Fault-prone Module Prediction Using Version Histories

A Dissertation

Submitted to the Graduate School of Information Science and Technology

of Osaka University

in Partial Fulfillment of the Requirements

for the degree

Doctor of Philosophy

in

Information Science and Technology

by

Hideaki Hata

January 2012

Abstract

Fault-prone module prediction is one of the most traditional and important areas in software engineering. Once fault-prone modules are predicted at an early stage of development, developers can track the predicted modules, which is useful in preventing the injection of additional faults.

One of recent findings in fault-prone module prediction studies is the usefulness of historical metrics, which can be collected from software repositories for fault-prone module prediction models. Many studies measure software development histories, such as changes on source code, events of development or maintenance processes, developer-related histories, and so on. In many papers, it is reported that historical metrics are more effective than traditional code complexity metrics. To find novel effective historical metrics, many researchers have conducted software repository mining.

First, we conduct a systematic review of recent fault-prone module prediction studies to clarify studied metrics and research trends. We investigate and report on two journals as well as five conferences from 2000 to 2010. Our findings are as follows. Historically, many metrics already exist to analyze version history information related to code, process, organization, and geography. New historical metrics tend to be proposed first in industry and then used in studies with open-source software projects. Compared to the accessibility of rich data in industry, it is not easy to collect rich historical information in open-source software projects. In addition, though historical metrics are considered effective in building prediction models, there are only a few studies conducting fine-grained prediction.

Based on a survey of fault-prone module studies, we address the following two problems: (1) *easily applicable prediction models* and (2) *fine-grained prediction with historical metrics*.

When applying fault-prone module prediction, the biggest problem is the lack of usable tools. In practical use, tools are needed because of the laboriousness of collecting metrics. For complexity metrics, source code analysis is needed, and tools should be implemented as program-language-specific. For most historical metrics, we need to analyze data from some software repositories, and repository-specific tools should be implemented. To tackle this problem, we propose text-mining-based prediction models for *easily applicable prediction models*. We treat source code as just text, and the number of tokens in source code is measured to build prediction models. Since we only measure the number of tokens, preparing program-language-specific and repository-specific tools is not necessary. To show the effectiveness of our prediction models, we compared prediction models using well-known metrics with our token metrics. Based on empirical study with open-source software projects, we show the higher prediction results with our prediction models than prediction models with well-known metrics. This result implies that: our token metrics are useful in building practical prediction models, and measuring sophisticated metrics is not always necessary for predicting fault-prone modules.

Fine-grained prediction with historical metrics is a desirable future direction in fault-prone module prediction. Predicting fault-prone modules on a fine-grained level is considered more cost-effective than coarse-gained prediction. This is because coarse-grained modules, whose sizes are bigger, require more cost to find and fix faults than fine-grained modules. Using complexity metrics, which requires only the source code of the present version, there are some studies predicting fault-prone methods, which are finer than files. However, there is no study using well-known historical metrics to predict fault-prone methods. This is because there has been no way to obtain entire version histories of methods as

rich as files. To obtain method-level version histories, we develop a fine-grained version control system, *Historage*. Using this system, we conduct fine-grained prediction using code-related, process-related, and organizational historical metrics. Method-level prediction models are compared with file-level prediction models with effort-based evaluation, which takes the cost of quality assurance activities into evaluation. An empirical study with open-source software projects implies that fine-grained prediction is cost-effective.

This dissertation is organized as follows. In Chapter 1, we give the background of the fault-prone module prediction studies and our results in this dissertation.

In Chapter 2, we present a survey of fault-prone module studies. We discuss the technique of extracting past fault information using version control systems and fault report management systems for study with open-source software projects, and the evaluation criteria of prediction results. In addition, we present a systematic review of recent studies.

Chapter 3 presents a study of text-mining-based prediction models. Text-mining-based metrics are compared with well-known complexity metrics and some historical metrics. Empirical evaluation is conducted with open-source software projects.

Chapter 4 presents a fine-grained prediction study. We collect historical metrics related to code, process, and organization for method-level and file-level modules. Both level prediction models are compared with an effort-based evaluation using open-source software data.

Finally, Chapter 5 concludes this dissertation with a summary and directions for future work.

List of Major Publications

(1) Hideaki Hata, Osamu Mizuno, and Tohru Kikuno, "An extension of fault-prone filtering using precise training and a dynamic threshold," In *Proc. of 5th Working Conference on Mining Software Repositories*, MSR '08, pages 89–98, May 2008.

(2) Hideaki Hata, Osamu Mizuno, and Tohru Kikuno, "Comparative study of fault-proneness filtering with PMD," In *Proc. of 19th International Symposium on Software Reliability Engineering*, ISSRE '08, pages 317–318, November 2008.

(3) Hideaki Hata, Osamu Mizuno, and Tohru Kikuno, "Fault-prone module detection using large-scale text features based on spam filtering," *Empirical Software Engineering*, 15(2):147–165, April 2010.

(4) Hideaki Hata, Osamu Mizuno, and Tohru Kikuno, "Reconstructing fine-grained versioning repositories with Git for method-level bug prediction," In *Proc. of 2nd International Workshop on Empirical Software Engineering in Practice*, IWESEP '10, pages 27–32, December 2010.

(5) Hideaki Hata, Ryosuke Morii, Osamu Mizuno, and Tohru Kikuno, "Quantitative analysis of method call changes related to bug fixing," *Trans. of Information Processing Society of Japan* (in Japanese), 52(2):801–816, February 2011.

(6) Hideaki Hata, Osamu Mizuno, and Tohru Kikuno, "Historage: fine-grained version control system for Java," In *Proc. of 3rd Joint International and Annual*

ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops, IWPSE-EVOL '11, pages 96–100, September 2011.

(7) Hideaki Hata, Osamu Mizuno, and Tohru Kikuno, "Inferring restructuring operations on logical structure of Java source code," In *Proc. of 3rd International Workshop on Empirical Software Engineering in Practice*, IWESEP '11, pages 17–22, November 2011.

(8) Hideaki Hata, Osamu Mizuno, and Tohru Kikuno, "A systematic review of software fault prediction studies and related techniques in the context of repository mining," *Computer Software* (in Japanese), (to appear).

(9) Hideaki Hata, Osamu Mizuno, and Tohru Kikuno, "Fault prediction on fine-grained modules based on historical metrics," *Trans. of Information Processing Society of Japan* (in Japanese), (conditionally accepted).

Acknowledgments

During the course of this work, I have been fortunate to have received assistance from many individuals. I would especially like to thank my supervisor Professor Tohru Kikuno for his continuous support, encouragement, and guidance for this work.

I am also very grateful to the members of my thesis review committee: Professor Shinji Kusumoto and Professor Takao Onoye for their invaluable comments and helpful criticisms of this thesis.

I would like to thank Associate Professor Tatsuhiro Tsuchiya for his assistance and invaluable advice.

I also would like to express my special thanks to Associate Professor Osamu Mizuno in the Kyoto Institute of Technology. He has guided me in all aspects of this work.

I wish to thank Assistant Professor Yoshiki Higo in Osaka University and Assistant Professor Norihiro Yoshida in the Nara Institute of Science and Technology for providing great feedback. I also wish to thank Dr. Livieri Simone for his helpful advice and support in this research.

Thanks are also due to every member in our laboratory. In addition, I wish to thank many friends who are interested in software engineering research in Osaka University and the Nara Institute of Science and Technology for their great comments.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Background

Software maintenance is a set of challenging activities in software development. Czerwonka et al. summarized common software maintenance characteristics as follows [18]:

- The software maintenance phase consumes the majority resources in the software product lifecycle.
- Maintenance activities are often done by people who have not created the software.
- The size of the maintenance team is much smaller than the size of the development team.
- Changes in the maintenance phase have high risks.

- The time for creating and verifying a fix is constrained.

In software maintenance, quality assurance activities including testing and inspections are inevitable. Finding and removing faults in the early phase of software development should save on the costs of the quality assurance. Due to the characteristics of software maintenance, focusing efforts on appropriate targets is essential. To tackle this problem, fault-prone module prediction has been studied. Fault-prone modules are predicted based on past faulty module information. Traditionally, prediction models have been built with *complexity metrics*, such as McCabe's cyclomatic complexity [78], Halstead complexity [41], and object-oriented CK metrics [16].

Recent findings in fault-prone module prediction studies center on the effectiveness of the historical information of modules. While *complexity metrics* are designed to measure fault-related complexities, *historical metrics* are designed to measure fault-related version histories.

To find effective historical metrics, researchers have mined software repositories including version control system repositories, fault report management repositories, mailing list archives, and so on. Proposed historical metrics include code-related metrics [90], process-related metrics [38,42,44,64], developer-related metrics [8,80,86,92,96,98,114,115], and so on.

In industry, there are some reports of fault-prone module prediction in practice. Microsoft Corporation built a system CRANE and reported its experiences with this system [18]. Historical metrics including code churn, regression histories, and details of fixes were collected to build failure prediction models in CRANE. The usefulness of the system is reported from empirical evaluation in this paper [18].

Recently, a fault-prone module prediction model has been adopted at Google*. Based on research papers [64,99], a prediction model was built using past fault-fix information. Both industrial prediction models were built with information of

*Bug Prediction at Google, `http://google-engtools.blogspot.com/2011/12/bug-prediction-at-google.html`

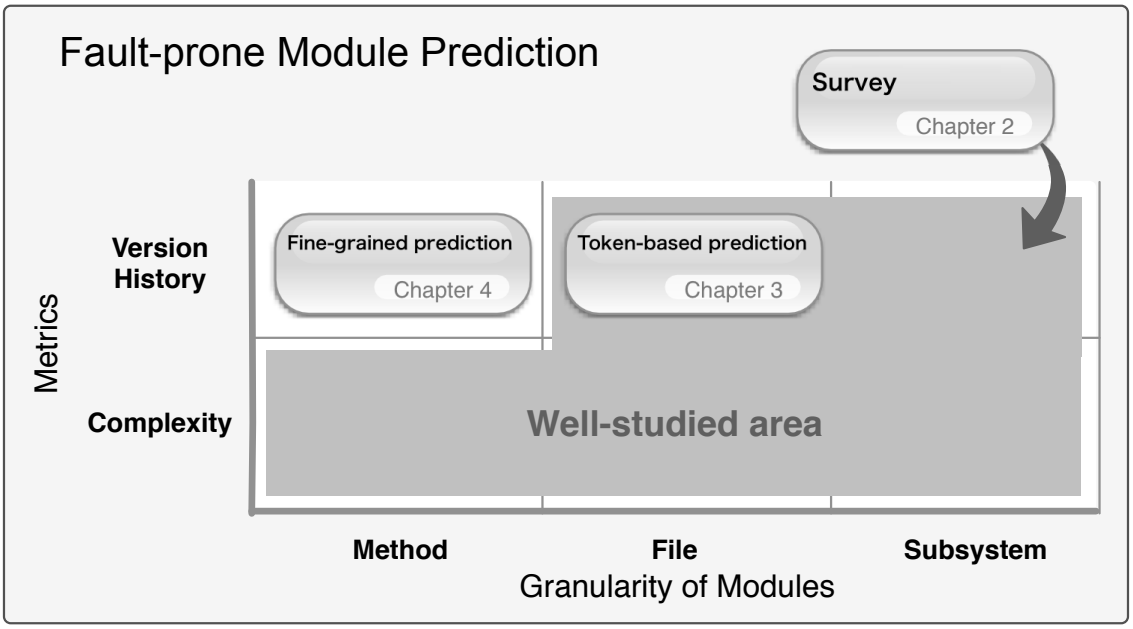


Figure 1.1: Research areas of this dissertation.

version histories, that is, *historical metrics*. In both reports, their effectiveness and understandability are of extreme importance.

In both industry and the academy, fault-prone prediction based on version histories has become the focus of attention. This dissertation summarizes the recent studies and address the difficulties of this hot topic.

1.2 Main Results

Figure 1.1 shows our research areas of this dissertation As shown in Figure 1.1, research areas in fault-prone module prediction studies can be divided into two metric types (complexity and version history) and three module granularity levels (method, file, and subsystem). The gray area is a well-studied research area including all granularity levels for complexity metrics, and file and subsystem levels of historical metrics.

In this dissertation, we present a survey of well-studied area in Chapter 2. Based on this survey, we address two open issues in fault-prone module prediction

studies.

The first issue concerns the difficulty of building models for practical use. Since collecting metrics is a laborious task, there are few usable tools for fault-prone module prediction. In Chapter 3, we propose a text-mining-based prediction model. This is a study of easily applicable prediction models on file level using token metrics, which are historical metrics.

The second challenge is fine-grained prediction. Though historical metrics are considered useful, there are few studies conducting method-level prediction because collecting historical metrics for fine-grained modules is difficult. In Chapter 4, we conduct a first study of method-level prediction using historical metrics. In this model, we collect historical metrics surveyed in Chapter 2 for method-level.

1.2.1 A survey of recent fault-prone module prediction studies

First, we present a survey of recent studies by conducting a systematic review. Papers from 2000 to 2008 in two journals and five conferences are investigated. We classified the studied metrics into eight categories based on measurement targets (code, process, organization, and geography) and version information (present version and previous versions). We clarified which metrics are used frequently. We also clarified that newer historical metrics were studied in industry first, and then widely used in studies in open-source software projects. In addition, granularity levels of prediction models are investigated, and it is revealed that there is no study using well-known historical metrics to build prediction models. We have provided our survey results at `http://www-ise4.ist.osaka-u.ac.jp/survey/`.

1.2.2 Easily applicable prediction models

Collecting metrics is a laborious task. This issue has been a big barrier for adopting fault-prone prediction models in practical use. For example, to collect complexity

metrics one needs to analyze source code, which requires program-language-specific tools, and collecting most historical metrics requires software repository mining, which needs repository-specific mining tools. Preparing these tools is a laborious task.

To tackle this problem, we studied prediction models using a text-mining technique, In our models, the number of tokens in source code is considered a metric. These token metrics are also *historical metrics*, but do not need laborious repository mining tools. The key idea is there may be fault-related tokens, that is, if modules contain particular tokens, which are also seen in past faulty modules, the modules seem to be fault-prone. Since we only have to count the number of tokens in the source code, we do not need specific tools.

Using these simple and large-scale token metrics, we built logistic regression and naive Bayes models. We conducted an empirical study with open-source software projects by comparing our token metrics and a well-know metrics suite including complexity metrics and some historical metrics, thereby achieving higher prediction results. The results imply that our text-mining-based metrics are useful in building practical prediction models. Moreover, text-mining approaches have several desirable features as follows: collecting metrics is independent from program languages, we can treat the flexible granularity of modules, and we do not need semantic information.

1.2.3 Fine-grained prediction with historical metrics

Fine-grained prediction is considered more cost-effective. Though there are many studies reporting the effectiveness of historical metrics, they remain at the file level or at a coarser level. Historical metrics based prediction on fine-grained modules is a big challenge in collecting metrics. Since existing software configuration management systems store file-level version histories, it is difficult to obtain version histories of fine-grained modules from these systems.

To tackle this difficulty, we developed *Historage*, a fine-grained version control

system. Historage is constructed on top of Git, which is a version control system. Making use of the architecture of Git, Historage can control version histories of fine-grained modules including renaming and moving changes. Empirical studies based on some open-source software projects show that Historage is useful practically, and it can track fine-grained module histories including renaming and moving efficiently. A tool to construct Historage is now publicly available at `https://github.com/hdrky/git2historage`.

With this system, we collected historical metrics on method-level and built method-level prediction models. Method-level prediction models are compared with file-level prediction models, which are built with the same historical metrics. Using open-source software projects we compared both prediction models with effort-based evaluation. The results indicated that method-level prediction is more cost-effective than file-level prediction. To the best of our knowledge, this is the first study of fine-grained prediction.

1.3 Overview of the Dissertation

The rest of this dissertation is organized as follows. In Chapter 2, we present a survey of fault-prone module studies. We discuss the technique of extracting past fault information using a version control system and fault report management system for studies with open-source software projects, and the evaluation criteria of prediction results. In addition, we present a systematic review of recent studies.

Chapter 3 presents a study of text-mining-based prediction models. Text-mining-based metrics are compared with well-known complexity metrics and some historical metrics. Empirical evaluation is conducted with open-source software projects.

Chapter 4 presents a fine-grained prediction study. We collect historical metrics related to code, process, and organization to method-level and file-level. Both level prediction models are compared with effort-based evaluation using open-

source software data.

Finally, Chapter 5 concludes the dissertation with a summary and directions for future work.

Chapter 2

Fault-prone Module Prediction

2.1 Mining Past Faults

For fault-prone module prediction, it is essential to obtain actual past fault information. It is necessary to know which modules of particular versions had contained actual faults. In open-source software projects, such information is not easily available because version history information and fault history information are separately stored in different software repositories. To conduct a study with open-source software projects, we have to extract faulty module information by mining these software repositories.

An algorithm proposed by Śliwerski et al. (SZZ algorithm) is well-known for identifying faulty modules and is used in many studies [105]. In this dissertation,

we adopt this algorithm to conduct prediction studies. The SZZ algorithm is designed to identify fault-introducing changes by mining version history repositories and fault report repositories. Faulty modules can be identified by choosing modified modules between fault-introducing changes and fault-fixing changes. With the SZZ algorithm, fault-introducing and fault-fixing changes can be linked with each fault ID in fault reports. This section discusses this algorithm.

First, we need fault reports from fault report management systems, such as Bugzilla and JIRA. When collecting fault reports, `enhancement` severity reports are excluded for Bugzilla, and only `Bug` issue type reports are included for JIRA. From a fault report of fault f_i, where i represents the fault ID, we obtain open date $OD(f_i)$ and changed date $CD(f_i)$.

With collected fault reports, first we identify fault-fixing changes. Fault-fixing changes and fault f_i are linked based on matching fault IDs in commit messages stored in version control repositories. While linking changes and fault f_i, this chapter investigates whether commit dates of the changes are before $CD(f_i)$ or not to remove improper identification of fault-fixing changes.

From each fault-fixing change, then we perform the following procedure to identify faulty modules:

1. Perform the 'diff' command on the same module between the fault-fixing version and the preceding version to locate modified regions on the fault-fixing changes.

2. Examine the initially inserted date of the modified regions using line tracking commands, such as 'git blame' and 'cvs annotate'. If the regions are inserted before $OD(f_i)$, changes creating those regions are identified as fault-introducing changes.

3. Identify a module as faulty if the module contains regions created in the fault-introducing changes and modified in the fault-fixing change.

Figure 2.1 illustrates an example of faulty modules identified with one fault

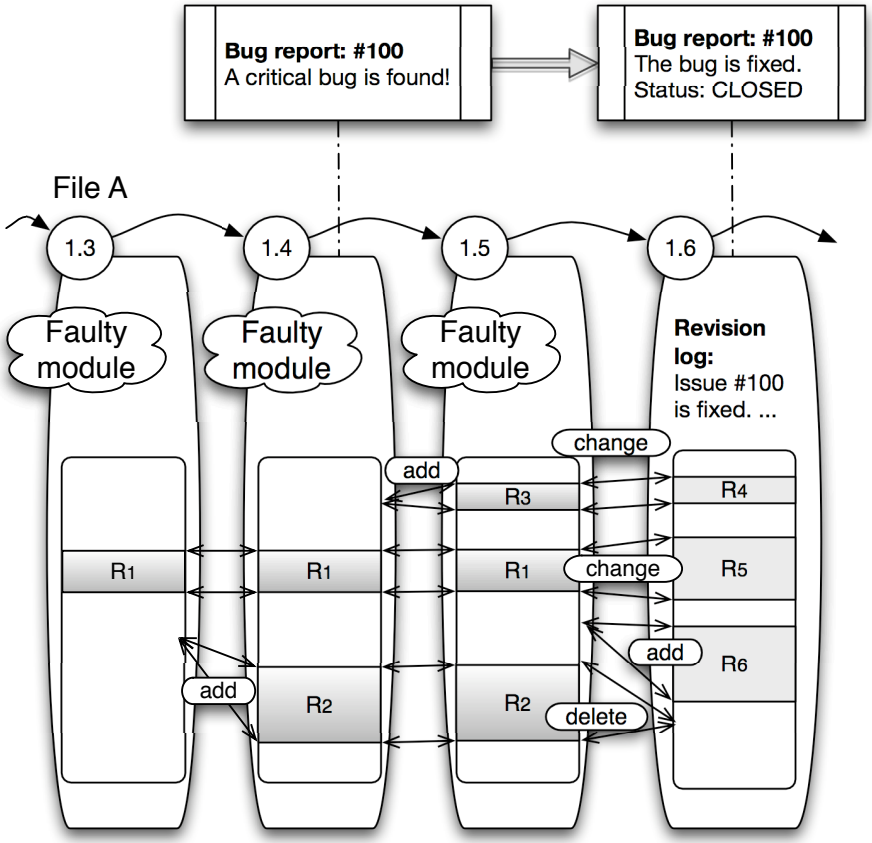


Figure 2.1: Identify faulty modules on one file.

on one file. The revision number of file A is increased from 1.3 to 1.6. When the revision number was 1.4, fault f_{100} was reported. After that, the fault was fixed. Next, we locate faulty modules related to fault f_{100}.

By searching all revision logs, we find a number '100' and a keyword 'fixed' at the log of file A in revision 1.6. We can assume that file A was modified in order to fix fault f_{100}. Therefore, we perform the diff command between revisions 1.5 and 1.6. The diff tool returns a list of regions that differ in the two files. As shown in Figure 2.1, from revision 1.5 to 1.6, region R_3 was changed to region R_4, region R_1 was changed to region R_5, region R_6 was added, and region R_2 was deleted. As a result, regions R_1, R_2, and R_3, which were in revision 1.5 and not in revision 1.6, are recognized to be modified regions. After examining when the modified regions R_1, R_2, and R_3 are inserted into file A, it is revealed that region R_1 and R_2

had been inserted before fault f_{100} was reported. Therefore modified regions R_1 and R_2 can be assumed to be fault-related regions. Since regions R_1 or R_2 spread over revision 1.3 to 1.5, we can identify modules of revision 1.3, 1.4, and 1.5 of file A as faulty modules.

2.2 Historical Metrics

In this section, we discuss recent metrics that can be collected by mining version histories of modules, which we call *historical metrics* in this dissertation. We classify historical metrics based on the target of measurement. We prepare four categories: code-related metrics, process-related metrics, organizational metrics, and geographical metrics.

2.2.1 Code-related Metrics

Nagappan and Ball proposed code churn metrics, which measure the changes made to a module over a development history [90]. They measured *Churned LOC / Total LOC*, *Deleted LOC / Total LOC*, for example. Churned LOC is the sum of added and changed lines of code between a baseline version and a new version of a module. Based on code churn metrics they built statistical regression models, and reported that code churn metrics are highly predictive of defect density performed on Windows Server 2003. These code-related metrics have been basic historical metrics and have been used in many studies [20, 54, 62, 67, 80, 88, 101, 125].

2.2.2 Process-related Metrics

There are many studies of historical metrics related to development processes.

Changes, fixes, past faults, etc. Graves et al. measured the number of changes, the number of past faults, and the average age of modules for predicting faults [38]. They reported the usefulness of such process-related metrics compared with traditional complexity metrics from a telephone switching system study.

These process-related metrics have been used in many studies, for example the number of changes [20, 44, 54, 64, 80, 88, 92, 93, 96, 114] , the number of past faults [30, 67, 93, 125], the number of fault-fix changes [20, 44, 54, 64, 68, 88, 114], and module ages [20, 44, 54, 64, 93, 101, 114].

Metrics from cache-based studies. Several cache-based prediction studies exist [44, 64, 99]. Hassan and Holt, for example, proposed a "Top Ten List," which dynamically creates a list of the top ten subsystems to have a fault [44]. The list is updated as the development progresses based on heuristics including most recently changed, most frequently fault fixed, and most recently fault fixed. Kim et al. [64] and Rahman et al. [99] studied *BugCache* and *FixCache* cache operations. The four heuristics used as cache update policies are as follows:

- *Changed locality:* recently changed modules tend to be faulty.
- *New locality:* recently created modules tend to be faulty.
- *Temporal locality:* recently fault fixed modules tend to be faulty.
- *Spatial locality:* a module recently co-changed with fault-introduced modules tends to be faulty.

The number of co-changes with faulty modules (logical coupling with fault-introducing modules) are also measured in other studies [86, 88].

Process complexity metrics. Hassan proposed complexity metrics of code changes [42]. The metrics are designed to measure the complexity of change process based on the conjecture that a chaotic change process is a good indicator of many project problems. Using different parameters, four *history complexity metrics* are proposed. *History complexity metrics* are better predictors than previous process-related metrics, i.e., prior modifications and prior faults from a study with open-source projects.

2.2.3 Organizational Metrics

Historical metrics related to organization are newer metrics and have been well studied recently.

Number of developers. Graves et al. measured the number of developers [38]. From a case study of a telephone switching system, it is reported that the number of developers did not help in predicting the number of faults. Weyuker et al. also reported that the number of developers is not a major influence on fault-prone module prediction models [114].

Structure of organization. To investigate a corollary of Conway's Law "structure of software system closely matches its organization's communication structure [17]," Nagappan et al. designed organizational metrics, which include the number of engineers, the number of ex-engineers, the number of changes, depth of master ownership, the percentage of organizational contribution, level of organizational ownership, overall organization ownership, and organization intersection factor [92]. They reported that these organizational metrics based failure-prone module prediction models achieved higher precision and recall values compared with models with churn, complexity, coverage, dependencies, and pre-release fault measures from a case study of Windows Vista.

Mockus investigated the relationship between developer-centric metrics of organizational volatility and the probability of customer-reported defects [86]. From a case study of a switching software project, it is reported that the number of leaving developers and the size of the organization have an effect on software quality, but the number of newcomers to the organization is not statistically significant.

Network metrics. Networks between developers and modules are analyzed for predicting failures [80,96,115]. Human factors, such as contributions of developers, coordination and communications are examined based on network metrics, such as centrality, connectivity, and structural holes.

Ownership. A relationship between ownership and quality is also investi-

gated. Bird et al. examined the effects of ownership on Windows Vista and Windows 7 [8]. They measured the number of minor contributors, the number of major contributors, the total number of contributors, and the proportion of ownership for the contributor with the highest proportion of ownership. They found the high ratio of ownership and many major contributors. A few minor contributors are associated with less defects.

Rahman and Devanbu examined the effects of ownership and experience on quality [98]. They conducted a fine-grained study about authorship and ownership of code fragments. They measured the number of lines contributed by an author divided by the number of lines changed to fix a fault as an authorship metric, and defined the authorship of the highest contributor as ownership. From a study of open-source projects, they reported that a high ownership value by a single author is associated with lines changed or deleted to fix faults, and that lack of specialized experience on a particular file is associated with such lines.

2.2.4 Geographical Metrics

Geographical metrics are measured for assessing the risks of distributed development. Bird et al. investigated the locations of engineers who developed binaries [7]. Distribution levels are classified into buildings, cafeterias, campuses, localities, and continents. From a case study of Windows Vista, they clarified that distributed development has little to no effect on post-release failures.

In a study of organizational volatility and its effects on software defects, Mockus measured the number of sites that modified the file and investigated the distribution of mentors and developers [86]. It is reported that the geographic distribution has a negative impact on software quality from a case study of a large switching software.

Table 2.1: Prediction result matrix

		Predicted	
		not fault prone	fault prone
Actual	not faulty	True negative (TN)	False positive (FP)
	faulty	False negative (FN)	True positive (TP)

2.3 Evaluation Criteria

2.3.1 Accuracy, Recall, Precision, and F_1

For prediction evaluation, there are well-known measures: accuracy, recall, precision, and F_1.

Table 2.1 shows a legend of the prediction result matrix. A true negative (TN) shows the number of modules that are predicted as not fault prone, and are actually not faulty. A false positive (FP) shows the number of modules that are predicted as fault prone, but are actually not faulty. On the contrary, a false negative (FN) shows the number of modules that are predicted as not fault prone, but are actually faulty. Finally, a true positive (TP) shows the number of modules that are predicted as fault prone which are actually faulty.

The accuracy ratio shows the ratio of correctly predicted modules to entire modules and is defined as follows:

$$\text{Accuracy} = \frac{TP + TN}{TN + FP + FN + TP}$$

Recall represents the ratio of modules correctly predicted as fault prone to the entire number of faulty modules. Recall is defined as follows:

$$\text{Recall} = \frac{TP}{TP + FN}$$

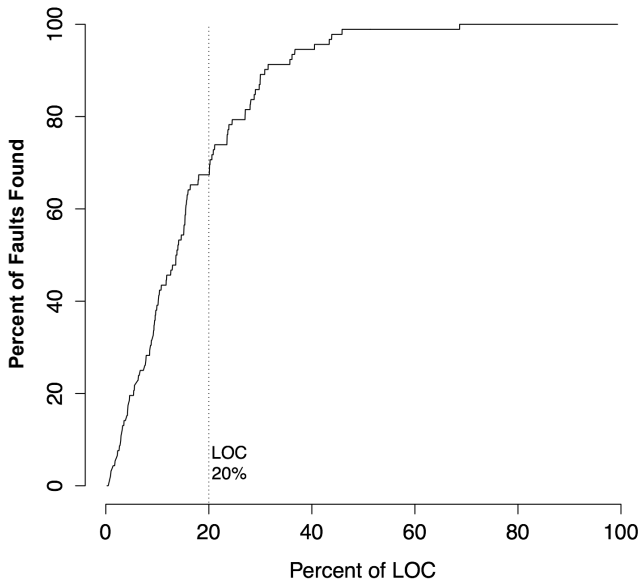


Figure 2.2: Cost-effectiveness curve.

Precision is the ratio of modules correctly predicted as fault prone to the number of the entire modules predicted fault prone. Precision is defined as follows:

$$\text{Precision} = \frac{TP}{TP + FP}$$

F_1 is used to combine recall and precision. F_1 is defined as follows:

$$F_1 = \frac{2 \times \text{recall} \times \text{precision}}{\text{recall} + \text{precision}}$$

2.3.2 Effort-based Evaluation

Recent studies take into account the effort of quality assurance activities, such as inspecting and testing predicted modules for evaluating prediction models [2, 69, 79, 83, 99]. These effort-based evaluations should be desirable for practical use of the prediction results. The key idea of evaluation that takes in to account effort is that it discriminates the cost of inspecting and testing for each module.

Arisholm et al. pointed out that the cost of such quality assurance activities on a module is roughly proportional to the size of the module [2].

Figure 2.2 illustrates an example of a cost-effectiveness curve. This curve shows that as the quality assurance cost increases, the percentage of found faults increases. The quality assurance cost is represented as the percentage of investigated LOC of software. When we inspect or test modules, the modules are ordered by fault-proneness. If we find most faults when we investigate the small percentage of the entire LOC, it is cost-effective. In Figure 2.2, a dotted line represents an example cutoff line set to LOC 20%. If cost-effectiveness curves cross the upper part of this cutoff line, it is better for the cost of inspection and testing.

2.4 A Systematic Review of Recent Studies

This section presents a systematic review of fault-prone module prediction studies. *Systematic review* is a repeatable method for identifying relevant studies to answer specific research questions [76]. Some papers reported the effectiveness of systematic reviews in software engineering [65, 66, 76].

As introduced in Section 2.2, there are many studies proposing new historical metrics. Possible reasons for this might be that many publicly available software data have been used recently. In addition, there are also easily available tools to build prediction models, such as WEKA [40] and R [107].

Catal and Diri have reported the first result of a systematic review of fault-prone module prediction in 2009 [13]. They investigated the studies between 1990 and 2007, and analyzed the types of datasets, prediction methods, and granularities of metrics. They reported that use of public datasets had increased, as did models based on machine learning techniques. On the point of metrics granularity, it is reported that though traditional complexity metrics targeted class-level prediction, there is an increase in file-level prediction.

Compared to the previous study, we concentrated more on the details of

metrics, especially newer historical metrics. As discussed in Section 2.2, many historical metrics have been proposed recently. We investigate the recent studies to clarify the recent trend of fault-prone module prediction studies.

2.4.1 Review Process

Research Questions

We prepared the following two research questions in a systematic review of recent fault-prone module prediction studies:

RQ1: What kinds of metrics have been proposed and used so far?

RQ2: Is there a trend in using new metrics?

RQ3: Which granularity of the prediction model is well studied?

Paper Selection

Table 2.2: Targeted journals and conferences

Journal	IEEE Transactions on Software Engineering Empirical Software Engineering
Conference	International Conference on Software Engineering Joint Meeting of European Software Engineering Conference and Symposium on Foundations of Software Engineering International Symposium on Foundations of Software Engineering International Conference on Software Maintenance Working Conference on Mining Software Repositories

As a paper selection method, adopting an automated keyword search using search engines is possible. However, Kitchenham et al. reported that though

broad automated searches find more studies than manually restricted searches, they may be of poor quality [66]. Therefore, we searched papers manually from two journals and five conferences as shown in Table 2.2.

The targets of this review are papers of fault-prone module prediction studies. Since not all papers have explicit titles, it is difficult to select papers using only simple keywords. Using the following criteria, we identified related papers.

First, we selected papers using the following inclusion criteria:

IC1: In the title or abstract, there are fault-related terms, such as fault, defect, bug, failure, and error.

IC2: The paper discusses the quality or dependability of software.

Next, we removed inappropriate papers using the following exclusion criteria:

EC1: The paper studies testing and inspection to detect faults.

EC2: The paper discusses repository mining techniques, for example, a technique of identifying commits related to faults.

EC3: The paper targets the process of fault fixing including prediction of such process.

EC4: The paper investigates actual faults using empirical studies.

Design of the Analysis

Based on the two research questions **RQ1** and **RQ2**, we will analyze selected papers. Unlike in the previous systematic review [13], we will concentrate on detailed metrics information.

To collect traditional complexity metrics, source code of the targeting version is needed for analysis. For historical metrics, software repositories needed to be mined, that is, we need to analyze the histories of some targets. To classify the studied metrics, we prepared the following classification items:

Target: code, process, organization, and geography.

Version: present version, and previous versions.

With these classification items, we will classify the studied metrics into eight categories (4×2).

Threats to Validity

Here, we discuss threats to validity based on Yin's classification [100, 120].

Construct validity (To what extent do the operational measures that are studied really represent what the researchers have in mind?)

Since we selected papers manually, there may be missing or inappropriate papers. However, we carefully searched papers using explicit inclusion/exclusion criteria shown in Section 2.4.1.

Internal validity (There is a risk that the investigated factor is also affected by a third factor when causal relations are examined.)

We did not examine causal relations.

External validity (To what extent is it possible to generalize the findings?)

We limited the papers in journals and conference proceedings presented in Table 2.2 by choosing the most well-known, highest quality, and well-cited journals and conferences in software engineering, empirical software engineering, software maintenance, and software repository mining, which are also important keywords in recent fault-prone module prediction studies.

Reliability (To what extent are the data and the analysis dependent on the specific researchers?)

Since we clarified our inclusion and exclusion criteria, our systematic review process is repeatable, and our results are reliable.

2.4.2 Results

Selected Papers

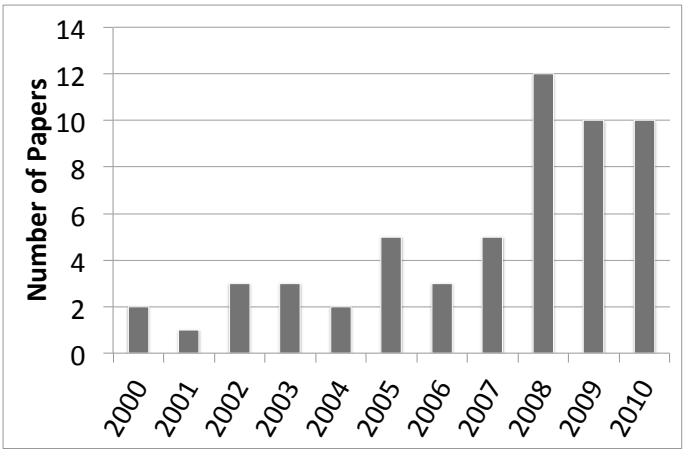


Figure 2.3: Number of papers per year.

We selected 26 papers from journals and 30 papers from conferences. Figure 2.3 shows the number of papers per year from 2000 to 2010. We can see that papers have increased recently, especially from 2005.

This result is similar to the result reported in the previous review, although Catal and Diri investigated different journals [13]. They also reported that there is an increase in the use of public datasets from 2005. One of the reasons for this increase, they insisted, was that the PROMISE repository [10], which is intended to share software development data to enable repeatable experiments, began in 2005.

In addition to the PROMISE repository, many other open-source software repositories have been made easily available recently, including version control systems, fault report management systems, and mailing archives. These environments seem to attract many researchers to empirical fault-prone module prediction studies.

Metrics

Table 2.3 presents our classification of studied metrics. For each metrics, papers using it are shown. As introduced in Section 2.4.1, we classified metrics into eight regions based on measurement targets and required version information. We numbered regions from (1) to (8). As shown in Table 2.3, there are various metrics.

Transition of Studied Metrics

To see the trend in proposing new metrics, we investigated the number of papers per year for each region shown in Table 2.3. Figure 2.4 presents this result. If papers used certain metrics, which can be categorized into different regions, we added them for corresponding regions. The regions (3) and (4), and (5) and (6) are added up. Since region (1) metrics had been used continuously, it is represented as a line chart. Other regions are represented as bar charts.

Some findings are as follows:

Metrics in regions (2) to (8). These regions are newer regions compared to traditional complexity metrics belonging to region (1). Metrics belonging to these regions are required to extract information by mining software repositories. They have been widely used from 2005.

Historical metrics: code-related (2) and process-related (4). Metrics in region (2) and (4) are needed to analyze version control system repositories, and fault report management system repositories.

These metrics were first used in industrial papers in 2000, and widely used from 2005 in studies with both industry and open-source software.

Historical metrics: organization (6) and geography (8). Simple organizational metrics was used in 2000, and from 2008 many papers exists using several organization metrics. From 2009 geography metrics have been studied.

Table 2.3: Classification of studied metrics

	Present Version	Previous Versions
Code	**Region (1)**	**Region (2)**
	LOC*	Churn
	Operators [20,23,54,56–58,68,72,74,82,91,112]	[20,54,62,67,80,88,90,101,125]
	McCabe [54,62,72,74,91,95,112,125]	Method call history [104]
	Halstead [23,72,74,82,112]	Warning history [101]
	CK [20,29,39,52,54,103,106,122]	
	Method call relation [91,104]	
	Reuse [87,97]	
	Warning [11,89,101]	
	Directory [62]	
	Concerns [28,32]	
	Tokens [45,46,62,84,85]	
	Dependency graph [124]	
Process	**Region (3)**	**Region (4)**
	Programing language [93,114]	Past faults [30,38,67,93,125]
	Commit log [62]	Age [20,38,44,54,64,93,101,114]
	Commit date [62]	Changes
		[20,38,44,54,64,80,88,92,93,96,114]
		Fixes [20,44,54,64,68,88,114]
		Refactorings [20,54,88]
		Logical Couplings [64,86,88]
		Change complexity [20,42,86]
		Test cases [67]
Organization	**Region (5)**	**Region (6)**
	Author [62]	Authors [20,38,80,88,92,96,114,125]
	Organization [38]	Organizations [92]
		Author & code [80,92,96,115]
		Social network [14,80,86,92,96,115]
		New/Ex authors [86,92,114]
Geography	**Region (7)**	**Region (8)**
	None	Locations [7,80,86]

*(LOC) [12,23,31,52,54,56–58,62,68,70–72,74,82,91,93,112,114,121]

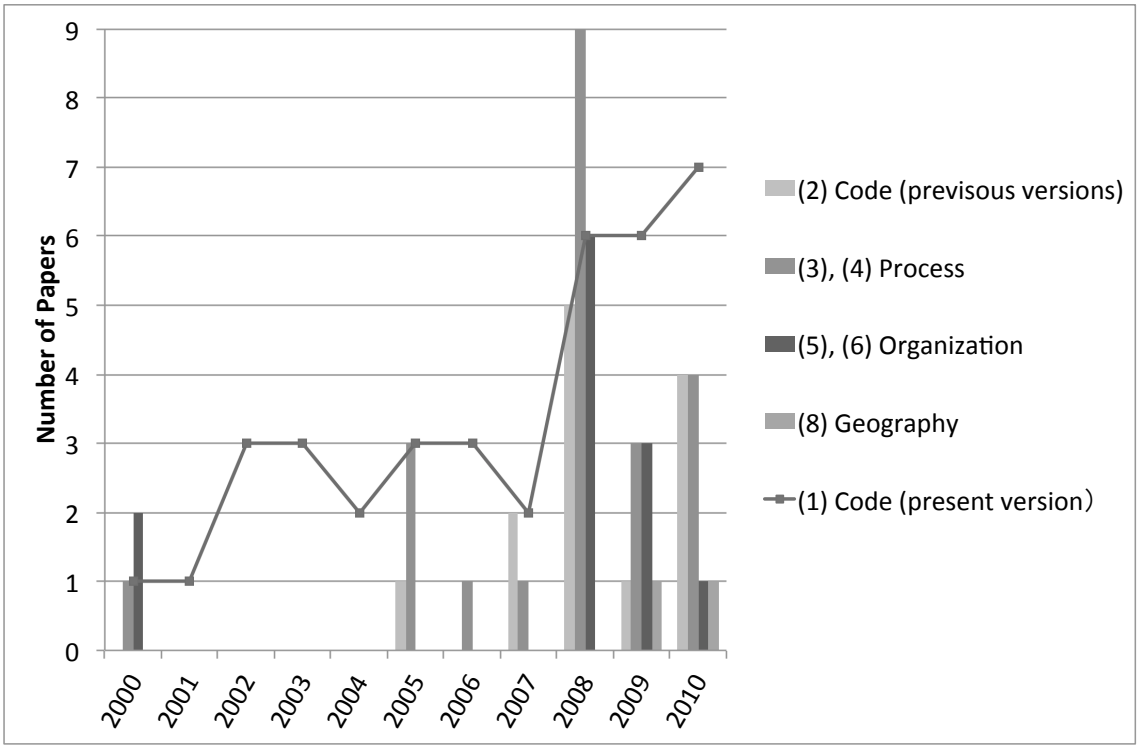


Figure 2.4: Transition of studied metrics.

However there are not many papers using these metrics. Compared to code-related and process-related historical metrics, collecting these metrics from open-source software projects has not been easy because there is no explicit public repository.

New metrics in region (1). LOC and complexity metrics, such as McCabe, Halstead, and CK have been used for a long time. In addition to these traditional metrics, there are also new metrics proposed in region (1) as presented in Table 2.3. These metrics have also been used.

Sources of Studied Data

Figure 2.5 shows the number of papers per year classified by sources of studied data. In certain early years, there are only papers using proprietary data in industry. From 2005, public data have been widely used. Public data include

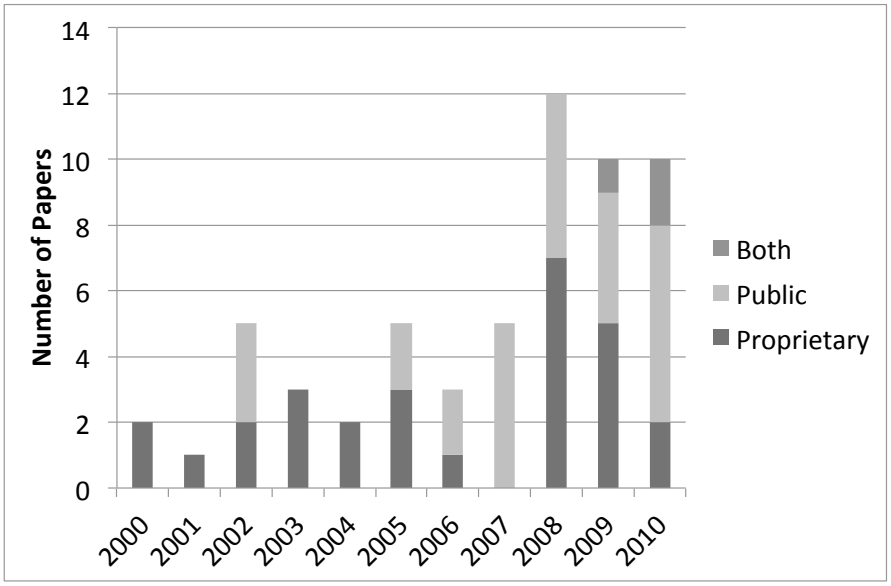
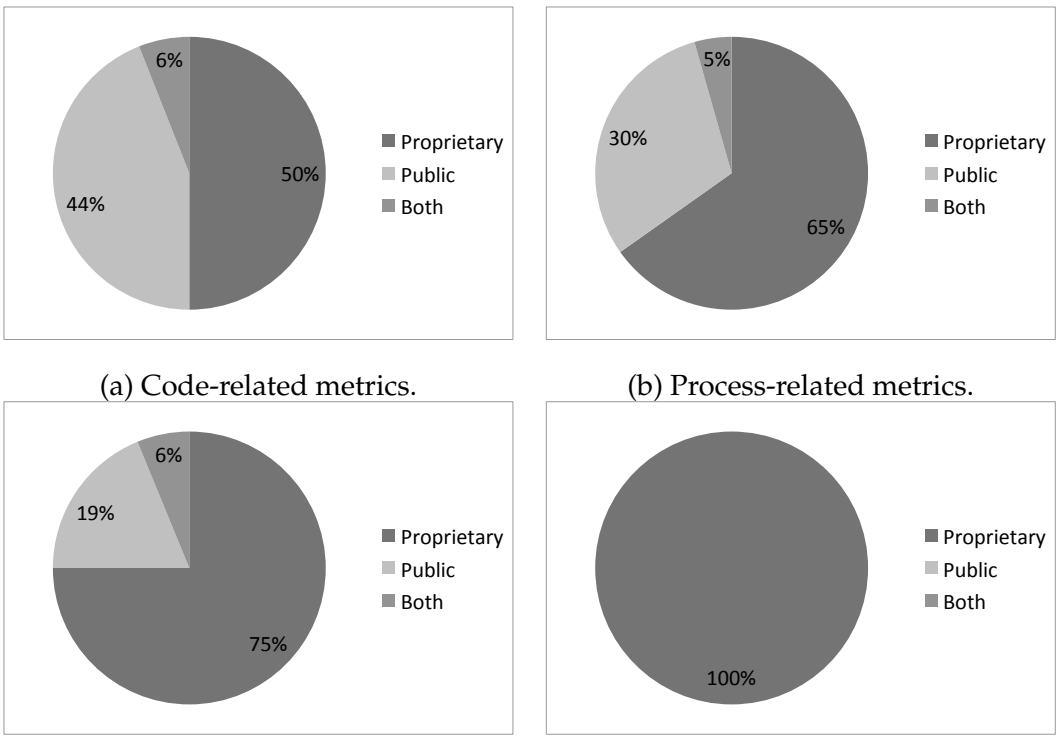


Figure 2.5: Number of papers per year classified by data.

open-source software data and publicly available industrial data. More recently some papers have used both proprietary and public data.

To generalize the findings from empirical studies, we should study different types of projects. This is because metrics may be effective for particular projects, and may be not effective for other projects. For example, Bird et al. reported that there is no impact of distributed development on software quality [7], but Mockus showed that geographic distribution has a negative impact on software quality [86].

To see how metrics have been studied, we classified papers by sources of data for four metrics categories. Figure 2.6 shows the proportion of studied data sources for (a) code-related metrics, (b) process-related metrics, (c) organizational metrics, and (d) geographical metrics. It is desirable for metrics to have been studied where both proprietary and public data for generalization. As seen in Figure 2.6 (a), nearly half of the studies applied code-related metrics to public data. However, not many studies applied the other, newer metrics to public data. No public data study use geographical metrics. From this analysis, we found that newer metrics have not been studied enough with public data.



(a) Code-related metrics.

(b) Process-related metrics.

(c) Organizational metrics.

(d) Geographical metrics.

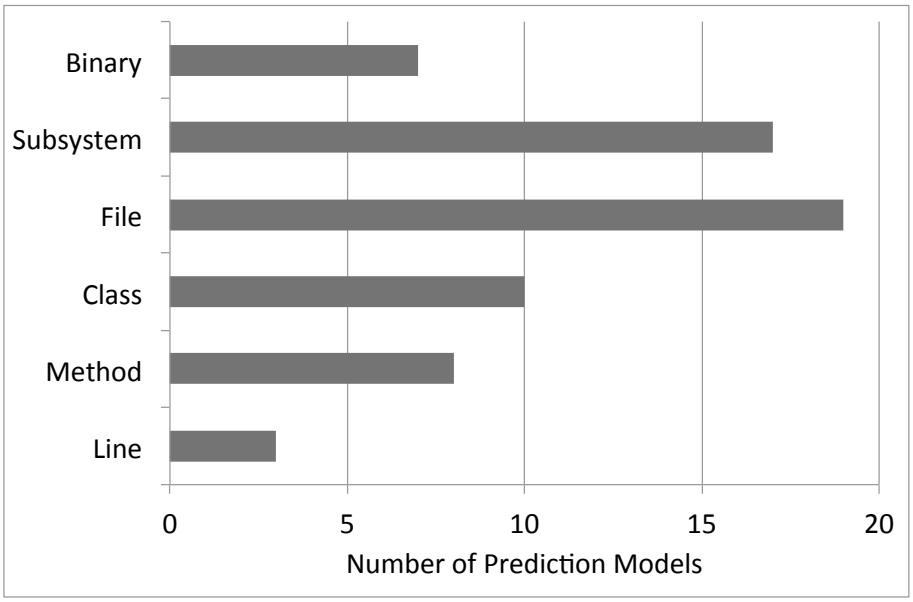
Figure 2.6: Distribution of data sources for each metric category.

Granularity

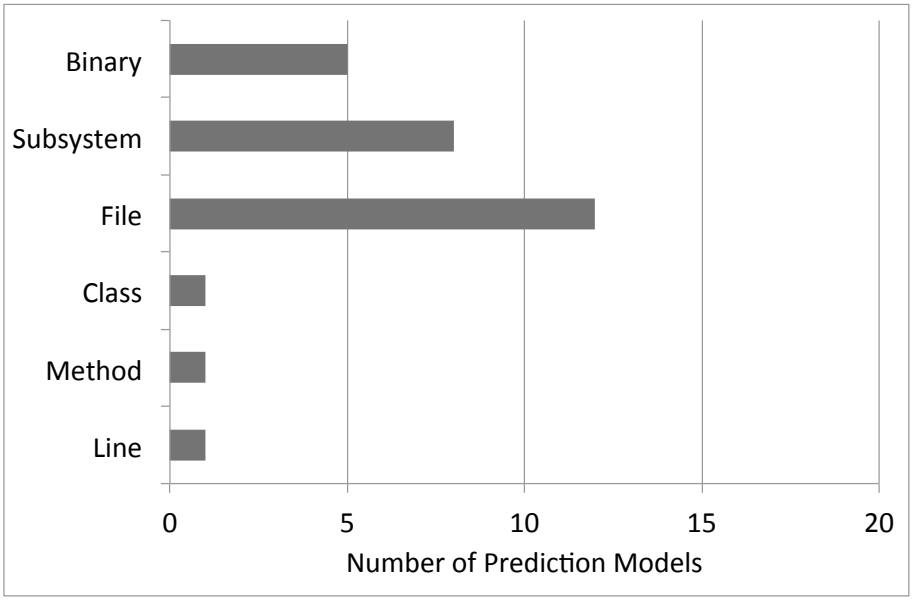
Figure 2.7 shows the number of prediction models for each granularity of the prediction model. Figure 2.7 (a) shows a summary of all prediction models, and Figure 2.7 (b) shows the summary of prediction models limited to models using historical metrics. There are various levels of granularity. A binary comprises several files compiled together, and a subsystem represents a module with several related files.

As seen in Figure 2.7 (a), file-level and subsystem-level have been widely studied. In addition, some studies targeting binary-level, class-level, and method-level exist. This is because collecting traditional metrics from fine-grained level to coarse-grained level is possible.

However, if we select only prediction models using historical metrics, there are only a few papers targeting class level and method level prediction, as shown



(a) All prediction models



(b) Prediction models using historical metrics

Figure 2.7: Granularity of prediction models.

in Figure 2.7 (b). Kim et al. targeted faulty Java methods using a cache-based approach [64]. Mizuno and Kikuno predicted fault-prone Java methods using a spam-filtering-based approach [85]. However, few studies predict fine-grained modules using well-known historical metrics. This is because it is not easy to collect proposed historical metrics on methods since version control repositories control file histories, but not method histories.

2.4.3 Summary

We conducted a systematic review of recent fault-prone module prediction studies focusing on metrics. Papers were selected from two journals and five conferences from 2000 to 2010. It was revealed that the number of papers has been increasing recently. From our review results, we can now answer the research questions. We have provided our survey results at `http://www-ise4.ist.osaka-u.ac.jp/survey/`.

RQ1: What kinds of metrics have been proposed and used so far?

Based on the differences of measurement targets and required version information, we classified studied metrics into the eight categories shown in Table 2.3. Various proposed metrics are used in fault-prone module prediction.

For historical metrics, which require previous version information, churn is a basic metric in code-related historical metrics. There are many kinds of historical metrics in process-related and organizational metrics.

RQ2: Is there a trend in using new metrics?

Historical metrics tend to be proposed first in proprietary papers, and then used in papers with public data such as open-source software projects. Code-related and process-related historical metrics have been popular since 2005, and organization metrics have been used since 2008. Geography metrics have been studied from 2009. In addition, there are some code-related metrics regarding the present version. Though we made four categories for measurement targets, new categories may exist in the future.

Code-related metrics have been well studied with both proprietary and public data. Though some studies applied process-related and organizational metrics to public data, the ratio of public data in studied sources is low. Geographical metrics have been studied only with proprietary data. For generalization of the effectiveness of metrics, metrics must be studied with public data.

RQ3: Which granularity of the prediction model is well studied?

Several granularities of prediction models exist. The most studied prediction granularity is at the file level. Including prediction models using traditional metrics, some studies exist on method level. However, there are a few studies building method-level prediction using historical metrics. Most studies using historical metrics built file-level or more coarse-grained prediction models.

2.5 Open Issues

This chapter surveys recent fault-prone module prediction studies. To discuss open issues, we introduce an interesting report.

At the conference of the Joint Meeting of European Software Engineering Conference and Symposium on Foundations of Software Engineering in 2011, which is one of top conferences in software engineering, there was a forum of PhD working groups to conduct short surveys on certain software engineering topics by interviewing conference participants and researching the field[*]. A group of "bug prediction models" was one of these working groups. They asked participants in the conference about the main open challenges in building fault prediction models[†]. There were 27 participants including five from industry and 22 from academia.

In the survey, working group asked: "What are the barriers for the adoption of fault-prone prediction among practitioners?" From both industry and academia, *models not available as tools* was selected by the greatest number of people. This

[*] http://pwg.sed.hu/

[†] http://pwg.sed.hu/node/2

problem has been a well-known one, but has not been solved yet. If there are enough data, building prediction models is not very difficult because there are some good tools, such as WEKA [40] and R [107]. We think that the **laboriousness of collecting metrics** is a more important problem.

In the same survey, the participants were also asked about the future directions of fault-prone module prediction. People from academia tend to consider generalization as a future direction, such as *models adaptable to different systems* and *improvement of benchmarks for comparison*. On the other hand, industrial participants seek practical directions, such as *models that deal with incomplete data* and *fine-grained prediction*. The first direction has broader problems, and requires robust prediction models, mining incomplete software data techniques, and so on. In this dissertation, we are interested in the second problem, **fine-grained prediction**.

2.5.1 Laboriousness of Collecting Metrics

Collecting metrics require analyzing source code, and mining software repositories. Since there are various program languages and software repositories, we need to implement program-language-specific or software-repository-specific tools. This task is very laborious, and it has been a barrier to adopting prediction models for practical use.

2.5.2 Fine-grained Prediction with Historical Metrics

Fault-prone module prediction on fine-grained modules is a desirable future direction. Such prediction is expected to be effective in the cost of quality assurance activities. Effort-based evaluation considers the required effort to find faults. If we can find the most faults while investigating the small percentage of the entire software source code, such prediction models should be desirable. Kamei et al. clarified that file-level prediction models are more effective than package-level, which has more coarse-grained modules than file-level, on Java software

projects [54]. From this result, we can infer that method-level prediction is more cost-effective than file-level prediction because the methods have a finer granularity than files.

As discussed in Section 2.4.2, there are a few studies of fine-grained prediction. Fine-grained prediction using well-know historical metrics is a big challenge.

Chapter 3

Text-mining-based Prediction

3.1 Motivations

As presented in Chapter 2, there are many fault-prone module prediction studies, and many studies have collected various metrics to build prediction models. To collect such metrics, we have to analyze source code and/or software repositories. As discussed in Section 2.5.1, these tasks for collecting metrics are laborious. In addition, selecting metrics for building models is also a tedious task. Several studies suggest that there is no best subset of metrics that enables a fault-prone module predictor to perform a perfect prediction [82, 91]. Nagappan et al. advised not using complexity metrics without validating them for a target project [91]. Men-

zies et al. concluded that if there is a metrics subset appropriate for a particular domain, all available metrics could be used to build prediction models [82]. They also insisted that how metrics are used to build models is much more important than which particular metrics is used. However, it is uncertain how many metrics should be collected.

To mitigate these difficulties in preparing metrics, Mizuno and Kikuno proposed a spam-filtering-based prediction model [84, 85]. In spam filtering, a classifier, which is a prediction model, is trained with large-scale text features from both spam and non-spam mails. Then, an incoming mail is classified into either spam or non-spam. The Bayesian spam filtering technique was first introduced in 1998 as a scholarly publication by Sahami et al. [102]. The model is a well-studied Bayesian model. Since the usefulness of Bayesian theory for spam filtering has been recognized recently, most spam filtering tools implement Bayesian theories. Consequently, the accuracy of spam prediction has improved dramatically. This technique has been studied to meet the needs of the spam mail problem, that is, spam filtering systems should be able to automatically adapt to the variable characteristics of spam mails. Moreover, the systems need to be personalized to the user's needs. This framework is based on the fact that spam e-mails usually include particular patterns of words or sentences.

From the viewpoint of source code, similar situations usually occur in faulty software modules. That is, similar faults may occur in similar contexts. Inspired by the spam filtering technique, the previous study tried to apply text-mining techniques to software modules. In fault-prone module prediction, Mizuno and Kikuno treat a software module as an e-mail message, and classify all software modules into either fault-prone (FP) or non-fault-prone (NFP). This approach means that the numbers of particular text features in a module are regarded as one of its metrics. With our approach, we need neither language-specific semantic analysis and storage-specific repository mining, so we can easily apply the prediction models to various software projects.

The previous study [85] proposed a prediction model with a spam-filtering framework that uses Bayesian models, and presented a comparative study with traditional prediction models only on a survey of research papers. To clarify the effectiveness of a text mining approach on fault-prone module prediction, this chapter presents a more generic study. Using text mining based metrics, we built logistic regression models as well as Bayesian models, and conducted a fair comparison with traditional metrics-based models.

3.2 Building Models with Text Features

3.2.1 Feature Extraction

Before extracting text features we remove comments in source code. This means that every token except for comment can be treated as a feature. The number of text features is counted per module. For replication of the experiment, we used the WEKA data mining toolkit [40]. To extract features properly, every variable, method name, function name, keyword, and operator connecting without a space or tab is separated.

Since using all features requires much time and memory, we limit the kinds of text features to 5,000 in this experiment*. This option is intended to discard other, less useful features. These text features can be regarded as one of the metrics **Num(**$token_i$**)**, where $token_i$ represents the ith text features. Text-feature metrics are very large-scale compared with other traditional metrics.

3.2.2 Prediction Models

Regarding text features as metrics **Num(**$token_i$**)**, it is easy to build well-known prediction models. In this dissertation, we built the following two models.

*java weka.filters.unsupervised.attribute.StringToWordVector -C -W 5000

Logistic Regression Model

The multivariate logistic regression model is represented as follows:

$$f(m_1, m_2, ..., m_n) = \frac{e^{C_0+C_1m_1+C_2m_2+,,,+C_nm_n}}{1 + e^{C_0+C_1m_1+C_2m_2+,,,+C_nm_n}}$$

where m_i is the value of the metric in a module. If $f(m_1, m_2, ..., m_n) > 0.5$, the module is classified as FP, otherwise, as NFP.

Naive Bayes Model

The naive Bayes model classifies a module as follows:

$$\underset{C \in \{FP,NFP\}}{\operatorname{argmax}} P(C) \prod_{i=1}^{n} P(m_i|C)$$

where C is a class, which is FP or NFP, and $P(C)$ is the prior probability of class C and $P(m_i|C)$ is the conditional probability of a metric m_i given class C. The previous study reported that prediction models using naive Bayes achieved standout good results compared with OneR, J48 in their experiment using the WEKA [85].

3.3 Experimental Setup

To show the effectiveness of using large-scale text features, we conducted a fair comparison with traditional metrics based models. In the experiments, we targeted Java programming language.

3.3.1 Compared Metrics

To show the effectiveness of our proposal, we compared large-scale text features with traditional metrics in experiments. We collected the CK metrics suite [16]. This metrics suite is collected with a tool developed by Higo et al. [51]. In addition, we collected the code churn, previous fix changes, and the LOC of each module. Table 3.1 shows all the collected metrics in this dissertation.

Table 3.1: Compared metrics

Metrics	Description	
LOC	Lines of code	
WMC	CK metrics suite	Weighted methods per class
DIT		Depth of inheritance tree
NOC		Number of children
CBO		Coupling between object classes
RFC		Response for class
LCOM		Lack of cohesion on methods
ADD	Churn metrics	# of added lines
CHG		# of changed lines
FIX	Fixed or not	

3.3.2 Target Projects

For the experiment, we selected open-source software projects in which we can track faults. For this reason, we targeted five projects in Eclipse[†]: Business Intelligence and Reporting Tools (BIRT), Eclipse (ECLP), Eclipse Modeling Project (MODE), the Test and Performance Tools Platform (TPTP), and the Eclipse Web Tools Platform (WTP). Table 3.2 shows the information of each target project. These projects are written in Java language, and revisions are maintained by CVS. The source repository of CVS used in this study was uploaded on the Eclipse project Web site, and was obtained on the 6th of January, 2009. We treated a Java file in each revision as a software module.

We also obtained fault reports from the fault report databases of each project. We extracted fault reports under the following conditions. The type of these faults is "bugs"; therefore, these faults do not include any enhancements or functional patches. The status of faults is "resolved", "verified", or "closed", and the resolution of faults is "fixed". This means that the collected faults have already been

[†]http://www.eclipse.org/

Table 3.2: Target project information

	Release 1			Release 2		
Project	Release	(Date)	Total LOC	Release	(Date)	Total LOC
BIRT	2.1	(2006-06-30)	768K	2.2.0	(2007-06-29)	1,135K
ECLP	3.2	(2006-06-29)	2,617K	3.3	(2007-06-28)	2,588K
MODE	Callisto	(2006-06-30)	1,730K	Europa	(2007-06-29)	2,191K
TPTP	4.2.0	(2006-06-30)	718K	4.4.0	(2007-06-29)	722K
WTP	1.5	(2006-06-30)	1,432K	2.0	(2007-06-29)	2,338K

Table 3.3: Result of module collection

	Release 1			Release 2		
Project	# of faulty modules		Total modules	# of faulty modules		Total modules
BIRT	227	(8.6%)	2,645	291	(8.2%)	3,563
ECLP	376	(4.5%)	8,429	236	(3.2%)	7,351
MODE	36	(0.6%)	5,649	44	(0.6%)	7,049
TPTP	792	(28.2%)	2,811	366	(15.8%)	2,310
WTP	183	(2.5%)	7,336	133	(1.7%)	7,996

fixed and have been resolved, and thus fixed revisions should be included in the entire repository. The severity of the faults is either "BLOCKER", "CRITICAL", or "MAJOR" so as not to include trivial faults. Herraiz et al. categorized these severity categories as important and the others without ENHANCEMENT as non-important [49]. Using our faulty modules collection tool, we collected both faulty and not faulty modules from these five projects. The result of the module collection is shown in Table 3.3.

3.3.3 Design of Experiments

Using the collected data shown in Table 3.3, we conducted the following two experiments.

1. Ten-fold cross validation

   For 10-fold cross validation, we used release 1 data only. The 10-fold cross validation can show relatively fair results for a given data set. However, it cannot take into account important features such as the order of building modules.

2. Prediction on post release

   Here, we used both release 1 data and release 2 data. Fault-prone modules are predicted on release 2 data using prediction models trained with release 1 data. On the release 2 data, we evaluate the prediction performance.

To show the effectiveness of using large-scale text features, the same two experiments were also conducted with well-known software metrics as shown in Table 3.1. Generally speaking, the performance of fault-prone module prediction varies according to the combination of these metrics used in a prediction model. In order to find the best metrics subset for the release 1 data, we prepared all ($= 2^{10} = 1,024$) combinations of the metrics shown in Table 3.1. Then, we performed the 10-fold cross validation for each combination, and obtained the best combination with the highest evaluation measurement. This procedure is iterated for all projects. Once we get the best combination of compared features, we built a prediction model using the best combination of metrics and the release 1 data. Next, we apply the built model to the release 2 data.

3.4 Results

3.4.1 Ten-fold Cross Validation

Table 3.4 shows the best subset of metrics in each project on naive Bayes models. As described in Section 3.3.3, each subset of metrics achieved the highest F_1 value with a naive Bayes model in each project. Similarly, the best subset of metrics for logistic regression models in each project and the regression coefficient of

Table 3.4: The best subset of metrics for naive Bayes models

Project	Subset of metrics
BIRT	LOC, CHG, DIT, CBO, NOC
ECLP	FIX, CHG, WMC, LCOM
MODE	CHG, CBO, RFC
TPTP	LOC, FIX, ADD, WMC, DIT, CBO, LCOM, RFC
WTP	FIX, WMC, DIT, LCOM, NOC

Table 3.5: Regression coefficients of selected metrics for logistic regression models

Project	LOC	FIX	ADD	CHG	WMC	DIT	CBO	LCOM	RFC	NOC
BIRT	0.0004	0.920				-0.079	0.008	-0.0002	0.0006	-0.114
ECLP	-0.001	1.292	-0.005	-0.0003	0.012	-0.053	0.001			
MODE				0.002						
TPTP	-0.001	0.776		0.005	0.001	0.030	0.002	-0.001	0.002	-0.025
WTP		1.737			0.007	-0.053			0.002	

each selected metrics are seen in Table 3.5. In Table 3.5, a blank represents a corresponding metrics not used in a corresponding project. For example, in project WTP, the best subset of metrics for logistic regression models are "FIX", "WMC", "DIT", and "RFC". Each value in Table 3.5 is an estimated regression coefficient value. The larger the absolute value of the regression coefficient, the stronger the impact of the metrics on fault-prone modules prediction. The used metrics sets are different from each other. From the viewpoint of the regression coefficient value, FIX and DIT are relatively high in the target projects.

Table 3.6 presents the top three text features ordered by positive and negative regression coefficient values of logistic regression models in each project. A positive regression coefficient indicates an increase in the probability of FP, while a negative regression coefficient indicates a decrease in the FP probability. For example, in project BIRT, if there is "pointer" and/or "getObject" in the source code of a module, the FP probability of the module is high. If there is "excel"

Table 3.6: Top three text features ordered by positive and negative regression coefficient values of logistic regression models

Project	Positive regression coefficient		Negative regression coefficient	
	Feature	Value	Feature	Value
BIRT	pointer	79.2	excel	-665.1
	getObject	73.9	em	-190.0
	package	71.8	Member	-148.7
ECLP	NavigatorPlugin	21.6	PerformanceTestSetup	-32.6
	launchConfigurations	14.3	AbstractUIPlugin	-18.0
	isBaseType	13.0	removeSelectionChangedListener	-16.8
MODE	org/uml2/2	11.7	0/UML	-12.0
	g1	4.5	getFactory	-6.7
	Factory	3.9	V	-5.3
TPTP	LF	75.6	atts	-153.4
	setTestInvocationId	52.3	scenario	-43.0
	createPlatformResourceURL	49.6	OK_STATUS	-39.5
WTP	Missing	10.0	ArrayCreation	-31.8
	extra	9.5	FieldAccess	-31.8
	COMPILATION_UNIT	8.7	SimpleName	-31.8

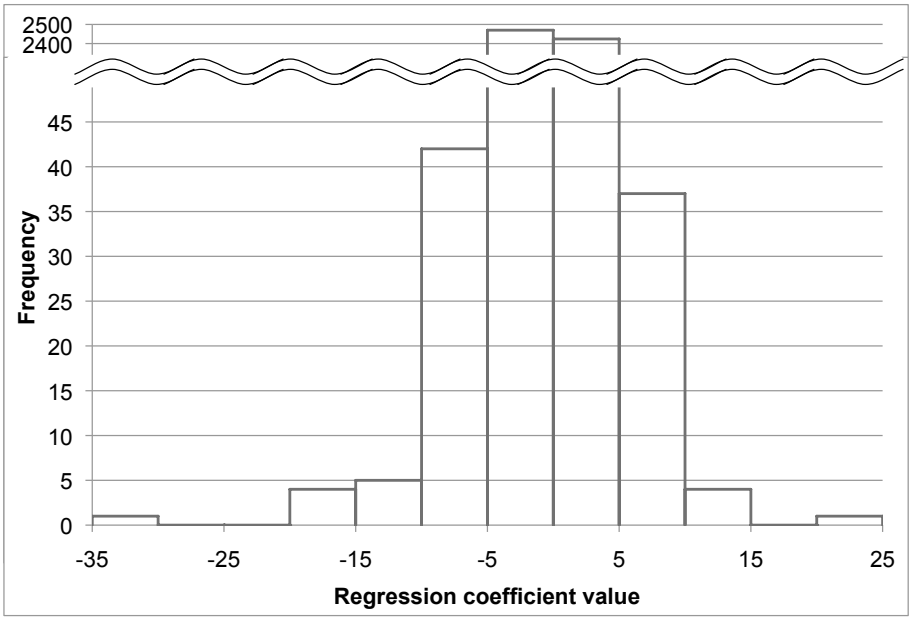
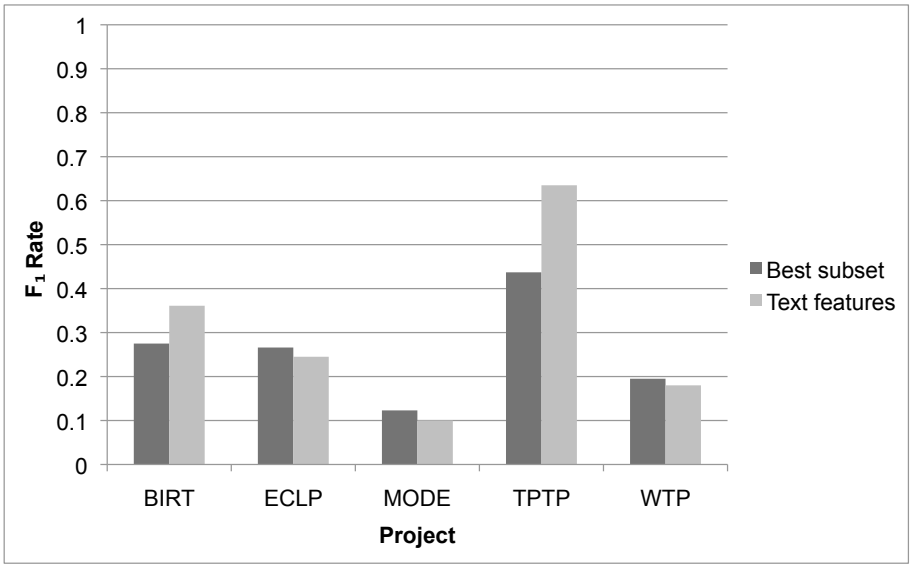


Figure 3.1: Histogram of the regression coefficient value of a logistic regression model in project ECLP.

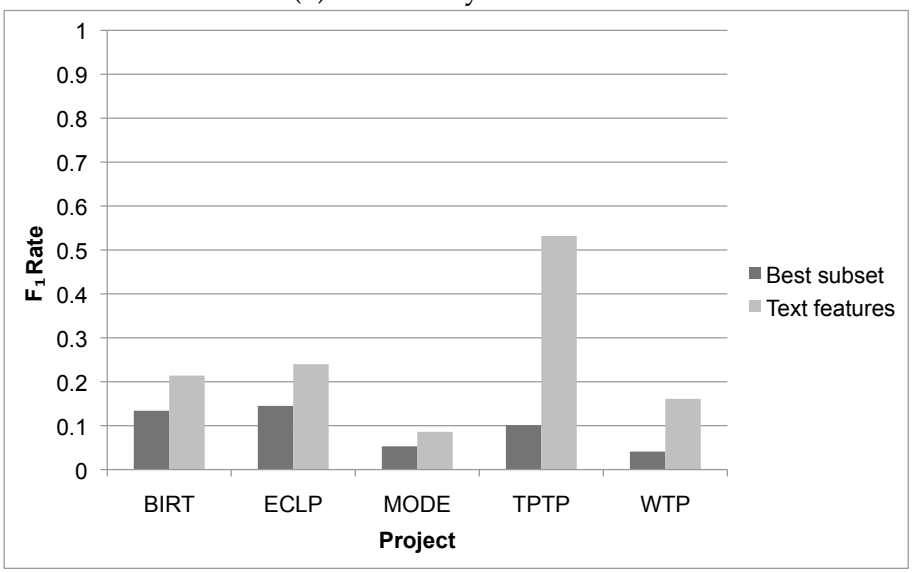
and/or "Member", the FP probability is low.

Next, the distribution of the regression coefficient value is investigated. Figure 3.1 shows the histogram of the regression coefficient value of a logistic regression model in project ECLP. A large regression coefficient represents a strong impact of the feature on the FP probability, while a near zero regression coefficient means little impact on the FP probability. As shown in Figure 3.1, most of the regression coefficient values are near zero. Such distribution of the regression coefficient values is seen in the other projects. These distributions can be interpreted as being able to train logistic regression models without distinguishing a few project-specific useful text features and other not so useful text features. However, there is only one text feature whose corresponding regression coefficient value is zero. Therefore, almost all large-scale text features are needed to build logistic regression models.

Figure 3.2 shows the F_1 rate in each project comparing the best subset from ten metrics and text features. Figure 3.2 (a) is the result of naive Bayes models and (b)



(a) Naive Bayes models



(b) Logistic regression models

Figure 3.2: Comparison of the F_1 rate of the 10-fold cross validation results.

Table 3.7: Detailed results of the 10-fold cross validation

Project (% of faulty modules)	Prediction model	Features	Accuracy	Recall	Precision	F_1
BIRT (8.6%)	Naive Bayes	Best subset	0.902	0.216	0.377	0.275
		Text features	0.806	0.634	0.252	0.361
	Logistic Regression	Best subset	0.917	0.075	0.654	0.134
		Text features	0.732	0.423	0.143	0.214
ECLP (4.5%)	Naive Bayes	Best subset	0.947	0.215	0.346	0.266
		Text features	0.879	0.449	0.169	0.245
	Logistic Regression	Best subset	0.956	0.082	0.585	0.145
		Text features	0.897	0.371	0.177	0.240
MODE (0.6%)	Naive Bayes	Best subset	0.980	0.222	0.085	0.123
		Text features	0.940	0.463	0.056	0.100
	Logistic Regression	Best subset	0.994	0.028	0.500	0.053
		Text features	0.966	0.220	0.054	0.086
TPTP (28.2%)	Naive Bayes	Best subset	0.353	0.891	0.290	0.437
		Text features	0.745	0.779	0.535	0.635
	Logistic Regression	Best subset	0.722	0.056	0.571	0.101
		Text features	0.703	0.594	0.482	0.532
WTP (2.5%)	Naive Bayes	Best subset	0.956	0.213	0.180	0.195
		Text features	0.854	0.623	0.105	0.180
	Logistic Regression	Best subset	0.974	0.022	0.308	0.041
		Text features	0.898	0.383	0.102	0.161

Table 3.8: Pearson's correlation in evaluation metrics and the percentage of faulty modules

Features	Naive Bayes		Logistic regression	
	Accuracy	F_1	Accuracy	F_1
Best metrics subset	-0.987	0.944	-0.999	0.276
Text features	-0.881	0.978	-0.830	0.975

is the result of logistic regression models. Table 3.7 presents the detailed results of 10-fold cross validation. As seen in Figure 3.2 (a), which shows the results of the naive Bayes model, though the F_1 rate of the results using text features are narrowly less than the results using the best subset of metrics in project ECLP, MODE, and WTP, the results using text features are much greater than the best subset in projects BIRT and TPTP. The results of the logistic regression models, which are shown in Figure 3.2, illustrate that large-scale text features have a greater capability of fault-prone module prediction than a best metrics subset. As shown in Table 3.7, the best metrics subset achieved a higher precision rate and the text features achieved a higher recall rate.

With the naive Bayes models using text features, the F_1 rates range from 0.100 to 0.635. To explain the difference of the prediction performance, Pearson's correlations are calculated between the evaluation metrics and the percentage of faulty modules. Table 3.8 lists the correlation values. The values show a strong negative correlation for accuracy. This means that if the percentages of faulty modules are low, accuracy rates are high. This is because it is easy to achieve high accuracy by classifying most modules as NFP when the percentages of faulty modules are low since most modules are not faulty. On the contrary, there are strong correlations between the F_1 rate and the percentage of faulty modules except for logistic regression models with a best metrics subset. Logistic regression models with the best metrics subset always obtained less than the 0.15 F_1 rate for the five projects. The other combination of prediction models and used

features revealed that the higher the percentage of faulty modules, the higher the F_1 rates can be achieved. This is because if there are few faulty modules, it is very difficult to predict the faulty modules with only a few false positives and false negatives.

3.4.2 Prediction on Post Release

Table 3.9 presents the detailed results of the prediction on post release. Table 3.10 shows the F_1 rate in each project by comparing the best subset from ten metrics and text features. Each value represents the F_1 rate with text features, minus the F_1 rate with the best metrics subset. Therefore, a positive value means that text features overcame the best metrics subset, and a negative value, vice versa. As seen in Table 3.10 results of the naive Bayes model, although the F_1 rate of the results using text features are narrowly less than the results using the best subset of metrics in projects ECLP and WTP, the results using text features are much greater than the best subset in projects BIRT, MODE, and TPTP. In TPTP especially, the text features achieved almost a 0.15 higher F_1 rate. The results of logistic regression models illustrate how large-scale text features overcame the best metrics subset in every project. As shown in Table 3.9, the best metrics subset tends to exhibit low recall and relatively high precision, and text features tend to exhibit high recall and low precision, similar to the results of the 10-fold cross validation.

Although the proposed approach using large-scale text features seems to work well, it is questionable whether the FP probability of a module may be strongly influenced by the number of text features in the module. That is, modules whose source code contains lots of text features might be simply predicted as FP. Since the number of text features is related to LOC, we computed the Pearson's correlation between the probability yields from the naive Bayes models and the LOC. Table 3.11 lists the correlation values. As shown in Table 3.11, every correlation value in the five projects is low. This means that there are weak correlations between the

Table 3.9: Detailed results of the prediction on post release

Project (% of faulty modules)	Prediction model	Features	Accuracy	Recall	Precision	F_1
BIRT (8.6%)	Naive Bayes	Best subset	0.893	0.199	0.279	0.232
		Text features	0.759	0.630	0.196	0.299
	Logistic Regression	Best subset	0.919	0.069	0.513	0.121
		Text features	0.802	0.526	0.213	0.303
ECLP (4.5%)	Naive Bayes	Best subset	0.946	0.191	0.181	0.186
		Text features	0.868	0.461	0.112	0.180
	Logistic Regression	Best subset	0.965	0.089	0.350	0.142
		Text features	0.946	0.557	0.303	0.392
MODE (0.6%)	Naive Bayes	Best subset	0.974	0	0	NaN
		Text features	0.926	0.023	0.002	0.004
	Logistic Regression	Best subset	0.994	0	0	NaN
		Text features	0.965	0.023	0.006	0.009
TPTP (28.2%)	Naive Bayes	Best subset	0.213	0.896	0.156	0.265
		Text features	0.631	0.807	0.276	0.411
	Logistic Regression	Best subset	0.831	0.126	0.397	0.191
		Text features	0.789	0.658	0.402	0.499
WTP (2.5%)	Naive Bayes	Best subset	0.938	0.188	0.061	0.092
		Text features	0.774	0.579	0.043	0.080
	Logistic Regression	Best subset	0.980	0.045	0.171	0.071
		Text features	0.805	0.609	0.052	0.096

Table 3.10: F_1(text features) - F_1(best metrics subset)

Prediction model	BIRT	ECLP	MODE	TPTP	WTP
Naive Bayes	0.067	-0.006	0.004	0.146	-0.012
Logistic regression	0.182	0.393	0.057	0.324	0.148

Table 3.11: Pearson's correlation in naive Bayes probability and LOC

BIRT	ECLP	MODE	TPTP	WTP
0.136	0.032	0.026	0.007	0.041

probability yielded from the naive Bayes models and the LOC. Therefore, it can be said that FP probability is not simply derived from naive Bayes models based on the number of text features in a module.

3.5 Discussion

3.5.1 Threats to Validity

There are four threats to the validity of this study.

Target projects are the Eclipse projects only. This is the external validity threat for generality of data used in the experiments. In general, the Eclipse projects do much better than other open-source projects when using machine-learning models to predict fault-prone modules. Using other open-source projects, different results may be obtained. In addition, industrial projects may lead to different results.

There may be incorrect identifications of faulty and not faulty modules. The algorithm adopted in this study to identify faulty modules has a limitation. As shown in Section 2.1, the SZZ algorithm can identify faults by linking fault reports and revision logs. Therefore, we cannot identify faults that is not written in either fault reports or revision logs. In addition, there may be false positives in identified faults.

Incorrect identifications of training data badly influence the quality of the

prediction models. In addition, if identifications of test data are incorrect, performance evaluation metrics cannot be calculated properly. To make a complete collection of faulty modules from a source code repository, further research is required.

Specific settings for implementing the approach may influence the prediction performance improperly. Because of the limitations of time and memory, we limit the number of text features used in each project to approximate 5,000. Important text features may be discarded by this setting. In addition, we removed all comments before counting the number of text features. These settings may result in improper prediction.

There might be flaws in the design of experiments In order to show the effectiveness of our approach using large-scale text features, we compared our approach with the best subset of well-known metrics including the CK metrics suite. However, these prepared metrics may be not enough. In addition, although we prepared two experiments including (1) 10-fold cross validation and (2) fault-prone module prediction on post release to compare fairly, there might be flaws in showing the effectiveness. For example, in the (2nd) experiment, every period between release 1 and release 2 is one year. If we vary the period, the results might change.

3.5.2 Related Work

Aversano et al. trained prediction classifiers with a weighted-term vector created from text belonging to software changes [3]. They used variables, names, language keywords etc. as terms. They concluded that the K-Nearest Neighbors classifier yielded a significant trade-off between precision and recall. Kim et al. introduced a change classification technique [62]. They gathered features from source code text and other meta data, and applied them to the Support Vector Machine to predict faulty changes. They obtained 78 percent accuracy and a 60 percent faulty change recall on average. Though these two studies used text features extracted

only from software changes, we targeted the entire text features in source code. In addition, although these two studies conducted only a 10-fold cross validation, we conducted not only 10-fold cross validation but also evaluate the prediction of post-release fault-prone modules.

Our text mining approach has some desirable points, such as:

- Independence from programming languages
- Flexibility in the granularity of a unit
- No need of semantic information

Different from generalized sophisticated metrics, more concrete and smaller granularity of the possible cause of faults have also been studied. Fowler and Beck introduced 22 software structures as problematic code, which they called "bad smells" [34]. Mäntylä et al. presented a subjective taxonomy that categorizes similar bad smells [77]. In addition, they empirically showed correlations between the bad smells. Pan et al. defined 27 fault-fix patterns [94]. Their studies of open-source projects showed that the method call and *if*-related fault-fix patterns commonly appear. However, software structures in these patterns that introduce faults do not always cause faults. Though there are fault-fix structure patterns, a fault-introducing change may be project-specific, package-specific, or other environment-specific. Livshits and Zimmermann tried to find out application-specific error patterns that are concrete method code patterns [75]. They looked for code smell patterns on a fine granularity level. Our approach may be related to such fine granularity code smell patterns.

3.6 Summary

We proposed an approach using large-scale text features for fault-prone module prediction. To show the effectiveness of our approach, we conducted two experiments and compared our approach with prediction models using traditional

metrics by applying it to five open-source Java projects in Eclipse, and obtained higher F_1 values. The performance results of our text-mining-based prediction models implies that:

- Large-scale text features are useful to build a practical model.
- Measuring sophisticated metrics is not always necessary for predicting fault-prone modules.

Built models with large-scale text features just predict fault-prone modules. While traditional sophisticated metrics can suggest how a developer should modify modules or what problems are in them, text features do not derive such suggestions. However, since there is no need to collect meaningful module features, applying our approach to projects is easy.

Moreover, the large-scale text-features approaches have several desirable points as follows:

- They are independent from programming languages.
- We can treat the flexible granularity of a unit as classified modules or as a training set. For example, a method can be treated as a module.
- We do not need semantic information.

Chapter 4

Prediction on Fine-grained Modules

4.1 Overview

For fine-grained prediction with well-known historical metrics, obtaining version histories on fine-grained modules is a big challenge. To tackle this problem, we developed a fine-grained version control system, *Historage* [47]. Historage is constructed on top of Git, and can control method histories of Java. With this system, we can collect the same historical metrics for methods and files, and compare the prediction results based on effort-based evaluation.

This chapter presents the architecture of Historage and then shows the results of fine-grained prediction with well-known historical metrics collected from the constructed Historage.

4.2 Fine-grained Version Histories

4.2.1 Problems

For fine-grained prediction with historical metrics, we need to analyze the histories of fine-grained modules. Software configuration management (SCM) system repositories have been mined and analyzed for many research purposes because they contain rich information on real software activities and products. File-level histories can be easily collected from SCM systems, but it is not easy to collect fine-grained module histories, such as the histories of classes or methods.

The concept of method-level version control in object-oriented programming can be seen in the Orwell SCM system [108]. Though several tools have been proposed to support fine-grained version control for development, no such a tool has been actually integrated into widely used SCM systems [21]. These systems intend to control fine-grained module histories during development. Since existing repositories remain at file-levels, what we have to do is construct a fine-grained module history storage with the data from the existing file-level SCM systems. The requirements of such a fine-grained module history storage is storing entire histories of all fine-grained modules even if modules are renamed or moved, which is satisfied with existing SCM systems for file-level.

Related Systems

Better tools are required for future research in software evolution [81]. There are several related tools proposed and used in research. *BEAGLE* is a framework incorporating subtools from software metrics software visualization, and relational databases [37,111]. On the point of fine-grained module histories, *BEAGLE*

performs *origin analysis* to identify change types including renaming, moving, splitting, and merging. However, it targets selected release revisions for applying *origin analysis*. *C-REX* is an evolutionary extractor [43]. It records fine-grained module changes over the development periods. Though *C-REX* targets entire revisions, it cannot identify renaming. *Kenyon* is designed to facilitate software evolution research [5]. It supports CVS, Subversion, and ClearCase SCM systems and conducts prepossessing tasks for fine-grained change analysis. Though it stores entire revisions, change types are limited to adding, deleting, and modifying. *APFEL* collects fine-grained changes in relational databases [123]. It investigates fine-grained changes at the token level. Though revisions are stored entirely, renaming is not identified.

In summary, existing systems store only limited histories of fine-grained modules, that is, each system does not satisfy both storing every version and identifying matches when renames or moving. In particular, match identification is a challenging task.

Change Type Identification Techniques

There are many studies about identifying changes.

One-to-one matching techniques. Based on the matching technique survey by M. Kim and Notkin, one-to-one software module matching techniques are summarized as follows: *module name matching*, *string matching*, *syntax tree matching*, *control flow graph matching*, *program dependence graph matching*, *binary code matching*, *clone detection*, and *origin analysis tools* [59]. S. Kim et al. applied several method matching techniques for *origin analysis* limited to renaming and moving to open-source software projects, and evaluated the effectiveness of the techniques [61]. They reported that though *clone detection* yields an accuracy value 67.4, *function body diff* achieved 90.2.

Splitting and merging. Splitting and merging of software entities are targeted by *origin analysis*. Godfrey and Zou proposed a technique of inferring such

events based on matching procedures using multiple criteria including names, signatures, metric values, and call dependencies [37]. Splitting and merging correspondence analysis is also known as one-to-many and many-to-one matching [117]. Wu et al. combined text similarity analysis and call dependency analysis for those method matching [117].

Systematic structural changes. Recognizing structure changes including refactorings and object-oriented design changes is one of the hot topics of change analysis. These analyses are based on techniques of matching object-oriented program elements. The differences of program elements, helps to infer what structural changes are occurred. *RefactoringCrawler* detect refactorings based on identifying renaming packages, classes, methods, and moving methods [24]. Those changes are identified using structural data, call-graph and tokens from entities. *MolhadoRef* [25, 26] is a semantics-based and refactoring-aware SCM system [35]. It adopts the *RefactoringCrawler* [24] and uses refactoring logs to support merging. Weißgerber and Diel presented a technique to detect changes that are likely to be refactorings [113]. Their matching technique is based on structural similarity and code clone analysis.

Framework usage changes. Xing and Stroulia proposed an approach for API-evolution support, called Diff-CatchUP [119]. At the step of change identification, UML-diff, which is based on name similarity and code dependency similarity of program elements [118], is used. After identifying changes, plausible API replacements are proposed. Dagenais and Robillard presented a technique to recommend adaptive changes for clients of framework code based on structure change analysis [19]. Their matching technique is based on structure similarity and out going call dependency similarity.

Discussion. Though there are some variations, change identification is a kind of matching problem. In the computer vision research area, similar problems are known as the *correspondence problem* and techniques are classified in following two classes [110]:

Table 4.1: Change identification techniques and using data

Technique	Graph	Feature
S. Kim et al. [61]	calls	name, text, metrics
Godfrey and Zou [37]	calls	name, metrics
Wu et al. [117]	calls	text
Dig et al. [24]	calls, structure	tokens
Weißgerber and Diel [113]	structure	name, text
Xing and Stroulia [118]	structure	name
Dagenais and Robillard [19]	calls, structure	name

Graph-based methods: checking if correlations on graph structures are similar or not.

Feature-based methods: finding features and seeing if they are similar or not.

Table 4.1 summarizes the studies based on this classification. As shown in Table 4.1, every study uses both methods for change type identification. As *graph-based methods* and *feature-based methods* have different advantages and limitations, the combination of both methods is expected to achieve better results. Most studies mainly adopt *graph-based methods* and use *feature-based methods* for improving method correspondence problems.

Graph-based methods require unchanged or easily understandable correlated parts. Therefore, identifying corresponding entities, if there is not enough of a correlated part, or if there are major changes is difficult. Wu et al. reported the limitations and insist that graph-based analysis cannot overcome this difficulty [117]. Though it is different entity (AST node) analysis, Fluri et al. proposed an algorithm based on *graph-based methods* and reported following two limitations [33]:

- Mismatching can propagate. Not only mismatching for each targeting entity, correlate entities can be mismatched.

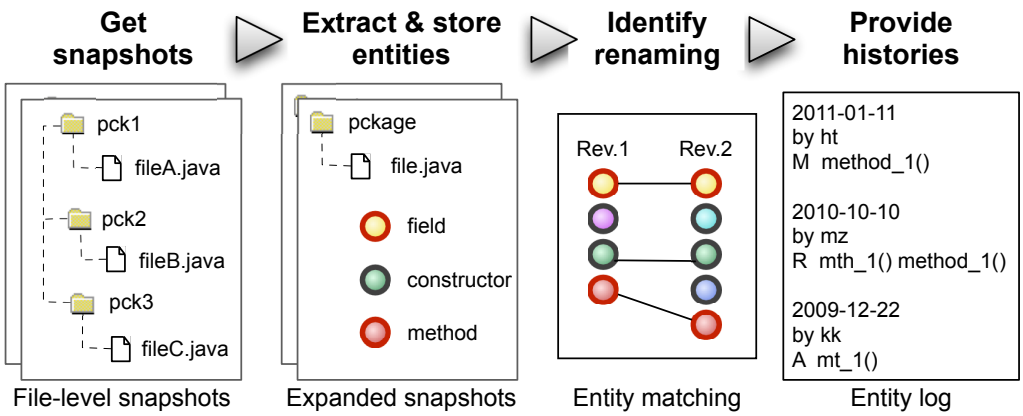


Figure 4.1: Providing fine-grained module histories from file-level repositories.

- The worst-case complexity increase. To decrease mismatching, complex algorithm is needed and this increase the worst-case complexity.

4.2.2 Historage: A Fine-grained Version Control System

Overview

For fine-grained module history storage, change types in every commit should be identified. As discussed in Section 4.2.1, the previous studies used graph-based methods, which have some problems. To satisfy the requirements of fine-grained module history storage, we identify change types only with feature-based methods. Figure 4.1 presents an overview of our system for providing fine-grained module histories. From file-level snapshots, each module content (text) is extracted and stored independently. Change types are identified between two revisions. Then each module history is presented even if there are renames and moving.

To develop our system, we make use of Git, which is a source code management system, as storage. Recently Git has attracted some researchers [9, 50]. Bird et al. reported both its promise and peril [9]. Though Git is known for decentralization of source code management, we found that Git architecture is also effective for our purpose, that is, for constructing fine-grained module history storage. We

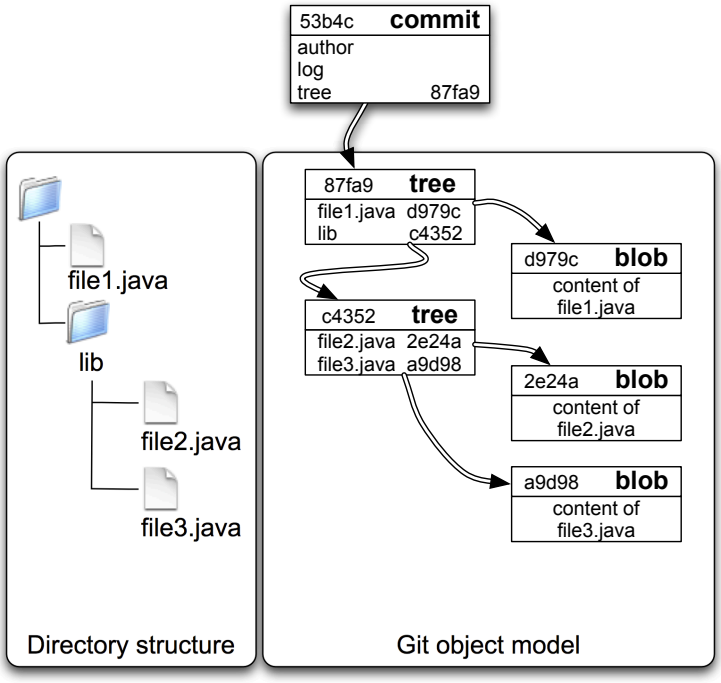


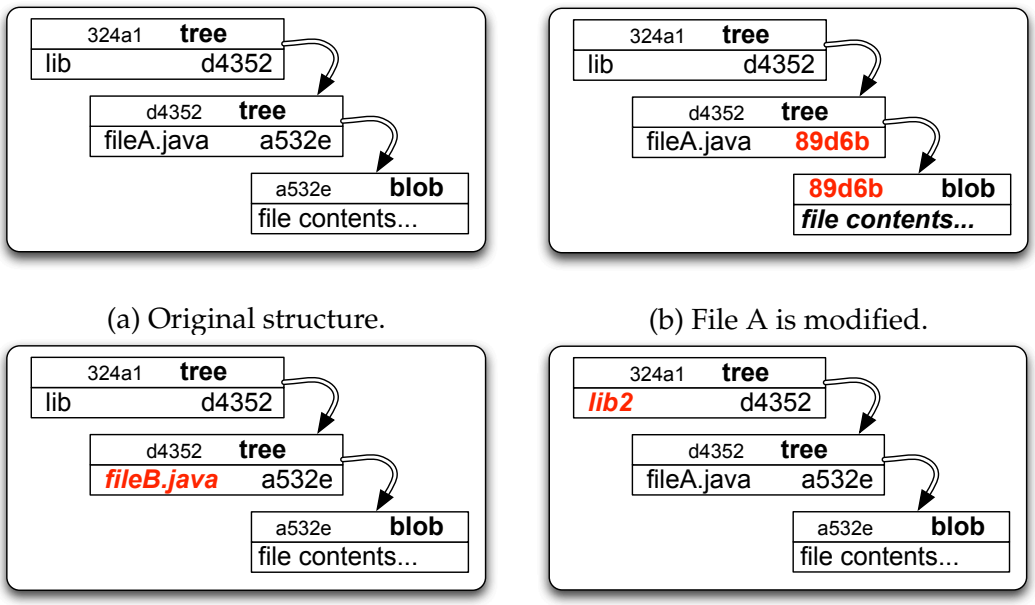
Figure 4.2: How a snapshot is stored in Git.

develop our system *Historage* on top of Git. This system can store entire histories of all fine-grained modules even if modules are renamed or moved.

Preliminary – Git

Git is a content-addressable file system [15]. Git controls file contents, directory structures, file histories, commit logs, etc., by managing Git objects. Each object is stored in a Git object database and is compressed and named by the SHA-1 (Secure Hash Algorithm) value of its contents.

Storage of snapshots. Figure 4.2 shows how directory structure of a snapshot is stored and managed with Git object model. The left side of Figure 4.2 represents a sample directory structure at the time of a commit and the right side shows a Git object model that reflects the directory structure. Each blob object, which represents a file, is referred by a tree object. A tree object, which represents a directory, refers blobs and trees. The top tree object is referred by a commit object, which contains the author and log of the commit. As shown in Git object model



(a) Original structure.

(b) File A is modified.

(c) Renamed to file B.

(d) File A is moved.

Figure 4.3: How changes are detected in Git.

in Figure 4.2, each object is identified with SHA-1 value.

Identifying changes. Here, we explain how Git identify change types with Figure 4.3. Figure 4.3 (a) shows an original directory structure in the Git object model. Figure 4.3 (a) shows that `fileA.java` exists in the directory named `lib`. The content of `fileA.java` is stored in a *blob*, which is named by SHA-1 value: `a5352e`. The `lib` directory is represented as *tree* named `d4352`, and the name of the directory is stored in the `324a1` *tree*.

If the `fileA.java` is modified, the Git object model changes to Figure 4.3 (b). Since the file content is changed, the corresponding *blob* is also changed. Figure 4.3 (c) represents the rename of the file. This can be identified because the same *blob* SHA-1 value is linked to a different file name, `fileB.java`. Figure 4.3 (d) represents a directory structure after moving the `fileA.java`. This can be detected because the directory, which has a different name `lib2`, contains the `fileA.java`.

When file paths are changed, it is often the case with files that the contents of the files are also modified. Even in such cases, Git is able to detect relationships

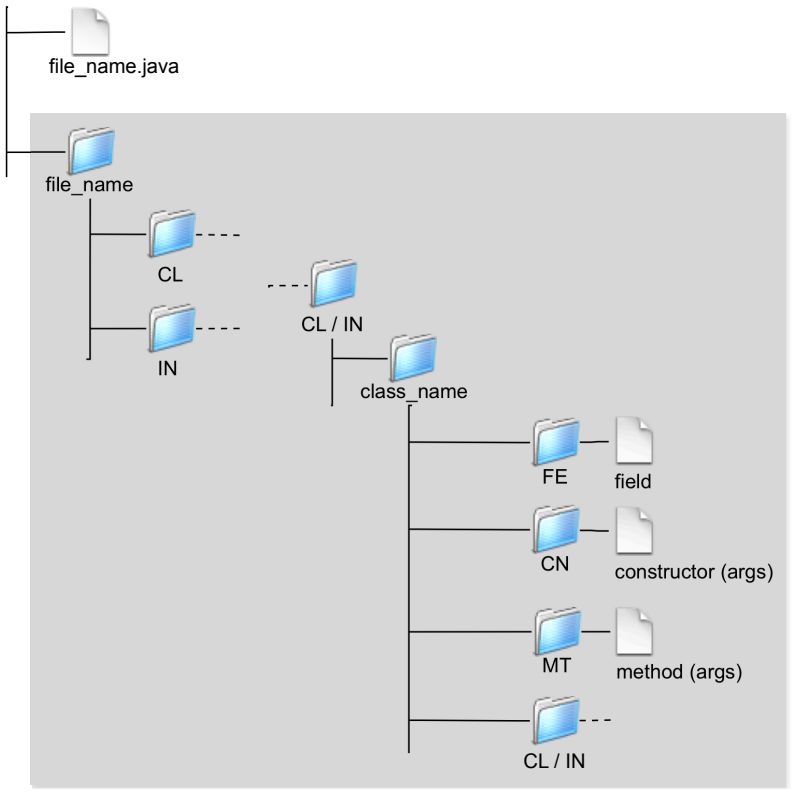


Figure 4.4: Directory structure for fine-grained entities.

between changes if the file contents are similar enough. This is performed by checking that the amount of deletion of the original content and insertion of new content is larger than a threshold, which is set to 50% of the size of the smaller files (original or modified). Therefore, if deletion or insertion is less than 50%, two files in the parent and child commits are detected as moving or renaming. The threshold value can be changed.

Technique

For storing fine-grained module files, the directory structure is designed as Figure 4.4*. If there are fine-grained entities in a Java file, `fine-name.java`, each module is additionally stored as a file.

*This is a prototypal structure. It is also reasonable to store class declarations for representing logical structures.

Three kinds of module files are stored in three kinds of directories, `FE` (for fields), `CN` (for constructors), and `MT` (for methods). These directories are stored in a directory identified with a class or interface name, which contains those entities as shown at the right part of Figure 4.4. Anonymous classes are ignored in this paper. Entire files and directories are stored in the `file-name` directory. Directories and files in the gray space of Figure 4.4 are newly prepared for new directory structure.

The entities we target in this dissertation are named as follows for files:

Field: field name.

Constructor: constructor name and parameter list.

Method: method name and parameter list.

Changes of module names correspond to file name changes, and the moving of entities correspond to moving files. If a module is deleted in a commit and reappear in a later commit, Git can output its history including disappearing periods.

As described in Section 4.2.2, matches between renamed or moved entities are identified based on file content similarity. If two entities are highly similar, it is rational to detect them as corresponding entities. Because this matching technique is simple, there may be obvious mismatches, that is, matches between different module types, such as a match between a method and a constructor, for example. These mismatches are distinguished easily by checking directory names whether they are the same or not. We filter out these mismatches before providing module histories.

Architecture

Figure 4.5 shows the architecture of Historage[†]. As shown in Figure 4.1, extracting and storing fine-grained entities are conducted on each snapshot. A snapshot in

[†]A to construct Historage is available from `https://github.com/hdrky/git2historage`.

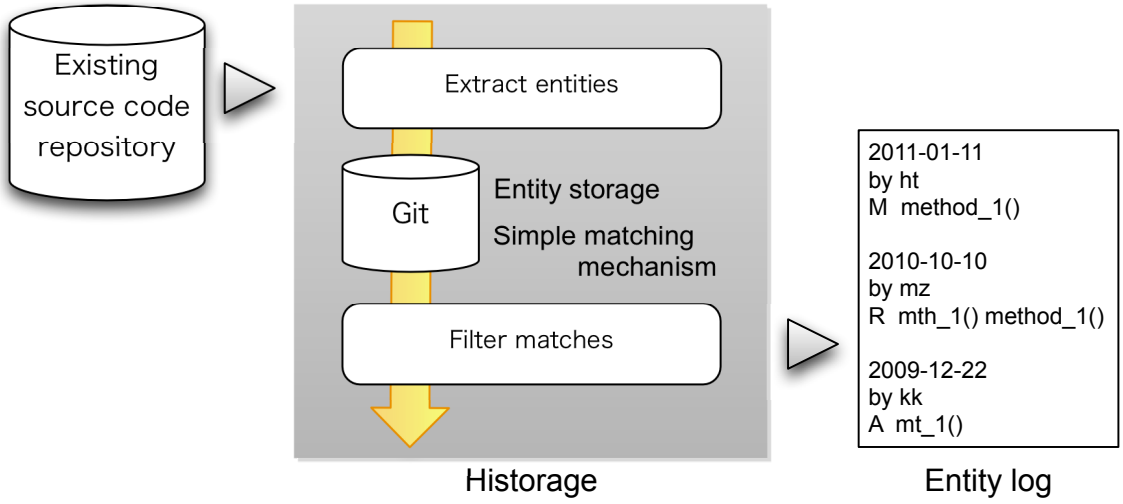


Figure 4.5: Historage architecture.

each revision can be obtained easily from Git. Even if existing repositories are not in the Git system, it is possible to convert them to Git repositories from most SCM systems. To extract the fine-grained entities in Java files, we use the source code analysis tool MASU[‡], which is an open-source tool. The threshold value for match identification is set to 30% (as a option of Git commands) based on empirical study reported in Section 4.2.3 for filtering appropriate matching entities beyond renaming and moving.

4.2.3 Empirical Evaluation

In this section, we empirically investigate the usefulness of our fine-grained version control system, *Historage*.

Target Projects

As shown in Table 4.2, we select five open-source software projects: Eclipse WTP incubator (WTP incubator), Apache Hadoop Common (Hdoop), Apache Subversion (Subversion), jEdit, and Android framework classes and services (Android).

[‡]`http://sourceforge.net/projects/masu/`

Table 4.2: Open-source software projects for evaluation

Project	First Commit	Last Commit	(# of .java)	Total Commits
WTP incubator	2007-11-10	2010-07-22	1,944	541
Hadoop	2009-05-19	2010-12-26	667	375
Subversion	2000-03-01	2010-11-29	127	738
jEdit	2001-09-02	2010-10-02	546	4,399
Android	2008-10-21	2010-12-23	2,690	25,965

These projects written in Java and Git repositories are available. We cloned the Git repositories on the 27th December, 2010.

The disk space overhead compared with original repositories and constructed *Historage* depends on the projects. The over head varies from nearly equal to a few times on the Git database.

Match Identification

We investigated every matching pair of fine-grained entities in the repositories (the number of commits are shown in Table 4.2) except for the Android project. As there are more than 180,000 matching pairs in the Android project, we limited the pairs to those existing on January, 2010, for the Android project. Module pairs are classified according to similarity values, which are calculated by Git, to see the impact of the threshold and to investigate the effectiveness of match identification. We determine by hand if a matching is correct or not.

Table 4.3 shows the results for the five projects. Mismatches are matches between different module types. It is possible to distinguish them automatically. Shown in bold fonts, the percentage of correct matches when similarity is greater than or equal to 30% is higher than 97% in all projects. This means that we can identify more than 97% correct matches of fine-grained entities. Although there are some rename changes whose similarity is less than 30%, it is now difficult to distinguish them with our system. The recall and precision values, where *Historage* provides matches with the threshold value 30%, are presented in Table

Table 4.3: Match identification results in five open-source software projects

Project		Sum†	Correct (%)		Measure (%)
WTP incubator	mismatches	62			
	$s < 30$	366	99	27.0	Rec. 96.7
	$30 \leq s < 100$	436	426	**97.7**	Prec. 99.6
	$s = 100$	2,641	2,641	**100.0**	
Hadoop	mismatches	32			
	$s < 30$	152	43	28.3	Rec. 88.1
	$30 \leq s < 100$	141	141	**100.0**	Prec. 100
	$s = 100$	178	178	**100.0**	
Subversion	mismatches	41			
	$s < 30$	148	88	59.5	Rec. 96.4
	$30 \leq s < 100$	528	521	**98.7**	Prec. 99.7
	$s = 100$	1,820	1,820	**100.0**	
jEdit	mismatches	254			
	$s < 30$	1,229	347	28.2	Rec. 94.4
	$30 \leq s < 100$	1,461	1,457	**99.7**	Prec. 99.9
	$s = 100$	4,421	4,421	**100.0**	
Android	mismatches	203			
	$s < 30$	1,125	98	8.7	Rec. 99.8
	$30 \leq s < 100$	912	903	**99.0**	Prec. 99.98
	$s = 100$	61,278	61,278	**100.0**	

†: module pairs exist in January, 2010, for the `Android` project, and entire module pairs for the other projects.

s: similarity.

Table 4.4: Match identification results for module types in the WTP incubator project

Module		Sum†	Correct (%)		Measure (%)
Field	$s < 30$	33	6	18.2	
	$30 \leq s < 100$	45	39	86.7	Rec. 99.4
	$s = 100$	1,142	1,142	**100.0**	Prec. 99.4
Constructor	$s < 30$	21	11	52.4	
	$30 \leq s < 100$	59	59	**100.0**	Rec. 94.4
	$s = 100$	129	129	**100.0**	Prec. 100
Method	$s < 30$	312	82	26.3	
	$30 \leq s < 100$	332	328	**98.8**	Rec. 95.4
	$s = 100$	1,370	1,370	**100.0**	Prec. 99.8

†: entire module pairs.

s: similarity.

4.3. The recall values range from 88.1% to 99.8%, and the precision values range from 94.4% to 100%.

Table 4.4 represents the match identification results for each fine-grained module type in the WTP incubator project. We can see that the percentages of correct pairs are different depending on the module types. For example, the result on `field` is relatively low. We think this is because it is more difficult to compare the similarity with the small contents of fields. On the contrary, change type identification of `constructor` achieved relatively high results. We think this is because there is a small number of potential constructor pairs compared to method pairs. Similar results can be seen in the other projects.

With the investigation of the results, we found that automatic match identification using Git and our filtering works relatively well. Distinguishing actual renaming/moving when similarity values are less than 30% is part of our future work. This result is practical enough.

4.3 Experimental Setup

Using our Historage, we can collect historical metrics on fine-grained modules, and conduct fine-grained prediction. We study with Java software, and Java methods are considered as fine-grained modules.

To investigate the effectiveness of fault-prone module prediction on fine-grained modules, we compare the prediction on different levels, that is, Java files and Java methods. File-level and method-level prediction models are built with well-known historical metrics proposed and used in previous studies, and are compared with effort-based evaluation.

4.3.1 Research Questions

We investigate the following three research questions:

RQ1: Are method-level prediction models more cost-effective than file-level prediction models?

RQ2: (when method-level prediction models are more cost-effective) Why are method-level prediction models more cost-effective than file-level prediction models?

Compared with package-level and file-level studies, there is a difference on file-level and method-level studies. Since packages consist of files, the sizes of faulty regions are equal in both levels. However, the faulty region sizes should not be equal in all files and all methods. This is because a file does not consist of methods only. For example, faults on Java fields are counted only on file-level. To conduct fair comparison with file-level and method-level fault-prone prediction, we target faults only in methods.

Table 4.5: Open-source software projects for study

Name	Initial Date	Last Date	# of commits	# of authors	Last LOC
ECF	2004-12-03	2011-05-31	9,748	23	15,337
WTP Incubator	2007-11-10	2010-07-22	1,133	17	370,910
Xpand	2007-12-07	2011-05-31	1,038	21	136,702
Ant	2000-01-13	2011-08-19	12,590	46	254,890
Cassandra	2009-03-02	2011-09-20	4,423	14	171,596
Lucene/Solr	2010-03-17	2011-09-20	3,485	27	26,390
OpenJPA	2006-05-02	2011-09-15	4,180	26	169,427
Wicket	2004-09-21	2011-09-20	15,033	25	339,292

4.3.2 Target Projects

We selected eight open-source projects for our study. Eclipse Communication Framework (ECF), WTP Incubator, and Xpand were chosen from the Eclipse Projects. Ant, Cassandra, Lucene/Solr, OpenJPA, and Wicket were chosen from the Apache Software Foundation. All projects are written in Java. We obtained each Git repository[§].

Table 4.5 summarizes information for each studied project. The development period ranges from 18 months to 11 years, and LOC on the last date of the studied period ranges from 15k to 370k. Table 4.5 also presents the number of commits (from 1k to 15k), the number of authors (from 14 to 46), and the number of files on the last date (from 700 to 4k).

4.3.3 Metrics Collection

We collected major metrics as introduced in Section 2.2.

Code-related metrics. For code-related metrics, we measured LOC and code

[§]Eclipse Projects from `http://git.eclipse.org/` and Apache Software Foundation from `http://git.apache.org/`

Table 4.6: Measured historical metrics

	Name	Description
Code	LOC	Lines of code
	AddLOC	Added lines of code from the initial version
	DelLOC	Deleted lined of code from the initial version
	AddPerLOC	AddLOC / LOC
	DelPerLOC	DelLOC / LOC
Process	ChgNum	# of changes
	FixChgNum	# of fault-fix changes
	FaultNum	# of fixed fault IDs
	Period	Existing period in weeks
	FaultIntroNum	# of logical coupling commits that introduce more than one fault in other modules
	LogCoupNum	# of logical coupling commits that change fault-existed modules
	AvgInterval	Period /ComNum
	MaxInterval	Maximum weeks between two sequential changes
	MinInterval	Minimum weeks between two sequential changes
	HCM	History complexity metric HCM^{3s}
Organizational	AuthTotal	Total number of authors
	AuthMinor	# of minor authors
	AuthMajor	# of major authors
	Ownership	The highest proportion of ownership for the authors

churn metrics. As stated in Section 2.2.1, these metrics are used in many studies. Code churn metrics for files are easily collected from version control repositories. With our Historage, we can collect code churn metrics for methods similarly to collecting files.

Process-related metrics. For process-related metrics, we collect basic metrics as stated in Section 2.2.2, such as the number of changes, the number of past faults, the number of fault-fix changes, and the existing period of modules (age). Inspired by cache-based approaches, we collect two types of logical coupling metrics: the number of logical couplings with fault-introduced modules and the number of logical couplings with modules that have been faulty. To investigate the frequency of changes, we measured average, maximum, and minimum intervals.

In addition, we also collected one of the *history complexity metrics*. Based on the empirical evaluation in [42], we selected HCM^{3s} because it performed well. For other parameters, we follow paper [42].

Organizational metrics. Organizational metrics and geographical metrics are relatively difficult to collect from open-source projects though it may be possible to mine from several software repositories. Hence, we measure ownership-related metrics designed in [8] although there are lots of metrics, especially for organizational metrics as stated in Section 2.2.3. Organizational metrics in [8] can be collected only from version control repositories.

To measure ownership-related metrics, we follow the definition of proportion of ownership in [8]. The proportion of ownership of an author for a particular module is the ratio of the number of changes by the author to the number of total changes for that module. If ownership of an author is below a threshold, the author is considered a minor author, and otherwise a major author. In [8], values ranging from 2% to 10% are suggested as the threshold based on a sensitivity analysis. Bird et al. targeted compiled binaries as modules for study, which tend to be developed by many developers [8]. On the contrary, files and methods, which are our modules for study, are a relatively small size and are developed by

Table 4.7: Summary of projects studied

			# of Files			# of Methods		
Project	Tag	LOC	All	Faulty	(%)	All	Faulty	(%)
ECF	Root_Release_3_0	113,787	1,715	166	(9.7)	11,121	643	(5.8)
WTP Incubator	v20090510	75,170	606	123	(20.3)	5,492	317	(5.8)
Xpand	Galileo_RC1	90,976	1,247	86	(6.9)	8,543	295	(3.5)
Ant	ANT_180_RC1	101,896	912	87	(9.5)	9,862	156	(1.6)
Cassandra	casandra-0.6.0-rc1	46,672	296	93	(31.4)	4,419	282	(6.4)
Lucene/Solr	lucene_solr_3_1	186,484	1,940	60	(3.1)	14,478	89	(0.6)
OpenJPA	2.0.0	148,800	1,305	91	(7.0)	21,323	165	(0.8)
Wicket	wicket-1.4.0	248,338	3,663	92	(2.5)	25,345	196	(0.8)

relatively only a few developers. To take into account this difference, we set the threshold value at 20%.

4.3.4 Fault Information

To identify fault information, we used the SZZ algorithm explained in Section 2.1. Fault reports are available from `https://bugs.eclipse.org/bugs/` for Eclipse Projects, `https://issues.apache.org/bugzilla/` for Ant, and `https://issues.apache.org/jira/` for the other projects in the Apache Software Foundation.

As reported in [63], naive differencing analysis on step 1 of the procedure should yield incorrect fault-introducing changes, such as non behavior changes and just format changes. To remove such false positives, we ignore changes on blank lines, comment changes, and format changes. In addition, we ignore changes not on methods to identify faults on methods as stated in Section 4.3.1.

We identify faulty files and methods in one revision for each project. We select particular tagged revisions or revisions that are nearby tagged revisions as

studied revisions. Table 4.7 shows the result of faulty module identification. The percentage of faulty files ranges from 2.5% to 31.4%, and the percentage of faulty methods ranges from 0.6% to 6.4%.

4.3.5 Prediction Models

We adopt the RandomForest algorithm [73] as a fault-prediction model. Lessmann et al. confirmed its good performance [72]. There are several other studies using the RandomForest algorithm for fault-prone prediction [54, 79]. We use a statistical computing and graphics tool R [107] and a *randomForest* package for our study. Using prepared modules in Table 4.7, we conduct a 10-fold cross validation analysis.

4.4 Results

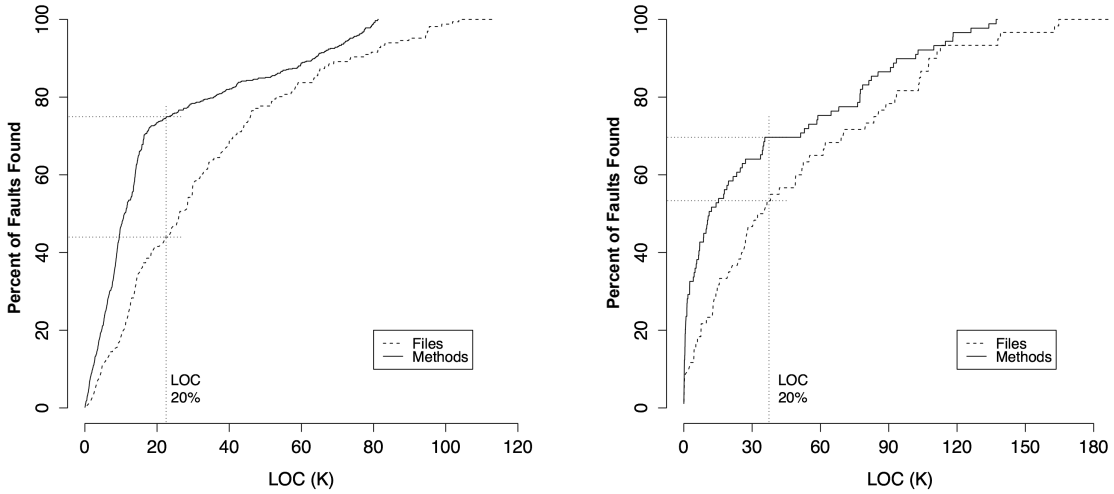
We present our results following the research questions stated in Section 4.3.1. Plots of the results are shown from the Eclipse Communication Framework (ECF) and the Lucene/Solr only. The other results are discussed in the text.

To compare different prediction models with effort-based evaluation, the percentage values of faults found on the same value of the percentage of LOC should be easy to understand. For this cutoff value, 20% of LOC is used in some studies [2, 54, 79, 99]. This value can be considered as more realistic than investigating entire modules.

4.4.1 Effort-based Evaluation: File-level vs. Method-level

RQ1: Are method-level prediction models more cost-effective than file-level prediction models?

Figure 4.6 shows two plots of cost-effectiveness curves. A file-level curve (dashed) and a method-level curve (solid) are plotted. We can see that the method-level curves rise larger than the file-level curves in a small LOC. As a result, more



(a) ECF (b) Lucene/Solr

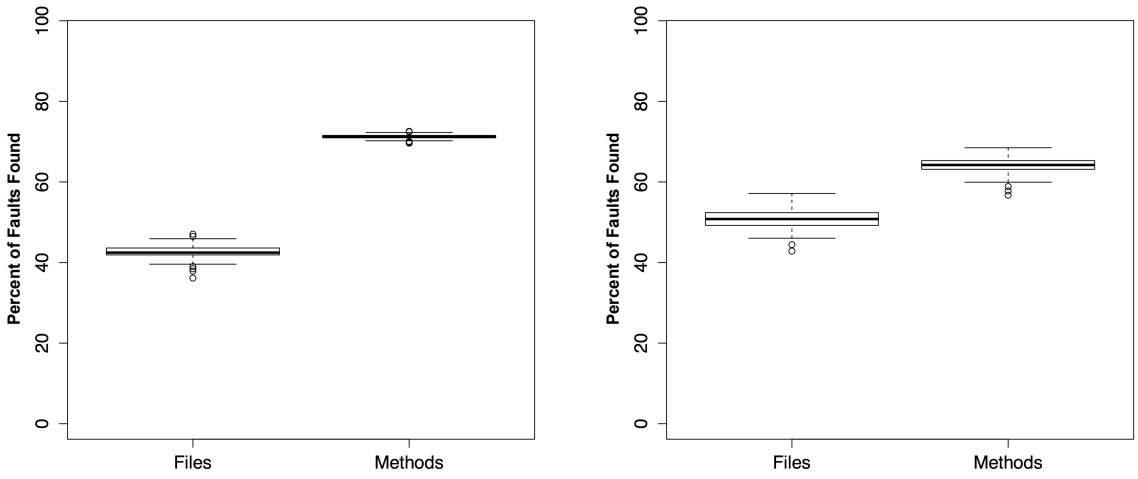
Figure 4.6: Cost-effectiveness curves of file-level and method-level prediction.

faults can be found by method-level prediction when investigating 20% of LOC, represented by the dotted lines. We found similar results for all projects.

As insisted by Arcuri and Briand, we should collect data from a large enough number of runs to assess the results of randomized algorithms because we obtain different results on every run when applied to the same problem instance [1]. RandomForest is a randomized algorithm. Figure 4.6 shows the result on one run. Following the suggested value of 1,000 as a very large sample [1], we conducted a 1,000 times run for all projects.

Figure 4.7 shows the results of the 1,000 run. In each project, boxplots of the value of percentages of faults found in 20% LOC for file-level and method-level are shown. In all projects, we observed the small distributions of the values, and method-level achieved higher than file-level.

In Table 4.8, we summarize the median values of the percentages of found faults when investigating 20% of LOC in the system. The second and third column shows the values of file-level and method-level results, and the fourth column shows the delta of the values (method-level value - file-level value). We

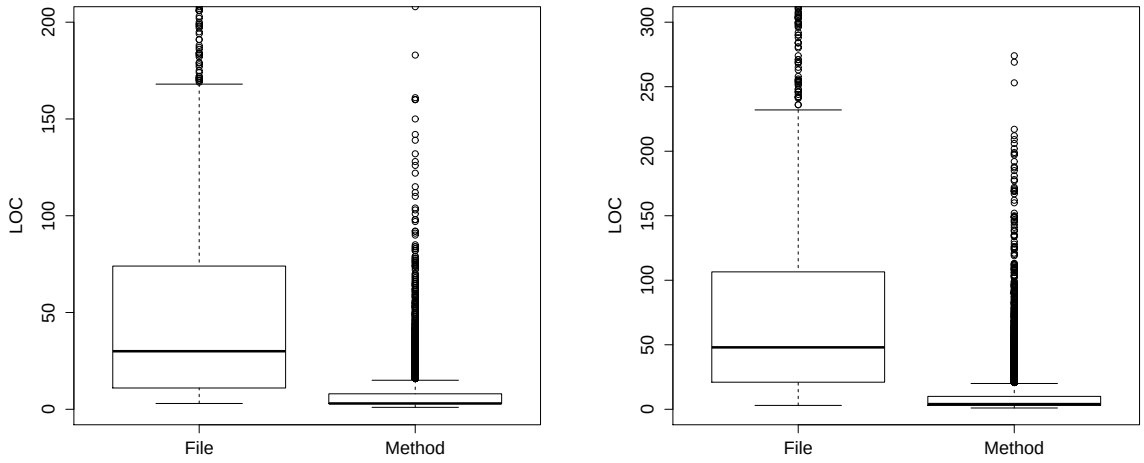


(a) ECF (b) Lucene/Solr

Figure 4.7: Boxplots of file-level and method-level prediction. Percentages of faults found in 20% LOC on a 1,000 run.

Table 4.8: Median values of the percentage of faults found in 20% LOC on the 1,000 run

Project	File	Method	Delta
ECF	0.446	0.748	0.302
WTP Incubator	0.398	0.697	0.299
Xpand	0.233	0.546	0.313
Ant	0.276	0.494	0.218
Cassandra	0.151	0.615	0.564
Lucene/Solr	0.533	0.674	0.141
OpenJPA	0.187	0.521	0.334
Wicket	0.685	0.883	0.198



(a) ECF (b) Lucene/Solr

Figure 4.8: Size of modules, file-level and method-level.

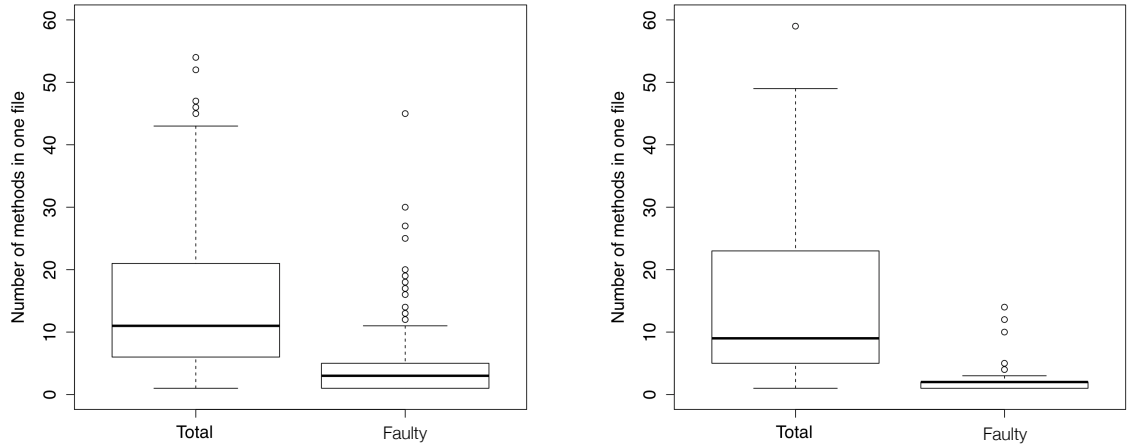
can see that method-level prediction improved by at least 0.14 values from file-level prediction. In the Cassandra project, which records the lowest median value on file-level, method-level prediction improved by more than 0.56. From method-level results, we observed that nearly 50% and more faults can be found during quality assurance activities on 20 LOC.

Based on these results from eight open-source projects, we can answer research question RQ1. The answer is clear: method-level prediction is more cost-effective than file-level prediction.

4.4.2 Why is Method-level Cost-effective?

RQ2: Why are method-level prediction models more cost-effective than file-level prediction models?

Intuitively, fine-grained prediction may more cost-effective than coarse-grained prediction because finding faults in large modules is hard. Figure 4.8 shows box-plots of LOC for files and methods. Comparing the median value of LOC, methods are nearly ten times smaller than files.



(a) ECF (b) Lucene/Solr

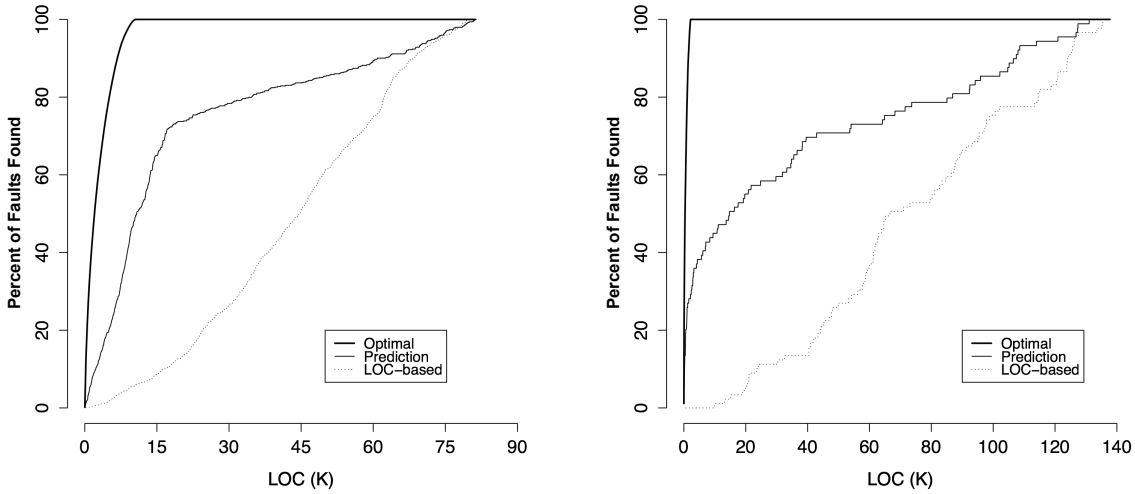
Figure 4.9: Number of total and faulty methods in faulty files.

Next, we investigated faulty files by considering how many methods exist in one file, and how many faulty methods exist in the file. The boxplots of Figure 4.9 present the results. In both projects, most of the faulty files contain nearly or more than 10 methods, but there are only a few faulty methods. From all of the projects, the median values of the number of entire methods range from 8 to22, and the median values of the number of faulty methods range from 1 to 3 methods. Although there are many methods in one faulty file, there are only a few actual faulty methods. This indicates that we need to investigate most of the not faulty methods in a file if the file is predicted as faulty. Because of this cost for file-level prediction results, method-level prediction is more cost-effective.

4.5 Discussion

4.5.1 Effectiveness of Prediction Models

To show the effectiveness of method-level prediction, we present some cost-effectiveness curves in Figure 4.10. Method-level prediction results are repre-



(a) ECF (b) Lucene/Solr

Figure 4.10: Cost-effectiveness curves of optimal, LOC-based ordering, and method-level prediction.

sented as solid lines. Optimal prediction results, that is, complete classification of faulty and non faulty modules, are represented as bold, solid lines. Dotted lines represent LOC-based results, which are the decreasing ordering of module sizes. As shown in Figure 4.10, our prediction curves are nearer to optimal curves than LOC-based ordering curves. With these results, we can see that our prediction models are effective.

4.5.2 Threats to Validity

Target projects are limited to open-source software written in Java. For external validity, there is a threat of generalization of our result. Projects we targeted are only open-source projects written in Java. One of good points of targeting only open-source software projects in Java is that there is no opposite result regarding the effectiveness of method-level prediction compared with file-level prediction.

As described in Section 4.3.2, eight targeted projects varied in size and development period. For example, the Lucene/Solr project has less than two periods,

and prediction is conducted with only a one-year history and yields a good result. This result may promote the adoption of historical metrics based prediction for young projects.

For future work we intend to widen our study to other projects written in other programming languages, and work on industrial projects.

Collection of faulty information has problems. For construct validity, the main threat is in the phase of collecting fault information. Though we adopted a well-known SZZ algorithm discussed in Section 2.1, it has been reported that there is a linking bias in identifying faults with revision logs and fault reports [6].

Recently, a new algorithm of linking faults and changes has been proposed [116]. This algorithm may mitigate this threat.

Effort-based evaluation may not reflect actual efforts. There is a threat to construct validity for our evaluation. To compare file-level and method-level prediction, we adopted an effort-based evaluation with cost-effectiveness curves, which has been researched [2,69,79,83,99]. This effort-based evaluation considers the cost of quality assurance activities to be roughly proportional to the size of the modules, that is, to the lines of code. For coarse-grained modules, such as files and subsystems, it seems acceptable to consider the sizes of modules as effort. However, for methods, it may not be acceptable. For example, though methods are small, they might require much more effort than big methods because of complex call relations or other deep dependencies. Because of this threat, we need empirical investigation of the actual effort of quality assurance activities for different levels of modules.

4.6 Summary

In this chapter, we developed Historage, a fine-grained version control system for Java to conduct fine-grained prediction with recently proposed historical metrics. Using eight open-source projects, method-level and file-level prediction models

are compared based on effort-based evaluation. From the study we clarify that method-level prediction is more cost-effective than file-level prediction.

Contributions of this study can be summarized as follows:

- **Development of a fine-grained version control system,** *Historage***.** To the best of our knowledge, this system is the first storage that can store entire histories of fine-grained modules including renaming and moving changes. We have made our tool publicly available at `https://github.com/hdrky/git2historage`.

- **The first study of fine-grained prediction with historical metrics.** Using traditional metrics, there are some studies conducting fine-grained prediction. Although recent studies have observed that historical metrics are more effective than traditional complexity metrics, it had been difficult to collect fine-grained historical metrics because there had been no technique to obtain entire fine-grained histories. After developing *Historage*, we can first conduct a study of fine-grained prediction. From our empirical evaluation, we clarified that method-level prediction is more cost-effective than file-level prediction.

Chapter 5

Conclusion

5.1 Contributions

In this dissertation, we have addressed fault-prone module prediction using version histories. In sum, this dissertation contributes to the following:

- A systematic review of recent fault-prone module prediction studies.
- Text-mining-based fault-prone module prediction.
- Fine-grained fault-prone module prediction with historical metrics.

First, we presented a survey of recent studies by conducting a systematic review*. Recent findings in empirical results of fault-prone module prediction illustrates the effectiveness of *historical metrics* compared to traditional complexity metrics. We classified the studied metrics into eight categories based on measurement targets (code, process, organization, and geography) and version

*We have provided our survey results at `http://www-ise4.ist.osaka-u.ac.jp/survey/`

information (present version and previous versions). We clarified which metrics are used frequently. We also clarified that newer historical metrics were studied in industry first, and then widely used in studies in open-source software projects. In addition, granularity levels of prediction models were investigated, and revealed that there is no study using well-known historical metrics to build prediction models.

Based on open issues of fault-prone module prediction studies, we developed prediction models: *text-mining-based prediction models* and *fine-grained prediction models*.

Text-mining-based prediction models were developed to tackle the issue of laboriousness in collecting metrics. This issue has been a big barrier for adopting fault-prone prediction models for practical use. For example, when collecting complexity metrics, analysis of source code is needed, and this requires program-language-specific tools, and collecting historical metrics requires software repository mining, which requires repository-specific mining tools. Preparing these tools is a laborious task. To develop easily applicable prediction models, we studied prediction models using a text-mining technique. In this model, the numbers of tokens in source code are considered metrics. Since we only have to count the number of tokens in source code, we do not need specific tools. Using these simple and large-scale token metrics, we built logistic regression and naive Bayes models. We conducted an empirical study with open-source software projects by comparing our token metrics and a well-know metrics suite which includes complexity metrics and some historical metrics, thereby achieving higher prediction results. The results imply that our text-mining-based metrics are useful in building practical prediction models.

Fine-grained prediction models are considered cost-effective. Although there are many studies reporting the effectiveness of historical metrics, they remain at the file level or at a coarser level. Big challenges exist in historical metrics based prediction on fine-grained modules in analyzing version histories to collect met-

rics. Since existing software configuration management systems store file-level version histories, it has been difficult to obtain version histories of fine-grained modules. To tackle this difficulty, we developed *Historage*, a fine-grained version control system[†]. Historage is constructed on top of Git, and can control method histories of Java as well as file histories. With this system, we collected the same historical metrics on method-level and file-level, and built prediction models. Using open-source software projects, we compared both prediction models with effort-based evaluation. The results indicate that method-level prediction models are more cost-effective than file-level prediction models. To the best of our knowledge, this is the first study of fine-grained prediction with well-known historical metrics.

5.2 Future Work

During the work on this dissertation, we encountered some required future research directions. In the following, we discuss future work to overcome the limitations of our studies and to strengthen the support for software maintenance.

(1) Generalization of proposed prediction models

In this dissertation, we developed two types of prediction models: text-mining-based prediction models and fine-grained prediction models. To show the effectiveness of these models, we have conducted empirical studies with open-source software projects written in Java. For generalization of our models, we want to apply our models to different types of projects including industrial projects. In addition, application to projects written in other languages should be required.

(2) Mining historical metrics related to organization and geography in open-source software projects

As seen in Section 2.4, organizational and geographical historical metrics have

[†]A tool to construct Historage is publicly available at `https://github.com/hdrky/git2historage`

not been studied with open-source software projects. Such metrics can not be easily collected for version history information only from publicly available data. Since some papers reported that these metrics are more effective than traditional complexity metrics, and code-related and process-related historical metrics. Therefore, collecting such metrics for open-source software projects too is desirable. In the research area of *mining software repositories*, mining social networks and activities has recently become a hot topic. If we can collect more organizational and geographical metrics, our fine-grained prediction models should improve.

(3) Integrating fault-prone module prediction tools with software development management tools

When considering practical management activities, just predicting fault-prone modules is not helpful enough. We think prediction tools should be integrated with other support tools for software maintenance. The followings are desirable requirements of an integrated system.

- Controlling every module history.
- Collecting metrics automatically.
- Predicting at all development and maintenance phases, such as at committing, during refactorings, before release, and at the fix stage.
- Tracking predicted fault-prone modules. The system supports improving modules so as not to introduce and infect additional faults.
- Reporting prediction results readably. Fault-prone module prediction is not only for developers; prediction should also become mandatory for managers so they can control projects appropriately. Visualization of prediction results should be useful for developers and managers.

(4) Empirical study of software evolution for fine-grained modules

Clarifying the characteristics of software evolution might be useful for improving software quality and preventing faults. To see the evolution, understanding

changes is important. Although *Historage* can track methods if names or paths are changed, it is not possible to distinguish whether the methods are moved or not with the technique stated in Section 4.2.2. To clarify a move or not, we have proposed a technique [48]. With this additional technique, we plan to investigate detailed version histories.

Recently, analyzing code clone evolution has become /an active research area [4, 22, 27, 60, 109]. Although the presence of code clones has been considered harmful, there are some recent empirical case studies that report unexpected results from the analysis of code clones.

- There are long-lived code clone instances that do not need to be avoided [36, 60].
- Though unintentional changes to code clone instances may lead to faults with higher a probability, not all code clones induce faults [53].
- As many as 71% of code clones could be considered to have a positive impact on maintainability [55].

Based on these reports, in the future we think it would be interesting to analyze software evolution related to code clones to improve software quality.

Bibliography

[1] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proc. of 33rd Int. Conf. on Softw. Eng.*, ICSE '11, pages 1–10, New York, NY, USA, 2011. ACM.

[2] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *J. Syst. Softw.*, 83(1):2–17, January 2010.

[3] L. Aversano, L. Cerulo, and C. Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Proc. of 9th Int. Workshop on Principles of Softw. Evolution*, IWPSE '07, pages 19–26, New York, NY, USA, 2007. ACM.

[4] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *Proc. of 11th European Conf. on Softw. Maintenance and Reengineering*, CSMR '07, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society.

[5] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. In *Proc. of 5th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, ESEC/FSE-13, pages 177–186, New York, NY, USA, 2005. ACM.

[6] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proc. of 7th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the*

Found. of Softw. Eng., ESEC/FSE '09, pages 121–130, New York, NY, USA, 2009. ACM.

[7] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. Does distributed development affect software quality? an empirical case study of windows vista. In *Proc. of 31st Int. Conf. on Softw. Eng.*, ICSE '09, pages 518–528, Washington, DC, USA, 2009. IEEE Computer Society.

[8] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proc. of 8th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, ESEC/FSE '11, pages 4–14, New York, NY, USA, 2011. ACM.

[9] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *Proc. of 6th IEEE Work. Conf. on Mining Softw. Repositories*, MSR '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[10] G. Boetticher, T. Menzies, and T. Ostrand. PROMISE Repository of empirical software engineering data. `http://promisedata.org/` repository, West Virginia University, Department of Computer Science, 2007.

[11] C. Boogerd and L. Moonen. Evaluating the relation between coding standard violations and faultswithin and across software versions. In *Proc. of 6th IEEE Work. Conf. on Mining Softw. Repositories*, MSR '09, pages 41–50, Washington, DC, USA, 2009. IEEE Computer Society.

[12] L. C. Briand, W. L. Melo, and J. Wüst. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Trans. Softw. Eng.*, 28(7):706–720, July 2002.

[13] C. Catal and B. Diri. Review: A systematic review of software fault prediction studies. *Expert Syst. Appl.*, 36(4):7346–7354, May 2009.

[14] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Trans. Softw. Eng.*, 35(6):864–878, November 2009.

[15] S. Chacon. *Pro Git*. Apress, Berkely, CA, USA, 1st edition, 2009.

[16] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994.

[17] M. Conway. How do committees invent. *Datamation magazine*, 14(4):28–31, 1968.

[18] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev. CRANE: Failure prediction, change analysis and test prioritization in practice – experiences from windows. In *Proc. of 4th IEEE Int. Conf. on Softw. Testing, Verification and Validation*, ICST '11, pages 357–366, Washington, DC, USA, 2011. IEEE Computer Society.

[19] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proc. of 30th Int. Conf. on Softw. Eng.*, ICSE '08, pages 481–490, New York, NY, USA, 2008. ACM.

[20] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Proc. of 7th IEEE Work. Conf. on Mining Softw. Repositories*, MSR '10, pages 31–41, 2010.

[21] A. De Lucia, F. Fasano, R. Oliveto, and D. Santonicola. Improving context awareness in subversion through fine-grained versioning of java code. In *Proc. of 9th Int. Workshop on Principles of Softw. Evolution*, IWPSE '07, pages 110–113, New York, NY, USA, 2007. ACM.

[22] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *Proc. of 25th IEEE Int. Conf. on*

Softw. Maintenance, ICSM '09, pages 169–178, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[23] G. Denaro and M. Pezzè. An empirical evaluation of fault-proneness models. In *Proc. of 24th Int. Conf. on Softw. Eng.*, ICSE '02, pages 241–251, New York, NY, USA, 2002. ACM.

[24] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *Proc. of 20th European Conf. on Object-oriented Programming*, ECOOP '06, pages 404–428. Springer Berlin / Heidelberg, 2006.

[25] D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen. Effective software merging in the presence of object-oriented refactorings. *IEEE Trans. Softw. Eng.*, 34(3):321–335, May 2008.

[26] D. Dig, T. Nguyen, and R. Johnson. Refactoring-aware software configuration management. Technical report, UIUCDCS, 2006.

[27] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proc. of 29th Int. Conf. on Softw. Eng.*, ICSE '07, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society.

[28] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho. Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.*, 34(4):497–515, July 2008.

[29] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Softw. Eng.*, 27(7):630–650, July 2001.

[30] N. Fenton, M. Neil, W. Marsh, P. Hearty, L. Radliński, and P. Krause. On the effectiveness of early life cycle defect prediction with bayesian nets. *Empirical Softw. Eng.*, 13(5):499–537, October 2008.

[31] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Softw. Eng.*, 26(8):797–814, August 2000.

[32] F. Ferrari, R. Burrows, O. Lemos, A. Garcia, E. Figueiredo, N. Cacho, F. Lopes, N. Temudo, L. Silva, S. Soares, A. Rashid, P. Masiero, T. Batista, and J. Maldonado. An exploratory study of fault-proneness in evolving aspect-oriented programs. In *Proc. of 32nd Int. Conf. on Softw. Eng.*, ICSE '10, pages 65–74, New York, NY, USA, 2010. ACM.

[33] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, November 2007.

[34] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[35] T. Freese. Refactoring-aware version control. In *Proc. of 28th Int. Conf. on Softw. Eng.*, ICSE '06, pages 953–956, New York, NY, USA, 2006. ACM.

[36] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *Proc. of 33rd Int. Conf. on Softw. Eng.*, ICSE '11, pages 311–320, New York, NY, USA, 2011. ACM.

[37] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, February 2005.

[38] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, July 2000.

[39] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng.*, 31(10):897–910, October 2005.

[40] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[41] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[42] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proc. of 31st Int. Conf. on Softw. Eng.*, ICSE '09, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[43] A. E. Hassan and R. C. Holt. C-REX: An evolutionary code extractor for C. CSER meeting, Montreal, Canada, 2004.

[44] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *Proc. of 21st IEEE Int. Conf. on Softw. Maintenance*, ICSM '05, pages 263–272, Washington, DC, USA, 2005. IEEE Computer Society.

[45] H. Hata, O. Mizuno, and T. Kikuno. An extension of fault-prone filtering using precise training and a dynamic threshold. In *Proc. of 5th Work. Conf. on Mining Softw. Repositories*, MSR '08, pages 89–98, New York, NY, USA, 2008. ACM.

[46] H. Hata, O. Mizuno, and T. Kikuno. Fault-prone module detection using large-scale text features based on spam filtering. *Empirical Softw. Eng.*, 15(2):147–165, April 2010.

[47] H. Hata, O. Mizuno, and T. Kikuno. Historage: fine-grained version control system for java. In *Proc. of 3rd Joint Int. and Annual ERCIM Workshops on*

Principles of Softw. Evolution and Softw. Evolution Workshops, IWPSE-EVOL '11, pages 96–100, New York, NY, USA, 2011. ACM.

[48] H. Hata, O. Mizuno, and T. Kikuno. Inferring restructuring operations on logical structure of java source code. In *Proc. of 3rd Int. Workshop on Empirical Softw. Eng. in Practice*, IWESEP '11, pages 17–22, November 2011. Nara, Japan.

[49] I. Herraiz, D. M. German, J. M. Gonzalez-Barahona, and G. Robles. Towards a simplification of the bug report form in eclipse. In *Proceedings of the 2008 international working conference on Mining software repositories*, MSR '08, pages 145–148, New York, NY, USA, 2008. ACM.

[50] I. Herraiz, G. Robles, and J. M. Gonzalez-Barahona. Research friendly software repositories. In *Proc. of 1st Joint Int. and Annual ERCIM Workshops on Principles of Softw. Evolution and Softw. Evolution Workshops*, IWPSE-EVOL '09, pages 19–24, New York, NY, USA, 2009. ACM.

[51] Y. Higo, A. Saitoh, G. Yamada, T. Miyake, S. Kusumoto, and K. Inoue. A pluggable tool for measuring software metrics from source code. In *Proc. of the Joint Conference of the 21th International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement (IWSM/MENSURA2011)*, pages pp.3–12, 11 2011.

[52] G. J. Pai and J. Bechta Dugan. Empirical analysis of software fault content and fault proneness using bayesian methods. *IEEE Trans. Softw. Eng.*, 33(10):675–686, October 2007.

[53] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proc. of 31st Int. Conf. on Softw. Eng.*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.

[54] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware

models. In *Proc. of 26th IEEE Int. Conf. on Softw. Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[55] C. J. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Softw. Eng.*, 13(6):645–692, December 2008.

[56] T. M. Khoshgoftaar and N. Seliya. Fault prediction modeling for software quality estimation: Comparing commonly used techniques. *Empirical Softw. Eng.*, 8(3):255–283, September 2003.

[57] T. M. Khoshgoftaar and N. Seliya. Comparative assessment of software quality classification techniques: An empirical case study. *Empirical Softw. Eng.*, 9(3):229–257, September 2004.

[58] T. M. Khoshgoftaar, X. Yuan, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Uncertain classification of fault-prone software modules. *Empirical Softw. Eng.*, 7(4):297–318, December 2002.

[59] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proc. of 3rd Int. Workshop on Mining Softw. Repositories*, MSR '06, pages 58–64, New York, NY, USA, 2006. ACM.

[60] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proc. of 5th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, ESEC/FSE-13, pages 187–196, New York, NY, USA, 2005. ACM.

[61] S. Kim, K. Pan, and E. J. Whitehead, Jr. When functions change their names: Automatic detection of origin relationships. In *Proc. of 12th Work. Conf. on Reverse Eng.*, WCRE '05, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society.

[62] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, March 2008.

[63] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *Proc. of 21st IEEE/ACM Int. Conf. on Automated Softw. Eng.*, ASE '06, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[64] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proc. of 29th Int. Conf. on Softw. Eng.*, ICSE '07, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[65] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman. Systematic literature reviews in software engineering – a systematic literature review. *Inf. Softw. Technol.*, 51(1):7–15, January 2009.

[66] B. A. Kitchenham, P. Brereton, M. Turner, M. K. Niazi, S. Linkman, R. Pretorius, and D. Budgen. Refining the systematic literature review process–two participant-observer case studies. *Empirical Softw. Eng.*, 15(6):618–653, December 2010.

[67] M. Kläs, F. Elberzhager, J. Münch, K. Hartjes, and O. von Graevemeyer. Transparent combination of expert and measurement data for defect prediction: an industrial case study. In *Proc. of 32nd Int. Conf. on Softw. Eng.*, ICSE '10, pages 119–128, New York, NY, USA, 2010. ACM.

[68] P. Knab, M. Pinzger, and A. Bernstein. Predicting defect densities in source code files with decision tree learners. In *Proc. of 3rd Int. Workshop on Mining Softw. Repositories*, MSR '06, pages 119–125, New York, NY, USA, 2006. ACM.

[69] A. G. Koru, K. E. Emam, D. Zhang, H. Liu, and D. Mathew. Theory of relative defect proneness. *Empirical Softw. Eng.*, 13(5):473–498, October 2008.

[70] A. G. Koru, D. Zhang, K. El Emam, and H. Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Trans. Softw. Eng.*, 35(2):293–304, March 2009.

[71] G. Koru, H. Liu, D. Zhang, and K. Emam. Testing the theory of relative defect proneness for closed-source software. *Empirical Softw. Eng.*, 15(6):577–598, December 2010.

[72] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Softw. Eng.*, 34(4):485–496, July 2008.

[73] A. Liaw and M. Wiener. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[74] Y. Liu, T. M. Khoshgoftaar, and N. Seliya. Evolutionary optimization of software quality modeling with multiple repositories. *IEEE Trans. Softw. Eng.*, 36(6):852–864, November 2010.

[75] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *Proc. of 5th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, ESEC/FSE-13, pages 296–305, New York, NY, USA, 2005. ACM.

[76] S. MacDonell, M. Shepperd, B. Kitchenham, and E. Mendes. How reliable are systematic reviews in empirical software engineering? *IEEE Trans. Softw. Eng.*, 36(5):676–687, September 2010.

[77] M. Mäntylä, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 381–384, Washington, DC, USA, 2003. IEEE Computer Society.

[78] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976.

[79] T. Mende and R. Koschke. Effort-aware defect prediction models. In *Proc. of 14th European Conf. on Softw. Maintenance and Reengineering*, CSMR '10, pages 107–116, Washington, DC, USA, 2010. IEEE Computer Society.

[80] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *Proc. of 16th ACM SIGSOFT Int. Symp. on Found. of Softw. Eng.*, SIGSOFT '08/FSE-16, pages 13–23, New York, NY, USA, 2008. ACM.

[81] T. Mens. The ERCIM working group on software evolution: the past and the future. In *Proc. of 1st Joint Int. and Annual ERCIM Workshops on Principles of Softw. Evolution and Softw. Evolution Workshops*, IWPSE-EVOL '09, pages 1–4, New York, NY, USA, 2009. ACM.

[82] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.*, 33(1):2–13, January 2007.

[83] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Softw. Eng.*, 17(4):375–407, December 2010.

[84] O. Mizuno, S. Ikami, S. Nakaichi, and T. Kikuno. Spam filter based approach for finding fault-prone software modules. In *Proc. of 4th Int. Workshop on Mining Softw. Repositories*, MSR '07, Washington, DC, USA, 2007. IEEE Computer Society.

[85] O. Mizuno and T. Kikuno. Training on errors experiment to detect fault-prone software modules by spam filter. In *Proc. of 6th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, ESEC-FSE '07, pages 405–414, New York, NY, USA, 2007. ACM.

[86] A. Mockus. Organizational volatility and its effects on software defects. In *Proc. of 18th ACM SIGSOFT Int. Symp. on Found. of Softw. Eng.*, FSE '10, pages 117–126, New York, NY, USA, 2010. ACM.

[87] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proc. of 26th Int. Conf. on Softw. Eng.*, ICSE '04, pages 282–292, Washington, DC, USA, 2004. IEEE Computer Society.

[88] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proc. of 30th Int. Conf. on Softw. Eng.*, ICSE '08, pages 181–190, New York, NY, USA, 2008. ACM.

[89] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proc. of 27th Int. Conf. on Softw. Eng.*, ICSE '05, pages 580–586, New York, NY, USA, 2005. ACM.

[90] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. of 27th Int. Conf. on Softw. Eng.*, ICSE '05, pages 284–292, New York, NY, USA, 2005. ACM.

[91] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. of 28th Int. Conf. on Softw. Eng.*, ICSE '06, pages 452–461, New York, NY, USA, 2006. ACM.

[92] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *Proc. of 30th Int. Conf. on Softw. Eng.*, ICSE '08, pages 521–530, New York, NY, USA, 2008. ACM.

[93] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, April 2005.

[94] K. Pan, S. Kim, and E. J. Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315, June 2009.

[95] M. Pighin, V. Podgorelec, and P. Kokol. Fault-threshold prediction with linear programming methodologies. *Empirical Softw. Eng.*, 8(2):117–138, June 2003.

[96] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *Proc. of 16th ACM SIGSOFT Int. Symp. on Found. of Softw. Eng.*, SIGSOFT '08/FSE-16, pages 2–12, New York, NY, USA, 2008. ACM.

[97] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? In *Proc. of 7th IEEE Work. Conf. on Mining Softw. Repositories*, MSR '10, pages 72–81, 2010.

[98] F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proc. of 33rd Int. Conf. on Softw. Eng.*, ICSE '11, pages 491–500, New York, NY, USA, 2011. ACM.

[99] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. BugCache for inspections: hit or miss? In *Proc. of 8th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, ESEC/FSE '11, pages 322–331, New York, NY, USA, 2011. ACM.

[100] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Eng.*, 14(2):131–164, April 2009.

[101] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proc. of 30th Int. Conf. on Softw. Eng.*, ICSE '08, pages 341–350, New York, NY, USA, 2008. ACM.

[102] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A bayesian approach to filtering junk e-mail. In *Proc. of AAAI Workshop on Learning for Text Categorization*. AAAI Technical Report WS-98-05, 1998.

[103] R. Shatnawi. A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. *IEEE Trans. Softw. Eng.*, 36(2):216–225, March 2010.

[104] Y. Shin, R. Bell, T. Ostrand, and E. Weyuker. Does calling structure information improve the accuracy of fault prediction? In *Proc. of 6th IEEE Work. Conf. on Mining Softw. Repositories*, MSR '09, pages 61–70, Washington, DC, USA, 2009. IEEE Computer Society.

[105] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. of 2nd Int. Workshop on Mining Softw. Repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[106] R. Subramanyam and M. S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310, April 2003.

[107] The R Project for Statistical Computing. R. `http://www.r-project.org/`.

[108] D. Thomas and K. Johnson. Orwell – a configuration management system for team programming. In *Proc. on Object-oriented Programming Syst., Languages and Appl.*, OOPSLA '88, pages 135–141, New York, NY, USA, 1988. ACM.

[109] T. G. Tibor Bakota, Rudolf Ferenc. Clone smells in software evolution. In *Proc. of 23rd IEEE Int. Conf. on Softw. Maintenance*, ICSM '07, pages 24–33, 2007.

[110] E. Trucco and A. Verri. *Introductory Techniques for 3-D Computer Vision*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[111] Q. Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. In *Proc. of 10th Int. Workshop on Program Comprehension*, IWPC '02, pages 127–136, Washington, DC, USA, 2002. IEEE Computer Society.

[112] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Softw. Eng.*, 14(5):540–578, October 2009.

[113] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proc. of 21st IEEE/ACM Int. Conf. on Automated Softw. Eng.*, ASE '06, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

[114] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Softw. Eng.*, 13(5):539–559, October 2008.

[115] T. Wolf, A. Schroter, D. Damian, and T. Nguyen. Predicting build failures using social network analysis on developer communication. In *Proc. of 31st Int. Conf. on Softw. Eng.*, ICSE '09, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[116] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proc. of 8th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, ESEC/FSE '11, pages 15–25, New York, NY, USA, 2011. ACM.

[117] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. AURA: a hybrid approach to identify framework evolution. In *Proc. of 32nd Int. Conf. on Softw. Eng.*, ICSE '10, pages 325–334, New York, NY, USA, 2010. ACM.

[118] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *Proc. of 20th IEEE/ACM Int. Conf. on Automated Softw. Eng.*, ASE '05, pages 54–65, New York, NY, USA, 2005. ACM.

[119] Z. Xing and E. Stroulia. API-evolution support with Diff-CatchUp. *IEEE Trans. Softw. Eng.*, 33(12):818–836, December 2007.

[120] R. K. Yin. *Case Study Research: Design and Methods*. Applied Social Research Methods. Sage Publications, 4th edition, 2008.

[121] H. Zhang. An investigation of the relationships between lines of code and defects. In *Proc. of 25th IEEE Int. Conf. on Softw. Maintenance*, ICSM '09, pages 274–283, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[122] Y. Zhou and H. Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Trans. Softw. Eng.*, 32(10):771–789, October 2006.

[123] T. Zimmermann. Fine-grained processing of CVS archives with APFEL. In *Proc. of OOPSLA Workshop on Eclipse Technol. eXchange*, eclipse '06, pages 16–20, New York, NY, USA, 2006. ACM.

[124] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proc. of 30th Int. Conf. on Softw. Eng.*, ICSE '08, pages 531–540, New York, NY, USA, 2008. ACM.

[125] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proc. of 7th Joint Meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM.