shows that breakages and surprises are two very different types of defects. However, further examination of their specific characteristics is needed to better understand the potential overlap.

Therefore, we investigate the characteristics of breakage and surprise defects along the different defect prediction dimensions. For example, previous work showed that the amount of activity (i.e., number of pre-release changes) is a good indicator of defect-proneness, therefore, we look at the activity of breakage and surprise defects to compare and contrast. We summarize our findings, which are statistically significant with a p-value < 0.05, as follows:

Activity. Comparing the activity, measured in number of MRs, of breakage and surprise files to non-breakage and non-surprise files shows that breakage and surprise files have less activity. This could indicate that perhaps breakage and surprise files were inspected less.

MR size. Breakage and surprise files are modified by larger MRs. On average, a breakage MR touches 47 files as compared to 8 files touched for a non-breakage MRs. Surprise MRs touch 71 files on average as compared to 13 files touched by non-surprise MRs.

Time. The start times of MRs that touch breakage and surprise files show that for a particular release, breakage files are modified earlier than average, whereas surprise defect files are on average, worked on closer to the release date. This suggests that considerably less time was available for the development and testing of surprise files.

Functionality. By definition, breakage files are involved with legacy features (e.g., core features of the communication software), whereas surprise defect files are involved with more recent features (e.g., the porting of the studied software system to work in virtual environments).

Maintenance efforts. Breakage and surprise defect files are involved in more code addition MRs than the average non-breakage or non-surprise files. However, breakage files were involved in more fixes than surprise defect files.

Do surprise defects have high impact?

The above comparisons show that breakage and surprise defects are different. To assess the impact of surprise defects we selected the five files with the highest surprise ratios in the last studied release of the software for an in-depth evaluation.

To evaluate the five files with the highest surprise score, we selected the latest changes prior to GA and the changes (defects) within one month after the GA for each of these files and investigated the nature and the origin of these defects by reading defect descriptions, resolution history and notes, inspection notes, as well as changes to the code. The five files were created eight, six, five, and, (for two files), one year(s) prior to GA. There were seven fixes within one month after the GA in these five files. Five defects had high severity and only two had medium severity, indicating that all of them were important defects. At least one fix in each of the five files was related to incomplete or inadequate requirements, i.e., introduced at the time of file creation or during the last enhancement of functionality.

For each post-GA defect in these five files, we traced back through changes to determine the first change that introduced the defect. Our intuition was that if the prediction can point to the change introducing the defect, it would limit the scope of the developers inspecting the flagged area to the kind of functionality changed and to the particular lines of code, thus simplifying the task of determining the nature of the potential defect or, perhaps, even highlighting the avenues for fixing it.

The results were surprising. In none of the cases the last pre-GA change could have been the cause of the defect. In the five defects related to inadequate requirements, we had to trace all the way to the initial implementation of the file. In the case of the two more recent files, at least we found a relationship between the pre-GA change (that was fixing inadequate requirements) and the post-GA fix that was fixing another shortcoming of the same requirements that was not addressed by the prior fix. For the two remaining defects introduced during coding or design phases, we had to trace back at least three changes to find the cause.

This qualitative investigation supports our intuition about the nature of surprise defects and our findings of the differences between breakages and surprise defects. The surprise defects appear to be an important kind of unexpected defect that appears to lurk in the code for long periods of time before surfacing. It also suggests a possible mechanism by which the prediction of surprise defects might work. As the usage profile or the intensity of usage changes over the years, early warnings are given by customers wanting to adjust a feature (one of the defects), or hard-to reproduce defects are starting to surface. Two of the defects appear to be caused by the increased use of multicore architecture that resulted in hard-to-reproduce timer errors and interactions with the system clock. At some point a major fault (surprise) is discovered (system restarting because of full message buffer, or because of data corruption) and the shortcomings of the requirements are finally addressed by the fix to the surprise defect.

Breakage and surprise defects are unique and different. Surprise defects also have high severity and appear to indicate problems in the requirements.

Table 3.2: List of factors used to predict breakage and surprise defects.

Category	Factor	Description	Rationale	Related Work
Traditional Factors	pre_defects	Number of pre-release defects	Traditionally performs well for post-release defect prediction.	Prior defects are a good indicator of future defects [295].
	pre_changes	Number of pre-release changes	Traditionally performs well for post-release defect prediction.	The number of prior modifications to a file is a good predictor of future defects [19, 105, 171].
	file_size	Total number of lines in the file	Traditionally performs well for post-release defect prediction.	The lines of code factor correlates well with most complexity factors (e.g., McCabe complexity) [105, 122, 171, 225].
Co-change Factors	(recent) num_co-changed_files	Number of files a file co-changed with	The higher the number of files a file co-changes with, the higher the chance of missing to propagate a change.	The number of files touched by a change is a good indicator of its risk to introduce a defect [200]. We apply this factor to the number of files co-changing with a file.
	(recent) size_co-changed_files	Cumulative size of the co-changed files	The larger the co-changed files are, the harder they are to maintain and understand.	The simple lines of code factor correlates well with most complexity factors (e.g., McCabe complexity) [105, 122, 171, 225]. We apply this factor to co-changing files.
	(recent) modification_size_co-changed_files	The number of lines changed in the co-changed files	The larger the changes are to the co-changed files, the larger the chance of introducing a defect.	Larger changes have a higher risk of introducing a defect [200]. We apply this factor to a file's co-changing files.
	(recent) num_changes_co-changed_files	Number of changes to the co-changed files	The higher the number of changes to the co-changed file, the higher the chance of introducing defects.	The number of prior modifications to a file is a good predictor of its future defects [19, 105, 171]. We apply this factor to a file's co-changing files.
	(recent) pre_defects_co-changed_files	Number of pre-release defects in co-changed files	The higher the number of pre-release defects in the co-changed files, the higher the chance of a defect.	Prior defects are a good indicator of future defects [295]. We apply this factor to a file's co-changing files.
Time Factors	latest_change_before_release	The time from the latest change to the release (in days)	Changes made close to the release date do not have time to get tested properly.	More recent changes contribute more defects than older changes [105].
	age	The age of the file (from first change until the release GA date)	The older the file, the harder it becomes to change and maintain.	Code becomes harder to change over time [70].

3.4 Case Study Setup

Now that we have analyzed the characteristics of breakage and surprise defects, we want to examine the effectiveness of predicting the locations of breakage and surprise defects. We address the following research questions:

RQ1. Can we effectively predict which files will have breakage/surprise defects?

RQ2. Which factors are important for the breakage/surprise defect prediction models?

RQ3. What effect does each factor have on the likelihood of finding a breakage/surprise defect in a file?

In this section, we outline and discuss the case study setup. First, we present the various factors used to predict breakage and surprise defects. Then, we outline the prediction modeling technique used. Finally, we outline how we evaluate the performance of our prediction models.

3.4.1 Factors Used to Predict Breakage and Surprise Defects

The factors that we use to predict files with breakage or surprise defects belong to three different categories: 1) traditional factors found in previous defect prediction work, 2) factors associated to co-changed files and 3) time-related factors (i.e., the age of a file and the time since the last change to the file).

These factors are based on previous post-release defect prediction work and on our findings in Section 3.3.4.

Traditional Factors: Previous work shows that the number of previous changes, the number of pre-release defects and the size of a file are good indicators of post-release defects [105, 310]. Since breakage and surprise defects are a special case of post-release defects, we believe that they can help us predict breakage and surprise defects as well.

Co-change Factors: By definition, breakage and surprise defects are not expected to happen. One possible reason for their occurrence could be hidden dependencies (e.g., logical coupling [96]). The intuition here is that a change to a co-changed file may cause a defect. Since more recent changes are more relevant, we also calculate for each factor its value in the three months prior to release, labeled as "recent".

Time Factors: To take into account the fact that breakage and surprise defects start earlier and later than average, respectively, we consider factors related to how close to a release a file is changed, as well as the file's age.

Each category comprises several factors, as listed in Table 3.2. For each factor, we provide a description, motivation and any related work.

3.4.2 Prediction Models

In this work, we are interested in predicting whether or not a file has a breakage or surprise defect. Similar to previous work on defect prediction [310], we use a logistic regression model. A logistic regression model correlates the independent variables in Table 3.2) with the dependent variable (probability of the file containing a breakage or surprise defect).

Initially, we built the logistic regression model using all of the factors. However, to avoid collinearity problems and to assure that all of the variables in the model are statistically significant, we removed highly correlated variables (i.e., any variables with correlation higher than 0.5). We performed this removal in an iterative manner, where we measured the correlation of a factor with all other factors and kept the factor that had the most factors correlated with it. We repeated this process until all the factors in the left in the model had a Variance Inflation Factor (VIF) below 2.5, as recommended by previous work [55]. This way, we are left with the least number of uncorrelated factors in the final model. To test for statistical significance, we measure the p-value of the independent variables in the model to make sure that this is less than 0.1.

Altogether, we extracted a total of 15 factors. After removing the highly correlated variables, nine factors were left that covered all three categories. It is important to note however that each release had a different set of factors.

3.4.3 Performance Evaluation of the Prediction Models

After building the logistic regression model, we verify its performance using two criteria: Explanatory Power and Predictive Power. These measures are widely used to measure the performance of logistic regression models in defect prediction [61, 310].

Explanatory Power. Ranges between 0-100%, and quantifies the variability in the data explained by the model. We also report and compare the variability explained by each independent variable in the model. Examining the explained variability of each independent variable allows us to quantify the relative importance of the independent variables in the model.

Predictive Power. Measures the accuracy of the model in predicting the files that have one or more breakage/surprise defects. The accuracy measures that we use (precision and recall) are based on the classification results in the confusion matrix (shown in Table 3.3).

1. **Precision:** the percentage of correctly classified breakage/surprise files over all of the files classified as having breakage/surprise defects: Precision $= \frac{TP}{TP+FP}$.
2. **Recall:** the percentage of correctly classified breakage/surprise files relative to all of the files that actually have breakage/surprise defects: Recall $= \frac{TP}{TP+FN}$.

Table 3.3: Confusion matrix

	True class	
Classified as	Breakage	No Breakage
Breakage	TP	FP
No Breakage	FN	TN

A precision value of 100% would indicate that every file we classify as having a breakage/surprise defect, actually has a breakage/surprise defect. A recall value of 100% would indicate that every file that actually has a breakage was classified as having a breakage/surprise.

We employ 10-fold cross-validation [69]. The data set is divided into two parts, a testing data set that contains 10% of the original data set and a training data set that contains 90% of the original data set. The model is trained using the training data and its accuracy is tested using the testing data. We repeat the 10-fold cross validation 10 times by randomly changing the fold. We report the average of the 10 runs.

Determining the threshold of the logistic regression model: The output of a logistic regression model is a probability (between 0 and 1) of the likelihood that a file belongs to the true class (e.g., a file is defective). Then, it is up to the user of the output of the logistic regression model to determine a threshold at which she/he will consider a file as belonging to the true class. Generally speaking, a threshold of 0.5 is used. For example, if a file has a likelihood of 0.5 or higher, then it is considered defective, otherwise it is not.

However, the threshold is different for different data sets and the value of the threshold affects the precision and recall values of the prediction models. In this chapter, we determine the threshold for each model using an approach that examines the tradeoff between type I and type II errors [200]. Type I errors are files that are identified as belonging to the true class, while they are not. Having a low logistic regression threshold (e.g., 0.01) increases type I errors: a higher fraction of identified files will not belong to the true class. A high type I error leads to a waste of resources since many non-faulty files may be wrongly classified. On the other hand, type II error is the fraction of files in the true class that are not identified as being true when they should be. Having a high threshold can lead to large type II errors, and thus missing many files that may be defective.

To determine the optimal threshold for our models, we perform a cost-benefit analysis between the type I and type II errors. Similar to previous work [200], we vary the threshold value between 0 to 1 and use the threshold where the type I and type II errors are equal.

3.4.4 Measuring the Effect of Factors on the Predicted Probability

In addition to evaluating the accuracy and explanatory power of our prediction models, we need to understand the effect of a factor on the likelihood of finding a breakage or surprise defect. Quantifying this effect helps practitioners gain an in-depth understanding of how the various factors relate to breakage and surprise defects.

To quantify this effect, we set all of the factors to their median value and record the predicted probabilities, which we call the Standard Median Model (SMM). Then, to measure the individual effect of each factor, we set all of the factors to their median value, except for the factor whose effect we want to measure. We double the median value of that factor and re-calculate the predicted values, which we call the Doubled Median Model (DMM).

We then subtract the predicted probability of the SMM from the predicted output of the DMM and divide by the predicted probability of the SMM. Doing so provides us with a way to quantify the effect a factor has on the likelihood of a file containing a breakage or surprise defect.

The effect of a factor can be positive or negative. A positive effect means that a higher value of the factor increases the likelihood, whereas a negative effect value means that a higher value of the factor decreases the likelihood of a file containing a breakage/surprise defect.

3.5 Case Study Results

In this section, we address the research questions posted earlier. First, we examine the accuracy (in terms of predictive and explanatory power) of our prediction models. Then, we examine the contribution of each factor on the models in terms of explanatory power. Lastly, we examine the effect of the factors on the breakage- and surprise-proneness.

RQ1. Can we effectively predict which files will have breakage/surprise defects?

Using the extracted factors, we build logistic regression models that aim to predict whether or not a file will have a breakage or surprise defect. The prediction was performed for five different releases of the large commercial software system. To measure predictive power, we present the precision, recall and the threshold (Th.) value used in the logistic regression model, for each release. The last row in the tables presents the average across all releases.

Predictive power: Tables 3.4 and 3.5 show the results of the breakage and surprise defect prediction models, respectively. On average, the precision for breakage and surprise defects is low, i.e., 4.4% for breakages and 5.9% for surprise defects.

It is important to note that the low precision value is due to the low percentage of breakage and surprise defects in the data set (as shown earlier in Table 3.1). As noted by Menzies *et al.* [190], in cases where the number of instances of an occurrence is so low (i.e., 2%), achieving a high precision is extremely difficult, yet not that important. In fact, a random prediction would be correct 2.0% of the time, on average, whereas our prediction model more than doubles that precision for breakages and approximately triples that for surprise defects. The more important measure of the prediction model's performance is recall [190], which, on average is 69.0% for breakage defects and 74.0% for surprise defects.

Explanatory power: The explanatory power of the models ranges between 6.85 - 16.96% (average of 12.35%) for breakage defects and between 4.1 - 30.64% (average of 17.26%) for surprise defects. The values that we achieve here are comparable to those achieved in previous work predicting post-release defects [55, 254].

The explanatory power may be improved if more (or better) factors are used in the prediction model. We view the factors used in our study as a starting point for breakage and surprise defect prediction and plan to (and encourage others to) further investigate in order to improve the explanatory power of these models.

Other considerations: For each prediction model, we explicitly report the threshold for the logistic regression models as Th. (pred) in Tables 3.4 and 3.5. In addition, the surprise threshold, which we use to identify files that had surprise defects, is given under the Th. (sup) column in Table 3.5.

Since the number of pre-release defects is used in the dependent variable of the surprise model ($Surprise(file) = \frac{post\ defects}{pre\ defects}$), we did not use it as part of the independent variables in the prediction model. This makes our results even more significant, since previous work [204, 310] showed that pre-release defects traditionally are the largest contributor to post-release defect prediction models.

Table 3.4: Performance of breakage prediction models

	Predictive Power			**Explanatory Power**
Release	Precision	Recall	Th. (pred)	Deviance Explained
R1.1	3.6	69.0	0.46	16.96%
R2.1	4.3	64.3	0.47	6.85%
R3	4.8	69.6	0.49	14.49%
R4	4.1	73.8	0.49	12.09%
R4.1	5.4	68.5	0.46	11.38%
Average	4.4	69.0	0.47	12.35%

Table 3.5: Performance of surprise prediction models

	Predictive Power				**Explanatory Power**
Release	Precision	Recall	Th. (pred)	Th. (sup)	Deviance Explained
R1.1	-	-	-	-	-
R2.1	1.4	75.0	0.56	2.4	4.10 %
R3	7.3	69.0	0.44	1.7	10.79 %
R4	9.8	75.8	0.38	1.6	30.64 %
R4.1	4.9	75.4	0.34	1.5	23.52 %
Average	5.9	74.0	0.43	1.8	17.26 %

Our prediction models predict breakage and surprise defects with a precision that is at least double that of a random prediction and a recall above 69%.

RQ2. Which factors are important for the breakage/surprise defect prediction models?

In addition to knowing the predictive and explanatory power of our models, we would like to know which factors contribute the most to these predictions. To answer this question, we perform an ANOVA analysis to determine the contribution of the three factor categories. We summarize the findings in Tables 3.6 and 3.7 as follows:

Traditional Defect Prediction Factors: are major contributors for the breakage defect prediction models, however, they only have a small contribution in predicting surprise defects.

Co-Change Factors provide a small contribution to breakage defect prediction models, however, they make a major contribution to predicting surprise defects.

Time Factors provide a minor contribution to breakage defect prediction models, however, they make a major contribution in predicting surprise defects.

Although there are exceptions to the above mentioned observations (e.g., traditional defect prediction factors make a major contribution to the surprise defect model in R2.1), our observations are based on the trends observed in the majority of the releases.

As mentioned earlier in Section 3.3.4, breakage files were mainly involved with defect fixing efforts and are, by definition, defects that are found in the field. Perhaps these characteristics of breakage defects help explain why the traditional post-release defect prediction factors perform well in predicting breakages.

Furthermore, earlier observations in Section 3.3.4 showed that surprise defect files were worked on later than other files (i.e., MRs that touched surprise defect files were started later than other MRs). Our findings show that being changed close to a release is one of the best indicators of whether or not a file will have a surprise defect.

Table 3.6: Contribution of factor categories to the explanatory power of breakage prediction models

	R1.1	**R2.1**	**R3**	**R4**	**R4.1**
Traditional Defect Prediction Factors	15.60%	6.80%	11.16%	8.92%	10.75%
Co-Change Factors	0.59%	0.03%	2.93%	1.07%	0.30%
Time Factors	0.77%	0.01%	0.39%	2.09%	0.32%
Overall Deviance Explained	**16.96%**	**6.85%**	**14.48%**	**12.09%**	**11.38%**

Table 3.7: Contribution of factor categories to the explanatory power of surprise prediction models

	R1.1	**R2.1**	**R3**	**R4**	**R4.1**
Traditional Defect Prediction Factors	-	2.94%	2.76%	0.68%	0.69%
Co-Change Factors	-	0.29%	4.56%	9.55%	1.52%
Time Factors	-	0.86%	3.46%	20.39%	21.30%
Overall Deviance Explained	-	**4.10%**	**10.79%**	**30.64%**	**23.52%**

Traditional defect prediction factors are good indicators of breakage defects. The factors related to co-changed files and time-related factors are good indicators of surprise defects.

RQ3. What effect does each factor have on the likelihood of finding a breakage/surprise defect in a file?

Thus far we examined the prediction accuracy and the importance of the factors to these prediction models. Now, we study the effect of each factor on the likelihood of finding a breakage or surprise defect in a file. In addition to measuring the effect, we also consider stability and explanatory impact. If an effect has the same sign/direction for all releases (i.e., positive or negative effect in all releases), then we label it as highly stable. If the effect of a factor has the same sign in all releases except for one, then we label it as being mainly stable. A factor having a different sign in more than two of the five releases is labeled as being unstable. The explanatory impact column is derived from the values in Tables 3.6 and 3.7. If a factor belongs to a category that had a strong explanatory power, then we label it as having high impact, otherwise we consider it as having low impact.

We use the stability and impact measure to help explain the strength of our findings. For example, if we find that a factor has a positive effect and has high impact to the explanatory power of the model, then we believe this effect to be strong.

Breakage Defects: Table 3.8 shows the effect of the different factors on the likelihood of predicting a breakage defect in a file. We observe that in all releases, pre-release defects have a positive effect on the likelihood of a breakage defect existing in a file (i.e., highly stable). In addition, Table 3.6 showed that the traditional defect prediction factors contributed the most to the explanatory power of the model (i.e., high impact). File size generally has a positive effect. The latest change before release factor had low impact that is non-stable.

As stated earlier, our manual examination of the breakage files showed that breakage files were especially involved with fixing efforts. Therefore, the fact that pre-release defects and size have a strong positive effect was expected (since these factors are positively correlated with post-release defects). In fact, we found that the average file size of breakage files is 50% larger than non-breakage files. The rest of the factors had low impact on the explanatory power of the prediction model, therefore we cannot conclude any meaningful results from their effect.

Release 4 (R4) in our results seems to be an outlier. For example, contrary to the other releases, file size shows a negative effect in R4. After closer examination, we found that R4 was a large major release that added a large amount of new functionality. This could be the reason why the effect values for R4 are so different from the effect results of the remaining releases.

Table 3.8: Effect of factors on the likelihood of predicting a file with a breakage defect. Effect is measured by setting a factor to double its median value (1 if the median is 0), while the rest of the factors are set to their median value.

	R1.1	**R2.1**	**R3**	**R4**	**R4.1**	**Stability**	**Explanatory Impact**
pre_defects	124%	105%	85%	68%	121%	Highly Stable	High Impact
file_size	263%	120%	575%	-51%	18%	Mainly Stable	High Impact
modification_size_co_changed_files	744%	-	-	-	-	-	Low Impact
num_co_changed_files	-71%	-	-	887%	-	-	Low Impact
num_co_changed_files_per_mr	-	-	-91%	-	-	-	Low Impact
recent_num_co_changed_files	-	-	-	-6%	-	-	Low Impact
recent_modification_size_co_changed_files	2%		13%	-	2%	-	Low Impact
recent_modification_size_co_changed_files_per_mr	-	1%	-	-	-	-	Low Impact
latest_change_before_release	-58%	-16%	85%	-88%	-55%	Not Stable	Low Impact

Surprise Defects: Table 3.9 shows the effect values for the surprise defect prediction model. In this model, file size has a large stable positive effect, however as shown earlier in Table 3.7 this factor category has very little contribution to the explanatory power of the model (i.e., low impact).

We find that making a change last minute increases the likelihood of a surprise defect (i.e., the negative effect of latest change before release). As shown in Section 3.3.4, in contrast to breakages, surprise defect files were worked on later than usual. We conjecture that starting late means less time for testing, hence the much higher effect of these late changes on surprise defect files compared to the breakage files.

Table 3.9: Effect of factors on the likelihood of predicting a file with a surprise defect. Effect is measured by setting a factor to double its median value (1 if the median is 0), while the rest of the factors are set to their median value.

	R1.1	**R2.1**	**R3**	**R4**	**R4.1**	**Stability**	**Explanatory Impact**
file_size	-	1417%	260%	184%	128%	Highly Stable	Low Impact
num_co_changed_files	-	-	-	-67%	-	-	High Impact
num_co_changed_files_per_mr	-	-	-75%	-	-	-	High Impact
recent_num_co_changed_files	-	-	-	-44%	-	-	High Impact
recent_modification_size_co_changed_files	-	-	-3%	-	-20%	-	High Impact
recent_modification_size_co_changed_files_per_mr	-	5%	-	-	-	-	High Impact
latest_change_before_release	-	-63%	-65%	-97%	-96%	Highly Stable	High Impact

Pre-release defects and file size have a positive effect on the likelihood of a file containing a breakage defect. The time since last change before release has a negative effect on the likelihood of a file having a surprise defect.

3.6 Effort Savings by Focusing on Surprise and Breakage Files

Thus far, we have shown that we are able to build prediction models to predict files that contain breakage and surprise defects. However, one question still lingers: what if we used the traditional post-release defect prediction model to predict breakage and surprise defects? Is it really worth the effort to build these specialized models?

To investigate whether building specialized prediction models for breakage and surprise defects is beneficial, we use defect prediction models that are trained to predict post-release defects, to predict files with breakages and surprise defects. Due to the fact that post-release defects are much more common than breakages or surprise defects, post-release defect models are more likely to say that most files have breakages or surprise defects. That will lead to a large amount of unnecessary work. Therefore, we use the number of false negatives to compare the performance of the post-release models and the specialized models. To make a meaningful comparison of effort (which is related to Type I error), we fix Type II error (i.e., the fraction of defective files that are not identified as being defective) to be the same in both models.

Breakage Defects. Table 3.10 shows the results of the specialized prediction model (Breakage -> Breakage) and the post-release prediction model (Post -> Breakage) for release 4.1. Both of these models predict files that have breakage defects. The false positives are highlighted (in grey) in Table 3.10. We observe that the specialized prediction model has approximately 3.3% (i.e., $\frac{688-665}{688}$) less false positives than the post-release model. This means that using the specialized model would reduce the inspection effort of files by 3.3%. We also convert this effort saving into LOC, which is approximately 24.3% of the total LOC.

Table 3.10: Breakage defect in release 4.1

	Breakage -> Breakage Predicted			Post -> Breakage Predicted	
Actual	0	1	Actual	0	1
0	748	665	0	725	688
1	7	26	1	7	26

Surprise Defects. Table 3.11 shows the results of the specialized model (Surprise -¿ Surprise) and the post-release prediction model (Post -¿ Surprise) for files that have surprise defects. In this case, the specialized models lead to approximately 30% (i.e., $\frac{673-471}{673}$) effort savings (i.e., less false positives). Comparing the savings in terms of LOC, we find that using the specialized prediction model leads to approximately 21.2% effort savings compared to using a traditional post-release defect prediction model. This is a considerable amount of effort savings and shows the benefits of building a specialized prediction model of files with surprise defects.

Table 3.11: Surprise defects in release 4.1

	Surprise -> Surprise Predicted			Post -> Surprise Predicted	
Actual	0	1	Actual	0	1
0	957	471	0	755	673
1	4	14	1	4	14

Using our custom prediction models reduces the amount of files inspected by practitioners by 3.3% for breakages and 30% for surprise defects.

3.7 Threats to Validity

Threats to Construct Validity consider the relationship between theory and observation, in case the measured variables do not measure the actual factors.

Breakage MRs were manually identified in our data by project experts. Although this manual linking was done by these project experts, some MRs may have been missed or incorrectly linked.

We used files in defects reported within one month after release to determine the surprise defect threshold. The assumption here is that defects reported within one month involve important functionality that is widely used after release. Defects that affect important functionality may be reported later than one month, however.

Threats to External Validity consider the generalization of our findings. The studied project was a commercial project written mainly in C/C++, therefore, our results may not generalize to other commercial or open source projects.

3.8 Lessons Learnt

After performing our study, we asked the opinions of the highly experienced quality manager in the project about the prediction results. The manager has a theory about the reported effect of our $last\_change\_before\_release$ factor. The theory is that the so called "late fix frenzies" that go on in organizations to bring down the number of open defects in a software system before the release, might have compromised the quality of inspections and other quality assurance activities. This suggests that prediction may help to quantify and confirm intuition about the relationships between the aspects of the development process and the high-impact defects.

However, when the quality manager considered the merits of our prediction by their utility for system verification she argued that identifying locations of defects is of limited use to system testers because they test system features or behaviors, not individual files. In addition, she doubted that the predicted location could be helpful even to developers doing inspections or unit tests without the additional information about the nature of the problem or how it should be fixed.

Despite the positive findings related to prediction quality, narrowing the scope, and effort savings, we still appear to be far from the state where the prediction results could be used in practice. Based on our quantitative and qualitative findings and experience we hypothesize that for defect prediction to become a practical tool, each predicted location has to also contain a clear suggestion on why the defect might be there and how it may be fixed.

To achieve this, we propose a procedure similar to the one we conducted to identify surprise defects, to classify defects into a variety of classes according to their nature, the ways they may have been introduced, and to the ways they may need to be fixed. We expect that each type of high-impact defect would have a different prediction signature, which, in turn, can be used to provide developers with a recommendation on where the defect may be, what nature it may have, and how it may be fixed. We can see an example of such a classification in the static analysis tools that not only provide a warning, but also give a reason why a particular pattern might be a defect and a clear suggestion on how the potential defect can be fixed. We are not aware of any similar patterns for defect prediction, however, enhancing SDP to provide such reasoning would be beneficial.

For example, our investigation of surprise defects could be used to provide a warning of the kind: "This file might contain a defect that has been introduced a while ago, perhaps because of incorrect requirements. Change patterns to this file suggest that the usage profile might have changed recently and the requirements may need to be reviewed to make sure they are accurate and complete." Obviously, a more extensive investigation may be needed to provide more specific recommendations and we believe that the defect prediction methods should be tailored not simply to predict defect locations, but, like basic static analysis tools such as `lint`, should also detect patterns of changes that are suggestive of a particular type of defect, and recommend appropriate remedies.

3.9 Conclusion

The majority of defect prediction work focuses on predicting post-release defects, yet the adoption of this work in practice remains relatively low [102, 115]. One of the main reasons for this is that defect prediction techniques generally identify too many files as having post-release defects, requiring significant inspection effort from developers.

Instead of considering all defects as equal, this Chapter focuses on predicting a small subset of defects that are highly impacting. Since there are many different interpretations of "high impact", we focus on one interpretation from the perspective of customers (breakage defects) and one from the perspective of practitioners (surprise defects). We find that:

- Both kinds of defects are different and that surprise defects, similar to the more established concept of breakage defects, have a high impact.
- Specialized defect prediction models can predict breakage and surprise defects effectively, yielding sizeable effort savings over using simple post-release defect prediction models.
- Traditional defect prediction factors (i.e., the number of pre-release defects and file size) are good predictors of breakage defects, whereas the number of co-changed files, the size of recently co-changed files and the time since the last change are good predictors of surprise defects.

Our findings suggest that building specialized prediction models is valuable to bring defect prediction techniques closer to adoption in practice.

However, our qualitative analysis of surprise defects and the feedback from the quality assurance manager clearly indicate that further work is needed to develop defect prediction into a practical tool. In particular, we found support for the idea of building specialized models that identify not only a defect's location, but also its nature, thus greatly simplifying the process of determining what the defect is and how it needs to be fixed.

In the following chapter, we focus on another type of high-impact defects, re-opened defects, i.e., defects that were closed by developers, but re-opened at a later time. Re-opened defects have a high impact since they take considerably longer to resolve. We extract a number of factors from the project's development history and build prediction models to predict re-opened defects. Then, we analyze these prediction models to determine factors that best indicate these re-opened defects.

Chapter 4

Studying Re-opened Defects

Fixing software defects accounts for a large amount of the software maintenance resources. Generally, defects are reported, fixed, verified and closed. However, in some cases defects have to be re-opened. Re-opened defects increase maintenance costs, degrade the overall user-perceived quality of the software and lead to unnecessary rework by busy practitioners. In this chapter, we study and predict re-opened defects through a case study on three large open source projects – namely Eclipse, Apache and OpenOffice. Re-opened defects are considered to have high-impact since they take considerably longer to resolve, increasing the software maintenance costs. We structure our study along 4 dimensions: 1) the work habits dimension (e.g., the weekday on which the defect was initially closed), 2) the bug report dimension (e.g., the component in which the defect was found) 3) the defect fix dimension (e.g., the amount of time it took to perform the initial fix) and 4) the people dimension (e.g., the experience of the defect fixer). We build decision trees using the aforementioned factors that aim to predict re-opened defects. We perform top node analysis to determine which factors are the most important indicators of whether or not a defect will be re-opened. Our study shows that the comment text and last status of the defect when it is initially closed are the most important factors related to whether or not a defect will be re-opened. Based on these dimensions we create decision trees that predict whether a defect will be re-opened after its closure. Using a combination of our dimensions, we can build explainable prediction models that can achieve a precision between 49.9-78.3% and a recall in the range of 72.6-93.5% when predicting whether a defect will be re-opened. We find that the factors that best indicate which defects might be re-opened vary based on the project. The comment text is the most important factor for the Eclipse and OpenOffice projects, while the last status is the most important one for Apache. These factors should be closely examined in order to reduce maintenance cost due to re-opened defects.

4.1 Introduction

Large software systems are becoming increasingly important in the daily lives of many people. A large portion of the cost of these software systems is attributed to their maintenance. In fact, previous studies show that more than 90% of the software development cost is spent on maintenance and evolution activities [82].

A plethora of previous research addresses issues related to software defects. For example, software defect prediction work uses various code, process, social structure, geographic distribution and organizational structure factors to predict defect-prone software locations (e.g., files or directories) [42, 55, 62, 105, 204, 214, 310]. Other work focuses on predicting the time it takes to fix a defect [152, 228, 282].

This existing work typically treats all defects equally, meaning, the existing work did not differentiate between re-opened and new defects. Re-opened defects are defects that were closed by developers, but re-opened at a later time. Defects can be re-opened for a variety of reasons. For example, a previous fix may not have been able to fully fix the reported defect. Or the developer responsible for fixing the defect was not able to reproduce the defect and might close the defect, which is later re-opened after further clarification.

Re-opened defects are highly impacting since they take considerably longer to resolve. For example, in the Eclipse Platform 3.0 project, the average time it takes to resolve (i.e., from the time the defect is initially opened till it is fully closed) a re-opened defect is more than twice as much as a non-reopened defect (371.4 days for re-opened defects vs. 149.3 days for non-reopened defects). An increased defect resolution time consumes valuable time from the already-busy developers. For example, developers need to re-analyze the context of the defect and read previous discussions when a defect is re-opened. In addition, such re-opened negatively impact the overall end user's experience.

This Chapter presents a study to determine factors that indicate whether a defect will be re-opened. Knowing which factors are attributed to re-opened defects prepares practitioners to think twice before closing a defect. For example, if it is determined that defects logged with high severity are often re-opened, then practitioners can pay special attention (e.g., by performing more thorough reviews) to such defects and their fixes.

We combine data extracted from the defect and source control repositories of the Eclipse, Apache and OpenOffice open source projects to extract 24 factors that are grouped into four different dimensions:

1. **Work habits dimension**: is used to gauge whether the work habits of the software practitioners initially closing the defect affect its likelihood of being re-opened. The main rationale for studying this dimension is to possibly provide some insights about process changes (e.g., avoiding closing defects during specific times) that might lead to a reduction in defect re-openings

2. **Bug report dimension**: is used to examine whether information in the bug report can be used to determine the likelihood of defect re-opening. The main rationale for examining this dimension is to see whether information contained in the bug report can be used to hint a higher risk of a defect being re-opened in the future. Practitioners can then be warned about such data in order to reduce defect re-openings.

3. **Defect fix dimension**: is used to examine whether the fix made to address a defect can be used to determine the likelihood of a defect being re-opened. The rationale for studying this dimension is to examine whether certain factors related to the defect fix increase the likelihood of it being re-opened later. Insights about issues that might increase the likelihood of a defect being re-opened are helpful to practitioners so they can know what to avoid when addressing defects.

4. **People dimension**: is used to determine whether the personnel involved with a defect can be used to determine the likelihood of a defect being re-opened. The rationale for using this dimension is to examine whether certain personnel (e.g., more experienced

personnel) should avoid or be encouraged to address defects, in order to reduce defect re-openings.

To perform our analysis, we build decision trees and perform a Top Node analysis [118, 246] to identify the most important factors in building these decision trees. Furthermore, we use the extracted factors to predict whether or not a closed defect will be re-opened in the future. In particular, we aim to answer the following research questions:

Q1. Which of the extracted factors indicate, with high probability, that a defect will be re-opened?

The factors that best indicate re-opened defects vary based on the project. The comment text is the most important factor for the Eclipse and OpenOffice projects, while the last status is the most important one for Apache.

Q2. Can we accurately predict whether a defect will be re-opened using the extracted factors?

We use 24 different factors to build accurate prediction models that predict whether or not a defect will be re-opened. Our models can correctly predict whether a defect will be re-opened with precision between 52.1-78.6% and recall between 70.5-94.1%.

4.1.1 Organization of Chapter

Section 4.2 describes the life cycle of a defect. Section 4.3 presents the methodology of our study. We detail our data processing steps in Section 4.4. The case study results are presented in Section 4.5. We compare the prediction results using different algorithms and examine the work habits dimensions in more detail in Section 4.6. The threats to validity and related work are presented in Sections 4.7 and 4.8, respectively. Section 4.9 concludes the chapter.

4.2 The Bug Life Cycle

Defect tracking systems, such as Bugzilla [1], are commonly used to manage and facilitate the defect resolution process. These bug tracking systems record various characteristics about reported defects, such as the time the defect was reported, the component the defect was found in and any discussions related to the defect. The information stored in bug tracking systems is leveraged by many researchers to investigate different phenomena (e.g., to study the time it takes to resolve defects [121, 198]).

The life cycle of a defect can be extracted from the information stored in the bug tracking systems. We can track the different states that defects have gone through and reconstruct their life cycles based on these states. For example, when defects are initially logged, they are confirmed and labeled as new defects. Then, they are triaged and assigned to developers to be fixed. After a developer fixes the defect, the fix is verified and the defect closed.

A diagram representing the majority of the states bugs/defects go through is shown in Figure 4.1. When developers, testers or users experience a defect, they log/submit a bug report in the bug tracking system. The defect is then set to the *Opened* state. Next, the defect is triaged to determine whether it is a real defect and whether it is worth fixing. After the triage process, the defect is accepted and its state is updated to *New*. It then gets assigned to a developer who will be responsible to fix it (i.e., its state is *Assigned*). If a defect is known to a developer beforehand[1], it is assigned to that developer who implements the fix and the defect goes directly from the *New* state to the *Resolved_FIXED* state. More typically, defects are assigned to a developer (i.e., go to the *Assigned* state), who implements a fix for the defect and its state is transitioned into *Resolved_FIXED*. In certain cases, a defect is not fixed by the developers because it is identified as being invalid (i.e., state *Resolved_INVALID*), a decision was made to not fix the defect (i.e., state *Resolved_WONTFIX*), it is identified as a duplicate

[1]For example, in some cases developers discover a defect and know how to fix it, however they create a bug report and assign it to themselves for book-keeping purposes.

of another defect (i.e., state *Resolved_DUPLICATE*) or the defect is not reproducible by the developer (i.e., state *Resolved_WORKSFORME*). Once the defect is resolved, it is verified by another developer or tester (state *Verified_FIXED*) and finally closed (state *Closed*).

However, in certain cases, defects are re-opened after their closure. This can be due to many reasons. For example, a defect might have been incorrectly fixed and resurfaces. Another reason might be that the defect was closed as being a duplicate and later re-opened because it was not actually a duplicate.

In general, re-opened defects are not desired by software practitioners because they degrade the overall user-perceived quality of the software and often lead to additional and unnecessary rework by the already-busy practitioners. Therefore, in this Chapter we set out to investigate which factors best predict re-opened defects. Then, we use these factors to build accurate prediction models to predict re-opened defects.

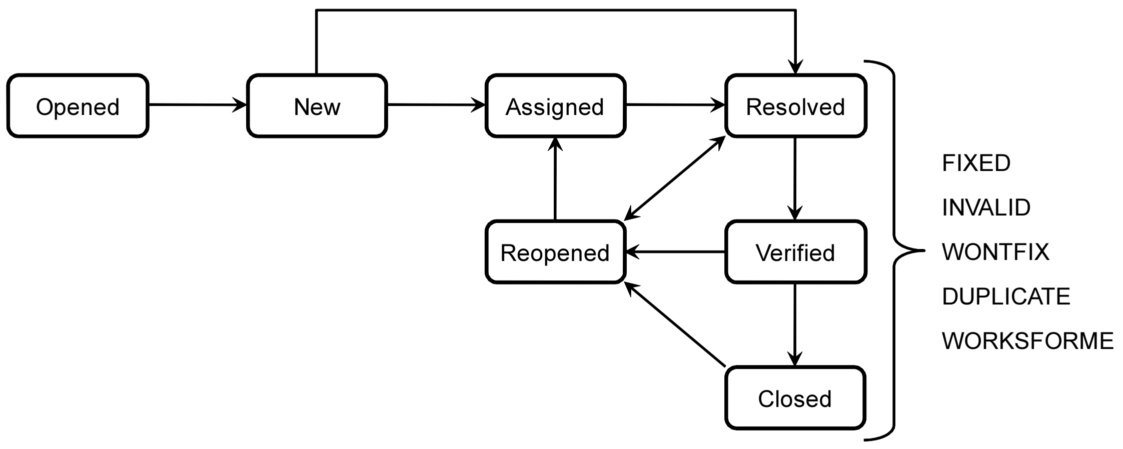


Figure 4.1: Bug resolution proces

Table 4.1: Factors considered in our study

Dim	Factor	Type	Explanation	Rationale
Work habits	Time	Nominal	Time in hours (Morning, Afternoon, Evening, Night) when the defect was first closed.	Defects closed at certain times in the day (e.g., late afternoons) are more/less likely to be re-opened.
	Weekday	Nominal	Day of the week (e.g., Mon or Tue) when the defect was first closed.	Defects closed on specific days of the week (e.g., Fridays) are more/less likely to be re-opened.
	Month day	Numeric	Calendar day of the month (0-30) when the defect was first closed.	Defects closed at specific periods like the beginning, mid or end of the month are more/less likely to be re-opened.
	Month	Numeric	Month of the year (0-11) when the defect was first closed.	Defects closed in specific months (e.g., during holiday months like December) are more/less likely to be re-opened.
	Day of year	Numeric	Day of the year (1-365) when the defect was first closed.	Defects closed in specific times of the year(e.g., later on in the year) are more/less likely to be re-opened.
Bug report	Component	Nominal	Component the defect was found in (e.g., UI, Debug or Search).	Certain components might be harder to fix; defects found in these components are more/less likely to be re-opened.
	Platform	Nominal	Platform (e.g., Windows, MAC, UNIX) the defect was found in.	Certain platforms are harder to fix defects for, and therefore, their defects are more likely to be re-opened.
	Severity	Numeric	Severity of the reported defect. A high severity (i.e., 7) indicates a blocker defect and a low severity (i.e., 1) indicates an enhancement.	Defects with high severity values are harder to fix and are more likely to be re-opened.
	Priority	Numeric	Priority of the reported defect. A low priority value (i.e., 1) indicates an important defect and a high priority value (i.e., 5) indicates a defect of low importance.	Defects with low priority value (i.e., high importance) are likely to get more careful attention and have a smaller chance of being re-opened.
	Number in CC list	Numeric	Number of persons in the cc list of the logged defects before the first re-open.	Defects that have persons in the cc list are followed more closely, and hence, are more/less likely to be re-opened.
	Description size	Numeric	The number of words in the description of the defect.	Defects that are not described well (i.e., have a short description) are more likely to be re-opened.
	Description text	Bayesian score	The text content of the defect description.	Words included in the defect description can indicate whether the defect is more likely to be re-opened.
	Number of comments	Numeric	The number of comments attached to a bug report before the first re-open.	The higher the number of comments, the more likely the defect is controversial. This might lead to a higher chance of it being re-opened.
	Comment size	Numeric	The number of words in all comments attached to the bug report before the first re-open.	The longer the comments are, the more the discussion about the defect and the more/less likely it will be re-opened.
	Comment text	Bayesian score	The text content of all the comments attached to the bug report before the first re-open.	The comment text attached to a bug report may indicate whether a defect will be re-opened.
	Priority changed	Boolean	States whether the priority of the defect was changed after the initial report before the first re-open.	Defects that have their priorities increased are generally followed more closely and are less likely to be re-opened.
	Severity changed	Boolean	States whether the severity of the defect was changed after the initial report before the first re-open.	Defects that have their severities increased are generally followed more closely and are less likely to be re-opened.
Defect fix	Time days	Numeric	The time it took to resolve a defect, measured in days. For re-opened defects, we measure the time to perform the initial fix.	The time it takes to fix a defect is indicative of its complexity and hence the chance of finding a good fix.
	Last status	Nominal	The last status of the defect when it is closed for the first time.	When defects are closed using certain statuses (e.g., Worksforme or duplicate), they are more/less likely to be re-opened.
	No. of files in fix	Numeric	The number of files edited to fix the defect for the first time.	Defects that require larger fixes, indicated by an increase in the number of files that need to be edited, are more likely to be re-opened.
People	Reporter Name	String	Name of the defect reporter.	Defects reported by specific individuals are more/less likely to be re-opened.
	Fixer Name	String	Name of the initial defect fixer.	Defects fixed by specific individuals are more/less likely to be re-opened.
	Reporter Experience	Numeric	The number of defects reported by the defect reporter before reporting this defect.	More experienced reporters are less likely to have their defects re-opened.
	Fixer Experience	Numeric	The number of defect fixes the initial fixer performed before fixing this defect.	Defects fixed by experienced fixers are less likely to be re-opened.

4.3 Approach to Predict Re-opened Defects

In this section, we describe the factors used to predict whether or not a defect will be re-opened. Then, we present decision trees and motivate their use in our study. Finally, we present the factors used to evaluate the performance of the prediction models.

4.3.1 Dimensions Used to Predict if a Defect will be Re-opened

We use information stored in the bug tracking system, in combination with information from the source control repository of a project to derive various factors that we use to predict whether a defect will be re-opened.

Table 4.1 shows the extracted factors, the type of the factor (e.g., numeric or nominal), provides an explanation of the factor and the rationale for using each factor. We have a total of 24 factors that cover four different dimensions. We describe each dimension and its factors in more detail next.

Work habits dimension. Software developers are often overloaded with work. This increased workload affects the way these developers perform. For example, Sliwerski *et al.* [259] showed that code changes are more likely to introduce defects if they were done on Fridays. Anbalagan *et al.* [13] showed that the time it takes to fix a defect is related to the day of the week when the defect was reported. Hassan and Zhang [118] used various work habit factors to predict the likelihood of a software build failure.

These prior findings motivate us to include the work habit dimension in our study on re-opened defects. For example, developers might be inclined to close defects quickly on a specific day of the week to reduce their work queue and focus on other tasks. These quick decisions may cause the defects to be re-opened at a later date.

The work habit dimension consists of four different factors. The factors of the work habit dimension are listed in Table 4.1. The time factor was defined as a nominal variable that can be `morning` (7 AM to 12 Noon), `afternoon` (Noon to 5 PM), `evening` (5 PM to 12 midnight) or `night` (midnight to 7 AM), indicating the hours of the day that they defect was initially closed on.

Bug report dimension. When a defect is reported, the reporter of the defect is required to include information that describes the defect. This information is then used by the developers to understand and locate the defect. Several studies use that information to study the amount of time required to fix a defect [198]. For example, Panjar [228] showed that the severity of a defect has an effect on its lifetime. In addition, a study by Hooimeijer and Weimer [125] showed that the number of comments attached to a bug report affects the time it takes to fix it.

We believe that attributes included in a bug report can be leveraged to determine the likelihood of a defect being re-opened. For example, defects with short or brief descriptions may need to be re-opened later because a developer may not be able to understand or reproduce them the first time around.

A total of 11 different factors make up the bug report dimension. They are listed in Table 4.1.

Defect fix dimension. Some defects are harder to fix than others. In some cases, the initial fix to the defect may be insufficient (i.e., it did not fully fix the defect) and, therefore, the defect needs to be re-opened. We conjecture that more complicated defects are more likely to be re-opened. There are several ways to measure the complexity of a defect fix. For example, if the defect fix requires many files to be changed, this might be an indicator of a rather complex defect [116].

The defect fix dimension uses factors to capture the complexity of the initial fix of a defect. Table 4.1 lists the three factors that measure the time it took to fix the defect, the status before the defect was re-opened and the number of files changed to fix the defect.

People dimension. In many cases, the people involved with the bug report or the defect fix are the reason that it is re-opened. Reporters may not include important information when reporting a defect, or they lack the experience (i.e., they have never reported a defect before). On the other hand, developers (or fixers) may lack the experience and/or technical expertise to fix or verify a defect, leading to the re-opening of the defect.

The people dimension, listed in Table 4.1, is made up of four factors that cover defect reporters, defect fixers and their experience.

The four dimensions and their factors listed in Table 4.1 are a sample of the factors that can be used to study why defects are reopened. We plan (and encourage other researchers) to build on this set of dimensions to gain more insights into why defects are re-opened.

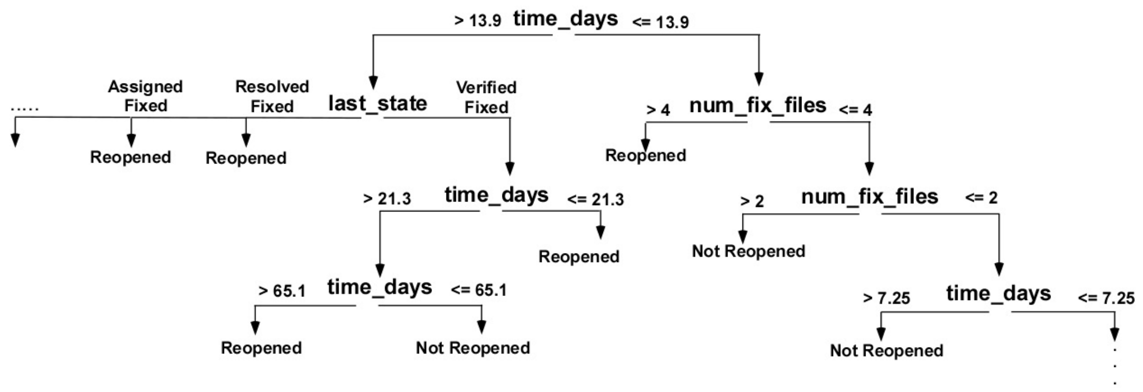


Figure 4.2: Sample decision tree.

4.3.2 Building Tree-Based Predictive Models

To determine if a defect will be re-opened, we use the factors from the four aforementioned dimensions as input to a decision tree classifier. Then, the decision tree classifier predicts whether or not the defect will be re-opened.

We chose to use a decision tree classifier for this study since it offers an explainable model.

This is very advantageous because we can use these models to understand what attributes affect whether or not a defect will be re-opened. In contrast, most other classifiers produce "black box" models, where it is hard explain which attributes affect the predicted outcome.

To perform our analysis, we divide our data set into two sets: a training set and a test set. The training set is used to train the decision tree classifier. Then, we test the accuracy of the decision tree classifier using our test set.

The C4.5 algorithm [233] was used to build the decision tree. Using the training data, the algorithm starts with an empty tree and adds decision nodes or leafs at each level. The information gain using a particular attribute is calculated and the attribute with the highest information gain is chosen. Further analysis is done to determine the cut-off value at which to split the attribute. This process is repeated at each level until the number of instances classified at the lowest level reaches a specified minimum. Having a large minimum value means that the tree will be strict in creating nodes at the different levels. On the contrary, making this minimum value be small (e.g., 1) will cause many nodes to be added to the tree. To mitigate noise in our predictions and similar to previous studies [126], in our case study, we set this minimum node size to be 10.

To illustrate, we provide an example tree produced by the fix dimension, shown in Figure 4.2. The decision tree indicates that when the time_days variable (i.e., the number of days to fix the defect) is greater than 13.9 and the last status is Resolved Fixed, then the defect will be re-opened. On the other hand, if the time_days variable is less than or equal to 13.9 and the number of files in the fix is less than or equal to 4 but greater than 2, then the defect will not be re-opened. Such explainable models can be leveraged by practitioners to direct their attention to defects that require closer review before they are closed.

Table 4.2: Confusion matrix

	True class	
Classified as	Re-open	Not Re-open
Re-open	TP	FP
Not Re-open	FN	TN

4.3.3 Evaluating the Accuracy of Our Models

To evaluate the predictive power of the derived models, we use the classification results stored in a confusion matrix. Table 4.2 shows an example of a confusion matrix.

We follow the same approach used by Kim *et. al* [153], using the four possible outcomes for each defect. A defect can be classified as re-opened when it truly is re-opened (true positive, TP); it can be classified as re-opened when actually it is not re-opened (false positive, FP); it can be classified as not re-opened when it is actually re-opened (false negative, FN); or it can be classified as not re-opened and it truly is not re-opened (true negative, TN). Using the values stored in the confusion matrix, we calculate the widely used Accuracy, Precision, Recall and F-measure for each class (i.e., re-opened and not re-opened) to evaluate the performance of the predictive models.

The accuracy measures the number of correctly classified defects (both the re-opened and the not re-opened) over the total number of defects. It is defined as:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}. \tag{4.1}$$

Since there are generally less re-opened defects than not re-opened defects, the accuracy measure may be misleading if a classifier performs well at predicting the majority class (i.e., not re-opened defects). Therefore, to provide more insights, we measure the precision and recall for each class separately.

1. **Re-opened precision:** Measures the percentage of correctly classified re-opened defects over all of the defects classified as re-opened. It is calculated as P(re) $= \frac{TP}{TP+FP}$.

2. **Re-opened recall:** Measures the percentage of correctly classified re-opened defects over all of the actually re-opened defects. It is calculated as R(re) $= \frac{TP}{TP+FN}$.

3. **Not re-opened precision:** Measures the percentage of correctly classified, not re-opened defects over all of the defects classified as not re-opened. It is calculated as P(nre) $= \frac{TN}{TN+FN}$.

4. **Not re-opened recall:** Measures the percentage of correctly classified not re-opened defects over all of the actually not re-opened defects. It is calculated as R(nre) $= \frac{TN}{TN+FP}$.

5. **F-measure:** Is a composite measure that measures the weighted harmonic mean of precision and recall. For re-opened defects it is measured as F-measure(re) $= \frac{2*P(re)*R(re)}{P(re)+R(re)}$ and for defects that are not re-opened F-measure(nre) $= \frac{2*P(nre)*R(nre)}{P(nre)+R(nre)}$.

A precision value of 100% would indicate that every defect we classified as (not) re-opened was actually (not) re-opened. A recall value of 100% would indicate that every actual (not) re-opened defect was classified as (not) re-opened.

To estimate the accuracy of the model, we employ 10-fold cross validation [69]. In 10-fold cross validation, the data set is partitioned into 10 sets. Each of the 10 sets contains 1/10 of the total data. Each of the 10 sets is used once for validation (i.e., to test accuracy) and the remaining nine sets are used for training. We repeat this 10-fold cross validation 10 times (i.e., we build 100 decision trees in total) and report the average.

4.4 Data Processing

To conduct our case study, we used three projects: Eclipse Platform 3.0, Apache HTTP Server and OpenOffice. The main reason we chose to study these three projects in our case study is because they are large and mature Open Source Software (OSS) projects that have a large user base and a rich development history. The projects also cover different domains (i.e., Integrated

Development Environment (IDE) vs. Web Server vs. Productivity Suite) and are written in different programming languages (i.e., C/C++ vs. Java). We leveraged two data sources from each project, i.e., the defect database and the source code control (CVS) logs.

To extract data from the bug databases, we wrote a script that crawls and extracts bug report information from the project's online Bugzilla databases. The reports are then parsed and different factors are extracted and used in our study.

Most of the factors can be directly extracted from the bug report, however, in some cases we needed to combine the data in the bug report with data from the CVS logs. For example, one of our factors is the number of files that are changed to implement the defect fix. In most cases, we can use the files included in the submitted patch. However, sometimes the patch is not attached to the bug report. In this case, we search the CVS logs to determine the change that fixed the defect.

We used the J-REX [251] tool, an evolutionary code extractor for Java-based software systems, to perform the extraction of the CVS logs. The J-REX tool obtains a snapshot of the Eclipse CVS repository and groups changes into transactions using a sliding window approach [259]. The extracted logs contain the date on which the change was made, the author of the change, the comments by the author to describe the change and files that were part of the change. To map the defects to the changes that fixed them, we used the approach proposed by Fischer *et al.* [93] and later used by Zimmermann *et al.* [310], which searches in the CVS commit comments for the defect IDs. To validate that the change is actually related to the defect, we make sure that the date of the change is on or prior to the close date of the defect. That said, there is no guarantee that the commits are defect fixes as they may be performing other types of changes to the code.

Table 4.3: Bug report data statistics

	Eclipse	Apache HTTP	OpenOffice
Total Extracted Bug Reports	18,312	32,680	106,015
Resolved Bug Reports	3,903	28,738	86,993
Bug Reports Linked to Code Changes or Patches	1,530	14,359	40,173
Re-opened Bug Reports	246	927	10,572
Not Re-opened Bug Reports	1,284	13,432	29,601

To use the bug reports in our study, we require that they be resolved and contain all of the considered factors in our study. Table 4.3 shows the number of bug reports used from each project. To explain the data in Table 4.3, we use the Eclipse project as an example. We extracted a total of 18,312 bug reports. Of these 18,312 reports, only 3,903 bug reports were resolved (i.e., they were closed at least once). Of the resolved bug reports, 1,530 could be linked to source code changes and/or submitted patches. We use those 1,530 bug reports in our study. Of the 1,530 defects reports studied, 246 were re-opened and 1,284 were not.

For each bug report, we extract 24 different factors that cover four different dimensions, described in Table 4.1. Most of the factors were directly derived from the defect or code databases. However, two factors in the bug report dimension are text-based and required special processing. We apply a Naive Bayesian classifier [193] on the description text and comment text factors to determine keywords that are associated with re-opened and non-reopened defects. For this, we use a training set that is made up of two-thirds randomly selected bug reports. The Bayesian classifier is trained using two corpora that are derived from the training set. One corpus contains the description and comment text of the re-opened defects[2] and the other corpus contains the description and comment text of the defects that were not re-opened. The content of the description and comments are divided into tokens, where each token represents a single word. Since the defect comments often contain different types of text (e.g., code snippets), we did not stem the words or remove stopwords. Prior work

[2]For re-opened defects, we used all the comments posted before the defects were re-opened.

shows that stemming and removing stopwords has very little influence on the final results [14].

The occurrence of each token is calculated and each token is assigned a probability of being attributed to a re-opened or none re-opened defect. These probabilities are based on the training corpus. Token probabilities are assigned based on how far their spam probability is from a neutral 0.5. If a token has never been seen before, it is assigned a probability of 0.4. The reason for assigning a low probability to new tokens is that they are considered innocent. The assumption here is that token associated with spam (or in our case re-opened defects) will be familiar.

The probabilities of the highest 15 tokens are combined into one [104], which we use as a score value that indicates whether or not a defect will be re-opened. The choice of 15 tokens is motivated by prior work (e.g., [126]). Generally, the number of tokens provides a tradeoff between accuracy and complexity (i.e., having too little tokens may not capture enough information and having too many tokens may make the models too complex). A score value close to 1 indicates that the defect is likely to be re-opened and vice versa. The score values of the description and comment text are then used in the decision tree instead of the raw text.

Dealing with imbalance in data: One issue that many real-world applications (e.g., in vision recognition [245], bioinformatics [29] and credit card fraud detection [58]) suffer from is data imbalance. What this means is that one class (i.e., majority) usually appears more than another class (i.e., minority). This causes the decision tree to learn factors that affect the majority class without trying to learn about factors that affect the minority class. For example, in Eclipse the majority class is non-reopened defects which has 1,284 defects and the minority class is re-opened defects, which contains 246 defects. If the decision tree simply predicts that none of the defects will be re-opened, then it will be correct 83.9% of the time (i.e., $\frac{1284}{1530}$). We discuss this observation in more detail in Section 4.6.1.

To deal with this issue of data imbalance, we must increase the weight of the minority class. A few different approaches have been proposed in the literature:

1. **Re-weighting the minority class:** Assigns a higher weight to each bug report of the minority class. For example, in our data, we would give a weight of 5.2 (i.e., $\frac{1284}{246}$) to each re-opened instance.

2. **Re-sampling the data:** Over- and under- sampling can be performed to alleviate the imbalance issue. Over-sampling increases the minority class instances to become at the same level as the majority class. Under-sampling decreases the majority class instances to reach the same level as the minority class. Estabrooks and Japkowicz [83] recommend performing both under- and over-sampling, since under-sampling may lead to useful data being discarded and over-sampling may lead to over-fitted models.

We built models using both re-weighting and re-sampling using the AdaBoost algorithm [95] available in the WEKA machine learning framework [289]. We performed both over- and under-sampling on the training data and predicted using a non-balanced test data set. We did the same using the re-weighting approach. Using re-sampling achieves better prediction results, therefore we decided to only use this in all our experiments. A similar finding was made in previous work [126].

It is important to note here that we re-sampled the training data set only. The test data set was not re-sampled or re-weighted in any way and maintained the same ratio of re-opened to non-re-opened defects as in the original data set.

4.5 Case Study Results

In this section, we present the results of our case study on the Eclipse Platform 3.0, Apache HTTP server and OpenOffice projects. We aim to answer the two research questions posed earlier. To answer the first question we perform a Top Node analysis [118, 246] using each of the dimensions in isolation (to determine the best factors within each dimension) and using all of the dimensions combined (to determine the best factors across all dimensions). Then, we use these dimensions to build decision trees that accurately predict whether or not a defect will be re-opened.

Q1. Which of the extracted factors indicate, with high probability, that a defect will be re-opened?

We perform *Top Node analysis* to identify factors that are good indicators of whether or not a defect will be re-opened. In Top Node analysis, we examine the top factors in the decision trees created during our 10-fold cross validation. The most important factor is always the root node of the decision tree. As we move down the decision tree, the factors become less and less important. For example, in Figure 4.2, the most important factor in the tree is time_days. As we move down to level 1 of the decision tree, we can see that last_state and num_fix_files are the next important factors and so on. In addition to the level of the tree that a factor appears in, the occurrence of a factor at the different levels is also important. The higher the occurrence is, the stronger confidence is of the importance of that factor.

People Dimension

Table 4.4 presents the results of the Top Node analysis for the team dimension. For the **Eclipse** project, the reporter name and the fixer name are the most important factors in the team dimension. Out of the 100 decision trees created (i.e., 10 x 10-fold cross validation), the reporter name is the most important in 51 trees and the fixer name was the most important in the remaining 49 trees. In **Apache** the reporter name is the most important factor in all 100 decision trees created for the team dimension. On the other hand, in **OpenOffice** the fixer name is the most important factor in all 100 decision trees.

Hence, our finding shows that the reporter name and the fixer name are the most important factors in the team dimension. This indicates that some reporters and developers are more likely to have their defects re-opened than others. The fixer experience is also important, ranking highly in level 1 of the decision trees of the Eclipse and OpenOffice projects.

It is important to note that in level 1 of the tree presented in Table 4.4, the frequencies of the attributes sum up to more than 200 (which would be the case when the attributes used were binary). This is because the Fixer name and reporter name variables are of type string and are converted to multiple nominal variables. Therefore, the frequencies of the attributes at level 1 of the tree sum up to more than 200.

This effect also made it hard to understand the concrete effect of the most important factors in each project. For example, a decision tree would say "if developer A is the fixer and the developer experience is $>$ 10 defects, then the defect is re-opened". Another branch of the tree might say "if developer B is the fixer and the developer experience is $>$ 10 defects, then the defect is not re-opened". In such a case, it is difficult to determine the effect of the developer experience factor.

Therefore, in addition to examining the decision trees, we generated logistic regression models and used the coefficients of the factors to quantify the effect of the factor on a defect being re-opened [197]. In particular, we report the odds ratios of the analyzed factors. Odds ratios are the exponent of the logistic regression coefficients and indicate the increase to the likelihood of a defect being re-opened that 1 unit increase of the factor value causes. Odds ratios greater than 1 indicate a positive relationship between the independent (i.e., factors) and dependent variables (i.e., an increase in the factor value will cause an increase in the likelihood of a defect being re-opened). For example, an odds ratio of 1.06 means that for each unit increase in the value of the factor, the likelihood of a defect being re-opened increases by 6%. Odds ratios less than 1 indicate a negative relationship, or in other words, an increase in the independent variable will cause a decrease in the likelihood of the dependent variable. For example, an odds ratio of 0.99 means that for a one unit increase in the value of a factor, the likelihood of a defect being re-opened decreases by 1%.

Different fixer and reporter names were associated with different effects on a defect being re-opened. For the sake of privacy, we do not mention the effect of specific fixer and reporter names, and only discuss the effect of the fixer and reporter experience on a defect being re-opened.

In Eclipse, we found a negative, but weak, effect between the reporter experience (odds ratio 0.99) and developer experience (odds ratio 0.99) and the likelihood of a defect being re-opened. This means that more experienced reporters and developers are less likely to have their defects re-opened. In Apache we also found a negative and weak effect on reporter experience (odds ratio 0.99) and the likelihood of a defect being re-opened. In OpenOffice, we found a negative and weak effect between the developer experience (odds ratio 0.99) and the likelihood of a defect being re-opened.

Table 4.4: Top Node Analysis of the Team Dimension

	Eclipse		**Apache**		**OpenOffice**	
Level	**#**	**Attribute**	**#**	**Attribute**	**#**	**Attribute**
0	51	Reporter name	100	Reporter name	100	Fixer name
	49	Fixer name				
1	315	Fixer experience	1098	Fixer name	1490	Fixer experience
	277	Reporter name	1013	Fixer experience	480	Reporter name
	248	Reporter experience	862	Reporter experience	176	Reporter experience
	202	Fixer name				

Work Habit Dimension

Table 4.5 shows the results of the Top Node analysis for the work habit dimension. In **Eclipse** and **Apache**, the day of the year and the day of the month the defect was closed in, were the most important factors. In the 100 decision trees created, the day of the year was the most important factor 76 and 97 times for Eclipse and Apache, respectively. For Apache, another important factors (i.e., in level 1) is the weekday that the defects were closed in. For **OpenOffice**, the week day factor is the most important factor in 90 of the decision trees built for the OpenOffice project. The time factor was the most important in 10 of the 100 decision trees. Similar to the Eclipse and Apache projects, the day of the year was also important for the OpenOffice project (i.e, in level 1).

Examining the effect of the important factors for Eclipse showed that defects closed later on in the year (i.e., day of the year) are more likely to be re-opened (odds ratio 1.05). We also find that defects reported later in the month are less likely to be re-opened (odds ratio 0.95). In Apache, defects closed later in the year have a negative, but weak, effect on defect re-opening (odds ratio 0.99). In OpenOffice, we found a negative relationship to defect re-opening for defects closed on all days of the week (odds ratios in the range of 0.79 - 0.99), except for Wednesday where there was a positive relationship (odds ratio 1.03). For time, we find a weak and positive relationship between defects closed in the morning (odds ratio 1.07) or at night (odds ratio 1.04) with defect re-opening.

Table 4.5: Top Node Analysis of the Work Habit Dimension

	Eclipse		**Apache**		**OpenOffice**	
Level	**#**	**Attribute**	**#**	**Attribute**	**#**	**Attribute**
0	76	Day of year	97	Day of year	90	Week day
	20	Month day	3	Month day	10	Time
	2	Time				
	1	Week day				
	1	Month day				
1	39	Day of year	73	Week day	412	Day of year
	31	Month day	68	Month day	134	Time
	22	Month	30	Day of year	73	Month
	20	Time	7	Time	28	Month day
	15	Week day	3	Month	23	Week day
	6	Day of year				

Defect Fix Dimension

Table 4.6 presents the Top Node analysis results for the defect fix dimension. For **Eclipse**, the time days factor, which counts the number of days it took from the time the defect was opened until its initial closure (i.e., the time it took to initially resolve the defect), is the most important factor in the fix dimension in 90 of the 100 decision trees. The number of files touched to fix the defect and the last status the defect was in when it was closed were the most important factor in 5 of the 100 trees. In the case of **Apache** and **OpenOffice**, the last status the defect was in when it was closed (i.e., before it was reopened) was the most important factor in all 100 decision trees. Also important are the number of days it took to close the defect (i.e., time days fact) and the number of files in the fix, as shown by their importance in levels 1 of the decision trees.

As for the effect of the factors, in Eclipse we found that there is practically no effect between the number of days it takes to close a defect (odds ratio 1.0) and the likelihood of a defect being re-opened. To put things in perspective, we found that an increase of 365 days, increases the chance of the likelihood of a defect being re-opened by 0.4%. In addition, we found that bugs in the "resolved_duplicate", "resolved_worksforme" and "resolved_invalid" states before their final closure had the strongest chance of being re-opened. This means that when a bug is in any of those three aforementioned states before being closed, it should be closely verified since it is likely that it will be re-opened.

For Apache, bugs in the "resolved_duplicate", "resolved_wontfix", "resolved_invalid", "verified_invalid" and "resolved_worksforme" states were the most likely to be re-opened. In OpenOffice, we found that bugs in the "resolved_duplicate", "verified_wontfix", "verified_invalid" and "verified_worksforme" states prior to being closed were the most likely to be re-opened.

Table 4.6: Top Node Analysis of the DefectFix Dimension

	Eclipse		**Apache**		**OpenOffice**	
Level	**#**	**Attribute**	**#**	**Attribute**	**#**	**Attribute**
0	90	Time days	100	Last status	100	Last status
	5	No. fix files				
	5	Last status				
1	93	No. fix files	261	Time days	293	Time days
	57	Last status	210	No. fix files	62	No. fix files
	38	Time days				

Bug Report Dimension

The Top Node analysis results of the bug report dimension are shown in Table 4.7. For the **Eclipse** project, the comment text content included in the bug report factor is the most important in this dimension across all 10 decision trees.

We examine the words that appear the most in the description and comments of the defects. These are the words the Naive Bayesian classifier associates with re-opened and not re-opened defects. Words such as “control”, “background”, “debugging”, “breakpoint”, “blocked” and “platforms” are associated with re-opened defects. Words such as “verified”, “duplicate”, “screenshot”, “important”, “testing” and “warning” are associated with defects that are not re-opened.

To shed some light on our findings, we manually examined 10 of the re-opened defects. We found that three of these re-opened defects involved threading issues. The discussions of these re-opened defects talked about running processes in the “background” and having “blocked” threads. In addition, we found that defects that involve the debug component were frequently re-opened, because they are difficult to fix. For example, we found comments such as “*Verified except for one part that seems to be missing: I think you forgot to add the...*” and “*This seems more difficult that[than] is[it] should be. I wonder if we can add...*”. Other work shows that security related defects in Firefox are re-opened more than other types of defects [297].

For **Apache**, the description text is the most important factor in the bug report dimension. This finding is contrary to our observation in the Eclipse project, in which the comment text is shown to be the most important factor. However, the comment text is also of importance in the Apache project, appearing in level 1 of the decision trees.

The words that the Naive Bayesian classifier associates with re-opened defects included words such as “cookie”, “session”, “block” and “hssfeventfactory” are associated with re-opened defects. Words such as “attachment”, “message”, “ant”, “cell” and “code” are associated with defects that are not re-opened.

Manual examination of 10 of the re-opened defects shows that four of the re-opened defects in Apache are related to compatibility issues. For example, in one of the defects examined, the defect was re-opened because the initial fix cloned a piece of code but did not modify the cloned code to handle all cases in its context. We found comments that said “*...The fix for this defect does not account for all error cases. I am attaching a document for which the code fails...*”. A later comment said “*...I see what’s missing. We borrowed the code from RecordFactory.CreateRecords but forgot to handle unknown records that happen to be continued...*”.

In another example, the defect was being deleted because it was difficult to fix, and even after a fix was done, it did not fix the defect entirely. One of the comments that reflects the difficulty of the defect says *“...This bug is complex to fix, and for this reason will probably not be fixed in the 4.0.x branch, but more likely for 4.1. This will be mentioned in the release notes...”*. After the defect was initially fixed, a later comment attached by the developer who re-opened the defect says *“While the new session is now being created and provided to the included resource, no cookie is being added to the response to allow the session to be retrieved when direct requests to the included context are recieved...”*

Similar to the Eclipse project, in **OpenOffice** the comment text is the most important factor in the bug report dimension. The description text and the number of comments are also shown to be important, appearing in level 1 of the decision trees.

The Naive Bayesian classifier associates words such as “ordinal”, “database”, “dpcc”, “hsqldb” and “sndfile” with re-opened defects, whereas words such as “attached”, “menu”, “button”, “wizard” and “toolbar” were associated with defects that are not re-opened.

A manual examination of 10 of the re-opened defects shows that seven of the re-opened defects in OpenOffice are related to database access issues. In one of the examined defects, the issue was related to a limitation of the used HSQL database engine, where the reporter says *"Fields (names, data types) in HSQL-based tables cannot be modified after the table has been saved.."*. The defect is closed since this seemed to be a limitation of the HSQL database engine, as reported in this comment *"that would be nice, however I think it's (currently) a limitation of the used hsqldb database engine"*. Later on, support was added and the defect was re-opened and fixed.

Table 4.7: Top Node Analysis of the Bug Report Dimension

	Eclipse		**Apache**		**OpenOffice**	
Level	**#**	**Attribute**	**#**	**Attribute**	**#**	**Attribute**
0	94	Comment text	100	Description text	100	Comment text
	6	Description text				
1	181	Description text	105	Comment text	101	Description text
	16	Comment text	89	No. of comments	84	No. of comments
	1	Severity changed	1	Description size	9	No. in CC list
					5	Comment text
					1	Comment size

All Dimensions

Thus far, we looked at the dimensions in isolation and used Top Node analysis to determine the most important factors in each dimension. Now, we combine all of the dimensions together and perform the Top Node analysis using all of the factors. The Top Node analysis of all dimensions is shown in Table 4.8.

In **Eclipse**, the comment text is determined to be the most important factor amongst all of the factors considered in this study. In addition, the description text content is the next most important factor. For **Apache**, the last status of the defect when it is closed is the most important factor, followed by the description and comment text. Similar to the Eclipse project, in **OpenOffice** the comment text is shown to be the most important factor. The next most important factor is the last status factor, which appears in level 1 of the decision trees.

Table 4.8: Top Node Analysis Across All Dimensions

	Eclipse		**Apache**		**OpenOffice**	
Level	**#**	**Attribute**	**#**	**Attribute**	**#**	**Attribute**
0	96	Comment text	100	Last status	100	Comment text
	4	Description text				
1	180	Description text	280	Description text	200	Last status
	11	Comment text	132	Comment text		
	3	Severity changed	2	Time		
	1	Time	19	Month		
			1	Description size		
			10	Month day		
			8	No. of fix files		
			37	No. of comments		
			1	Severity		
			3	Fixer experience		

The comment text is the most important factor for the Eclipse and OpenOffice projects, while the last status is the most important one for Apache.

Q2. Can we accurately predict whether a defect will be re-opened using the extracted factors?

Following our study on which factors are good indicators of re-opened defects, we use these factors to predict whether a defect will be re-opened. First, we build models that use only one dimension to predict whether or not a defect will be re-opened. Then, all of the dimensions are combined and used to predict whether or not a defect will be re-opened.

Table 4.9 shows the prediction results produced using decision trees. The results are the averages of the 10 times 10-fold cross validation. Since we are reporting the average of 100 runs, we also report the variance to give an indication of how much the results vary across runs. The variance of the measures is shown in brackets besides each average value. Ideally, we would like to obtain high precision, recall and F-measure values, especially for the re-opened defects.

Eclipse: Out of the four dimensions considered, the defect report dimension was the best performing. It achieves a re-opened precision of 51.3%, a re-opened recall of 72.5% and 59.5% re-opened F-measure. The defect report dimension was also the best performer for not-reopened defects; achieving a not re-opened precision of 94.3%, not re-opened recall of 86.2% and 89.9% not re-opened F-measure. The overall accuracy achieved by the defect report dimension is 83.9%. The rest of the dimensions did not perform nearly as well as the defect report dimension.

To put our prediction results for re-opened defects in perspective, we compare the performance of our prediction models to that of a random predictor. Since the re-opened class is a minority class that only occurs 16.1% of the time, a random predictor will be accurate 16.1% of the time. Our prediction model achieves 51.3% precision, which is approximately a three-fold improvement over a random prediction. In addition, our model achieves a high recall of 72.5%.

Apache: The defect fix dimension is the best performing dimension. It achieves a re-opened precision of 40.1%, a re-opened recall of 89.8% and F-measure of 55.4%. The defect fix dimension also achieves the best not re-opened precision of 99.2%, recall of 90.7% and F-measure of 94.7%. The overall accuracy of the defect fix dimension is 90.6%.

Combining all of the dimensions improves the re-opened precision to 52.3%, the re-opened recall to 94.1% and the re-opened F-measure to 67.2%. Furthermore, combining all of the dimensions improves the not re-opened precision to 99.6%, recall to 94.1% and F-measure to 96.7%. The overall accuracy is improved to 94.0%.

In the case of Apache, re-opened defects appear in only 6.5% of the total data set. This means that our re-opened precision improves over the random precision by more than 8 times.

At the same time, we are able to achieve a very high recall of 94.1%.

OpenOffice: Similar to the Eclipse project, the bug report dimension is the best performing dimension for the OpenOffice project. The re-opened precision is 63.4%, the re-opened recall is 87.3% and the re-opened F-measure is 71.3%. The not re-opened precision is 93.0%, recall of 83.2% and not re-opened F-measure of 87.6%. The overall accuracy of the bug report dimension is 82.7%.

Using all of the dimensions in combination improves the re-opened precision to 78.6% (a three-fold improvement over a random predictor), the re-opened recall to 89.3% and the re-opened F-measure to 83.6%. The not re-opened precision improves to 95.9%, the not re-opened recall improves to 91.3% and the not re-opened F-measure improves to 93.6%. The overall accuracy improves to 90.8%.

Final remarks

As shown in Table 4.9, the precision varies across projects. For example, for Eclipse the precision is 52.1%, whereas for OpenOffice it is 78.6%. One factor that influences the precision value of prediction models is the ratio of re-opened to not re-opened defect reports [190]. This ratio is generally used a baseline precision value [169, 255]. Table 4.3 shows that the baseline precision for Eclipse is $\frac{246}{1530} = 16.1\%$, whereas the baseline precision for OpenOffice is $\frac{10572}{40173} = 26.3\%$. Therefore, we expect our prediction models to perform better in the case of OpenOffice compared to Eclipse. Another factor that influences the variation in prediction results is the fact that we are using the same factors for all projects. In certain cases, some factors may perform better for some projects than others.

Overall, our results show that fairly accurate prediction models can be created using a combination of the four dimensions. However, although combining all of the dimensions provides a considerable improvement over using the best single dimension for the Apache and OpenOffice projects, it only provides a slight improvement for the Eclipse project. Having a predictor that can perform well without the need to collect and calculate many complex factors makes it more attractive for practitioners to adopt the prediction approach in practice.

We can build explainable prediction models that can achieve a precision of 52.1-78.6% and a recall of 70.5-94.1% recall when predicting whether a defect will be re-opened and a precision between 93.9-99.6% and recall of 86.8-94.1% when predicting if a defect will not be re-opened.

Table 4.9: Prediction results

	Dimension	Re-opened Precision	Re-opened Recall	Re-opened F-measure	Not Re-reopened Precision	Not Re-opened Recall	Not Re-opened F-measure	Accuracy
Eclipse	Team	18.3(0.13) %	45.5(1.1) %	25.9(0.26) %	85.4(0.05) %	61.0(0.29) %	71.0(0.15) %	58.5(0.19) %
	Work habit	21.8(0.14) %	54.1(1.2) %	30.9(0.28) %	87.7(0.07) %	62.5(0.36) %	72.8(0.19) %	61.2(0.23) %
	Fix	18.9(0.06) %	68.2(2.0) %	29.5(0.17) %	88.2(0.13) %	44.1(0.94) %	58.1(0.70) %	47.9(0.46) %
	Bug	**51.3(0.74) %**	**72.5(0.81) %**	**59.5(0.48) %**	**94.3(0.03) %**	**86.2(0.22) %**	**89.9(0.07) %**	**83.9(0.15) %**
	All	52.1(0.99) %	70.5(0.69) %	59.4(0.55) %	93.9(0.02) %	86.8(0.21) %	90.2(0.07) %	84.2(0.15) %
Apache	Team	8.4(0.01) %	46.7(0.29) %	14.4(0.02) %	94.7(0.00) %	65.3(0.04) %	77.2(0.02) %	64.1(0.03) %
	Work habit	7.0(0.01) %	38.3(0.32) %	11.9(0.03) %	93.9(0.00) %	65.1(0.02) %	76.9(0.01) %	63.4(0.02) %
	Fix	**40.1(0.08) %**	**89.8(0.12) %**	**55.4(0.08) %**	**99.2(0.00) %**	**90.7(0.01) %**	**94.7(0.00) %**	**90.6(0.01) %**
	Bug	28.5(0.06) %	73.3(0.18) %	40.9(0.08) %	97.9(0.00) %	87.2(0.02) %	92.2(0.01) %	86.3(0.02) %
	All	52.3(0.09) %	94.1(0.05) %	67.2(0.07) %	99.6(0.00) %	94.1(0.01) %	96.7(0.00) %	94.0(0.00) %
OpenOffice	Team	57.5(0.02) %	71.9(0.03) %	63.8(0.01) %	88.8(0.00) %	81.0(0.01) %	84.8(0.00) %	78.6(0.01) %
	Work habit	50.2(0.01) %	75.6(0.03) %	60.3(0.01) %	89.4(0.00) %	73.2(0.02) %	80.5(0.01) %	73.9(0.01) %
	Fix	44.0(0.01) %	**87.3(0.04) %**	58.5(0.01) %	**93.0(0.01) %**	60.3(0.04) %	73.1(0.02) %	67.4(0.01) %
	Bug	**63.4(0.01) %**	81.4(0.02) %	**71.3(0.01) %**	92.6(0.00) %	**83.2(0.01) %**	**87.6(0.00) %**	**82.7(0.00) %**
	All	78.6(0.02) %	89.3(0.01) %	83.6(0.01) %	95.9(0.00) %	91.3(0.01) %	93.6(0.00) %	90.8(0.00) %

4.6 Discussion

4.6.1 Comparison with Other Prediction Algorithms

So far, we used decision trees to predict whether a defect will be re-opened. However, decision trees are not the only algorithm that can be used. Naive Bayes classification and Logistic regression are two very popular algorithms that have been used in many prediction studies (e.g., [228]). In this section, we compare the prediction results of various prediction algorithms that can be used to predict whether or not a defect will be re-opened. In addition, we used the prediction from the Zero-R algorithm as a baseline for the prediction accuracy. The Zero-R algorithm simply predicts the majority class, which is not re-opened in our case.

The prediction results using the different algorithms are shown in Table 4.10. Since different algorithms may provide a tradeoff between precision and recall, we use the F-measure to compare the different prediction algorithms. As expected, the Zero-R algorithm achieves the worst performance, since it does not detect any of the re-opened defects (it basically predicts the majority class). The Naive Bayes algorithm performs better in some cases. For example, for the Eclipse project the Naive Bayes algorithm achieves a re-opened recall of 73.9% and not re-opened precision of 94.5%. The Logistic Regression model performs slightly worse achieving re-opened F-measure of 53.8% (precision: 45.3%, recall: 67.2%) and not re-opened F-measure of 88.1% (precision: 93.1%, recall: 84.1%). Decision trees perform slightly worse (in some cases) than Naive Bayes for Eclipse, Apache and OpenOffice. More importantly however is that decision trees provide explainable models. Practitioners often prefer explainable models since it helps them understand why the predictions are the way they are.

Table 4.10: Results using different prediction models

	Algorithm	Re-opened Precision	Re-opened Recall	Re-opened F-measure	Not Re-reopened Precision	Not Re-opened Recall	Not Re-opened F-measure	Accuracy
Eclipse	Zero-R	NA	0 %	0 %	83.9(0.00) %	100(0.00) %	**91.3(0.00) %**	83.9(0.00) %
	Naive Bayes	49.0(0.33) %	**73.9(0.81) %**	58.7(0.34) %	**94.5(0.03) %**	85.1(0.10) %	89.5(0.03) %	83.3(0.08) %
	Logistic Reg	45.3(0.41) %	67.2(0.77) %	53.8(0.35) %	93.1(0.03) %	84.1(0.15) %	88.1(0.15) %	81.4(0.10) %
	C4.5	**52.1(0.99) %**	70.5(0.69) %	**59.4(0.55) %**	93.9(0.02) %	**86.8(0.21) %**	90.2(0.07) %	**84.2(0.15) %**
Apache	Zero-R	NA	0 %	0 %	93.5(0.00) %	100(0.00) %	96.7(0.00) %	93.5(0.00) %
	Naive Bayes	46.5(0.10) %	69.8(0.35) %	55.7(0.10) %	97.8(0.01) %	**94.4(0.01) %**	96.1(0.00) %	92.8(0.00) %
	Logistic Reg	46.5(0.15) %	78.3(0.23) %	58.2(0.14) %	98.4(0.01) %	93.7(0.01) %	96.0(0.00) %	92.7(0.01) %
	C4.5	**52.3(0.09) %**	**94.1(0.05) %**	**67.2(0.07) %**	**99.6(0.00) %**	94.1(0.01) %	**96.7(0.00) %**	**94.0(0.00) %**
OpenOffice	Zero-R	NA	0 %	0 %	73.7(0.00) %	100(0.00) %	84.8(0.00) %	73.7(0.00) %
	Naive Bayes	48.7(0.06) %	**90.5(0.02) %**	63.3(0.04) %	95.1(0.00) %	65.7(0.13) %	77.7(0.06) %	72.3(0.06) %
	Logistic Reg	69.8(0.01) %	88.9(0.01) %	78.2(0.01) %	95.6(0.00) %	86.3(0.01) %	90.7(0.00) %	86.9(0.00) %
	C4.5	**78.6(0.02) %**	89.3(0.01) %	**83.6(0.01) %**	**95.9(0.00) %**	**91.3(0.01) %**	**93.6(0.00) %**	**90.8(0.00) %**

4.6.2 Commit vs. Defect Work Habits

Our work habits factors are based on the time that the defect was initially closed. The reason for using the time the defect was initially closed was due to the fact that we wanted to investigate whether developers were more inclined to close defects during specific times (e.g., to reduce their work queue). However, another side that may contribute to defect re-opening is the work habits factors of the commit or change performed to fix the defect. For example, commits made at specific times may be associated with a higher chance of a defect being re-opened later on.

Table 4.11: Commit work habits factors

Dim	Factor	Type	Explanation	Rationale
Change work habits	Change time	Nominal	Time in hours (Morning, Afternoon, Evening, Night) when the change to address the defect was made.	Changes made to address defects at certain times in the day (e.g., late afternoons) are more/less likely to be re-opened.
	Change week-day	Nominal	Day of the week (e.g., Mon or Tue) when the change to address the defect was first made.	Changes made to address defects on specific days of the week (e.g., Fridays) are more/less likely to be re-opened.
	Change month day	Numeric	Calendar day of the month (0-30) when the change to address the defect was made.	Changes made to address defects at specific periods like the beginning, mid or end of the month are more/less likely to be re-opened.
	Change month	Numeric	Month of the year (0-11) when the change to address the defect was made.	Changes made to address defects in specific months (e.g., during holiday months like December) are more/less likely to be re-opened.
	Change day of year	Numeric	Day of the year (1-365) when the change to address the defect was made.	Changes made to address defects in specific times of the year(e.g., later on in the year) are more/less likely to be re-opened.

To examine the effect of the commit work habits dimension, we extract the same factors for the work habits dimension, but now for each commit instead of for each bug report. The factors are shown in Table 4.11. Since not all bugs could be linked to a commit and we need all of the factors to perform our analysis, our dataset reduced in size. For Eclipse, we were able to link 1,144 bugs where 187 were re-opened and 957 were not. For OpenOffice, we were able to link 19,393 bugs where 7181 were re-opened and 12,212 were not. For Apache were only able to link 278 bug reports. After examination of the linked data, we decided to perform the analysis for Eclipse and OpenOffice, but not for Apache (i.e., due to the low number of linked bug reports).

First, we redo the top node analysis of the work habits factors, this time including both defect closing and commit work habits. The results are shown in Table 4.12. To be clear, we explicitly label each factor with its association to defects or commits, shown in brackets. We observe that for both projects, the work habits factors from the initial defect closure are the most important factors. In particular, the day of the year and the month were the most important. Commit work habits factors are placed in level 1 of the decision trees, showing that they are also important, but clearly less important than the initial defect closure.

To examine whether including the commit work habit factors improves prediction accuracy, we present the prediction results in Table 4.13. For each project, the first row presents the prediction results when the defect work habits are only considered. The second row presents the prediction results when the commit and defect work habits factors are combined. We see that for both, Eclipse and OpenOffice, the prediction results improve. For Eclipse, we see an improvement of 3.8% in re-opened precision and 4.5% in re-opened recall, whereas, for OpenOffice we see an improvement in prediction results of 19.6% in re-opened precision and 15.4% improvement in re-opened recall.

Table 4.12: Top node analysis of the commit and defect work habit dimension

	Eclipse		OpenOffice	
Level	**#**	**Attribute**	**#**	**Attribute**
0	66	(Bug) Day of year	100	(Bug) Day of year
	21	(Bug) Month		
	5	(Bug) Time		
	3	(Commit) Month		
	2	(Bug) Week day		
	2	(Commit) Time		
	1	(Commit) Month day		
1	16	(Bug) Week day	106	(Commit) Day of year
	5	(Commit) Month day	94	(Commit) Month
	11	(Bug) Month		
	25	(Commit) Month		
	36	(Bug) Day of year		
	9	(Commit) Weekday		
	12	(Bug) Time		
	20	(Bug) Month day		
	37	(Commit) Day of year		
	14	(Commit) Time		

Table 4.13: Prediction results when considering commit work habits

	Dimension	Re-opened Precision	Re-opened Recall	Re-opened F-measure	Not Re-reopened Precision	Not Re-opened Recall	Not Re-opened F-measure	Accuracy
Eclipse	Work habit (defect only)	20.7(0.17) %	50.6(1.2) %	29.2(0.32) %	86.5(0.07) %	61.7(0.47) %	71.8(0.26) %	59.9(0.32) %
	Work habit (defect and commit)	24.5(0.23) %	55.1(1.5) %	33.8(0.43) %	88.4(0.08) %	66.7(0.33) %	75.9(0.17) %	64.7(0.24) %
OpenOffice	Work habit (defect only)	62.7(0.04) %	75.4(0.06) %	68.4(0.02) %	83.6(0.01) %	73.5(0.07) %	78.2(0.02) %	74.2(0.02) %
	Work habit (defect and commit)	82.3(0.02) %	87.3(0.02) %	84.7(0.01) %	92.3(0.00) %	88.9(0.01) %	90.6(0.00) %	88.3(0.01) %

4.7 Threats to Validity

In this section, we discuss the possible threats to validity of our study.

Threats to Construct Validity consider the relationship between theory and observation, in case the measured variables do not measure the actual factors. To identify re-opened defects, we used the bug reports in the Buzilla bug tracking system. In certain cases, re-opened defects may not be reported in the bug tracking system. Some of the re-opened defects considered in our study were re-opened more than once. In such cases, we predict for the first time the defect was re-opened. In future studies, we plan to investigate defects that are re-opened several times.

One of the attributes used in the People dimension is the fixer name. We extracted the names of the fixers from the committed CVS changes. In certain cases, the fixer and the committer of the changes are two different people. In the future, we plan to use heuristics that may improve the accuracy of the fixer name factor.

For the work habits dimension, we use the dates in the Bugzilla bug tracking system. These times refer to the server time and may not be the same as the local user time.

Threats to Internal Validity refers to whether the experimental conditions makes a difference or not, and whether there is sufficient evidence to support the claim being made.

The percentage of bug reports that met the prerequisites to be included in our study is small (e.g., for Eclipse we were able to extract a total of 18,312 bug reports, of which 1,530 met our prerequisites). At first glance, this seems to be a low bug reports used to extracted bug reports ratio. However, such a relatively low ratio is a common phenomenon in studies using bug reports [228, 282]. In addition, we would like to note that the percentage of open-to-reopened defects in the used data set and the original data set are quite close. For example, in the Eclipse project, 16.1% of the bug reports were re-opened, whereas 10.2% of the bug reports were re-opened in the original data set.

We use 24 different factors that cover four dimensions to predict re-opened defects. Although this set of factors is large, it is only a subset of factors that may be used to predict re-opened defects. Other factors such as social networks factors for example can be used to further improve the prediction results.

Bird *et al.* [41] showed that bias due to imperfect linking between historical databases is common and may impact the performance of prediction techniques. To mitigate such issues, we used the state-of-the-art techniques to link the data from different repositories.

Threats to External Validity consider the generalization of our findings. In this study, we used three large, well established Open Source projects to conduct our case study. Although these are large open source projects, our results may not generalize (and as we have seen do not generalize) to all open source or commercial software projects.

We use decision trees to perform our prediction and compared our results to 3 other popular prediction algorithms. Decision trees performed well, for all three projects, compared to the 3 algorithms we compared with, however, using other prediction algorithms may produce different results. One major advantage to using decision trees is that they provide explainable models that practitioners can use to understand the prediction results.

In our manual examination of the re-opened defects, we only examined a very small sample of 10 bug reports. The purpose of this analysis was to shed some light on the type of information that we were able to get from the re-opened bug reports. Our findings may not generalize to all re-opened defects.

4.8 Related Work

We survey the state-of-the-art in SDP in Chapter 2. In this section, we discuss the work that is most closely related to this Chapter. We divide the related work into two parts: the work related to the used dimension and work related to bug report quality and triage.

4.8.1 Work Related to Used Dimensions

The work closest to this work is the work by Zimmermann *et al.* [308] which characterizes and predicts re-opened bugs in Windows. The authors perform a qualitative and quantitative study and find that some of the reasons for bug reopens are the fact that bugs were difficult to reproduce, developers misunderstood the root cause, bug reports had insufficient information and the fact that priority of the bug may have been initially underestimated. Through their quantitative analysis, the authors find that bugs reported by customers or found during system testing are more likely to be re-opened. Also, bugs that are initially assigned to someone on a different team or geographic location are more likely to be re-opened. In many ways their paper complements our study since we both focus on bug reopens. However, our study is done on OSS projects, whereas Zimmermann *et al.* use commercial systems. Also, their study surveys Microsoft developers and provides more insight about the reasons for bug re-opens at Microsoft.

Work habit dimension: Anbalagan and Vouk [13] and Marks *et al.* [180] studied the time it takes to fix a defect. In a case study performed a case study on the Ubuntu Linux distribution and showed that the day of the week on which a defect was reported impacts the amount of time it will take to fix the defect [13]. Śliwerski *et al.* citeSliwerski2005MSR measured the frequency of defect introducing changes on different days of the week. Through a case study on the Eclipse and Mozilla projects, they showed that most defect introducing changes occur on Fridays. Hassan and Zhang [118] used the time of the day, the day of the week and the month day to predict the certification results of a software build and Ibrahim *et al.* [126] used the time of the day, the week day and the month day that a message was posted to predict whether a developer will contribute to that message. Eyolfson *et al.* [85] examine the effect of time of day and developer experience on commit bugginess in two open source projects. The authors find that approximately 25% of commits are buggy, that commits checked in between 00:00 and 4:00 AM are more likely to be buggy, developers who commit on a daily basis write less-buggy commits and bugginess for commits per day of the week vary for different projects. No prediction was performed. In a recent study, Zimmermann et al. [308] characterize when defects are re-opened in Microsoft Windows. They find that defects found during system testing and defects found by customers are most likely to be re-opened. They also found that when defects are assigned to people in a different location (from where it was found)

The work habit dimension extracts similar information to those used in the aforementioned related work. However, our work is different in that we use the information to investigate whether these work habit factors affect the chance of a defect being re-opened.

Bug report dimension: Mockus *et al.* [198] and Herraiz *et al.* [121] used information contained in bug reports to predict the time it takes to resolve defects. For example, in [198], the authors showed that in the Apache and Mozilla projects, 50% of the defects with priority P1 and P3 were resolved within 30 days and half of the P2 defects were resolved within 80 days. On the other hand, 50% of the defects with priority P4 and P5 took much longer to resolve (i.e., their resolution time was in excess of 100 days). They also showed that defects logged against certain components were resolved faster than others.

Similar to the previous work, we use the information included in bug reports, however, we do not use this information to study the resolution time of a defect. Rather, we use this information to predict whether or not a defect will be re-opened.

Defect fix dimension: Hooimeijer *et at.* [125] built a model that measures bug report quality and predicts whether a developer would choose to fix the bug report. They used the total number of attachments that are associated with bug reports as one of the features in the model. Similarly, Bettenburg *et al.* [37] used attachment information to build a tool that recommends to reporters how to improve their bug report. Hewett and Kijsanayothin [123] used the status of a defect (e.g., Worksforme) as one of the features to model the defect resolution time.

Similar to the previous studies, we use information about the initial defect fix as input into our model, which predicts whether or not a defect will be re-opened.

People dimension: Schröter *et al.* [249] analyzed the relationship between human factors and software reliability. Using the Eclipse defect dataset, they examined whether specific developers were more likely to introduce defects than others. They observed a substantial difference in defect densities in source code developed by different developers. Anvik *et al.* [16] and Jeong *et al.* [127] were interested in determining which developers were most suitable to resolve a defect.

We use the names and the experience of the defect reporters and fixers to predict whether or not a defect will be re-opened. Although our Chapter is similar to other previous work in terms of the used factors, to the best of our knowledge, this Chapter is the first to empirically analyze whether or not a defect will be re-opened.

4.8.2 Work on Bug Report Quality and Triage

Antoniol *et al.* [15] use decision trees, naive bayes and logistic regression to correctly classify issues in bug tracking systems as defects or enhancements. Bettenburg *et al.* [37] investigate what makes a good bug report. They find that there is a mismatch between information that developers need (i.e., stack traces, steps to reproduce and test cases) and what users supply. Aranda and Venolia [18] report a field study of coordination activities related to defect fixing. They find that data stored in repositories can be incomplete since they rarely take into account social, organizational and technical knowledge. Bettenburg *et al.* [35] examine the usefulness of duplicate bug reports and find that duplicate bug reports contain valuable information that can be combined with other bug reports. Guo *et al.* [108] chaterize factors that affect whether a defect is fixed in Windows Vista and Windows 7. They find that defects reported by people with better reputation, and on the same team or within the same geographic proximity are more likely to get fixed.

Another line of work aims to assist in the defect triaging process. This work focused on predicting who should be assigned to fix a particular defect [16], the analysis of bug report reassignments [127] and predicting the severity of bug reports [167]. In contrast to prior work, in this work we focus on re-opened defects.

4.9 Conclusion

Re-opened bugs increase maintenance costs, degrade the overall user-perceived quality of the software and lead to unnecessary rework by busy practitioners. Therefore, practitioners are interested in identifying factors that influence the likelihood of a bug being re-opened to better deal with, and minimize the occurrence of re-opened bugs. In this chapter, we used information extracted from the bug and source code repositories of the Eclipse, Apache and OpenOffice open source projects to derive 24 different factors, which make up four different dimensions, to predict whether or not a bug will be re-opened. We performed Top Node analysis to determine which factors are the best indicators of a bug being re-opened. The Top Node analysis showed that the factors that best indicate re-opened bugs depends on the project. The comment text is the most important factor for the Eclipse and OpenOffice projects, while the last status is the most important one for Apache. In addition, we provide insight about the way in which the important factors impact the likelihood of a bug being re-opened. Then, with the derived factors, we can build explainable prediction models that can achieve 52.1-78.6% precision and 70.5-94.1% recall when predicting whether a bug will be re-opened. The findings of this work contribute towards better understanding of what factors impact bug re-openings so they can be examined more carefully. Doing so will reduce the number of re-opened bugs and the maintenance costs associated with them.

In this Part of the thesis, we proposed approaches that focus on predicting high-impact defects in order to tackle the limitation of not considering impact of defects. In the next part, we propose approaches that tackle the issue of providing guidance on how to make use of SDP research results.

Part II

Providing Guidance on Using SDP Results

We believe that a limitation of current SDP research is its lack to provide guidance on what to do once the predictions are made. Often, the proposed prediction models are too complex for practitioners to understand. Furthermore, no guidance on how to use the results of the SDP models is given. It is often left up to the practitioners to decide what to do. Practitioners need to be provided with guidance on what to do once the predictions are made. This can be achieved by making SDP models as simple as possible and by tailoring SDP approaches to address a specific scenario. This way practitioners will be able to easily understand the simple models, as well as, know which scenarios these models are best applied in.

In this Part of the thesis, we present two approaches that show how SDP approaches can address the issue of providing guidance on using SDP results:

- **Simplifying and Understanding SDP Models [Chapter 5]:** In order to make SDP models easy to understand, we reduce the number of independent variables (i.e., predictors) in SDP models. To show the value of our approach, we replicate a study done by Zimmermann *et al.* [310] and were able to reduce the number of independent variables in the SDP models from 34 to only 4, while maintaining comparable precision and recall values.

  The main recommendations based on the findings of this Chapter are:

  - Practitioners should use a small number of metrics in their prediction models since it makes the prediction models simple and easy to understand.
  - Using a small number of metrics can achieve prediction and explanative powers similar to more complex models. On a case study using the Eclipse project, using 3 or 4 metrics achieves precision and recall values that are comparable to more complex models that use 34 metrics.

- **Prioritizing the Creation of Unit Tests [Chapter 6]:** We present an approach to prioritize the creation of unit tests in large software systems. We show how SDP results can be more applicable in practice by using SDP techniques to flag functions that need to have unit test created for them. We conduct a study on two large software systems, a commercial system and the Eclipse open source project. We find that factors based on the function size, modification frequency and defect fixing frequency should be used to prioritize the creation of unit tests.

The main recommendations based on the findings of this Chapter are:

- Practitioners can leverage the history of a project to effectively prioritize the creation of unit tests for their large software projects.
- Using historical data can achieve a three-fold improvement over a naive strategy that randomly selects functions to write.
- Performing a case study on a large commercial and open source project, we find that the size of a function should be used to prioritize the creation of unit tests.

This Part shows how SDP results can be simplified, understood and applied in practice. It also shows how SDP can be applied to assist in testing efforts, in particular unit testing, which is one of the most widely used testing techniques in industry today [287].

Chapter 5

Simplifying and Understanding SDP Models

Research studying the quality of software applications continues to grow rapidly with researchers building regression models that combine a large number of factors. However, these models are hard to deploy in practice due to the cost associated with collecting all the needed factors, the complexity of the models and the black box nature of the models. For example, techniques such as PCA merge a large number of factors into composite factors that are no longer easy to explain. In this chapter, we use a statistical approach recently proposed by Cataldo et al. *to create explainable regression models. A case study on the Eclipse open source project shows that only 4 out of the 34 code and process factors impacts the likelihood of finding a post-release defect. In addition, our approach is able to quantify the impact of these factors on the likelihood of finding post-release defects. Finally, we demonstrate that our simple models achieve comparable performance over more complex PCA-based models while providing practitioners with intuitive explanations for its predictions.*

5.1 Introduction

A large portion of software development costs is spent on maintenance and evolution activities [82, 196]. Fixing software defects is one area that takes up a large amount of this maintenance effort. Therefore, practitioners and managers are always looking for ways to reduce the defect fixing effort. In particular, they are interested in identifying which parts of the software contain defects, to achieve short term goals, such as prioritizing the testing efforts for the following releases. In addition, practitioners are interested in understanding the main factors that impact these defects, to achieve longer term goals, such as driving process improvement initiatives to deliver better quality software.

An extensive body of work has focused on finding the fault-prone locations in software systems. The majority of this work builds prediction models that predict where future defects are likely to occur. The work varies in terms of the domains covered (i.e., open source [116, 310] vs. commercial software [55,209,210]), in terms of the factors used to predict the defects (i.e., using process [116] vs. code factors [310]) and in terms of the types of defects it aims to predict for (i.e., pre-release [209] vs. post-release [310] or both [257]).

At the same time, these prediction models are becoming more and more complex over time. New studies are investigating more aspects that may help improve prediction accuracy, which leads to more factors (i.e., independent variables) being input to the prediction models.

Although adding more factors (i.e., independent variables) to the prediction models may increase the overall prediction accuracy, it also introduces some negative side effects. First, it makes the models more complex and therefore, makes them less desirable to adopt in practical settings. Second, adding more independent variables to the prediction model makes determining which independent variables *actually* produce the effect (i.e., impact) on post-release defects more complicated (due to multicollinearity [205]). The aforementioned problems turn the complex prediction models into black-box solutions that can be used to know where the defects are, but make it difficult to know the underlying reasons for what impacts the defects.

The black-box nature of such models defect prediction is a major hurdle in the adoption of these models in practice.

The goal of our study is to use the code and process factors previously used to build complex prediction models and to narrow this set of factors to a much smaller number that can be used in a logistic regression model to *understand what impacts* post-release defects. Understanding what impacts post-release defects can be leveraged by practitioners to drive process changes, build better tools (e.g., monitoring tools) and drive future research efforts.

We apply a statistical approach, recently proposed by Cataldo *et al.* [55], to identify statistically significant and minimally collinear factors (i.e., independent variables in a logistic regression model) that impact post-release defects. In addition, we use odds ratios to quantify the impact of these independent variables on the dependent variable, post-release defects. For example, we quantify the increase in the likelihood of finding post-release defects if a file increases in size by 100 lines.

We formalize our work in the following research questions:

Q1 Which code and process factors impact the post-release defects? Do these factors differ for different releases of Eclipse?

Q2 By how much do the factors impact the post-release defects? Does the level of impact change across different releases?

To evaluate our approach, we perform a case study on the Eclipse project. We were able to identify a small set (3 or 4 out of 34) of the independent variables that explain the majority of the impact on the dependent variable, post-release defects.

We also examine the predictive and explanative powers of the models built using the minimal set of independent variables. The results show that the simple logistic regression models built using our approach (i.e. using the small set of 3 or 4 independent variables) achieve prediction and explanative results that are comparable to the more complex models that use the full set (i.e., 34) of independent variables.

5.1.1 Organization of Chapter

Section 5.2 presents the related work. Section 5.3 describes our approach and the data used in our study. We present the case study results and the research questions posed in Section 5.4 and follow with a discussion and comparison of the models built using our approach in Section 5.5. Section 5.6 presents the threats to validity and Section 5.7 concludes the chapter.

5.2 Related Work

We survey the state-of-the-art in SDP in Chapter 2. In this section, we discuss the work that is most closely related to this Chapter. The majority of the relevant related work comes from the area of defect prediction. Most of these efforts build multivariate logistic regression models (e.g., [55, 62, 116, 310]) to predict faulty locations (e.g., files or directories). We divide the related work into two categories, based on the type of factors used: code factors and process factors.

5.2.1 Using Code Factors

Ohlsson and Alberg [222] use factors that were automatically derived from design documents to predict fault-prone modules. Their set of factors also included McCabe's cyclomatic complexity factor [181]. They performed their case study on a Ericsson Telecom software system and showed that based on design and fault data, one can build accurate prediction models, even before any coding starts. Basili *et al.* [30] used the Chidamber and Kemerer (CK) factors suite [59] to predict class fault-proneness in 8 medium-sized information management systems. Subramanyam and Krishnan [263] performed a similar study (using the CK factors) on a commercial system and Gyimothy *et al.* [109] performed a similar analysis on Mozilla. Their studies confirmed the findings by Basili's study. El Emam *et al.* [78] used the CK factors, in addition to Briand's coupling factors/metrics [49] to predict faulty classes. They reported high prediction accuracy of faulty classes using their employed factors. Nagappan and Ball [209] used a static analysis tool to predict the pre-release defect density of the Windows Server 2003. In another study, Nagappan *et al.* [212] predicted post-release defects, at the module level, using source code factors. They used 5 different Microsoft projects to perform their case study and found that it is possible to build prediction models for an individual project, but no single model can perform well on all projects. Zimmermann *et al.* [310] extracted an extensive set of source code factors and used them to predict post-release defects.

The majority of the work that use code factors to predict defects leverage multiple factors. Some of this work recognize the possibility of multicollinearity problems, and therefore, employ PCA (e.g. [209, 212]). Although PCA may reduce the complexity of the prediction model, it does not necessarily reduce the number of factors that need to be collected. In addition, once PCA is applied, it is difficult to directly map the independent variables to the dependent variable, since the PCs are linear combinations of many independent variables.

5.2.2 Using Process Factors

Other work uses process factors, such as the number of prior defects or prior changes to predict defects. Graves *et al.* [105] showed that the number of prior changes to a file is a good predictor of defects. They also argued that change data is a better predictor of defects than code factors in general. Studies by Arisholm and Briand [19] and Khoshgoftaar *et al.* [143] also reported that prior changes are a good predictor of defects in a file. Hassan [116] used the complexity of a code change to predict defects. He showed that prior faults is a better predictor of defects than prior changes. He then showed that using the entropy of changes is a good predictor of defects. Moser *et al.* [204] showed that process factors perform better than code factors, to predict post-release defects in Eclipse. They also reported that for the Eclipse project, pre-release defects seem to perform extremely well in predicting post-release defects. Yu *et al.* [295] also showed that prior defects are a good indicator of future defects.

The previous work using process factors highlighted two factors that seemed to perform well: prior changes and prior defect fixing changes. In our work, we annotated Zimmermann's Eclipse data set with the well-known historical predictors of defects, prior changes and prior defect fixing changes.

Our work differs from previous work, in that we focus on studying the impact of code and process factor on post-release defects, rather than predict where the post-release defects are. We use statistical techniques to identify a small, statistically significant and minimally collinear, set of the *original* factors that impact post-release defects. We also quantify the impact of these factors on post-release defects.

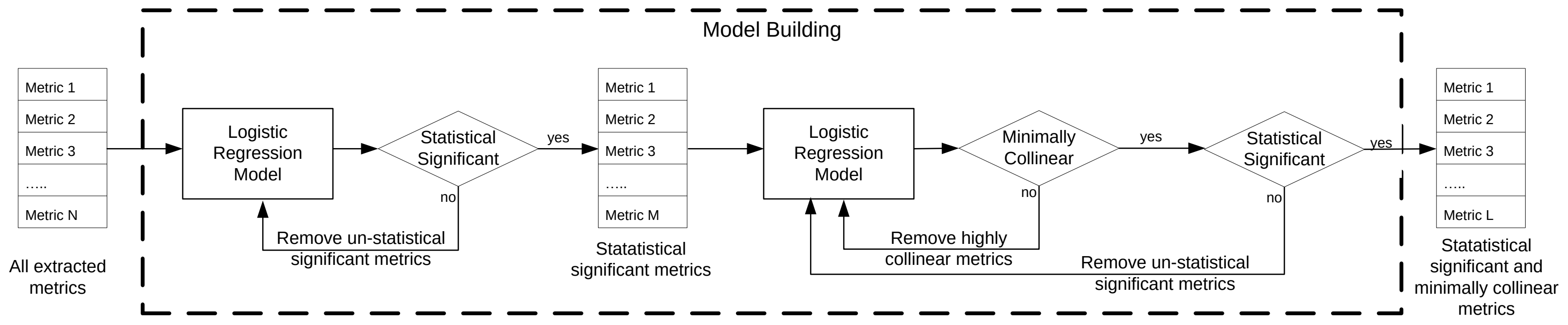

Model Building
Metric 1
Metric 2
Metric 3
…..
Metric N
All extracted metrics
Logistic Regression Model
Statistical Significant
yes
no
Remove un-statistical significant metrics
Metric 1
Metric 2
Metric 3
…..
Metric M
Statatistical significant metrics
Logistic Regression Model
Minimally Collinear
yes
no
Remove highly collinear metrics
Statistical Significant
yes
no
Remove un-statistical significant metrics
Metric 1
Metric 2
Metric 3
…..
Metric L
Statatistical significant and minimally collinear metrics

Figure 5.1: Overview of our proposed approach

5.3 Approach

In this section, we detail the steps of our approach, shown in Figure 6.1. Our approach is inspired by the previous work by Cataldo *et al.* [55]. In a nutshell, our approach takes as input an extensive list of all code and process factors (34 factors). Then, we build a logistic regression model and analyze the statistical significance and collinearity characteristics of the independent variables (i.e., factors) used to build the model. We eliminate the statistically insignificant and highly collinear independent variables, which leaves us with a much smaller (3 or 4 factors) set of statistically significant and minimally collinear independent variables. The small set of factors is then used to build a final logistic regression model, which we use to understand the impact of these factors on post-release defects.

Each step of our approach is discussed in detail in the following subsections.

5.3.1 Collection and Description of Input Factors

We use a number of factors to study their impact on post-release defects. We acquired the latest version (i.e., 2.0a) of the publicly available Eclipse data set provided by Zimmermann *et al.* [310]. Zimmermann's data set contain a number of code factors, as well as pre and post-release defects. We annotate the data set with the well-known process factors: the total number of prior changes (TPC) and prior defect fixing changes (DFC).

The process factors were extracted from the CVS [267] repository of Eclipse. We used the J-REX [251] tool, a code extractor for Java-based software systems, to extract the annotated process factors. The J-REX tool obtains a snapshot of the Eclipse CVS repository and groups changes into transactions using a sliding window approach [259]. The CVS commit comments of the changes are examined and key words such as "bug", "fix", etc. are used to identify the defect fixing changes. A similar approach was used by Moser *et al.* [204] to classify defect fixing changes.

A total of 34 different factors (shown in Table 5.2) were extracted for three different releases of Eclipse – versions 2.0, 2.1 and 3.0. All of the extracted factors were mapped to the software locations at the file level. We list the factors used and provide a brief description in the following subsections.

Process Factors

Numerous previous studies by [105,116,204] show that process factors perform well to predict software defects. In this subsection, we highlight the process factors used in our study.

1. Total Prior Changes (TPC): Measures the total number of changes to a file in the 6 months before the release. Previous work by Moser *et al.* [204] and Graves *et al.* [105] showed that the total number of changes is a good indicator of future defects.

2. Prior Defect Fixing Changes (DFC): The number of defect fixing changes done to a file in the 6 months before the release. This factor is extracted from the CVS repository exclusively. Previous work by Yu *et al.* [295] and Hassan [116] showed that the number of previous defect fixing changes is a good indicator of future defects.

3. Pre-release defects (PRE): The number of pre-release defects in a file in the 6 months before the release. Zimmerman *et al.* used a pattern matching approach that searches for defect identification numbers in source control change comments and used the defect identifiers to classify the changes as defect fixing changes [310].

4. Post-release defects (POST): The number of post-release defects in a file in the 6 months after the release [310]. This factor is used as the dependent variable in our logistic regression models.

Code Factors

An extensive set of code factors was obtained from the Promise data set provided by Zimmermann *et al.* [310]. The majority of the factors are complexity factors that have been successfully used in the past [204, 310] to predict post-release defects. We list, and briefly explain the different code factors used in our study:

1. Total Lines of Code (TLOC): Measures the total number lines of code of a file.

2. Fan out (FOUT): Measures the number of method calls of a file. Three measures are provided for FOUT, avg, max and total.

3. Method Lines of Code (MLOC): Measures number of method lines of code. Three measures are provided for MLOC, avg, max and total.

4. Nested Block Depth (NBD): Measures the nested block depth of the methods in a file. Three measures are provided for NBD, avg, max and total.

5. Number of Parameters (PAR): Measures the number of parameters of the methods in a file. Three measures are provided for PAR, avg, max and total.

6. McCabe Cyclomatic Complexity (VG): Measures the McCabe cyclomatic complexity of the methods in a file. Three measures are provided for VG, avg, max and total.

7. Number of Fields (NOF): Measures the number of fields of the classes in a file. Three measures are provided for NOF, avg, max and total.

8. Number of Methods (NOM): Measures the number of methods of the classes in a file. Three measures are provided for NOM, avg, max and total.

9. Number of Static Fields (NSF): Measures the number of static fields of the classes in a file. Three measures are provided for NSF, avg, max and total.

10. Number of Static Methods (NSM): Measures the number of static methods of the classes in a file. Three measures are provided for NSM, avg, max and total.

11. Anonymous Type Declarations (ACD): Measures the number of anonymous type declarations in a file.

12. Number of Interfaces (NOI): Measures the number of interfaces in a file.

13. Number of Classes (NOT): Measures the number of classes in a file.

Table 5.1: Example using Eclipse 3.0 factors

	Iteration 1		**Iteration 2**			**Iteration 7**		**Iteration 8**	
Factor	**P-value**	**VIF**	**P-value**	**VIF**	...	**P-value**	**VIF**	**P-value**	**VIF**
TLOC	7.72e-05 ***	27.754974	1.36e-07 ***	13.819629	...	< 2e-16 ***	1.366347	<2e-16 ***	1.362840
PRE	< 2e-16 ***	1.319629	< 2e-16 ***	1.289907	...	< 2e-16 ***	1.275472	<2e-16 ***	1.244094
TPC	0.073345 +	2.782466	0.04729 *	2.733476	...	0.06941 +	2.698888	0.0056 **	1.093982
DFC	1.08e-05 ***	2.825389	3.93e-06 ***	2.782801	...	2.27e-06 ***	2.739084	-	-
ACD	0.087893 +	1.654916	0.00066 ***	1.437178	...	< 2e-16 ***	1.222093	0.0178 *	1.216477
FOUT avg	0.841697	46.585807	-	-	...	-	-	-	-
FOUT max	0.382543	26.176911	-	-	...	-	-	-	-
FOUT sum	0.948411	89.959363	-	-	...	-	-	-	-
MLOC avg	0.204205	105.763769	-	-	...	-	-	-	-
MLOC max	0.330231	42.238222	-	-	...	-	-	-	-
MLOC sum	0.055794 +	211.302843	0.25559	28.082202	...	-	-	-	-
NBD avg	0.092112 +	193.777683	0.09060 +	156.684860	...	-	-	-	-
NBD max	0.169100	12.477594	-	-	...	-	-	-	-
NBD sum	0.053443 +	1421.056428	0.03509 *	1206.186455	...	-	-	-	-
NOF avg	0.328731	206.270137	-	-	...	-	-	-	-
NOF max	0.229602	137.421047	-	-	...	-	-	-	-
NOF sum	0.109088	256.810067	-	-	...	-	-	-	-
NOI	0.654592	90.361904	-	-	...	-	-	-	-
NOM avg	0.138702	236.364314	-	-	...	-	-	-	-
NOM max	0.712747	154.720922	-	-	...	-	-	-	-
NOM sum	0.383255	210.771232	-	-	...	-	-	-	-
NOT	0.622797	87.947327	-	-	...	-	-	-	-
NSF avg	0.312735	61.944164	-	-	...	-	-	-	-
NSF max	0.762748	654.599118	-	-	...	-	-	-	-
NSF sum	0.582049	608.061575	-	-	...	-	-	-	-
NSM avg	0.619926	50.435605	-	-	...	-	-	-	-
NSM max	0.832793	625.599123	-	-	...	-	-	-	-
NSM sum	0.970193	544.627797	-	-	...	-	-	-	-
PAR avg	0.646235	10.135751	-	-	...	-	-	-	-
PAR max	0.003790 **	3.793562	0.41411	1.471249	...	-	-	-	-
PAR sum	0.115339	32.672658	-	-	...	-	-	-	-
VG avg	0.020032 *	376.900050	0.05768 +	310.938499	...	-	-	-	-
VG max	0.917081	23.922610	-	-	...	-	-	-	-
VG sum	0.015354 *	1928.733922	0.02532 *	1513.000445	...	-	-	-	-

(p<0.001 ***; p<0.01 **; p < 0.05 *; p<0.1 +)

5.3.2 Model Building

We are interested in finding out the files that are likely to have one or more post-release defects. Logistic regression models are generally used for this purpose. A logistic regression model correlates independent variables with a discrete dependent variable. In our case, the independent variables are the collection of code and process factors and the dependent variable is a two-value variable that represents whether or not a file has one or more post-release defects. The model outputs the likelihood of a file to have one or more post-release defects.

We use the *glm* command in the R statical package [234] to build the logistic regression model. R provides us with a few tools that we can use to analyze the statistical characteristics of the model we build. We leverage these tools to study the statistical significance and collinearity attributes of the independent variables used to build the model.

Initially, we build a multivariate logistic regression model using all 34 factors as the independent variables. Then, we perform an iterative process where we remove the statistically insignificant independent variables. Next, we perform a similar iterative process to remove highly collinear independent variables from the logistic regression model. This process is repeated until we reach a model that only contains statistically significant and minimally collinear independent variables.

Statistical Significance Analysis

We perform an analysis using R to study the statistical significance of each independent variable. We use the well known p-value to determine the statistical significance. Since some of the independent variables can have no statistically significant effect on the likelihood of post-release defects, including them in the prediction model may improve its overall prediction accuracy, but makes it difficult to claim that they produce the effect that we are observing.

We remove all of the independent variables that have a p-value greater than a specified threshold value. In this chapter, we retained all independent variables with p-value < 0.1. At the end of this step, the multivariate logistic regression model only contains independent variables that are statistically significant.

Collinearity Analysis

Multicollinearity can be caused by high intercorrelation between the independent variables. The problem with multicollinearity is that as the independent variables become highly correlated, it becomes more difficult to determine which independent variable is actually producing the effect on the dependent variable. In addition to making it difficult to determine the independent variable that is causing the effect, multicollinearity causes higher standard error. Therefore, it is beneficial to minimize collinearity within the independent variables of the logistic regression model.

Tolerance and Variance Inflation Factor (VIF) is often used to measure the level of multicollinearity. A tolerance value close to 1 means that there is little multicollinearity, whereas a tolerance value close to 0 indicates that multicollinearity is a threat. The VIF is the reciprocal of the tolerance. We used the `vif` command in the `Design` package for R to examine the VIF values of all independent variables used to build the multivariate logistic regression model. In this chapter, we set the maximum VIF value to be 2.5, as suggested in [55].

Once we narrow down to only having statistically significant and minimally collinear independent variables, we use these variables to build the final logistic regression model.

To give an overview of the entire process, we provide an example of the multivariate logistic regression model built for Eclipse 3.0 in the next subsection.

Table 5.2: Descriptive statistics of examined factors

Factor	**Mean**	**SD**	**Min**	**Max**	**Skew**	**Kurtosis**
TLOC	123.3	233.4	3.0	4886.0	6.8	79.3
PRE	0.7	2.1	0	43	7.3	81.0
TPC	1.4	3.6	0	82.0	7.5	99.2
DFC	0.45	1.6	0	44.0	9.6	157.5
ACD	0.46	1.7	0	56.0	8.9	164.1
FOUT avg	3.0	3.6	0	60.2	3.0	22.3
FOUT max	11.2	17.0	0	334.0	5.2	55.1
FOUT sum	44.2	95.9	0	2162.0	6.3	55.1
MLOC avg	5.7	6.6	0	159.2	4.2	48.9
MLOC max	20.7	34.9	0	995.0	9.0	168.9
MLOC sum	83.9	190.5	0	4266.0	7.6	96.9
NBD avg	1.3	0.81	0	7.0	0.26	1.0
NBD max	2.4	1.9	0	17.0	0.72	0.47
NBD sum	16.9	29.7	0	621.0	5.9	62.3
NOF avg	2.5	6.3	0	355.0	24.8	1120.2
NOF max	2.9	6.7	0	355.0	20.6	848.8
NOF sum	3.1	7.1	0	355.0	18.0	681.5
NOI	0.16	0.37	0	1	1.8	1.3
NOM avg	8.5	13.3	0	284.0	7.2	90.4
NOM max	9.5	14.8	0	284.0	6.2	67.3
NOM sum	10.2	16.2	0	290.0	6.1	63.4
NOT	0.84	0.38	0	6.0	-1.4	5.1
NSF avg	2.1	18.5	0	1254.0	40.5	2287.0
NSF max	2.3	18.5	0	1254.0	39.9	2242.9
NSF sum	2.3	18.5	0	1254.0	39.9	2239.9
NSM avg	1.1	14.3	0	845.0	47.0	2443.7
NSM max	1.2	14.4	0	845.0	46.1	2380.5
NSM sum	1.2	14.4	0	845.0	46.0	2375.4
PAR avg	0.97	0.76	0	9.0	2.3	10.9
PAR max	2.3	1.8	0	30.0	2.2	13.9
PAR sum	12.1	41.7	0	2100.0	32.3	1392.9
VG avg	1.9	2.0	0	68.5	7.0	155.9
VG max	5.8	10.6	0	310.0	11.8	246.6
VG sum	28.5	61.9	0	1479.0	7.7	100.0

Table 5.3: Factor correlations

Factor	**PRE**	**TLOC**	**TPC**	**DFC**
POST	0.381	0.33	0.128	0.181
TLOC	0.421	1.00	0.210	0.228
PRE	1.000	0.42	0.248	0.328
TPC	0.248	0.21	1.000	0.695
DFC	0.328	0.23	0.695	1.000
ACD	0.258	0.44	0.123	0.163
FOUT avg	0.313	0.77	0.144	0.162
FOUT max	0.375	0.87	0.185	0.203
FOUT sum	0.400	0.94	0.191	0.214
MLOC avg	0.314	0.80	0.152	0.155
MLOC max	0.380	0.90	0.185	0.195
MLOC sum	0.403	0.96	0.200	0.214
NBD avg	0.303	0.74	0.158	0.171
NBD max	0.368	0.85	0.192	0.210
NBD sum	0.392	0.95	0.197	0.219
NOF avg	0.242	0.60	0.087	0.102
NOF max	0.256	0.63	0.098	0.114
NOF sum	0.260	0.63	0.102	0.118
NOI	-0.160	-0.55	-0.022	-0.064
NOM avg	0.296	0.71	0.171	0.182
NOM max	0.314	0.74	0.184	0.196
NOM sum	0.319	0.76	0.186	0.200
NOT	0.160	0.55	0.022	0.064
NSF avg	0.174	0.33	0.079	0.109
NSF max	0.186	0.35	0.088	0.119
NSF sum	0.186	0.35	0.089	0.120
NSM avg	0.197	0.34	0.059	0.073
NSM max	0.202	0.35	0.063	0.075
NSM sum	0.202	0.35	0.063	0.075
PAR avg	0.094	0.26	0.076	0.044
PAR max	0.257	0.60	0.143	0.130
PAR sum	0.350	0.82	0.198	0.200
VG avg	0.300	0.78	0.136	0.139
VG max	0.359	0.87	0.170	0.178
VG sum	0.389	0.95	0.190	0.205

5.3.3 Example Using Our Approach

The multivariate logistic regression model used for Eclipse 3.0 is depicted in Table 5.1. We include the first 2 and last 2 iterations in the table.

We start by building the model using all 34 independent variables. An examination of the model statistics reveals that only 11 of the 34 factors are statistically significant, as shown in the iteration 1 column of Table 5.1. We removed the statistically insignificant independent variables and re-built the model. Once again, we examine the statistically significance of the independent variables. This time, we observe that another 2 of the independent variables become statistically insignificant, shown in the iteration 2 column of Table 5.1. We continued this process of removing the statistically insignificant independent variables, rebuilding the model, re-examining the significance until all independent variables in the model were significant. This was achieved after the fourth iteration.

Then, we remove all independent variables that had a VIF value greater than 2.5 [55]. Each time a variable is removed, we made sure to check the p-values of all independent variables left in the model to assure they are still statistically significant. We did this for 4 more iterations. In the eighth iteration, we finally end up with a logistic regression model that contains 4 statistically significant and minimally collinear independent variables. The final model for the Eclipse 3.0 release only contain ACD, PRE, TPC and TLOC as independent variables.

5.4 Case Study Results

We perform a study on three different revisions of the Eclipse project. We want to examine our approach on Eclipse and identify the independent variables that produce the impact and quantify by how much they impact post-release defects. We also examine the evolution of these independent variables by building and comparing the logistic regression models for 3 different releases of Eclipse.

5.4.1 Preliminary Analysis of Data

Table 5.4: VIF and p-values of code and process factors in Eclipse 2.0, 2.1 and 3.0

	Eclipse 2.0		**Eclipse 2.1**		**Eclipse 3.0**	
Factor	**P-value**	**VIF**	**P-value**	**VIF**	**P-value**	**VIF**
ACD					0.0178 *	1.216477
NSM avg			0.000266 ***	1.096400		
PAR max	5.25e-08 ***	1.289606				
PRE	< 2e-16 ***	1.178974	< 2e-16 ***	1.096400	<2e-16 ***	1.244094
TPC	< 2e-16 ***	1.084916			0.0056 **	1.093982
TLOC	< 2e-16 ***	1.392259	< 2e-16 ***	1.417422	<2e-16 ***	1.362840

(p<0.001 ***; p<0.01 **; p < 0.05 *; p<0.1 +)

Before delving into the results of our case study, we perform some preliminary analysis on the collected factors. We calculated a few of the most common descriptive statistics, mean, min, max, standard deviation (SD) which are reported in Table 5.2. In addition, we calculated the skew and kurtosis measures for each factor.

Skew measures the amount of asymmetry in the probability distribution of a variable, in relation to the normal distribution. Skew can have a positive or negative value. A positive skew value indicates that the distribution is positively skewed, meaning the factor values are mostly on the low end of the scale. In contrast, a negative skew value indicates a negatively skewed distribution, where most of the factor values are on the high end of the scale. The normal distribution has a skew value of 0.

Kurtosis on the other hand characterizes the relative peakedness or flatness of a distribution, in relation to the normal distribution. A positive kurtosis value indicates a curve that is too tall and a negative kurtosis value indicates a curve that is too flat. A normal distribution has a kurtosis value of 0.

It is important to study the descriptive statistics of the factors used to better understand the dataset at hand and perform any needed transformations. Most real world data have high to moderate skew and kurtosis values and transformations such as log or square root transformations are usually employed (e.g., [55, 257, 310]).

We can observe from Table 5.2 that most of the factors suffer from positive skew (i.e., all the factor values are on the low scale) and have positive kurtosis values (i.e., too tall). To alleviate some of the issues caused by these higher than expected skew and kurtosis values, we log transformed all of the factors. From this point on, whenever we mention a factor, we actually are referring to the log transformation of the factor.

In addition, Table 5.3 calculates the pairwise correlation measures of all the factors against the factors that are known to perform well in defect prediction. First, we observe that the PRE and TLOC factors have higher correlation with POST than the change based TPC and DFC factors. Furthermore, we observe that the TLOC factor is highly correlated with the majority of the code factors, especially, the FOUT, MLOC, NDB, NOF and VG factors. Similar observations were made by Graves *et al.* [105]. The high correlation values can be used as an indication of possible multicollinearity problems that may arise if these independent variables were combined in a single logistic regression model.

Q1: Which code and process factors impact the post-release defects? Do these factors differ for different releases of Eclipse?

To answer this question, we followed the same steps outlined in our approach section. We build the models for the 3 different releases, Eclipse 2.0, Eclipse 2.1 and Eclipse 3.0. The results are presented in Table 5.4. The results indicate that using this approach, we are able to successfully build models for all 3 releases. In all 3 releases, all of the independent variables are statistically significant, with p-value < 0.1 and minimally collinear, with VIF values $<$ 2.5. The Eclipse 2.0 and 3.0 models are composed of 4 different independent variables, while the Eclipse 2.1 model contains only 3 independent variables.

Next, we investigate whether these independent variables are the same for the different releases of Eclipse or whether they change from one release to the other. Our findings indicate that some of the independent variables change for different releases. These are mainly the code factors (i.e., ACD for Eclipse 3.0, NSM avg for Eclipse 2.1 and PAR max for Eclipse 2.0). The TPC factor is in 2 of the 3 models, while, the TLOC and the PRE factors were apparent in all 3 models. This finding is interesting because it shows that the simple factors (i.e., TLOC and PRE) are actually the most stable independent variables in our model.

In the next subsection, we quantify the impact by each independent variable on the post-release defects.

Using the p-value and the VIF measures, we are able to determine which of the code and process factors impact post-release defects. These factors change for different releases of Eclipse.

Q2: By how much do the factors impact the post-release defects? Does the level of impact change across different releases?

We would like to quantify the impact caused by the independent variables on post-release defects. For example, what if a file has 3 pre-release defects versus 4 pre-release defects. It would make intuitive sense that the chance of a post-release defect will increase, but by how much? Will an increase in 1 pre-release defect double the chance of a post-release defect?

To quantify the impact, we use odds ratios. Odds ratios are the exponent of the logistic regression coefficients. Odds ratios greater than 1 indicate a positive relationship between the independent and dependent variables (i.e., an increase in the independent variable will cause an increase in the likelihood of the dependent variable). Odds ratios less than 1 indicate a negative relationship, or in other words, an increase in the independent variable will cause a decrease in the likelihood of the dependent variable.

The value of the odds ratios indicate the amount of increase that 1 unit increase of the independent variable will cause to the dependent variable. Since we log all of the independent variables, 1 unit increase means a unit increase in the log-scale. We list the odds ratios of Eclipse 2.0, Eclipse 2.1 and Eclipse 3.0 in Tables 5.5, 5.6 and 5.7, respectively. The table lists the χ^2, p-value and deviance explained of each model. The last row of the table lists the difference of the deviance explained in percent. The models are built in a hierarchical way, meaning we build starting with 1 independent variable and keep adding the independent variables until the final model is built.

Table 5.5: Logistic regression model for Eclipse 2.0

	Model 1	**Model 2**	**Model 3**	**Model 4**
TLOC	2.57***	2.40***	2.11***	1.88***
TPC		1.87***	1.62***	1.62***
PRE			1.87***	1.90***
PAR max				1.73***
Model χ^2	979	1255	1375	1404
Model p-value	<0.001	<0.001	<0.001	<0.001
Deviance Explained	17.6%	22.5%	24.7%	25.2%
Model Comparison (%)	-	276 (4.9%)	120 (2.2%)	29 (0.5%)

(p<0.001 ***; p<0.01 **; p < 0.05 *; p<0.1 +)

Table 5.6: Logistic regression model for Eclipse 2.1

	Model 1	**Model 2**	**Model 3**
TLOC	2.05***	1.49***	1.44***
PRE		3.27***	3.27***
NSM avg			1.21***
Model χ^2	605	944	957
Model p-value	<0.001	<0.001	<0.001
Deviance Explained	11.2%	17.5%	17.7%
Model Comparison (%)	-	339 (6.3%)	13 (0.2%)

(p<0.001 ***; p<0.01 **; p < 0.05 *; p<0.1 +)

Table 5.7: Logistic regression model for Eclipse 3.0

	Model 1	**Model 2**	**Model 3**	**Model 4**
TLOC	2.25***	1.30***	1.66***	1.70***
TPC		2.15***	1.10**	1.11**
PRE			3.24***	3.28***
ACD				0.87*
Model χ^2	1289	1347	1877	1883
Model p-value	<0.001	<0.001	<0.001	<0.001
Deviance Explained	14.5%	15.2%	21.1%	21.2%
Model Comparison (%)	-	58 (0.7%)	530 (5.9%)	6 (0.1%)

(p<0.001 ***; p<0.01 **; p < 0.05 *; p<0.1 +)

Firstly, we examine the odds ratios of Eclipse 3.0, depicted in Table 5.7. It can be observed that as we add factors to the logistic regression model, the odds ratios change. This means that as we add more independent variables to the model, the impact of the individual independent variables will vary. For example, we can see that TLOC has an odds ratio of 2.25 in Table 5.7, model 1. This changes to 1.70 in model 4. This means that if we were *only* using the TLOC variable to build the logistic regression model, then increasing the total lines of code by 1 log unit, increases the likelihood of having a post-release defect by 125% (i.e., more than double the likelihood). On the other hand, if we combined the TLOC variable with other independent variables (as we did in model 4), then the likelihood of finding a post-release defect due to a 1 unit increase in TLOC is only 70%. However, we can see from the same table that as we add more of the independent variables to the model, the deviance explained and the χ^2 value increase significantly, indicating a significant improvement in the explanative power of the model.

Using model 4 in Table 5.7, we can see that for TLOC , TPC and PRE the odds ratios suggest an increase in the likelihood of a post-release defect. On the other hand, a log unit increase in number of anonymous declaration types (ACD) produces a negative impact on the likelihood of finding one or more post-release defects. In fact, for every unit increase in ACD, the chance of finding a post-release defect decreases by 13%.

Table 5.8: Comparison of precision, recall and accuracy of prediction models

	Eclipse 2.0		**Eclipse 2.1**		**Eclipse 3.0**	
	Ours	**All factors**	**Ours**	**All factors**	**Ours**	**All factors**
Precision (%)	66.3	63.6	60.0	58.6	64.1	64.7
Recall (%)	28.5	32.4	15.8	17.2	25.7	26.5
Accuracy (%)	87.5	87.5	89.8	89.8	86.9	87.0

We can also use the odds ratios value to examine the impact on post-release defects if a file was to increase by 100 lines. To do so, we exponentiate the odds ratio value for that independent variable by the quantity increase in units. For example, if TLOC was increased by 100 lines, and since we are using log this would be 2 units (i.e. log(100) = 2)). Therefore, the impact of a 100 line increase, using model 4 in Table 5.7, is $1.70^2 = 2.89$. This means that if we increase the TLOC by 100 lines, then the chance of a post-release defect is increased by 189%.

Comparing the odds ratios of the independent variables for different releases, we can see that the odds ratios change slightly. At the same time, we can observe that they are stable, meaning, if they are above 1 for one of the releases, they stay that way for all the other releases. For example, the odds ratio value for TLOC is positively associated with post-release defects in all 3 releases. This finding was also observed by Briand et al. [51]. A similar observation holds for the PRE factor as well. Studying the stability is important because the stability of a factor tells us whether we can draw conclusions about the impact produced by the factor for this project.

We are able to quantify the impact produced by the code and process factors on post-release defects using odds ratios. The impact on post-release defects changes for different releases of Eclipse.

5.5 Discussion

The main goal of our study is to minimize the large number of independent variables in the multivariate logistic regression model, in order to better understand the impact of the various independent variables on the dependent variable, post-release defects. In the previous section, we performed a case study where our initial set of factors is 34 and we were able to reduce that number to 3 or 4 statistically significant and minimally collinear set of factors.

Although we were successful in using our approach to understand the impact of the process and code factors on post-release defects, a few questions still linger: *How does our approach affect the prediction accuracy of the model and how does it compare to previous techniques used to deal with the issue of multicollinearity, namely PCA?* We answer these questions in the following subsections.

5.5.1 Comparing Prediction Accuracy

To study the prediction accuracy, we build two multivariate logistic regression models: one that uses all of the factors and one that uses the smaller set of statistically and minimally collinear factors. The logistic regression models predict the likelihood of a file being defect prone or not. The output of the model is given as a value between 0 and 1. We classified files with predicted likelihoods above 0.5 as defect prone, otherwise they are classified as being defect free.

The classification results of the prediction models were stored in a confusion matrix, as shown in Table 5.9.

Table 5.9: Confusion matrix

		True Class	
		Defect	No Defect
Predicted	Defect	a	b
	No Defect	c	d

The performance of the prediction model is measured using three different measures:

1. **Precision:** Relates number of files predicted *and* observed as defect prone to the number of files predicted as defect prone. It is calculated as $\frac{a}{a+b}$.
2. **Recall:** Relates number of files predicted *and* observed as defect prone to the number of files that actually had defects. It is calculated as $\frac{a}{a+c}$.
3. **Accuracy:** Relates the total number of correctly classified files to the total number of files. It is calculated as $\frac{a+d}{a+b+c+d}$.

The prediction results for all 3 release of Eclipse are presented in Table 5.8. The results in the table agree with previous results obtained by Zimmermann *et al.* in [310]. In all releases, our results were very close to those generated by the model that uses all 34 factors. It is important to highlight that the main goal of our approach is not to achieve better prediction results. Our main goal is to understand the impact of the independent variables on post-release defects. We proved that we are able to study the impact without significantly affecting the prediction accuracy of the models. Furthermore, it is important to note here that our models are far less complex, using only 3 or 4 factors.

Using a much smaller set of statistically significant and minimally collinear set of factors does not significantly affect the prediction results of the logistic regression model.

5.5.2 Comparing with Principle Component Analysis

Multicollinearity is caused by using highly correlated independent variables, which makes it more and more difficult to determine which one of the independent variables is producing the effect on the dependent variable. Previous research (e.g., [62, 209, 210, 212]) addressed the multicollinearity problem by employing Principal Component Analysis (PCA) [135]. PCA uses the original factors to build Principal Components (PCs) that are orthogonal to each other. The PCs are linear combinations of the factors. These PCs are then used as the independent variables in the logistic regression model.

Although using PCA solves the issue of multicollinearity, it has its disadvantages as well. First, PCA does not necessarily reduce the number of independent variables, since each PC is a linear combination of all the input factors. For this reason, models that use PCA may still need to collect many input factors. Second, once the PCs are used to predict defects, it is very difficult to pin-point which of the original factors (used to build the PCs) actually produced the effect. Not being able to pin-point which of the input factors actually caused the effect is a major disadvantage. It makes it much harder for practitioners and managers to understand the prediction models, causing them to disregard the models or search somewhere else for answers.

Table 5.10: Comparison with PCA

	Eclipse 2.0				**Eclipse 2.1**				**Eclipse 3.0**			
Cumulative Variability	**Ours**	**95%**	**99%**	**100%**	**Ours**	**95%**	**99%**	**100%**	**Ours**	**95%**	**99%**	**100%**
Model χ^2	1404	1375	1487	1552	957	710	981	979	1886	1451	1924	1953
No. of factors	4	34	34	34	3	34	34	34	4	33	33	33
Min. no. of PCs	-	7	15	32	-	7	15	33	-	8	15	33
Model p-value	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Deviance Explained	25.2%	24.7%	26.7%	27.9%	17.7%	13.1%	18.1%	18.1%	21.2%	16.3%	21.7%	22.0%
Comparison (%)	-	+0.5%	-1.5%	-2.7%	-	+4.6%	-0.4%	-0.4%	-	+4.9%	-0.5%	-0.8%

In this section, we compare the results of the models generated using our approach to models that we build using PCA. We perform this comparison to verify the validity of our approach and measure its performance in comparison to models that would be built with all 34 factors.

To build the PCA models, we input all of the independent variables and build the PCs. Then, we measure the % cumulative variation when a different number of PCs is used. Based on the % of cumulative variation we wish to achieve, we use a different number of PCs as input to the logistic regression model. For example, in Eclipse 3.0, to achieve a % cumulative variation of 95%, we would require a minimum of 8 PCs. To achieve a 99% cumulative variation, we require a minimum of 15 PCs. A 95% cumulative variation was commonly used in the previous work on defect prediction [62, 66, 209, 210, 212].

The main reason previous work used 95% cumulative variation was to reduce the data needed to build the models. For example, for Eclipse 3.0 using 8 PCs instead of 34 PCs, converts to a data reduction of 76.5%. However, as we will show, this does not necessarily mean that less factors need to collected, as the number of factors used for Eclipse 3.0 is 33 out of 34. This is a data reduction of 2.9%. For the sake of comparison, we compare the models generated using our approach, to PCA-based models that can achieve 95%, 99% and 100% cumulative variation.

To compare, we use 4 different measures:

1. We measure the χ^2 value achieved by the different models.

2. We report the percentage of deviance explained for each model to examine the explanative power of the models.
3. We record the number of PCs required to achieve the various levels of cumulative proportions of variance. The number of independent variables required to build the PCA-based models is also reported.
4. We calculate the p-value of the models generated to make sure that the models are statistically significant.

Our comparison is reported in Table 5.10. The last row of the table represents the difference in deviance explained in comparison to our model. A positive value means our model outperforms the PCA-based model, and vice versa. In all 3 releases, our model can outperform the PCA-based models with 95% cumulative probability of variance in terms of χ^2 value and deviance explained. As we stated earlier, most of the previous work (e.g., [62, 66, 209, 210, 212]), used the PCA-based models with 95% cumulative variation. Furthermore, we can see that our model uses far less factors than the PCA models. To calculate the number of factors required for the PCA-models, we examined the 'loadings' of the PCs. Hair *et al.* [111] suggested that loading values below 0.4 are considered to have a low rank in the PCs. We counted the number of factors that have a loading value greater than 0.4. As shown in Table 5.10, this number was 34 factors for release 2.0, 34 factors for release 2.1 and 33 factors for release 3.0. The only factor that did not have a loading value greater than 0.4 is the VG sum factor in Eclipse 3.0.

Finally, we would like to point out two main advantages of our models. First, our models require far less factors, meaning that there is a significant amount of savings in the effort that needs to be put into the extraction of these factors. Second, and most importantly, our models are simple and explainable. They can be used by practitioners to *understand* the impact of the independent variables on post-release defects. This is not easily achieved with PCA-based models.

Using our approach, we are able to build logistic regression models that can be used to understand the impact of the code and process factors on post-release defects. Our models can achieve better explanative power than PCA-based models that explain 95% cumulative variation.

5.6 Threats to Validity

Threats to Internal Validity refers to whether the experimental conditions makes a difference or not, and whether there is sufficient evidence to support the claim being made. The list of factors used in our study to build the logistic regression models is by no means complete. Therefore, using other factors may yield different results. However, we believe that the same approach can be applied on any list of factors.

The VIF and p-value thresholds used in our study were chosen because they proved to be successful in previous studies [55]. Other cutoff values may be used for the VIF and p-value and may yield slightly different results.

Threats to External Validity consider the generalization of our findings. Our analysis is based on 3 different releases of the Eclipse project. Although the Eclipse project is a large open source project, our results may not generalize to other projects.

5.7 Conclusion

A large amount of effort has been put into prediction models that aim to find the locations (i.e., files or folders) of defects in a software system. As this area of research grows, a greater number of factors is being used to predict defects. This increase in factors increases the complexity of the prediction models, decreasing the chance of their adoption in practice. In addition, increasing the number of factors increases the chance of multicollinearity, which makes it difficult to determine which of these factors *actually* impact post-release defects.

We put forth an approach that reduces the number of factors to a much smaller, statistically significant and minimally collinear set. The small set of factors are then used to build logistic regression models. We use odds ratios to quantify the impact of the various independent variables (i.e., code and process factors) on the dependent variable, post-release defects.

Finally, we compared the prediction accuracy of the models built using our approach to models that use the full set of factors. We found very little difference in the prediction accuracy, yet our models used significantly less factors. We also compared the explanative power of the logistic regression models using our approach and found that models built using our approach can outperform PCA-based models that explain 95% cumulative variation, and perform within 2.7% of PCA-based models that explain 100% cumulative variation.

We believe that it is important to take the cost of each metric (e.g., cost of extracting the metric) into consideration. Therefore, in the future, we plan to extend this work to consider the cost of a metric when narrowing down the number of metrics to use in the model.

In the following chapter, we present an approach that shows how simple models can be used to effectively prioritize the creation of unit tests in large software systems. The approach illustrates how SDP approaches, designed with a specific scenario in mind, can be tailored to provide guidance on how to make use of their results.

Chapter 6

Using SDP to Prioritize the Creation of Unit Tests

Test-Driven Development (TDD) is a software development practice that prescribes writing unit tests before writing implementation code. Recent studies have shown that TDD practices can significantly reduce the number of pre-release defects. However, most TDD research thus far has focused on new development. We investigate the adaptation of TDD-like practices for already-implemented code. We call such an adaptation "Test-Driven Maintenance" (TDM). In this chapter, we present a TDM approach, based on simple SDP models, that assists software development and testing managers to use the limited resources they have for testing large software systems efficiently. The approach leverages the development history of a project to generate a prioritized list of functions that managers should focus their unit test writing resources on. The list is updated dynamically as the development of the large software system progresses. We evaluate our approach on two large software systems: a large commercial system and the Eclipse Open Source Software system. For both systems, our findings suggest that factors based on the function size, modification frequency and defect fixing frequency should be used to prioritize the unit test writing efforts for software systems.

6.1 Introduction

Test-Driven Development (TDD) is a software development practice where developers write and run unit tests that would pass once the requirements are implemented. Then they implement the requirements and re-run the unit tests to make sure they pass [31,288]. The unit tests are generally written at the granularity of the smallest separable module, which is a function in most cases [244].

Previous empirical studies have shown that TDD can reduce pre-release defect densities by as much as 90%, compared to other similar projects that do not implement TDD [213]. In addition, other studies show that TDD helps produce better quality code [98,99], improve programmer productivity [79] and strengthen the developer confidence in their code [207].

Most of the previous research to date studied the use of TDD for new software development. However, prior studies show that more than 90% of the software development cost is spent on maintenance and evolution activities [82,196]. Other studies showed that an average Fortune 100 company maintains 35 million lines of code and that this amount of maintained code is expected to double every 7 years [206]. For this reason, we believe it is extremely beneficial to study the adaptation of TDD-like practices for the maintenance of already implemented code. In this Chapter we call this "Test-Driven Maintenance" (TDM).

Applying TDM to, for example, legacy systems is important because legacy systems are often instilled in the heart of newer, larger systems and continue to evolve with new code [33]. In addition, due to their old age, legacy systems lack proper documentation and become brittle and error-prone over time [45]. Therefore, TDM should be employed for these legacy systems to assure quality requirements are met and to reduce the chance of failures due to evolutionary changes.

However, legacy systems are typically large and writing unit tests for an entire software system at once is time consuming and practically infeasible. To mitigate this issue, TDM uses the same divide-and-conquer idea of TDD. However, instead of focusing on a few tasks from the requirements documents, developers that apply TDM isolate functions of the software system and individually unit test them. Unit tests for the functions are incrementally written until a desired quality target is met.

The idea of incrementally writing unit tests is very practical and has three main advantages. First, it gives resource-strapped managers some breathing room in terms of resource allocation (i.e., it alleviates the need for long-term resource commitments). Second, developers can get more familiar with the code (especially if it is legacy code) through the unit test writing efforts [114], which may ease future maintenance efforts. Third, unit tests can be easily maintained and updated in the future to assure the high quality of the software system [244].

The major challenge for TDM is determining how to *prioritize* the writing of unit tests to achieve the best return on investment. Do we write unit tests for functions in a random order? Do we write unit tests for the functions that we worked on most recently? Often, development and testing teams end up using ad hoc practices, based on gut feelings, to prioritize the unit test writing efforts. However, using the right prioritization strategy can save developers time, save the organization money and increase the overall product quality [68, 74].

This Chapter presents an approach that prioritizes the writing of unit tests for large software systems, based on different history-based factors. Our goal is to determine the most effective factors and investigated the effect of various simulation parameters on our study results. We evaluate our approach on a commercial system and an open source system. Evaluating our approach on the two different systems reduces the threat to external validity and improves the generalizability of our findings since both systems follow different development practices (i.e., commercial vs. open source), come from different domains (i.e., communication system vs. integrated development environment) and are written in different programming languages (i.e., C/C++ vs. Java).

Our results show that the proposed factors significantly improve the testing efforts, in terms of potential defect detection, when compared to random test writing. Factors that prioritize unit testing effort based on function size, modification frequency and defect fixing frequency are the best performing for both systems. Combining the factors improves the performance of some factors, but did not outperform our best performing factors.

6.1.1 Organization of Chapter

Section 6.2 provides a motivating example for our work. Section 6.3 details our approach. Section 6.4 describes the simulation-based case study. Section 6.5 presents the results of the case study. Section 6.6 details two techniques used to combine the factors and presents their results. Section 6.7 discusses the effects of the simulation parameters on our results. Section 6.8 presents the list of threats to validity and Section 6.9 discusses the related work. Section 6.10 concludes the chapter.

6.2 Motivating Example

In this section, we use an example to motivate our approach. Lindsay is a software development manager for a large software system that continues to evolve with new code. To assure a high level of quality for the software system, Lindsay's team employs TDM practices. Using TDM, the team isolates functions of the software system and writes unit tests for them. However, deciding which functions to write unit tests for is a challenging problem that Lindsay and his team must face.

Writing unit tests for all of the code base is nearly impossible. For example, if a team has enough resources to write unit tests to assess the quality of 100 lines of code per day, then writing unit tests for a 1 million lines of code (LOC) system would take over 27 years. At the same time, the majority of the team is busy with new development and maintenance efforts. Therefore, Lindsay has to use his resources effectively in order to obtain the best return on his resource investment.

A primitive approach that Lindsay tries is to randomly pick functions and write unit tests for them or write tests for functions that have been recently worked on. However, he quickly realizes that such an approach is not very effective. Some of the recently worked on functions are rarely used later, while others are so simple that writing unit tests for them is not a priority. Lindsay needs an approach that can more effectively assist him and his team prioritize the writing of unit tests for the software system.

To assist development and testing teams like Lindsay's, we present an approach that uses the history of the project to prioritize the writing of unit tests. The approach uses factors extracted from the project history to recommend a prioritized list of functions to write unit tests for. The size of the list can be customized based on the amount of available resources at any specific time. The approach updates its recommended list of functions as the project progresses.

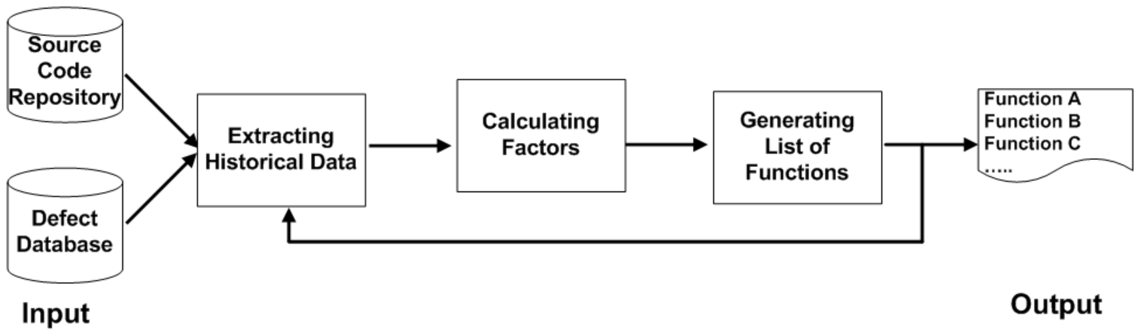


Figure 6.1: Overview of our approach

6.3 Approach

In this section, we detail our approach, which is outlined in Figure 6.1. In a nutshell, the approach extracts a project's historical data from its code and defect repositories, calculates various factors and recommends a prioritized list of functions to write unit tests for. Once the unit tests are written, we remove the recommended functions, re-extracts new data from the software repositories to consider new development activity and repeats the process of calculating factors and generating a list of functions. In the next four subsections, we describe each phase in more detail.

6.3.1 Extracting Historical Data

The first step of the approach is to extract historical data from the project's development history. In particular, we combine source code modification information from the source code control system (e.g., SVN [242] and CVS [267]) with defect data stored in the defect tracking system (e.g., Bugzilla [1]). Each modification record contains the time of the modification (day, month, year and local time), the author, the changed files, the version of the files, the changed line numbers and a modification record log that describes the change.

In order to determine whether a modification is a defect fix, we used a lexical technique to automatically classify modifications [117, 199, 311]. The technique searches the modification record logs, which are stored in the source code repository, for keywords, such as "bug" or "bugfix", and defect identifiers (used to search the defect database) to do the classification. If a modification record contains a defect identifier or one of the keywords associated with a defect, then it is classified as a defect fixing modification. In some cases, the modification record logs contain a defect identifier only. In this case, we automatically fetch the defect report's type and classify the modification accordingly. We check the defect report's type, because in certain cases defect reports are used to submit feature enhancements instead of reporting actual defects. Eventually, our technique groups modification records into two main categories: defect fixing modifications and general maintenance modifications.

The next step involves mapping the modification types to the actual source code functions that changed. To achieve this goal, we identify the files that changed and their file version numbers (this information is readily available in the historical modification log). Then, we extract the source code versions of the files that changed and their previous version, parse them to identify the individual functions, and compare the two consecutive versions of the files to identify which functions changed. Since we know which files and file versions were changed by a modification, we can pinpoint the functions modified by each modification. We annotate each of the changed functions with the modification type.

To better illustrate this step, we use the example shown in Figure 6.2. Initially, change 1 commits the first version of files X and Y. There are 3 defects (italicized) in the committed files, one in each of the functions `add`, `subtract` and `divide`. Change 2 fixes these defects. We would determine that change 2 is a defect fix from the change log message and comparing versions 1 and 2 of files X and Y would tell us that functions `add`, `divide` and `subtract` changed. Therefore, we would record 1 defect fix change to each of these functions. Thus far, function `multiply` did not change, therefore it will not have anything recorded against it. In change 3, comments are added to functions `subtract` and `multiply`. By the end of change 3, function `add` would have 1 defect fixing change, function `subtract` will have 1 defect fix change and 1 enhancement change, function `divide` will have 1 defect fix change, and function `multiply` will have 1 enhancement change.

Although the use of software repositories (i.e., source code control systems and defect tracking systems) is becoming increasingly popular in software projects, there still exist some issues with using data from such repositories. For example, developer may forget to mention the defect number that a change fixes. And even if they do include the defect numbers, in certain cases, the defect mentioned in the change description could refer to a defect that was created after the change itself or the defect mentioned is missing from the defect database altogether [310]. These issues may introduce bias in the data [41], however, recent work showed that the effect of such bias does not significantly affect the outcome of our findings [217].

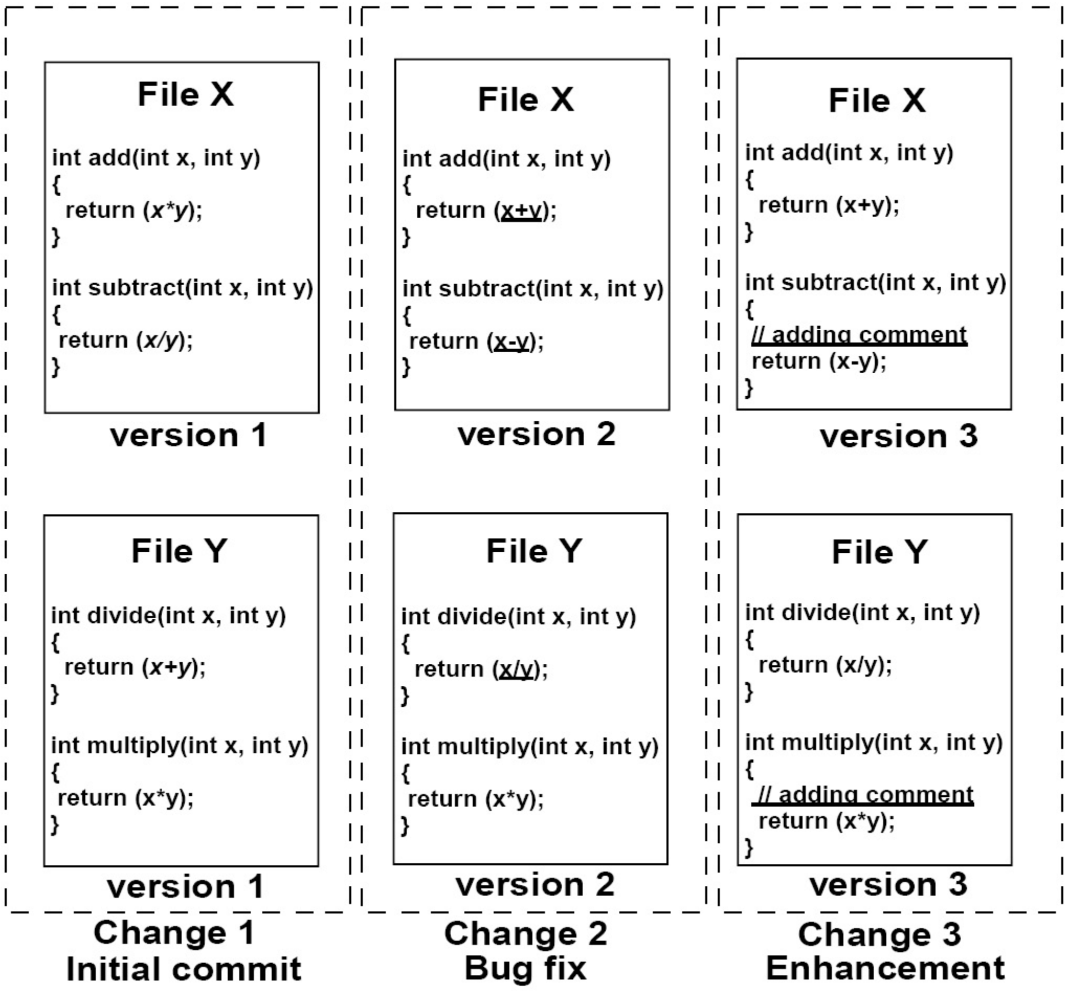


Figure 6.2: Linking changes to functions

6.3.2 Calculating Factors

We use the extracted historical data to calculate various factors. The factors are used to generate the prioritized list of functions for testing. We choose to use factors that can be extracted from a project's history for two main reasons: 1) large software systems, especially legacy systems, ought to have a very rich history that we can use to our advantage and 2) previous work in fault prediction showed that history based factors are good indicators of future defects (e.g., [19, 211]). We conjecture that factors used for fault prediction will perform well, since ideally we want to write unit tests for the functions that have defects in them.

The factors fall under four main categories: modification based factors, defect fix based factors, size based factors and risk based factors. The factors are listed in Table 6.1. We also include a random factor that we use as a baseline factor to compare the aforementioned factors to. For each factor, we provide a description, our intuition for using the factor and related work that influenced our decision to study the factors.

The factors listed in Table 6.1 are a small sample of the factors that can be used to generate the list of functions. We chose to use these factors since previous work on fault prediction has proven their ability to perform well. However, any factor that captures the characteristics of the software system and can be linked to functions may be used to generate the list of functions.

6.3.3 Generating a List of Functions

Following the factor calculation phase, we use the factors to generate a prioritized list of functions that are recommended to have unit tests written for. Each factor generates a different prioritized list of functions. For example, one of the factors we use (i.e., MFM) recommends that we write tests for functions that have been modified the most since the beginning of the project. Another factor recommends that we write tests for functions that are fixed the most (i.e., MFF).

Then, we loop back to the historical data extraction phase, to include any new development activity and run through the factor calculation and list generation phases. Each time, a new list of functions is generated for which unit tests should be written.

6.3.4 Removing Recommended Functions

Once a function is recommended to have a test written for it, we remove it from the pool of functions that we use to generate future lists. In other words, we assume that once a function has had a unit test written for it, it will not need to have any additional new test written for it in the future; at most the test may need to be updated. We make this assumption for the following reason: once the function is recommended and the initial unit test is written, then this initial unit test will make sure all of the function's current code is tested. Also, since the team adopts TDM practices any future additions/changes to the function will be accompanied with changes to the associated unit tests.

Table 6.1: List of all factors used for prioritizing the unit test writing efforts for large software systems

Category	Factor	Order	Description	Intuition	Related Work
Modifications	Most Frequently Modified (MFM)	Highest to lowest	Functions that were modified the most since the start of the project.	Functions that are modified frequently tend to decay over time, leading to more defects.	The number of prior modifications to a file is a good predictor of its future defects [19, 105, 143, 171].
	Most Recently Modified (MRM)	Latest to oldest	Functions that were most recently modified.	Functions that were modified most recently are the ones most likely to have a defect in them (due to the recent changes).	More recent changes contribute more defects than older changes [105].
Defect Fixes	Most Frequently Fixed (MFF)	Highest to lowest	Functions that were fixed the most since the start of the project.	Functions that are frequently fixed in the past are likely to be fixed in the future.	Prior defects are a good indicator of future defects [295].
	Most Recently Fixed (MRF)	Latest to oldest	Functions that were most recently fixed.	Functions that were fixed most recently are more likely to have a defect in them in the future.	The recently fixed factor has been used to prioritize defective subsystems [117], files and functions [155].
Size	Largest Modified (LM)	Largest to smallest	The largest modified functions, in terms of total lines of code (i.e, source, comment and blank lines of code).	Large functions are more likely to have defects than smaller functions.	The simple lines of code factor correlates well with most complexity factors (e.g., McCabe complexity) [105, 122, 171, 225].
	Largest Fixed (LF)	Largest to smallest	The largest fixed functions, in terms of total lines of code (i.e, source, comment and blank lines of code).	Large functions that need to be fixed are more likely to have more defects than smaller functions that are fixed less.	The simple lines of code factor correlates well with most complexity factors (e.g., McCabe complexity) [105, 122, 171, 225].
Risk	Size Risk (SR)	Highest to lowest	Riskiest functions, defined as the number of defect fixing changes divided by the size of the function in lines of code.	Since larger functions may naturally need to be fixed more than smaller functions, we normalize the number of defect fixing changes by the size of the function. This factor will mostly point out relatively small functions that are fixed frequently (i.e., have high defect density).	Using relative churn factors performs better than using absolute values when predicting defect density [209].
	Change Risk (CR)	Highest to lowest	Riskiest functions, defined as the number of defect fixing changes divided by the total number of changes.	The number of defect fixing changes normalized by the total number of changes. For example, a function that changes 10 times in total and out of those 10 times 9 of them were to fix a defect should have a higher priority to be tested than a function that changes 10 times where only 1 of those ten is a defect fixing change.	Using relative churn factors performs better than using absolute values when predicting defect density [209].
Random	Random	Random	Randomly selects functions to write unit tests for.	Randomly selecting functions to test can be thought of as a base line scenario. Therefore, we use the random factor's performance as a base line to compare the performance of the other factors to.	Previous studies on test prioritization use a random factor to compare their performance [68, 73, 74].

6.4 Simulation-Based Case Study

To evaluate the performance of our approach, we conduct a simulation-based case study on two large software systems: a commercial system and the Eclipse system. The commercial software system is a legacy system, written in C and C++, which contains tens of thousands of functions totalling hundreds of thousands of lines of code. We used 5.5 years of the system's history to conduct our simulation, in which over 50,000 modifications were analyzed. We cannot disclose any more details about the studied system for confidentiality reasons.

On the other hand, the Eclipse system is an Integrated Development Environment (IDE), written in Java. We analyzed a total of 48,718 files, which contained 314,910 functions over their history. Again, we used 5.5 years of the system's history to conduct our simulation, in which 81,208 modifications were analyzed, of which 12,332 were defective changes.

In this section, we detail the steps of our case study and introduce our evaluation factors.

6.4.1 Simulation study

The simulation ran in iterations. For each iteration we: 1) extract the historical data, 2) calculate the factors, 3) generate a prioritized list of functions, 4) measure the time it takes to write tests for the recommended list of functions and 5) remove the list of functions that were recommended. Then, we advance the time (i.e., we account for the time it took to write the unit tests) and do all of the aforementioned steps over again.

Step 1. We used 5.5 years of historical data from the commercial and OSS systems to conduct our simulation. The first 6 months of the project were used to calculate the initial set of factors and the remaining 5 years are used to run the simulation.

Step 2. To calculate the factors, we initially look at the first 6 months of the project. If, for example, we are calculating the MFM factor, we would look at all functions in the first 6 months of the project and rank them in descending order based on the number of times they were modified during that 6 month period. The amount of history that we consider to calculate the factors increases as we advance in the simulation. For example, an iteration 2 years into the simulation will use 2.5 years (i.e., the initial 6 months and 2 years of simulation time) of history when it calculates the factors.

Step 3. Next, we recommend a prioritized list of 10 functions that should have unit tests written for them. One list is generated for each of the factors. The list size of 10 functions is an arbitrary choice we made. If two functions have the same score, we randomly choose between them. We study the effect of varying the list size on our results in detail in Section 6.7. Furthermore, in our simulation, we assume that once a list of 10 functions is generated, tests will be written for all 10 functions before a new list is generated.

Step 4. Then, we estimate the time it takes to write tests for these 10 functions. To do so, we use the size of the recommended functions and divide by the available resources. The size of the functions is used as a measure for the amount of effort required to write unit tests for those 10 functions [21, 183]. Since complexity is highly correlated with size [105], larger functions will take more effort/time to write unit tests for.

The number of available resources is a simulation parameter, expressed as the number of lines of code that testers can write unit tests for in one day. For example, if one tester is available to write unit tests for the software system and that tester can write unit tests for 50 lines of code per day, then a list of functions that is 500 lines will take him 10 days. In our simulation, we set the total test writing capacity of the available resources to 100 lines per day. We study the effect of varying the resources available to write unit tests in more detail in Section 6.7.

After we calculate the time it takes to write unit tests for the functions that we recommend, we update the factors with the new historical information. We do this to account for the continuous change these software systems undergo. For example, if we initially use six months to calculate the factors and the first list of recommended functions takes one month to write unit tests for, then we update the factors for the next iteration to use seven months of historical information. We use the updated factors to calculate the next set of functions to write tests for and so on.

Step 5. Once a function is recommended to have a test written for it, we remove it from the pool of functions that we use to generate future lists. In other words, we assume that once a function has had a unit test written for it, it will not need to have a new test written for it from scratch in the future; at most the test may need to be updated.

Once the parameters are set, the simulation is entirely automated. Any of the parameters can be modified at any time, however, there is no need for any manual work after the initial setup is done.

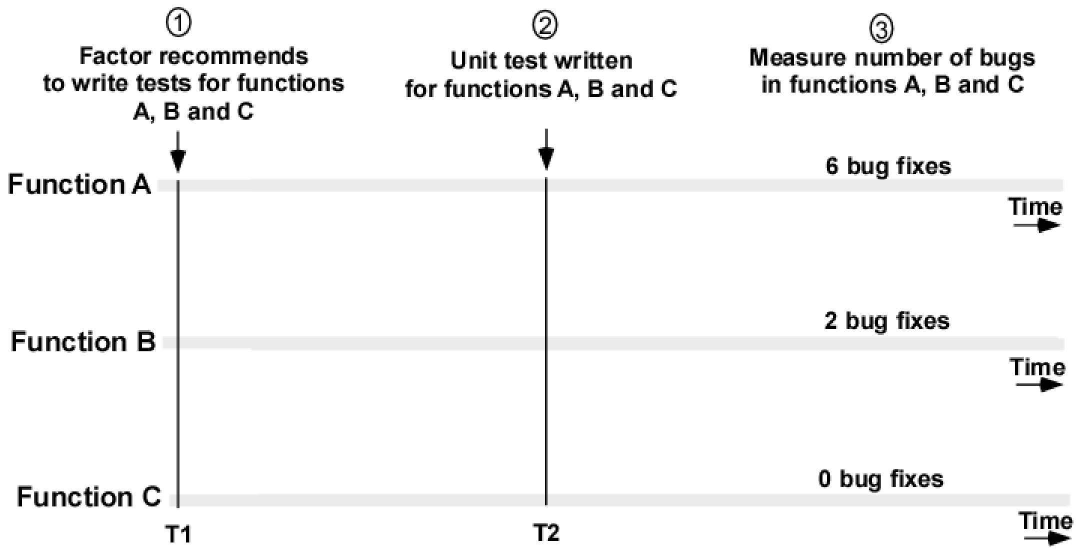
We repeat the 5-step process mentioned above for a period of 5 years. To evaluate the performance of the different factors, we periodically (every 3 months) measure the performance using two factors: Usefulness and Percentage of Optimal Performance (POP), which we describe next.

6.4.2 Performance Evaluation Metrics

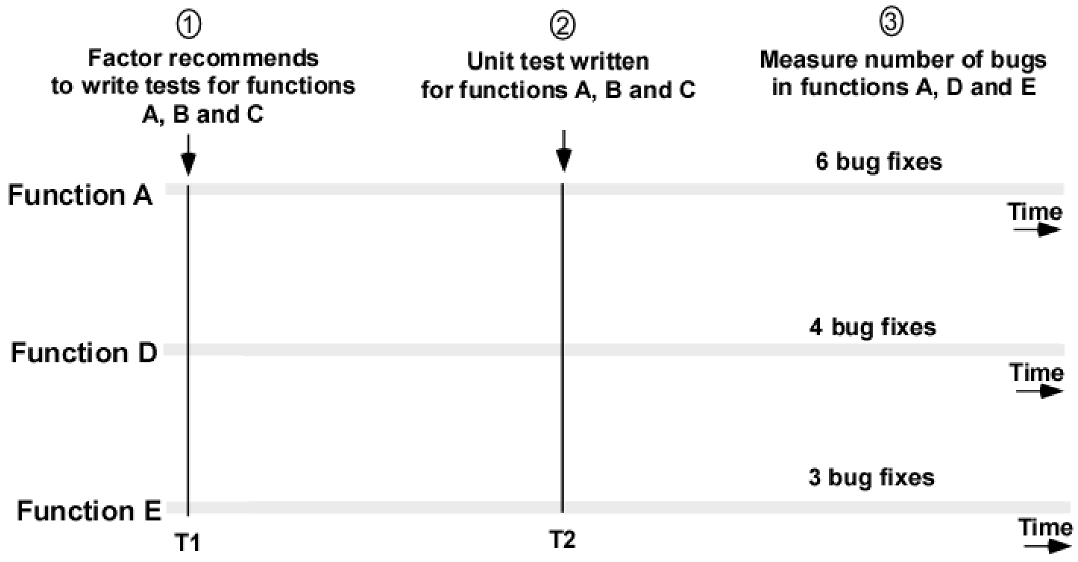
Usefulness: The first question that comes up after we write unit tests for a set of functions is - was writing the tests for these functions worth the effort? For example, if we write unit tests for functions that rarely change and/or have no defects after the tests are written, then our effort may be wasted. Ideally, we would like to write unit tests for the functions that end up having defects in them.

We define the usefulness metric as *the percentage of functions for which we write unit tests that catch at least one defect after the tests are written*. The usefulness metric indicates how much of our effort on writing unit tests is actually worth the effort. This metric is similar to the hit rate metric used by earlier dynamic defect prediction studies [117, 155].

We use the example in Figure 6.3(a) to illustrate how we calculate the usefulness. Functions A and B have more than 1 defect fix after the unit tests were written for them (after point 2 in Figure 6.3(a)). Function C did not have any defect fix after we wrote the unit test for it. Therefore, for this list of three functions, we calculate the usefulness as $\frac{2}{3} = 0.666$ or 66.6%.


①
Factor recommends to write tests for functions A, B and C
②
Unit test written for functions A, B and C
③
Measure number of bugs in functions A, B and C
Function A
6 bug fixes
Time
Function B
2 bug fixes
Time
Function C
0 bug fixes
Time
T1
T2

(a) Usefulness evaluation example


①
Factor recommends to write tests for functions A, B and C
②
Unit test written for functions A, B and C
③
Measure number of bugs in functions A, D and E
Function A
6 bug fixes
Time
Function D
4 bug fixes
Time
Function E
3 bug fixes
Time
T1
T2

(b) POP example

Figure 6.3: Performance evaluation example

Percentage of Optimal Performance (POP): In addition to calculating the usefulness, we would like to measure how well a factor performs compared to the optimal (i.e., the best we can ever do). Optimally, one would have perfect knowledge of the future and write unit tests for functions that are the most defective. This would yield the best return on investment.

To measure how close we are to the optimal performance, we define a metric called Percentage of Optimal Performance (POP). To calculate the POP, we generate two lists: one is the list of functions generated by a factor and the second is the optimal list of functions. The optimal list contains the functions with the most defects from the time the unit tests were written till the end of the simulation. Assuming that the list size is 10 functions, we calculate the POP as the number of defects in the top 10 functions (generated by the factors), divided by the number of defects contained in the top 10 optimal functions. Simply put, the POP is *the percentage of defects we can avoid using a factor compared to the best we can do if we had perfect knowledge of the future*.

We illustrate the POP calculation using the example shown in Figure 6.3(b). At first, we generate a list of functions that we write unit tests for using a specific factor (e.g., MFM or MFF). Then, based on the size of these functions, we calculate the amount of time it takes to write unit tests for these functions (point 2 in Figure 6.3(b)). From that point on, we calculate the number of defects for all of the functions and rank them in descending order. For the sake of this example, let us assume we are considering the top 3 functions. Assuming our factor identifies functions A, B and C as the functions for which we need to write unit tests, however, these functions may not be the ones with the most defects. Assuming that the functions with the most defects are functions A, D and E (i.e., they are the top 3 on the optimal list). From Figure 6.3(a) , we can see that functions A, B and C had 8 defect fixes in total after the unit tests were written for them. At the same time, Figure 6.3(b) shows that the optimal functions (i.e., functions A, D and E) had 13 defect fixes in them. Therefore, the best we could have done is to remove 13 defects. We were able to remove 8 defects using our factor, hence our POP is $\frac{8}{13} = 0.62$ or 62%.

It is important to note that the key difference between the usefulness and the POP values is that usefulness is the percentage of *functions* that we found useful to write unit tests for. On the other hand, POP measures the percentage of *defects* that we could have avoided using a specific factor.

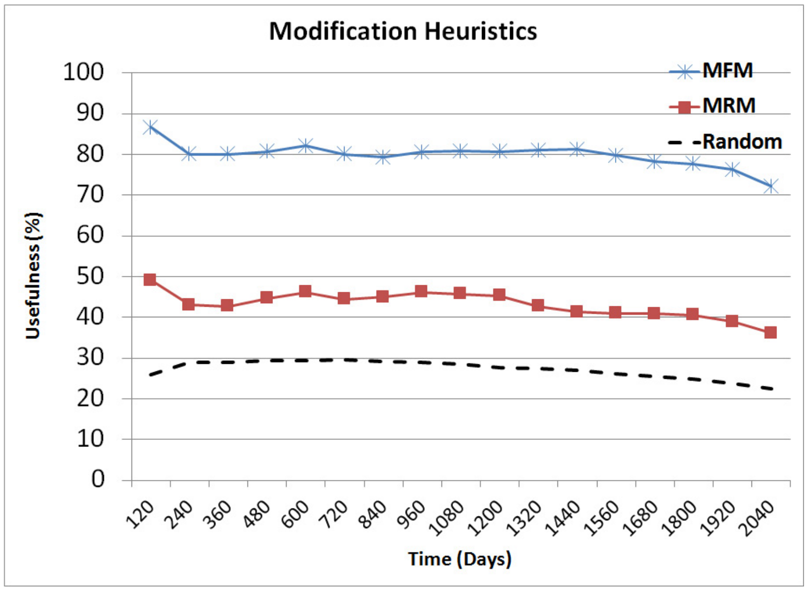


Figure 6.4: Usefulness of modification factors compared to the random factor for the commercial system

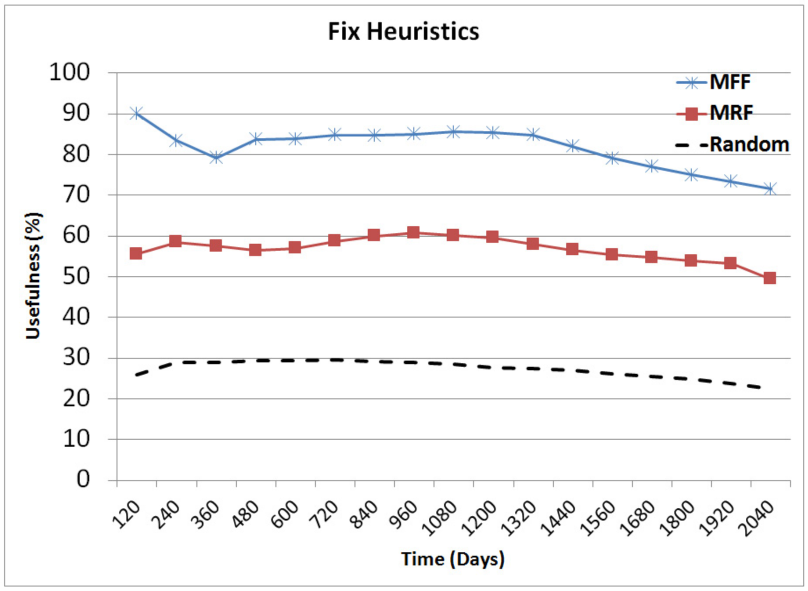


Figure 6.5: Usefulness of fix factors compared to the random factor for the commercial system

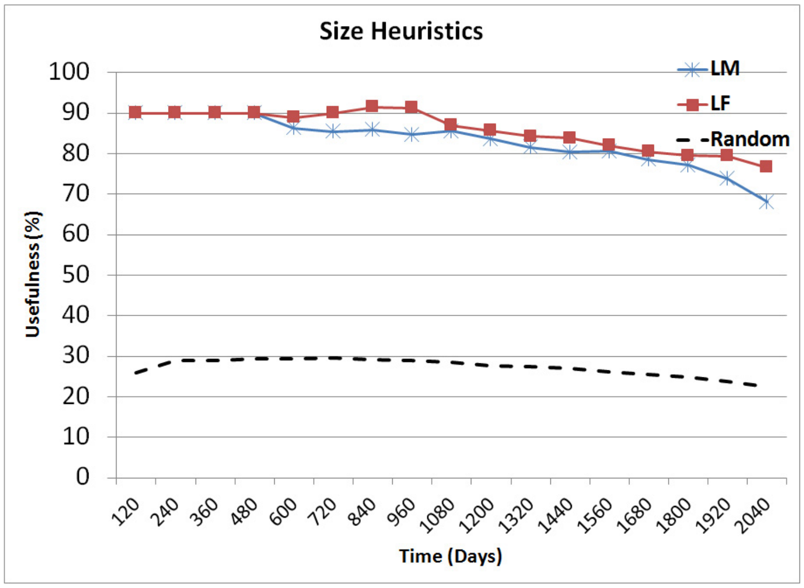


Figure 6.6: Usefulness of size factors compared to the random factor for the commercial system

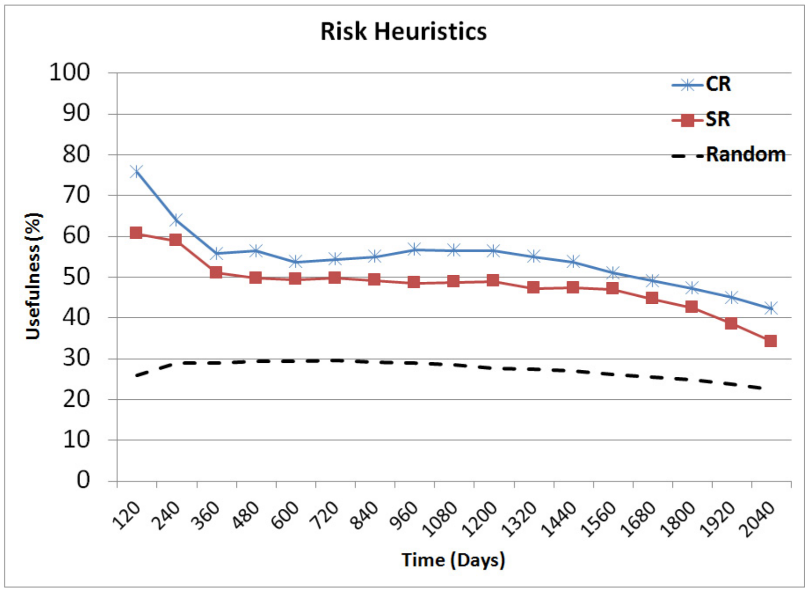


Figure 6.7: Usefulness of risk factors compared to the random factor for the commercial system

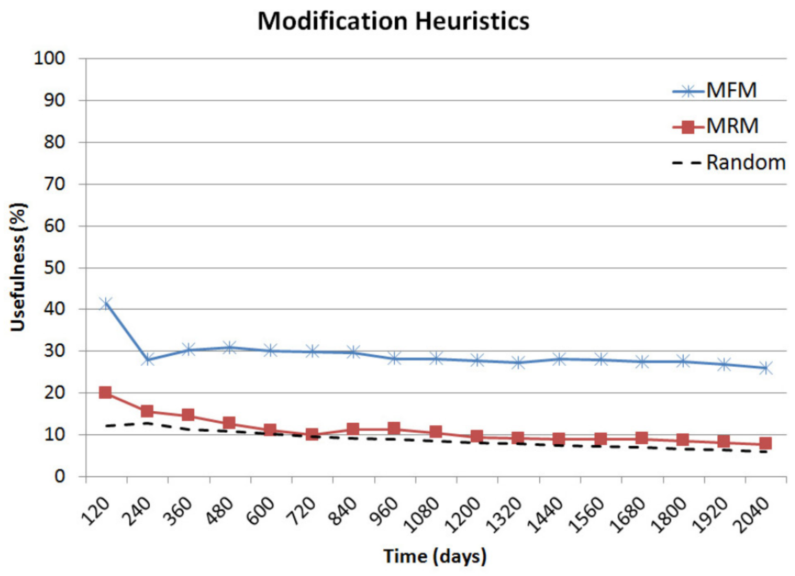


Figure 6.8: Usefulness of modification factors compared to the random factor for the Eclipse system

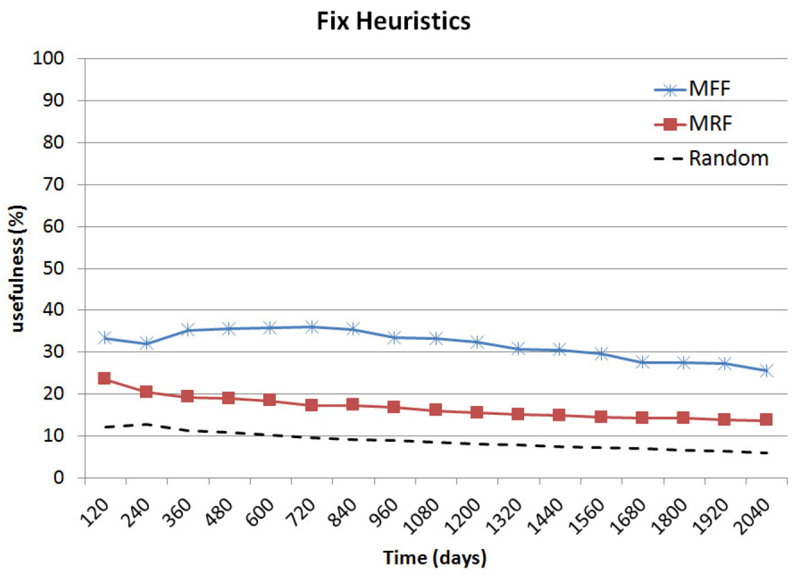


Figure 6.9: Usefulness of fix factors compared to the random factor for the Eclipse system

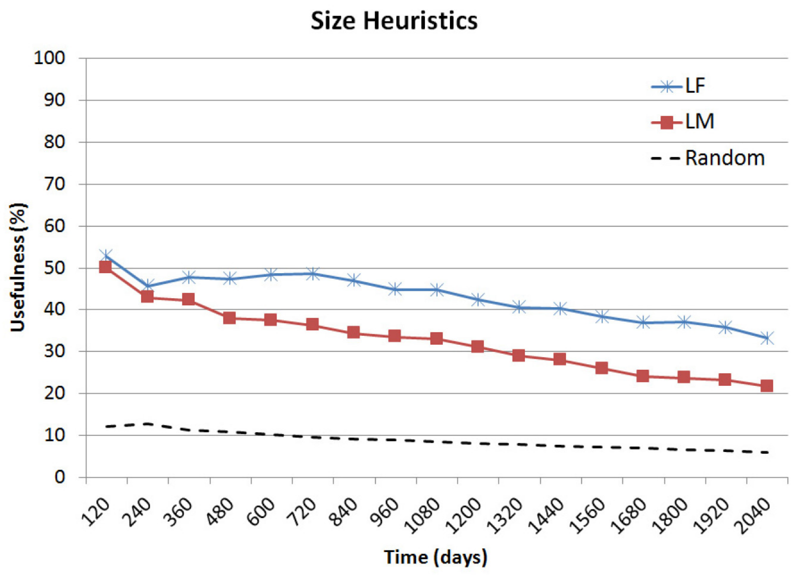


Figure 6.10: Usefulness of size factors compared to the random factor for the Eclipse system

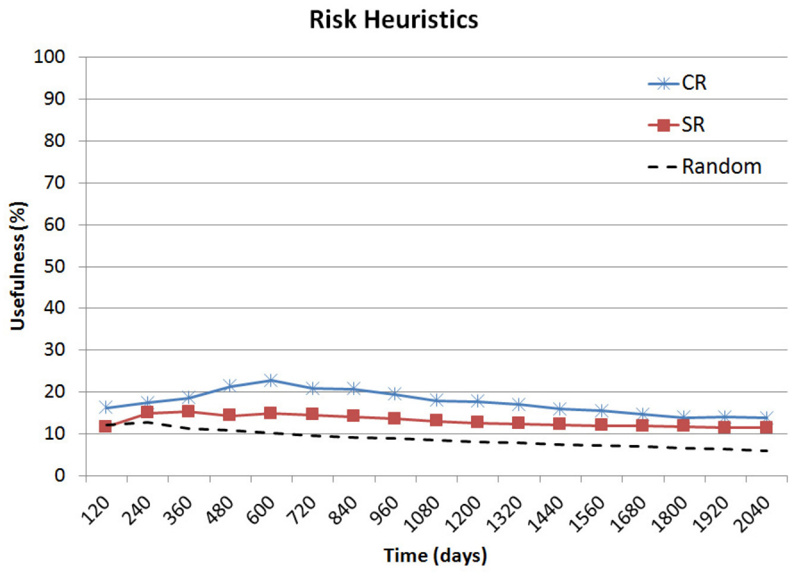


Figure 6.11: Usefulness of risk factors compared to the random factor for the Eclipse system

6.5 Case Study Results

In this section, we present the results of the usefulness and POP metrics for the proposed factors. Ideally, we would like to have high usefulness and POP values. To evaluate the performance of each of the factors, we use the random factor as our baseline [68, 73, 74]. If we cannot do better than just randomly picking functions to add to the list, then the factor is not that effective. Since the random factor can give a different ordering each time, we use the average of 5 runs, each of which uses different randomly generated seeds.

6.5.1 Usefulness

We calculate the usefulness for the factors listed in Table 6.1 and plot it over time for the commercial system (Figures 6.4, 6.5, 6.6, and 6.7) and the Eclipse system (Figures 6.8, 6.9, 6.10, and 6.11). The dashed black line in each of the figures depicts the results of the random factor. From the figures, we can observe that in the majority of the cases, the proposed factors outperform the random factor. However, our top performing factors (e.g., LF and LM) substantially outperform the random factor, in both projects.

The median usefulness values for each of the factors in the commercial system and the Eclipse system are listed in Tables 6.2 and 6.3, respectively. Since the usefulness values change over the course of the simulation, we chose to present the median values to avoid any sharp fluctuations. The last row of the table shows the usefulness achieved by the random factor. The factors are ranked from 1 to 9, with 1 indicating the best performing factor and 9 the worst.

Commercial system: Table 6.2 shows that the LF, LM, MFF and MFM are the top performing factors, having median values in the range of 80% to 87%. The third column in Table 6.2 shows that these factors perform approximately 3 times better than the random factor.

A strategy that developers may be inclined to apply is to write tests for functions that they worked most recently on. The performance of such a strategy is represented by the recency factors (i.e., MRM and MRF). We can observe from Figures 6.4 and 6.5 that the recency factors (i.e., MRM and MRF) perform poorly compared to their frequency counterparts (i.e., MFM and MFF) and the size factors.

Eclipse system: Similar to the commercial system, Table 6.3 shows that the LF, LM, MFF and MFM factors are the top performing factors. These factors achieve median values in the range of 28% to 44%. Although these usefulness values are lower than those achieved in the commercial system, they perform 3 to 5 times better than the random factor. The 3 to 5 times improvement is consistent with the results achieved in the commercial system.

Again, we can see that the frequency based factors (i.e., MFM and MFF) substantially outperform their recency counterparts (i.e., MRM and MRF). Examining the median values in Tables 6.2 and 6.3, we can see that for MFM we were able to achieve approximately 30%

median usefulness for the Eclipse system (80% for the commercial system). This means that approximately 3 (8 for the commercial system) out of the 10 functions we wrote unit tests for had one or more defects in the future. Therefore, writing the unit tests for these functions was useful. On the contrary, for the random factor, approximately 1 (3 for the commercial system) out of every 10 functions we wrote unit tests for had 1 or more defects after the unit tests were written.

Table 6.2: Usefulness results of the commercial system

Factor	Median Usefulness	Improvement over random	Rank
LF	87.0%	3.1 X	1
LM	84.7%	3.1 X	2
MFF	83.8%	3.0 X	3
MFM	80.0%	2.9 X	4
MRF	56.9%	2.1 X	5
CR	55.0%	2.0 X	6
SR	48.8%	1.8 X	7
MRM	43.1%	1.6 X	8
Random	27.7%	-	9

Table 6.3: Usefulness results of the Eclipse system

Factor	Median Usefulness	Improvement over random	Rank
LF	44.7%	5.3 X	1
LM	32.9%	3.9 X	2
MFF	32.3%	3.8 X	3
MFM	28.1%	3.3 X	4
CR	17.4%	2.0 X	5
MRF	16.0%	1.9 X	6
SR	12.6%	1.5 X	7
MRM	9.9%	1.2 X	8
Random	8.5%	-	9

Size, modification frequency and fix frequency factors should be used to prioritize the writing of unit tests for software systems. These factors achieve median usefulness values between 80–87% for the commercial system and between 28–44% for the Eclipse system.

6.5.2 Percentage of Optimal Performance (POP)

In addition to calculating the usefulness of the proposed factors, we would like to know how close we are to the optimal list of functions that we should write unit tests for if we have perfect knowledge of the future. We present the POP values for each of the factors in Figures 6.12, 6.13, 6.14, 6.15 for the commercial project and in Figures 6.16, 6.17, 6.18, 6.19 for Eclipse. The performance of the random factor is depicted using the dashed black line. The figures show that in all cases, and for both the commercial and the Eclipse system, the proposed factors outperform the random factor.

Commercial system: The median POP values are shown in Table 6.4. The POP values for the factors are lower than the usefulness values. The reason is that usefulness gives the percentage of functions that have one or more defects. However, POP measures the percentage of defects the factors can potentially avoid in comparison to the best we can do, if we have perfect knowledge of the future.

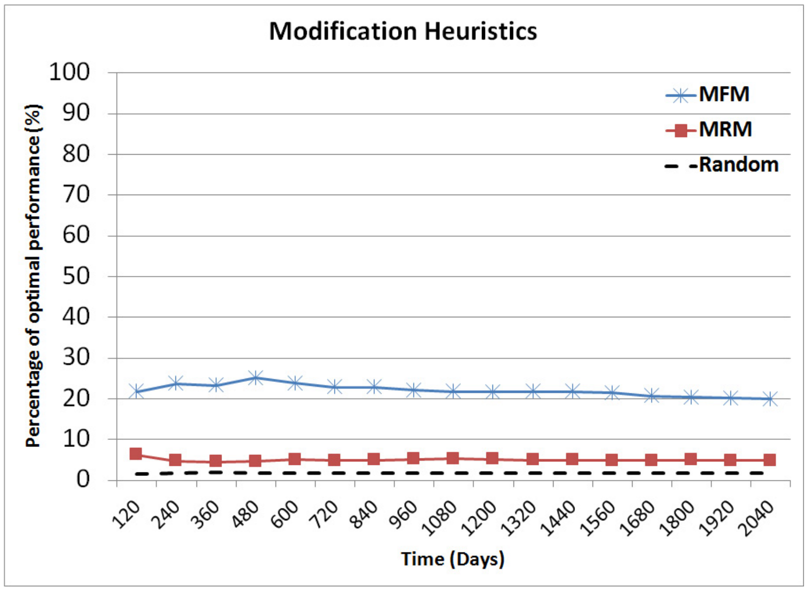


Figure 6.12: POP of modification factors compared to the random factor for the commercial system

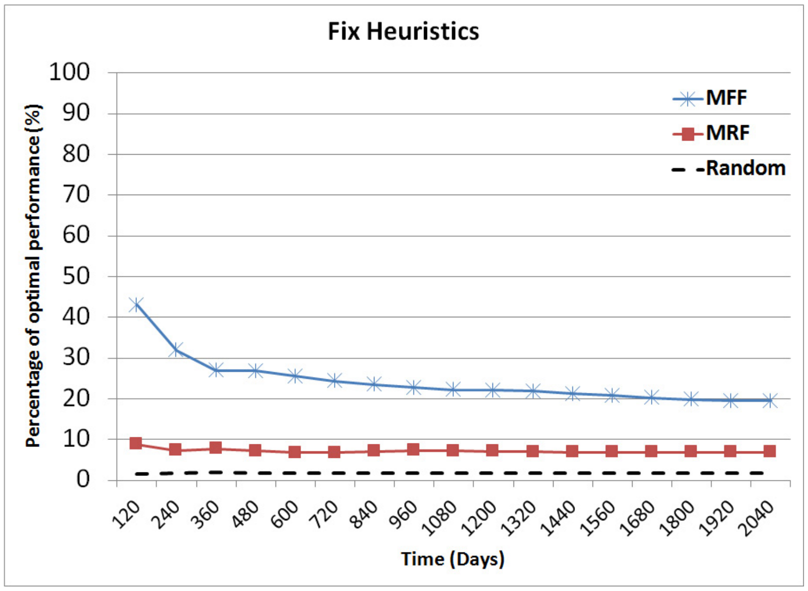


Figure 6.13: POP of fix factors compared to the random factor for the commercial system

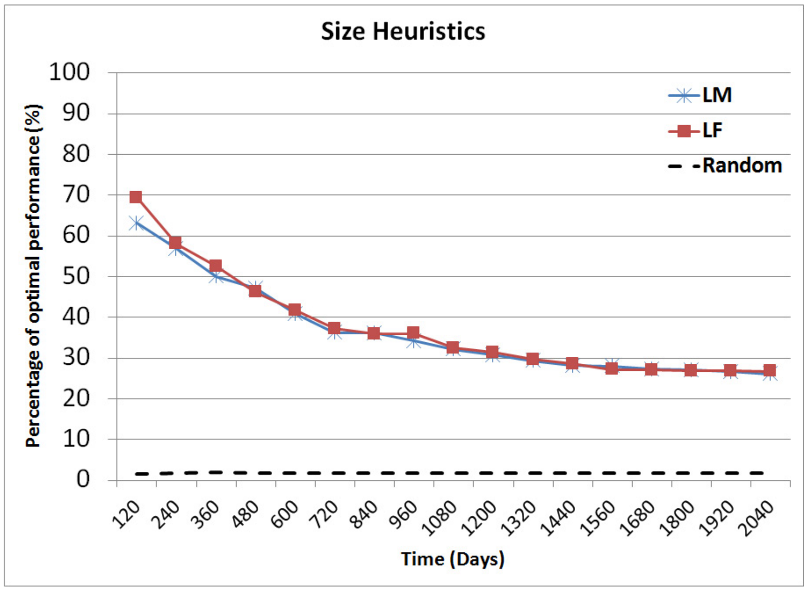


Figure 6.14: POP of size factors compared to the random factor for the commercial system

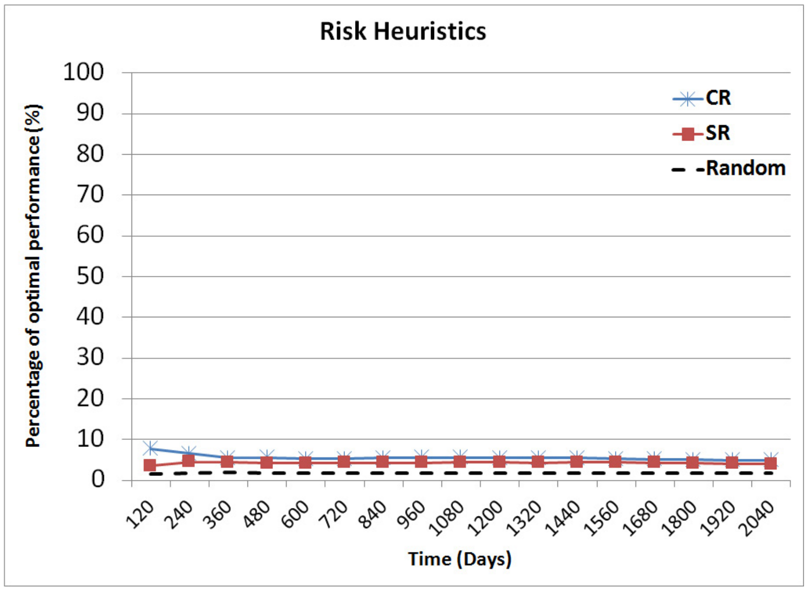


Figure 6.15: POP of risk factors compared to the random factor for the commercial system

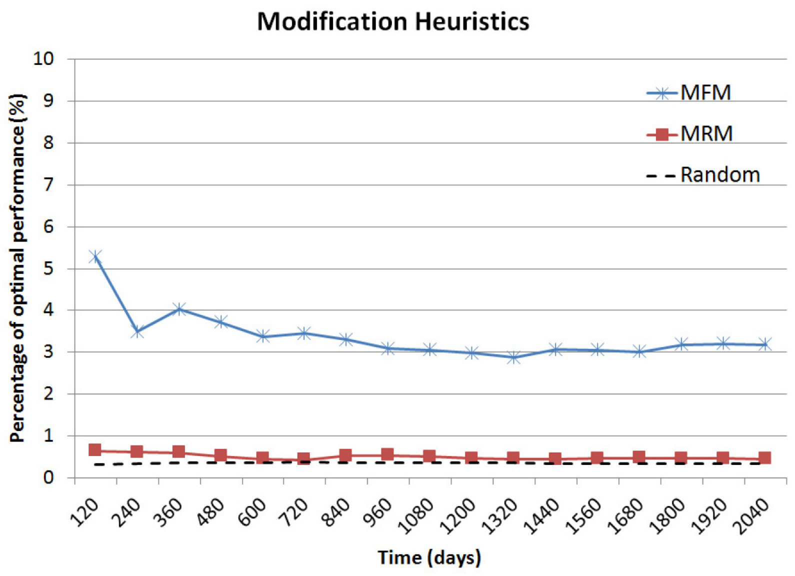


Figure 6.16: POP of modification factors compared to the random factor for the Eclipse system

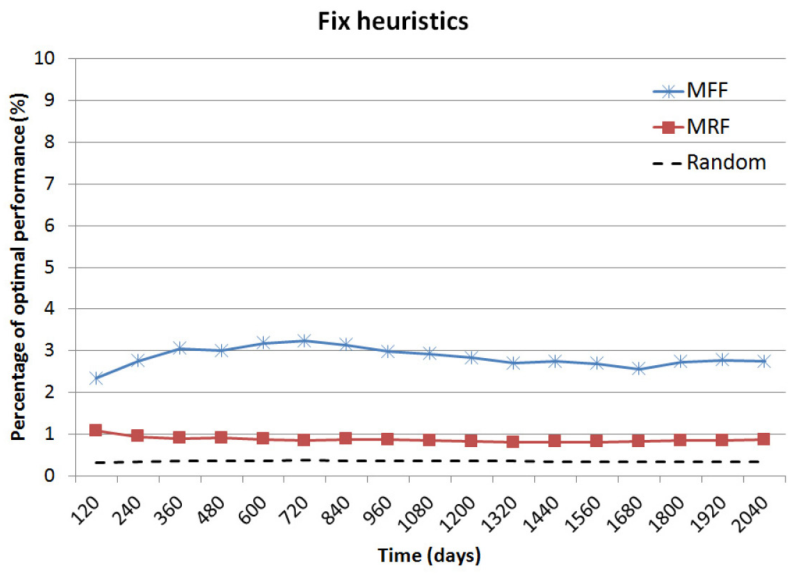


Figure 6.17: POP of fix factors compared to the random factor for the Eclipse system

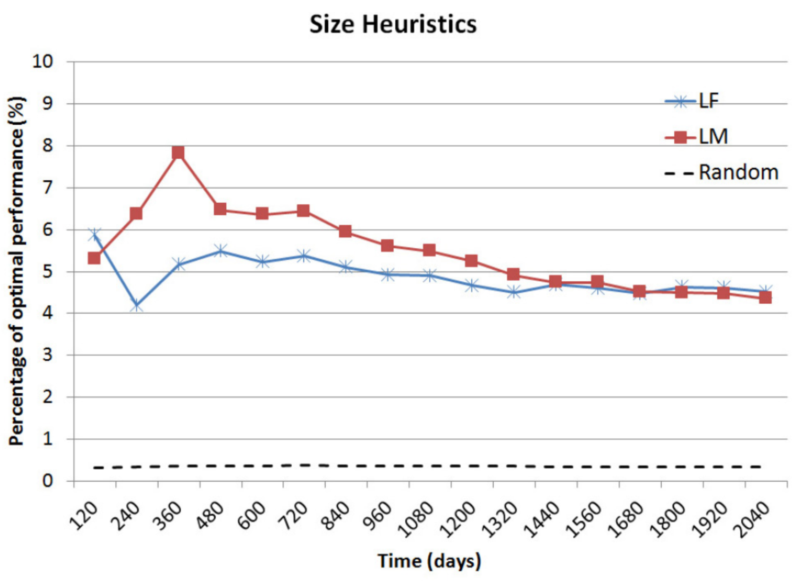


Figure 6.18: POP of size factors compared to the random factor for the Eclipse system

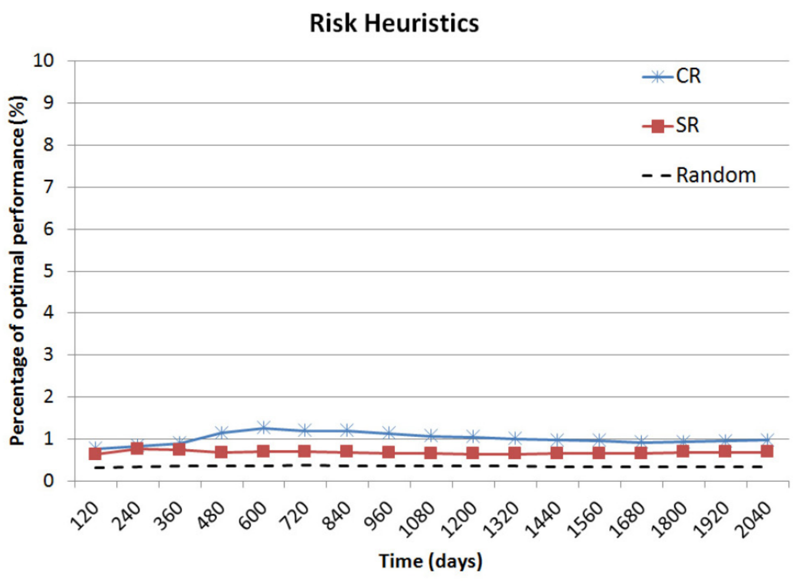


Figure 6.19: POP of risk factors compared to the random factor for the Eclipse system

Although the absolute POP percentages are lower compared to the usefulness measure, the ranking of the factors remains quite stable (except for the SR and MRM, which exchanged 7th and 8th spot). Once again, the best performing factors are LF, LM, MFF and MFM. The median values for these top performing factors are in the 20% to 32.4% range. These values are 12 to 19 times better than the 1.7% that can be achieved using the random factor.

Eclipse system: The median POP values are shown in Table 6.5. The LM, LF, MFM and MFF are also the top performing factors for the Eclipse system. Their median values are between 2.76% to 5.31%. Although these values are low, they are much higher than the value of the random factor.

Table 6.4: Percentage of Optimal Performance results of the commercial system

Factor	Median POP	Improvement over random	Rank
LF	32.4%	19.1 X	1
LM	32.2%	18.9 X	2
MFF	22.2%	13.1 X	3
MFM	21.8%	12.8 X	4
MRF	7.0%	4.1 X	5
CR	5.5%	3.2 X	6
MRM	4.9%	2.9 X	7
SR	4.3%	2.5 X	8
Random	1.7%	-	9

Table 6.5: Percentage of Optimal Performance results in the Eclipse system

Factor	Median POP	Improvement over random	Rank
LM	5.31%	15.2 X	1
LF	4.68%	13.4 X	2
MFM	3.18%	9.1 X	3
MFF	2.76%	7.9 X	4
CR	0.97%	2.8 X	5
MRF	0.85%	2.4 X	6
SR	0.66%	1.9 X	7
MRM	0.46%	1.3 X	8
Random	0.35%	-	9

The median Eclipse POP values are considerably lower than the median POP values of the commercial system. A preliminary examination into this issue shows that the distribution of changes and defects in the Eclipse system is quite sparse. For example, in a certain year a set of files changed and then those files do not change for a while. This sort of behavior affects our simulation results, since we use the history of the functions to prioritize. One possible solution is to restrict how far into the future the simulator looks at when calculating the POP values, however, since our goal is to compare the factors, our current simulation suffices. In the future, we plan to investigate different strategies to improve the performance of the proposed factors.

Regardless, our top performing factors performed approximately 8 to 15 times better than the random factor. The 8 to 15 times improvement is consistent with the finding in the commercial system.

Finally, we can observe a decline in the usefulness and POP values at the beginning of the simulation, shown in Figures 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 6.10, 6.11, 6.12, 6.13, 6.14, 6.15, 6.16, 6.17, 6.18, and 6.19 . This decline can be attributed to the fact that initially, there are many defective functions for the factors to choose from. Then, after these defective functions have been recommended, we remove them from the pool of functions that we can recommend. Therefore, the factors begin to recommend some functions that are not or less defective. Previous studies by Ostrand *et al.* [225] showed that the majority of the defects (approximately 80%) are contained in a small percentage of the code files (approximately 20%). These studies support our findings.

Size, modification frequency and fix frequency factors should be used to prioritize the writing of unit tests for software systems. These factors achieve median POP values between 21.8–32.4% for the commercial system and 2.76–5.31% for the Eclipse system.

6.6 Combining Factors

Thus far, we have investigated the effectiveness of prioritizing the unit test creation using each factor individually. Previous work by Kim *et al.* showed that combining factors yields favorable results [155]. Therefore, we investigate whether or not combining the factors can further enhance the performance.

To achieve this goal, we use two combining strategies which we call COMBO and WEIGHTED COMBO. With COMBO, we generate a list of functions for each factor. Then, we take the top ranked function from each factor and write a test for them. The WEIGHTED COMBO factor on the other hand gives higher weight to functions put forward by higher ranked factors.

Table 6.6: Combining factors example

	Factor A (HA) Rank 1	Factor B (HB) Rank 2	Factor C (HC) Rank 3
1.	f1A	f1B	f1C
2.	f2A	f2B	f2C
3.	f3A	f3B	f3C

To illustrate, consider the example in Table 6.6. In this example, we have 3 factors: HA, HB and HC ranked 1, 2 and 3, respectively. Each of the factors lists 3 functions: f1, f2 and f3 ranked 1, 2 and 3, respectively.

In this case, the COMBO factors would recommend f1A from HA, f1B from HB and f1C from HC. The WEIGHTED COMBO factor uses the weights assigned to each function to calculate the list of recommended functions. The weight for each function is based on the rank of the factor relative to other factors and the rank of that function relative to other functions within its factor. Mathematically, the weight is defined as:

$$Function\ Weight = \frac{1}{Factor\ rank} * \frac{1}{Function\ rank} \tag{6.1}$$

For example, in Table 6.6 HA has rank 1. Therefore, f1A would have weight 1 (i.e. $\frac{1}{1} * \frac{1}{1}$), f2A would have weight 0.5 (i.e., $\frac{1}{1} * \frac{1}{2}$) and f3A would have weight 0.33 (i.e., $\frac{1}{1} * \frac{1}{3}$). Factor HB on the other hand has rank 2. Therefore, the weight for f1B is 0.5 (i.e., $\frac{1}{2} * \frac{1}{1}$), for f2B is 0.25 (i.e., $\frac{1}{2} * \frac{1}{2}$) and for f3B is 0.17 (i.e. $\frac{1}{2} * \frac{1}{3}$). Following the same method, the weight for f1C is 0.33, for f2C is 0.17 and for f3C is 0.11. In this case, if we were to recommend the top 5 functions, then we would recommend f1A, f2A, f1B, f3A, and f1C. In the case of a tie in the function weights (e.g., f2A and f1B), we choose the function with the higher ranked factor first (i.e., f2A).

We obtain the factor rankings from Tables 6.2 , 6.3 , 6.4, 6.5 and run the simulation using the combined factors. The Usefulness and the POP metrics were used to evaluate the performance of the COMBO and WEIGHTED COMBO metrics. Tables 6.7 and 6.8 present the Usefulness and POP results, respectively. The best performing factors (i.e., LF and LM) outperform the COMBO and WEIGHTED COMBO factors in all cases. However, the COMBO and WEIGHTED COMBO factors provide a significant improvement over the worst performing factor (i.e., MRM and SR). The COMBO factor outperforms the WEIGHTED COMBO factor in most cases (except for Usefulness of the Eclipse system). Taking into consideration the amount of time and effort required to gather and combine all of the different factors, and the performance of the combined factors, we suggest using only the top performing factor.

Although, combining factors did not yield a considerable improvement in our case, previous work by Kim *et al.* [155] showed favourable improvements in performance when factors are combined. We conjecture that these differences in performance are attributed to a few differences between the two approaches: 1) we recommend 10 functions in each iteration, while Kim *et al.'s* [155] the cache is made of 10% of the functions in the system; 2) in our approach, once a function is recommended once, it is removed from the pool of functions to be recommended in the future. In Kim *et al.'s* [155] the same function can be added and removed from the cache multiple times; 3) we consider the effort required to write a test for a function (depending on its size), while in [155] the effort is not considered.

Table 6.7: Median usefulness of combined factors

	Commercial system				Eclipse system			
	LF	**COMBO**	**WEIGHTED COMBO**	**MRM**	**LF**	**COMBO**	**WEIGHTED COMBO**	**MRM**
Usefulness (%)	87.0	67.5	64.5	43.1	44.7	32.8	34.1	9.9

Table 6.8: Median POP of combined factors

	Commercial system				Eclipse system			
	LF	**COMBO**	**WEIGHTED COMBO**	**SR**	**LF**	**COMBO**	**WEIGHTED COMBO**	**MRM**
POP (%)	32.4	18.9	14.7	4.3	5.31	5.03	3.75	0.46

6.7 Discussion

During our simulation study, we needed to decide on two simulation parameters: list size and available resources. In this section, we discuss the effect of varying these simulation parameters on our results. It is important to study the effect of these simulation parameters on our results because it helps us better understand the results we obtain from the simulation. In addition, it provides some insight into ways that could lead to more effective approaches.

6.7.1 Effect of List Size

In our simulations, each of the factors would recommend a list of functions that should have unit tests. Throughout our study, we used a list size of 10 functions. However, this list size was an arbitrary choice. We could have set this list size to 5, 20, 40 or even 100 functions. The size of the list will affect the usefulness and POP values.

To analyze the effect of list size, we vary the list size and measure the corresponding POP values at a one particular point in time. We measure the median POP, from the 5 year simulation, for each list size and plot the results in Figure 6.20(a). The y-axis is the log of the median POP value and the x-axis is the list size. We observe a common trend - an increase in the list size increases the POP for all factors. Once again, our top performing factors are unchanged with LF, LM, MFF and MFM scoring in the top for all list sizes. We performed the same analysis for usefulness and obtain similar results.

This trend can be explained by the fact that a bigger list size will make sure that more functions have unit tests written for them earlier on in the project. Since these functions are tested earlier on, we are able to avoid more defects and the POP increases.

The results for the Eclipse system are consistent with the findings in Figure 6.20(a).

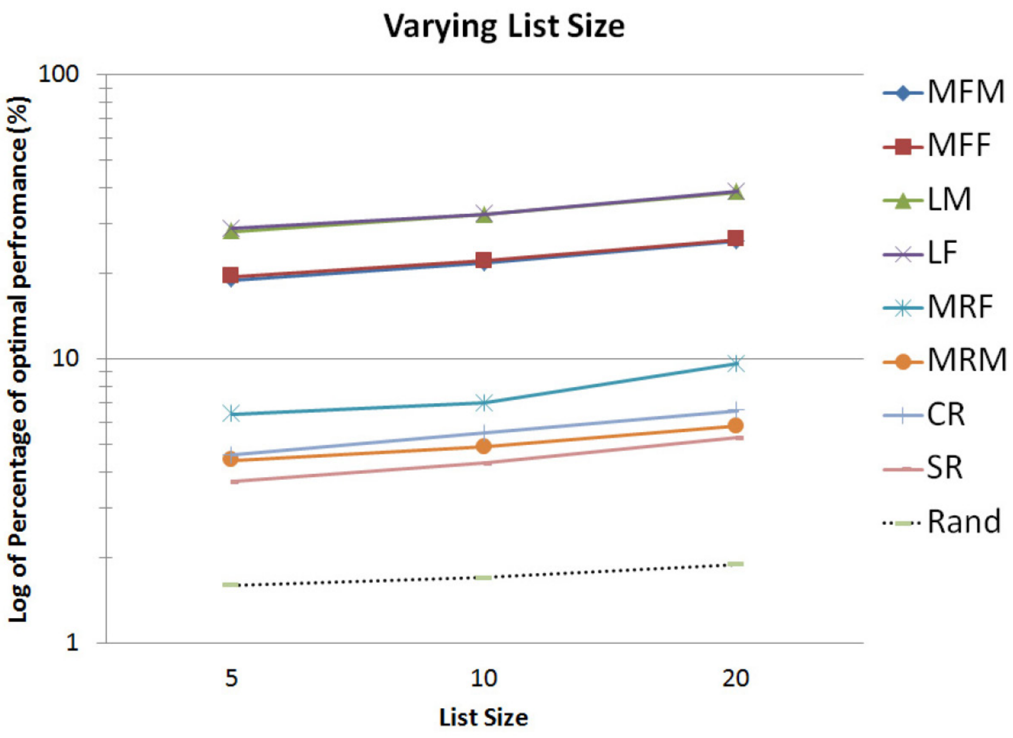
6.7.2 Effect of Available Resources

A second important simulation parameter that we needed to set in the simulations is the effort available to write unit tests. This parameter determines how fast a unit test can be written. For example, if a function is 100 lines of code, and a tester can write unit tests for 50 lines of code per day, then she will be able to write unit tests for that function in 2 days.

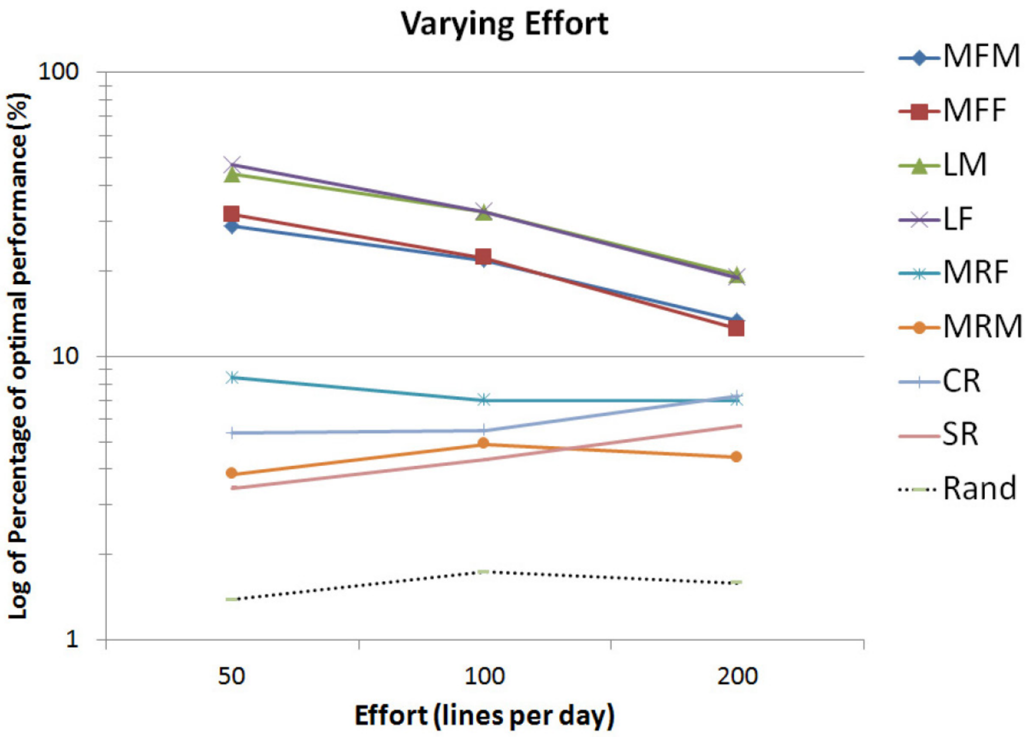
Throughout our study, we set this value to 100 lines per day. If this value is increased, then testers can write unit tests faster (due to an increase in man power or due to more efficient testers) and write tests for more functions. On the other hand, if we decrease this value, then it will take longer to write unit tests.

We varied this value from 50 to 200 lines per day (assuming the same effort is available each day) and measured the median POP, from the entire 5 year simulation. The results are plotted in Figure 6.20(b). We observe three different cases:

1. POP decreases as we increase the effort for factors LF, LM, MFF and MFM.
2. POP increases as we increase the effort for factors CR and SR.
3. POP either increases, then decreases or decreases, then increases as we increase the effort for factors MRF, MRM and Random.



(a) Effect of varying list size on POP



(b) Effect of varying effort on POP

Figure 6.20: Effect of simulation parameters for the commercial system

We examined the results in more depth to try and explain the observed behavior. We found that in case 1, the POP decreases as we increase the effort because as we write tests for more functions (i.e., increasing effort available to 200 lines per day), we were soon writing tests for functions that did not have defects after the tests were written. Or in other words, as we decrease the effort, less functions have unit tests written for them, which reduces the chance of prioritizing functions that do not have as many (or any) defects in the future. In case 2, we found that the risk factors by themselves mostly identified functions that had a small number of defects. Since an increase in effort means more functions can have unit tests written for them, therefore, we see an increase in the POP as effort is increased. In case 3, the MRF and MRM factors identify functions that are most recently modified or fixed. Any change in the effort will change the time it takes to write unit tests. This change in time will change the list of functions that should have unit tests written for them. Therefore, an increase or decrease in the effort randomly affects the POP. As for the random factor, by definition, it picks random functions to write unit tests for.

The results for the Eclipse system are consistent with the findings in Figure 6.20(b).

6.8 Threats to Validity

This section discusses the threats to validity of our study.

Threats to Construct Validity consider the relationship between theory and observation, in case the measured variables do not measure the actual factors. We used the POP and Usefulness measures to compare the performance of the different factors. Although POP and Usefulness provide effective means for use to evaluate the proposed factors, they may not capture all of the costs associated with creating the unit tests, maintaining the test suites and managing the cost of different kinds of defects (i.e., minor vs. major defects).

Threats to Internal Validity refers to whether the experimental conditions makes a difference or not, and whether there is sufficient evidence to support the claim being made. In our simulations, we used 6 months to calculate the initial set of factors. Changing the length of this ramp-up period may effect the results from some factors. In the future, we plan to study the effect of varying this initial period in more detail.

Our approach assumes that each function has enough history such that the different factors can be calculated. Although our approach is designed for large software systems, in certain cases new functions may be added, in which case little or no history can be found. In such cases, we ask practitioners to carefully examine and monitor such functions manually until enough history is accumulated to use our approach.

Although the approach presented in this Chapter assumed software systems that did not have any unit tests written for them in the past, we would like to note that this is not the only use case for this approach. In the future we plan to adapt our approach to work for software systems that may have some unit tests written for them already (since this might be commonly encountered in practice). In addition, we plan to extend the approach to leverage any other historical data (e.g., systems tests or domain knowledge) when recommending functions to write unit tests for. Domain knowledge information and existing system tests can help guide us toward specific parts of the software system that might be more (or less) important to test. In that case, we can use this information to know where we should start our prioritization.

Throughout our simulation study, we assume that all defect fixes are treated equally. However, some defects have a higher severity and priority than others. In the future, we plan to consider the defect severity and priority in our simulation study.

Threats to External Validity consider the generalization of our findings. Performing our case studies on a large commercial software system and the Eclipse system with a rich history significantly improves the external validity of our study. However, our findings may not generalize to all commercial or open source software systems since each project may have its own specific circumstances.

Our approach uses the comments in the source control system to determine whether or not a change is a defect fix. In some cases, these comments are not properly filled out or do not exist. In this case, we assume that the change is a general maintenance change. We would like to note that at least in the case of the commercial system, the comments were very well maintained.

When calculating the amount of time it takes to write a the unit test for a function in our simulations, we make the assumption that all lines in the function will require the same effort. This may not be true for all functions.

Additionally, our simulation assumes that if a function is recommended once, it needs not be recommended again. This assumption is fueled by the fact that TDM practices are being used and after the initial unit test, all future development will be accompanied by unit tests that test the new functionality.

We assume that tests can be written for individual functions. In some cases, functions are closely coupled with other functions. This may make it impossible to write unit tests for the individual functions, since unit tests for these closely coupled functions need to be written simultaneously.

In our study of the effect of list size and available resources (Section 6.7), we set the time to a fixed point and varied the parameters to study the effect on POP. Using a different time point may lead to steeper/flatter curves. However, we believe that the trends will still be the same.

6.9 Related Work

We survey the state-of-the-art in SDP in Chapter 2. In this section, we discuss the work that is most closely related to this Chapter. The most relevant related work can be categorized into two main categories: test case prioritization and fault prediction using historical data.

Test Case Prioritization

The majority of the existing work on test case prioritization has looked at prioritizing the execution of tests during regression testing to improve the fault detection rate [10,73,74,243, 290].

Rothermel *et al.* [243] propose several techniques that use previous test execution information to prioritize test cases for regression testing. Their techniques ordered tests based on the total coverage of code components, the coverage of code components not previously covered and the estimated ability to reveal faults in the code components that Rothermel *et al.* cover. They showed that all of their techniques were able to outperform untreated and randomly ordered tests. Similarly, Aggrawal *et al.* [10] proposed a model that optimizes regression testing while achieving 100% code coverage.

Elbaum *et al.* [74] showed that test case prioritization techniques can improve the rate of fault detection of test suites in regression testing. They also compared statement level and function level techniques and showed that at both levels, the results were similar. In [73], the same authors improved their test case prioritization by incorporating test costs and fault severities and validated their findings using several empirical studies [73,75,76].

Kim *et al.* [149] used historical information from previous test suite runs to prioritize tests. Walcott *et al.* [279] used genetic algorithms to prioritize test suites based on the testing time budget.

Our work differs from the aforementioned work in that we do not assume that tests are already written, rather, we are trying to deal with the issue of which functions we should write tests for. We are concerned with the prioritization of unit test writing, rather than the prioritization of unit test execution. Due to the fact that we do not have already written tests, we have to use different factors to prioritize which functions of the software systems we should write unit tests for. For example, some of the previous studies (e.g., [149]) use historical information based on previous test runs. However, we do not have such information, since the functions we are trying to prioritize have never had tests written for them in the first place.

Fault Prediction using Historical Data

Another avenue of closely related work is the work done on fault prediction. Nagappan *et al.* [209, 211] showed that dependency and relative churn measures are good predictors of defect density and post-release failures. Holschuh *et al.* [124] used complexity, dependency, code smell and change factors to build regression models that predict faults. They showed that these models are accurate 50-60% of the time, when predicting the 20% most defect-prone components. Additionally, studies by Arisholm *et al.* [19], Graves *et al.* [105], Khoshgoftaar *et al.* [143] and Leszak *et al.* [171] have shown that prior modifications are a good indicator of future defects. Yu *et al.* [295], and Ostrand *et al.* [225] showed that prior defects are a good indicator of future defects. In their follow-up work, Ostrand *et al.* [226] showed that 20% of the files with the highest number of predicted faults contain between 71-92% of the faults, for different systems that may follow different development processes [32]. Hassan [116] showed that the complexity of changes is a good indicator of potential defects.

Mende and Koschke [183] examined the use of various performance measures of defect prediction models. They concluded that performance measures should always take into account the size of source code predicted as defective, since the cost of unit testing and code reviews is proportional to the size of a module.

Other work used the idea of having a cache that recommends defect-prone code. Hassan and Holt [117] used change and fault factors to generate a Top Ten list of subsystems (i.e., folders) that managers need to focus their testing resources on. Kim *et al.* [155] extended Hassan and Holt's work [117] and use the idea of a cache that keeps track of locations that were recently added, recently changed and where faults were fixed to predict where future faults may occur (i.e., faults within the vicinity of a current fault occurrence). They performed their prediction at two levels of granularity: file- and method/function-level.

There are some key differences between our work and the work on fault prediction:

1. Our work prioritizes functions at a finer granularity than most previous work on fault prediction (except for *Kim et al.'s* approach [155] which predicts at the file and function/method level). Instead of identifying defect-prone files or subsystems, we identify defect-prone functions. This difference is critical since we are looking to write unit tests for the recommended functions. Writing unit tests for entire subsystems or files may be wasteful, since one may not need to test all of the functions in the file or subsystem.

2. Our work considers the effort required to write the unit tests for the function/method. Furthermore, since our approach is concerned with the unit test creation, we removed functions/methods after they are recommended once.

3. Fault prediction techniques provide a list of potentially faulty components (e.g., faulty directories or files). Then it is left up to the manager to decide how to test this directory or file. Our work puts forward a concrete approach to assist in the prioritization of unit test writing, given the available resources and knowledge about the history of the functions.

6.10 Conclusions

In this Chapter we present an approach to assist practitioners applying TDM prioritize the writing of unit tests for large software systems. Different factors are used to generate lists of functions that should have unit tests written for them. To evaluate the performance of each of the factors, we perform a simulation-based case study on a large commercial legacy software system and the Eclipse system. We compared the performance of each of the factors to that of a random factor, which we use as a base line comparison. All of the factors outperformed the random factor in terms of usefulness and POP. Our results showed that, in both systems, factors based on the function size (i.e., LF and LM), modification frequency (i.e., MFM) and defect fixing frequency (i.e., MFF) perform the best for the purpose prioritization of unit test writing efforts. Furthermore, we studied whether we can enhance the performance by combining the factors. The results showed that combining the factors does not improve the performance when compared to the best performing factor (i.e., LF). Finally, we examine the effect of varying list size and the resources available to write unit tests on the simulation performance.

In this Part of the thesis, we proposed approaches that aim to address the limitation of providing guidance on using SDP results. The approaches focused on making SDP models simpler and easier to understand (Chapter 5) and using simple SDP models to prioritize the creation of unit tests in large software systems (Chapter 6). In the following part, we argue that SDP is too reactive and defect-centered. We propose an approach that aims to further improve the adoption of SDP in practice by making it more encompassing and proactive by predicting risky changes.

Part III

Making SDP More Encompassing and Proactive

The majority of SDP research only focuses on predicting defects and is reactive in nature, i.e., it assumes that the defects exist in the code, and aims to identify the code that contains these defects. However, organizations are interested in more than just defects, they are interested in managing risk. Risk is more encompassing than defects. Risky changes may not even introduce defects but they could delay the release of projects, and/or negatively impact customer satisfaction. Therefore, we argue that practitioners require approaches that are more encompassing and focus on risk, rather than simply predicting defects. At the same time, we need to make SDP more proactive and develop approaches to avoid risky changes before they are incorporated into the code base. This Part presents an approach shows how SDP can be more encompassing and proactive.

- **Studying the Risk of Software Changes [Chapter 7]:** We present an approach that leverages historical data about software changes in order to predict the risk of a software change. The approach is built and validated through an empirical study that uses changes classified by the developers who made the changes at commit time. The approach shows that historical data can be used to predict risky changes so they can be avoided before they are incorporated into the code base.

  The main recommendations based on the findings of this Chapter are:

  - The developer making the change and the team they belong to need to be considered when studying the risk of a software change.
  - Developers are accurate 96.1% of the time when identifying changes that introduce defects. However, developers' identification of risky changes is less reliable when changes have many related changes.
  - Practitioners should use factors such as the number of lines and chunks added by the changes, the bugginess of the files being changed, the number of bug reports

linked to the change and the experience of the developer making the change to identify risky changes.

This Part is likely to be of interest to practitioners who work on large software systems, since such systems have many changes made to them. The approach presented here can be used to avoid or mitigate the risk of changes before they are committed into the code base. In addition, this part is of interest to the SDP research community since it shows how we can make SDP more encompassing (by focusing on risk, rather than just defects) and proactive (by predicting changes and mitigating risk before the risky code is incorporated into the code base).

Chapter 7

Studying the Risk of Software Changes

Modelling and understanding defects has been the focus of much of the Software Engineering research today. However, organizations are interested in more than just defects. In particular, they are more concerned about managing risk, i.e., the likelihood that a code or design change will cause a negative impact on their products and processes, regardless of whether or not it introduces a defect. In this chapter, we conduct a year-long study involving more than 450 developers of a large enterprise, spanning more than 60 teams, to better understand risky changes, i.e., changes for which developers believe that additional attention is needed in the form of careful code/design reviewing and/or more testing. Our findings show that different developers and different teams have their own criteria for determining risky changes. Using factors extracted from the changes and the history of the files modified by the changes, we are able to accurately identify risky changes with a recall of more than 67%, and a precision improvement of 87% (using developer specific models) and 37% (using team specific models), over a random model. We find that the number of lines and chunks of code added by the change, the defectiveness of the files being changed, the number of defect reports linked to a change and the developer experience are the best indicators of change risk. In addition, we find that when a change has many related changes, the reliability of developers in marking risky changes is affected. Our findings and models are being used today by an industrial partner to manage the risk of their software projects.

7.1 Introduction

Risk management plays a crucial part in successful project management. This is especially true for software projects. For example, a survey of 600 firms showed that 35% of them had at least one runaway project [46]. Another study showed that, industry-wide, only 16.2% of software projects are on time and on budget. Of the rest, 52.7% are delivered with reduced functionality and 31.1% are cancelled before completion. The main reason for this large amount of late projects is the lack of proper software risk management (i.e., activities used to manage the possibility of harm or loss) [46, 65].

Due to its importance in the success of software projects, researchers and industry have become more interested and active in the area of software risk management [94, 194]. One line of work that received an increasing amount of attention recently is software defect prediction, where code and/or repository data is used to predict where defects might appear in the future (e.g., [204, 310]). In fact our literature review showed that in the past decade more than 100 papers were published on defect prediction alone.

However, organizations are interested in effective management of risk in general, which covers more than just defects. For example, a recent initiative on managing technical debt aims at studying how compromises that developers make today will affect their software in the future [8]. Risky changes could introduce defects but they could also delay the release of projects, and/or negatively impact customer satisfaction. For example, changes that might have a widespread impact on the code (e.g., switching threading models) or on the user (e.g., making the software application autosave every 1 min instead of 30 seconds, for optimization reasons) are considered risky, regardless of whether or not they introduce defects. The risk is caused by the uncertainty introduced by the changes.

A risky change ideally requires additional attention through careful code/design review and possibly more testing. This is why organizations are interested in identifying risky changes as soon as possible, so that there are enough time and resources available for risk mitigation. Although prior work investigated mitigation strategies (e.g., code reviews [240]) and defect-introducing changes [153], the risk of changes, which is at the core of the software creation process, has rarely been studied.

In this chapter, we sought to better understand risk at a fine granularity, i.e., the individual software changes. We conducted a year-long study where developers from a large commercial company were asked to specify, at commit time, whether or not they consider their change to be risky. When assigning a change to be risky they are indicating that they wish additional attention to be considered for that change throughout the organization. The study involved more than 450 developers, spanning over 60 teams.

We use this large, unique data set to understand risky changes and find that:

- The interpretation of risk varies between different developers and teams. Therefore, prediction models that factor in the developer and/or team that made the change perform considerably better than general prediction models that aim to predict risk for generic changes. This finding has implications on prior work on defect-introducing changes (e.g., [85,153,259]) and for future work related to risky and defect-introducing changes.

- We can build high accuracy models to automatically identify risky changes with a recall of 67% and a precision that is 87% (developer models) vs. 37% (team models) better than a random model. Our industrial partner found these models to be of great value in their risk management processes, especially for changes done by inexperienced developers.

- Each developer and team has their own factors that best model change risk. However, in general, the number of lines and chunks of code added by the change, the defectiveness of the files being changed, the number of linked defect reports to a change and the developer experience are the best indicators of change risk.

- Risk and defectiveness are related, yet different concepts. In general, developers are accurate when classifying defective changes as being risky changes. However, changes that have many related changes are more likely to be classified incorrectly.

7.1.1 Organization of Chapter

Section 7.2 discusses the related work. Section 7.3 describes the data used in our study, while Section 7.4 introduces the case study setup. Section 7.5 presents our preliminary analysis on change risk assignment, followed by the results of our case study in Section 7.6. Section 7.7 discusses the differences between defect-introducing and risky changes. Section 7.8 reflects on the lessons learned. Finally, Section 7.9 presents the threats to validity of our study and Section 7.10 concludes the chapter.

7.2 Related Work

We survey the state-of-the-art in SDP in Chapter 2. In this section, we discuss the work that is most closely related to this Chapter. The domains most closely related to this Chapter are software risk management, defect prediction and defect-introducing change prediction.

Software risk management. Prior work by Boehm [46] proposed principles and practices of software risk management. In this work, Boehm outlines the six phases (i.e., risk identification, analysis, prioritization, management, resolution and monitoring) of risk management. Dedolph [65] studied the role of software risk management practices at Lucent Technologies in order to understand why risk management is often neglected, and he discusses examples of successful risk management. Freimut *et al.* [94] study the implementation of software risk management in an industrial setting. They proposed Riskit, a systematic risk management method, and showed that Riskit provides benefit for the risk management team with acceptable costs.

Our work is different from prior work on software risk management in that we are interested in the risk of one particular change at a time and not the risk of the entire project. Therefore, our work is done at a much finer granularity. Also, prior work on software risk management is concerned about all types of risk in the project, e.g., risk due to technicalities, risk due to personnel and/or risk due to work environment. Although our definition of risk is much wider than defects alone, we still focus on the risk due to software changes only.

File level defect prediction. Researchers in this domain train prediction models to predict defect-prone locations (e.g., files or directories). Complexity factors (e.g., McCabe's cyclomatic complexity factor [181] and the Chidamber and Kemerer (CK) factors suite [59]), size (measured in lines of code) [105, 122, 171], and the number of prior changes and defects are good predictors of defective locations [19, 30, 109, 116, 143, 171, 204, 209, 222, 263, 295, 310].

There are some key differences between the aforementioned work and our work. First, our focus is on modelling risk, not only defects. Defects are a special case of risk. Second, we perform our modelling at the change level instead of at the file level. This difference is important. Flagging risky changes makes it easier to address the risk since changes can be flagged while they are still fresh in the developer's mind and fixed before they are integrated with the rest of the code base. In contrast, defect prediction flags files later in the release cycle, at which time a developer may have forgotten the issues surrounding their changes [153]. Furthermore, changes can be easily assigned to the developer who made the change, in contrast to files in defect prediction, which are changed by many developers, making it hard to decide who to assign the file to. Finally, changes provide the necessary context to address the flagged issue, whereas in defect prediction in some cases a defect spans many files that are changed together.

Change level defect prediction. The majority of the change-level related work aims to predict defect-introducing changes. On the other hand, our aim is to understand and identify risky changes (which are a superset of defect-introducing changes). The work that most closely relates to ours is the prior work by Mockus and Weiss [200] and Kim *et al.* [153].

Sliwerski *et al.* [259] studied defect-introducing changes in Mozilla and Eclipse. They find that defect-introducing changes are part of large transactions and that defect fixing changes and changes done on Fridays have a higher chance of introducing defects. Eyolfson *et al.* [85] study the correlation between a change's bugginess/defectiveness and the time of the day the change was committed and the experience of the developer making the change. They perform their study on the Linux kernel and PostgreSQL and find that changes performed between midnight and 4AM are more defect-prone than changes committed between 7AM and noon and that developers who commit regularly produce less buggy/defective changes. Yin *et al.* [294] performed a study that characterizes incorrect defect-fixes in Linux, OpenSolaris, FreeBSD and a commercial operating system. They find that 14.8 - 24.2% of fixes are incorrect and affect end users, that concurrency defects are the most difficult to correctly fix and that developers responsible for incorrect fixes usually rarely have enough knowledge about the code being changed. Kim *et al.* [153] use change features like the terms in added and deleted deltas, terms in directory/file names, terms in change logs, terms in source code, change metadata and complexity factors to classify changes as being defective (i.e., defect-introducing) or clean (i.e, not defect-introducing).

Our work complements the prior work on defect-introducing changes in a number of ways. First, we use a more general definition of change risk, assigned by developers who make the changes, which includes defect-introducing changes, i.e., defect-introducing changes are a special case of risky changes. Second, we perform our study on a large commercial system, whereas most of the prior work is performed on open source systems. Third, we quantify the effect of factors that indicate risky changes and compare them to those of defect-introducing changes.

Mockus and Weiss [200] assess the risk of Initial Modification Requests, called IMRs, which are groups of code changes, of the 5ESS commercial project. They predict the potential of an IMR to cause a failure (i.e., introduce a defect) using IMR diffusion, size, interval, purpose and experience factors. Czerwonka *et al.* [60] present their experiences with CRANE, a tool used within Microsoft for failure prediction, change risk analysis and test prioritization.

Our work complements the work by Mockus and Weiss [200] in a number of ways. First, we provide recommendations at a finer granularity (i.e., at the individual change level). Second, our unique data set allows us to study the risk as viewed by the developers making the changes, i.e., the risk is assigned by the individual developers making the changes instead of simply using the potential of change to cause a failure. Third, our work studies the impact of more factors and quantifies the effect of the important factors. Also, our work is different than the work by Czerwonka *et al.* [60] in two ways. First, in their definition of risk, the authors use the likelihood of a change to introduce a defect as a proxy for risk. Second, the authors perform their analysis at the granularity of a binary, which is generally made up of hundreds or thousands of files (i.e., equivalent to an IMR). In contrasts, our work is performed at the change level, a granularity much finer than the binary level. *To the best of our knowledge, our study is the only study that focuses on studying the risk of a change assigned by developers.*

7.3 Case Study Data

In this section, we describe the software system and the data used in our study.

7.3.1 The Software System

Our study is performed on a large, well established commercial mobile-phone software system. The system is written mainly in Java, with lower-level functionality implemented in the C/C++ language, and is used by tens of millions of users across the globe. The system is developed by many different teams, of which more than 60 were involved with our study, and has been in development for over 20 years. The current size of the code base is in the order of several millions of lines of code.

7.3.2 Change Data

To conduct our study, we used change-level information. Changes (or commits) are submitted to the Source Configuration Management (SCM) system by developers to perform maintenance tasks (e.g., fix defects) or enhance features (e.g., implement new functionality). A change is done by one developer and may touch one or more files. The SCM stores metadata about each change, such as the change ID, date and time of the change, the developer's name, the change type (e.g., defect fix?), whether the change fixes a previous change, the files touched and how many lines and chunks of code were added, deleted or modified for each file, together with a short description of the change. For the purpose of our study, an optional field (drop-down menu) was provided for developers to indicate whether they consider their change to be risky or not. By default, the drop-down is blank, indicating an unclassified change. Developers are then given the option to assign the right level of change risk.

The general rule communicated to all developers regarding risky changes is that a change is considered risky if additional attention like careful code reviewing and possibly more testing is deemed necessary. On the other hand, a non-risky change is one where the change does not need any special treatment in terms of code review or testing. The change can be safely integrated into the code without having any (expected) negative impact.

The change data was extracted and parsed in order to extract different factors that we use to perform our study. The factors are detailed later in Section 7.4.

7.3.3 Summary of Data

We observed and collected changes made between December 2009 and December 2010. Our final data set contained changes made by over 450 unique developers, spanning more than 60 different teams. Since assigning the risk to changes is optional, we found that, on average, developers assigned risk to more than 40% of all the changes they submitted. After removing sync and branch changes (which did not modify any code), our final data set contained a total of over 7,000 changes with risk assigned.

Given the fact that such data is rarely available to researchers, we were extremely grateful to have such a rich data set to perform our study. Due to confidentiality restrictions, we are not able to be more specific about the exact numbers or provide any more details about the data used. That said, we believe that the details given provide sufficient context about our study.

7.4 Case Study Setup

We begin by presenting the various factors used to study the risk of changes. Then, we describe the modelling techniques used and our evaluation criteria for the generated models.

The goal of our study is to better understand risky changes, so that they can be addressed by practitioners and their risk can be mitigated. In particular, we would like to identify risky changes and determine which factors are associated with them. We formalize our study into the following research questions:

RQ1 Can we effectively identify risky changes?

RQ2 Which factors play an important role in identifying risky changes? What is the effect of these factors on the riskiness of a change?

To answer the aforementioned questions, we extract a number of factors that we use to empirically study risky changes.

7.4.1 Studied Factors

Table 7.1 shows all of the factors used in our study. For each factor, we provide its type (e.g., numeric), an explanation of the factor and the rationale for using the factor in our study.

We group the studied factors into six different dimensions:

Time: The main motivation for using this dimension is to study whether the time when changes are made has an impact on their risk, since prior work used time factors to model defect introducing changes in Open Source software [85, 259] .

Size: Prior work showed that churn is a good indicator of defectiveness at the file [210] and IMR [200] level. IMRs consist of multiple Modification Requests (MR), which are made up of multiple changes, hence they are at a much coarser granularity than our changes. We investigate whether the size of a change (measured in number of lines added, deleted and changed) is also a good indicator of change risk. Furthermore, we use size as a proxy for complexity since prior work showed that complexity measures are highly correlated with size [105]. In addition to counting the number of lines, we also consider the number of locations (i.e., chunks) that these lines were spread over.

Files: Prior work showed that process factors, such as the number of prior defects, are a good indicator of future defects [204, 310]. Therefore, this dimension considers the history of the files being modified by the change. For example, if a file has been changed many times in the past, then a change that modifies this file may be more risky. Since most other factors only provide a snapshot view at the time the change was made, using the history of the files modified by the change is a way of incorporating history into the change risk models.

Code: The motivation behind using the code dimension is to study whether the code being modified (e.g., API code) by the change is a good indicator of whether or not a change is risky. Since the software system under study is written in different programming languages, we introduced four boolean variables that indicate whether or not a change modifies Java code, C++ code, other code (e.g., html or xml pages) or API code. The type of code being modified provides insight into whether the change deals with application layer code or lower level OS code. The file extension was used to determine the type of code changed.

Purpose: Prior work showed that the purpose of an IMR (e.g., whether it was a defect fixing change) is a good indicator of its failure potential [200]. We study whether the purpose of a change impacts its risk.

Personnel: Prior work showed that the experience of the developers changing an IMR is a good indicator of its risk [200]. Therefore, we also investigate the usefulness of the developer experience in identifying risky changes.

In total, we extracted 23 different factors that covered six different dimensions. It is important to note that all of our factors can be easily extracted from the change log stored in widely available source code control repositories. This was pointed out by the industrial partner to be a major advantage of this work and makes this work applicable to any company or project that uses source code control repositories.

Table 7.1: Factors used to study risky changes

Dim.	Factor	Type	Explanation	Rationale
Time	Hour	Numeric	Time when the change was made, measured in hours (0-23).	Changes performed at certain times in the day, e.g., late afternoons, might be done by over-worked or less aware developers, hence, these changes may be more risky [85].
	Weekday	Numeric	Day of the week (e.g., Mon, Tue, etc.) when the change was performed.	Changes performed on specific days of the week (e.g., Fridays) are not as carefully examined and might be more risky [259].
	Month day	Numeric	Calendar day of the month (1-31) when the change was performed.	Changes performed during specific periods, i.e., beginning, mid or end of the month might be rushed to meet end-of-the-month quotas and are likely to be more risky.
	Month	Numeric	Month of the year (0-11) when the change was performed.	Changes performed in specific months, e.g., later in the year or during holiday months like December, when less developers and expertise are available, might be more risky.
Size	Lines Added	Numeric	The number of lines added as part of the change.	Changes that add more lines add new functionality that has not been tested as thoroughly, therefore, they might be more risky.
	Chunks Added	Numeric	The number of chunks (i.e., different sections) added as part of the change.	Changes that add more chunks, i.e., are more spread out, are harder to make and hence are considered more risky.
	Lines Deleted	Numeric	The number of lines deleted as part of the change.	Changes that delete more code might remove too much or remove code incorrectly, making the change more risky.
	Chunks Deleted	Numeric	The number of chunks (i.e., different sections) deleted as part of the change.	Changes that delete more chunks, i.e., are more invasive, are harder to make and are more risky.
	Lines Modified	Numeric	The number of lines modified as part of the change.	Changes that modify more lines have a higher chance of making incorrect changes and are therefore more risky.
	Chunks Modified	Numeric	The number of chunks (i.e., different sections) modified as part of the change.	Changes that modify more chunks, i.e., are more invasive, are harder to make and are considered more risky.
	Churn	Numeric	The total number of lines added, deleted and modified as part of the change.	Changes that have high churn are harder to make and are considered more risky [210].
Files	Number of Files	Numeric	The number of files modified by the change.	Changes that touch more files require a higher degree of knowledge of the different files and are therefore more risky [116, 259].
	No. file devs	Numeric	The number of unique developers that modified the changed files. If a change modifies multiple files we use the number of developers of the file that has the most developers.	Files that have been changed by many developers are hard to modify. A change that touches a file that has been modified by many different developers is more risky [44].
	No. file changes	Numeric	The number of past changes to the files modified by the change. If a change modifies multiple files, we use the number of changes of the file with the most past changes.	Files that are changed often are hard to modify. A change that touches such a file is more risky [204].
	No. file fixes	Numeric	The number of past defect fixes to the files modified by the change. If a change modifies multiple files, we use the number of defect fixes of the file with the most past defect fixes.	Files that are fixed often tend to be defective. A change that touches such a file is more risky [310].
	File defectiveness	Numeric	The ratio of defect fixes to total changes of a file. If a change touches more than one file, we use the value of the file with the largest file defectiveness.	Files may be changed often to make additions or general improvements. If most of those changes are fixing defects, then a change that touches such a file is more risky.
Code	Modify Java	Boolean	Indicates whether the change modifies Java code.	Changes that modify code are changing application behaviour and hence are more/less likely to be risky.
	Modify CPP	Boolean	Indicates whether the change modifies C++ code. For this project, only low-level functionality was implemented in C++.	Changes that change low-level functionality are more risky.
	Modify Other	Boolean	Indicates whether the change modifies anything other than Java and C++ code, e.g., documentation files.	Changes that do not change code are less risky.
	Modify API	Boolean	Indicates whether the change modifies any APIs.	Changes that modify APIs can potentially affect all client code using the API, hence they are more likely to be risky.
Purpose	Defect Fix?	Boolean	Indicates whether the change fixes a defect.	Changes that fix a defect are more complex and are therefore more risky [259].
	No. of Linked Defect Reports	Numeric	Indicates the number of defect reports that are linked to the change.	Changes that are linked to multiple defect reports need to make larger changes and are therefore more risky.
Person	Dev. Experience	Numeric	Indicates the experience of the developer who made the change. Experience is measured as the number of previous changes done by the developer.	Changes done by experienced developers are less risky [44].

7.4.2 Logistic Regression Models

In this work, we are interested in identifying risky changes and determining which factors best indicate risky changes. Similar to prior work [310], we use a logistic regression model. A logistic regression model correlates the independent variables (i.e., the 23 factors in Table 7.1) with the dependent variable (i.e., whether or not a change is risky).

The output of our logistic regression model is a probability (between 0 and 1) of the likelihood that a change is risky. Then, it is up to the user of the output of the logistic regression model to determine a threshold at which she/he will consider a change as being risky. Generally speaking, a threshold of 0.5 is used. For example, if a change has a likelihood of 0.5 or higher, then it is considered risky, otherwise it is not.

However, the threshold is different for different data sets and the value of the threshold affects the precision and recall values of the prediction models. In this chapter, we determine the threshold for each model using an approach that examines the tradeoff between type I and type II errors [200]. Type I errors correspond to files that are identified as being risky, while they are not. Having a low logistic regression threshold (e.g., 0.01) increases type I errors: a higher fraction of identified changes will not really be risky. A high type I error leads to a waste of resources since many non-risky changes need to be reviewed in vain. On the other hand, the type II error is the fraction of risky changes that are not identified as being risky when they should be. Having a high threshold can lead to large type II errors, and thus missing many risky changes.

To determine the optimal threshold for our models, we perform a cost-benefit analysis between the type I and type II errors. Similar to previous work [200], we vary the threshold value between 0 to 1, in increments of 0.01, and use the threshold where the type I and type II errors are equal. We report the thresholds used for each model in the results tables of Sections 7.5 and 7.6.

Initially, we built the logistic regression model using all 23 factors. Having a large number of factors is beneficial since it allows us to conduct a comprehensive study (i.e., take into account many factors in our models). However, using many factors in our models introduces the risk of having issues due to multi-collinearity. Multi-collinearity is caused by having highly correlated factors in a single model, making it difficult to determine which factors are actually causing the effect being observed and introducing high variance to the corresponding coefficients [55]. To alleviate such collinearity issues, we employ feature selection [97] to remove all redundant (i.e., highly correlated) factors from our models. In particular, we use the `cfs` selector [112], which performs the feature selection based on correlation and entropy. To ensure that the effect of the independent variables is statistically significant, we perform an ANOVA analysis and retain all variables with p-value < 0.05. We provide a list of the factors used in our models in Section 7.6, Tables 7.6 and 7.7.

7.4.3 Evaluating the Accuracy of Our Models

We use two criteria to evaluate the performance of the logistic regression models: Predictive Power and Explanative Power.

Explanative Power

Explanative power ranges between 0-100%, and quantifies the variability in the data explained by the model, i.e., how well the model fits the data. When calculating the explanative power, the model is built using all of the data (i.e., we do not split the data into training and testing sets). In addition, we report and compare the variability explained by each factor used in the model, to determine which of the factors are most important. The relative importance of each factor is determined by comparing its explained variability to that of the other factors in the model.

Table 7.2: Confusion matrix

	True class	
Classified as	Yes	No
Yes	TP	FP
No	FN	TN

Predictive Power

Predictive power measures the accuracy of the model in modelling the risk of a change. We calculate recall and relative precision based on the classification results in the confusion matrix (shown in Table 7.2).

Relative Precision: is the improvement in precision by our prediction model over the precision of a baseline model. In our case, the baseline model is a model that randomly predicts risky changes. For example, if a baseline model randomly predicts risky changes and achieves a precision of 20%, while our proposed prediction model achieves a precision of 40%, then the relative precision is given as $\frac{40}{20} = 2X$. In other words, using our model provides twice the precision of the baseline model. The higher the relative precision value the better the model is at classifying risky changes. We use relative precision instead of actual precision for confidentiality reasons, since precision allows one to infer the ratio of risky-changes in our dataset.

Recall: is the percentage of correctly classified risky changes relative to all of the changes that are actually risky: Recall $= \frac{TP}{TP+FN}$. A recall value of 100% indicates that every risky change was classified as being risky.

When evaluating the predictive power of our models, we employ 10-fold cross validation [69], where the data set is divided into 10 sets, each containing 10% of the data. One set serves as the testing data and the remaining nine sets are used as training data. The model is trained using the training data and its accuracy is tested using the testing data. In our results, we report the results from the 10-fold cross validation.

7.5 Preliminary Analysis

Prior to delving into our case study, we discuss our initial findings concerning change risk assignment. We started our analysis by building a general model based on the changes of all the developers combined, similar to prior work (e.g., [153]). The results of the model are shown in the first row (labeled "All Factors") of Table 7.3. This table also contains the threshold value used for the logistic regression prediction model that we determined based on the training data set (not the testing data set).

Our findings show that the model achieves good predictive power (i.e., precision and recall), however, the explanative power of the model is very low. We qualitatively examined a random subset of 50 risky changes to try and understand this low explanative power. We found that, although all developers were given the same criteria to label risky changes, the concrete interpretation of risk is ultimately a concept that depends on the individual developers and teams. For example, teams that worked on application-level code were less likely to mark their changes risky unless they were large. On the other hand, members of the UI team would mark their changes risky if they thought that their changes would impact other parts of the code, regardless of the size of the change. The same was observed for developers as well, each had their own criteria for marking risky changes.

Following this finding, we decided to investigate whether or not the team and the developer assigning the risk played a role in risk being assigned to the changes. In our case, a team is composed of multiple developers and each team works on one component. We added the team name to the initial model as an additional factor. The results were much better, as shown in the second row of Table 7.3 (labeled "All Factors + Team") . Adding the team name improves both predictive and explanative power, indicating that when the risk of a change is considered one needs to discriminate between changes from different teams. Next, we added the developer name to the model containing all factors, as shown in the third row of Table 7.3 (labeled "All Factors + Developers"). We observe that adding the developer name to the model improves the predictive and explanative power even further.

Our findings here show that, although all developers where given the same rule to classify risky changes, the risk assigned to a change depends on the developer that is assigning the risk and the team that the developer belongs to. Based on these findings we recommend that developer or team specific models should be built when modelling change risk. Building one model to model the risk of *all* changes (i.e., changes from different developers) is not an effective solution.

Table 7.3: Role of developer and team name on change risk classification

	Predictive Power			**Explanative Power**
Component	Precision	Recall	Thresh.	Deviance Explained
All Factors	1.32X	59.4%	0.482	4.4%
All Factors + Team	1.42X	67.5%	0.496	14.2%
All Factors + Developers	1.72X	77.5%	0.496	32.6%

Even when all developers are given the same rule to classify risky changes, the risk of a change varies and depends on the developer that is assigning the risk and the team that the developer belongs to.

7.6 Case Study

In this section, we answer the research questions posted earlier. In particular, we examine the accuracy of our approach in identifying risky changes. Then, we determine the most important factors when identifying risky changes, as well as the factors' specific impact.

RQ1. Can we effectively identify risky changes?

Motivation: In order to address and assign the proper quality assurance efforts, we need to be able to effectively identify risky changes. Our goal is to examine whether it is feasible to build accurate models that flag risky changes.

Approach: In the previous subsection, we showed that the team and the developer names play a major role in accuracy of the change risk models. Therefore, we now build specific models at two levels: the *developer level* and the *team level*. At the developer level, we build a specific model for each developer (instead of one global model with the developer name as an independent variable). At the team level, we build a specific model for each team. Since these models are tailored to the individual team and developers, we expect them to be more accurate than a global model that does not consider the team or developer.

In order to build the logistic regression models, we needed to make sure that enough data was available for each developer. Therefore, we selected developers who made at least 20 changes over the year studied. Since we are building developer-specific models, we also require that a developer has both risky and non-risky changes. This is needed to train our models (i.e., we cannot train a good model using only risky changes or only non-risky changes). Therefore, we required that at least 20% of a developer's changes belong to either class, risky or non-risky. Then, we ranked the developers based on the total number of changes they committed and built models for the top 10. Ideally, we would want to make predictions for the developers with the most committed changes, since a manual risk assessment would be the most difficult for them. For developers that have fewer changes, manual examination might be a viable solution.

For the team models, we aggregated developers based on the team they belong to. We ranked the teams based on the total number of changes and built models for the top 10 teams. Teams that have the most changes will benefit the most from our models since manual risk assessment of their changes will be a resource intensive task. As mentioned earlier for developers, for teams that have fewer changes, manual examination might be a viable solution.

Results - Developer Level: Table 7.4 shows the predictive and explanative power results for the top 10 developers. In terms of predictive power, our models achieved very promising results. On average the model achieves 1.87X relative precision (or a 87% improvement in precision over the baseline model), while achieving an average recall of 67.7%.

On average, the explanative power of our models is 20.8%. This explanative power is comparable to models that have been built in previous work to predict post-release defects in files [36, 55].

Results - Team Level: Table 7.5 presents the results for the team level models. On average, the team level models achieve a relative precision of 1.37X and an average recall of 67.9%. In terms of explanative power, the team level models achieve an average explanative power of 13.3%.

As suggested by our preliminary analysis, the developer models outperform the team models in terms of predictive and explanative power. The main reason for this is the fact that the team models are less specific, since they incorporate more developers. However, an advantage of team level models is that they are more practical, since we would need less models to be built (all developers of a team could share the same model).

Final Remarks: Another point worth addressing is the fact that relative precision values range between 1.5X - 2.76X for developers, and 1.09X - 1.76X for teams, and recall values range between 48.0 - 81.8% for developers, and 57.2 - 80.9% for teams. This range is due to the fact that different developers and teams have a different distribution of risky to non-risky changes. For example, Dev4 had more changes and a better balance of risky to non-risky changes than Dev7. Therefore, our prediction models were able to provide better accuracy for Dev4 than Dev7. That said, we believe that the average improvements provided by our prediction models are high enough to make them applicable in practice.

Table 7.4: Performance of developer-level change risk classification

	Predictive Power			**Explanative Power**
Developer	Precision	Recall	Thresh.	Deviance Explained
Dev1	1.58X	66.8%	0.464	22.6%
Dev2	1.50X	55.1%	0.43	22.0%
Dev3	2.03X	64.1%	0.32	15.3%
Dev4	2.76X	75.5%	0.302	42.6%
Dev5	1.69X	76.6%	0.544	12.3%
Dev6	1.61X	64.9%	0.518	18.5%
Dev7	1.28X	48.0%	0.394	8.0%
Dev8	1.82X	65.2%	0.416	19.9%
Dev9	1.72X	81.8%	0.55	27.4%
Dev10	2.72X	77.8%	0.482	19.0%
Average	1.87X	67.6%	-	20.8%

We can accurately identify risky changes, achieving average recall of 67% and precision improvement of 87% (for developer models) and 37% (for team models), over a baseline model.

RQ2. Which factors play an important role in identifying risky changes? What is the effect of these factors on the riskiness of a change?

Motivation: In addition to identifying risky changes with high accuracy, we are interested in knowing which factors are good indicators of risky changes and by how much these factors affect the riskiness of a change. Knowing which and by how much each factor relates to risky changes helps practitioners determine what factors they should be on the look out for when determining which changes to carefully examine.

Approach: To study the importance of the factors in the prediction models, we perform an ANOVA analysis and examine the relative contribution (in terms of explanative power) of each factor to the logistic regression model.

In addition, similar to prior work [197], we measure the effect of each factor by building a model where all factors are set to their median values. Then, we double the median of one of the factors (while holding all other factors at their median values) and measure the difference in the modeled probabilities. The effect of a factor can be positive or negative. A positive effect indicates that a higher level of a factor corresponds to an increase in change risk, while a negative effect indicates that a higher level of a factor corresponds to a decrease in change risk.

The analysis is done for the models of the top 10 developers and teams mentioned in Tables 7.4 and 7.5, respectively. We show the results for the developers and teams 1, 5 and 10 in detail and summarize and discuss all of the models afterwards.

Results - Developer Level: Table 7.6 shows the most important factors for Dev1, Dev5 and Dev10. Only the factors used in the final model (i.e., after applying feature selection and checking for statistical significance) are shown. The Explanative Power column shows the variability explained by each factor. The higher the deviance explained of the factor, the more important it is to the model. We use this measure as a way of gauging the importance of the factors. For example, for Dev1 the "Chunks Added" factor is the most important factor in determining the risky changes. This means that if a future change is made by Dev1 and there are many "Chunks Added", one has to be cautious about the change since it likely increases the risk of the change.

The Effect column in Table 7.6 shows the effects of each factor for Dev1, Dev5 and Dev10. All of the factors have a strong positive effect with change risk. Comparing the different factors shows that for Dev1, the number of defect reports linked to a change (e.g., the change addresses a major defect or multiple defects) has the strongest relationship with change risk. For Devs5 and 10, the number of code lines added also has a strong positive relationship with change risk. In addition, file defectiveness has an extremely large positive relationship with change riskiness for Dev10.

Results - Team Level: Table 7.7 shows the most important factors for Team1, Team5 and Team10. In all three models, code additions (either the number of code chunks added or lines added) are strong indicators of risky changes. Once again, we find that all of the factors have a positive effect with risky changes, i.e., higher values indicate higher risk. For Team1, the number of defect reports linked to a change has a strong effect on risky changes. For team5, we find that the number of fixes to the file modified by the change has the strongest effect. We were not able to calculate the effect for the hour factor, since doubling the median does not make sense (i.e., doubling hour 23 to be 46 does not make sense). For Team10, we find that the number of lines added and file defectiveness both have a strong and positive relationship with change risk.

From the aforementioned results, we make two noteworthy observations. First, each developer and team has their own set of factors that best predict the risk of their changes. Second, the models are very simple, containing at most 3 or 4 factors. This simplicity makes these models more attractive to practitioners, who can easily apply and interpret such simple models in practice.

Table 7.5: Performance of team level change risk classification

	Predictive Power			**Explanative Power**
Component	Precision	Recall	Thresh.	Deviance Explained
Team1	1.76X	57.2%	0.448	6.9%
Team2	1.57X	80.9%	0.524	22.6%
Team3	1.28X	60.5%	0.486	7.38%
Team4	1.15X	77.3%	0.588	9.0%
Team5	1.14X	57.7%	0.502	5.6%
Team6	1.09X	79.4%	0.59	10.0%
Team7	1.69X	65.6%	0.478	15.6%
Team8	1.43X	69.9%	0.508	16.6%
Team9	1.30X	69.3%	0.506	25.33%
Team10	1.25X	71.2%	0.534	13.9%
Average	1.37X	67.9%	-	13.3%

Results - Summary In addition to providing the important factors for developers and teams 1, 5 and 10, we provide a summary of the important factors in each dimension for all 10 developers and teams in Table 7.8. The most important factor in each dimension is shown in the column labeled "Most Important Factor". The "Importance" column shows the number of the top 10 developers that a dimension was important for. For example, the lines added factor was the most important factor for 7 of the top 10 developers (as shown in the second row of Table 7.8). On the other hand, factors in the time dimension were not important for any of the top 10 developers.

From Table 7.8 we observe that, for both developer and team levels, the most important dimensions are the size and file dimensions, with the number of lines of code added, number of chunks of code added, number of files and file defectiveness being the most important factors within these dimensions. Purpose (No. of linked defect reports) and Personnel (Dev. experience) factors are the next most important dimensions, with code (modify CPP, for team level) and time (hour, for team level) dimensions being the least important.

Table 7.6: Most important factors for Dev1, Dev5 and Dev10

Model	Factor	Explanative Power	Effect
Dev1	Chunks Added*	11.7%	142%
	Chunks Deleted**	5.8%	120%
	Chunks Modified*	2.8%	131%
	No. of Linked Defect Reports*	2.3%	162%
Dev5	Lines Added**	12.3%	274%
Dev10	Lines Added*	9.8%	268%
	File Defectiveness**	9.2%	1114%

($p < 0.001$ ***; $p < 0.01$ **; $p < 0.05$ *)

The number of lines and chunks being added, the defectiveness of the files being changed, the number of linked defect reports to a change and the developer experience are the most important indicators of risky changes.

7.7 Discussion

The majority of prior work (e.g., [85, 153]) used defect-introducing changes as a measure of risky changes. However, we argued that risky changes are more than just defect-introducing changes. We believe that risky changes encompass defect-introducing changes as well as other changes that may have a high impact on the software product and/or its users.

To better understand this difference, we compare risky changes to defect-introducing changes by comparing the factors that best indicate risky changes and defect-introducing changes, as well as by analyzing the classification of defect-introducing changes.

7.7.1 Factors used to indicate defect-introducing and risky changes

Similar to the preliminary analysis in Section 7.5, we build a global model that includes the risky changes from all developers and we compare it to a model that contains the defect-introducing changes from all developers. Table 7.9 shows the factors in the resulting two models. We find that the number of lines added is an important factor for both defect-introducing and risky changes, having a higher effect for defect-introducing changes. However, for risky changes, two additional factors (i.e., the file defectiveness and the number of developers who touched the changed files in the past) are considered to be important. This finding gives an indication that risky changes are likely different from defect-introducing changes, since different (and in this case more) factors are required to identify them.

Table 7.7: Most important factors for Team1, Team5 and Team10

Model	Factor	Explanative Power	Effect
Team1	Chunks Added**	3.99%	137%
	No. of Linked Defect Reports**	1.68 %	195%
	Number of Files*	1.21%	174%
Team5	Chunks Added**	3.08%	120%
	No. File Fixes**	1.8%	125%
	Hour*	0.75%	-
Team10	Lines Added***	11.7%	134%
	File Defectiveness*	2.2%	138%

($p < 0.001$ ***; $p < 0.01$ **; $p < 0.05$ *)

Table 7.8: Summary of most important factors for top 10 developers and teams

	Developer-Level		Team-Level	
Dim.	**Most Important factor**	**Importance**	**Most Important factor**	**Importance**
Time	-	0	Hour	2
Size	Lines Added	7	Chunks Added	10
Files	No. of Files & File Defectiveness	7	File Defectiveness	6
Code	-	0	Modify CPP	3
Purpose	No. of Linked Defect Reports	2	No. of Linked Defect Reports	4
Personnel	Dev. Experience	1	Dev. Experience	4

7.7.2 Classification of Defect-Introducing Changes as Risky Changes

We now investigate how accurate developers are at identifying *defect-introducing* changes as *risky* changes. To do so, we examine the entire set of 7,000 changes that had been assigned a risk value by the developers. We examine all of the changes labeled as *not* risky and determine how many of those changes introduced a defect, i.e., how often defect-introducing changes slipped through without being noticed as risky.

We construct a confusion matrix, similar to that shown in Table 7.10. Due to confidentiality reasons, we are unable to show the exact numbers for the confusion matrix, i.e., we can only provide the corresponding ratio of accuracy. We calculate the ratio of accuracy of developers in classifying defect-introducing changes as $\frac{LI}{LI+LNI}$. That is, we measure the ratio of changes labeled as being *not* risky and introducing a defect (i.e., LI) divided by the total number of changes labeled as being *not* risky (i.e., LI+LNI). This value was 3.1%. This means that when developers classify a change as being *not* risky, they are correct 96.9% of the time that the change will not introduce a defect (although it could still cause other issues, such as delay, which are not considered to be defects). This high level of accuracy is encouraging, showing that developers are good at assessing non-risky changes. However, this now brings up the question: Why are some defect-introducing changes misclassified by developers as being non-risky changes?

7.7.3 Why are Some Defect-Introducing Changes Misclassified?

Our finding shows that in some cases developers incorrectly classified defect-introducing changes, marking them as safe changes. In order to better understand why such changes were incorrectly classified, we compare all the correctly classified (i.e., marked as not risky and not introducing a defect) and all the incorrectly classified (i.e., marked as being not risky and later introducing a defect) changes on the following:

- **Cause for the change**: For each change, developers entered a reason for the change. We compared the percentages of each of the eight possible causes (shown in Table 7.11) between the correctly and incorrectly classified changes. The purpose of this analysis is to investigate whether there is a specific cause of a change that is more likely to be incorrectly classified.
- **Defect fixing change?**: We compare the percentage of defect-fixing changes in the correctly and incorrectly classified changes. The purpose of this analysis is to investigate whether defect fixing changes are more likely to be classified incorrectly.
- **Has related Changes**: If a change has other changes related to it (e.g., it requires changes made by others or depends on functionality recently modified by other changes), those changes are explicitly added in the change commit log. We compared the percentage of changes that have related changes for the correctly and incorrectly classified changes. The intuition for looking at related changes is to examine whether changes that have related changes are harder to classify.
- **Modifies API**: If a change modifies API code, it is flagged by developers. The main idea is to make other developers aware that this change could potentially affect other code. We examine the difference between correctly and incorrectly classified changes to see whether changes that change API code are more likely to be incorrectly classified.

Table 7.11 summarizes our findings. The table presents the average percentage of changes in each category. For example, the first row of the table shows that 11.7% of the incorrectly classified changes were due to unclear requirements, whereas 11.5% of the correctly classified changes were due to unclear requirements. In this case, it is clear that a change caused by unclear requirements has no increased chance of being incorrectly classified. We also see a very small difference in classification accuracy for changes made as a side-effect of other changes. Changes due to unclear documentation, due to inadequate testing, due to coding errors, due to design flaws and defect fixing changes are more likely to be correctly classified than not. Changes due to a scope change are slightly more likely to be incorrectly classified. In contrast, changes due to integration errors (i.e., the change was made to fix an integration error) and changes that modify API code are twice as likely to be incorrectly classified. Also, *changes that have related changes are 10 times as likely to be incorrectly classified*. This finding indicates that although developers are aware of the fact that there are related changes, they are not aware of the potential risk of these related changes (i.e., since they are marking them as being not risky, these changes end up introducing defects later on).

To make sure that our findings are chosen from a representative sample, we measured the number of unique developers responsible for the changes used in this analysis. We found that the incorrectly classified changes were made by more than 60 unique developers and the correctly classified changes were made by more than 370 developers.

Based on these findings, we recommend that developers carefully consider their risk assignments for changes that are caused by integration errors, that have related changes and that modify API code.

Table 7.9: Most important factors when classifying defect introducing changes

Model	**Factor**	**Effect**
Defect-Introducing Changes	Lines Added***	+180%
Risky Changes	Lines Added***	+128%
	File Defectiveness***	+102%
	File Devs***	+131%

(p < 0.001 ***; p < 0.01 **; p < 0.05 *)

Table 7.10: Risky vs. defect-introducing changes

	Defect-Introducing	
Risky	Yes	No
High	HI	HNI
Low	LI	LNI

7.8 Lessons Learnt

After performing our study, we asked the opinions of an experienced development manager in the company about our findings. The manager leads one of the teams studied as part of this chapter.

The development manager was excited about the findings and suggested that we build a recommendation tool that can be leveraged by him and other team managers to assign quality assurance efforts for risky changes. Based on the prediction models in Section 7.6, we built a prototype tool that is currently being used by teams within the company to automatically classify their changes. The tool is still in its early stages and features are being added to improve it.

At this early stage, the tool is just starting to be used to classify risky changes. Instead of having to rely on gut feelings, developers can now verify their intuition with a tool that can quantify the risk of a change. Changes that have a mismatch between the tool's classification and the manual classification are being investigated in more detail. Furthermore, our approach is also used to classify the many "unclassified" changes (e.g., during the period of our study, nearly 60% of the changes were unclassified).

As for the developer-specific versus team-level models, the manager shared our belief that team-level models are more practical. However, he suggested that developer-level models would be more beneficial in cases where new developers join a team for short periods of time (e.g., when interns join the development team). This of course assumes that we have enough history to train the models on.

The manager pointed out that the strength of this work lies in the fact that its findings are simple and easy to understand. A model that is made up of 4-5 factors can be easily understood by managers, so they will know why changes are being flagged. This makes the model much more appealing than a black-box type solution where changes are flagged without any insight as to the rationale. In addition, he pointed out that it would be desirable for the work to also provide a possible course of action to mitigate the risk of a flagged change. For example, a model that flags a risky change might suggest the reduction in risk that can be achieved if unit testing or code reviews were performed on the change.

Table 7.11: Comparison between correctly and incorrectly classified changes

Category		**Incorrectly Classified**	**Correctly Classified**
Cause	Unclear Requirement	11.7%	11.5%
	Side-effect of Other Changes	7.3%	6.4%
	Unclear Documentation	0.7%	1.5%
	Inadequate Testing	0.73%	2.3%
	Scope Change	4.4%	3.2%
	Coding Error	28.5%	37.2%
	Integration Error	2.2%	0.8%
	Design Flaw	10.9%	11.5%
Defect fixing change		70.1%	77.9%
Has related changes		70.8%	7.4%
Modifies API		2.9%	1.5%

7.9 Threats to Validity

Threats to Construct Validity consider the relationship between theory and observation, in case the measured variables do not measure the actual factors.

Changes that introduced defects were manually mapped in this project (i.e., the change that caused a defect was mapped to the change that caused the defect). Although this mapping was done by the project developers themselves, in certain cases, some changes might not have been mapped correctly or not mapped at all.

The risk value used in our study was manually assigned to changes by the developers who made the change. Hence, it is possible that the wrong risk value is assigned. However, our analysis of the percentage of non-risky changes that introduced defects showed that developers are accurate 96.9% of the time. Also, it is important to note that the risk was not assigned by a manager or any other person. The fact that this risk is assigned by the developer who made the change makes it very credible. Furthermore, we are not aware of any other data set that has manually assigned risk values to changes.

When asked to assign the risk to changes, developers assigned risk to 40% of the changes. Our results may be affected by the fact that not all changes were assigned a risk value. However, our response rate of 40% from developers is at least as good as other software engineering studies, which have a response rate in the range of 14 - 33% [37, 231].

During our investigation related to how correct developers are in classifying defect-introducing changes, we looked at how correct developers are when they mark changes as being non-risky. We did not look into how correct developers are when marking a change as risky. There are two reasons for this. First, we are limited by how much we can disclose about risky changes, since they may provide insights about the quality of the commercial system we are using in our case study. Second, changes marked as being risky undergo more scrutiny and might be modified before being integrated into the code base. Hence, the link between risky changes and defect-introducing changes is biased. In contrast, our analysis on changes flagged as non-risky does not exhibit such bias.

Threats to External Validity consider the generalization of our findings. The studied project was a commercial project written mainly in Java and C/C++, therefore, our results may not generalize to other commercial or open source projects.

7.10 Conclusion

Organizations are strongly interested in managing risk which is a considerably more encompassing concept than bugs which has been extensively studied by the software engineering research community. While a risky change might not introduce bug, it might lead to delays and large cost overruns. In this empirical study, the first of its kind, we looked at a unique data set about the risk of software changes to better understand the characteristics of risky changes. The main findings of our study are:

- When studying risky changes, the developer making the change and the team they belong to need to be considered.
- Risky changes can be effectively identified using factors such as the number of lines and chunks added by the changes, the bugginess of the files being changed, the number of bug reports linked to the change and the experience of the developer making the change.
- We find that developers are accurate 96.1% of the time when identifying bug-introducing changes. However, developers' identification of risky changes is less reliable. Especially, when changes have many related changes.

Our study opens a new avenue for Software Engineering research related to risk management within software organizations, and not only bugs, introduced by changes. We plan (and encourage other researchers) to further develop on the findings of this paper. We see many potential avenues for future work related to risk management as a key and important concept in the production of software today.

Furthermore, one of the key lessons that we learned through this study is that practitioners are willing to get involved in research, as long as their commitment is kept to a minimum and the data collection is done in a non-intrusive manner.

Part IV

Conclusion

Chapter 8

Summary, Contributions and Future Work

A plethora of work focused on SDP. However, the adoption of SDP in practice is still limited. We surveyed the state-of-the-art in SDP, and our industrial experience, to explore some of the challenges that hinder the adoption of SDP in practice. We presented a number of approaches that show how SDP research can be tailored to address the challenges of pragmatic SDP. We overview our findings, highlight our contributions and discuss opportunities for future work.

8.1 Summary of Addressed Topics

The main focus of our thesis is to tackle the challenges of pragmatic SDP. First, we conducted a survey of the state-of-the-art in SDP research in order to understand the main challenges. Then, we proposed a number of different approaches to tackle some of the pragmatic challenges current SDP research faces. The remainder of this section details the major topics covered in this thesis.

Chapter 2 surveys the state-of-art in SDP. We performed a review of SDP research from the year 2000 till the year 2011. We believe that such a review is necessary at this time, since a plethora of strong research focused on SDP. Therefore, it is an ideal time to reflect on the field and report the challenges that have been addressed and the challenges that remain open. We found that a large amount of research focused on the models and factors used in SDP, however, very little work focused on applying SDP research in practice.

In **Part I (Chapters 3 and 4)** we present approaches that study and predict high-impacting defects, in particular breakage, surprise and re-opened defects. Chapter 3 presents an approach that focuses on two types of high-impact defects, breakage and surprise defects. We show that these types of defects can be effectively predicted. In addition, we investigate the factors that best indicate breakage and surprise defects. Our findings show that the number of pre-release defects and file size are good indicators of breakages, whereas the number of co-changed files and the amount of time between the latest pre-release change and the release date are good indicators of surprises.

Chapter 4 presents an approach that focused on another type of high-impact defects, re-opened defects, i.e., defects that are more likely to be re-opened after they are addressed. We illustrate how factors extracted from the source code and defect repositories can be used to predict which defects are more likely to be re-opened. We find that the factors that best indicate re-opened defects vary based on the project. For example, the comment text is the most important factor for the Eclipse and OpenOffice projects, while the last status of the defect before it is closed is the most important one for Apache.

In **Part II (Chapters 5 and 6)** we present approaches that show how SDP research can provide guidance on how make use their results. Chapter 5 presents an approach to simplify SDP models in order to make them easier to understand by practitioners. In addition, our approach is able to quantify the impact of the different factors used in SDP models on the likelihood of finding future defects. In a case study on the Eclipse open source project, we show that our approach can reduce the number of factors from 34 to only 4, while achieving comparable performance over the more complex models.

Chapter 6 presents an approach that assists software development and testing managers to use the limited resources they have for testing large software systems efficiently. The approach leverages the development history of a project to generate a prioritized list of functions that managers should focus their unit test writing resources on. Our findings show that factors based on the function size, modification frequency and defect fixing frequency should be used to prioritize the unit test writing efforts for legacy software systems.

In **Part III (Chapter 7)** we argue that SDP approaches need to be more encompassing and proactive. Chapter 7 presents an approach that leverages historical data about software changes in order to predict the risk of a software change. Risk is more encompassing than defects. Risky changes may not even introduce defects but they could delay the release of projects, and/or negatively impact customer satisfaction. Therefore, we build models that predict risky changes so they can be thoroughly reviewed before they are incorporated into the code base. Our findings show that we can effectively predict risky changes and that the number of lines and chunks of code added by the change, the defectiveness of the files being changed, the number of defect reports linked to a change and the developer experience are the best indicators of change risk.

8.2 Contributions

The goal of this thesis was to explore the challenges that hinder the adoption of SDP research in practice and propose approaches to tackle these challenges. We make several contributions towards this goal. These contributions were motivated by our survey of the state-of-the-art in SDP research and our industrial experience. We now highlight the main contributions of the thesis in more detail.

1. **An Extensive Review of the State-of-the-art in SDP**: We performed a review of SDP research in the from the year 2000-2011. Our review included more than 100 papers on SDP. The papers were characterized based on the data, factors, models and performance evaluation methods used. We use this survey to help us better understand what has been covered in the are of SDP (especially those related to the adoption of SDP in practice). In addition, we believe that this survey is useful for the entire research community since it helps identify areas that are open for future research.

2. **Studying and Predicting Breakage and Surprise Defects**: We proposed an approach that predicts and studies breakage and surprise defects. The goal of our approach is to show how SDP research can be tailored to focus on high-impact defects. We showed the number of pre-release defects and file size are good indicators of breakages, whereas the number of co-changed files and the amount of time between the latest pre-release change and the release date are good indicators of surprises. Furthermore, we found that focusing on these defects can reduce the amount of files to be inspected by up to 30%. This chapter contributes towards solving the challenge of taking impact into consideration in SDP research by studying breakage and surprise defects.

3. **Studying and Predicting Re-opened Defects**: We proposed an approach that predicts and studies re-opened defects. We showed that we are able to effectively predict re-opened defects, achieving precision between 49.9-78.3% and a recall in the range of

72.6-93.5%. In addition, we examined the factors that best indicate re-opened defects and found that the factors that best indicate which defects might be re-opened vary based on the project. For example, the comment text is the most important factor for the Eclipse and OpenOffice projects, while the last status of the defect before it is closed is the most important one for Apache. We argue that these factors should be closely examined in order to reduce maintenance cost due to re-opened defects. Similar to the previous chapter, this chapter contributes towards solving the challenge of taking impact into consideration in SDP research by studying re-opened defects.

4. **Simplifying and Understanding SDP Models**: We proposed an approach to simplify prediction models by reducing the number of independent variables (i.e., predictors) they use. Reducing the number of independent variables makes the SDP models easier to understand and use. We showed that our models were able to reduce the number of factors in the SDP models from 34 to just 4, while achieving comparable prediction performance. This chapter simplifies SDP models and contributes towards addressing the challenge of providing guidance on how to make use of SDP results.

5. **Using SDP to Prioritize the Creation of Unit Tests**: We proposed an approach that uses development history in order to guide the prioritization of unit test creation. We showed that a simple SDP model can be used to effectively prioritize the creation of unit tests in large software systems. In particular, we find that factors based on the function size, modification frequency, and defect fixing frequency perform the best for the purpose prioritization of unit test writing efforts. This chapter contributes towards the challenge of providing guidance on how to make use of SDP results by showing how SDP can be used to prioritize the creation of unit tests.

6. **Studying and Predicting the Risk of Software Changes**: We proposed an approach that makes SDP more encompassing and proactive by not only predicting defects, but

focusing on risk. Our approach effectively identifies potentially risky changes so their risk can be mitigated before they are incorporated into the code base. In addition, our study shows the most important indicators of risky changes. This chapter argues that we need to make SDP more encompassing and proactive to increase its chances of adoption in industry. It contributes to the thesis by studying risky changes.

8.3 Future Work

We believe that our thesis makes a positive contribution towards the goal of making SDP research more pragmatic. However, there are still many open challenges that need to be tackled in order to increase the adoption of SDP in practice. We now highlight some avenues for future work.

8.3.1 Formally Investigating Reasons for Lack of SDP Adoption in Practice

In this thesis, we relied on our experience when deciding some of the challenges that hinder the adoption of SDP in practice. The reasons given in this thesis are by no means complete. In the future, we plan to conduct more detailed and formal studies regarding the reasons that hinder the adoption of SDP in practice.

8.3.2 Considering Other Types of High-Impact Defects

In this thesis, we focused on three different types of high-impact software defects. We believe that this is a good start, however, there remains more work to do in this area. Different types of high-impact defects need to be examined. For example, another type of defect that might have a high impact is defects in software artifacts that many other artifacts depend on. In the future, we would like to continue this line of work and study and predict other types of high-impact defects.

8.3.3 Building Tools to Guide Practitioners

Today, most SDP research proposes solutions and empirically evaluates them based on historical data. This type of work has significantly contributed to the research side of software engineering, however, very little work actually builds tools based on their research to advance and applicability of SDP research in practice. In the future, we plan to focus more on how to build tools so our research can be easier to incorporate in industry.

8.3.4 More Realistic Evaluations

As shown in our survey in Chapter 2, the vast majority of SDP studies evaluate their approaches using the precision and recall measures. However, as pointed out by other researchers [173] and from our own industrial experience, standard statistical measures of performance such as precision and recall might not be the best way to evaluate the practical value of SDP approaches. Whenever possible, we strived to obtain feedback from practitioners about our proposed approaches. In the future, we plan to investigate and propose evaluation criteria that practitioners use to measure the value of SDP approaches. We believe that using such criteria will provide a more realistic evaluation of SDP approaches and significantly improve the adoption of SDP in practice.

8.3.5 Examining Replicability

The majority of SDP research heavily depends on historical development data. Till now, the availability of open source data has been relatively easy, however, acquiring commercial data is still a challenge. The fact that commercial data is not widely available makes it difficult to examine the repeatability of SDP studies. Examining repeatability is important since it indicates how generalizable the finding are. We believe that the entire software engineering research community needs to address this issue of making data (especially commercial data) available in order to facilitate the repeatability of proposed approaches.

Bibliography

[1] Bugzilla. http://www.bugzilla.org/.

[2] Coverity. http://www.coverity.com/.

[3] The economic impacts of inadequate infrastructure for software testing. http://www.nist.gov/director/planning/upload/report02-3.pdf.

[4] The international conference on predictor models in software engineering. http://promisedata.org/.

[5] The r project. http://www.r-project.org/.

[6] Statistical analysis with sas/stat software. http://www.sas.com/technologies/analytics/statistics/stat/.

[7] The working conference on mining software repositorie. http://www.msrconf.org/.

[8] Second international workshop on managing technical debt, 2011.

[9] Roberto Abreu and Rahul Premraj. How developer communication frequency relates to bug introducing changes. In *Proceedings of the joint International and annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, IWPSE-Evol '09, pages 153–158, 2009.

[10] K. K. Aggrawal, Yogesh Singh, and A. Kaur. Code coverage based technique for prioritizing test cases for regression testing. *SIGSOFT Software Engineering Notes*, 29(5), 2004.

[11] Fumio Akiyama. An example of software system debugging. In *IFIP Congress (1)*, pages 353–359, 1971.

[12] Sousuke Amasaki, Yasunari Takagi, Osamu Mizuno, and Tohru Kikuno. A bayesian belief network for assessing the likelihood of fault content. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE ’03, pages 215–226, 2003.

[13] Prasanth Anbalagan and Mladen Vouk. ”days of the week” effect in predicting the time taken to fix defects. In *DEFECTS ’09: Proceedings of the 2nd International Workshop on Defects in Large Software Systems*, pages 29–30, 2009.

[14] Ion Androutsopoulos, John Koutsias, Konstantinos V. Chandrinos, and Constantine D. Spyropoulos. An experimental comparison of naive bayesian and keyword-based anti-spam filtering with personal e-mail messages. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR ’00, pages 160–167, 2000.

[15] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON ’08, pages 23:304–23:318, 2008.

[16] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 361–370, 2006.

[17] John Anvik and Gail C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology*, 20:10:1–10:35, August 2011.

[18] Jorge Aranda and Gina Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 298–308, 2009.

[19] Erik Arisholm and Lionel C. Briand. Predicting fault-prone components in a java legacy system. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 8–17, 2006.

[20] Erik Arisholm, Lionel C. Briand, and Audun Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30:491–506, August 2004.

[21] Erik Arisholm, Lionel C. Briand, and Magnus Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *Proc. International Symposium on Software Reliability (ISSRE'07)*, 2007.

[22] Erik Arisholm, Lionel C. Briand, and Eivind B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[23] Lerina Aversano, Luigi Cerulo, and Concettina Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Ninth International Workshop on Principles of Software Evolution*, IWPSE '07, pages 19–26, 2007.

[24] A. Bachmann and A. Bernstein. When process data quality affects the number of bugs: Correlations in software engineering datasets. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 62 –71, may 2010.

[25] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: bugs and bug-fix commits. In *Proceedings of the eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 97–106, 2010.

[26] C. G. Bai, Q. P. Hu, M. Xie, and S. H. Ng. Software failure prediction based on a markov bayesian network model. *Journal of Systems and Software*, 74:275–282, February 2005.

[27] B. Balzer, M. Litoiu, H. Muller, D. Smith, M. Storey, S. Tilley, and K. Wong. 4th international workshop on adoption-centric software engineering. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 748 – 749, May 2004.

[28] R. Balzer, J. Jahnke, M. Litoiu, H.A. Muller, D.B. Smith, M.A. Storey, S.R. Tilley, and K. Wong. 3rd international workshop on adoption-centric software engineering acse 2003. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 789 – 790, May 2003.

[29] Ricardo Barandela, J. Salvador Sánchez, Vicente García, and Edgar Rangel. Strategies for learning in class imbalance problems. *Pattern Recognition*, 36(3):849–851, 2003.

[30] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[31] K. Beck and M. Fowler. *Planning extreme programming*. Addison-Wiley, 2001.

[32] Robert M. Bell, Thomas J. Ostrand, and Elaine J. Weyuker. Looking for bugs in all the right places. In *Proc. International Symposium on Software Testing and Analysis (ISSTA'06)*, ISSTA '06, pages 61–72, 2006.

[33] Keith Bennett. Legacy systems: Coping with success. *IEEE Software*, 12(1), 1995.

[34] Abraham Bernstein, Jayalath Ekanayake, and Martin Pinzger. Improving defect prediction using temporal features and non linear models. In *Ninth International Workshop on Principles of Software Evolution*, IWPSE '07, pages 11–18, 2007.

[35] N. Bettenburg, R. Premraj, T. Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful...really? In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 337 –345, October 2008.

[36] Nicolas Bettenburg and Ahmed E. Hassan. Studying the impact of social structures on software quality. In *Proc. International Conference on Program Comprehension (ICPC'10)*, pages 124–133, 2010.

[37] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 308–318, 2008.

[38] Pamela Bhattacharya and Iulian Neamtiu. Assessing programming language impact on development and maintenance: a study on c and c++. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 171–180, 2011.

[39] S. Biffl. Evaluating defect estimation models with major defects. *Journal of Systems and Software*, 65:13–29, January 2003.

[40] David Binkley, Henry Feild, Dawn Lawrie, and Maurizio Pighin. Increasing diversity: Natural language measures for software fault prediction. *Journal of Systems and Software*, 82:1793–1803, November 2009.

[41] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: bias in bug-fix datasets. 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 121–130, 2009.

[42] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009.

[43] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *Proceedings of the 2009 20th International Symposium on Software Reliability Engineering*, ISSRE '09, pages 109–119, 2009.

[44] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 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, 2011.

[45] Jesús Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. Legacy information systems: Issues and directions. *IEEE Software*, 16(5), 1999.

[46] Barry W. Boehm. Software risk management: Principles and practices. *IEEE Software*, 8(1):32–41, January 1991.

[47] Gary D. Boetticher. Nearest neighbor sampling for better defect prediction. In *Proceedings of the 2005 workshop on Predictor Models in Software Engineering*, PROMISE '05, pages 1–6, 2005.

[48] L.C. Briand, K. El Emam, B.G. Freimut, and O. Laitenberger. A comprehensive evaluation of capture-recapture models for estimating software defect content. *IEEE Transactions on Software Engineering*, 26(6):518 –540, jun 2000.

[49] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[50] Lionel C. Briand, Walcelio L. Melo, and Jurgen Wust. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Transactions on Software Engineering*, 28:706–720, July 2002.

[51] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, 51(3):245–273, 2000.

[52] Gul Calikli and Ayse Bener. Preliminary analysis of the effects of confirmation bias on software defect density. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 68:1–68:1, 2010.

[53] M. Cartwright and M. Shepperd. An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering*, 26(8):786 –796, aug 2000.

[54] Cagatay Catal and Banu Diri. Review: A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36:7346–7354, May 2009.

[55] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 99(6):864–878, 2009.

[56] Marcelo Cataldo and Sangeeth Nambiar. On the relationship between process maturity and geographic distribution: an empirical analysis of their impact on software quality. 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 101–110, 2009.

[57] Luigi Cerulo. On the use of process trails to understand software development. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 303–304, 2006.

[58] Philip Chan and Salvatore J. Stolfo. Toward scalable learning with non-uniform class and cost distributions: A case study in credit card fraud detection. In *In Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, pages 164–168, 1998.

[59] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[60] Jacek Czerwonka, Rajiv Das, Nachiappan Nagappan, Alex Tarvo, and Alex Teterev. Crane: Failure prediction, change analysis and test prioritization in practice – experiences from windows. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 357–366, 2011.

[61] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Proc. International Working Conference on Mining Software Repositories*, pages 31–41, May 2010.

[62] Marco D'Ambros, Michele Lanza, and Romain Robbes. On the relationship between change coupling and software defects. In *Proc. Working Conference on Reverse Engineering (WCRE'09)*, pages 135–144, 2009.

[63] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, pages 1–47, 2011.

[64] Vidroha Debroy and W. Eric Wong. On the estimation of adequate test set size using fault failure rates. *Journal of Systems and Software*, 84:587–602, April 2011.

[65] F. Michael Dedolph. The neglected management activity: Software risk management. *Bell Labs Technical Journal*, 8(3):91–95, 2003.

[66] Giovanni Denaro and Mauro Pezzè. An empirical evaluation of fault-proneness models. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 241–251, 2002.

[67] Michael Diaz and Joseph Sligo. How software process improvement helped motorola. *IEEE Software*, 14:75–81, September 1997.

[68] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Empirical studies of test case prioritization in a junit testing environment. In *Proc. International Symposium on Software Reliability (ISSRE'04)*, 2004.

[69] Bradley Efron. Estimating the error rate of a prediction rule: Improvement on Cross-Validation. *Journal of the American Statistical Association*, 78(382):316–331, 1983.

[70] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27:1–12, January 2001.

[71] Jayalath Ekanayake, Jonas Tappolet, Harald C. Gall, and Abraham Bernstein. Tracking concept drift of software projects using defect prediction quality. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 51–60, 2009.

[72] Sebastian Elbaum, Satya Kanduri, and Anneliese Andrews. Trace anomalies as precursors of field failures: an empirical study. *Empirical Software Engineering*, 12:447–469, October 2007.

[73] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proc. International Conference on Software Engineering (ICSE'01)*, 2001.

[74] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *Proc. International Symposium on Software Testing and Analysis (ISSTA'00)*, 2000.

[75] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2), 2002.

[76] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Control*, 12(3), 2004.

[77] Karim O. Elish and Mahmoud O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81:649–660, May 2008.

[78] Khaled El Emam, Walcelio Melo, and Javam C. Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56:63–75, February 2001.

[79] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3), 2005.

[80] Ana Erika and Camargo Cruz. Exploratory study of a uml metric for fault prediction. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 361–364, 2010.

[81] Ana Erika, Camargo Cruz, and Koichiro Ochimizu. Towards logistic regression models for predicting fault-prone code across software projects. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 460–463, 2009.

[82] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[83] Andrew Estabrooks and Nathalie Japkowicz. A mixture-of-experts framework for learning from imbalanced data sets. In *IDA '01: Proceedings of the 4th International Conference on Advances in Intelligent Data Analysis*, pages 34–43, 2001.

[84] William M. Evanco. Prediction models for software fault correction effort. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, CSMR '01, pages 114–120, 2001.

[85] Jon Eyolfson, Lin Tan, and Patrick Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 153–162, 2011.

[86] J.-M. Favre, J. Estublier, and R. Sanlaville. Tool adoption issues in a very large software company. *CMU/SEI-2003-SR-004*, 2003.

[87] Norman Fenton, Martin Neil, William Marsh, Peter Hearty, Lukasz Radlinski, and Paul Krause. Project data incorporating qualitative factors for improved software defect prediction. In *Proceedings of the 29th International Conference on Software Engineering Workshops*, pages 69–78, 2007.

[88] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25:675–689, September 1999.

[89] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26:797–814, August 2000.

[90] A. Ferdinand. A Theory of System Complexity. *International Journal of General Systems*, 1:19–33, 1974.

[91] Fabiano Ferrari, Rachel Burrows, Otávio Lemos, Alessandro Garcia, Eduardo Figueiredo, Nelio Cacho, Frederico Lopes, Nathalia Temudo, Liana Silva, Sergio Soares, Awais Rashid, Paulo Masiero, Thais Batista, and José Maldonado. An exploratory study of fault-proneness in evolving aspect-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 65–74, 2010.

[92] Javed Ferzund, Syed Nadeem Ahsan, and Franz Wotawa. Software change classification using hunk metrics. In *International Conference on Software Maintenance*, pages 471–474, 2009.

[93] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 23–32, 2003.

[94] Bernd Freimut, Susanne Hartkopf, Peter Kaiser, Jyrki Kontio, and Werner Kobitzsch. An industrial case study of implementing software risk management. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 277–287, 2001.

[95] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting, 1995.

[96] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proc. International Conference on Software Maintenance (ICSM'98)*, pages 190–198, 1998.

[97] Kehan Gao, Taghi M. Khoshgoftaar, Huanjing Wang, and Naeem Seliya. Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Software Practice and Experience*, 41:579–606, April 2011.

[98] Boby George and Laurie Williams. An initial investigation of test driven development in industry. In *SAC '03: Proceedings of the 2003 ACM Symposium on Applied computing*, 2003.

[99] Boby George and Laurie Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5), 2003.

[100] Emanuel Giger, Martin Pinzger, and Harald Gall. Using the gini coefficient for bug prediction in eclipse. In *Proceedings of the joint International and annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (EVOL) Workshops*, pages 51–55, 2011.

[101] Emanuel Giger, Martin Pinzger, and Harald C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 83–92, 2011.

[102] Michael W. Godfrey, Ahmed E. Hassan, James Herbsleb, Gail C. Murphy, Martin Robillard, Prem Devanbu, Audris Mockus, Dewayne E. Perry, and David Notkin. Future of mining software archives: A roundtable. *IEEE Software*, 26(1):67 –70, Jan.-Feb. 2009.

[103] Iker Gondra. Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software*, 81:186–195, February 2008.

[104] P. Graham. A plan for spam, 2002.

[105] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, July 2000.

[106] L. Guo, B. Cukic, and H. Singh. Predicting fault prone modules by the dempster-shafer belief networks. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 249 – 252, 2003.

[107] Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. Robust prediction of fault-proneness by random forests. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 417–428, 2004.

[108] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In *Proceedings of the 32th International Conference on Software Engineering*, May 2010.

[109] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31:897–910, October 2005.

[110] S.W. Haider, J.W. Cangussu, K.M.L. Cooper, and R. Dantu. Estimation of defects based on defect decay model: Ed^{3}m. *IEEE Transactions on Software Engineering*, 34(3):336 –356, may-june 2008.

[111] Joseph F Hair, Jr., Rolph E Anderson, and Ronald L Tatham. *Multivariate data analysis with readings (2nd ed.).* Macmillan Publishing Co., Inc., 1986.

[112] M A Hall and L A Smith. Practical feature subset selection for machine learning. *Computer Science*, 98:181–191, 1998.

[113] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11(1), 2009.

[114] Elliotte Rusty Harold. Testing legacy code. http://www.ibm.com/developerworks/java/library/j-legacytest.html.

[115] A.E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008*, pages 48 –57, Oct. 2008.

[116] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, 2009.

[117] Ahmed E. Hassan and Richard C. Holt. The top ten list: Dynamic fault prediction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 263–272, 2005.

[118] Ahmed E. Hassan and Ken Zhang. Using decision trees to predict the certification result of a build. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 189–198, 2006.

[119] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Fault-prone module detection using large-scale text features based on spam filtering. *Empirical Software Engineering*, 15:147–165, April 2010.

[120] I. Herraiz, J.M. Gonzalez-Barahona, G. Robles, and D.M. German. On the prediction of the evolution of libre software projects. In *International Conference on Software Maintenance, 2007*, pages 405 –414, oct. 2007.

[121] Israel Herraiz, Daniel M. German, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Towards a simplification of the bug report form in eclipse. In *MSR '08: Proceedings of the 2008 International Working Conference on Mining Software Repositories*, pages 145–148, 2008.

[122] Israel Herraiz, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Towards a theoretical model for software growth. In *Proc. International Workshop on Mining Software Repositories (MSR'07)*, pages 21–28, 2007.

[123] Rattikorn Hewett and Phongphun Kijsanayothin. On modeling software defect repair time. *Empirical Software Engineering*, 14(2):165–186, 2009.

[124] Tilman Holschuh, Markus P?user, Kim Herzig, Thomas Zimmermann, Premraj. Rahul, and Andreas Zeller. Predicting defects in sap java code: An experience report. In *International Conference on Software Engineering*, pages 172–181, 2009.

[125] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *ASE '07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 34–43, 2007.

[126] W. M. Ibrahim, N. Bettenburg, E. Shihab, B. Adams, and A. E. Hassan. Should i contribute to this discussion? In *MSR '10: Proceedings of the 2010 International Working Conference on Mining Software Repositories*, 2010.

[127] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The foundations of Software Engineering*, pages 111–120, 2009.

[128] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computer Surveys*, 41:21:1–21:54, October 2009.

[129] Hao Jia, Fengdi Shu, Ye Yang, and Qi Li. Data transformation and attribute subset selection: Do they help make differences in software failure prediction? In *International Conference on Software Maintenance*, pages 519–522, 2009.

[130] Letian Jiang and Guozhi Xu. Modeling and analysis of software aging and software failure. *Journal of Systems and Software*, 80:590–595, April 2007.

[131] Yue Jiang, Bojan Cuki, Tim Menzies, and Nick Bartlow. Comparing design and code metrics for software quality prediction. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, PROMISE '08, pages 11–18, 2008.

[132] Yue Jiang, Bojan Cukic, and Yan Ma. Techniques for evaluating fault prediction models. *Empirical Software Engineering*, 13:561–595, October 2008.

[133] Yue Jiang, Bojan Cukic, and Tim Menzies. Cost curve evaluation of fault prediction models. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 197–206, 2008.

[134] Yue Jiang, Jie Lin, Bojan Cukic, and Tim Menzies. Variance analysis in software fault prediction models. In *Proceedings of the 20th IEEE International Conference on Software Reliability Engineering*, ISSRE'09, pages 99–108, 2009.

[135] I. T. Jolliffe. *Principal Component Analysis*. Springer, second edition, 2002.

[136] Hemant Joshi, Chuanlei Zhang, S. Ramaswamy, and Coskun Bayrak. Local and global recency weighting approach to bug prediction. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 33–34, 2007.

[137] Ho-Won Jung, YiKyong Lim, and Chang-Shin Chung. Modeling change requests due to faults in a large-scale telecommunication system. *Journal of Systems and Software*, 72(2):235–247, 2004.

[138] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E. Hassan. Revisiting common bug prediction findings using effort aware models. In *Proc. International Conference on Software Maintenance (ICSM'10)*, pages 1–10, 2010.

[139] Yasutaka Kamei, Akito Monden, Shinsuke Matsumoto, Takeshi Kakimoto, and Kenichi Matsumoto. The effects of over and under sampling on fault-prone module detection. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 196–204, 2007.

[140] Yasutaka Kamei, Akito Monden, Shuji Morisaki, and Ken-ichi Matsumoto. A hybrid faulty module prediction using association rule mining and logistic regression analysis. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 279–281, 2008.

[141] Marouane Kessentini, Stéphane Vaucher, and Houari Sahraoui. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In *Proceedings of the IEEE/ACM international conference on Automated Software Engineering*, ASE '10, pages 113–122, 2010.

[142] Taghi Khoshgoftaar, Kehan Gao, and Robert M. Szabo. An application of zero-inflated poisson regression for software fault prediction. In *International Symposium on Software Reliability Engineering*, ISSRE '01, pages 66–73, 2001.

[143] Taghi M. Khoshgoftaar, Edward B. Allen, Wendell D. Jones, and John P. Hudepohl. Data mining for predictors of software quality. *International Journal of Software Engineering and Knowledge Engineering*, 9(5):547–564, 1999.

[144] Taghi M. Khoshgoftaar and Naeem Seliya. Fault prediction modeling for software quality estimation: Comparing commonly used techniques. *Empirical Software Engineering*, 8:255–283, September 2003.

[145] Taghi M. Khoshgoftaar and Naeem Seliya. Comparative assessment of software quality classification techniques: An empirical case study. *Empirical Software Engineering*, 9:229–257, September 2004.

[146] Taghi M. Khoshgoftaar, Xiaojing Yuan, Edward B. Allen, Wendell D. Jones, and John P. Hudepohl. Uncertain classification of fault-prone software modules. *Empirical Software Engineering*, 7:297–318, December 2002.

[147] T.M. Khoshgoftaar, Ruqun Shan, and E.B. Allen. Improving tree-based models of software quality with principal components analysis. In *International Symposium on Software Reliability Engineering*, pages 198 –209, 2000.

[148] T.M. Khoshgoftaar, V. Thaker, and E.B. Allen. Modeling fault-prone modules of subsystems. In *International Symposium on Software Reliability Engineering*, pages 259 –267, 2000.

[149] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 119–129, 2002.

[150] Sunghun Kim and Michael D. Ernst. Which warnings should i fix first? In *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 45–54, 2007.

[151] Sunghun Kim, Kai Pan, and E. E. James Whitehead, Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 35–45, 2006.

[152] Sunghun Kim and E. James Whitehead, Jr. How long did it take to fix bugs? In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 173–174, 2006.

[153] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34:181–196, March 2008.

[154] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. Dealing with noise in defect prediction. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 481–490, 2011.

[155] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 489–498, 2007.

[156] Michael Kläs, Frank Elberzhager, Jürgen Münch, Klaus Hartjes, and Olaf von Graevemeyer. Transparent combination of expert and measurement data for defect prediction: an industrial case study. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 119–128, 2010.

[157] Michael Kläs, Frank Elberzhager, and Haruka Nakao. Managing software quality through a hybrid defect content and effectiveness model. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 321–323, 2008.

[158] Michael Kläs, Haruka Nakao, Frank Elberzhager, and Jürgen Münch. Predicting defect content and quality assurance effectiveness by combining expert judgment and defect data - a case study. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 17–26, 2008.

[159] Patrick Knab, Martin Pinzger, and Abraham Bernstein. Predicting defect densities in source code files with decision tree learners. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 119–125, 2006.

[160] A. Güneş Koru, Khaled El Emam, Dongsong Zhang, Hongfang Liu, and Divya Mathew. Theory of relative defect proneness. *Empirical Software Engineering*, 13:473–498, October 2008.

[161] A. Güneş Koru and Jeff Tian. An empirical comparison and characterization of high defect and high complexity modules. *Journal of Systems Software*, 67:153–163, September 2003.

[162] A. Gunes Koru and Hongfang Liu. Building defect prediction models in practice. *IEEE Software*, 22:23–29, November 2005.

[163] A. Günes Koru and Hongfang Liu. An investigation of the effect of module size on defect prediction using static measures. In *Proceedings of the 2005 Workshop on Predictor Models in Software Engineering*, PROMISE '05, pages 1–5, 2005.

[164] A. Gunes Koru, Dongsong Zhang, and Hongfang Liu. Modeling the effect of size on defect proneness for open-source software. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 10–19, 2007.

[165] Heiko Koziolek, Bastian Schlich, Carlos Bilich, Roland Weiss, Steffen Becker, Klaus Krogmann, Mircea Trifu, Raffaela Mirandola, and Anne Koziolek. An industrial case study on quality impact prediction for evolving service-oriented software. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 776–785, 2011.

[166] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Design evolution metrics for defect prediction in object oriented systems. *Empirical Software Engineering*, 16:141–175, February 2011.

[167] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 1 –10, may 2010.

[168] Lucas Layman, Gunnar Kudrjavets, and Nachiappan Nagappan. Iterative identification of fault-prone binaries using in-process metrics. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 206–212, 2008.

[169] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 311–321, 2011.

[170] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34:485–496, July 2008.

[171] Marek Leszak, Dewayne E. Perry, and Dieter Stoll. Classification and evaluation of defects in a project retrospective. *Journal of Systems and Software*, 61:173–187, April 2002.

[172] T.C. Lethbridge. Value assessment by potential tool adopters: towards a model that considers costs, benefits and risks of adoption. *IEE Seminar Digests*, 2004(906):46–50, 2004.

[173] Paul Luo Li, James Herbsleb, Mary Shaw, and Brian Robinson. Experiences and results from initiating field defect prediction and product test prioritization efforts at abb inc. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 413–422, 2006.

[174] Paul Luo Li, Jim Herbsleb, and Mary Shaw. Forecasting field defect rates using a combined time-based and metrics-based approach: A case study of openbsd. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 193–202, 2005.

[175] Paul Luo Li, Mary Shaw, Jim Herbsleb, Bonnie Ray, and P. Santhanam. Empirical evaluation of defect projection models for widely-deployed production software systems. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 263–272, 2004.

[176] Yi Liu, Taghi M. Khoshgoftaar, and Naeem Seliya. Evolutionary optimization of software quality modeling with multiple repositories. *IEEE Transactions on Software Engineering*, 36:852–864, November 2010.

[177] Jung-Hua Lo and Chin-Yu Huang. An integration of fault detection and correction processes in software reliability analysis. *Journal of Systems and Software*, 79:1312–1323, September 2006.

[178] Yan Ma and Bojan Cukic. Adequate and precise evaluation of quality models in software engineering studies. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 1–9, 2007.

[179] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287 –300, march-april 2008.

[180] Lionel Marks, Ying Zou, and Ahmed E. Hassan. Studying the fix-time for bugs in large open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, Promise '11, pages 11:1–11:8, 2011.

[181] Thomas J. McCabe. A complexity measure. In *Proc. International Conference on Software Engineering (ICSE'76)*, page 407, 1976.

[182] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec. 1976.

[183] Thilo Mende and Rainer Koschke. Revisiting the evaluation of defect prediction models. In *Proc. International Conference on Predictor Models in Software Engineering (PROMISE'09)*, pages 1–10, 2009.

[184] Thilo Mende and Rainer Koschke. Effort-aware defect prediction models. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 107–116, 2010.

[185] Thilo Mende, Rainer Koschke, and Marek Leszak. Evaluating defect prediction models for a large evolving software system. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pages 247–250, 2009.

[186] Thilo Mende, Rainer Koschke, and Jan Peleska. On the utility of a defect prediction model during hw/sw integration testing: A retrospective case study. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 259–268, 2011.

[187] Andrew Meneely, Pete Rotella, and Laurie Williams. Does adding manpower also affect quality?: an empirical, longitudinal analysis. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 81–90, 2011.

[188] Andrew Meneely and Laurie Williams. Strengthening the empirical analysis of the relationship between linus' law and software security. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 9:1–9:10, 2010.

[189] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 13–23, 2008.

[190] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33:2–13, January 2007.

[191] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17:375–407, December 2010.

[192] Tim Menzies, Burak Turhan, Ayse Bener, Gregory Gay, Bojan Cukic, and Yue Jiang. 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, 2008.

[193] T. A. Meyer and B. Whateley. SpamBayes: Effective open-source, Bayesian based, email classification system. *Proceedings of the First Conference on Email and Anti-Spam*, 2004.

[194] J.A. Miccolis, K. Hively, B.W. Merkley, Tillinghast-Towers Perrin, and Institute of Internal Auditors Research Foundation. *Enterprise Risk Management: Trends and Emerging Practices*. Institute of Internal Auditors Research Foundation, 2001.

[195] Osamu Mizuno, Shiro Ikami, Shuya Nakaichi, and Tohru Kikuno. Spam filter based approach for finding fault-prone software modules. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 4–7, 2007.

[196] J. Moad. Maintaining the competitive edge. *Datamation*, 64(66):61–62, 1990.

[197] Audris Mockus. Organizational volatility and its effects on software defects. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 117–126, 2010.

[198] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[199] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, 2000.

[200] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[201] Audris Mockus, David M. Weiss, and Ping Zhang. Understanding and predicting effort in software projects. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 274–284, 2003.

[202] Audris Mockus, Ping Zhang, and Paul Luo Li. Predictors of customer perceived software quality. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 225–233, 2005.

[203] Sandro Morasca and Günther Ruhe. A hybrid approach to analyze empirical software engineering data and its application to predict module fault-proneness in maintenance. *Journal of Systems and Software*, 53:225–237, September 2000.

[204] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 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, 2008.

[205] Harvey Motulsky. Multicollinearity in multiple regression. Online: http://www.graphpad.com/articles/Multicollinearity.htm, Accessed March 2010.

[206] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. Understanding software systems using reverse engineering technology perspectives from the rigi project. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '93, pages 217–226, 1993.

[207] Matthias M. Müller and Walter F. Tichy. Case study: extreme programming in a university environment. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, 2001.

[208] J. Munson and Y. M. Khoshgoftaar. Regression modelling of software quality: empirical investigation. *Journal of Electronic Materials*, 19:106–114, June 1990.

[209] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *International Conference on Software Engineering(ICSE'05)*, pages 580–586, 2005.

[210] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *International Conference on Software Engineering (ICSE'05)*, pages 284–292, 2005.

[211] Nachiappan Nagappan and Thomas Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 364–373, 2007.

[212] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *International Conference on Software Engineering (ICSE'06)*, pages 452–461, 2006.

[213] Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering*, 13(3), 2008.

[214] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 521–530, 2008.

[215] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change bursts as defect predictors. In *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering*, November 2010.

[216] Ann Marie Neufelder. How to measure the impact of specific development practices on fielded defect density. In *International Symposium on Software Reliability Engineering*, ISSRE '00, pages 148–159, 2000.

[217] Thanh H. D. Nguyen, Bram Adams, and Ahmed E. Hassan. A case study of bias in bug-fix datasets. In *Proceedings of the 17th Working Conference on Reverse Engineering*, WCRE '10, pages 259–268, 2010.

[218] Thanh H. D. Nguyen, Bram Adams, and Ahmed E. Hassan. Studying the impact of dependency network measures on software quality. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, 2010.

[219] Tung Thanh Nguyen, Tien N. Nguyen, and Tu Minh Phuong. Topic-based defect prediction (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 932–935, 2011.

[220] Allen P. Nikora and John C. Munson. Building high-quality software fault predictors: Papers from compsac 2004. *Software Practice and Experience*, 36:949–969, July 2006.

[221] A. Nugroho, M.R.V. Chaudron, and E. Arisholm. Assessing uml design metrics for predicting fault-prone classes in a java system. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 21 –30, may 2010.

[222] Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, 1996.

[223] Hector M. Olague, Letha H. Etzkorn, Sampson Gholston, and Stephen Quattlebaum. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on Software Engineering*, 33:402–419, June 2007.

[224] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 55–64, 2002.

[225] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 86–96, 2004.

[226] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[227] Kai Pan, Sunghun Kim, and E. James Whitehead, Jr. Bug classification using program slicing metrics. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 31–42, 2006.

[228] Lucas D. Panjer. Predicting eclipse bug lifetimes. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, pages 29–32, 2007.

[229] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 447–456, 2010.

[230] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 2–12, 2008.

[231] Teade Punter, Marcus Ciolkowski, Bernd Freimut, and Isabel John. Conducting on-line surveys in software engineering. In *Proceedings of the 2003 International Symposium on Empirical Software Engineering*, ISESE '03, pages 80–88, 2003.

[232] Tong-Seng Quah and Mie Mie Thet Thwin. Application of neural networks for software quality prediction using object-oriented metrics. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 116–126, 2003.

[233] J. Ross Quinlan. *C4.5: Programs for machine learning*. Morgan Kaufmann Publishers Inc., 1993.

[234] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2009.

[235] Keld Raaschou and Austen W. Rainer. Exposure model for prediction of number of customer reported defects. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 306–308, 2008.

[236] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 491–500, 2011.

[237] Rudolf Ramler, Claus Klammer, and Thomas Natschläger. The usual suspects: a case study on delivered defects per developer. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 48:1–48:4, 2010.

[238] Jacek Ratzinger, Harald Gall, and Martin Pinzger. Quality assessment based on attribute series of software evolution. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 80–89, 2007.

[239] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the relation of refactorings and software defect prediction. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 35–38, 2008.

[240] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. Open source software peer review practices: A case study of the apache server. In *ICSE '08: Proceedings of the 30th International Conference on Software engineering*, pages 541–550, 2008.

[241] Brian Robinson and Patrick Francis. Improving industrial adoption of software engineering research: a comparison of open and closed source software. In *Proceedings*

of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '10, pages 21:1–21:10, 2010.

[242] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4), 1975.

[243] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10), 2001.

[244] Per Runeson. A survey of unit testing practices. *IEEE Software*, 23(4), 2006.

[245] J. S. Sánchez, R. Barandela, A. I. Marqués, and R. Alejo. Performance evaluation of prototype selection algorithms for nearest neighbor classification. In *SIBGRAPI '01: Proceedings of the XIV Brazilian Symposium on Computer Graphics and Image Processing*, page 44, 2001.

[246] Jelber Sayyad and C. Lethbridge. Supporting software maintenance by mining software update records. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance*, page 22, 2001.

[247] Norm Schneidewind and Mike Hinchey. A complexity reliability model. In *Proceedings of the 2009 20th International Symposium on Software Reliability Engineering*, ISSRE '09, pages 1–10, 2009.

[248] Norman F. Schneidewind. Modelling the fault correction process. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, ISSRE '01, pages 185–190, 2001.

[249] Adrian Schröter, Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. If your bug database could talk... In *ISESE '06: Proceedings of the 5th International Symposium on Empirical Software Engineering. Volume II: Short Papers and Posters*, pages 18–20, 2006.

[250] Hanna Scott and Philip M. Johnson. Generalizing fault contents from a few classes. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 205–214, 2007.

[251] Weiyi Shang, Zhen Ming Jiang, Bram Adams, and Ahmed E. Hassan. Mapreduce as a general framework to support research in mining software repositories (MSR). In *MSR '09: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 10, 2009.

[252] Martin Shepperd, Michelle Cartwright, and Gada Kadoda. On building prediction systems for software engineers. *Empirical Software Engineering*, 5:175–182, November 2000.

[253] Emad Shihab. Pragmatic prioritization of software quality assurance efforts. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1106–1109, 2011.

[254] Emad Shihab, Zhen Ming Jiang, Walid M. Ibrahim, Bram Adams, and Ahmed E. Hassan. Understanding the impact of code and process metrics on post-release defects: A case study on the Eclipse project. In *Proc. International Symposium on Empirical Software Engineering and Measurement (ESEM'10)*, pages 1–10, 2010.

[255] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. High-impact defects: a study of breakage and surprise defects. In *Proceedings of the*

19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, pages 300–310, 2011.

[256] Y. Shin, A. Meneely, L. Williams, and J. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, PP(99):1, 2010.

[257] Yonghee Shin, Robert Bell, Thomas Ostrand, and Elaine Weyuker. Does calling structure information improve the accuracy of fault prediction? In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 61–70, 2009.

[258] Shivkumar Shivaji, E. James Whitehead Jr., Ram Akella, and Sunghun Kim. Reducing features to improve bug prediction. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 600–604, 2009.

[259] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, 2005.

[260] Will Snipes, Brian Robinson, and Penelope Brooks. Approximating deployment metrics to predict field defects and plan corrective maintenance activities. In *Proceedings of the 2009 20th International Symposium on Software Reliability Engineering*, ISSRE '09, pages 90–98, 2009.

[261] Qinbao Song, Zihan Jia, Martin Shepperd, Shi Ying, and Jin Liu. A general software defect-proneness prediction framework. *IEEE Transactions on Software Engineering*, 37:356–370, May 2011.

[262] Qinbao Song, Martin Shepperd, Michelle Cartwright, and Carolyn Mair. Software defect association mining and defect correction effort prediction. *IEEE Transactions on Software Engineering*, 32:69–82, February 2006.

[263] Ramanath Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.

[264] Giancarlo Succi, Witold Pedrycz, Milorad Stefanovic, and James Miller. Practical assessment of the models for identification of defect-prone classes in object-oriented commercial systems using design metrics. *Journal of Systems and Software*, 65:1–12, January 2003.

[265] Hall T, Beecham S, Bowes D, Gray D, and Counsell S. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, To appear, 2011.

[266] Alexander Tarvo. Using statistical models to predict software regressions. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 259–264, 2008.

[267] W. F. Tichy. RCS - a system for version control. *Software: Practice and Experience*, 15(7):637–654, 1985.

[268] S. Tilley, H. Muller, L. O'Brien, and K. Wong. Report from the second international workshop on adoption-centric software engineering (acse 2002). In *Software Technology and Engineering Practice, 2002. STEP 2002. Proceedings. 10th International Workshop on*, pages 74 – 78, Oct. 2002.

[269] Piotr Tomaszewski and Lars-Ola Damm. Comparing the fault-proneness of new and modified code: an industrial case study. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 2–7, 2006.

[270] Piotr Tomaszewski, Hakan Grahn, and Lars Lundberg. A method for an accurate early prediction of faults in modified classes. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 487–496, 2006.

[271] Piotr Tomaszewski, Jim Håkansson, Håkan Grahn, and Lars Lundberg. Statistical models vs. expert estimation for fault prediction in modified code - an industrial case study. *Journal of Systems and Software*, 80:1227–1238, August 2007.

[272] Ayse Tosun and Ayse Bener. Reducing false alarms in software defect prediction by decision threshold optimization. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 477–480, 2009.

[273] Ayse Tosun, Burak Turhan, and Ayse Bener. Ensemble of software defect predictors: a case study. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 318–320, 2008.

[274] Burak Turhan, Tim Menzies, Ayşe B. Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14:540–578, October 2009.

[275] Olivier Vandecruys, David Martens, Bart Baesens, Christophe Mues, Manu De Backer, and Raf Haesen. Mining software repositories for comprehensible software fault prediction models. *Journal of Systems and Software*, 81:823–839, May 2008.

[276] Alan Veevers and Adam C. Marshall. A relationship between software coverage metrics and reliability. *Software Testing, Verification and Reliability*, 4(1):3–8, 1994.

[277] Jeffrey M. Voas and Keith W. Miller. Software testability: The new verification. *IEEE Software*, 12:17–28, 1995.

[278] Stefan Wagner. A model and sensitivity analysis of the quality economics of defect-detection techniques. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 73–84, 2006.

[279] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Timeaware test suite prioritization. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, 2006.

[280] Shinya Watanabe, Haruhiko Kaiya, and Kenji Kaijiri. Adapting a fault prediction model to allow inter language reuse. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, PROMISE '08, pages 19–24, 2008.

[281] Michael Wedel, Uwe Jensen, and Peter Göhner. Mining software code repositories and bug databases using survival analysis models. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 282–284, 2008.

[282] Cathrin Weiß, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proc. International Working Conference on Mining Software Repositories (MSR'07)*, pages 1–8, 2007.

[283] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. Using developer information as a factor for fault prediction. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 8–14, 2007.

[284] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. Comparing negative binomial and recursive partitioning models for fault prediction. In *Proceedings of the 4th*

International Workshop on Predictor Models in Software Engineering, PROMISE '08, pages 3–10, 2008.

[285] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13:539–559, October 2008.

[286] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering*, 15:277–295, June 2010.

[287] Laurie Williams, Gunnar Kudrjavets, and Nachiappan Nagappan. On the effectiveness of unit test automation at microsoft. In *Proceedings of the 20th IEEE international conference on software reliability engineering*, ISSRE'09, pages 81–89, 2009.

[288] Laurie Williams, E. Michael Maximilien, and Mladen Vouk. Test-driven development as a defect-reduction practice. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, 2003.

[289] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques (Second Edition)*. Morgan Kaufmann Publishers Inc., 2005.

[290] W. Eric Wong, Joseph R. Horgan, Saul London, and Hira Agrawal Bellcore. A study of effective regression testing in practice. In *ISSRE '97: Proceedings of the Eighth International Symposium on Software Reliability Engineering*, 1997.

[291] W. Eric Wong, Joseph R. Horgan, Michael Syring, Wayne Zage, and Dolores Zage. Applying design metrics to predict fault-proneness: a case study on a large-scale software system. *Software: Practice and Experience*, 30(14), 2000.

[292] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 15–25, 2011.

[293] Shujian Wu, Qing Wang, and Ye Yang. Quantitative analysis of faults and failures with multiple releases of softpm. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 198–205, 2008.

[294] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 26–36, 2011.

[295] T.-J. Yu, V. Y. Shen, and H. E. Dunsmore. An analysis of several software defect models. *IEEE Transactions on Software Engineering*, 14(9):1261–1270, 1988.

[296] X. Yuan, T.M. Khoshgoftaar, E.B. Allen, and K. Ganesan. An application of fuzzy clustering to software quality prediction. In *IEEE Symposium on Application-Specific Systems and Software Engineering Technology, 2000.*, pages 85 –90, 2000.

[297] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 93–102, 2011.

[298] Hongyu Zhang. 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, 2008.

[299] Hongyu Zhang and Rongxin Wu. Sampling program quality. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, 2010.

[300] Xuemei Zhang and Hoang Pham. Software field failure rate prediction before software deployment. *Journal of Systems and Software*, 79:291–300, March 2006.

[301] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32:240–253, April 2006.

[302] Yuming Zhou and Hareton Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering*, 32:771–789, October 2006.

[303] Yuming Zhou, Baowen Xu, and Hareton Leung. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *Journal of Systems and Software*, 83:660–674, April 2010.

[304] T. Zimmermann and N. Nagappan. Predicting subsystem failures using dependency graph complexities. In *International Symposium on Software Reliability, 2007*, pages 227 –236, nov. 2007.

[305] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 531–540, 2008.

[306] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects with program dependencies. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 435–438, 2009.

[307] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 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, 2009.

[308] Thomas Zimmermann, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy. Characterizing and predicting which bugs get reopened. In *Proceedings of the 34th International Conference on Software Engineering*, June 2012.

[309] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *Proc. International Conference on Software Testing, Verification and Validation (ICST'10)*, pages 421–428, 2010.

[310] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for Eclipse. In *PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, pages 1–7, 2007.

[311] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, 2004.

Appendix A

Selection of Surveyed Papers

In order to conduct our survey, presented in Chapter 2, of the state-of-the-art in SDP research, we needed a method for selecting which papers to include. In this Appendix, we detail our selection process. To select the papers, we decided to start with a survey of the literature on SDP. To perform our systematic review, we used a number of search queries, shown in Table A.2. We considered all publications in major software engineering journals and conferences. Table A.1 shows the venues considered for our survey. Once a list of all the papers was returned, we manually sorted through each paper, reading its title and abstract, to decide whether it will included in our survey or not. In total, we had a total of 185 papers of which 102 were formally included in our survey, 24 were summarized or cited but not formally surveyed and 46 were rejected. Table A.3 provides a list of the rejected papers and the reason for rejection.

We carefully document all of the steps taken to select our papers to facilitate the replication of our survey. Having the process well documented means that other researchers will be able to know how we generated the list of papers in this survey.

Table A.1: List of considered venues

Type	Acronym	Description
Journal	TSE	IEEE Transactions on Software Engineering
	TOSEM	ACM Transactions on Software Engineering and Methodology
	ESE	Empirical Software Engineering
	JSS	Journal of Systems and Software
	ASE	Automated Software Engineering
	JSME	Journal of Software Maintenance and Evolution: Research and Practice
	SPE	Software - Practice & Experience
Conference	ICSE	International Conference on Software Engineering
	ICSM	International Conference on Software Maintenance
	PROMISE	International Conference on Predictive Models in Software Engineering
	ESEM	International Symposium on Empirical Software Engineering and Measurement
	MSR	Working Conference on Mining Software Repositories
	ESEC/FSE	ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering
	SCAM	Int'l Working Conference on Source Code Analysis and Manipulation
	ASE	International Conference on Automated Software Engineering
	ISSTA	International Symposium on Software Testing and Analysis
	ISSRE	International Symposium on Software Reliability Engineering
	WCRE	Working Conference on Reverse Engineering

Table A.2: List of queries

Query name	Query	No. returned results
Q1	('defect prediction' OR 'fault prediction' OR 'failure prediction' OR 'bug prediction')AND("Publication Title": 'Software Engineering'OR "Publication Title": 'Software Maintenance'OR "Publication Title": 'Predictive Models in Software Engineering'OR "Publication Title": 'Empirical Software Engineering'OR "Publication Title": 'Mining Software Repositories'OR "Publication Title": 'Foundations of Software Engineering'OR "Publication Title": 'Source Code Analysis and Manipulation'OR "Publication Title": 'Automated Software Engineering'OR "Publication Title": 'Software Testing and Analysis'OR "Publication Title": 'Software Reliability Engineering'OR "Publication Title": 'Reverse Engineering'OR "Publication Title": 'Software Engineering, IEEE Transactions on ')	206
Q2	After manually selecting papers from Q1	81
Q3 (ACM)	Searching for: ("defect prediction" or "failure prediction" or "fault prediction" or "bug prediction") and (PublishedAs:journal OR PublishedAs:proceeding OR PublishedAs:transaction OR PublishedAs:magazine)	276
Q4	After manually selecting papers from Q3	65
Q5	DBLP manual "find" search in ESE	12
Q6	DBLP manual "find" search in JSS	18
Q7	DBLP manual "find" search in ASE	1
Q8	DBLP manual "find" search in SPE	3

Rejected Papers

In this subsection, we list the rejected papers.

Table A.3: List of rejected papers and reason for rejecting the paper

Papers Rejected	Reason
[252], [201], [278], [262]	Focus on effort and cost estimation
[48]	Focus on inspections
[248], [84], [157, 158], [9]	Focus on process
[171]	Focus on root-cause analysis
[161], [269], [72], [91]	Focus on characterizing risk and defects
[39]	Focus on defects in non-source code artifacts
[137], [20], [57]	Focus on change proneness
[174], [177], [300], [247]	Focus on reliability
[26], [110], [235]	Focus on models
[151], [141], [229]	Focus on defect finding tool
[250], [299], [139]	Focus on data sampling
[130]	Focus on software aging
[120]	Focus on predicting size
[179], [260]	Focus is on metrics
[258]	Focus on feature selection
[241]	Focus on project similarity
[52]	Focus on confirmation bias
[17]	Focus on bug report triage

1. Shepperd *et al.* [252] study different accuracy indicators for cost estimation prediction. The paper did not perform defect prediction.

2. Braind *et. al* [48] study the applicability of using capture-recapture to determine the number of remaining defects in inspected artifacts. The main focus of the paper is inspections, not defect prediction.

3. Schneidewind [248] and Evanco [84] study the fault correction process. The main focus of the paper is on the process rather than predicting defects.

4. Lezak *et al.* [171] perform a retrospective study of defect modification requests (MRs). The main focus of the paper is to perform root-cause defect analysis and investigate process and code complexity metrics with defects. The authors do not perform defect prediction.

5. Koru and Tian [161] analyze a large set of complexity metrics and defect data collected from six large-scale software products to compare and characterize the similarities between high defect and high complexity modules. They find that high defect modules are not typically the most complex. The focus of the paper is defect characterization, not prediction.

6. Mockus *et al.* [201] predict the effort required to address modification requests that repair defects. The paper does not perform any defect prediction.

7. Biffl [39] estimate major defects in software requirements documents. They do not predict defects in source code.

8. Jung *et al.* [137] use product metrics to build a prediction model of the number of change requests, not defects. The main focus of the paper is predicting changes. The main focus is not predicting defects.

9. Arisholm *et al.* [20] study the use of dynamic coupling measures to predict change proneness, not defects.

10. Li *et al.* [174] perform an empirical study on the OpenBSD project where they aim to predict model parameters of software reliability growth models. The main focus is reliability, not predicting defects.

11. Bai *et al.* [26] examine how Bayesian Networks can be used to model the number of remaining defects. The main focus of the paper was on the technicalities of the Bayesian Networks and not on defect prediction.

12. Wagner [278] proposes an analytical, stochastic model of the economics of defect detection and removal. The main focus of the papers was on economics, not defect prediction.

13. Tomaszewski and Damm [269] characterize the risk of introducing faults in modified and new classes. The main focus of the paper was characterizing risk, not defect prediction.

14. Lo and Huang [177] focus on software reliability models. The paper does not perform defect prediction.

15. Zhang and Pham [300] build reliability growth models that take into account fault fixes. The main focus of the paper is reliability, not defect prediction.

16. Kim *et al.* [151] present a bug finding tool, BugMem, that uses history to find project-specific bugs. The focus of the paper is on the bug finding tool. The paper does not perform any prediction.

17. Cerulo [57] propose the use of software repositories in the areas of impact analysis, change request assignment and crosscutting concern mining and show how using software repositories can improve the performance of these software engineering models. The paper does not perform defect prediction.

18. Song *et al.* [262] predict defect associations and the defect correction effort. The paper does not perform defect prediction.

19. Elbaum [72] propose the monitoring of system field behaviour and their deviations to identify conditions that precede failures in deployed software. They do not perform any defect prediction.

20. Scott and Johnson [250] present a method where metrics are combined with a few sampled classes to generalize the fault content to an entire system. The paper does not perform defect prediction.

21. Jiang and Xu [130] build a stochastic time series decomposition algorithm to model software aging. The paper does not perform defect prediction.

22. Joshi *et al.* [136] uses the number of defects in one month to predict the number of defects in the next month. The paper was a challenge report and was missing many details, therefore, it was not surveyed.

23. Herraiz *et al.* [120] use time series analysis predict the evolution (i.e., size) of libre software. The paper does not predict defects.

24. Fenton *et al.* [87] present a comprehensive dataset of 31 software development projects. Different characteristics of the data set are discussed, no defect prediction is performed.

25. Kamei *et al.* [139] examine how over and under sampling of the training data affects the performance of the prediction of fault-prone modules.

26. Kim and Ernst [150] use the history of changes to prioritize which warnings should be fixed. The focus of the paper is not defect prediction.

27. Haider *et al.* [110] propose a ED^3M model to estimate defects. The focus of the paper is on the model being proposed, rather than defect prediction.

28. Rasschou and Rainer [235] describe the Exposure Model, which they apply to estimate the number of customer defects and defect correction effort. The focus of the paper is on the exposure model, rather than defect prediction.

29. Klas *et al.* [157,158] outline their method which combines measurement data and expert judgement. The paper does not perform any defect prediction.

30. Marcus *et al.* [179] propose a new cohesion metric for OO classes that are based on the unstructured information in source code. The focus of the paper is on the cohesion metric, rather than defect prediction.

31. Schneidewind and Hinchey [247] model software complexity and reliability. No defect prediction is performed.

32. Snipes *et al.* [260] build models to approximate deployment metrics. The paper does not perform defect prediction.

33. Abreu and Premraj [9] investigate the relationship between developer communication and bug-introducing changes. They perform a case study on the Eclipse JDT project and show that developer communication is correlated with bug-introducing changes. The paper does a correlation study, not a defect prediction study.

34. Shivaji *et al.* [258] propose a feature selection technique that can be applied to classification-based bug prediction. The focus of the paper is the feature selection technique, not defect prediction.

35. Robinson and Francis [241] present a metric-based study that compares open source and industrial programs. They identify open source projects that are most similar to industrial programs. The paper does not perform any defect prediction.

36. Calikli and Bener [52] empirically analyze the effect of confirmation bias (i.e., the tendency of people to verify hypotheses rather than refute them) of software developers on software defect density. The focus of the paper is on confirmation bias, not defect prediction.

37. Zhang and Wu [299] propose a sampling based approach which requires a small percentage of source files to estimate the quality of large software systems. The focus of the paper is on data sampling, rather than defect prediction.

38. Kessentini *et al.* [141] propose an approach that is based on the notion that the more code deviates from good practices, the more likely it is bad. The paper builds a tool and does not perform any defect prediction.

39. Pham *et al.* [229] developed a tool called SecureSync that can detect recurring software vulnerabilities in systems that reuse source code or libraries. The paper builds a tool and does not perform any defect prediction.

40. Ferrari *et al.* [91] perform an exploratory study on the fault-proneness of aspect-oriented programs. They report their findings on the mechanisms of AOP that affect fault-proneness. The paper performs a characterization of defects due to AOP, but does not predict defects.

41. Anvik and Murphy [17] present a machine learning approach to assist bug triagers in assigning bug reports. The do not perform defect prediction, rather they predict who should address a bug report.

42. Rahman *et al.* [236] study the impact of ownership and developer experience on software defects. The authors use statistical techniques to perform their study, but do not perform any prediction.

43. Debroy and Wong [64] investigate the relationship between failure rates and the number of test cases required to detect the faults. Their goal is to estimate the test size using failure rates.

44. Eyolfson *et al.* [85] examine the effect of time of day and developer experience on commit bugginess in two open source projects. The authors find that approximately 25% of commits are buggy, that commits checked in between 00:00 and 4:00 AM are more likely to be buggy, developers who commit on a daily basis write less-buggy

commits and bugginess for commits per day of the week vary for different projects. No prediction was performed.

45. Bhattacharya and Neamtiu [38] examine the impact of programming language on software quality. No prediction is performed.

46. Koziolek *et al.* [165] apply a model-driven prediction method to evaluate the evolution scenarios of large software systems.

Appendix B

Summaries of Surveyed SDP Papers

In Chapter 2, we provided a survey of the state-of-the-art in SDP research. In our survey, we read and summarized a total of 102 papers published from the year 2000-2011. In this Appendix, we provide short summaries of the surveyed papers and highlight the main findings of prior SDP research. Each paper is summarized in a paragraph and related papers are grouped in subsections. To improve readability, we summarize the main findings of the surveyed papers in the boxes below.

B.0.6 Data Sources and Granularity

Data Sources

As shown earlier in Table 2.2, the majority of the surveyed papers use data from source code and bug repositories. We mention data sources used for each paper in its summary. There has not been any research specifically about data sources for SDP (i.e., which data sources to use or the usefulness of the different data sources), but a number of recent studies focused on effect of bias in the data used in SDP studies.

Nikora and Munson [220] show that the method by which faults are counted can have a significant effect on fault predictors. They propose a standard for precise enumeration of faults from configuration control documents and show that using their definition provides higher quality fault models.

Bird *et al.* [41] studied how improper linking of data from multiple repositories effects the performance of SDP techniques. To mitigate some of the issues related to improper linking, Bachmann *et al.* [25] present a tool that assists in the linking of bug-fixing commits (from source code repositories) to the bug reports which are stored in bug repositories.

Bachmann and Bernstein [24] investigate the effect of process data quality and product quality. They show that, for example, a high number of empty commit messages in Eclipse correlates with bug report quality and the number of bugs reported.

Nguyen *et al.* [217] study the effects of bias for a commercial project where strict development guidelines and rules on the quality of the data are enforced. They find that even in such near-ideal circumstances, bias in data still exists. Therefore, they conclude that bias is related to the underlying software development process and not the heuristics being used to link repository data.

Kim *et al.* [154] investigate the impact of, and propose approaches to deal with noise in defect data. They find that defect prediction models are resilient to false positive and false negative noises up to 20-35%. Then, they propose a noise detection algorithm called CLNI, which detects noisy instances so they can be removed. They find that using CLNI can help improve the defect prediction accuracy. Also, Wu *et al.* [292] propose an algorithm that links bug reports and commits that address them.

Main findings of surveyed papers:

SDP studies should be aware of potential bias due to the use of repository data. Noise reduction algorithms such as CLNI should be applied to reduce noise in defect data.

Data Granularity

Koru and Liu [163] used the J48 and KStar machine learning algorithms to predict defects in five NASA projects. They used 31 product metrics in their predictions. The authors showed that partitioning data according to module size and increasing the level of abstraction at which predictions are made (i.e., class-level vs. method-level), improves the prediction performance. They used precision, recall and the f-measure to evaluate their predictions and showed that they can achieve an f-measure of 0.56 for defective classes and 0.97 for non-defective classes.

Holschuh *et al.* [124] predict defects in SAP Java code. They used 78 different metrics, covering complexity, dependency, code smell and change metrics. The authors perform two types of predictions, short term - where they predict defects 2 months after release and long term - where they predict defects 8 months after release. They perform regression and classification. They use a linear regression model and SVMs to predict the number of defects of individual components. For classification, they used SVMs to classify the most 5%, 10% and 20% most defect-prone components. Spearman correlation was used to measure the predictive power of the regression analysis and precision and recall were used to gauge the quality of the classification. The authors perform cross-release prediction and find that: 1) correlations are stronger at the package level than at the class level. At the package level, for the entire software system, the Spearman correlation ranges between 0.3-0.5 and the hit rate (i.e., precision) is between 45 - 55%. When focusing on specific projects, the correlations improve to 0.7 and the hit rate improves to above 60%. At the class level, the Spearman correlations are between 0.4-0.5 and the hit rate is around 50%. Precision and recall are both around 0.5, suggesting that every second file predicted to be defect-prone, actually had defects and that every second file with defects was correctly predicted as being defect-prone.

Zimmermann *et al.* [310] used 45 product metrics to predict post-release defects in the Eclipse open source project. The authors build linear and logistic regression models to predict defects in releases 2.0, 2.1 and 3.0 of Eclipse. To evaluate the effectiveness of the predictions, the authors use Spearman correlation and precision and recall. The main findings were: 1) that the number of pre-release defects is highly correlated with post-release defects, 2) predictions are more accurate at the package level than at the file-level AMS 3) models built on earlier releases can be used to predict for later releases. At the package level, the authors achieve precision between 0.64-0.78 and recall between 0.62 - 0.79. At the file level, the predictions achieve a precision between 0.45 - 0.67 and recall between 0.18 - 0.38.

Main findings of surveyed papers:

Increasing the level of abstraction at which the prediction is performed at (i.e., predicting at the package level vs. file or function level) produces better results.

Our remarks:

Although increasing the level of abstraction improves the prediction performance, the practical value of the predictions decrease as the abstraction level increases.

Khoshgoftaar *et al.* [147] show how principle component analysis (PCA), a mathematical procedure that transforms correlated variables into a set of linearly uncorrelated variables, can improve classification-tree models. They use 42 metrics to predict post-release defects in a large telecommunication system. They find that the PCA models had better accuracy (in terms of misclassification rates) than the raw models which use the actual metric values, and argue that using PCA can aid classification tree modelling. In a related paper [148], they evaluate the hypothesis that models that are specifically built for a subsystem will yield more accurate results than system-wide models (i.e., the subsystem that a module belongs to provides be valuable information that should be used when modelling software quality). They use classification trees using the classification and regression trees algorithm and show that indeed a model built with training data on the subsystem alone was more accurate than a similar model built with training data on the entire system.

Main findings of surveyed papers:

Using data from the same subsystem to build prediction models provides better accuracy than a similar model built using data from the entire system.

B.0.7 Metrics

Product Metrics

Nagappan and Ball [209] use static analysis tools to predict the density of pre-release defects in Windows Server 2003. They build linear regression models to predict pre-release defect density and show that there is strong correlation between the predicted pre-release defect density and actual pre-release defect density. Then, the authors used discriminant analysis to categorize components into fault-prone and not fault-prone based on the defect densities and achieve an overall classification accuracy of 82.9%.

Tomaszewski *et al.* [270] propose the use of 3 metrics that can be constructed before code is implemented to predict faults in classes. They perform a case study on three releases of two large telecommunication systems produced by Ericsson. They find that using their early prediction metrics, they are able to provide predictions that are of similar quality to predictions based on metrics available after the code is implemented.

Zhou and Leung [302] use six OO design metrics to predict low, ungraded (i.e., no severity assigned) and high severity defects in the KC1 Nasa project. They use logistic regression and machine learning (i.e., Naive Bayes, Random Forest and NNge) and perform their predictions at the class level. They found that the OO design metrics predict low-severity fault-prone classes better than high severity faults in fault-prone classes.

Arisholm and Briand [19] use OO and historical metrics to predict fault-prone classes in a large legacy telecommunication system. They use a logistic regression model and show that their models can achieve less than 20% false positives and negatives. They also show that change and fault data in previous releases is paramount in to developing a practically useful prediction model. In fact, using these historical metrics can save verification efforts by 20%.

Olague *et al.* [223] compare the use of three different OO metrics suites: the Chidamber and Kemerer (CK) metrics, Abreu's Metrics for Object-Oriented Design (MOOD) and Bansiya and Davis' Quality Metrics for Object-Oriented Design (QMOOD) to predict fault-proneness of Java classes. They build logistic regression models for six revisions of the Rhino open source project and show that CK and QMOOD outperform (achieve accuracy 69.0 - 85.9%) models using the MOOD metrics suite in detecting fault-prone classes.

Jiang *et al.* [131] compare the use of design and code metrics in predicting fault-prone modules. They perform a comparison using 13 NASA projects and find that code-based models outperform design-level models. However, the authors mention that combining both types of metrics outperforms models which use either one of the metric sets.

Erika and Cruz [80] propose the use of a UML based metric, called UML_RFC to predict faulty classes. UML_RFC accounts for the number of different messages received by all objects of a class and the total number of messages sent to different objects by all objects of a class. They build logistic regression models and compare the use of their UML-based metric to code-based metrics. Using specificity, sensitivity and correctness as the performance measures, they show that their UML metric can predict faulty classes just as good as the code metrics.

Nugroho *et al.* [221] evaluate the use of UML design metrics to predict fault-prone Java classes. They build logistic regression models to predict fault-proneness in a healthcare system called IPS. They find that UML design metrics are more effective in predicting fault-prone Java classes than using code metrics.

Kpodjedo *et al.* [166] use design evolution metrics such as the number of added, deleted and modified attributes, methods and relations. They predict the presence of defects, the number of defects and defect density and compare their design evolution metrics to using CK and complexity metrics for several versions of Rhino, ArgoUML and Eclipse. They find that design evolution metrics can improve the detection of defective classes and defect density.

Cartwright and Shepperd [53] predict post-release defects in a commercial telecommunication subsystem, written in C++, using 12 product metrics. The authors build linear regression models and predict the number of defects in each of the 32 classes that make up the subsystem. The authors use Spearman correlation to measure the relationship between the different product metrics and the number defects. In addition they report the R^2 value of the linear regression models. One of the most important findings of the paper is that *inheritance classes (i.e., classes that inherit other classes) were approximately 3 times more defect-prone than classes that did not participate in inheritance structures*.

Wong *et al.* [291] use five design metrics to predict fault-prone functions. They aim to predict the top 5, 10, 15 and 20% fault-prone functions. The design metrics are: 1) D_i which represents the internal structure of a function. In particular, D_i accounts for the number of function invocations, the number of references to complex data types and the number of external device accesses. D_e focuses on a function's external relationships with other functions. D_e accounts for the amount of data flowing through the function (i.e., inflow and outflow) and the functions that are structurally below and above a given function (i.e., inflow and outflow). The remaining three metrics are combinations of the two aforementioned metrics: $D(G) = D_i + D_e$ and the union and intersection of D_i and D_e. The authors measure the performance of their predictions using precision and recall and show that their design metrics are good indicators of fault-proneness. For the top 20% fault-prone functions, they achieve a precision between 28.4-35.7% and a recall between 60.0 - 84.0%. They also compared their metrics to using function size and showed that function size does not replace any of their metrics.

El Emam *et al.* [78] use design metrics to predict faulty classes in a commercial software system, written in Java. The authors build a logistic regression model that was trained on one release and predicted the fault-prone classes in a future release. The authors show that the prediction models are accurate, achieving a precision of 80.9% and a recall of 70.8%. The authors control for size of a class, by using the number of attributes defined in a class and the number of methods in a class as proxy for class size. After controlling for size, they find that export coupling (which indicates if a class is used) and inheritance had the strongest relationship with fault-proneness.

Main findings of surveyed papers:

Design metrics are good predictors of defect-prone classes.

Ostrand *et al.* [224–226] use 6 metrics to predict which files in a large software systems will contain the highest number of faults in the next release. They showed that using a negative binomial regression model, they are able to correctly identify files that contained 71% to 92% of the faults across 15 different releases. They also find that LOC is the strongest predictor and explore the use of LOC to identify the most faulty files. They find that simply using LOC correctly predicts 73% to 74% of the faults. In a follow on study [32], the authors perform the same prediction on a voice response system that incorporates “continuous releases”. They find that nearly 75% of the defects are in 20% of the files.

Zhou *et al.* [303] use complexity metrics to predict post-release defects in three releases of Eclipse. The authors show that odds ratio associated with one standard deviation increase, rather than one unit increase, should be used to compare the relative magnitude independent variables. LOC and weighted methods per class (WMC) are the best predictors of fault-proneness. The explanatory power of other complexity metrics in addition to LOC is limited.

Main findings of surveyed papers:

LOC is a good predictor of post-release defects. Nearly 75% of the defects lie in 20% of the files.

Our remarks:

Although prior work showed that LOC is a good predictor, it is difficult to act on the LOC metric. For example, systems have to continue to evolve and LOC will increases. Therefore, LOC can be used as an indicator of defect-prone entities, but it is difficult to act upon.

Process Metrics

Graves *et al.* [105] use process and product metrics to predict faults in a large telephone switching system. The authors built Generalized Linear Models to examine which characteristics of a module's change history impact its quality. They performed their analysis at the module level. They found that the majority of complexity metrics were highly correlated with LOC and that the number of changes in the past and a module's age are the best predictors of its future faults. The authors also report that the number of developers who changed a module and the frequency with which a module co-changes with other modules were poor indicators of future faults.

Nagappan and Ball [210] use 8 process metrics related to code churn to predict defect density of binaries in the Windows Server 2003 project. The authors showed that high values of relative churn metrics indicate an increase in defect density. The authors apply discriminant analysis using relative churn metrics and were able to discriminate between fault and not fault-prone binaries with an accuracy of 89.0%.

Ratzinger *et al.* [238] use 17 different evolution metrics (e.g., number of authors, commit messages and bug fix counts) to predict defect densities in two open source and one commercial system. They utilize genetic programming and linear regression to predict software defects in the three software systems. The authors found that, contrary to other studies which found size and complexity measures to be good predictors, they found that the number of commit messages and the number of authors to be better predictors of defect densities.

Wu *et al.* [293] examine the relationship between pre-release and post-release defects in the SoftPM project. They use LOC and complexity metrics to evaluate the modules. They showed that in their project, modules that had a *few pre-release defects contained most of the post-release defects*. They also used fault density data to predict the failure density in the field.

Tarvo [266] uses change, code and dependency metrics to predict software regressions. They used logistic regression, multilayer perception and CART tree models to predict regressions. They used ROC to measure the performance of their prediction models and were able to achieve a mean area under ROC curve of 0.77.

Moser *et al.* [204] compare the predictive power of change metrics and static code attributes. The authors build logistic regression, Naive Bayes and decision tree models. They perform a case study on the Eclipse open source project and show that process metrics and more efficient predictors than code metrics. One novel contribution of this work is that it considers the cost of a prediction. The cost parameter α takes into consideration tradeoffs between precision and recall. Using an $\alpha = 5$, i.e., the cost of missing a defect is 5 times the cost of wasted effort of inspecting a clean file, they achieve more than 75% precision, 80% recall and less than 30% false positive rate.

Main findings of surveyed papers:

Process metrics such as the number of changes, or churn are good predictors of post-release defects. They are as good or better than product metrics.

Our remark:

This finding has been supported by many SDP studies. These studies were done on open source and commercial projects.

D'Ambros *et al.* [62] examine the relationship between change coupling metrics and defect predictions. They use correlations and build defect prediction models using code and change coupling metrics. They find that change coupling metrics can improve (in terms of model fit) prediction models that are built using complexity metrics.

Hassan [116] use the complexity of a software code change to predict faults. He uses linear regression models to predict faults in five open source systems. He compares using the complexity of a code change to using prior changes or prior defects and finds that using the complexity of a code change are better predictors.

Ferzund *et al.* [92] use hunk (i.e., small code units that compose software changes) metrics to predict whether or not a hunk is bug inducing. They use logistic regression and random forests to predict the bug-introducing hunks in seven open source projects. They find that hunk metrics can classify hunks with a 81% accuracy, 77% buggy hunk precision and 67% buggy hunk recall.

Giger *et al.* [101] use fine-grained source code changes (SCC) to predict bugs in Eclipse. A comparison is made to using churn in terms of the lines modified (LM), of the changes. The authors build prediction models using a number of classifiers, including LR, NB and RF. They show that SCC outperforms LM when correlating with the number bugs, when predicting the number of bugs and when whether or not a file is bug-prone.

Main findings of surveyed papers:

Change coupling and complexity metrics are good predictors of defect-prone files.

Other Metrics

Mockus *et al.* [202] use 9 metrics that cover deployment issues, usage, platform and hardware configurations to predict customer perceived quality. They perform their study on a large telecommunication system and use logistic regression to study the importance of the various metrics. They find that deployment schedule, hardware configurations and software platform may increase the probability of observing a software failure by more than 20 times.

Li *et al.* [173] perform an empirical study at ABB Inc and share their experiences of applying defect prediction in practice. They perform their study on two large commercial projects. The authors make the following findings: accuracy measures such as precision and recall are not the most important criterion in certain settings. Explainability (i.e., the ability to attribute the effects to a predictor) and quantifiability (i.e., the ability to quantify effects of a predictor) may be more important in practice. Also, cross validation and data withholding may not be adequate in practice, rather, cross-release evaluations may be more important. Acknowledging and dealing with missing information is an important issue and more accurate and valid models may be produced by replacing this missing information with information from similar projects/releases. They also examined the importance of metric categories used for post-release defect prediction and found that development, deployment, usage and SW/HW configuration metrics are important.

Main findings of surveyed papers:

Deployment schedule, hardware configurations and software platform may increase the probability of observing a software failure by more than 20 times.

Our remarks:

These findings have only been validated on commercial systems. Whether the same observation holds for open source software projects remains as an open question.

Pan *et al.* [227] use 13 program slicing metrics to predict defects in source code files. They perform a case study on Apache HTTP and Latex2rtf, where they compare the performance of program slicing metrics to product metrics extracted by the Understand C++ tool. They find that program slicing metrics are effective in classifying defective files with an accuracy between 82.6 - 92.0%, compared to the product metrics which were able to achieve an accuracy between 80.4 - 88.0%. However, the authors do admit that program slicing metrics are expensive to calculate.

Main findings of surveyed papers:

Program slicing metrics are a good predictor of faulty-classes.

Our remarks:

To the best of our knowledge, only the study by Pan et al. *[227] investigated the use of slicing metrics for SDP. Other studies, especially on commercial systems, are need to examine whether this finding holds for other software systems.*

Bernstein *et al.* [34] use temporal features (e.g., the time that changes and faults occur within release time), in addition to product and process metrics to predict post-release defects in classes of the Eclipse project. The authors also use non-linear regression models, rather than using linear models. They show that using the temporal features and non-linear models, they are able to predict whether or not a file will have a post-release defect with 99% accuracy and predict the number of defects with a mean absolute error of 0.019.

Main findings of surveyed papers:

Combining temporal features (i.e., time-related features) with process and product metrics can significantly improve defect prediction results.

Weyuker *et al.* [283, 285] use developer information to further improve fault prediction models. They add the developer information to standard file and change characteristics (i.e., product and process metrics) and show that their models perform slightly better. When predicting the top 20% most fault-prone files, the models improve by 0.4%.

Meneely *et al.* [189] use the developer collaboration structure to predict failures in a Nortel networking software system. The authors use regression analysis to perform their prediction and show that using their model's prioritization revealed 58% of the defects in 20% of the files, compared to the optimal which finds 61% of the defects in 20% of the files.

Ramler *et al.* [237] conducted a study on eight releases of a software product to explore and describe the variance in delivered defects associated to different developers. They find that there are differences between the defects delivered from different developers, which may be related to testing intensity.

Mockus [197] investigates the relationship between developer-centric measures and post-release, customer reported defects. He builds logistic regression models to perform his investigation on a large switching software. He finds that an organization's volatility (measured by the proximity to organizational change) increases customer reported defects, new members joining an organization had no impact on software quality, however, departures from the organization were associated with higher probability of customer reporter defects. Finally, he also finds that larger organizations have a higher probability of having defects.

Lee *et al.* [169] extract and use 56 developer interaction metrics to predict defects. The micro interaction metrics (MIM) covered a number of dimensions, such as the effort spent on a file, based on interaction data extracted from Mylyn. The authors build regression and classification models and show that MIM metrics can outperform or improve defect prediction performance over using code or history metrics, improving F-measure by 0.2.

Main findings of surveyed papers:

Adding developer information can improve the accuracy of defect prediction models.

Zimmerman and Nagappan [304, 306] use dependency metrics to predict the number of post-release failures in Windows server 2003 subsystems. The authors employ linear regression to perform their predictions. To measure the performance of their prediction, the authors used correlation. They showed that dependency metrics had correlations as high as 0.6 and 0.7. In a follow on study [305], the same authors use network analysis on dependency graphs of binaries in Windows server 2003 and compare them to complexity metrics. They find that using dependency metrics yields a 10% improvement in recall and identifies twice as many binaries that are considered critical by developers, compared to complexity metrics.

Pinzger *et al.* [230] investigate the relationship between the developer networks (i.e., fragmentation of developer contributions) and the probability and number of post-release failures in Windows Vista binaries. They build logistic and linear regression models and show that central modules are more failure-prone than modules located in surrounding areas of the network. They also confirm prior findings that the number of authors and commits are good predictors of post-release failures.

Bird *et al.* [43] argue that program dependencies and social factors need to be considered together since they interact so closely. They propose the use of so called socio-technical network metrics to predict fault-proneness. They perform a study on Windows Vista and Eclipse and show that using socio-technical metrics outperforms dependency and contribution models, achieving precision and recall values up to 85%.

Nguyen *et al.* [218] perform a replication study of Zimmermann's study [305] where they examine the impact of dependency network measures on post-release defect in Eclipse. They build logistic regression models to predict the quality of modules and find that only a small number of dependency network measures have a large impact post-release defects.

Main findings of surveyed papers:

Software and developer dependency metrics can be used to improve defect prediction.

Nagappan *et al.* [214] use metrics that quantify organizational complexity to predict fault-prone binaries in Windows Vista. The authors build logistic regression models and use precision and recall to evaluate their organizational structure models to models built using churn, complexity, coverage, dependencies and pre-release bugs. They show that organization structure metrics achieve a precision of 86.2% and recall of 84.0% when predicting failure-prone binaries.

Main findings of surveyed papers:

Organizational metrics are better predictors of defects than churn, complexity, coverage, dependencies and pre-release defects.

Our remarks:

This finding has only been examined at Microsoft. Whether this finding holds for other software systems is still an open issue.

Shin *et al.* [257] investigate the effectiveness of adding calling structure metrics in fault prediction models. They build negative binomial regression models to predict the number of post-release faults in a file. They rank the files based on the number of faults and compare their prediction of the top 20% most fault-prone files. They find that calling structure metrics only provide marginal benefit (0.6%) to models that contain non-calling structure code metrics and change history metrics.

Main findings of surveyed papers:

Calling structure does not improve defect prediction.

Binkley *et al.* [40] use natural language metrics to improve fault prediction. The measures are based on the use of natural language in identifiers and code comments. They perform a case study using the Mozilla web browser project and a commercial project, called MP. They build linear mixed-effects regression models and show that these natural language features can improve fault prediction models, achieving a R^2 value between 0.32-0.61. This R^2 value is quite high, since Zimmermann *et al.* [310] were able to achieve R^2 values between 0.24-0.41 using product and process metrics (at the file level) in the Eclipse project.

Nguyen *et al.* [219] propose the use of topic models to predict defects. They view a software system as a collection of software artifacts that describe different technical concerns and use these concerns as input to machine learning-based defect predictors. They perform a case study on using the Eclipse JDT project and show that topic models achieve better performance than change, code, history, and churn metrics.

Main findings of surveyed papers:

Natural language metrics based on the source code can improve defect prediction.

Our remarks:

The idea of using natural language in SDP is a hot and upcoming topic.

Previous work used a large number of different metrics to predict defect-prone files, classes and functions. Based on this prior work, the size of the code and the number of pre-release defects are the best predictors of defect-prone files.

B.0.8 Metrics Used to Predict Defect-prone Changes

Aversano *et al.* [23] use a weighted terms vector representation of source code to predict whether or not a change will introduce a bug. The use five different prediction algorithms: K-nearest neighbor, logistic regression, multi-boosting, C4.5 and SVM to perform their prediction. They evaluate their approach on two open source projects and conclude that bug introducing changes can be predicted with 10.5 - 58.5% recall and 39.9 - 80.0% precision. The K-nearest neighbour model was ideal since it gave a good tradeoff between precision and recall.

Kim *et al.* [153] use metrics extracted from source code changes to predict whether a change will be buggy (i.e., introduce a defect) or clean. The features include text from the change log, metadata about the change (e.g., the author and time of the change) and complexity metrics related to the source code being changed. They employ SVM models to perform their prediction on 10 open source projects. The authors show that they can classify changes with an accuracy of 78% and achieve an average of 60% recall of buggy changes.

Main findings of surveyed papers:

Source code metrics are good predictors of defect-introducing changes.

B.0.9 Models

Models Used in SDP Studies

Yuan *et al.* [296] used 10 product and process metrics to predict post-release defects. They used fuzzy subtractive clustering to predict fault-prone modules. The predictions were performed on a commercial large legacy telecommunication system written in a high level language. The authors used the misclassification rates, effectiveness and efficiency to evaluate the accuracy of their predictions. Using a cutoff of 0.7, the authors achieve a misclassification rate of 0.27 for Type I errors and 0.30 for Type II errors. They achieve an effectiveness value of 0.70 and efficiency of 0.17.

Morasca and Ruhe [203] use 8 product and process metrics to predict post-release defects in software modules of the DATATRIEVE project. They compare using logistic regression-based and rough sets-based models and find that rough sets-based models perform better in correctly determining faulty modules. However, using a hybrid approach (i.e., using logistic regression and rough sets) performs best, correctly identifying between 81 - 100% of the faulty modules.

Khoshgoftaar *et al.* [142] compare Zero-Inflated Poisson regression to standard Poisson regression for defect prediction. They use five process and product metrics to predict defects in two large applications written in C++. They predict the number of post-release faults at the file level and find that Zero-Inflated Poisson regression provides better prediction accuracy than the standard poisson regression model. They mention that Zero-Inflated Poisson regression is more appropriate to use when the response variable of the data set includes a large number of zeros, which is common in most software engineering defect data sets.

Khoshgoftaar *et al.* [146] present a technique to determine which predictions by a classification tree should be considered uncertain. They performed a cross-release study where they predict post-release defects in a large telecommunication system. The authors used the TREEDISC algorithm and assessed the classes assigned to the leaves of the tree. They found that a sizeable subset of modules (16.9%) had uncertain classification. The authors argue that determining the modules with uncertain classification helps users of the prediction models by flagging which modules should be carefully investigated.

Quah and Thwin [232] used 14 OO design metrics to predict pre and post-release defects in a large commercial software system. They used two neural network models: Ward neural network and General Regression Neural Network (GRNN). The predictions were performed at the class level. The authors found that GRNN network model can predict software quality more accurately than the Ward network model.

Guo *et al.* [106] used the Dempster-Shafer (D-S) belief networks predict post-release defects in a NASA project. The authors used 21 product metrics and showed that using the (D-S) belief network outperforms logistic regression and discriminant analysis models.

Succi *et al.* [264] use code and design metrics to predict the number of pre-release defects in a commercial OO software system. They compare the use of negative binomial (NB) regression, zero-inflated NB regression and Poisson regression. They find that design aspects related to communication between classes and inheritance (especially the cardinality of the set all internal methods and external methods invoked by them and depth of inheritance tree) can be used to predict the most defect-prone classes. zero-inflated NB regression provides the best results in their case.

Amasaki *et al.* [12] use BBNs to predict post-release defects. They extract a number of metrics from the different development phases (e.g., design and coding or test and debug) and use metrics from prior phases to predict defects in subsequent phases. The authors show that they can achieve an error rate of 42.6% when predicting poor software quality.

Guo *et al.* [107] propose using Random Forests to predict post-release defects. They use 21 product metrics to predict faults in 5 different NASA projects. They perform their predictions at the subsystem level and use specificity, sensitivity, accuracy and probability of false alarm to evaluate their predictions. The authors show that their Random Forests based models achieves up to 87% defect detection rate with generally less than 25% of false alarms. The overall accuracy of their models were in the rang of 75% to 94%. They also showed that their Random Forests based models generally outperform common prediction models such as logistic regression and discriminant analysis.

Li *et al.* [175] examine and predict post-release defects across 22 releases of two commercial and two open source projects. They evaluate how well different models (i.e., Exponential, Gamma, Power, Logarithmic and Weibull) predict post-release defects and propose the use of a Weibull model due to its flexibility (i.e., its ability to take into account changes in the number of defects as the project evolves in time) in capturing defect-occurrence behaviour across a wide range of software systems. The Weibull model is the best performing model (in terms of AIC) in 16 out of the 22 releases studied. However, they do not discuss how to estimate the parameters of the Weibull model is difficult and warrants future research.

Hassan and Holt [117] use 4 process metrics to continuously highlight the ten most susceptible subsystems to have a fault. They perform a case study on six open source and use hit rate (i.e., the number of the top ten list that with a recently discovered fault) and the average prediction age (i.e., how early a prediction about a fault-prone subsystem can be made). The authors compare the four metrics and show that the most frequently fixed (MFF) and most frequently modified (MFM) metrics are the best predictors of fault-prone subsystem. Kim *et al.* [155] extended Hassan and Holt's work [117] and use the idea of a cache that keeps track of locations that were recently added, recently changed and where faults were fixed to predict where future faults may occur (i.e., faults within the vicinity of a current fault occurrence). They showed that by using a cache which contains 10% of the files, they are able to detect

73-95% of the faults.

Knab *et al.* [159] use 16 product and process metrics to predict the defect densities of files in the Mozilla web browser using decision tree learners. The authors perform a case study using seven different releases of Mozilla and show that decision tree learners can achieve good results. In addition, the decision trees are analyzed to determine the most important predictors of defect density.

Menzies *et al.* [190] use 38 product metrics to predict post-release defects in eight NASA projects. The authors compare the use of three ML algorithms, J48, Naive Bayes and OneR. They show that Naive Bayes is the best performing classifier. Also, the authors show that static code attributes can be used to build useful predictors that have a mean probability of detection of 71% and mean false alarms rate of 25%, however the best attribute differs from one data set to the other.

Mizuno *et al.* [195] apply spam filters on the source code to predict defects in two open source projects, argoUML and Eclipse BIRT. They show that applying spam filters on source code they are able to predict fault-prone files with a precision between 52.4 - 74.5%, a recall between 69.7 - 97.7% and accuracy between 56.4 - 79.7%. The authors extend their study and apply it to five open source projects in [119]. They show that using spam filters is effective in identifying fault-prone modules.

Koru *et al.* [164] argue that size, which is a good indicator of fault-proneness, needs to be monitored in a continuous manner, especially since size of a module evolves over time. They propose the use of a Cox proportional hazard model with recurrent events to study the effect of size on defect proneness in the Mozilla open source project. They find that the effect of size was significant and its effect on defect proneness was quantified.

Wedel *et al.* [281] propose the use of survival analysis to predict defects. They discuss the automated retrieval and pre-processing of raw data from code repositories so they can be used in the survival analysis models.

Kamei *et al.* [140] use association rule mining in combination with logistic regression to improve the accuracy of defect prediction of fault-prone modules. They perform a case study on the Eclipse open source project and compare their proposed approach to standard logistic regression, linear discriminant models and decision trees. They show that their approach can improve the f-measure of the prediction by as much as 0.163.

Zhang [298] uses polynomial functions to predict the number of defect in 14 different Eclipse packages. They use the MRE measure to measure the performance of their prediction models and show that their predictions achieve an MRE between 0 - 29.76%.

Wejuker *et al.* [284] compare the use of NBR model to recursive partitioning (RP). They perform their case study using 3 industrial systems and, using the same metrics, they find that NBR models identified files that contained 76-93% of the faults and RP models identified files that contain 68-85%.

Tosun *et al.* [273] propose the use of ensemble of classifiers to improve defect prediction. They propose combining three different algorithms - Naive Bayes, neural networks and voting feature intervals. The authors perform a study using 7 NASA projects. They find that using their ensemble of classifiers, they are able to improve performance of the prediction and detect 76% of the defects by inspecting 32% of the code, whereas using a standard predictor requires inspecting 50% of the code to detect 70% of the defects.

Layman *et al.* [168] use churn and structure metrics to predict faults in binaries of a large Microsoft project. They collected metrics in an iterative manner (i.e., in four-month snapshots) and used logistic regression models to identify fault-prone binaries. Their iteratively-built models are able to predict fault-prone binaries with an accuracy of 80.0%.

Gondra [103] propose the use of Artificial Neural Networks (ANN) to determine the importance of software metrics. They apply the ANN on the training data and use the identified metrics as the basis of an ANN-based predictive model. They use NASA data to examine the effectiveness of their prediction models in predicting fault-prone modules. They compare their ANN-based predictive model to an SVM-based model and find that their ANN model achieves an accuracy of 72.6%, whereas the SVM model achieves 87.4% accuracy.

Vandecruys *et al.* [275] use the Ant Colony Optimization (ACO)-based classification technique AntMiner+ to predict failure-prone modules. They use three NASA projects and compare their predictive accuracy to the c4.5, RIPPER, logistic regression, 1-nearest neighbor, SVM and majority vote classification techniques. They use specificity, sensitivity and accuracy to compare the accuracy of their technique and find that it performs comparable to the other techniques. However, the authors argue that the intuitiveness and comprehensibility of their technique is advantageous since it can help software managers know the reason behind the classification.

Elish and Elish [77] use SVM to predict fault-prone modules in four NASA projects. They compare the SVM models to eight other statistical and machine learning models. The authors use precision, recall, accuracy and f-measure to compare the performance and show that in most cases SVM has higher recall, however in terms of accuracy, precision and f-measure it has comparable performance to other techniques.

Liu *et al.* [176] propose the use of genetic-programming based approaches for fault-proneness prediction. They also propose combining data from multiple repositories since multiple repositories will provide additional information to improve predictive performance. They compare the genetic-programming based model to 17 other classifiers used in the prior work, such as Naive Bayes, Decision trees, etc. and show that the GP-based models perform better.

Weyuker *et al.* [286] compare the effectiveness of four modeling methods (i.e., NBR, recursive partitioning, random forests and Bayesian additive regression trees) for defect prediction. The metrics used to perform the prediction are: LOC, file age, the number of faults in the previous release, the number of changes in the prior two releases and the programming language used. They perform the prediction for 28-35 releases of three commercial systems and find that NBR and random forests outperform recursive partitioning and Bayesian additive regression trees.

Main findings of surveyed papers:

Many models such as Logistic Regression, Decision Trees, SVM and Random Forests are effective in predicting pre- and post-release defects.

Comparing Different Models for SDP

Gyimothy *et al.* [109] used 8 product (object oriented) metrics to predict the number of (pre and post-release) defects in seven releases of the Mozilla open source project. They compared the use of logistic, linear regression, decision trees and neural networks in predicting pre- and post-release defects. Their predictions were performed at the class level. *The authors found that all four prediction algorithms had similar performance.* The CBO metrics was the best predictor of defect-proneness of classes, however, the authors note that LOC also performed fairly well.

Khoshgoftaar and Seliya [144] use 42 product, process and execution metrics to compare the effectiveness of six commonly used fault prediction techniques in predicting post-release defects in a large legacy telecommunication system. They use Classification and Regression Trees (CART) based on least squares (CART-LS) and least absolute deviation (CART-LAD), S-plus regression trees (S-PLUS), multiple linear regression, artificial neural networks and case-based reasoning (CBR). They find that the CART-LAD model performs the best, while the S-PLUS model performs the worst. In a follow on study [145], the same authors compare several other classification techniques, namely: logistic regression, case-based reasoning, CART, S-PLUS, Sprint-Sliq, C4.5 and Treedisc. They also introduce a new performance measure called Expected Cost of Misclassification (ECM), which takes into account the cost of incorrectly classifying and instance. In this work, *the authors were not able to find any one* *best classification technique*. They mention that the performance varies across releases due to the characteristics of the data and the system being modeled.

Menzies *et al.* [192] examine three sub-sampling techniques (i.e., under-, over- and micro-sampling) and show their effect on defect prediction studies. Probability of detection, probability of false alarm and balance to measure performance. The authors perform experiments using 12 NASA projects and *show that a simple Naive Bayes classifier performs as well as other classifiers*. Furthermore, the authors show that micro-sampling, which uses a very small number of sample to train the models, yields performance that is similar to much larger training sets. This means that prediction models can be built with a less work since only a small number of samples is required to build accurate models.

Lessmann *et al.* [170] compare the 22 classifiers used for defect prediction using 10 different NASA data sets. They use code attributes as their predictors and use the AUC measure to gauge the accuracy of the defect predictions. *They find that the predictive accuracy across all classifiers are similar*, suggesting that the which model is used for the prediction is not of great importance.

Main findings of surveyed papers:

Prior SDP studies show that there is no, or a very small difference, in performance between different prediction algorithms.

Our remark:

This is an interesting finding. We believe that using simple models is desired since it makes SDP research easier to understand and implement in practice.

Tomaszewski *et al.* [271] empirically compare statistical fault prediction models with expert estimations. They use 9 product metrics to predict defects in two commercial telecommunication systems. One systems were made up of 249 classes (35 components) and 180 classes (43 components). They find that expert judgement is better than a random model, however, expert judgement performs worse than the statistical models, especially at fine granularity (i.e., class-level). In their study, the experts were able to perform their predictions at the component level only. However, the authors conclude that up to 15% of the code size, the expert judgement does not perform too badly. When more than 15% of the code is inspected, the statistical models significantly outperform the expert judgement. Klas *et al.* [156] propose a hybrid approach which incorporates expert and measurement data for defect prediction. They show that combining expert judgement improves the accuracy of model-based defect prediction models.

Main findings of surveyed papers:

Prediction models are most useful when more than 15% of the code needs to be inspected.

B.0.10 Performance Evaluation

Cross-project and Cross-release Studies

Fenton and Ohlsson [89] correlate the number of post-release faults in a telecom system from Ericsson Telecom AB. The main goal of the paper was to empirically validate existing hypotheses regarding defect prediction. Through their case study the authors made the following observations: 1) a small number of modules contains most of the faults discovered during pre-release testing and a similar observation was made regarding post-release (or operation) faults, 2) that size, complexity and pre-release failures are not a good indicator of post-release faults, 3) that fault densities remain roughly constant across major releases, both pre- and post-release and 4) that *software systems produced in similar environments have broadly similar fault densities*, pre- and post-release.

Denaro and Pezzé [66] predict the fault-prone files in the Apache Web server project. They built logistic regression models that use 38 different product metrics from Apache 1.3 to predict post-release faults in Apache 2.0. The authors report precision values between

77.8-85.7% and recall values between 50.0-68.2%. They also report R^2 values in between 0.38-0.54. One of the main conclusions of the study is that "high quality multivariate models can predict faults in software, *as long as the software applications belong to the same class*, ie., share the application domain, the development process and the development teams".

Briand *et al.* [50] investigate the feasibility of cross-project defect prediction. They use a number of design metrics exacted from the source code to predict the fault-proneness of classes. In addition to using logistic regression models, the authors use a novel exploratory analysis technique called Multivariate Adaptive Regression Splines (MARS). To evaluate their models, the authors use correctness, completeness and cost effectiveness (a measure they propose). They show that models built using one project can be used to accurately *rank* classes of another project, however the predicted fault probabilities will not be representative. They also show that their MARS model are more cost effective than simple logistic regression models. One important note is that the two projects have been developed in a nearly identical development team, using similar technologies (i.e., OO using Java) but different strategies and coding standards were used.

Nagappan *et al.* [212] use 18 product metrics to predict post-release defects in five Microsoft software systems. They use regression models and accurately predict binaries with post-release defects. The authors also explore the use of metrics across projects and conclude *that complexity metrics cannot be used for cross-project prediction, unless the projects are similar*.

Watanabe *et al.* [280] compare intra- (i.e., cross-release) and inter-project defect prediction. They use training data from one release or one project to predict faults in another release or another project. They use a decision tree model and find that they can perform intra-project prediction with a precision between 0.375 - 0.575 and a recall between 0.598 - 0.818. The authors also propose a compensation scheme for inter-project prediction which involves normalizing the metric values by the average from the project. They show that using their compensation scheme improves recall by approximately 15%.

Zimmermann *et al.* [307] examine the potential of cross-project defect prediction. They use data from three open source and seven commercial projects to examine which factors impact cross-project defect prediction. They find that 1) OSS projects are strong predictors of closed-source projects (even if they are in a different domain) , but not other OSS projects and 2) OSS projects cannot be predicted by any other projects. When examining which factors impact *how well a project can predict cross-project they find that having the same domain increases accuracy and having higher medians of the code measures in the test project seems to increase precision and recall*.

Erika *et al.* [81] suggest the use of log transformations on software metrics in order to enable cross-project defect prediction. They find that using these log transformations is useful for cross-project defect prediction.

Turhan *et al.* [274] examine the applicability of using cross-company static code features to perform defect prediction. They compare the cross-company models with in-company models and show, as expected, that in-company models perform better. They then examine the minimum number of defect reports required from in-company data (that needs to be added to cross-company data) in order to learn defect predictors and show that only a small number of in-company reports is needed. Hence, *cross-company models can be built with minimal effort, i.e., with a small number of in-company reports*.

Main findings of surveyed papers:

Cross-project prediction remains as an open issue. However, prior research indicates that cross-project prediction may be achievable for projects that share the same domain or development processes.

Accuracy Measures

Tosun and Bener [272] optimize the decision threshold to deal with the imbalance and high skew of software defect data. They apply decision threshold optimization to improve the performance of Naive Bayes prediction models. They use the AUC under the ROC to show that using the optimized threshold decreases the number of false positives.

The study by Li *et al.* [173] mentioned earlier performs an empirical study at ABB Inc and share their experiences of applying defect prediction in practice. Through their case study on two large commercial projects, the authors argue that accuracy measures such as precision and recall are not the most important criterion in certain settings. Explainability (i.e., the ability of the predictor to improve the fit of the prediction model) and quantifiability (i.e., the ability to quantify effects of a predictor) may be more important in practice.

Main findings of surveyed papers:

Accuracy measures such as precision and recall are not the most important criterion in certain settings. Explainability (i.e., the ability to attribute the effects to a predictor) and quantifiability (i.e., the ability to quantify effects of a predictor) may be more important in practice.

Our remarks:

Based on our experience with applying SDP in practice, we agree with the findings of the study by Li et al. *[173]. We believe that more work should focus on how to measure the practical performance of SDP research.*

Ma and Cukic [178] argue that performance measures used to determine the performance of defect prediction models are not adequate. The authors perform a number of experiments using machine learning algorithms on five NASA projects. They show how using sensitivity, specificity, precision and overall accuracy in isolation can be misleading, since each of these measures only tells one side of the story. They propose the use of F-measure or G-mean, which are measures that combine precision, specificity and sensitivity.

Main findings of surveyed papers:

Using sensitivity, specificity, precision and overall accuracy in isolation can be misleading. Using F-measure or G-mean is more appropriate.

Jiang *et al.* [132, 133] propose the use of a cost curve to analyze and gauge the performance of prediction models. Cost curves are supposed to allow software quality engineers to introduce project-specific costs associated with misclassification. The authors perform a case study on 16 NASA projects and find that fault prediction models do not necessarily improve software quality of low or medium risk projects. For low and medium risk projects, the authors argue that even good prediction models do no outperform trivial classification. However, for high risk projects, defect prediction yields the most benefits.

Mende *et al.* [185] build Random Forests-based models for a multi-release telecommunication system. They develop a new evaluation measure, which they are argue should be used, based on the comparison to an optimal model which takes into account the LOC per file, called p_{opt}. They compare p_{opt} to AUC and show that p_{opt} has a low correlation, hence, it is different than AUC.

Main findings of surveyed papers:

Cost of misclassification should be taken into account when evaluating defect prediction techniques. Cost curves and LOC should be taken into account when evaluating defect prediction.

Song *et al.* [261] propose a framework for software defect prediction. They argue that their framework facilitates un comparison of defect prediction studies. They find that different learning schemes need to be applied for different data sets and that small details about how a study is evaluated can reverse findings.

Jiang *et al.* [134] examine variance in defect prediction studies. They conduct experiments on 12 NASA projects and find the following: 1) amongst the performance indicators precision, recall, f-measure and AUC, they find that AUC has the smallest variance and that as the data set increases, the variances of the performance measures shrinks; 2) larger projects offer more stable fault prediction models; 3) the lowest variance is associated with models developed and evaluated using 50% module subsets regardless of the classification method and that significant variances among different sizes of training/testing subsets are observed for small projects (with less than 500 modules); 4) 10-fold cross validation has a high variation for projects with a small number of modules.

Main findings of surveyed papers:

AUC should be used to measure the performance of SDP techniques since it has the lowest variance.

B.0.11 Other Considerations

Attribute and Feature Selection

Jia *et al.* [129] examine the effect of data transformation and attribute selection on post-release defect prediction. They perform a case study on three releases of Eclipse and an in-house project called QMP. They find that data transformations do not have a significant improvement, however, attribute selection may improve accuracy. However, which attribute selection technique is best depends on the classifier used. Generally speaking, the authors find that InfoGain provides good improvements.

Gao *et al.* [97] proposed a hybrid attribute selection technique that can be used to reduce the number of features used in defect prediction models. The hybrid approach first ranks the features and then a selection process is performed. They authors evaluate seven different feature ranking techniques and four different selection approaches. They perform their prediction using five commonly used classification algorithms on a large commercial project. They find that they are able to achieve the same or better performance while reducing the metrics in the model by 85%.

Shihab *et al.* [254] propose the use of odds ratio and a minimal set of metrics to better understand the impact of product and process metrics on post-release defects. They build logistic regression models on three versions of the Eclipse open source project. They show that three or four metrics provide the same prediction performance as using 34 metrics.

Main findings of surveyed papers:

Feature selection should be used to reduce the number of independent metrics.

B.0.12 Survey and Other SDP Papers

Survey Papers

Fenton and Neil [88] perform a critique of defect prediction studies prior to 1999. They surveyed papers using size and complexity metrics (e.g., [11, 90]), testing metrics (e.g., [276, 277]), process quality data (e.g., [67]) and multivariate approaches (e.g., [208]). Through a critical review, the authors point out some flaws that may influence defect prediction findings. The main flaws can be summarized as: 1) the lack or knowledge about the relationship between defects and failures, 2) the lack of consistency between the terminology used in different studies, 3) problems with multivariate approaches (i.e., factor analysis) which make it difficult to interpret terms of program features (i.e., LOC or complexity), 4) the problems with using size and complexity metrics to predict defects (i.e., the assumption that size and complexity have a straightforward relationship with defects) and 5) problems with data quality (i.e., the removal of data points, using averaged data) and statistical methodology (i.e., multi-collinearity, factor analysis or principal component analysis and fitting models vs. predicting data).

Catal and Diri [54] provide a systematic review of 74 fault prediction studies. Their study focused on answering the following questions: 1) which journal is the dominant software fault prediction journal, 2) what datasets are the most commonly used in fault prediction studies, 3) what methods are most commonly used for fault prediction, 4) what kinds of metrics are mostly used for fault prediction and 5) what percentage of fault prediction papers have been published after the year 2000. They found that 1) the IEEE Transactions on Software Engineering, the Software Quality Journal, the Journal of Systems and Software and the Empirical Software Engineering journal are the most important journals for software fault prediction research. They found that 60% of papers use private data, 31% use publicly available data, 8% use partially available data (i.e., data using open source projects that have not been shared with the research community) and 1% of the datasets are unknown (i.e., no information is given about the dataset in the paper). However, more recently (since 2005) publicly available datasets are being used (51%). With regards to the method used for fault prediction, they found that 59% of studies use machine learning based methods and only 22% of papers used statistical methods. The trends remain the same for recent work as well. With regards to the metrics used, the authors grouped papers into six categories: method-level, class-level, component-level, file-level, process-level and quantitative-level. The authors consider metrics that can be measured at the method-level as method-level metrics (e.g., lines of code or McCabe cyclomatic complexity). They found that the majority of paper use method-level (60%) and class-level (24%) metrics. Finally, the author note that 86% of the papers they survey were published after the year 2000, indicating that fault prediction work is becoming increasing popular.

Hall *et. al* [265] performed a literature review of fault prediction studies. Their study included 208 papers of which 36 papers were examined in detail. The authors were mainly interested in finding 1) how context affects fault prediction, 2) which independent variables should be included in fault prediction models and 3) which modeling techniques perform best when used in fault prediction. The authors found that predictive performance improves as software systems get larger. They also found that most studies report that models perform poorly when transferred to another project. With regards to independent variables, the authors found that models using only static code metrics (e.g., complexity metrics) perform poorly. Combining static code and Object Oriented (OO) metrics does not seem to improve performance. However, OO models outperform static code metric models. The authors also note that models based on LOC seem to be generally useful in fault prediction. The authors also mention that the use of process and developer data is not particularly related to good predictive performance. They note that models using a combination of metrics perform best. With regards to modeling techniques, the authors note that performance seems to be linked to the modeling technique used, which contradicts the finding by Lessmann *et al.* [170]. Studies using Support Vector Machines (SVM) techniques perform less well, whereas, models based on C4.5 seem to under-perform if imbalanced data is used (which is commonly the case for fault prediction). Naive Bayes and Logistic regression seems to perform relatively well.

D'Ambros *et al.* [61, 63] present a benchmark for defect prediction to facilitate the comparison of defect prediction approaches across different systems, to validate the difference in performance between different approaches and to investigate the stability of approaches using difference learners. They extract 44 process, source code, entropy of changes, churn of source code and entropy of source code metrics for five open source Java projects. The authors find that 1) metrics evaluated in isolation and metrics evaluated alongside larger sets of attributes have different behaviors and 2) different learners select different attributes, hence, it is difficult to achieve stability across projects or learners.

Using SDP to Empirically Examine the Relationships with Software Quality

A number of recent studies used SDP models to empirically examine the relationship been various aspects of the software development process and software quality. We summarize these papers below.

Cataldo and Nambiar [56] study the relationship between process maturity and geographic distribution. In particular, they study the combined impact of process maturity and geographic distribution on software quality. They find that process maturity and certain dimensions of distribution have a significant impact on software quality. Using statistical models, they also show that the benefit of process maturity diminishes as the development work becomes more distributed.

Meneely and Williams [188] examine the effect of the too many cooks in the kitchen phenomena on software security vulnerabilities. They perform their case study on three open source systems, Linux kernel, PHP and Wireshark. They find that files changed by six developers or more were at least four times more likely to have a vulnerability than files changed by less than six developers.

Giger *et al.* [100] use the Gini coefficient to investigate the role of ownership on software defects in the Eclipse platform. They find that less defects can be expected if a large share of all changes are performed by relatively few developers.

Bird *et al.* [44] study the relationship between ownership and software quality, measured as the number of pre- and post-release defects. They build linear regression models and perform a case study on Windows Vista and 7. They find that the number of minor contributors (i.e., contributors that make less than 5% of the changes to a binary) has a strong positive relationship with pre- and post-release failures, even when complexity, churn and size are controlled for.

Meneely *et al.* [187] examine the effects of team size, expansion and structure on software quality. They examine the correlations between monthly team level metrics and monthly product quality. They find that periods of accelerated team expansion are correlated with later periods of reduced software quality. However, linear team expansion was correlated with later periods of better software quality.

Other SDP Papers

Neufelder [216] investigated the correlation between software development practices and defect density. The author performed a study on 17 different organizations and showed that the following parameters: 1) consistent and documented formal reviews of system and software requirements, 2) the language and OS is well supported by industry, 3) the existence and use of test beds, 4) incremental testing, 5) scheduled regression testing, 6) the use of defect and failure tracking systems, 7) the use of a defined life cycle model, 8) the involvement of testers during requirements and design, 9) the use of automated unit testing tools and 10) the use of explicit test cases for user documentation have strong negative correlation with defect density.

Boetticher [47] argues that traditional sampling techniques such as random, stratified, systematic and clustered, all face a common problem - that is - they focus on the class attribute rather than the non-class attribute. Using a set of 20 experiments on five NASA projects, he shows that training on nice neighbours provides an average accuracy of 94%, whereas, training on nasty neighbours (i.e., neighbours with opposite values in the training set) achieves an accuracy of 20%. The author then proposes a new nearest neighbour sampling technique.

Koru *et al.* [160] observe a power-law relationship where defect-proneness increases at a slower rate than size. They come up with a theory, which they call the theory of Relative Defect Proneness (RDP) which states that smaller modules are *proportionally* more defect prone compared to larger ones. They perform a study on two commercial systems, they show an improvement of up to 341% in cost-effectiveness, when smaller modules are inspected first.

Ratzinger *et al.* [239] study the influence of refactoring on software defects. They extract

110 features that are related to refactoring and non-refactoring activities. They use four different classification techniques and show that refactoring and non-refactoring features lead to high quality defect prediction models. They also show an inverse relationship between refactoring activities and defects, suggesting that refactoring should be done to reduce defects.