On the Relation of Refactoring and Software Defects

Jacek Ratzinger, Thomas Sigmund
Vienna University of Technology
Information Systems Institute
A-1040 Vienna, Austria
ratzinger@infosys.tuwien.ac.at

Harald C. Gall
University of Zurich
Department of Informatics
CH-8050 Zurich, Switzerland
gall@ifi.uzh.ch

ABSTRACT

This paper analyzes the influence of evolution activities such as refactoring on software defects. In a case study of five open source projects we used attributes of software evolution to predict defects in time periods of six months. We use versioning and issue tracking systems to extract 110 data mining features, which are separated into refactoring and non-refactoring related features. These features are used as input into classification algorithms that create prediction models for software defects. We found out that refactoring related features as well as non-refactoring related features lead to high quality prediction models. Additionally, we discovered that refactorings and defects have an inverse correlation: The number of software defects decreases, if the number of refactorings increased in the preceding time period. As a result, refactoring should be a significant part of both bug fixes and other evolutionary changes to reduce software defects.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Evolution—*software defects, prediction*

General Terms

Software evolution, refactoring, mining software archives

1. INTRODUCTION

We investigate the influence of evolution activities such as refactoring on bug fixes required in the future. Prediction models can help us to find out characteristics of files (or Java classes) with or without bug fixes in their history, especially in relation to refactoring. Information gained from that models can support software developers to apply refactoring in a way that reduces error-proneness of software together with an optimization of efforts.

In this work we analyze data from versioning and issue tracking systems of five open source projects: ArgoUML, JBoss Cache, Liferay Portal, the Spring framework, and XDoclet. These projects are developed in Java, whereby each class is usually placed in a separate file. We perform some preprocessing steps to derive non-refactoring and refactoring attributes from that data. These attributes are input into WEKA [11] that generates prediction models.

Our research hypotheses for our case study evaluation are as follows:
H_0: There is no relation between refactorings and the quality of defect prediction.
H_1: Refactoring reduces the probability of software defects.
H_2: Refactoring is more important than bug fixing for software quality.

The remainder of the paper is organized as follows: Section 2 briefly discusses related work with respect to our research question. Then, we describe our prediction model (Section 3), the used methodology (Section 4), and present the results of the five case studies (Section 5). Finally, we draw our conclusions and indicate future work (Section 6).

2. RELATED WORK

Previous works have addressed areas such as refactoring analysis, change type analysis, software metrics, or bug prediction. Some researchers have investigated refactorings based on history information. For example, van Ryssel-berghe has found methods to identify move and inheritance change operations as well as refactorings [9].

The qualification of refactorings also has been addressed in [10] where it was investigated whether or not refactorings are less error-prone than other changes. In contrast to our work, their approach did not use feature and prediction models and did not come to a uniform conclusion for defect prediction.

Fluri *et al.*developed an approach that investigates change types between releases of software entities [3]. In a case study of a medium-sized open source project more than 50% of all change transactions turned out not to be significant structural changes [2]. According to this *ChangeDistilling* approach, we trace change couplings back to co-changed files that correspond to a change transaction. But we do not filter out change coupling groups that were not structurally changed, *e.g.* changes to Javadoc.

Nagappan *et al.* [6] predict the pre-release defect density. They revealed a strong positive correlation between the defect density determined by static analysis and the pre-release defect density gained from testing.

Kim *et al.* [5] screened the versioning history of several open source projects to predict entities and files most fault-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MSR'08, May 10-11, 2008, Leipzig, Germany.
Copyright 2008 ACM 978-1-60558-024-1/08/05 ...$5.00.

prone. In contrast, we do not detect bug-introducing changes, but use the number of bug fixes, detected in the respective target period for our predictions. In line with [5] we identify bug fix revisions by analyzing log files from versioning history.

Khoshgoftaar et al. [4] use classification trees to predict fault-prone modules. The generated trees describe important points of interest (*e.g.*characteristics of the software). Fenton and Neil give a good review of several software metrics and a wide range of prediction models [1].

In previous work [8] we analyzed the versioning history of ArgoUML and the Spring framework to predict refactoring activities. There we used evolution measures extracted from versioning systems as input into classification algorithms to generate the prediction models. These models enable to predict refactorings with high accuracy. Additionally, refactoring-prone and non-refactoring-prone classes can be identified accurately.

In addition to [8], in this paper we are interested in predicting software defects, and therefore we enhanced our evolution measures with refactoring related features.

3. DATA AND PREDICTION MODELS

In this section we describe the formal modeling stage of our data preparation and analysis.

3.1 Evolution Data

The model of the evolution data is composed of information extracted from versioning systems in the following way: Versioning systems such as CVS contain data about files and the change attributes, e.g. change dates, authors of changes, commit messages, and lines of code changed. In a first step we have to reconstruct the change transactions as described in [3]: Two entities (e.g. files) are change coupled, if modifications of one entity usually also affect the other entity. The intensity of change coupling between two entities a, b can be determined by counting all change sets where a and b are members of the same transaction T_n, *i.e.* $C = \{\langle a, b\rangle | a, b \in T_n\}$ is the set of change coupling and $|C|$ is the intensity of the change coupling.

To compute our attribute *targetBugs* we search for changes that have an issue attached of type "bug fix." Additionally, we investigate the commit messages and add changes that do not provide a reference to an issue but contain terms such as "bug", "fix", "solv", etc. The details of the algorithm are described in [8].

3.2 Time Periods and Features

We used two consecutive time periods for our prediction:

- *Feature Period* in which certain properties of software evolution are accumulated into attributes (features) to serve as input to our prediction. All source code modifications within this time period are used to compute a condensed history of each file.
- *Target Period* as the time frame immediately following the feature period, when we count the number of bug fixes. This number defines the data mining target attribute for our case studies.

3.3 Data Mining Features

From the evolution data we compute 110 features that are used for data mining. We separate these features into two groups: non-refactoring and refactoring related features. For the purpose of creating a balanced prediction model, as argued in [1], the features represent several domains such as code measures, team and co-change aspects, or complexity of implemented solution. For a detailed description of the features we refer to [7].

3.4 Classifiers—Data Mining Algorithms

These classifiers separate entities into different groups such as classes with or without bug fixes.

- *C4.5* induces decision trees: it compares one of the input attributes against a threshold value and partitions the input space into distinctive sets.
- *LMT* is a data mining algorithm for building logistic model trees, which are classification trees with logistic regression functions at the leaves.
- *Rip* (Repeated Incremental Pruning) is a propositional rule learner. It uses a growth phase, in which antecedents are greedily added until the rule reaches 100% accuracy. Then in the pruning phase, metrics are used to prune rules until a defined length is reached.
- In *NNge* a nearest-neighbor algorithm is used to build rules using non-nested generalized exemplars.

3.5 Evaluation of Prediction Models

In our analysis of prediction models for bug fixes we use *precision, recall*, and *F-measure* as three essential markers characterizing model performance. These evaluation measures are defined based on rates for true positives, false positives, true negatives, and false negatives [11].

4. METHODOLOGY

4.1 Identifying Refactoring

For generation of refactoring features we do not distinguish between different types of refactorings (*e.g.*extract class or method, etc.). These features cover the fact that developers try to improve the quality of code.

Similar to our previous work [8] we start our identification by searching for texts including "refactor" and then we exclude phrases such as "needs refactoring" to improve the results. For each project we developed between 10 and 20 SQL queries to mark modifications as refactorings. We used a statistical evaluation to estimate the number of refactorings that we correctly identified with our method. Therefore, we took a random sample of 100 modifications for each project and checked whether it was a refactoring or not.

Table 1 shows high rates of correct classifications for each investigated project. As an example, in ArgoUML and Liferay Portal all revisions labeled as "refactoring" actually were refactorings.

4.2 Data Processing with Weka

Weka is a collection of machine learning algorithms for data mining and is used to generate prediction models distinguishing between the class of files "No bugs" and the class "One or more bugs." Next we describe our two steps of creating and analyzing the section models.

(1) In the first step *we create section models*: If two classes ("No bugs" and "One or more bugs") do not have the same

Project	Modifications	Identified Refactorings	Other Changes	False Positives	False Negatives
ArgoUML	100	12	88	0	2
JBoss Cache	100	22	78	1	3
Liferay Portal	100	10	90	0	1
Spring Framework	100	14	86	2	1
XDoclet	100	21	79	1	3

Table 1: Evaluation to Classify Modifications as Refactorings

size, the set containing more instances is subdivided into sections that consist of the same number of instances as in the small set. For example, Project Liferay Portal contains 1816 files that have been changed during the feature period. 1338 files exhibited no bug fixes and 478 files had one or more bug fixes in the corresponding target period The set of instances with no bug fixes is decomposed into three different sections each holding 478 instances, where the last one contains the remaining 382 instances. For the first evaluation we use the first data set with the topmost 478 files of class "No bugs", for the second we use the next 478 files, and for the third we use the bottommost 478 files. As we see, 96 files are used two times, for data set two and data set three.

After the splitting three different models can be generated based on those three sections. Such a model made up of the same number of non-error-prone files ("Bug fixes=0") and error-prone files ("Bug fixes>=1") is called *section model*.

(2) In the second step *we analyze the section models*: To investigate our hypothesis we apply a statistic analysis on the sections models. For hypothesis H_0 we compare the number of experiments where prediction models based only on refactoring related features perform better than prediction models based on non-refactoring features. Hypothesis H_1 is investigated based on a feature indicating the *number of refactorings* compared to the overall number of changes. For each class of files ("No bugs" and "One or more bugs") we analyze the number of predicted instances where the number of refactorings is above a threshold value. The ratio between refactorings and bug fixes is finally used to address hypothesis H_2.

5. RESULTS

We analyzed three different time frames for each project. Every time frame consisted of a feature period and a target period and spanned one year.

5.1 Do refactoring and non-refactoring related features lead to high quality prediction models? (H_0)

We decided to exemplary display the results for section model 1 of one out of the three analyzed time periods per investigated project, whereby classification algorithm C4.5 is used to generate the respective *section models* (Table 2). Although this is only a small sample of the available prediction models, the results are representative for all generated models with respect to model quality.

The results of models generated from refactoring and non-refactoring features are presented using two main columns that are subdivided in a column for "No bugs" and a column for "One or more bugs". Each line provides information about model quality through the F-measure. All models show sufficient good prediction results to form a basis for a further analysis. Especially, the balanced distribution between both bins (files with and without bug fixes) is satisfying. The project Liferay Portal exhibits extraordinary good prediction results with a maximum F-measure of 0.925. Next, the composition of the model trees can be examined more closely, since concrete sequences of the decision rules lead to promising prediction results. Thus we reject the null hypothesis and conclude:

Both refactoring and non-refactoring related features lead to high quality defect prediction models.

For H_1 and H_2 we investigated *section models* of all analyzed projects that contain the refactoring feature of interest (H_1: number *refactoringChanges*, H_2: ratio *bugfixRefactorings*) together with a respective threshold value.

5.2 Is refactoring related with the number of future software defects? (H_1)

The majority of *model sequences* using the feature *refactoringChanges* shows that instances holding a value equal or below a certain threshold value are assigned to bin "One or more bugs" (71.7%), and instances above are assigned to bin "No bugs" (75.0%) (see Figure 1).

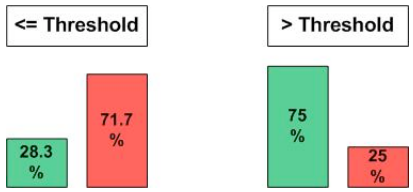

Figure 1: Distribution of non-defect-prone (green) and defect-prone (red) instances in case of a low and a high level of refactorings.

This is an essential result since refactoring can positively influence the software quality by decreasing the occurrence of bug fixes. Additionally, it seems to be generally good when the number of total changes remains low with respect to bug fix reduction. Thus we conclude:

The number of software defects in the target period decreases if the number of refactorings increases as overall change type.

5.3 Does refactoring in contrast to bug fixing reduce software defects? (H_2)

Most of the *model sequences*, using feature *bugfixRefactorings* (i.e. the ratio between bug fixes and refactorings) show that files that have a value equal or below the threshold are assigned to bin "No bugs" (69.2%), and files above are assigned to bin "One or more bugs" (77.8%) (see Figure 2).

Again, refactoring helps to decrease bug fixes in the target period. Furthermore the number of bug fixes in the feature period directly correlates to the number of bug fixes in the

Project	Refactoring Models		Non-Refactoring Models	
	Bugfixes=0	Bugfixes>=1	Bugfixes=0	Bugfixes>=1
ArgoUML	0.718	0.716	0.725	0.718
JBoss Cache	0.747	0.745	0.758	0.742
Liferay Portal	0.925	0.925	0.916	0.918
Spring Framework	0.859	0.851	0.887	0.890
XDoclet	0.798	0.794	0.862	0.845

Table 2: Predicting non Bug fix prone vs Bug fix prone Classes for each project using C4.5

Figure 2: Distribution of non-defect-prone (green) and defect-prone (red) instance in case of a low and a high ratio between bug fixes and refactorings.

target period. This result is well known to the scientific community since defect-prone files tend to stay defect-prone in the course of time. A balanced fraction of refactoring and bug fixes is necessary to support understandability and maintainability of source code, as well as to solve actual or upcoming problems.

Thus, we conclude: *The number of software defects in the target period decreases, if the number of refactorings increases compared to bug fixes.*

5.4 Threats to validity

As many of such studies also our study is undermined by external and internal threats to validity ranging from the case studies chosen to the prediction models computed. For a detailed discussion of all features, classifiers, prediction models and the threats to validity we refer to [7].

6. CONCLUSIONS AND FUTURE WORK

In this paper we investigated the interrelationship of evolution activities such as refactoring to predict software defects in the near future. Our study is based on five open source projects originating from different domains to support some level of generality. Our work extends previous work on refactoring qualification to evaluate the impact on software defect prediction.

We use versioning and issue tracking data to extract 110 data mining features to predict medium-term defects. These features are separated in refactoring and non-refactoring related features and cover software characteristics such as code measures, team and co-change aspects, or complexity of implemented solution.

We found that refactoring related features as well as non-refactoring related features produce high quality prediction models. These findings support our hypotheses that the number of software defects in the target period decreases, if more refactorings are applied and if these refactorings increase compared to bug fixes. This means that an increase in refactorings has a significant positive impact on the quality of the software.

In our future work we will further integrate attributes of software change such as severity levels and improve our queries to further reduce false positives and false negatives with respect to refactoring as well as bug fix detection.

Acknowledgements

We are grateful to the reviewers for their valuable comments. This project was supported by the Hasler Foundation Switzerland as part of the project "ProMedServices."

7. REFERENCES

[1] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, September 1999.

[2] B. Fluri, H. C. Gall, and M. Pinzger. Fine-grained analysis of change couplings. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, 2005.

[3] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *Transactions on Software Engineering*, 33(11):725–743, November 2007.

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

[5] S. Kim, T. Zimmermann, E. J. Whitehead, Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of the International Conference on Software Engineering*, May 2007.

[6] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the International Conference on Software Engineering*, May 2005.

[7] J. Ratzinger. *sPACE – Software Project Assessment in the Course of Evolution*. PhD thesis, Vienna University of Technology, Austria, October 2007.

[8] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall. Mining software evolution to predict refactoring. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, September 2007.

[9] F. Van Rysselberghe. *Studying Historic Change Operations: Techniques and Observations*. PhD thesis, Universiteit Antwerpen, 2008.

[10] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *Proceedings of the International Workshop on Mining Software Repositories (MSR '06)*. ACM, 2006.

[11] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.