Master's Thesis

# Analyzing the Interrelationship of Evolution Activities such as Refactoring to Estimate the Influence on Software Defect Prediction

carried out at the

Information Systems Institute
Distributed Systems Group
Vienna University of Technology

under the guidance of
Univ.Prof. Dipl.-Ing. Dr.techn. Harald Gall
and
Dipl.-Ing. Jacek Ratzinger
as the contributing advisor responsible

by

Thomas Sigmund
Wallensteinstrasse 13/11, 1200 Wien
Matr.Nr. 9107059

Vienna, 31. July 2007 _____

## Acknowledgements

It is a pleasure to thank Prof. Harald Gall for giving me the opportunity to work under his guidance on a so interesting and important topic in the field of software engineering.

I am deeply grateful to Jacek Ratzinger for encouragement, sound advice, and lots of good ideas.

I am indebted to my many student colleagues for providing a supporting and stimulating environment.

Lastly, and most importantly, I wish to thank my parents Anna and Josef Sigmund and my sister Birgit for constant support over the years of my studies. To them I dedicate this thesis.

## Abstract

This thesis analyzes the influence of evolution activities such as refactoring to predict the occurrence of software defects in the near future. Improvement of software quality is the key to reduce error-proneness of software. In this relation non-functional requirements, like understandability and maintainability are of major importance. But how can these requirements be realized? One possibility is to apply refactorings, which affect the design of existing source code, without influencing the external behavior.

In a case study of five open source projects we use attributes of software evolution to predict medium-term defects. We use versioning and issue tracking systems to extract 110 data mining features. These features are separated in refactoring and non-refactoring related features to predict software defects in time periods of six months. Our approach covers software characteristics such as size and complexity measures, relational aspects, time constraints, or team related aspects. This information is 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. Furthermore, we revealed that refactoring should be associated with other software activities in a certain way to reduce software defect appearance. Additionally, we show that certain subsets of refactoring features are of major importance for the prediction models of each project. Concluding, author related aspects showed variable results, nevertheless we could make several interesting findings.

## Zusammenfassung

Die vorliegende Diplomarbeit analysiert den Einfluss von evolutionären Aktivitäten, wie Refactoring, um das Auftreten von Softwarefehlern in naher Zukunft vorherzusagen. Die Verbesserung der Softwarequalität ist der Schlüssel, um die Fehlerdisposition von Software zu reduzieren. In diesem Zusammenhang sind nicht-funktionale Anforderungen, wie Verständlichkeit und Wartbarkeit, von größter Wichtigkeit. Aber wie können diese Erfordernisse realisiert werden? Eine Möglichkeit besteht darin Refactorings anzuwenden, welche das Design von vorhandenem Sourcecode betreffen, ohne das Verhalten nach Außen zu beeinflussen.
Im Rahmen einer Fallstudie mittels fünf Open-Source Projekten verwenden wir Eigenschaften der Softwareevolution, um mittelfristige Fehler vorherzusagen. Wir verwenden Versionierungs- und Bugtracking-Systeme, um 110 Datamining Features zu extrahieren. Diese Features lassen sich in solche, mit und ohne Bezug zu Refactoring einteilen, um Softwarefehler in Zeitabschnitten von sechs Monaten vorherzusagen. Unser Ansatz umfasst Softwarecharakteristika, wie Größen- und Komplexitätsmaße, relationale Aspekte, Zeitbedingungen, oder teambezogene Aspekte. Diese Informationen dienen als Input für Klassifikationsalgorithmen, die Vorhersagemodelle für Softwarefehler erzeugen.
Wir haben herausgefunden, dass Features mit und ohne Refactoring-Bezug zu Vorhersagemodellen von hoher Qualität führen. Weiters haben wir entdeckt, dass Refactoring auf eine bestimmte Art mit anderen Softwareaktivitäten assoziiert sein sollte, um das Auftreten von Softwarefehlern zu reduzieren. Zusätzlich zeigen wir, dass gewisse Teilmengen von Refactoring Features von größter Wichtigkeit für die Vorhersagemodelle jedes einzelnen Projekts sind. Abschließend zeigten autorenbezogene Aspekte variable Ergebnisse, dennoch konnten wir mehrere interessante Erkenntnisse gewinnen.

# Contents

# List of Figures

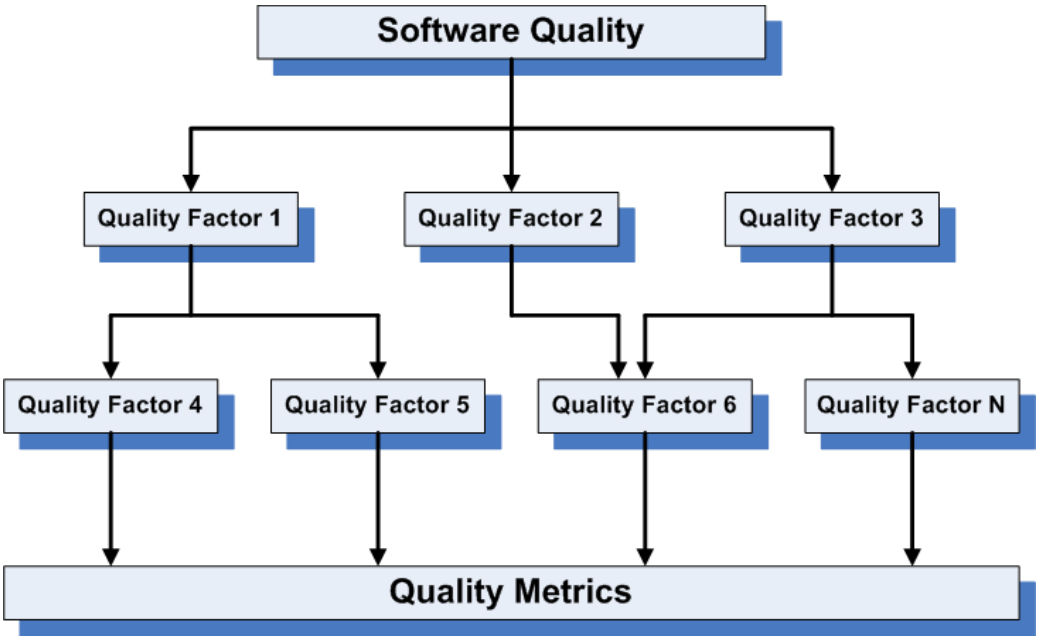


Figure 1: Software Quality Model

# 1 Problem Description

## 1.1 Introduction

Software engineering is a complex process that consists of a number of steps to produce reliable software. The development process is separated in certain phases, which are restricted with respect to time and content. Key processes are planning, analysis, concept, programming, validation and verification. Supporting processes are project management, quality management, configuration management, and documentation.
Software development is a continuous process, hence it is not sufficient to produce software without any further input. In the course of time non-functional requirements, like maintainability and understandability, amongst others, are watered down, because of continuous software development to fulfill new tasks. Additionally, several bug fixes have to be made to solve certain problems with already existing software parts no longer working correctly.
Software quality is specified with the aid of quality models that are visualized as trees made up of quality factors, and quality metrics, as leafs (Figure 1).

IEEE Standard 1061 defines a quality metric as a function that maps a quality factor to a measured value [3]. Quality metrics stretch across several areas, e.g. process, product, costs, time, complexity, and business. Bug fixes are attached to the field of process metrics.
Software quality is strongly influenced by necessary bug fixes. So it is the software engineer's intent to reduce those bug fixes for the future. Refactoring, a widely spread technique in software engineering, improves non-functional requirements, like maintainability and understandability and therefore helps to positively influence software quality. Refactorings are structural changes that do not influence the functionality of a software system.
This thesis investigates the influence of evolution activities such as refactoring on bug fixes required in the future, an important measure of software quality.

## 1.2 Motivation

Each software project is subjected to restrictions that ultimately pertain monetary facts. Any changes required to extend or improve existing software generate costs. As a result of limited funds, refactoring should only be applied where useful. That is why, an estimation of costs to apply refactoring has to be related to the possible improvement of software quality. If this proportion is acceptable, refactoring generates short-term additional expenses, but in the long run these inputs pay off.
Prediction models can help us to find out characteristics of classes, with or without bug fixes, in relation to refactoring. Those properties of classes regarding refactoring affect several areas of software development, represented as size measures, team aspects, complexity measures, relational aspects, and time constraints. Information gained from that models can support software developers to apply refactoring in a way that reduces error-proneness of software together with optimal usage of funds.

## 1.3 Problem Definition

In line with this thesis, the relationship between evolution activities such as refactoring and the prediction of software defects is analyzed. Moreover, we are interested in information that allows us to decide where and when to apply refactoring. A number of research questions are discussed to deal with that problem.

How should refactoring be associated with other software activities? Is an increase in refactoring activities accompanied by a decrease in software defect appearance? Are certain refactoring attributes of major importance to predict bug fixes in the future? Are team related aspects, especially in relation to refactoring, relevant to predict bug fixes?
The main question is to what extent regularly applied refactorings influence the appearance of software defects in the source code. In other words, does refactoring measurably improve software quality?

### 1.3.1 Hypotheses

In line with the thesis we are investigating a number of hypotheses:

1. Both, refactoring related features and non-refactoring related features lead to high quality prediction models.
2. The quality of prediction models is improved by combining features of both groups.
3. Refactoring should be associated with other software activities in a certain way to reduce software defect appearance.
4. Certain subsets of refactoring features, are of major importance for the prediction models of each project.
5. There is a common subset of refactoring features important for all investigated projects.
6. It is essential that the number of authors and author switches, and the number of people switches remains low to reduce error-proneness.
7. It is essential that the number of bug fix authors and bug fix author switches remains low to reduce error-proneness.
8. It is essential that the number of refactoring authors and refactoring author switches remains low to reduce error-proneness.
9. It is essential that the same author, who made the last revision refactors the code to reduce error-proneness.

## 1.4 Organization of this thesis

After the problem description chapter two depicts the state of the art in several relevant information areas.
Papers concerning *Software Evolution* analyze software development and the evolution of object-oriented source code, investigate the design of a software system, analyze change coupling groups and the significance level of changes, and describe fine-grained analysis of CVS data.
Papers related to *Software Quality* investigate coupling and cohesion measures for object-oriented systems and fault-prone classes to rate the quality of software, and try to identify characteristics of files that can predict fault-proneness based on information from previous releases.
Research work in the field of *Software Metrics* investigates object-oriented design metrics for predicting fault-prone classes, describes metrics of Chidamber and Kemerer, and uses metrics to understand and control the software evolution process.
Research papers in the area of *Prediction* show that well selected models can predict software fault-proneness across different applications, examine software decay by using change history data, use certain sets of predictors, and try to quantify the probability of change in future releases.
Papers regarding *Refactoring* analyze software systems, to find out, whether refactoring occurred, validate heuristics for identifying refactoring, try to measure the maintainability enhancement effect of program refactorings based on coupling metrics, and describe the process of refactoring.

Promising techniques to study further describes research work that exhibits methodology valuable to follow up.

In chapter three we describe two fundamental techniques to track software evolution. On the one hand, the versioning system (*e.g.* CVS) to keep all changes at the source code level, and on the other hand the bugtracking system (*e.g.* JIRA) to systematically capture every issue.
After that, the reconstruction of transactions and the time periods—feature period and target period—used for our analysis are described. Our datamining features are subdivided in refactoring features and non-refactoring features and their meaning is explained in detail. Concluding, we enumerate the classifiers, respectively data mining algorithms used for our predictions.

Next, chapter four covers our methodology to extract versioning and bug-tracking data into a relational database, to process that data with Weka [26], a datamining tool, and to analyze the generated prediction models.
Thereby, the data extraction is subdivided in a number of process steps: import versioning data, compute commit transactions, identify refactoring, import issue data, connect issues and revisions, relate accounts of issues and revisions, and calculate features. Data processing is made up of the following working steps: import edited files, discretize target attribute, condition data sets, generate prediction model, and process result file. Finally, data analysis is made up of several building blocs: quality of prediction, association of refactoring and other software activities, subsets of important refactoring features, and influence of author related activities.

Chapter five outlines the evaluation of the prediction models. Due to the fact that a nominal target attribute is used for our predictions, appropriate measures have to be used. Recall, precision, and f-measure help to quantify the quality of the prediction models. Next, the open source projects under investigation are described, together with information on project start and project size.
In section results, we discuss nine hypotheses to answer numerous questions. At the beginning, we show that the generated models predict our target attribute—bug fixes in the target period—sufficiently well. Based on that fact, we try to answer further research questions. We try to find out, whether refactoring is related to other software activities in a certain way and more specific, if refactoring helps to decrease software defect appearance in the future. Moreover, we try to find out subsets of refactoring features important to each single open source project, or more generic to all projects. Concluding, we investigate the influence of author related attributes on software defect occurrence.

Finally, chapter six summarizes our research results and lists possible topics for future work.

# 2 Related Work

## 2.1 Review of the State of the Art

The related work can be assigned to certain research areas.

### 2.1.1 Software Evolution

Antoniol *et al.* [2] analyze the evolution of object-oriented source code at the class level. Thereby, class evolution discontinuities are identified that reveal refactoring activities. A case study of an Java open source domain name server identified nearly all applied refactorings.
In contrast to Antoniol *et al.*, we consider refactorings in general and determine refactoring based on the commit messages that are part of the versioning system.

Capiluppi *et al.* [7] focus on the investigation of the structure of source code and thereby, consider structure as a proxy for the conceptual architecture of an application. A Goal-Questions-Metrics approach is used to analyze decay and refactoring of several open source projects. The authors found out that understandability was increased by refactoring in several projects.
In dependence on this work, we found out that refactoring activities help to decrease software defect appearance.

Fischer *et al.* [17] introduce an approach for populating a release history database. Thereby, versioning data and bugtracking data are combined, and additionally missing data—such as merge points—is added. The authors investigated the design of the open source project Mozilla based on evolution data, extracted from source code management systems.
Our work is based on the techniques outlined in this research work to extract data from versioning and bugtracking systems.

Fluri *et al.* [19] use an approach that adds structural change information to existing release history data. In a case study applied to a medium-sized open source project more than 50% of all transactions turned out, not to be triggered by structural changes.
Ying *et al.* [55] extend existing static and dynamic analysis to better reveal

dependencies between existing source code parts. The authors developed an approach to detect change patterns by using data mining techniques that are applied to the change history of projects Eclipse and Mozilla. Together, with the evaluation of the predictability of the recommendations for actual modification tasks, valuable dependencies were gained.
Common to [19] and [55], we trace change couplings back to commonly commited modified files that correspond to a transaction. In contrast to [19], we do not filter out change coupling groups that were not structurally changed, *e.g.* changes to Javadoc or insertion of whitespace.

Fluri and Gall [18] present a taxonomy of source code changes that defines source code change types with tree edit operations on the abstract syntax tree. The aim is to analyze, if change couplings between source code entities are significant or only minor textual adaptions. A case study using open source project ArgoUML showed, that lines added and/or deleted do not suffice to identify the significance level of source code changes.
In addition to our work, this approach allows to relate change couplings to the significance of identified change types, and based on this information more relevant change couplings can be extracted.

Lehman and Ramil [34] state that most of the software regularly used in business can not be specified and implemented in its entirety. Implementation and maintenance activities depend on the knowledge gained from every day usage, and are influenced by internal and external changes that affect the usability of software. After discussing technology, software process, and related domains in relation to software evolution, facets of the evolution phenomenon are outlined, together with a discussion of their influence on the evolution process.

Mockus and Votta [36] try to answer several hypotheses, *e.g.* that a textual description of a change is the basis to understand the reasons for that change. The authors developed a program to automatically classify maintenance activities based on textual description of changes.
In the style of this research work, we use the commit message of a revision to determine, if that revision is a bug fix or a refactoring. Hereby, we base our decision on extensive queries that contain several keywords representing a certain change type.

Nierstrasz [42] analyzes software development and criticizes only technology-centric views, which consider software rather as a static object, than as a living and evolving entity. The author presents some assertions to think about, enumerates the key difficulties of software development from his view and completes with recommendations for research of software practices that include software evolution.

Zimmermann *et al.* [57] try to give guidelines that support programmers who apply changes to a software system. The authors developed a ROSE tool that suggests and predicts likely changes, after a certain change has been made. The appliance of the tool on several open source projects showed attractive rates of correct predictions for further files to be changed.
In contrast to this work, we do not try to predict further changes after an initial change, but try to predict future bug fixes. Although, we also use coupling metrics to support our predictions.

Zimmermann and Weißgerber [56] present four essential preprocessing tasks important for fine-grained analysis of CVS data. The authors address data extraction, restoring of transactions, mapping of changes to fine-grained entities, and data cleaning.
In our work, we also use the basic techniques to import CVS data and calculate commit transactions. In contrast to this work, we do not map changes to fine-grained entities, but analyze the influence of changes on future bug fixes. Furthermore, we use information on large transactions condensed in a calculated feature to support our predictions.

### 2.1.2 Software Quality

Briand *et al.* [5] investigate coupling and cohesion measures for object-oriented systems and fault-prone classes to rate the quality of software. The authors applied their approach on an industrial case study and revealed that especially method invocation and import coupling are strong indicators for error-proneness. Brito e Abreu and Melo [6] use a suite of object-oriented design metrics to estimate software quality of eight small-sized information management systems. The authors analyzed attribute and method inheritance, polymorphism, and coupling to reveal that software design has a strong influence on defect appearance.

In our work, we also use coupling metrics as input for the generated prediction models. Additionally, we investigate coupling measures with respect to refactoring to refine our analysis.

D'ambros and Lanza [10] use a visual approach to show the relationship between software evolution and software bugs. The authors could show the impact of diverse change patterns on software evolution at any level of granularity, based on a case study of three large open source projects. Ratzinger *et al.* [47] developed and tested a visualization tool based on temporal lens views to explore evolution data across multiple dimensions. The focus is on revealing change couplings, architectural shortcomings, and detection of internal and external dependencies. The authors applied their tool on a large industrial case study implemented in Java.
In our work, we also use versioning data to extract several features to assess the evolution of a certain software system over the course of time. In contrast, we present no graphical support, but try to predict the influence of software attributes on the occurrence of software defects in the future.

### 2.1.3 Software Metrics

Chidamber and Kemerer's metrics are described in [8] and [9]. Several design metrics were developed and later on analytically evaluated against a set of measurement principles. An automated tool collected an empirical sample of these metrics to approve their usefulness.
Basili *et al.* [4] present the results of a case study, which investigates object-oriented design metrics. On the basis of empirical and quantitative analysis, the authors discuss advantages and disadvantages of these metrics and reveal their usefulness during early phases of development for predicting fault-prone classes.
Demeyer and Mens [13] classify a number of approaches that use metrics to understand, predict, plan, and control the software evolution process. The authors subdivide their classification in predictive analysis—before the evolution, and retrospective analysis—after the evolution occurred. For each category, concrete examples and references to the literature are given.
On the basis of Chidamber and Kemerer's metrics and in line with the other research works we use a large set of evolution metrics to predict the occurrence of software defects. Additionally, we extended our metrics suite with metrics especially adapted for analysis of refactoring activities.

### 2.1.4 Prediction

Denaro and Pezzè [14] show that well selected models can predict software fault-proneness across different applications. The prediction models are based on data of Apache 1.3, whereby the most accurate models are applied on data of Apache 2.0. The authors could achieve reliable results to predict error-proneness. In line with this work, we can also achieve significant prediction results based on data of each investigated open source project within our case study. We take these results as a basis to receive an impression of interrelationships in the area of software evolution.

Graves *et al.* [23] examine software decay by using change history data of a long-lived software system. Measurements are investigated with respect to their ability to predict the distribution of faults over certain modules. The authors revealed that process measures based on the change history are better measures than product metrics of code. We also base our evolution metrics on change history and regard relative measures as better indicators of software quality than absolute measures, like lines of code added.

Khoshgoftaar *et al.* [28] use a set of predictors to build software quality models based on classification-tree modeling. The authors investigated four releases of a very large telecommunications system to discover fault-prone modules. Thereby, they discovered that additionally to product metrics, process metrics and execution metrics can efficiently predict fault-proneness. These results show similarities to the research work of Graves *et al.* [23].
Khoshgoftaar *et al.* [29] extend their previous work [28] with statistical techniques to discover uncertain predictions of the classification trees. The authors assessed based on the telecommunications system to what extent modules are wrongly assigned to the group of error-prone modules.
In line with both research works, we use a set of metrics that attach great importance to process metrics amongst others. In contrast to these works, we investigate the error-proneness of software entities at the class level.

Mockus and Weiss [37] use predictors based on lines of code measures, change couplings, change types, and measures of developer experience. In a case study, the authors use their prediction models developed as a web-based tool and found out that change couplings and developer experience are essential to predict error-proneness.

In our work we use similar measures extended with refactoring related information.

Nagappan *et al.* [39] use an approach to early predict the actual pre-release defect density for Windows Server 2003 based on the defects detected with static analysis tools. The authors revealed a strong positive correlation between the defect density determined by static analysis and the pre-release defect density gained from testing. Additionally, the actual pre-release defect density and the predicted pre-release defect density strongly correlate.
Nagappan *et al.* [41] investigated the post-release defect history of five Microsoft software systems. The authors revealed for each project a set of complexity metrics that statistically correlates with post-release defects. Additionally, they found out that predictors gained from principal component analysis are useful to build appropriate regression models and that predictors of a given project are only applicable to the same project or related projects.
Schröter *et al.* [51] build models to predict failure-prone components based on software design and past failure history. In a case study of 52 Eclipse plugins the authors revealed that prediction on the package level works best. Additionally, the classification yields the best results for components that are ranked as most failure-prone. Furthermore, models trained in a previous version are good predictors of failure-prone components in later versions.
Ostrand and Weyuker [44] try to identify characteristics of files that can predict fault-proneness on the basis of a large industrial inventory tracking system. The authors try to discover the distribution of faults over different files and are interested in the influence of the size of modules on their fault density. Furthermore, the propagation of faults from one release to another and the behavior of newly introduced files with regard to fault-proneness is analyzed.
Ostrand *et al.* [45] use the same inventory system as for [44], extended with several releases to predict those files that will most likely exhibit the largest number of faults in the next release. The authors used a negative binomial regression model based on information from previous releases and achieved extremely accurate and correct predictions of classes with high rates of faults.
In our work, we do not distinguish between pre-release and post-release defects as in [39], [41], [44], and [45], but generally analyze selected time frames as temporal snapshots of the projects under investigation. In contrast to [41], who chose modules as the entities to investigate and [51], who compare packages to files with respect to prediction quality, we operate at the class level.

Tsantalis *et al.* [53] use a probabilistic approach to evaluate the change proneness of an object-oriented software system in future releases. The authors applied their approach to two open source projects and determined a significant correlation between the predicted probabilities for change and the actual changes in the investigated classes.

Kim *et al.* [30] analyze bug-introducing changes to identify important properties of software bugs. The authors use algorithms to automatically and accurately detect those changes that are unlike bug-fixes, hard to obtain. After a series of accurate pre-processing steps to refine the data basis, examples of bug-introducing changes verify the usefulness of this approach.
Kim *et al.* [31] screened the versioning history of several open source projects to predict entities and files most fault-prone. In their approach fault-prone locations are cached to later on support developers that just fixed a fault, in finding those hot spots. The major advantage of the proposed cache model is the adaptability to new fault distributions.
In contrast to [30] and [31] 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 [30] and [31] we identify bug fix revisions by analyzing log files from versioning history.

Ratzinger *et al.* [49] use a mining approach to predict short-term defects based on evolution measures extracted from versioning and bugtracking systems. The authors compare the predictability of different classes of software defects by using an exact numerical prediction of defects on the basis of regression models. Additionally, certain components are classified as defect-prone based on the C4.5 decision tree.
The evolution measures used in this research work correspond to our non-refactoring features. In contrast to this work, we do not distinguish between different kinds of software defects and do not differentiate between pre-release and post-release defects.

Ratzinger *et al.* [50] analyze the versioning history of ArgoUML and the Spring framework to predict refactoring activities in the future. The authors use evolution measures extracted from versioning systems as input into classification algorithms to generate their prediction models. The models make it possible to predict refactoring with a high accuracy. Additionally,

refactoring-prone and non-refactoring-prone classes can be identified very accurately.
Our research work is based on the approach we used for [50] in many respects. We use the same open source projects extended with JBoss Cache, Liferay Portal, and XDoclet. Additionally, the non-refactoring features, the classification algorithms, and the identification of refactorings are the same. In contrast to [50] we are now interested in predicting software defects, and thereby enhanced our evolution measures with refactoring related features. Additionally, we import data from bugtracking systems of each investigated project into our relational database to support our predictions.

### 2.1.5 Refactoring

Fowler [20] describes the process of refactoring and explains how to do the various refactorings. Our research work is based upon the fundamentals outlined in this crucial work in the field of software engineering.

Opdyke [43] defines a number of refactorings that help to support software design, software evolution and reuse of object-oriented application frameworks. The thesis aims to automate refactorings in a way that preserves the behavior of software.

Mens and Tourwé [35] present an extensive overview of current research work in the field of software refactoring. The authors discuss refactoring activities, techniques and formalisms that support these activities and the types of software entities that are being refactored. Additionally, important issues related to the generation of refactoring tool support, and the influence of refactoring on the software process are discussed.

Advani *et al.* [1] investigate a range of open source projects in multiple versions to find out locations where refactoring took place and which were the most common types of refactoring. The authors used an automated tool to detect fifteen different refactorings from the pool of 72 refactorings proposed by Fowler [20]. In the majority of cases less complex refactorings occurred and the most common refactorings empirically revealed, were central to larger refactorings.
Demeyer *et al.* [12] evaluate locations where the implementation has changed and thereby, apply object-oriented metrics to successive versions of software

systems to find out refactorings. The authors use a tool to extract classes, methods, and attributes from three case studies. The advantages of the used heuristics are a good focus on small software parts with reliable identification of refactorings.
Kataoka *et al.* [27] quantitatively determine the maintainability enhancement effect of program refactoring. Thereby, the authors estimate the enhancement effect based on the comparison of the coupling before and after applying a certain refactoring. The proposed method shows good results in evaluating the refactoring effect and assists in choosing adequate refactorings.
Ratzinger *et al.* [48] use an approach to detect change smells. Refactoring of those software parts supports the evolvability of software. The authors analyzed a large software system and detected locations for refactorings based on change coupling. The application of these refactorings showed that combining change coupling analysis with refactoring works fine.
In contrast to [1], [12], [27], we use evolution measures based on refactoring as a input for our predictions to determine software defect appearance in the future, in common with these research works refactorings are detected that are used for our predictions. In contrast to [1], we do not focus on a set of selected refactorings.

### 2.1.6 Tools

Witten and Frank [54] describe machine learning tools and techniques used by Weka—a datamining tool. The authors explain how datamining algorithms work and help to select appropriate approaches to certain problems. In addition, techniques to improve performance and the usage of the Weka machine learning workbench are explained.

## 2.2 Promising Techniques to study further

Antoniol *et al.* [2] use an automatic approach, inspired on vector space information retrieval, to identify cases of possible refactoring. Thereby, classes are mapped into elements of vector spaces, namely vectors that contain the information about identifiers extracted from each class. The similarity of two classes is calculated based on the cosine of the angle between the vectors that represent each of the investigated classes. A value of 1 means that the classes are identical. Now, several variants of refactoring are analyzed, whereby a defined threshold value for the cosine determines, whether a certain type of refactoring happened. The threshold values vary, depending on the system size and consequently on the search space, or depending on the desired recall, even so the effort rises to check the results. The authors use different tools to support their approach, *e.g.* to extract identifiers from Java source code, and the implementation of the applied search algorithm.

Fluri *et al.* [19] use several plugins of the Eclipse IDE to access CVS repositories and compare source code files. First of all, the information gained from modification reports of the CVS repository is stored in a release history database. Afterwards, the change coupling relationships between source code files are calculated, and the corresponding change coupling clusters are determined. A change coupling group is made up of a set of files that were commited together more than once in the evolution of a software system. Each change coupling group can be expressed as a revision vector that contains the revision number of each involved file. A change coupling cluster is consequently represented as a matrix with revision vectors as columns. In a further process step source code files of successive revisions—extracted from CVS—, are structurally compared. The authors developed an Eclipse plugin to automatically extract structural changes: All revision numbers of a certain source file are determined by parsing the CVS log information. After this, every revision and its subsequent revision of that file is structurally compared, and the comparison results are stored in the release history database. This plugin can be applied to a single file or even a whole project. Finally, a change coupling filter combines the information of these process steps. The information of structural differences and change coupling clusters is used to evaluate those change couplings that underwent structural changes. To get a better overview, the authors developed a change coupling cluster browser that allows to easily search for change coupling groups, change coupling clus-

ters, and their structural changes.

Fluri and Gall [18] divide a class within an object-oriented programming language (OOPL) in body- and declaration-parts: class body and method body as well as class declaration, attribute declaration, and method declaration. The described taxonomy is based on the abstract syntax tree (AST) with statements as the smallest entities. A source code entity can be every language construct provided by an OOPL, and is represented as an sub-AST or a leaf within the given AST. ASTs are made up of entity nodes with labels—representing the kind of source code entities, and values—the textual representation depending on the kind of entity. Source code changes are based on elementary tree edit operations, since source code changes transform an AST, which is a rooted tree. The authors use as elementary tree edit operations insert, delete, and substitute of a tree node. The significance level of a change is defined as the impact of that change on other source code entities. Additionally, it is relevant for the significance level whether that change is functionality-preserving or functionality-modifying. The authors use the significance levels low, medium, high, and crucial. A detailed listing of a multitude of source code changes that concern body-part changes as well as declaration-part changes is presented. The authors developed an Eclipse plugin that implements the source code change extraction algorithm of Chawathe et al. to extract source code changes into a relational database. Later on these changes are classified by the criteria used for the presented taxonomy.

# 3 Prediction Foundation

## 3.1 Versioning System

Raw data is gained from versioning systems such as CVS (Concurrent Versions System), which allows to handle different versions of files in cooperating software development teams. Certain classes of the software system are checked out by each developer. Now these files can be edited and the changes are afterwards commited to the versioning system, which merges all modifications from different developers. CVS makes it possible to get each version of all versionized files based on the date or version number. CVS logs every action to keep the history of a certain file, which provides the necessary information about the history of a software system. All information for our datamining approach—pure textual, human readable information—is retrieved via standard command line tools. These tools create log-files about all modifications in the past and are parsed and stored in a relational database [17].

## 3.2 Bugtracking System

The issue tracking system JIRA is used to manage data about project issues, like bug reports, patches, improvements, or feature requests. It allows the search for distinct requirements and their implementation, if already done, on project level. We extract the issue data from that tracking system by using the export functionality into an excel file. Hereby, all bug reports, concerning already fixed bugs are the matter of our interest. The information of these files is imported into our release history database. After that, we connect issues to revisions by using the CVS log information. Thereby the commit messages entered by developers are searched for issue numbers. Additionally, we try to relate versioning authors to issue reporters to distinguish between issues created by developers and issues created by customers. If a certain issue reporter can be linked to a CVS author the issue is assigned to a developer, otherwise it is assigned to customer related staff such as service personnel. Finally, we conclude our data-processing steps by applying comprehensive regular expressions on the commit messages of all versionized files to find out bug fixes not revealed by now.

## 3.3 Evolution Data

The data model of the evolution database is composed of information extracted from versioning systems in the following way. Versioning systems like CVS strongly relate revisions of files to modification reports. They contain past data about files, e. g. change dates, authors of changes, commit messages, and lines of code changed. Each new revision replaces the code of the former one, which is maintained in modification reports. Collections of certain revisions of all files managed by the particular versioning system, are called releases. If a developer commits changes to several files at once, only the time stamp of the these new revisions is stored. Hence, we have to reconstruct the transaction information in a post-processing step.

### 3.3.1 Reconstructing Transactions of Versioning Systems

Transactions $T_n$ are sets of files that were checked-in into the versioning system by a single author with equal commit message within a short timeframe, typically a few minutes. We use a dynamic time adaption approach to capture the entire transaction. Each transaction can take several minutes, so we initially set each transaction to last 60 seconds. Every revision with same author and commit message within the transaction window is added to that transaction. Now the transaction window is expanded to last further 60 seconds, based on the time stamp of last detected change event within that transaction. Transactions are necessary to evaluate change couplings between software entities. Change couplings are defined as follows [50]:

> *Two entities (e.g. files) are coupled, if a modification to the implementation affected both entities.* The intensity of coupling between two entities $a$, $b$ can be determined easily by counting all log groups where $a$ and $b$ are members of the same transaction, i.e., $C = \{\langle a, b\rangle | a, b \in T_n\}$ is the set of change coupling and $|C|$ is the intensity of coupling.

## 3.4 Time Periods for Analysis

We use two different time periods to obtain relevant attributes for our experiments:

- *Feature Period* is a time frame where certain properties of software evolution are accumulated into a lot of features (attributes) to serve

as a input to our prediction. All modifications to source code within this time period are used to compute a condensed history of each file. All datamining features are created for object-oriented classes, due to the fact that files are (roughly) equal to classes in the Java programming language. Most of the time, one java file contains only one class. For example all lines added, changed, or deleted are counted together to measure the changes of a certain class (instance). Classes that are modified together are treated in a similar way, based on commit transactions described in section 3.3.1

- *Target Period* is the time frame immediately after the feature period, where we count the number of bug fixes. This number defines the datamining target attribute for our case study. Naturally, a number of other target attributes can be calculated, depending on demand.

## 3.5 Data Mining Features

From evolution data stored in our release history database we compute 110 features that are used for data mining. We separate these features in two groups, namely non-refactoring and refactoring features. These evolution measures are gathered on file basis, whereas data from all revisions within a predefined time period is summarized. For the purpose of creating a balanced prediction model the features represent several important information areas such as team and relational aspects, process orientation, and complexity of implemented solution, etc. [16]. Since previous studies showed that relative features outmatch absolute features with regard to prediction performance, we use only relative features [32, 40].

### 3.5.1 Non-Refactoring Features

**Size.**
We group size measures such as lines of code from an evolution point of view. Attributes *linesAdded*, *linesModified*, or *linesDeleted* are related to the total *LOC* (lines of code) of a file. These features reflect a certain aspect of clean-up mentality, since developers delete source code parts no longer necessary for the current implementation. Feature *linesType* relates *linesAdded* to *linesModified*, and consequently shows whether more lines were added than changed. Feature *largeChanges* describes those changes that are double of

the *LOC* of the average change size and feature *smallChanges* describes those changes that are half of the *LOC* of the average change size. This two features are interesting indicators for us that can give us deeper insight in the evolution of a software. Several studies discovered that small modules are more defect-prone than large ones [24, 38].

**Team.**
The number of authors and author switches influences several facets of software development. Our expectations are that the more authors work on a software and the more author switches occur, the higher the probability of errors and required bug fixes. Feature *authorCount* relates the number of authors to the number of overall changes. Feature *authorSwitches* relates the number of author switches to the number of authors. Two more specialized features *bugfixAuthorCount* and *bugfixAuthorSwitches* help to analyze the influence of team aspects regarding bug fixes. Feature *peopleSwitches* relates the number of people assigned to an certain issue and the authors contributing to the implementation of that issue.

**Process Orientation.**
This category comprehends features that describe how disciplined people follow software development processes and how this affects their development work. Developers have to include the issue number in the commit message, if they commit source code changes to the versioning system. The feature *issueChanges* relates issue modifications to the number of changes. Since developers are requested to provide some rationale in the commit message, we use *withNoMessage*—measuring changes without any commit message—for our prediction. Additionally, feature *commitMessage* relates the number of commits without any commit message to the number of issue modifications. The feature *issueAttachements* relates the number of attachements to the number of issues.
The distribution between different priorities of issues in each software project should be balanced. A high number of issues with high priority may indicate problems that affect quality and re-work amount. We investigate *blockerIssues*, *criticalIssues*, *majorIssues*, and *resolvedIssues* in relation to the total number of issues. To estimate the work habits of the developers we inspect the number of *addingChanges*, *modifyingChanges*, and *deletingChanges* per file. The *changesType* analyzes if there were more adding changes or modifying changes. This information supports the defect prediction of files.

**Complexity of existing Solution.**
In accordance with the laws of software evolution [33], software constantly becomes more complex in the course of time. Changes are harder to apply, since software comprehension is increasingly complicated. Consequently, we use feature *relNumberChanges* to relate the number of changes in the inspection period to the number of changes during the entire evolution of each file. The *changeActivityRate* relates the number of changes during the whole lifetime of a file to the months of the lifetime. Furthermore, the *changeFrequencyBefore* investigates the number of changes before the inspection period relative to the months the file existed before. The *linesActivityRate* specifies the number of lines of code in relation to the age of the file in months. Moreover, the *linesActivityRateTotal* relates the total lines of code to the age of the file. The average lines of code for changes are expressed by *linesAddPerChange*, *linesModifiedPerChange*, and *linesDeletedPerChange*.
We use *fixrateBefore*—which relates the number of bug fixes before the inspection period to the corresponding number of overall changes—to estimate the quality of the existing solution. In this context, we expect a causal relationship between bug fixes that occurred in the past and software quality in the future. The *bugfixChanges* relates the number of bug fix changes to the number of general changes. Lines of code measures *bugfixLinesAdded*, *bugfixLinesModified*, and *bugfixLinesDeleted* are related to LOC added, modified, and deleted. In this context, we assume that bug fixes do not introduce much new code, in that new requirements basically need most code additions. The average lines of code for bug fixes are expressed by *linesAddPerBugfix*, *linesModifiedPerBugfix*, and *linesDeletedPerBugfix*.

**Difficulty of Problem.**
New requirements within object-oriented software systems require the addition of new classes. Therefore, *fileNew*, which contains the information whether a file was newly introduced, is used as input to our prediction. We use *coChangedNew* to show how often a certain file was changed together with the introduction of other new files. Identification of co-changed files is accurately explained in [22].

**Relational Aspects.**
The interrelationship between classes in object-oriented software systems is a basic input for our prediction. We use *couplings* to relate the number of

couplings to the number of overall changes. Thereby, it is possible to judge the interconnection between files. The feature *coChangedFiles* relates the number of co-changed files to the number of entire changes. The features *largeTransactions* and *smallTransactions* calculate the LOC within the considered transaction in relation to the average change size of a transaction.
Besides, quantifying co-changed couplings with size measures for single files, we similarly generate features based on commit transactions. Therefore, we use *TLinesAdded*, *TLinesModified*, and *TLinesDeleted* in relation to lines of code added, modified, and deleted. The *TLinesType* analyzes whether the transactions contained more lines added or lines modified. The *TChangesType* is a coarser grained feature that analyzes whether this file was involved in transactions with more adding revisions or more modifying revisions.
The features *TLinesAddedPerChange* and *TLinesChangedPerChange* relate the number of LOC added or changed per transaction to the number of overall changes. Additionally, *TLinesAddedPerBugfix* and *TLinesChanged-PerBugfix* relate the number of LOC added or changed per transaction to the number of bug fixes. Further, we use *TBugfixLinesAdded*, *TBugfixLinesModified*, and *TBugfixLinesDeleted* in relation to the *linesAdded*, *linesModified*, and *linesDeleted*.

**Time Constraints.**
We are of the opinion that time constraints provide valuable input to our predictions, since the demand for maintenance activities and new features increases in the course of time. The *avgDaysBetweenChanges* represents the average number of days between successive revisions and the *avgDaysPerLine* expresses the number of days per line of code added or changed.
Several events in software projects caused by peaks and outliers have shown interesting results [22]. The *lastChangeMonth* describes the month a file was lastly edited. The features *peakmonth* and *relativePeakMonth* calculate the location of the peak month—the month a file experienced most revisions within the prediction period—with respect to the number of months the inspection period lasts. The number of changes occurring during the peak month related to the number of overall changes is covered by the feature *peakMonthChanges*. The *bugfixVersusChangePeakMonth* feature relates the bug fix peak month to the peak month, based on overall changes.

**Testing.**
Testing metrics allow to estimate the remaining number of bugs. The feature

*bugfixesDiscoveredByDeveloper* measures the number of bug fixes found by developers themselves and provides insight in the quality attentiveness of the software team.

### 3.5.2 Refactoring Features

**Size.**
This category groups size measures such as refactoring lines of code: *refactoringLinesAdded*, *refactoringLinesModified*, or *refactoringLinesDeleted* relative to the appropriate adversary *linesAdded*, *linesModified*, and *linesDeleted* of a file. The features *linesAddPerRefactoring*, *linesModifiedPerRefactoring*, and *linesDeletedPerRefactoring* are interesting indicators, which measure the average lines of code for refactorings. Other features of this category relate bug fix lines added, changed and deleted to refactoring lines added, changed and deleted to compare the effort in both areas of software maintenance. *refactoringLinesType* defines, if there were more *refactoringLinesAdded* or *refactoringLinesModified*. We investigate the number of *addingChanges*, *modifyingChanges*, and *deletingChanges* related to refactoring per file to estimate the work habits of the refactoring authors. Additionally, the feature *refactoringChangesType* decides, if there were more adding or modifying refactoring changes. This information helps to understand the influence of refactoring activities on software defects. Furthermore, we regard *refactoringLargeChanges* as double of the *LOC* of the average refactoring change size and *refactoringSmallChanges* as half of the average refactoring *LOC*. We expect that these features are useful for datamining with respect to refactoring, too.

**Team.**
We estimate that the more refactoring authors are working on the various refactoring changes the higher is the possibility that these changes produce errors. We define two features *refactoringAuthorCount*—relates refactoring authors to the number of refactorings—and *refactoringAuthorSwitches*, which analyzes the work rotation within the group of refactoring authors. The feature *refactoringVSrevisionAuthorSwitches* investigates, whether the refactoring author and the author, who made the last changes before that refactoring, are the same.

**Complexity of existing Solution.**
As described in [33], software continuously becomes more complex. Refactor-

ings are more difficult to add as the software is more difficult to understand and existing software parts have to support the same functionality on the same level of quality. This applies especially to refactorings concerning the inheritance hierarchy. To give consideration to that fact we inspect the *numberRefactorings* in the inspection period relative to the number of refactorings during the whole life cycle of the file. The Feature *refactoringsBefore* relates the number of refactorings before the inspection period to the total number of changes before the inspection period and *refactoringFrequencyBefore* describes the number of refactorings occurred before the inspection period related to the months the file already existed before. Both features give insight in the development behavior, concerning refactoring. We expect that a high refactoring rate improves software quality. The *refactoringActivityRate* is defined as the number of refactorings in relation to the inspection period in months and *refactoringLinesActivityRate* describes the number of lines of code added and changed relative to the inspection period in months. Additionally, *linesActivityRateRefactorings* describes the number of refactoring lines relative to the age of the investigated file in months. The feature *refactoringChanges* relates the number of refactorings to the number of overall changes in the inspection period and the feature *bugfixRefactorings* relates the number of bug fixes to the number of refactorings. Both features are valuable parameters to estimate the influence of refactoring on software defects in the future.

**Relational Aspects.**
We also use the co-change coupling between files that are involved in refactoring to estimate their interrelationship. A feature of this category is *refactoringCouplings*, which measures the number of refactorings whereat other files have been committed, too. We use the number of co-changed files during refactoring relative to the number of refactorings as feature *refactoringCoChangedFiles*. To differentiate between refactorings with many, or only few co-changed files the features *LargeTransactions* and *SmallTransactions* contribute to the prediction. Refactoring features based on commit transactions are related to size measures the same way as for single files: *refactoringTLinesAdded*, *refactoringTLinesModified*, and *refactoringTLinesDeleted* relative to transaction lines of code added, modified, and deleted. The *refactoringTLinesType* describes if the transactions contained more refactoring lines added or lines modified. *refactoringTChangesType* is a coarser grained feature that describes if this file was part of transactions with more

adding refactorings or more modifying refactorings. The features *refactoringTLinesAdded*, *refactoringTLinesModified*, and *refactoringTLinesDeleted* are related to the number of refactorings. Additionally, the bug fix lines added/changed/deleted and the refactoring lines added/changed/deleted are related on transactional basis to get further input to our prediction.

**Time Constraints.**
We believe that time constraints are essential to estimate refactoring activities. Feature *avgDaysBetweenRefactorings* is defined as the average number of days between refactorings. The number of days per refactoring line of code added or changed is captured as *avgDaysPerRefactoringLine*. Peaks and outliers have been shown to give interesting events in software projects [22], which also applies to refactoring. For feature *refactoringPeakMonth* we measure the location of the month, which contains most refactorings, within the prediction period. Feature *peakMonthRefactorings* measures the number of refactorings happening during the refactoring peak month normalized by the overall number of refactorings. Feature *refactoringVersusChangePeakMonth* relates the month during which most refactorings occurred to the peak month, based on overall changes. Feature *bugfixVersusRefactoringPeakMonth* relates the peak months of bug fixes and refactorings to analyze, whether most refactorings happen before or after most bug fixes.

## 3.6 Classifiers — Data Mining Algorithms

A number of data mining algorithms that make up certain classifiers are used to generate prediction models [50]. These classifiers separate object-oriented entities like classes into particular groups such as classes with or without bug fixes.

- *C4.5* One broadly accepted classification algorithm is C4.5. This algorithm induces decision trees and is based on a series of improvements of another classifier called ID3. This classifier compares one of the input attributes against a threshold value and partitions the input space with axis parallel splits. It includes improvements for dealing with numeric attributes, missing values, and noisy data.

- *LMT* This is a data mining algorithm for building logistic model trees, which are classification trees with logistic regression functions at the

leaves. It uses validation to determine how many iterations to run, when fitting the logistic regression function at a node of the decision tree.

- *Rip* Repeated Incremental Pruning is a propositional rule learner. It generates rules instead of decision trees like the other data mining techniques. However, rules and trees are two very similar methods for representing machine knowledge. It uses a growth phase, where antecedents are greedily added until the rule reaches 100% accuracy. Then in the pruning phase, metrics are used to prune rules until the defined length is reached.

- *NNge* In this case a nearest-neighbor algorithm is used to build rules using non-nested generalized exemplars.

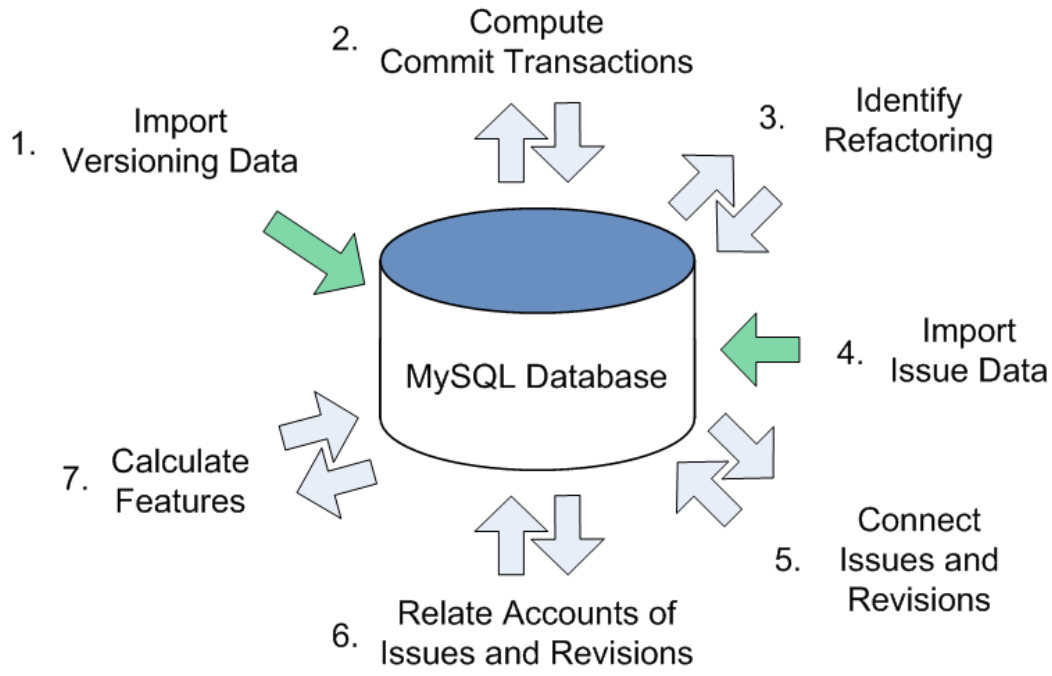


Figure 2: Data Extraction into MySQL Database

# 4 Methodology

## 4.1 Data Extraction into Database

Figure 2 depicts the workflow of data extraction of a given project into the MySQL database:

### 4.1.1 Import Versioning Data and Computation of Commit Transactions

After the versioning data (CVS) is read in from a before generated log file, the CVS commit transactions have to be computed, since this information got lost within the version management system.

### 4.1.2 Identifying Refactoring

The number of refactorings is determined with the help of information contained in the modification reports of the versioning system. Thereby, we iden-

| Project | Modifications | Refacs | Other Changes | FP | FN |
|---|---|---|---|---|---|
| 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

tify modifications to the source code that are refactorings. For generation of refactoring features—many times based on number of refactorings—we do not distinguish different types of refactorings (e.g. extract class/interface/method, move class/field/method, pull up/push down field/method, etc.). These features cover the fact that developers try to improve the quality of code. For all analyzed projects, the commit messages provided by developers as part of the modification reports are the basis to identify refactoring. We start our identification by search for part of a word called "refactor". We analyzed the results and discovered that the code is not a refactoring, when "needs refactoring" is included in the commit message [50]. With several refinements we used for each project 10-20 SQL queries to mark modifications as refactorings. We used a statistical evaluation to estimate the number of refactorings, we identified correctly with our method. Therefore, we took a random sample of 100 modifications for each project and checked, whether it was a refactoring or not. Now we labeled this subset of revisions with our SQL queries and proofed if the labels were correct. Table 1 shows high rates of correct labels for each investigated project. For ArgoUML [52] and Liferay Portal [46] all revisions (instances) labeled as refactoring actually were refactorings. This applies to projects JBoss Cache [25] and XDoclet [15] with only one exception and to the spring framework [21] with two exceptions (false positive, FP). For the Liferay Portal and the spring framework only one refactoring was missed (false negative, FN), for ArgoUML two refactorings were missed, and three refactorings were missed for the remaining projects in each case. Hence, all projects show high identification rates, with a remarkable low false positive rate.

### 4.1.3 Import Issue Data

The next step concerns the import of bugtracking data, which demands a number of processing steps, in that we have no single log file at hand that allows us to easily import that information. That is, why we use the export functionality that is offered by the bugtracking system on the corresponding homepages of each analyzed open source project. JBoss Cache, Liferay Portal, Spring Framework, and XDoclet use JIRA as a bugtracking system. ArgoUML uses an own bugtracking system to cover its issues. We are interested in issues that concern defects (bugs) with resolution fixed. After the selection of that criteria and clicking on the submit button for that query, a list of affected issues is generated. Figures 3–5 exemplarily illustrate that process for projects ArgoUML and Liferay Portal.
All projects offer output files in different formats that represent the current state of the bugtracking system. We decided to choose an excel output file for the issues that matched our query. For each investigated project a lot of project related information per issue is presented in several columns, therefore we have to focus on information important to us. This is done by a further processing step that eliminates unnecessary information. Additionally, we produce two tab-separated files from the data included in each excel file and reorganize the sequence of columns to match our data schema. The bugtracking data included in these files is imported into the corresponding database tables of our MySQL database by appropriate SQL statements.

### 4.1.4 Connect Issues and Revisions

After executing these working steps we connect issues of the bugtracking system—in our case already fixed bugs—and revisions of the versioning system. We analyze the commit message of each revision to find out those revisions that are related to a distinct issue. This is done by pattern matching on the issue number. For project ArgoUML, this process was a bit tricky, since no recurrent sequence of characters is used in combination with a given issue number. So we searched for words like issue, number, bug, or task. For the other projects the approach was straightforward, since a certain keyword notifies that a revision is related to a certain issue: JBoss Cache (JBCACHE), Liferay Portal (LEP), Spring Framework (SPR), and XDoclet (XDT). The result of this step is stored in the appropriate database table.

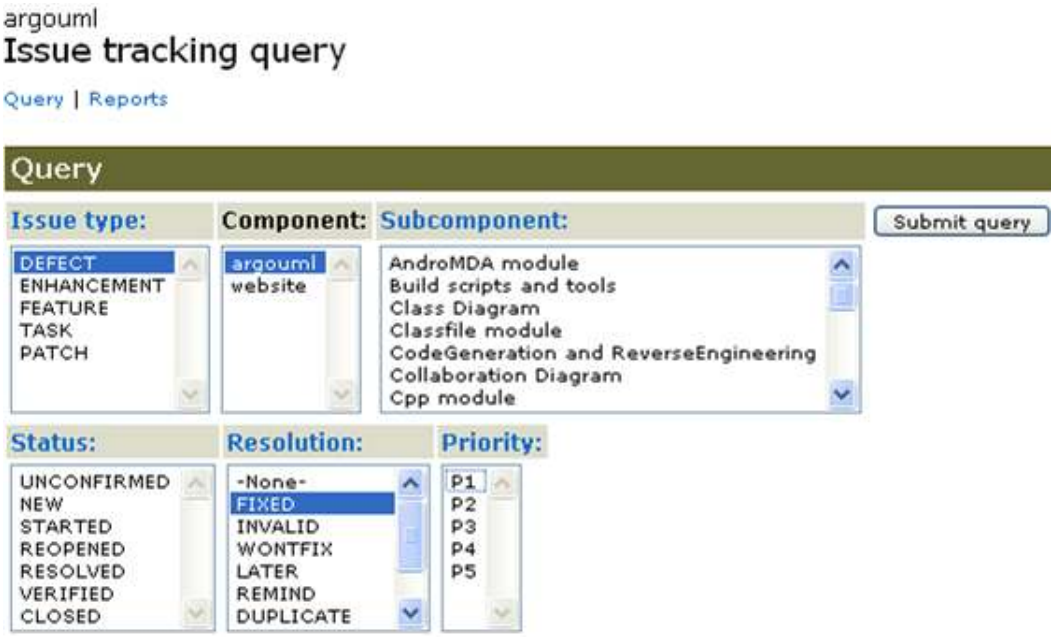


Figure 3: ArgoUML - Issue Query

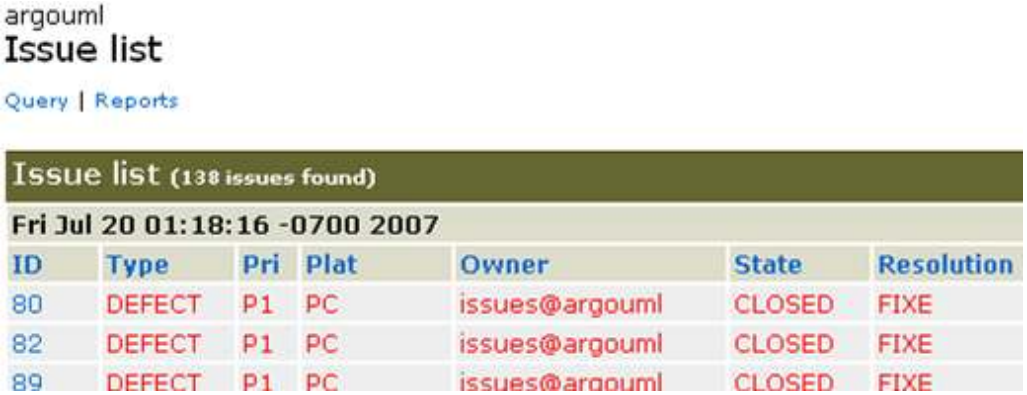


Figure 4: ArgoUML - Issue Result

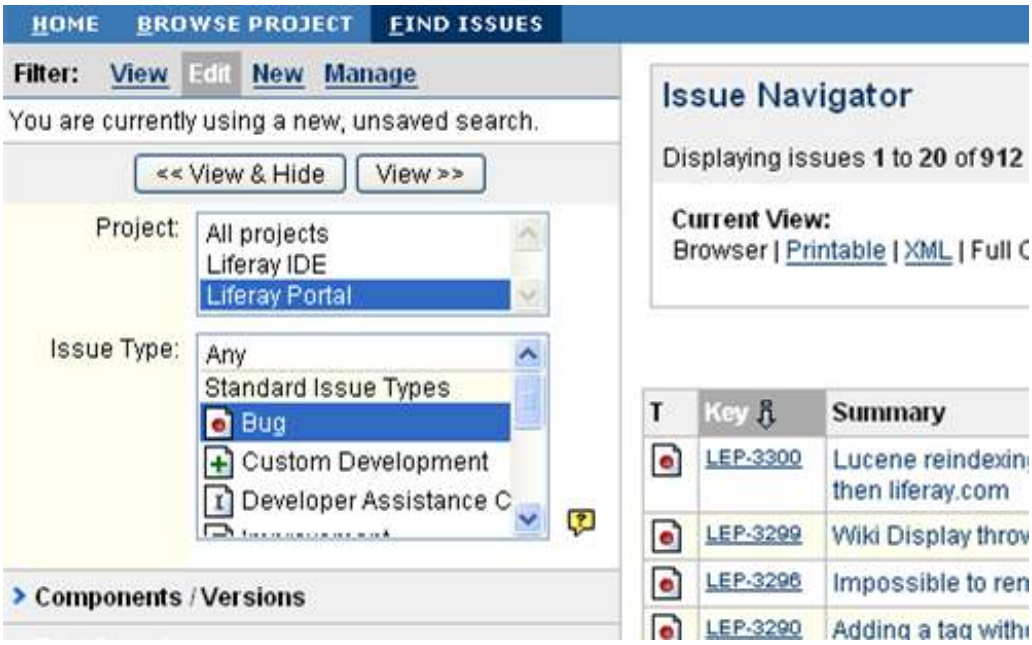


Figure 5: Liferay Portal - Issue Query and Issue Result

### 4.1.5 Relate Accounts of Issues and Versioning

To cover interconnections between the bugtracking and versioning system with respect to author related aspects, we relate accounts of versioning and bugtracking systems. This is useful, because several times authors work on the versioning system, as well as the bugtracking system.

### 4.1.6 Calculate Features

Finally, we generate refactoring and non-refactoring features on the basis of this information to cover several aspects of the software development process. These features are outlined in more detail under section Prediction Foundation. Naturally, the target attribute—targetBugs—is also computed by that working step. There are two possibilities to determine, whether a revision is a bug fix, or not. On the one hand we search for issues related to that revision that are bug fixes, on the other hand we investigate the commit message provided by developers as part of the modification reports. For the analysis of the commit message we start our identification by search for part of a word called "fix". We analyzed the results and discovered that the revision

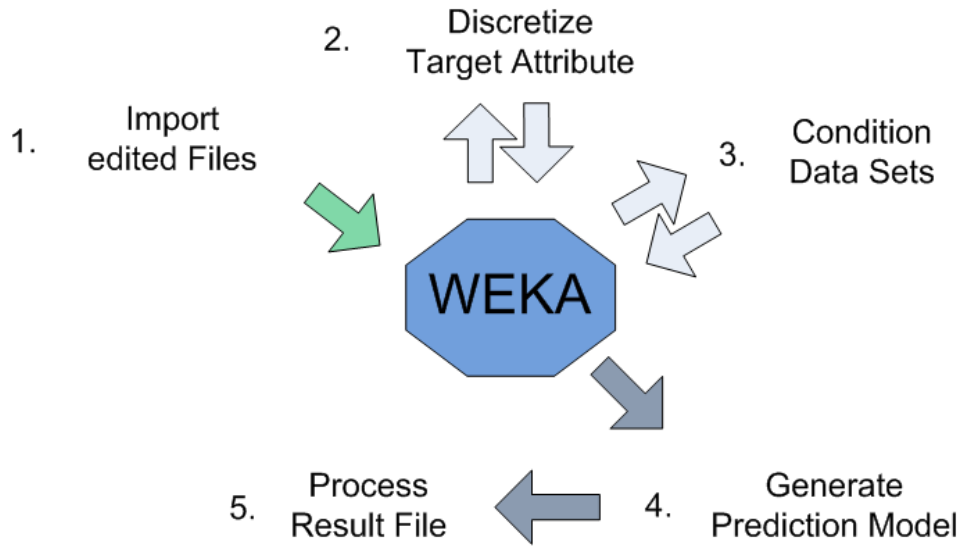


Figure 6: Data Processing with Weka

is not a bug fix, when "will fix" is included in the commit message. With several refinements we used for each project 6-10 queries to automatically detect bug fixes. Now the MySQL database holds the extracted information in only one database table for each project under investigation.

## 4.2 Data Processing with Weka

Weka is a collection of machine learning algorithms for data mining tasks and is used to generate prediction models for provided input data. The algorithms can either be applied directly to a data set or called from your own Java code. Weka contains tools for data pre-processing, classification, regression, clustering, association rules, and visualization. It is also well-suited for developing new machine learning schemes.
Figure 6 depicts the workflow of data processing of a given project with Weka:

### 4.2.1 Data Import into Weka

First of all, Weka connects to our MySQL [11] database and retrieves the necessary instances (files) by using appropriate SQL statements. In our case

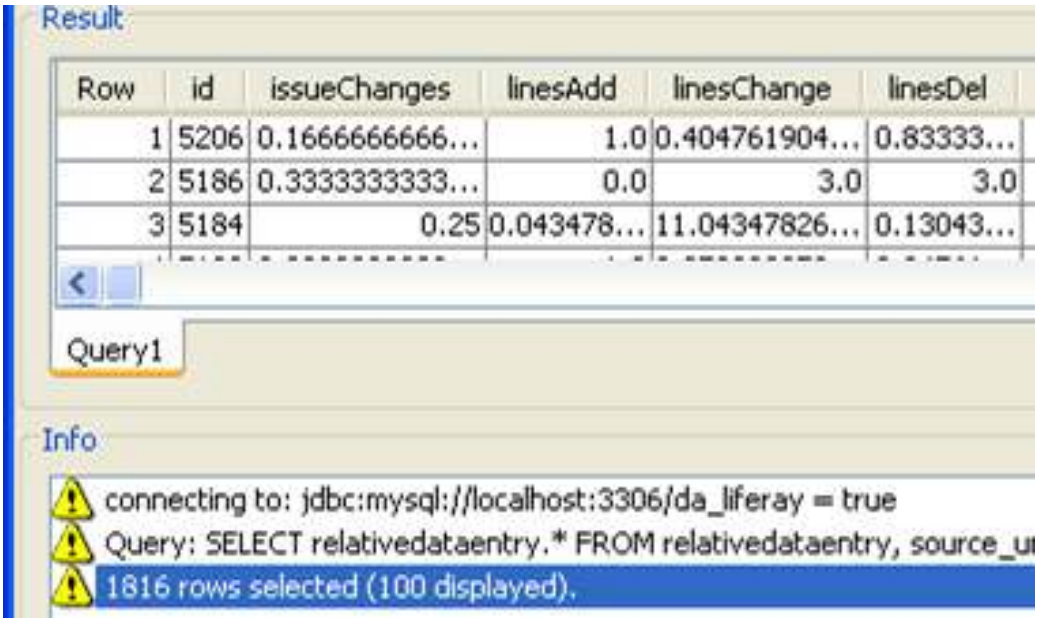


Figure 7: Weka - SQL-Viewer - Import Data From a Database

we are interested in files with the ending ".java"—our focus is on source code files—and with at least one, out of the set of attributes lines added, lines changed, and lines deleted, above zero—that are edited files (Figure 7).

### 4.2.2 Discretize Target Attribute

In the next step the filter "weka.filters.unsupervised.attribute.Discretize" is used to convert our target attribute targetBugs—number of bugs in the target period—from numeric into nominal data. This means that an distinct instance is assigned to a group of instances, called bin that holds values of the target attribute within the same range. We use two bins, one bin contains files with no bugs and the other bin contains files with one or even more bugs in the target period (Figure 8 and Figure 9). The bin "No bugs" corresponds to '(-inf-0.5]' and the bin "One or more bugs" corresponds to '(0.5-inf)'.

### 4.2.3 Conditioning Data Sets

Since, the number of instances in both bins is different, independent from the investigated project and analyzed time frame (Figure 10 and Figure 11)

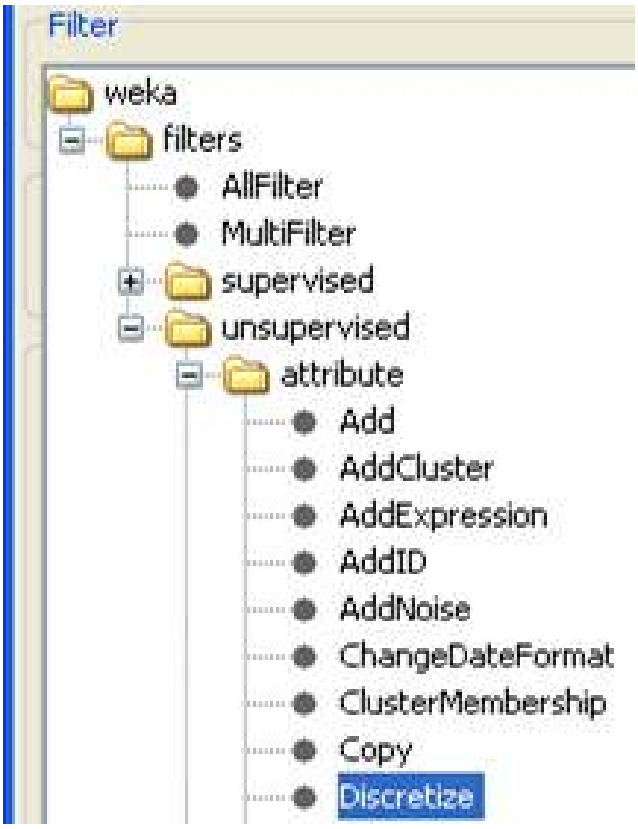


Figure 8: Weka - Filter - Choose the appropriate Filter

a further step of pre-processing is required. This affects mostly the bin for buggy files, since, with only one exception—JBoss Cache October 2005—files with no bug fixes in the target period form the majority of instances. The bin, holding more instances is subdivided into sections that consist of the same number of instances as in the smaller bin. Each time a different set of files—a different section—of the total data set of the larger bin is used together with the total number of files of the smaller bin. An example helps to clarify our approach:
Project Liferay Portal with time frame June 2005 – April 2006 is made up of 1816 classes that have been changed during the feature period. 1338 classes show no bug fixes and 478 classes show one or more bug fixes, in the corresponding target period. The data at hand is processed to form three different data sets, each holding 956 instances, whereby classes with bug fixes

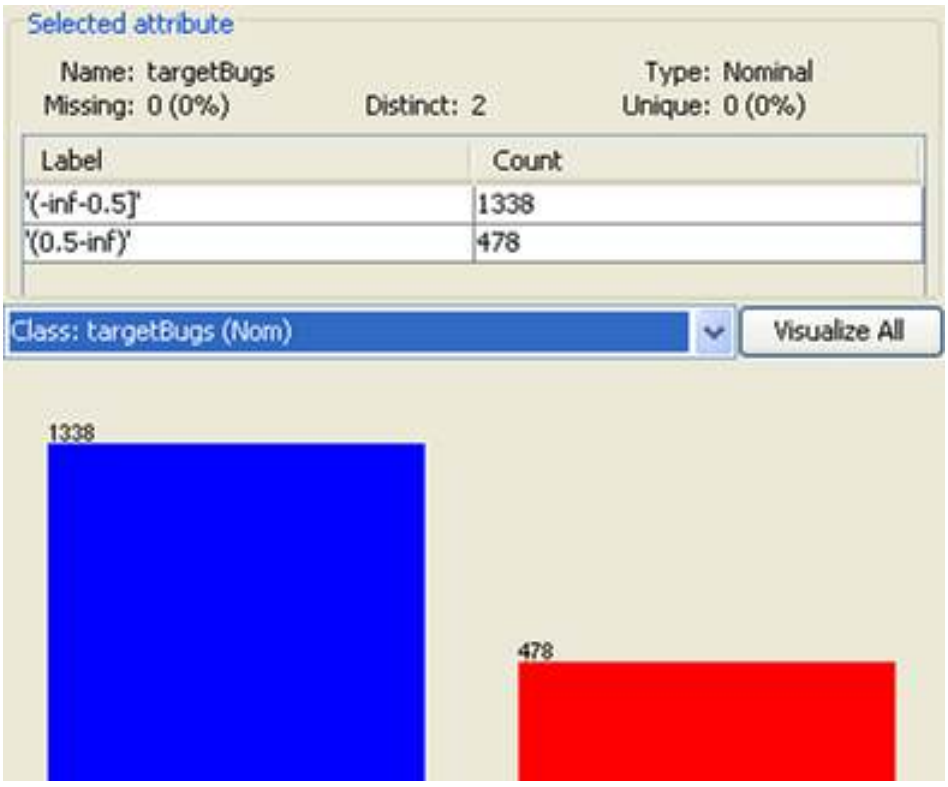


Figure 9: Weka - Imported Data with Nominal Target Attribute "targetBugs"

always remain the same. The first data set uses the topmost 478 classes of bin "No bugs", the second data set uses the next 478 classes, and the third data set uses the bottommost 478 classes. As one can see, 96 classes are used two times, for data set two and data set three. This approach has to be used, since the number of classes contained in the smaller bin is no integral multiple of the number of classes contained in the larger bin. Figure 12 shows the accordant result after conditioning for project Liferay Portal June 2005. Sometimes the difference in size between bin "No bugs" and bin "One or more bugs" exceeds the factor ten, which accordingly leads to more than ten prediction models for that case to use all available information. The set of instances in each processed data set is naturally smaller, than the total set of instances, but the prediction qualitity of models based on that data rises. Hence, a balanced data set that holds instances to the same extent of both

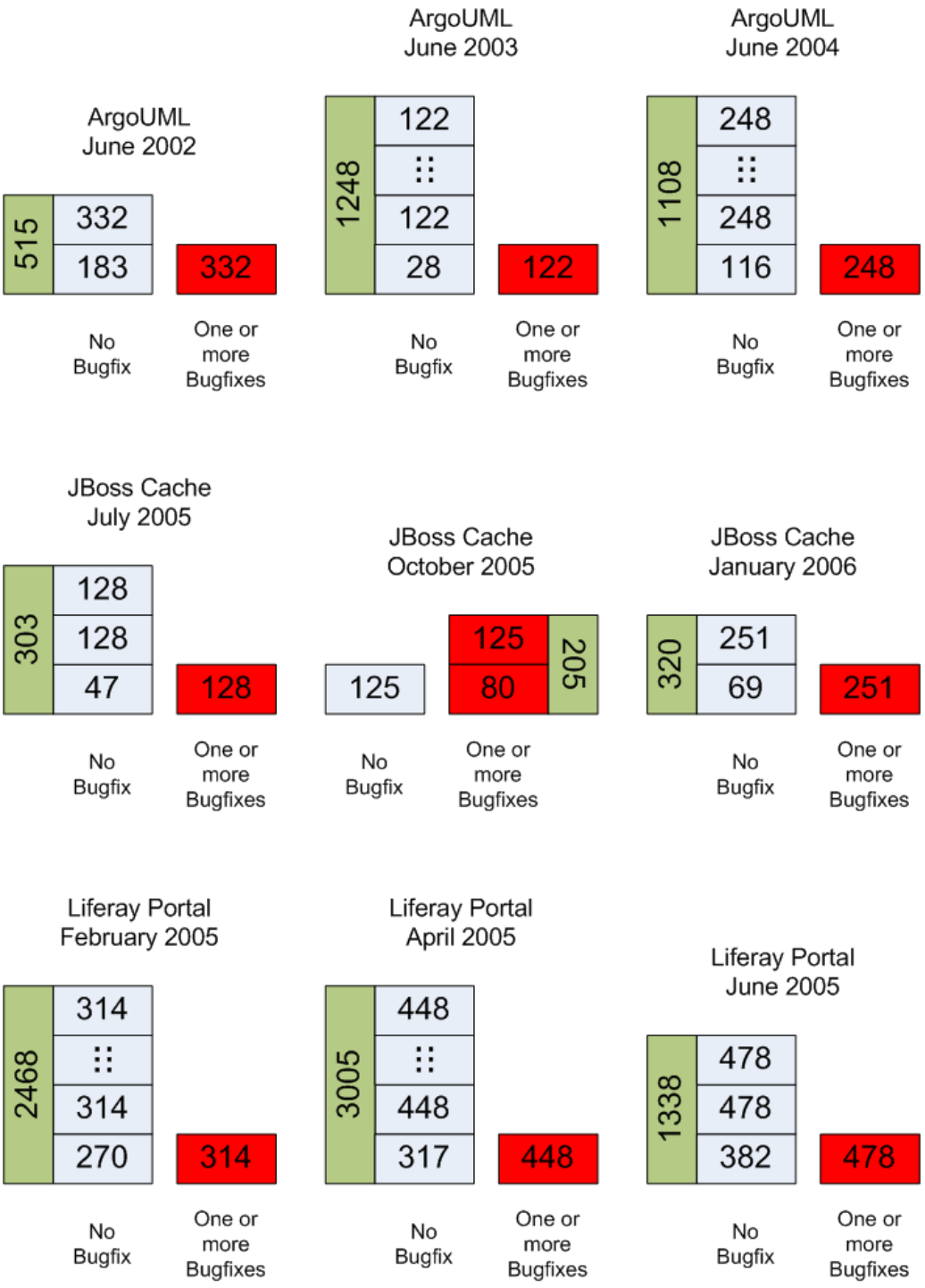


Figure 10: Distribution of Classes with or without Bug fixes for Projects ArgoUML, JBoss Cache, and Liferay Portal

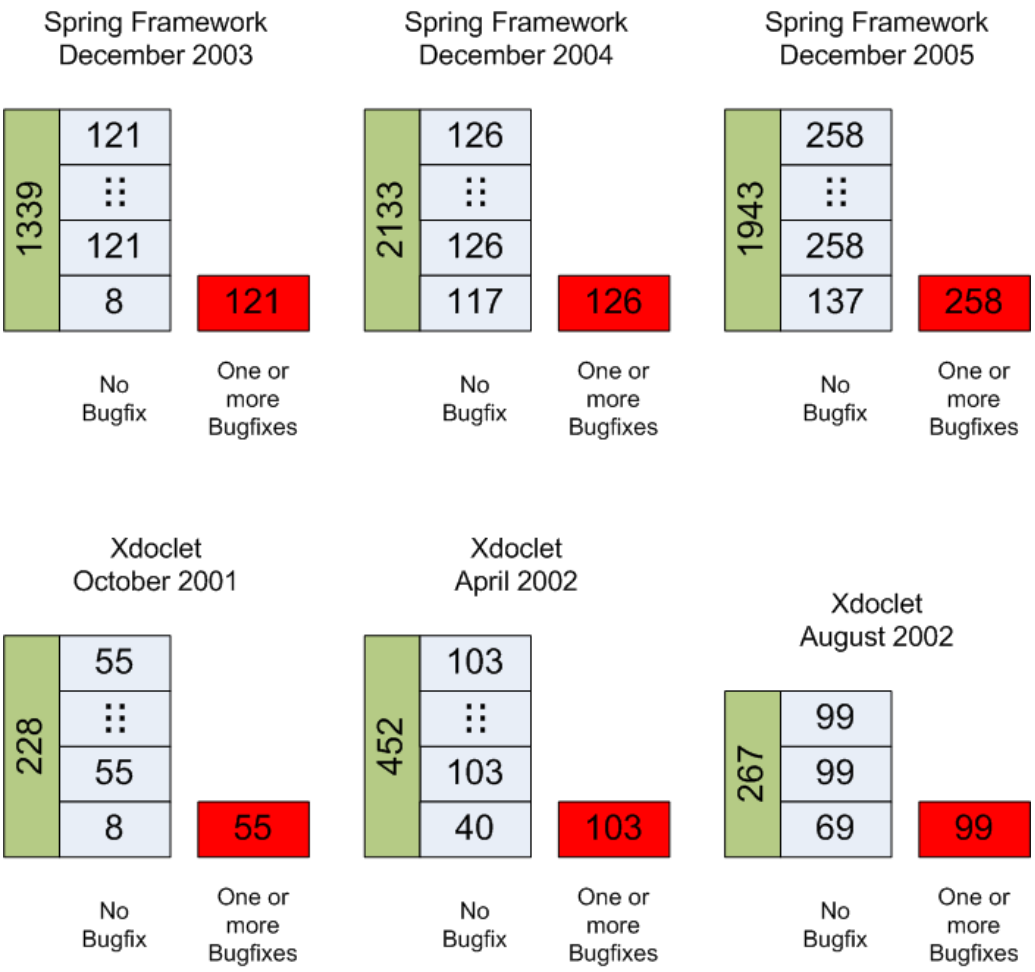


Figure 11: Distribution of Classes with or without Bug fixes for Projects Spring Framework and XDoclet

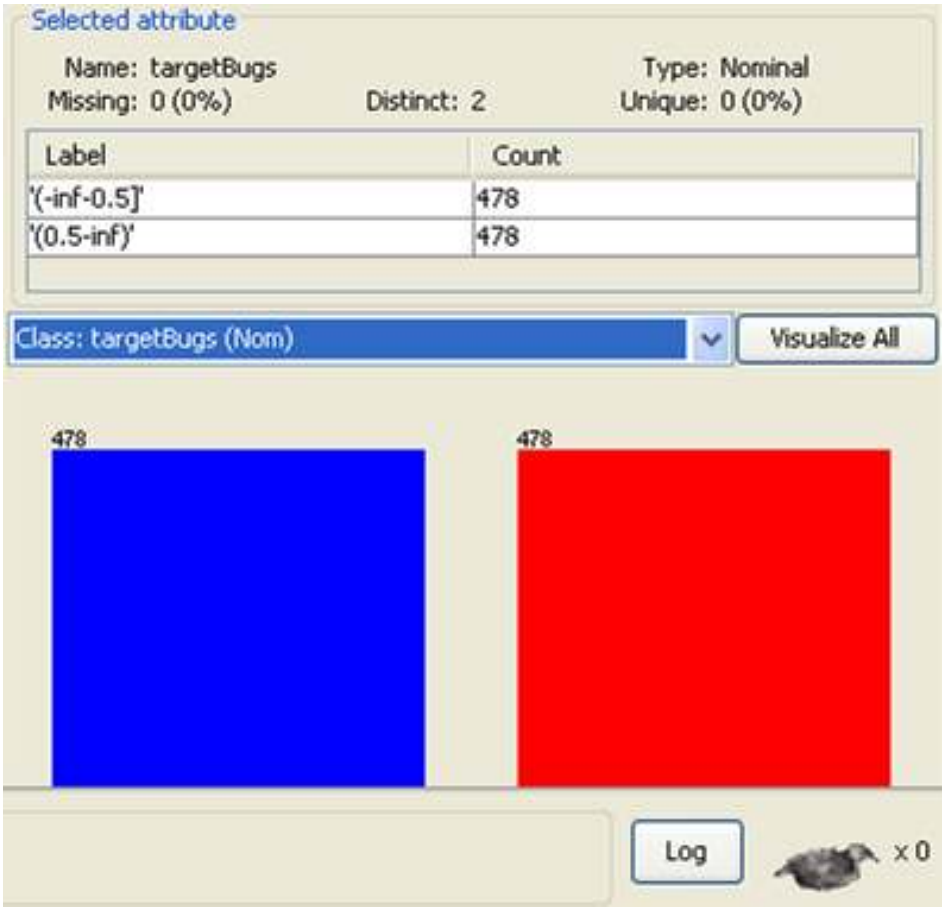


Figure 12: Weka - The same Number of Instances for Bin "No bugs" and Bin "One or more bugs"

categories, supports the prediction.
To return to our example, now three different models can be generated based on those three data sets. A model made up of the same number of non-error-prone classes (bin "Bug fixes=0") and error-prone classes (bin "Bug fixes>=1") is called *section model*.

### 4.2.4 Prediction Model Generation and Result File Processing

After appliance of certain classifiers on the data a model is generated and the output that provides statistical information about the quality of the

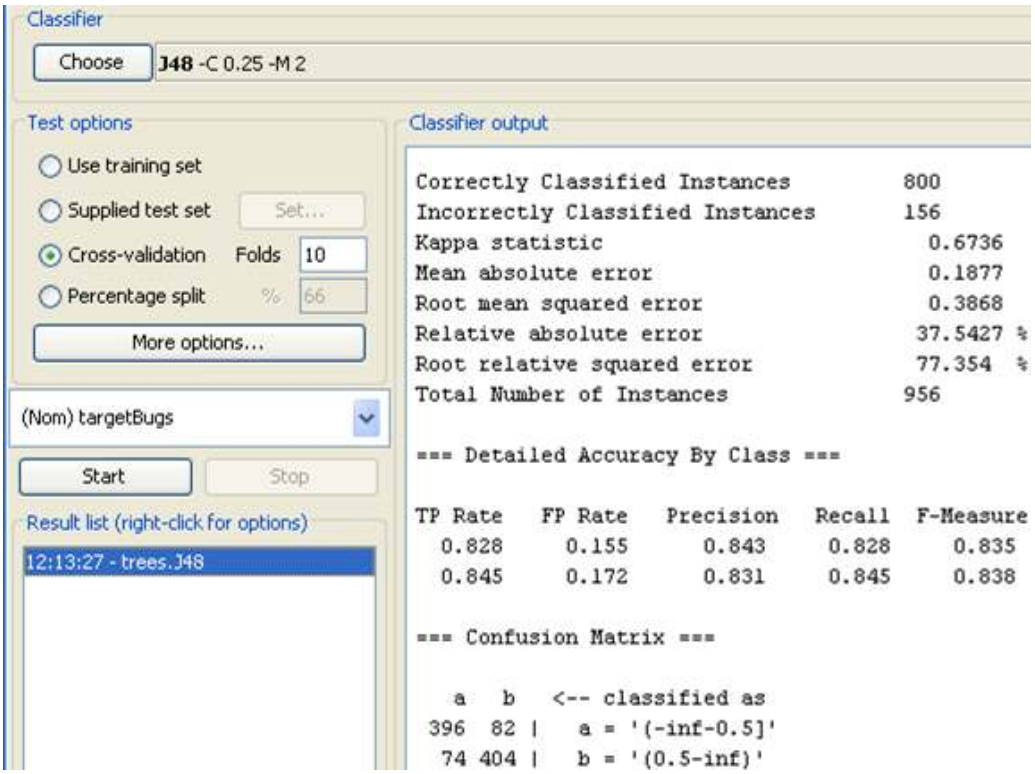


Figure 13: Weka - Classify - Output for Classifier J48

model, is stored in a result file. In the case of tree-generating classifiers—e.g. J48, which uses classification algorithm C4.5—, the result file additionally includes the appropriate model tree. Figure 13 shows the necessary settings for classifier J48 to gain the corresponding result file for project Liferay Portal June 2005 – section model 1, using the total set of features.
Now, let us focus on the assembly of the model tree for project Liferay Portal June 2005 – section model 1, using refactoring features only (Figure 14). Exemplarily, we show the upper part of the respective model tree. Generally, each model tree consists of a certain sequence of rules, whereby the number of rules correlates to the size of the tree. Next to each rule the number of instances, which follows that rule, is listed in brackets. If a certain number of instances has been misclassified that number is given after a slash.
Most of the time, the tree models are very nested, since they try to predict the target attribute as good as possible. That is the reason, why many rules

only affect a small subset of all investigated instances. Hence, we decided only to consider rules, which apply to more than 5% of the total quantity of instances. By conditioning the result file, rules concerning less instances are cut out of the model trees. Figure 15 shows the situation after conditioning for project Liferay Portal June 2005.
Based on certain threshold values for each decision rule the model trees decide, whether the instances are assigned to the group of instances (classes) with no bugs (bin "No bugs"), or with one or more bugs (bin "One or more bugs"). For an appointed feature (attribute) a threshold value of 10% indicates that e.g. 1 refactoring is related to 10 changes, or 2 refactorings are related to 20 changes, and so on—in this case for feature *refactoringChanges*. Sometimes an investigated attribute is ranked high in the tree hierarchy—for example tree level 1 to 3—and sometimes it is applied after a series of other attributes have been applied—for example tree level 8.

## 4.3 Data Analysis

Figure 16 shows the fundamental composition of our analysis. Each of the four displayed blocs covers different approaches to analyze certain hypotheses. Thereby, the three upper blocs are based on the quality of prediction. This means that hypotheses related to these topics are founded on two hypotheses that address the quality of the generated prediction models.

### 4.3.1 Quality of Prediction

The prediction models are generated based on refactoring features (47), non-refactoring features (62), and the total set of features (109) and different classifiers, respectively classification algorithms, namely C4.5, LMT, JRip, and NNge. We summarize the results for each investigated project in the respective tables. The prediction quality is evaluated using three relevant measures, namely Precision (Pre.), Recall (Rec.), and F-measure (F-m).

### 4.3.2 Association of Refactoring and other Software Activities

Each *section model* consists of an often large number of decision rules and a certain feature is possibly used several times for different decisions. It is necessary to state that a decision rule consists of at least one and at most of two decisions. A decision rule with two decisions splits up the data set in two

```
Decision tree C4.5
------------------

Liferay Portal
 jun05_section1
 --------------

=== Run information ===

Scheme:       weka.classifiers.trees.J48 -C 0.25 -M 2
Instances:    956
Attributes:   48
Test mode:    10-fold cross-validation

=== Classifier model (full training set) ===

relRefactorings <= 0
| linesActivityRateRefactorings <= 0.03125: '(-inf-0.5]' (63.0/2.0)
| linesActivityRateRefactorings > 0.03125
| | bugfixVersusRefactoringLinesDel <= 4
| | | refactoringChanges <= 0.333333
| | | | refactoringLinesChange <= 0.125
| | | | | linesActivityRateRefactorings <= 0.2
| | | | | | bugfixRefactorings <= 0: '(-inf-0.5]' (7.0)
| | | | | | bugfixRefactorings > 0
| | | | | | | refactoringFrequencyBefore <= 0.045455: '(-inf-0.5]' (9.0/1.0)
| | | | | | | refactoringFrequencyBefore > 0.045455: '(0.5-inf)' (5.0)
| | | | | linesActivityRateRefactorings > 0.2
| | | | | | linesActivityRateRefactorings <= 0.4: '(0.5-inf)' (55.0/20.0)
| | | | | | linesActivityRateRefactorings > 0.4: '(-inf-0.5]' (6.0/1.0)
| | | | refactoringLinesChange > 0.125
| | | | | refactoringVersusChangePeakMonth <= 0.166667: '(0.5-inf)' (4.0/1.0)
| | | | | refactoringVersusChangePeakMonth > 0.166667
| | | | | | TBugfixVersusTRefactoringLinesDel <= 106: '(-inf-0.5]' (51.0/4.0)
| | | | | | TBugfixVersusTRefactoringLinesDel > 106
| | | | | | | refactoringTLinesAdd <= 0.000382
| | | | | | | | refactoringTLinesDel <= 0.001862: '(-inf-0.5]' (4.0)
| | | | | | | | refactoringTLinesDel > 0.001862: '(0.5-inf)' (2.0)
| | | | | | | refactoringTLinesAdd > 0.000382: '(0.5-inf)' (2.0)
| | | refactoringChanges > 0.333333
| | | | ...

Number of Leaves  :     55
Size of the tree :      109
```

Figure 14: Model Tree based on Classification Algorithm C4.5 for Project Liferay Portal - June 2005 - Section Model 1 — Before conditioning

```
Decision tree C4.5
------------------

Liferay Portal
 jun05_section1
 --------------

relRefactorings <= 0
| linesActivityRateRefactorings <= 0.03125: '(-inf-0.5]' (63.0/2.0)
| linesActivityRateRefactorings > 0.03125
| | bugfixVersusRefactoringLinesDel <= 4
| | | refactoringChanges <= 0.333333
| | | | refactoringLinesChange > 0.125
| | | | | refactoringVersusChangePeakMonth > 0.166667
| | | | | | TBugfixVersusTRefactoringLinesDel <= 106: '(-inf-0.5]' (51.0/4.0)
| | | refactoringChanges > 0.333333
| | | | ...
```

Figure 15: Model Tree based on Classification Algorithm C4.5 for Project Liferay Portal - June 2005 - Section Model 1 — After conditioning

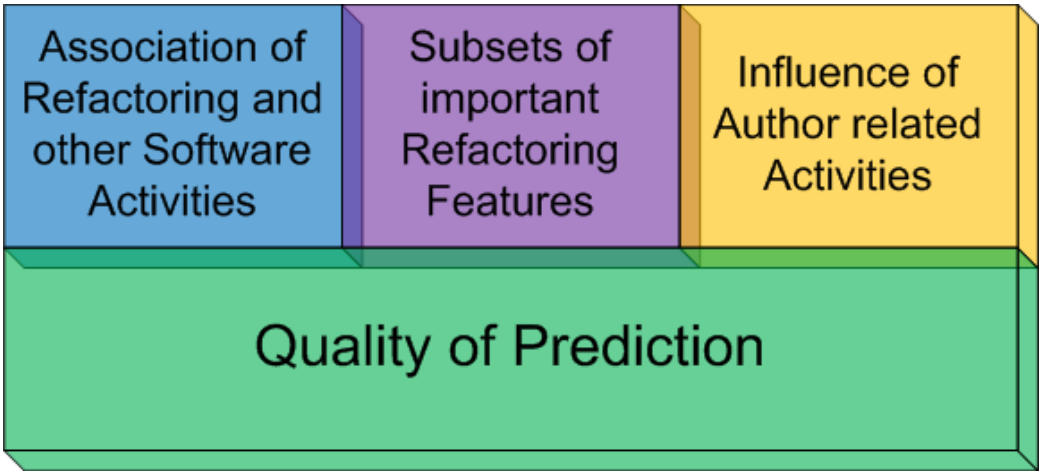


Figure 16: Data Analysis

subsets mostly not of the same size. A decision rule with only one decision also further restricts the number of instances, but does not offer an option for that chain of rules at hand—with the actual feature as last element of the chain. A chain of decision rules that assigns a set of instances to a distinct bin ("No bugs", or "One or more bugs") is called *model sequence*.
We analyze refactoring features, which relate refactoring properties to other software properties, or more generic to overall change properties. Thereby, we investigate the number of occurrences of refactoring features of interest in decision rules of the generated *section models*. Additionally, we proof, whether classes that are assigned to bin "No bugs" or bin "One or more bugs" hold values for the considered feature equal or below, or above a calculated threshold value. Based on these results we discover coherences between refactoring and other software characteristics that affect software defect appearance.

### 4.3.3 Subsets of important Refactoring Features

We analyze the frequency of occurrence of refactoring features in decision rules of the generated models to find out those, most significant for predicting future bug fixes. Decision rules of features that occur on the first three tree levels are rated higher, since their influence on the prediction is stronger. Therefore, we evaluated the appearance of refactoring features within decision rules on the first three tree levels. Those features most important to the prediction models of each singular project, or to the prediction models of several different projects, are discovered.

### 4.3.4 Influence of Author related Activities

We analyze the number of decisions in *section models* based on author related features only. Tables summarize for each author related feature the number of decisions that use threshold values within a certain range. Again, classes are attached to the bin "No bugs", or the bin "One or more bugs", if the respective feature holds a value equal or below, or above a certain threshold value. Some of the presented tables include fields with no actual data. An table entry "— (—)" means that there are no decision rules that fulfill that kind of relational interrelationship.

# 5 Evaluation

## 5.1 Evaluation of Prediction Models

In our analysis of prediction models for bug fixes we use *precision*, *recall*, and $f-measure$—three essential markers characterizing model performance. These evaluation measures are defined based on formulas regarding different rates such as true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). True positives describe the predictions that are correctly classified. False positives are the ones that are classified to be in a particular group (e.g. number bug fixes = 0), but the classification is wrong. The number of elements that are correctly classified not to belong to the given group form the true negatives. False negatives are elements that belong to the group of interest, but are erroneously classified to be outside of the group [50]. (see Table 2)

- *Precision* describes the percentage of correctly classified entities.

$$precision \;=\; \frac{TP}{TP \;+\; FP} \cdot 100\% \;=\; \frac{predicted\ correct}{total\ predicted}$$

  The higher the precision the more predictions are correct.

- *Recall* describes the percentage of entities classified from the group of positive entities.

$$recall \;=\; \frac{TP}{TP \;+\; FN} \cdot 100\% \;=\; \frac{predicted\ correct}{total\ positive}$$

  The higher the recall the more elements can be found.

- *F-measure* is a dimensionless measure combining precision and recall through the given formula.

$$F-measure \;=\; \frac{2 \;\cdot\; TP}{2 \;\cdot\; TP \;+\; FP \;+\; FN} \;=\; \cdots$$

| | | Predicted | |
|---|---|---|---|
| | | yes | no |
| Actual | yes | true positive | false negative |
| | no | false positive | true negative |

Table 2: Outcome of prediction of two groups

$$\cdots = \frac{2 \times recall \times precision}{recall + precision}$$

We use the F-measure to compare the performance of our prediction models. However, a high value for a certain group (e.g. classes with bug fixes) is not a strict indicator for a good model. Instead, the measures of all possible groups have to be regarded.

What happens if the prediction model mismatches the groups? Do false positives outweigh false negatives? We are of the opinion that false negatives are worse, because essential bug fixes are not predicted correctly, whereas false positives anticipate bug fixes, which will never take place.

## 5.2 Open Source Projects

In line with the thesis the following list of open source projects is investigated:

- *ArgoUML* is an UML modeling tool, which includes support for all standard UML 1.4 diagrams and runs on any Java platform.

  Project Start: January 1998

  Project Size[1] (classes): 1555

  Team Members: > 32

- *JBoss Cache* allows to cache frequently accessed Java objects in order to improve the performance of applications. This is done by elimination of unnecessary database access, which decreases network traffic and increases scalability.

  Project Start: November 2003

  Project Size (classes): 222

  Team Members: > 59

[1] The project size of all projects is determined on the basis of the project status at the beginning of December 2006.

- *Liferay Portal* is a framework, which offers integrated Web publishing and content management, service-oriented architecture, and compatibility with many IT infrastructures.

  Project Start: March 2002

  Project Size (classes): 3124

  Team Members: 76

- *Spring Framework* is a layered application framework, which eases the use of J2EE, supports the use of interfaces and object-oriented design, and enhances testability.

  Project Start: February 2003

  Project Size (classes): 4492

  Team Members: 35

- *XDoclet* is a code generation engine, which enables attribute-oriented programming for java by adding meta data (JavaDoc tags) to the source code. These information is used to generate artifacts such as XML descriptors or source code.

  Project Start: July 2001

  Project Size (classes): 690

  Team Members: 33

## 5.3 Results

We analyze three different time frames for each project under investigation to underline our findings (Table 3). Every time frame consists of a feature period and a target period and extends over one year. For example the first listed time frame of project ArgoUML uses a feature period from june 2002 till november 2002 and a target period from december 2002 till may 2003. Additionally, the number of classes for each time frame is given. Our example

| Project | Feature Period | Target Period | Classes |
|---|---|---|---|
| ArgoUML | Jun.02 - Nov.02 | Dec.02 - May.03 | 847 |
| | Jun.03 - Nov.03 | Dec.03 - May.04 | 1370 |
| | Jun.04 - Nov.04 | Dec.04 - May.05 | 1356 |
| Cache | Jul.05 - Dec.05 | Jan.06 - Jun.06 | 431 |
| | Oct.05 - Mar.06 | Apr.06 - Sep.06 | 330 |
| | Jan.06 - Jun.06 | Jul.06 - Dec.06 | 571 |
| Liferay | Feb.05 - Jul.05 | Aug.05 - Jan.06 | 2782 |
| | Apr.05 - Sep.05 | Oct.05 - Mar.06 | 3453 |
| | Jun.05 - Nov.05 | Dec.05 - May.06 | 1816 |
| Spring | Dec.03 - May.04 | Jun.04 - Nov.04 | 1460 |
| | Dec.04 - May.05 | Jun.05 - Nov.05 | 2259 |
| | Dec.05 - May.06 | Jun.06 - Nov.06 | 2201 |
| XDoclet | Oct.01 - Mar.02 | Apr.02 - Sep.02 | 283 |
| | Apr.02 - Sep.02 | Oct.02 - Mar.03 | 555 |
| | Aug.02 - Jan.03 | Feb.03 - Jul.03 | 366 |

Table 3: Total Number of Classes for analyzed Periods by Project

time frame of project ArgoUML affects 847 classes that have been altered in the feature period. This set of classes shows bug fixes, or not in the target period, whereby classes with no bug fixes outweigh classes with one or more bug fixes, most of the time. There is only one exception, namely time frame JBoss Cache with feature period starting from october 2005.
Table 4 is built-on very similar to Table 3, but however there are some important differences. The open source projects together with the appropriate time frames remain the same. This time the number of *section models* for each analyzed time frame, together with the number of classes that are used to generate each *section model* of that category, is shown. Let us take a look at the same example as before—ArgoUML with feature period starting from june 2002. In this case we can generate two *section models* out of the total set of classes. Thereby, each *section model* is based on a subset of classes (664) from the total set of classes (847). Each subset consists of the same number of classes with and without bug fixes. In our example 332 classes without bug fixes and 332 classes with bug fixes in the target period are the instances our prediction models are based on.

| Project | Feature Period | Target Period | Number of Section Models | Classes per Section Model |
|---|---|---|---|---|
| ArgoUML | Jun.02 - Nov.02 | Dec.02 - May.03 | 2 | 664 |
| | Jun.03 - Nov.03 | Dec.03 - May.04 | 10 | 244 |
| | Jun.04 - Nov.04 | Dec.04 - May.05 | 5 | 496 |
| Cache | Jul.05 - Dec.05 | Jan.06 - Jun.06 | 3 | 256 |
| | Oct.05 - Mar.06 | Apr.06 - Sep.06 | 2 | 250 |
| | Jan.06 - Jun.06 | Jul.06 - Dec.06 | 2 | 502 |
| Liferay | Feb.05 - Jul.05 | Aug.05 - Jan.06 | 8 | 628 |
| | Apr.05 - Sep.05 | Oct.05 - Mar.06 | 7 | 896 |
| | Jun.05 - Nov.05 | Dec.05 - May.06 | 3 | 956 |
| Spring | Dec.03 - May.04 | Jun.04 - Nov.04 | 11 | 242 |
| | Dec.04 - May.05 | Jun.05 - Nov.05 | 17 | 252 |
| | Dec.05 - May.06 | Jun.06 - Nov.06 | 8 | 516 |
| XDoclet | Oct.01 - Mar.02 | Apr.02 - Sep.02 | 4 | 110 |
| | Apr.02 - Sep.02 | Oct.02 - Mar.03 | 5 | 206 |
| | Aug.02 - Jan.03 | Feb.03 - Jul.03 | 3 | 198 |

Table 4: Number of *Section Models* for analyzed Periods and Subset of Classes for each *Section Model* by Project (each subset consists of the same number of classes with and without bug fixes)

Models generated from refactoring features (47), non-refactoring features (62), and the set union of both feature groups (109) are used to discuss the first two hypotheses. Both hypotheses use classifier, respectively classification algorithm C4.5, LMT, JRip, and NNge.
For hypotheses 3–9 the analyzed models are solely based on classifier J48, which uses classification algorithm C4.5. On the basis of models generated from refactoring features only, we investigate hypotheses 3–5. This set of 47 refactoring features can give us a lot of information to reveal coherences within the generated models. The feature *refactoringChanges* and other selected features are discussed by hypothesis 3. For the analysis of hypotheses 4 and 5 two tables outline later on the used refactoring features together with information concerning the occurrence in decision rules of *section models* for the projects under investigation.
Models generated from author related features—chosen from the set of refac-

toring features, as well as non-refactoring features—are used to analyze hypotheses 6–9. This set of 8 author related features suffices to build models with good values for recall and precision. Therefore, the models can be considered as a good basis to investigate the interconnection between author related information and future bug fixes.
Table 5 enumerates the author related features we use, together with information concerning the occurrence in decision rules of *section models* for each analyzed project. The top three author related features of each open source project are marked bold to find out those that are of major importance. For projects Liferay Portal and XDoclet four values are bolded, since two features are third-placed. To tackle shortage of space within the table we abbreviate the project names: ArgoUML (A.), JBoss Cache (C.), Liferay Portal (L.), Spring Framework (S.), and XDoclet (X.).

| Author related Feature | A. | C. | L. | S. | X. |
|---|---|---|---|---|---|
| authorCount | **9** | **16** | **56** | **67** | **19** |
| authorSwitches | 3 | **19** | **15** | **31** | **9** |
| peopleSwitches | **6** | **23** | 6 | **29** | **18** |
| bugfixAuthorCount | 2 | 6 | **26** | 5 | 3 |
| bugfixAuthorSwitches | **8** | 13 | 11 | 19 | 8 |
| refactoringAuthorCount | 3 | 8 | **15** | 10 | **9** |
| refactoringAuthorSwitches | 3 | 12 | 13 | 18 | 7 |
| refactoringVSrevisionAuthorSwitches | 3 | 7 | — | 4 | 1 |

Table 5: Number of Decision Rules for Features in Models based on Author related Features (investigated projects abbreviated)

### 5.3.1 Do refactoring related features and non-refactoring related features lead to high quality prediction models?

We decided to exemplarily display the results for section model 1 of two different time periods per investigated project, whereby four different classifiers are used to generate the *section models* for each project (Tables 6– 15). Even though, this is only a small sample of the available prediction models, the results of the chosen models are representative for all generated models with respect to model quality.

| Project ArgoUML Jun.02 | Classifier (Algorithm) | Bug fix = 0 | | | Bug fix >= 1 | | |
|---|---|---|---|---|---|---|---|
| | | Prec. | Rec. | F-m | Prec. | Rec. | F-m |
| Refac-features | C4.5 | 0.716 | 0.720 | 0.718 | 0.718 | 0.714 | 0.716 |
| | LMT | 0.711 | 0.756 | 0.733 | 0.740 | 0.693 | 0.715 |
| | Rip | 0.716 | 0.729 | 0.722 | 0.724 | 0.711 | 0.717 |
| | NNge | 0.735 | 0.726 | 0.730 | 0.729 | 0.738 | 0.734 |
| Non-Refac-features | C4.5 | 0.716 | 0.735 | 0.725 | 0.728 | 0.708 | 0.718 |
| | LMT | 0.717 | 0.777 | 0.746 | 0.757 | 0.693 | 0.723 |
| | Rip | 0.736 | 0.747 | 0.741 | 0.743 | 0.732 | 0.737 |
| | NNge | 0.763 | 0.687 | 0.723 | 0.715 | 0.786 | 0.749 |
| Total-features | C4.5 | 0.713 | 0.780 | 0.745 | 0.757 | 0.687 | 0.720 |
| | LMT | 0.705 | 0.798 | 0.749 | 0.767 | 0.666 | 0.713 |
| | Rip | 0.734 | 0.732 | 0.733 | 0.733 | 0.735 | 0.734 |
| | NNge | 0.760 | 0.705 | 0.731 | 0.725 | 0.777 | 0.750 |

Table 6: Predicting non Bug fix prone vs Bug fix prone Classes for Project ArgoUML using Time Frame June 2002 – May 2003

| Project ArgoUML Jun.03 | Classifier (Algorithm) | Bug fix = 0 | | | Bug fix >= 1 | | |
|---|---|---|---|---|---|---|---|
| | | Prec. | Rec. | F-m | Prec. | Rec. | F-m |
| Refac-features | C4.5 | 0.794 | 0.852 | 0.822 | 0.841 | 0.779 | 0.809 |
| | LMT | 0.777 | 0.828 | 0.802 | 0.816 | 0.762 | 0.788 |
| | Rip | 0.731 | 0.779 | 0.754 | 0.763 | 0.713 | 0.737 |
| | NNge | 0.794 | 0.820 | 0.806 | 0.814 | 0.787 | 0.800 |
| Non-Refac-features | C4.5 | 0.758 | 0.795 | 0.776 | 0.784 | 0.746 | 0.765 |
| | LMT | 0.778 | 0.861 | 0.817 | 0.844 | 0.754 | 0.797 |
| | Rip | 0.777 | 0.828 | 0.802 | 0.816 | 0.762 | 0.788 |
| | NNge | 0.762 | 0.762 | 0.762 | 0.762 | 0.762 | 0.762 |
| Total-features | C4.5 | 0.766 | 0.779 | 0.772 | 0.775 | 0.762 | 0.769 |
| | LMT | 0.743 | 0.828 | 0.783 | 0.806 | 0.713 | 0.757 |
| | Rip | 0.764 | 0.795 | 0.779 | 0.786 | 0.754 | 0.770 |
| | NNge | 0.765 | 0.852 | 0.806 | 0.833 | 0.738 | 0.783 |

Table 7: Predicting non Bug fix prone vs Bug fix prone Classes for Project ArgoUML using Time Frame June 2003 – May 2004

| Project Cache Jul.05 | Classifier (Algorithm) | Bug fix = 0 | | | Bug fix >= 1 | | |
|---|---|---|---|---|---|---|---|
| | | Prec. | Rec. | F-m | Prec. | Rec. | F-m |
| Refac-features | C4.5 | 0.744 | 0.750 | 0.747 | 0.748 | 0.742 | 0.745 |
| | LMT | 0.681 | 0.734 | 0.707 | 0.712 | 0.656 | 0.683 |
| | Rip | 0.672 | 0.719 | 0.694 | 0.697 | 0.648 | 0.672 |
| | NNge | 0.701 | 0.586 | 0.638 | 0.644 | 0.750 | 0.693 |
| Non-Refac-features | C4.5 | 0.735 | 0.781 | 0.758 | 0.767 | 0.719 | 0.742 |
| | LMT | 0.767 | 0.773 | 0.770 | 0.772 | 0.766 | 0.769 |
| | Rip | 0.708 | 0.797 | 0.750 | 0.768 | 0.672 | 0.717 |
| | NNge | 0.741 | 0.672 | 0.705 | 0.700 | 0.766 | 0.731 |
| Total-features | C4.5 | 0.700 | 0.711 | 0.705 | 0.706 | 0.695 | 0.701 |
| | LMT | 0.746 | 0.758 | 0.752 | 0.754 | 0.742 | 0.748 |
| | Rip | 0.693 | 0.688 | 0.690 | 0.690 | 0.695 | 0.693 |
| | NNge | 0.695 | 0.711 | 0.703 | 0.704 | 0.688 | 0.696 |

Table 8: Predicting non Bug fix prone vs Bug fix prone Classes for Project JBoss Cache using Time Frame July 2005 – June 2006

| Project Cache Oct.05 | Classifier (Algorithm) | Bug fix = 0 | | | Bug fix >= 1 | | |
|---|---|---|---|---|---|---|---|
| | | Prec. | Rec. | F-m | Prec. | Rec. | F-m |
| Refac-features | C4.5 | 0.754 | 0.688 | 0.720 | 0.713 | 0.776 | 0.743 |
| | LMT | 0.758 | 0.728 | 0.743 | 0.738 | 0.768 | 0.753 |
| | Rip | 0.651 | 0.672 | 0.661 | 0.661 | 0.640 | 0.650 |
| | NNge | 0.766 | 0.656 | 0.707 | 0.699 | 0.800 | 0.746 |
| Non-Refac-features | C4.5 | 0.620 | 0.640 | 0.630 | 0.628 | 0.608 | 0.618 |
| | LMT | 0.690 | 0.712 | 0.701 | 0.702 | 0.680 | 0.691 |
| | Rip | 0.641 | 0.672 | 0.656 | 0.655 | 0.624 | 0.639 |
| | NNge | 0.702 | 0.584 | 0.638 | 0.644 | 0.752 | 0.694 |
| Total-features | C4.5 | 0.710 | 0.744 | 0.727 | 0.731 | 0.696 | 0.713 |
| | LMT | 0.689 | 0.672 | 0.680 | 0.680 | 0.696 | 0.688 |
| | Rip | 0.640 | 0.696 | 0.667 | 0.667 | 0.608 | 0.636 |
| | NNge | 0.711 | 0.648 | 0.678 | 0.676 | 0.736 | 0.705 |

Table 9: Predicting non Bug fix prone vs Bug fix prone Classes for Project JBoss Cache using Time Frame October 2005 – September 2006

| Project Liferay Feb.05 | Classifier (Algorithm) | Bug fix = 0 | | | Bug fix >= 1 | | |
|---|---|---|---|---|---|---|---|
| | | Prec. | Rec. | F-m | Prec. | Rec. | F-m |
| Refac-features | C4.5 | 0.927 | 0.924 | 0.925 | 0.924 | 0.927 | 0.925 |
| | LMT | 0.915 | 0.895 | 0.905 | 0.897 | 0.917 | 0.907 |
| | Rip | 0.923 | 0.911 | 0.917 | 0.912 | 0.924 | 0.918 |
| | NNge | 0.949 | 0.882 | 0.914 | 0.890 | 0.952 | 0.920 |
| Non-Refac-features | C4.5 | 0.925 | 0.908 | 0.916 | 0.909 | 0.927 | 0.918 |
| | LMT | 0.905 | 0.879 | 0.892 | 0.882 | 0.908 | 0.895 |
| | Rip | 0.968 | 0.879 | 0.922 | 0.889 | 0.971 | 0.928 |
| | NNge | 0.919 | 0.863 | 0.890 | 0.871 | 0.924 | 0.896 |
| Total-features | C4.5 | 0.941 | 0.920 | 0.931 | 0.922 | 0.943 | 0.932 |
| | LMT | 0.908 | 0.911 | 0.909 | 0.911 | 0.908 | 0.909 |
| | Rip | 0.956 | 0.901 | 0.928 | 0.907 | 0.959 | 0.932 |
| | NNge | 0.941 | 0.863 | 0.900 | 0.874 | 0.946 | 0.908 |

Table 10: Predicting non Bug fix prone vs Bug fix prone Classes for Project Liferay Portal using Time Frame February 2005 – January 2006

| Project Liferay Apr.05 | Classifier (Algorithm) | Bug fix = 0 | | | Bug fix >= 1 | | |
|---|---|---|---|---|---|---|---|
| | | Prec. | Rec. | F-m | Prec. | Rec. | F-m |
| Refac-features | C4.5 | 0.932 | 0.951 | 0.941 | 0.950 | 0.931 | 0.940 |
| | LMT | 0.923 | 0.942 | 0.933 | 0.941 | 0.922 | 0.931 |
| | Rip | 0.931 | 0.902 | 0.916 | 0.905 | 0.933 | 0.919 |
| | NNge | 0.944 | 0.908 | 0.926 | 0.912 | 0.946 | 0.929 |
| Non-Refac-features | C4.5 | 0.931 | 0.940 | 0.936 | 0.939 | 0.931 | 0.935 |
| | LMT | 0.930 | 0.922 | 0.926 | 0.923 | 0.931 | 0.927 |
| | Rip | 0.939 | 0.895 | 0.917 | 0.900 | 0.942 | 0.920 |
| | NNge | 0.922 | 0.902 | 0.912 | 0.904 | 0.924 | 0.914 |
| Total-features | C4.5 | 0.948 | 0.929 | 0.938 | 0.930 | 0.949 | 0.939 |
| | LMT | 0.942 | 0.951 | 0.947 | 0.950 | 0.942 | 0.946 |
| | Rip | 0.951 | 0.913 | 0.932 | 0.916 | 0.953 | 0.934 |
| | NNge | 0.948 | 0.893 | 0.920 | 0.899 | 0.951 | 0.924 |

Table 11: Predicting non Bug fix prone vs Bug fix prone Classes for Project Liferay Portal using Time Frame April 2005 – March 2006

| Project Spring Dec.03 | Classifier (Algorithm) | Bug fix = 0 | | | Bug fix >= 1 | | |
|---|---|---|---|---|---|---|---|
| | | Prec. | Rec. | F-m | Prec. | Rec. | F-m |
| Refac-features | C4.5 | 0.836 | 0.884 | 0.859 | 0.877 | 0.826 | 0.851 |
| | LMT | 0.857 | 0.793 | 0.824 | 0.808 | 0.868 | 0.837 |
| | Rip | 0.809 | 0.909 | 0.856 | 0.896 | 0.785 | 0.837 |
| | NNge | 0.849 | 0.835 | 0.842 | 0.837 | 0.851 | 0.844 |
| Non-Refac-features | C4.5 | 0.898 | 0.876 | 0.887 | 0.879 | 0.901 | 0.890 |
| | LMT | 0.854 | 0.967 | 0.907 | 0.962 | 0.835 | 0.894 |
| | Rip | 0.868 | 0.975 | 0.918 | 0.972 | 0.851 | 0.907 |
| | NNge | 0.884 | 0.884 | 0.884 | 0.884 | 0.884 | 0.884 |
| Total-features | C4.5 | 0.918 | 0.926 | 0.922 | 0.925 | 0.917 | 0.921 |
| | LMT | 0.893 | 0.967 | 0.929 | 0.964 | 0.884 | 0.922 |
| | Rip | 0.864 | 0.942 | 0.901 | 0.936 | 0.851 | 0.892 |
| | NNge | 0.899 | 0.884 | 0.892 | 0.886 | 0.901 | 0.893 |

Table 12: Predicting non Bug fix prone vs Bug fix prone Classes for Project Spring Framework using Time Frame December 2003 – November 2004

| Project Spring Dec.04 | Classifier (Algorithm) | Bug fix = 0 | | | Bug fix >= 1 | | |
|---|---|---|---|---|---|---|---|
| | | Prec. | Rec. | F-m | Prec. | Rec. | F-m |
| Refac-features | C4.5 | 0.872 | 0.921 | 0.896 | 0.916 | 0.865 | 0.890 |
| | LMT | 0.891 | 0.905 | 0.898 | 0.903 | 0.889 | 0.896 |
| | Rip | 0.880 | 0.929 | 0.903 | 0.924 | 0.873 | 0.898 |
| | NNge | 0.869 | 0.841 | 0.855 | 0.846 | 0.873 | 0.859 |
| Non-Refac-features | C4.5 | 0.937 | 0.937 | 0.937 | 0.937 | 0.937 | 0.937 |
| | LMT | 0.917 | 0.968 | 0.942 | 0.966 | 0.913 | 0.939 |
| | Rip | 0.886 | 0.929 | 0.907 | 0.925 | 0.881 | 0.902 |
| | NNge | 0.919 | 0.905 | 0.912 | 0.906 | 0.921 | 0.913 |
| Total-features | C4.5 | 0.937 | 0.944 | 0.941 | 0.944 | 0.937 | 0.940 |
| | LMT | 0.904 | 0.976 | 0.939 | 0.974 | 0.897 | 0.934 |
| | Rip | 0.889 | 0.952 | 0.920 | 0.949 | 0.881 | 0.914 |
| | NNge | 0.911 | 0.897 | 0.904 | 0.898 | 0.913 | 0.906 |

Table 13: Predicting non Bug fix prone vs Bug fix prone Classes for Project Spring Framework using Time Frame December 2004 – November 2005

| Project XDoclet Apr.02 | Classifier (Algorithm) | Bug fix = 0 | | | Bug fix >= 1 | | |
|---|---|---|---|---|---|---|---|
| | | Prec. | Rec. | F-m | Prec. | Rec. | F-m |
| Refac-features | C4.5 | 0.790 | 0.806 | 0.798 | 0.802 | 0.786 | 0.794 |
| | LMT | 0.768 | 0.835 | 0.800 | 0.819 | 0.748 | 0.782 |
| | Rip | 0.812 | 0.796 | 0.804 | 0.800 | 0.816 | 0.808 |
| | NNge | 0.847 | 0.806 | 0.826 | 0.815 | 0.854 | 0.834 |
| Non-Refac-features | C4.5 | 0.817 | 0.913 | 0.862 | 0.901 | 0.796 | 0.845 |
| | LMT | 0.891 | 0.874 | 0.882 | 0.876 | 0.893 | 0.885 |
| | Rip | 0.863 | 0.854 | 0.859 | 0.856 | 0.864 | 0.860 |
| | NNge | 0.920 | 0.777 | 0.842 | 0.807 | 0.932 | 0.865 |
| Total-features | C4.5 | 0.893 | 0.893 | 0.893 | 0.893 | 0.893 | 0.893 |
| | LMT | 0.904 | 0.913 | 0.908 | 0.912 | 0.903 | 0.907 |
| | Rip | 0.819 | 0.835 | 0.827 | 0.832 | 0.816 | 0.824 |
| | NNge | 0.892 | 0.806 | 0.847 | 0.823 | 0.903 | 0.861 |

Table 14: Predicting non Bug fix prone vs Bug fix prone Classes for Project XDoclet using Time Frame April 2002 – March 2003

| Project XDoclet Aug.02 | Classifier (Algorithm) | Bug fix = 0 | | | Bug fix >= 1 | | |
|---|---|---|---|---|---|---|---|
| | | Prec. | Rec. | F-m | Prec. | Rec. | F-m |
| Refac-features | C4.5 | 0.796 | 0.828 | 0.812 | 0.821 | 0.788 | 0.804 |
| | LMT | 0.765 | 0.788 | 0.776 | 0.781 | 0.758 | 0.769 |
| | Rip | 0.742 | 0.727 | 0.735 | 0.733 | 0.747 | 0.740 |
| | NNge | 0.745 | 0.737 | 0.741 | 0.740 | 0.747 | 0.744 |
| Non-Refac-features | C4.5 | 0.750 | 0.818 | 0.783 | 0.800 | 0.727 | 0.762 |
| | LMT | 0.794 | 0.818 | 0.806 | 0.813 | 0.788 | 0.800 |
| | Rip | 0.723 | 0.737 | 0.730 | 0.732 | 0.717 | 0.724 |
| | NNge | 0.842 | 0.646 | 0.731 | 0.713 | 0.879 | 0.787 |
| Total-features | C4.5 | 0.778 | 0.778 | 0.778 | 0.778 | 0.778 | 0.778 |
| | LMT | 0.808 | 0.808 | 0.808 | 0.808 | 0.808 | 0.808 |
| | Rip | 0.793 | 0.737 | 0.764 | 0.755 | 0.808 | 0.780 |
| | NNge | 0.805 | 0.707 | 0.753 | 0.739 | 0.828 | 0.781 |

Table 15: Predicting non Bug fix prone vs Bug fix prone Classes for Project XDoclet using Time Frame August 2002 – July 2003

Let us take a closer look at Table 6 to discuss the presented information. The results for each set of features are presented using four lines, one line for each classification algorithm. In other words, the first four lines are based on the same set of classes, but lead to four unique *section models*, since different classifiers are used. Each line provides statistical information about model quality to estimate, to what extent the considered classes are assigned correctly. The first line has to be interpreted in the following way:
Section model 1 for project ArgoUML using time frame June 2002 – May 2003 is generated by the usage of refactoring features and classification algorithm C4.5. Thereby, classes with or without bug fixes are assigned correctly with certain values of precision, recall, and f-measure. Classes with no bugs in the target period are assigned correctly to bin "No bugs" with a precision of 0.716, a recall of 0.720, and a f-measure of 0.718. Classes with one or more bugs in the target period are assigned correctly to bin "One or more bugs" with a precision of 0.718, a recall of 0.714, and a f-measure of 0.716.

- ArgoUML

  Table 6 and Table 7 show a F-measure, which ranges from 0.718-0.822 (non-error prone) and 0.715-0.809 (error prone) for refactoring features and 0.723-0.817 (non-error prone) and 0.718-0.797 (error prone) for non-refactoring features.

- JBoss Cache

  Table 8 and Table 9 show a F-measure, which ranges from 0.638-0.747 (non-error prone) and 0.650-0.753 (error prone) for refactoring features and 0.630-0.770 (non-error prone) and 0.618-0.769 (error prone) for non-refactoring features.

- Liferay Portal

  Table 10 and Table 11 show a F-measure, which ranges from 0.905-0.941 (non-error prone) and 0.907-0.940 (error prone) for refactoring features and 0.890-0.936 (non-error prone) and 0.895-0.935 (error prone) for non-refactoring features.

- Spring Framework

  Table 12 and Table 13 show a F-measure, which ranges from 0.824-0.903 (non-error prone) and 0.837-0.898 (error prone) for refactoring features

and 0.884-0.942 (non-error prone) and 0.890-0.939 (error prone) for non-refactoring features.

- XDoclet

  Table 14 and Table 15 show a F-measure, which ranges from 0.735-0.826 (non-error prone) and 0.740-0.834 (error prone) for refactoring features and 0.730-0.882 (non-error prone) and 0.724-0.885 (error prone) for non-refactoring features.

We are of the opinion that all models of our investigated projects show sufficient good prediction results to form a basis for the further analysis. Especially, the distribution between both bins (classes with and without bug fixes) is satisfying. Liferay Portal and Spring Framework exhibit extraordinary good prediction results. The maximum value of F-measure is 0.941 for Liferay and 0.942 for Spring.
These results represent a good basis to interpret the composition of the model trees, since concrete sequences of the decision rules lead to promising prediction results.

Thus, we conclude:

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

### 5.3.2 Is the quality of prediction models improved by combining features of both groups?

This is another question of great importance to find out interrelations between both feature groups. We expect no great differences, since the prediction quality is already on a high level, although small improvements seem to be reachable.

- ArgoUML

  Table 6 and Table 7 show a F-measure, which ranges from 0.731-0.806 (non-error prone) and 0.713-0.783 (error prone) for the total set of refactoring and non-refactoring features.

- JBoss Cache

  Table 8 and Table 9 show a F-measure, which ranges from 0.667-0.752 (non-error prone) and 0.636-0.748 (error prone) for the total set of refactoring and non-refactoring features.

- Liferay Portal

  Table 10 and Table 11 show a F-measure, which ranges from 0.900-0.947 (non-error prone) and 0.908-0.946 (error prone) for the total set of refactoring and non-refactoring features.

- Spring Framework

  Table 12 and Table 13 show a F-measure, which ranges from 0.892-0.941 (non-error prone) and 0.892-0.940 (error prone) for the total set of refactoring and non-refactoring features.

- XDoclet

  Table 14 and Table 15 show a F-measure, which ranges from 0.753-0.908 (non-error prone) and 0.778-0.907 (error prone) for the total set of refactoring and non-refactoring features.

The prediction models of all projects generated from the total set of features, display very similar results in relation to the models based on features of each group, separately. Some prediction models based on all features present a slight improvement, in comparison to models, consisting only of, either refactoring, or non-refactoring features. For example Liferay Portal shows values for f-measure based on the total set of features often above the values for f-measure for models based on refactoring, or non-refactoring features only.

Thus, we conclude:

| The quality of prediction models is generally kept constant and sometimes slightly improved by combining refactoring and non-refactoring features. |
|---|

### 5.3.3 Should refactoring be associated with other software activities in a certain way to reduce software defect appearance?

We investigate *section models* of all projects under investigation that contain the refactoring feature of interest. Tables for the selected features show the tree level where that feature is used, the particular threshold value, and the percentage of classes assigned to a certain bin by that rule, if a value below or above the threshold value is used.
To examine this hypothesis in detail, it is necessary to analyze refactoring features, which relate refactoring properties to other software properties, or more generic to overall change properties—overall changes can be different kinds of software events, naturally including refactorings and bug fixes. A set of 16 refactoring features out of the total quantity of refactoring features (47) fulfills that condition and can give us valuable information to reveal coherences:

- *refactoringLinesAdd*
- *refactoringLinesChange*
- *refactoringLinesDel*
- *refactoringTLinesAdd*
- *refactoringTLinesChange*
- *refactoringTLinesDel*
- *bugfixVersusRefactoringLinesAdd*
- *bugfixVersusRefactoringLinesChange*
- *bugfixVersusRefactoringLinesDel*
- *TBugfixVersusTRefactoringLinesAdd*
- *TBugfixVersusTRefactoringLinesChange*
- *TBugfixVersusTRefactoringLinesDel*
- *bugfixVersusRefactoringPeakMonth*
- *refactoringVersusChangePeakMonth*

- *bugfixRefactorings*
- *refactoringChanges*

We concentrate on the analysis of six out of the total quantity of relevant refactoring features (16), namely *refactoringChanges*, *bugfixRefactorings*, *bugfixVersusRefactoringLinesChange*, *TBugfixVersusTRefactoringLinesChange*, *bugfixVersusRefactoringPeakMonth*, and *refactoringVersusChangePeakMonth*. These features are interesting indicators for the occurrence of future bug fixes.
Some of the tree sequences—including the feature *refactoringChanges*—for all investigated open source projects are outlined as examples (Figure 17– 18). Let us take a closer look at Figure 17 to better understand the coherences. The first example shows data of section model 2 for project ArgoUML using time frame June 2002 — May 2003. Feature *refactoringChanges* appears three times in the presented *model sequences*. Hereby, the first *model sequence* consists of only one decision. The decision rule at the first tree level includes two decisions (Lines 1 and 2) and the decision rule at the sixth tree level is made up of only one decision (Line 7). The first line means that 61 instances (classes) of 664 instances (see Table 4, first line, column "Classes per section model") are assigned to bin "One or more bugs", whereby only one class is not correctly assigned. The result of the division of 60 by 664 (9.0%) is recorded in Table 16 at the third line in column "<=" of "Bug fix>=1" and has the following meaning: 9.0% of classes show bug fixes in the target period, if they hold values equal or below 13% for feature refactoringChanges in the corresponding feature period. As a second example we discuss the second decision rule at tree level six that is part of a larger *model sequence*. This time 72 classes are assigned to bin "No bugs", with 11 classes misclassified. Now, 9.2% of classes show no bug fixes in the target period, if a threshold value of 25% for feature *refactoringChanges* is overrun. The corresponding entry is made in Table 16 at the fourth line in column ">" of "Bug fix=0".

**refactoringChanges** Tables 16-18 list the models for each analyzed project, where the feature *refactoringChanges* appears. A threshold value of 1 means that every change in the feature period has been a refactoring, since refactoring is also a certain change type.
Classes that are assigned to bin "No bugs" hold values for *refactoringChanges* equal or below a given threshold value 15 times, and 54 times above a certain

```
Decision tree C4.5
------------------

ArgoUML
 jun02_section2
 --------------
refactoringChanges <= 0.125: '(0.5-inf)' (61.0/1.0)
refactoringChanges > 0.125
|refactoringLinesChangePerRefactoring <= 3
| | bugfixVersusRefactoringLinesAdd <= 3
| | | TBugfixVersusTRefactoringLinesAdd <= 8
| | | | TBugfixVersusTRefactoringLinesChange <= 4
| | | | | refactoringChanges > 0.25
| | | | | | refactoringFrequencyBefore <= 0.058824
| | | | | | | refactoringChangesType <= 0
| | | | | | | | refactoringTLinesType > -97
| | | | | | | | | refactoringVersusChangePeakMonth > 0.2
| | | | | | | | | | refactoringFrequencyBefore <= 0
| | | | | | | | | | | TBugfixVersusTRefactoringLinesAdd <= 0.018868
| | | | | | | | | | | | refactoringLinesChange <= 0.75: '(-inf-0.5]' (72.0/11.0)


JBoss Cache
 jan06_section1
 --------------
refactoringPeakMonth > 0.833333
| refactoringChanges > 0.4
| | TBugfixVersusTRefactoringLinesAdd > 0.002488: '(-inf-0.5]' (40.0/1.0)


Liferay Portal
 jun05_section1
 --------------
relRefactorings <= 0
| linesActivityRateRefactorings <= 0.03125: '(-inf-0.5]' (63.0/2.0)
| linesActivityRateRefactorings > 0.03125
| | bugfixVersusRefactoringLinesDel <= 4
| | | refactoringChanges <= 0.333333
| | | | refactoringLinesChange > 0.125
| | | | | refactoringVersusChangePeakMonth > 0.166667
| | | | | | TBugfixVersusTRefactoringLinesDel <= 106: '(-inf-0.5]' (51.0/4.0)
| | | refactoringChanges > 0.333333
```

Figure 17: Tree Sequences including Feature *refactoringChanges* based on Classification Algorithm C4.5 — *PART 1*

```
Decision tree C4.5
------------------

Spring Framework
 dec05_section7
 --------------
refactoringChanges <= 0.181818
| refactoringChanges <= 0.125: '(0.5-inf)' (44.0)
refactoringChanges > 0.181818
| refactoringCoChangedFiles <= 38.5
| | bugfixVersusRefactoringLinesAdd <= 0.037037
| | | refactoringCoChangedFiles <= 16.5
| | | | refactoringChangesType <= 0
| | | | | refactoringsBefore <= 0.194444
| | | | | | refactoringsBefore <= 0
| | | | | | | refactoringChanges > 0.285714
| | | | | | | | refactoringLargeTransactions <= 0.5
| | | | | | | | | bugfixRefactorings <= 0.5
| | | | | | | | | | refactoringTLinesChange <= 0.030303: '(-inf-0.5]' (45.0/11.0)
| | | | | refactoringsBefore > 0.194444
| | | | | | linesActivityRateRefactorings <= 0.142857: '(-inf-0.5]' (83.0/17.0)
| | bugfixVersusRefactoringLinesAdd > 0.037037
| | | refactoringsBefore <= 0.157895: '(0.5-inf)' (76.0/14.0)
| refactoringCoChangedFiles > 38.5: '(-inf-0.5]' (30.0/1.0)


XDoclet
 oct01_section4
 --------------
refactoringChanges <= 0.235294
| refactoringVersusChangePeakMonth <= 1.25
| | avgDaysBetweenRefactorings <= 31.2: '(0.5-inf)' (27.0)
| | avgDaysBetweenRefactorings > 31.2
| | | refactoringVersusChangePeakMonth <= 0.6
| | | | bugfixVersusRefactoringPeakMonth <= 2: '(0.5-inf)' (6.0)
refactoringChanges > 0.235294
| linesActivityRateRefactorings <= 34.8
| | refactoringLinesChange > 0.02439
| | | refactoringLinesAdd <= 0.041667: '(-inf-0.5]' (33.0/1.0)
| | | refactoringLinesAdd > 0.041667
| | | | refactoringChangingChanges <= 0.666667
| | | | | refactoringPeakMonth <= 0.5
| | | | | | refactoringChanges <= 0.333333
| | | | | | | avgDaysPerRefactoringLine <= 1.068493: '(-inf-0.5]' (9.0/1.0)
| | | | refactoringChangingChanges > 0.666667: '(-inf-0.5]' (6.0)
| linesActivityRateRefactorings > 34.8: '(0.5-inf)' (7.0)
```

Figure 18: Tree Sequences including Feature *refactoringChanges* based on Classification Algorithm C4.5 — *PART 2*

limit. Classes that are assigned to bin "One or more bugs" hold values for *refactoringChanges* equal or below a given threshold value 38 times, and 18 times above a given limit.
Thus, the majority of *model sequences*, using the feature *refactoringChanges* shows that instances holding a value equal or below a distinct threshold value are assigned to bin "One or more bugs" (71.7%), and instances above are assigned to bin "No bugs" (75.0%). In other words, the number of bug fixes in the target period decreases, if the number of refactorings increases and the number of overall changes decreases. The same rationale applies to the contrary situation where the number of bug fixes in the target period increases.
This is an very important 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. Recapitulatory, we can say that a well-balanced proportion between refactoring and total changes makes the source code clearer, as well as allows to introduce new features and improvements.

**bugfixRefactorings** Table 19 lists the models for each analyzed project, where the feature *bugfixRefactorings* appears. A threshold value of 1 means that a certain number of bug fixes is related to the same number of refactorings in the feature period.
Classes that are assigned to bin "No bugs" hold values for *bugfixRefactorings* equal or below a given threshold value 18 times, and 2 times above a certain limit. Classes that are assigned to bin "One or more bugs" hold values for *bugfixRefactorings* equal or below a given threshold value 8 times, and 7 times above a given limit.
Hence, most of the *model sequences*, using feature *bugfixRefactorings* show that classes, which hold a value equal or below a distinct threshold value are assigned to bin "No bugs" (69.2%), and classes above are assigned to bin "One or more bugs" (77.8%). Thus in general, the number of bug fixes in the target period decreases, if the number of refactorings increases and the number of bug fixes decreases. The same rationale applies to the contrary situation where the number of bug fixes in the target period increases.
The influence of refactoring remains the same as by feature *refactoringChanges*; refactoring helps to decrease bug fixes in the target period. Secondly, the

| Project | Model No. | Tree-Level | Thres-hold | Bug fix=0 [%] | | Bug fix>=1 [%] | |
|---|---|---|---|---|---|---|---|
| | | | | <= | > | <= | > |
| ArgoUML-jun02 | 1 | 1 | 0.12 | — | — | 8.0 | — |
| | | 8 | 0.18 | — | 20.0 | — | — |
| | 2 | 1 | 0.13 | — | 9.2 | 9.0 | — |
| | | 6 | 0.25 | — | 9.2 | — | — |
| ArgoUML-jun03 | 1 | 3 | 0.17 | — | 34.8 | — | — |
| | 6 | 4 | 0.44 | 25.0 | 9.8 | — | — |
| | 7 | 5 | 0.18 | 5.3 | 27.0 | — | — |
| ArgoUML-jun04 | 1 | 6 | 0.29 | — | 11.1 | — | — |
| | 2 | 3 | 0.18 | — | — | 6.7 | — |
| | 4 | 2 | 0.38 | — | — | 12.3 | — |
| Cache-jul05 | 1 | 6 | 0.22 | — | 30.5 | — | — |
| Cache-oct05 | 1 | 7 | 0.40 | — | — | 7.5 | — |
| Cache-jan06 | 1 | 2 | 0.40 | — | 18.1 | — | — |
| Liferay-feb05 | 2 | 6 | 0.17 | — | 5.9 | — | — |
| | 4 | 4 | 0.17 | — | 6.8 | — | — |
| | 5 | 3 | 0.40 | — | — | 17.7 | 11.9 |
| | 7 | 5 | 0.17 | — | 7.5 | — | — |
| Liferay-apr05 | 1 | 3 | 0.50 | — | 7.3 | 11.8 | — |
| | 7 | 3 | 0.50 | — | — | 19.4 | — |
| Liferay-jun05 | 1 | 4 | 0.33 | — | — | 5.6 | — |
| | 2 | 6 | 0.50 | 14.7 | — | — | — |
| | 3 | 8 | 0.33 | — | — | 13.4 | — |
| | | 7 | 0.20 | — | 6.8 | — | — |
| XDoclet-oct01 | 2 | 3 | 0.22 | — | 45.5 | — | — |
| | 4 | 1 | 0.24 | — | 41.8 | 30.0 | 6.4 |
| | | 7 | 0.33 | — | — | — | 7.3 |
| XDoclet-apr02 | 1 | 3 | 0.35 | — | 34.0 | 14.1 | 6.3 |
| | 3 | 2 | 0.56 | 11.7 | 21.4 | 9.7 | — |
| | 5 | 3 | 0.29 | — | 9.7 | — | — |
| XDoclet-aug02 | 1 | 4 | 0.22 | — | — | 9.1 | 7.6 |
| | | 3 | 0.40 | — | 7.1 | — | — |
| | 2 | 6 | 0.40 | — | 12.1 | — | — |
| | 3 | 3 | 0.25 | — | — | — | 7.1 |

Table 16: Percentage of Classes with or without Bug fixes based on Threshold Values of Decision Rules concerning Feature *refactoringChanges — PART 1*

| Project | Model No. | Tree-Level | Thres-hold | Bug fix=0 [%] | | Bug fix>=1 [%] | |
|---|---|---|---|---|---|---|---|
| | | | | <= | > | <= | > |
| Spring-dec03 | 1 | 2 | 0.50 | — | 6.2 | 36.4 | — |
| | 2 | 8 | 0.29 | — | — | 7.4 | — |
| | 4 | 3 | 0.22 | — | 11.2 | — | — |
| | | 5 | 0.67 | 8.7 | 5.4 | 6.2 | — |
| | 5 | 1 | 0.50 | 19.4 | 15.3 | 25.2 | — |
| | | 14 | 0.29 | — | 5.4 | — | — |
| | 6 | 4 | 0.50 | — | — | 7.9 | — |
| | 8 | 8 | 0.29 | — | 15.7 | — | — |
| | 9 | 2 | 0.22 | — | 34.3 | 11.2 | 6.2 |
| | | 8 | 0.40 | — | 15.7 | — | — |
| | 10 | 8 | 0.21 | — | 19.4 | 5.8 | — |
| Spring-dec04 | 2 | 1 | 0.67 | 17.9 | 21.0 | 29.8 | — |
| | 3 | 7 | 0.04 | — | — | — | 7.5 |
| | 4 | 1 | 0.17 | — | 37.7 | 17.9 | 23.0 |
| | 5 | 1 | 0.29 | — | 32.1 | 15.9 | 5.6 |
| | | 5 | 0.40 | 10.7 | 5.2 | — | — |
| | | 8 | 0.67 | 5.6 | 5.2 | — | — |
| | 6 | 2 | 0.17 | — | 38.9 | — | 5.6 |
| | 7 | 1 | 0.22 | — | 40.9 | 23.4 | 10.3 |
| | 8 | 1 | 0.20 | — | 33.7 | 22.2 | 15.1 |
| | 9 | 1 | 0.22 | — | 35.7 | 22.6 | 7.1 |
| | | 4 | 0.67 | 16.7 | 19.0 | — | — |
| | 10 | 1 | 0.15 | — | 39.3 | 15.5 | 19.8 |
| | 11 | 2 | 0.26 | — | 39.7 | 16.3 | — |
| | | 4 | 0.22 | — | — | 16.3 | — |
| | 12 | 9 | 0.22 | — | 5.6 | — | — |
| | | 10 | 0.33 | — | — | 5.6 | — |
| | 13 | 2 | 0.50 | — | — | 30.6 | — |
| | | 10 | 0.40 | 10.3 | — | — | — |
| | 14 | 3 | 0.36 | 38.1 | — | — | — |
| | | 4 | 0.40 | — | 7.1 | — | — |
| | 15 | 2 | 0.17 | — | 38.9 | — | — |
| | | 6 | 0.22 | — | 15.5 | — | — |

Table 17: Percentage of Classes with or without Bug fixes based on Threshold Values of Decision Rules concerning Feature *refactoringChanges — PART 2*

| Project | Model No. | Tree-Level | Thres-hold | Bug fix=0 [%] | | Bug fix>=1 [%] | |
|---|---|---|---|---|---|---|---|
| | | | | <= | > | <= | > |
| Spring-dec04 | 16 | 2 | 0.15 | — | 42.9 | 9.5 | — |
| | | 6 | 0.29 | — | 30.1 | — | — |
| | 17 | 4 | 0.22 | — | 18.7 | — | — |
| | | 5 | 0.40 | 12.3 | 6.3 | — | — |
| | | 7 | 0.67 | — | 6.3 | — | — |
| Spring-dec05 | 1 | 2 | 0.157895 | — | — | 7.6 | — |
| | | 3 | 0.75 | 10.1 | — | — | — |
| | | 11 | 0.285714 | — | — | — | 13.8 |
| | 2 | 7 | 0.285714 | — | — | — | 5.2 |
| | 3 | 4 | 0.666667 | — | — | 10.7 | — |
| | 5 | 6 | 0.285714 | 9.1 | 5.6 | — | — |
| | 7 | 1 | 0.181818 | — | 31.4 | 8.5 | 12.0 |
| | | 2 | 0.125 | — | — | 8.5 | — |
| | | 8 | 0.285714 | — | 6.6 | — | — |
| | 8 | 2 | 0.4 | — | — | 20.2 | — |

Table 18: Percentage of Classes with or without Bug fixes based on Threshold Values of Decision Rules concerning Feature *refactoringChanges — PART 3*

number of bug fixes in the feature period directly correlates to the number of bug fixes in the target period. This is an result, well known to the scientific community, since error-prone classes tend to produce more errors 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.

**bugfixVersusRefactoringLinesChange** Table 20 lists the models for each analyzed project, where the feature *bugfixVersusRefactoringLinesChange* appears. A threshold value of 1 means that a certain number of bug fix lines changed is related to the same number of refactoring lines changed in the feature period.
Classes that are assigned to bin "No bugs" hold values for *bugfixVersusRefactoringLinesChange* equal or below a given threshold value 5 times, and 2 times above a certain limit. Classes that are assigned to bin "One or more bugs" hold values for *bugfixVersusRefactoringLinesChange* equal or below a given threshold value 2 times, and 1 time above a given limit.
The majority of *model sequences*, using feature *bugfixVersusRefactoringLinesChange* shows that instances holding a value below a distinct threshold value are assigned to bin "No bugs" (71.4%). For instances holding values above a given threshold value no clear statement can be made, since only 3 cases are available. Two times the regarded instances are assigned to bin "No bugs", and one time to bin "One or more bugs".
In other words, most of the time the number of bug fixes in the target period decreases, if the number of bug fix lines changed decreases, and the number of refactoring lines changed increases. The complementary coherence can not be proved for lack of decision rules that assign classes above a certain threshold value.

**TBugfixVersusTRefactoringLinesChange** Tables 21 lists the models for each analyzed project, where the feature *TBugfixVersusTRefactoringLinesChange* appears. A threshold value of 1 means that a certain number of bug fix lines changed per transaction is related to the same number of refactoring lines changed per transaction in the feature period.
Classes that are assigned to bin "No bugs" hold values for *TBugfixVersus-*

| Project | Model No. | Tree-Level | Thres-hold | Bug fix=0 [%] | | Bug fix>=1 [%] | |
|---|---|---|---|---|---|---|---|
| | | | | <= | > | <= | > |
| ArgoUML-jun03 | 3 | 2 | 0.25 | 24.6 | — | — | — |
| ArgoUML-jun04 | 1 | 2 | 1.25 | 28.8 | — | 5.4 | — |
| | 4 | 6 | 0.5 | — | — | 9.9 | — |
| Cache-jul05 | 1 | 4 | 0.33 | — | — | — | 5.5 |
| Cache-oct05 | 1 | 2 | 2 | 18.4 | — | 16.8 | 6.8 |
| | 2 | 1 | 1 | 26.8 | — | 10.4 | 24.4 |
| Cache-jan06 | 1 | 8 | 0.75 | 7.8 | — | — | — |
| Liferay-feb05 | 1 | 2 | 0 | — | — | — | 37.3 |
| | 2 | 5 | 0.5 | 20.4 | 15.0 | — | — |
| | 4 | 5 | 0.5 | 6.8 | — | — | — |
| | 7 | 6 | 0.5 | 7.5 | — | — | — |
| Liferay-apr05 | 5 | 6 | 0.5 | 9.7 | — | — | — |
| | 6 | 5 | 0.5 | — | 5.9 | — | — |
| | 7 | 4 | 1 | — | — | 5.2 | — |
| Liferay-jun05 | 2 | 3 | 1 | 5.4 | — | — | — |
| Spring-dec03 | 5 | 13 | 0.5 | 5.8 | — | — | — |
| | 6 | 3 | 1 | 8.7 | — | — | — |
| Spring-dec05 | 1 | 1 | 1 | 17.6 | — | 19.0 | 7.6 |
| | 2 | 2 | 1 | 10.3 | — | 20.3 | 7.4 |
| | 3 | 1 | 1 | 26.4 | — | 15.7 | 13.8 |
| | 6 | 1 | 1 | 19.4 | — | — | — |
| | 7 | 10 | 0.5 | 6.6 | — | — | — |
| XDoclet-aug02 | 2 | 4 | 1 | 34.3 | — | — | — |

Table 19: Percentage of Classes with or without Bug fixes based on Threshold Values of Decision Rules concerning Feature *bugfixRefactorings*

| Project | Model No. | Tree-Level | Thres-hold | Bug fix=0 [%] | | Bug fix>=1 [%] | |
|---|---|---|---|---|---|---|---|
| | | | | <= | > | <= | > |
| Cache-jul05 | 2 | 6 | 0.02 | — | 10.2 | — | — |
| Cache-oct05 | 1 | 7 | 0.5 | 10.8 | — | — | — |
| Liferay-feb05 | 3 | 3 | 0.78 | — | 29.1 | — | — |
| Liferay-apr05 | 1 | 5 | 1.5 | 7.3 | — | — | — |
| | 2 | 4 | 3.33 | 8.3 | — | — | — |
| Liferay-jun05 | 3 | 2 | 0.5 | — | — | 10.3 | — |
| Spring-dec03 | 9 | 7 | 0.5 | 15.7 | — | — | — |
| | 11 | 4 | 0 | — | — | — | 12.8 |
| Spring-dec04 | 8 | 3 | 0.04 | 33.7 | — | — | — |
| Spring-dec05 | 2 | 8 | 4.07 | — | — | 15.1 | — |

Table 20: Percentage of Classes with or without Bug fixes based on Threshold Values of Decision Rules concerning Feature *bugfixVersusRefactoringLinesChange*

*TRefactoringLinesChange* equal or below a given threshold value 13 times, and 1 time above a certain limit. Classes that are assigned to bin "One or more bugs" hold values for *TBugfixVersusTRefactoringLinesChange* equal or below a given threshold value 5 times, and 5 times above a given limit.
The majority of *model sequences*, using the feature *TBugfixVersusTRefactoringLinesChange* shows that instances holding a value below a distinct threshold value are assigned to bin "No bugs" (72.2%), and instances above are assigned to bin "One or more bugs" (83.3%).
In other words, most of the time the number of bug fixes in the target period decreases, if the number of bug fix lines changed per transaction decreases, and the number of refactoring lines changed per transaction increases. The same rationale applies to the contrary situation where the number of bug fixes in the target period increases.

**bugfixVersusRefactoringPeakMonth** Table 22 lists the models for each analyzed project, where the feature *bugfixVersusRefactoringPeakMonth* appears. A threshold value of 1 means that the bug fix peak month and the refactoring peak month are the same within the feature period.

| Project | Model No. | Tree-Level | Thres-hold | Bug fix=0 [%] | | Bug fix>=1 [%] | |
|---|---|---|---|---|---|---|---|
| | | | | <= | > | <= | > |
| ArgoUML-jun02 | 2 | 5 | 4 | 18.7 | — | — | — |
| Cache-jul05 | 2 | 4 | 642 | 5.5 | — | 18.8 | — |
| Cache-oct05 | 1 | 3 | 28 | 18.4 | — | 16.8 | — |
| Cache-jan06 | 1 | 3 | 26 | 13.5 | — | 14.5 | 8.2 |
| | | 6 | 13 | 7.8 | — | — | — |
| | | 7 | 13 | — | — | 14.5 | — |
| Liferay-feb05 | 2 | 5 | 0.001 | 5.9 | — | — | — |
| | | 5 | 184 | — | — | 7.0 | — |
| | 3 | 5 | 0.06 | — | — | — | 10.4 |
| Liferay-apr05 | 2 | 6 | 1.22 | 8.3 | — | — | — |
| | 3 | 7 | 0.001 | 8.0 | — | — | — |
| Spring-dec03 | 3 | 8 | 0.008 | 5.8 | — | — | — |
| | 9 | 5 | 0.009 | — | — | — | 5.4 |
| Spring-dec05 | 13 | 8 | 11 | 10.3 | — | — | — |
| Spring-dec05 | 3 | 4 | 72 | 26.4 | — | — | — |
| | 4 | 3 | 110 | 37.2 | — | — | 5.4 |
| | 5 | 8 | 127 | 5.6 | — | — | — |
| XDoclet-apr02 | 1 | 3 | 0.001 | — | — | — | 14.1 |
| | 5 | 4 | 0.06 | — | 9.7 | — | — |

Table 21: Percentage of Classes with or without Bug fixes based on Threshold Values of Decision Rules concerning Feature *TBugfixVersusTRefactoringLinesChange*

Classes that are assigned to bin "No bugs" hold values for *bugfixVersus-RefactoringPeakMonth* equal or below a given threshold value 21 times, and 2 times above a certain limit. Classes that are assigned to bin "One or more bugs" hold values for *bugfixVersusRefactoringPeakMonth* equal or below a given threshold value 16 times, and 18 times above a given limit.
The majority of *model sequences*, using feature *bugfixVersusRefactoringPeak-Month* shows that instances holding a value below a distinct threshold value are assigned to bin "No bugs" (56.8%), and instances above are assigned to bin "One or more bugs" (90.0%).
In other words, the number of bug fixes in the target period decreases, if the bug fix peak month is before the refactoring peak month (threshold value <1), or if there is a small distance between both peak months, when the refactoring peak month precedes the bug fix peak month. The same rationale applies to the contrary situation where the number of bug fixes in the target period increases.

**refactoringVersusChangePeakMonth** Tables 23–24 list the models for each analyzed project, where the feature *refactoringVersusChangePeakMonth* appears. A threshold value of 1 means that the refactoring peak month and the peak month of overall changes—that include e.g. bug fixes, or refactorings—are the same within the feature period.
Classes that are assigned to bin "No bugs" hold values for *refactoringVersusChangePeakMonth* equal or below a given threshold value 23 times, and 32 times above a certain limit. Classes that are assigned to bin "One or more bugs" hold values for *refactoringVersusChangePeakMonth* equal or below a given threshold value 18 times, and 25 times above a given limit.
All *model sequences*, using feature *refactoringVersusChangePeakMonth* show a balanced proportion. Instances holding a value below or above a certain threshold value are assigned to bin "No bugs" (56.1%), and bin "One or more bugs" (43.9%) with similar probability. This means that the relation between the refactoring peak month and the peak month of overall changes seems to have no obvious influence on the occurrence of bug fixes.

Thus, we conclude:

| Project | Model No. | Tree-Level | Thres-hold | Bug fix=0 [%] | | Bug fix>=1 [%] | |
|---|---|---|---|---|---|---|---|
| | | | | <= | > | <= | > |
| ArgoUML-jun02 | 1 | 2 | 2 | 20.0 | — | — | — |
| ArgoUML-jun04 | 1 | 7 | 0 | — | — | 5.4 | — |
| Cache-jul05 | 1 | 1 | 1 | — | — | — | 21.9 |
| Cache-jan06 | 1 | 7 | 3 | 7.8 | — | — | — |
| Liferay-feb05 | 2 | 3 | 1 | 26.3 | — | — | 7.0 |
| | 3 | 2 | 1 | 29.1 | — | — | 10.4 |
| | 5 | 2 | 0 | — | — | — | 29.6 |
| | | 7 | 0.25 | — | — | — | 11.9 |
| | 6 | 2 | 1.25 | 6.8 | — | 11.8 | 7.5 |
| | | 3 | 1.33 | 7.5 | — | — | — |
| Liferay-apr05 | 3 | 2 | 2 | 31.7 | — | 5.7 | 8.8 |
| Liferay-jun05 | 1 | 4 | 0.5 | — | — | — | 6.8 |
| | 2 | 4 | 0.17 | 14.7 | — | — | — |
| | 3 | 7 | 3 | — | — | 13.4 | — |
| Spring-dec03 | 3 | 1 | 0.33 | 31.4 | — | 7.9 | 19.4 |
| | 5 | 9 | 0.6 | 11.2 | — | — | — |
| | 6 | 6 | 1.5 | — | — | 7.9 | — |
| | 7 | 3 | 1 | — | — | 7.0 | — |
| | | 6 | 2 | 5.8 | — | — | — |
| | 8 | 2 | 1.67 | 28.1 | — | 8.7 | — |
| | | 6 | 0.67 | — | — | 8.7 | — |
| Spring-dec04 | 4 | 3 | 0.2 | 32.9 | — | 7.9 | 15.1 |
| | 6 | 1 | 0.17 | 38.9 | — | 5.6 | 23.4 |
| | | 3 | 0.25 | — | — | 6.0 | — |
| | 10 | 4 | 3 | 39.3 | — | — | — |
| | 12 | 2 | 1 | 35.3 | — | 12.3 | 7.5 |
| | 14 | 2 | 0.5 | 38.1 | — | — | 5.6 |
| | 15 | 1 | 0.17 | 38.9 | — | — | 24.2 |
| | 17 | 1 | 0 | 32.5 | — | — | 27.0 |
| Spring-dec05 | 2 | 6 | 3 | 5.0 | — | — | — |
| | | 7 | 3 | — | — | 15.1 | — |
| | 8 | 7 | 0.17 | — | — | — | 6.0 |
| XDoclet-oct01 | 4 | 5 | 2 | — | — | 5.5 | — |
| XDoclet-apr02 | 1 | 2 | 1.25 | 40.0 | — | 20.4 | 14.1 |
| | 5 | 3 | 0 | — | 5.8 | — | 10.2 |
| | | 12 | 0.2 | — | 5.8 | — | — |

Table 22: Percentage of Classes with or without Bug fixes based on Threshold Values of Decision Rules concerning Feature *bugfixVersusRefactoringPeak-Month*

In general, the number of bug fixes in the target period decreases, if the number of refactorings increases and the number of overall changes decreases.

In general, the number of bug fixes in the target period decreases, if the number of refactorings increases and the number of bug fixes decreases.

Most of the time, the number of bug fixes in the target period decreases, if the number of bug fix lines changed decreases, and the number of refactoring lines changed increases.

Most of the time, the number of bug fixes in the target period decreases, if the number of bug fix lines changed per transaction decreases, and the number of refactoring lines changed per transaction increases.

Mostly, the number of bug fixes in the target period decreases, if the bug fix peak month is before the refactoring peak month, or if there is a small distance between both peak months, when the refactoring peak month precedes the bug fix peak month.

The relation between the refactoring peak month and the peak month of overall changes seems to have no obvious influence on the occurrence of bug fixes.

### 5.3.4 Are certain subsets of refactoring features of major importance for the prediction models of each project?

We analyze the frequency of occurrence of refactoring features for the generated models to find out those, most significant for predicting future bug fixes. Table 25 and Table 26 enumerate for each refactoring feature the number of occurrences in decision rules that are part of the topmost three tree levels of the generated prediction models for each project under investigation. The abbreviations are the same as for models generated from author related features only (ArgoUML (A.), JBoss Cache (C.), Liferay Portal (L.), Spring Framework (S.), and XDoclet (X.)). To easily find out those features

| Project | Model No. | Tree-Level | Thres-hold | Bug fix=0 [%] | | Bug fix>=1 [%] | |
|---|---|---|---|---|---|---|---|
| | | | | <= | > | <= | > |
| ArgoUML-jun02 | 1 | 8 | 1 | — | — | 7.1 | — |
| | 2 | 10 | 0.2 | — | 19.1 | — | — |
| ArgoUML-jun03 | 5 | 2 | 0.2 | — | — | — | 32.4 |
| | 7 | 6 | 2 | 5.3 | — | — | — |
| | 9 | 3 | 0.25 | — | 26.2 | — | — |
| ArgoUML-jun04 | 1 | 9 | 0.75 | — | — | 5.4 | — |
| | 2 | 5 | 1.67 | 16.3 | — | — | — |
| Cache-jul05 | 1 | 2 | 0.25 | — | — | — | 21.9 |
| | | 4 | 0.2 | — | 30.5 | — | — |
| | 2 | 1 | 0 | 28.5 | 15.6 | — | 38.7 |
| | | 7 | 0.33 | — | 5.5 | — | — |
| | 3 | 1 | 0 | 14.5 | 19.9 | — | 25.8 |
| | | 5 | 1.33 | — | — | 8.6 | — |
| Cache-oct05 | 1 | 4 | 0.25 | 10.8 | 7.6 | — | — |
| | | 5 | 0.75 | — | 7.6 | 8.4 | 8.4 |
| | | 6 | 0.4 | — | — | — | 8.4 |
| Cache-jan06 | 1 | 9 | 0.25 | — | 7.8 | — | — |
| | | 9 | 0.25 | — | — | 8.8 | 5.8 |
| | 2 | 6 | 0 | 7.8 | — | — | 15.7 |
| Liferay-feb05 | 1 | 3 | 0.2 | — | — | — | 37.4 |
| Liferay-apr05 | 1 | 5 | 0.4 | — | — | 11.8 | — |
| | 5 | 6 | 0.2 | — | — | 9.3 | 8.4 |
| | 6 | 2 | 0.67 | 5.9 | 31.8 | 19.1 | — |
| Liferay-jun05 | 1 | 6 | 0.17 | — | 5.0 | — | — |
| | 3 | 1 | 0.17 | — | 19.8 | 10.3 | 20.9 |
| | 2 | 2 | 0.17 | — | 14.7 | — | — |
| | | 5 | 0.33 | — | 14.7 | — | — |
| Spring-dec03 | 2 | 6 | 0.4 | — | — | 7.4 | 8.3 |
| | 4 | 7 | 0.25 | — | 8.7 | — | — |
| | 5 | 4 | 0.25 | — | 17.8 | 7.9 | 7.0 |
| | | 5 | 0.17 | — | — | — | 7.9 |
| | 6 | 7 | 0.4 | 8.7 | — | — | — |
| | 9 | 6 | 0.67 | 15.7 | 10.3 | — | — |
| | 10 | 9 | 0.4 | 19.4 | — | — | — |
| | 11 | 3 | 0 | — | — | — | 12.8 |
| | | 4 | 0.2 | — | 22.7 | — | 5.8 |
| | | 6 | 0.33 | 13.6 | 9.1 | — | — |

Table 23: Percentage of Classes with or without Bug fixes based on Threshold Values of Decision Rules concerning Feature *refactoringVersusChangePeak-Month — PART 1*

| Project | Model No. | Tree-Level | Thres-hold | Bug fix=0 [%] | | Bug fix>=1 [%] | |
|---|---|---|---|---|---|---|---|
| | | | | <= | > | <= | > |
| Spring-dec04 | 1 | 4 | 0.83 | 21.0 | 17.9 | — | — |
| | | 5 | 0.33 | 21.0 | — | — | — |
| | 3 | 4 | 0.83 | — | 27.4 | — | — |
| | 4 | 2 | 1.25 | 32.9 | — | — | 23.0 |
| | | 5 | 0.75 | 21.8 | 0.11 | — | — |
| | 5 | 2 | 1.25 | — | — | 15.9 | — |
| | | 6 | 0.75 | 10.7 | — | — | — |
| | | 7 | 0.33 | 10.7 | — | — | — |
| | 6 | 2 | 0.83 | — | — | 17.5 | 6.0 |
| | | 3 | 0.33 | 38.9 | — | 5.6 | — |
| | 8 | 3 | 0.25 | — | — | — | 15.1 |
| | 9 | 3 | 0.25 | 35.7 | — | — | — |
| | 10 | 2 | 0.33 | 39.3 | — | — | 19.8 |
| | 11 | 5 | 0.33 | 5.6 | — | — | — |
| | 12 | 9 | 0.17 | — | — | — | 5.6 |
| | 14 | 6 | 0.83 | 5.2 | 6.0 | — | — |
| | 15 | 3 | 0.83 | — | — | 15.5 | — |
| Spring-dec05 | 1 | 2 | 0 | 10.1 | 7.6 | — | 19.0 |
| | 2 | 2 | 0.67 | — | — | — | 6.6 |
| | 3 | 7 | 0.25 | — | 15.1 | — | — |
| | 4 | 6 | 0.2 | — | 37.2 | — | — |
| | 5 | 10 | 0.2 | — | 5.6 | — | — |
| XDoclet-oct01 | 1 | 2 | 0.2 | — | 30.9 | — | — |
| | | 5 | 0.2 | — | 14.5 | — | — |
| | 4 | 2 | 1.25 | — | — | 30.0 | — |
| | | 4 | 0.6 | — | — | 5.5 | — |
| XDoclet-apr02 | 1 | 7 | 0 | — | — | — | 6.3 |
| | 2 | 5 | 0 | — | — | — | 12.1 |
| | 4 | 5 | 0.33 | — | 18.9 | — | — |
| XDoclet-aug02 | 2 | 5 | 0.5 | — | — | 7.1 | — |
| | | 6 | 0.33 | — | 22.2 | — | — |
| | 3 | 3 | 0.33 | — | 41.9 | — | — |

Table 24: Percentage of Classes with or without Bug fixes based on Threshold Values of Decision Rules concerning Feature *refactoringVersusChangePeak-Month — PART 2*

that are of major importance for the model trees of each project, the values of the five most commonly used features are marked bold. For project JBoss Cache this happens eight times, since five features hold the same value.

The tree hierarchy in given models is of major importance to estimate the influence on the prediction. We rate decision rules higher, which occur on the first three tree levels, in contrast to features appearing later on.
In two tables (Table 25 and Table 26) the occurrence of features in the top three tree levels is shown to gain insight in interconnections within each analyzed open source project. Let us now discuss the situation for each project separately:
ArgoUML uses most often, *refactoringChanges*, *bugfixVersusRefactoringLinesAdd*, *linesActivityRateRefactorings*, *refactoringActivityRate*, and *refactoringTLinesDelPerRefactoring*.
JBoss Cache focuses on *refactoringLinesChange*, *refactoringFrequencyBefore*, and *refactoringVersusChangePeakMonth*. Contrary to the other four projects, this time eight features are marked, since five features hold the same value.
Liferay Portal favors *refactoringCoChangedFiles*, *refactoringLinesChange*, *bugfixVersusRefactoringPeakMonth*, *linesActivityRateRefactorings*, and *refactoringLinesActivityRate*.
Spring Framework uses *refactoringCoChangedFiles*, *refactoringChanges*, *bugfixVersusRefactoringPeakMonth*, *linesActivityRateRefactorings*, and *refactoringVersusChangePeakMonth* most of the time.
XDoclet mostly applies features *refactoringLinesChange*, *refactoringChanges*, *linesActivityRateRefactorings*, *refactoringFrequencyBefore*, and *refactoringVersusChangePeakMonth*.
As one can see, all five projects show a different pattern, since each project runs through a different process of development. As a result of that fact certain groups of refactoring features are preferred by each project.
Refactoring features belonging to the group of complexity are of great importance for all projects. These are the features starting from *relRefactorings* until *refactoringChanges* at the bottom of Table 25.
Features part of the other groups appear to a lesser extent in the top three tree levels, whereby some time related features stick out a bit—namely features *refactoringVersusChangePeakMonth* and *bugfixVersusRefactoringPeakMonth* (Table 26).
Concluding, it is necessary to regard each project independently to find out those features that are most suitable to predict future bug fixes.

| Refactoring Feature | Project | | | | |
|---|---|---|---|---|---|
| | A. | C. | L. | S. | X. |
| refactoringLinesAdd | 4 | — | — | 8 | — |
| *refactoringLinesChange* | — | **3** | **5** | 2 | **5** |
| refactoringLinesDel | — | — | 1 | — | 1 |
| refactoringLinesAddPerRefactoring | 1 | — | — | 3 | 1 |
| refactoringLinesChangePerRefactoring | 2 | — | 1 | 1 | — |
| refactoringLinesDelPerRefactoring | 3 | — | — | 2 | 2 |
| bugfixVersusRefactoringLinesAdd | **5** | — | — | 4 | — |
| bugfixVersusRefactoringLinesChange | — | — | 2 | 1 | — |
| bugfixVersusRefactoringLinesDel | — | **2** | 1 | 3 | — |
| refactoringLinesType | — | 1 | 2 | — | 1 |
| refactoringAddingChanges | — | — | — | — | — |
| refactoringChangingChanges | — | — | — | — | — |
| refactoringChangesType | 4 | — | 4 | — | 1 |
| refactoringLargeModifications | — | — | — | — | — |
| refactoringSmallModifications | — | — | — | — | — |
| refactoringAuthorCount | 1 | — | — | — | — |
| refactoringAuthorSwitches | — | — | — | — | 1 |
| refactoringVSrevisionAuthorSwitches | 2 | — | — | 1 | — |
| relRefactorings | 2 | 1 | 2 | 4 | — |
| refactoringsBefore | 1 | — | 2 | 2 | 1 |
| *refactoringFrequencyBefore* | 4 | **3** | 2 | 5 | **6** |
| refactoringLinesActivityRate | 2 | — | **6** | — | — |
| refactoringActivityRate | **5** | 1 | — | — | — |
| *linesActivityRateRefactorings* | **10** | — | **9** | **13** | **4** |
| bugfixRefactorings | 2 | **2** | 2 | 5 | — |
| *refactoringChanges* | **5** | 1 | 3 | **23** | **7** |

Table 25: Number of Decision Rules of the Top 3 Tree Levels for Refactoring Features (investigated projects abbreviated) — *PART 1*

| Refactoring Feature | Project | | | | |
|---|---|---|---|---|---|
| | A. | C. | L. | S. | X. |
| refactoringCouplings | — | — | 1 | 1 | — |
| *refactoringCoChangedFiles* | 2 | 1 | **9** | **9** | — |
| refactoringLargeTransactions | — | — | — | — | — |
| refactoringSmallTransactions | 1 | — | — | — | — |
| refactoringTLinesAdd | 2 | 1 | 1 | 4 | 1 |
| refactoringTLinesChange | — | — | 1 | 8 | 2 |
| refactoringTLinesDel | 1 | 1 | — | 2 | 2 |
| refactoringTLinesType | 1 | **2** | 4 | 1 | 1 |
| refactoringTChangesType | 1 | 1 | — | 5 | 1 |
| refactoringTLinesAddPerRefactoring | 1 | — | 4 | 1 | — |
| refactoringTLinesChangePerRefactoring | 2 | — | 2 | 2 | 1 |
| refactoringTLinesDelPerRefactoring | **5** | — | 2 | 6 | 1 |
| TBugfixVersusTRefactoringLinesAdd | 1 | 1 | — | 3 | 2 |
| TBugfixVersusTRefactoringLinesChange | — | **2** | — | 1 | 1 |
| TBugfixVersusTRefactoringLinesDel | — | 1 | — | 3 | 2 |
| avgDaysBetweenRefactorings | — | — | — | 4 | 1 |
| avgDaysPerRefactoringLine | 1 | — | 4 | — | 1 |
| refactoringPeakMonth | 2 | **2** | 2 | 8 | 2 |
| peakMonthRefactorings | — | — | 1 | — | — |
| *refactoringVersusChangePeakMonth* | 2 | **3** | 4 | **11** | **3** |
| *bugfixVersusRefactoringPeakMonth* | — | 1 | **6** | **10** | 2 |

Table 26: Number of Decision Rules of the Top 3 Tree Levels for Refactoring Features (investigated projects abbreviated) — *PART 2*

Thus, we conclude:

| Certain subsets of refactoring features are of major importance for the prediction models of each project. |
|---|

| The composition of the subsets varies with respect to the investigated project. |
|---|

### 5.3.5 Is a common subset of refactoring features important for all investigated projects?

Again Table 25 and Table 26 are the basis of discussion. Features most often used in at least two different projects are emphasized:

- *refactoringLinesChange*
- *refactoringFrequencyBefore*
- *linesActivityRateRefactorings*
- *refactoringChanges*
- *refactoringCoChangedFiles*
- *refactoringVersusChangePeakMonth*
- *bugfixVersusRefactoringPeakMonth*

Especially, *refactoringLinesChange*, *refactoringChanges*, *linesActivityRateRefactorings*, and *refactoringVersusChangePeakMonth* show coherences, which reach beyond a single project. These refactoring features are used in at least three projects—even four in case of *linesActivityRateRefactorings*—in the topmost three tree levels.
We can say that there is no subset of features, important for all investigated projects. Nevertheless, some projects have several, most frequently used refactoring features in common. For example, projects ArgoUML, Spring Framework, and XDoclet frequently apply features *linesActivityRateRefactorings*, and *refactoringChanges* in their prediction models. JBoss Cache and

XDoclet commonly use *refactoringLinesChange*, *refactoringFrequencyBefore*, and *refactoringVersusChangePeakMonth* in the topmost three tree levels. Liferay Portal and Spring Framework both utilize *linesActivityRateRefactorings*, *refactoringCoChangedFiles*, and *bugfixVersusRefactoringPeakMonth*. Liferay Portal and XDoclet most often use *refactoringLinesChange* and *linesActivityRateRefactorings*. Spring Framework and XDoclet have *linesActivityRateRefactorings*, *refactoringChanges*, and *refactoringVersusChangePeakMonth* in common. On the basis of these results we can say that certain coherences, concerning the assembly of tree models exist in selected combinations of projects.

Thus, we conclude:

| In general, there is no subset of refactoring features that is important for all projects under investigation. |
|---|

| Some variable subsets of features are important for certain combinations of projects. |
|---|

### 5.3.6 Is it essential that the number of authors and author switches, and the number of people switches remains low to reduce error-proneness?

Within this section we discuss the behavior of three features: *authorCount*, *authorSwitches*, and *peopleSwitches*. Features *authorCount* and *authorSwitches* are related to the current versioning system (e.g. CVS) and *peopleSwitches* refers to the bugtracking system (e.g. JIRA).

**authorCount** Table 27 summarizes the results for each investigated project. A threshold value of 1 means that there is one author per change.
Let us take a closer look at Table 27 to better understand the presented information. The first line analyzes characteristic and frequency of decision rules that use feature *authorCount*, in *section models* generated from data of open source project ArgoUML. Three decisions assign classes that hold a value equal or below a threshold range from 0.43—0.58 for feature *authorCount*—boundaries included—, to bin "No bugs", that is no bug fixes happen in the

| Project | Bug fix=0 | | Bug fix>=1 | |
|---|---|---|---|---|
| | <= | > | <= | > |
| Argouml | 0.43-0.58 (3) | 0.18-0.83 (12) | 0.43-0.83 (8) | 0.45-0.83 (4) |
| Cache | 0.29-0.75 (3) | 0.22-0.75 (8) | 0.22-0.75 (4) | 0.22-0.29 (4) |
| Liferay | 0.33-0.75 (11) | 0.15-0.75 (28) | 0.19-0.75 (33) | 0.17-0.75 (32) |
| Spring | 0.21-0.75 (11) | 0.23-0.80 (41) | 0.23-0.80 (43) | 0.10-0.75 (29) |
| XDoclet | 0.29-0.83 (4) | 0.29-0.83 (12) | 0.20-0.83 (6) | 0.20-0.83 (9) |

Table 27: Range of Threshold Values (Number of Decisions) for Feature *authorCount* to predict Bug fixes

target period. Twelve decisions attach classes that hold a value above a range of threshold values from 0.18–0.83 to bin "No bugs". Finally, eight decisions with values for *authorCount* equal or below a threshold range of 0.43–0.83 and four decisions with values above 0.45–0.83 attach classes that fulfill these conditions to bin "One or more bugs", one or more bug fixes occur in the corresponding target period of these instances.
For projects ArgoUML (72.7%), Liferay (75.0%), and Spring (79.6%) a multiple of instances shows bug fixes, if the threshold value is below 0.2-0.8. Cache (57.1%) and XDoclet (60.0%) show the same trend alleviated, too. All projects with only one exception have more instances without bug fixes, if the threshold value is above 0.2-0.8. In that situation ArgoUML (75.0%), Cache (66.7%), Spring (58.6%), and XDoclet (57.1%) show no bug fixes in the majority of cases. Liferay (46.7%) shows an roughly equal ratio for instances with or without bug fixes.

Although the given ranges in reference to both situations— bug fix or no bug fix—are overlapping the trend is obvious: In the case of falling below a certain value (0.2) the number of bug fixes increases and in the opposite situation, when a limit is exceeded (0.8) the number of bug fixes decreases. This behavior is inconvenient, since in general one would guess that the more authors work on a software, the more bug fixes will appear in the future and vice versa.
An explanation for that characteristic could be that a minimum level of authors per change is required to permit a good functioning software development. A project can take advantage of a larger number of authors, since everybody supports the development process with his knowledge and

| Project | Bug fix=0 | | Bug fix>=1 | |
|---|---|---|---|---|
| | <= | > | <= | > |
| Argouml | 0.50-1.38 (7) | 0.50-1.38 (12) | 0.50-1.25 (12) | 0.50-1.20 (8) |
| Cache | — (—) | 0.75 (2) | 0.75-1.33 (2) | 0.67 (1) |
| Liferay | — (—) | 0.50-0.67 (8) | 0.50-0.67 (5) | 0.33-0.75 (14) |
| Spring | 0.50-1.50 (17) | 0.50-1.33 (17) | 0.75-1.50 (10) | 0.50-1.50 (15) |
| XDoclet | 0.67-2.40 (3) | 0.50-0.67 (4) | 0.67-2.40 (5) | 0.67-2.40 (7) |

Table 28: Range of Threshold Values (Number of Decisions) for Feature *authorSwitches* to predict Bug fixes

valuable ideas to solve problems.
In that, no limit value is above 1, it is hard to say, whether there is an upper limit that once overstepped leads to more bug fixes. But we assume an upper bound and consequently an ideal range for feature authorCount to reduce the probability of bug fixes in the future.

**authorSwitches** Table 28 shows the results for each analyzed project. A threshold value of 1 means that there is one author switch per author that means a specific author works on a software entity without interruption.
For this feature all projects show very different results so it is necessary to discuss each project separately.
ArgoUML uses a threshold range of about 0.50-1.4 for all four categories and shows a slight trend to more bugs (63.2%), if there are less author switches and backwards (60.0%). Projects Cache and Liferay lack of instances for category "<=" for "Bug fix=0". We found out that project Cache (66.7%) shows a trend to no bug fixes in the case of more author switches. In contrast, project Liferay (63.6%) shows a tendency to more bug fixes in the case of more author switches. Both projects, the Spring framework and XDoclet display no obvious drift to a definite classification.
Our expectations were that more author switches cause more bug fixes and otherwise less author switches lead to no bug fixes. In this way the results were disappointing to us, since no clear statement can be made with respect to all projects under investigation.

**peopleSwitches** Table 29 lists the results for each project under investigation. A threshold value of 1 means that there is one issue assignee per

| Project | Bug fix=0 | | Bug fix>=1 | |
|---|---|---|---|---|
| | <= | > | <= | > |
| Argouml | 0-8 (16) | 0-3 (7) | 0-4 (6) | 0-5 (6) |
| Cache | 0-1 (6) | — (—) | — (—) | 0 (3) |
| Liferay | 0 (2) | — (—) | 0 (2) | — (—) |
| Spring | 0-2 (21) | (-1)-1 (8) | 0-2 (12) | 0-1 (16) |
| XDoclet | 0-6 (14) | 0-4 (6) | 1-6 (6) | 0-6 (11) |

Table 29: Range of Threshold Values (Number of Decisions) for Feature *peopleSwitches* to predict Bug fixes

author switch.
Project ArgoUML shows a trend to no bug fixes (72.7%), if values below a certain threshold value occur. This does not apply to the appearance of bug fixes, since about the same number of decisions for both situations leads to inconsistent results. Cache and Liferay use only few decisions based on feature peopleSwitches. Cache exhibits a tendency that values below a threshold value lead to no bug fixes and values above a threshold value induce one or more bug fixes. By contrast, Liferay does not help us to draw a conclusion, since the results are controversial. Finally, projects Spring and XDoclet correspond to the same pattern as for JBoss Cache. For projects Spring (63.6%), and XDoclet (70.0%) most of the instances show no bug fixes, if the threshold value is below a given limit. In the contrary situation instances mostly show bug fixes for projects Spring (66.7%), and XDoclet (64.7%).
Generally, we can say that most of the investigated projects follow the scheme that more people switches lead to more bug fixes and the other way round.

Thus, we conclude:

| |
|---|
| The number of authors negatively correlates to the number of bug fixes in the target period. |

| |
|---|
| The number of author switches can not definitely predict the occurrence of bugs in the future. |

| Project | Bug fix=0 | | Bug fix>=1 | |
|---|---|---|---|---|
| | <= | > | <= | > |
| Argouml | 0-0.75 (6) | 0.67 (1) | 0.75 (2) | — (—) |
| Cache | 0.40-0.50 (2) | — (—) | — (—) | — (—) |
| Liferay | 0.25-0.50 (3) | 0.50 (1) | 0.33-0.67 (18) | 0.33-0.67 (14) |
| Spring | 0-0.50 (3) | — (—) | 0.50 (2) | — (—) |
| XDoclet | 0.25-0.50 (2) | — (—) | 0.67 (1) | 0.50-0.80 (2) |

Table 30: Range of Threshold Values (Number of Decisions) for Feature *bugfixAuthorCount* to predict Bug fixes

| The number of people switches positively correlates to the number of bug fixes in the target period. |
|---|

### 5.3.7 Is it essential that the number of bug fix authors and bug fix author switches remains low to reduce error-proneness?

**bugfixAuthorCount** Table 30 lists the results for each project under investigation. A threshold value of 1 means that there is one bug fix author per bug fix.
As one can see there are many blank fields in the table, anyway there is slight trend to produce no bugs, if a certain threshold value is under-run. This applies to projects ArgoUML (75.0%), Cache (2 decisions), Spring (60.0%), and XDoclet (66.7%). In contrast, project Liferay shows a great many of bug fixes for instances that hold values above (93.3%) and below (85.7%) certain threshold values.
We expected that the more bug fix authors, the more bug fixes will occur in the target period. This assumption can not be underlined clearly, because of variability of results depending on the analyzed project.

**bugfixAuthorSwitches** Table 31 lists the results for each project under investigation. A threshold value of 1 means that there is one bug fix author switch per bug fix author that means a specific bug fix author works on a bug fix for a software entity without interruption.
In the majority of cases ArgoUML (77.8%), Cache (60.0%), Liferay (75.0%), Spring (93.3%), and XDoclet(83.3%) show one or more bug fixes for instances

| Project | Bug fix=0 | | Bug fix>=1 | |
|---|---|---|---|---|
| | <= | > | <= | > |
| Argouml | 0-0.50 (11) | 0-0.50 (2) | 0-0.50 (7) | 0-0.50 (7) |
| Cache | 0 (5) | 0-0.75 (4) | 0 (3) | 0 (6) |
| Liferay | 0.50 (2) | 0-0.50 (2) | 0-0.50 (4) | 0-0.50 (6) |
| Spring | 0-0.67 (16) | 0 (1) | 0-0.50 (12) | 0-0.50 (14) |
| XDoclet | 0-0.67 (4) | 0 (1) | 0-0.75 (4) | 0-0.67 (5) |

Table 31: Range of Threshold Values (Number of Decisions) for Feature *bugfixAuthorSwitches* to predict Bug fixes

that hold values above a given limit. In accordance with these results the projects ArgoUML (61.1%), Cache (62.5%), and Spring(57.1%) exhibit an alleviated trend to no bug fixes, if a certain threshold value is under-run. There are some exceptions concerning the category "<=" with "Bug fix>= 1" for projects Liferay (33.3%), and XDoclet (50.0%).
Most of the time, the number of bug fix author switches positively correlates to the occurrence of bug fixes in the target period.

Thus, we conclude:

| The number of bug fix authors can not definitely predict the occurrence of bugs in the future. |
|---|

| The number of bug fix author switches positively correlates to the number of bug fixes in the target period. |
|---|

### 5.3.8 Is it essential that the number of refactoring authors and refactoring author switches remains low to reduce error-proneness?

**refactoringAuthorCount** Table 32 lists the results for each project under investigation. A threshold value of 1 means that there is one refactoring author per refactoring.
All projects show very different results, so it is hard to find out a certain trend to predict future bug fixes. We found out that ArgoUML (75.0%), Cache

| Project | Bug fix=0 | | Bug fix>=1 | |
|---|---|---|---|---|
| | <= | > | <= | > |
| Argouml | 0.67-0.75 (3) | 0.67-0.75 (5) | 0.75 (1) | 0.33-0.75 (3) |
| Cache | 0.50 (1) | 0.50 (2) | 0.50 (1) | 0.50 (1) |
| Liferay | 0.14-0.50 (4) | 0.50 (3) | 0.33-0.50 (4) | 0.14-0.50 (8) |
| Spring | 0.25-0.50 (3) | 0.50-0.67 (2) | 0.50-0.67 (7) | 0.25 (2) |
| XDoclet | 0.33-0.80 (6) | 0.33-0.44 (2) | 0.33-0.75 (6) | 0.50-0.75 (6) |

Table 32: Range of Threshold Values (Number of Decisions) for Feature *refactoringAuthorCount* to predict Bug fixes

| Project | Bug fix=0 | | Bug fix>=1 | |
|---|---|---|---|---|
| | <= | > | <= | > |
| Argouml | 0-0.67 (5) | 0-0.67 (4) | 0.67-1 (5) | 0 (5) |
| Cache | 0-0.50 (2) | 0-0.50 (2) | 0.67 (1) | 0 (1) |
| Liferay | 0-0.50 (4) | 0.50 (3) | 0.33-0.50 (5) | 0-0.50 (10) |
| Spring | 0-0.67 (13) | 0.50-0.67 (9) | 0-0.67 (12) | 0-0.50 (10) |
| XDoclet | 0-1 (4) | 0-0.67 (5) | 0-0.50 (3) | 0 (1) |

Table 33: Range of Threshold Values (Number of Decisions) for Feature *refactoringAuthorSwitches* to predict Bug fixes

(50.0%), Liferay (50.0%), Spring (30.0%), and XDoclet(50.0%) show no bug fixes for instances that hold values below a given limit. Additionally, we detected that projects ArgoUML (37.5%), Cache (33.3%), Liferay (72.7%), Spring (50.0%), and XDoclet(75.0%) show one or more bug fixes for instances that hold values above a given limit. By reason of these varying results, we can not answer the question how the number of refactoring authors influences software defect occurrence.

**refactoringAuthorSwitches** Table 33 lists the results for each project under investigation. A threshold value of 1 means that there is one refactoring author switch per refactoring author that means a specific refactoring author works on a refactoring for a software entity without interruption.
We found out that ArgoUML (50.0%), Cache (66.7%), Liferay (44.4%), Spring (52.0%), and XDoclet(57.1%) show no bug fixes for instances that hold values below a given limit. Additionally, we detected that projects ArgoUML (55.6%), Cache (33.3%), Liferay (76.9%), Spring (52.6%), and

| Project | Bug fix=0 | | Bug fix>=1 | |
|---|---|---|---|---|
| | <= | > | <= | > |
| Argouml | 0.75 (1) | 0.67-0.75 (4) | 0.75-1 (3) | 0.50-0.75 (2) |
| Cache | 0.67 (1) | 0.50-0.67 (2) | 0.67 (1) | 0.50-0.67 (2) |
| Liferay | — (—) | — (—) | — (—) | — (—) |
| Spring | 0.50 (1) | 0.50 (1) | 0.50-1.50 (4) | 0.50 (1) |
| XDoclet | — (—) | 0.67 (1) | 0.67 (1) | — (—) |

Table 34: Range of Threshold Values (Number of Decisions) for Feature *refactoringVSrevisionAuthorSwitches* to predict Bug fixes

XDoclet(16.7%) show one or more bug fixes for instances that hold values above a given limit. Again, all analyzed projects show diverging results, so we can not answer the question how the number of refactoring author switches influences software defect appearance.

Thus, we conclude:

| It is not possible for us to find out an underlying trend, how the number of refactoring authors and refactoring author switches influence the occurrence of future bug fixes. |
|---|

### 5.3.9 Is it essential that the same author, who made the last revision refactors the code to reduce error-proneness?

**refactoringVSrevisionAuthorSwitches** Table 34 lists the results for each project under investigation. A threshold value of 1 means that there is one switch from revision author to refactoring author per refactoring author, e.g. a specific refactoring author works on a refactoring for a software entity after another author has made a change—a revision.
We detected that ArgoUML (75.0%), Cache (50.0%), Spring (80.0%), and XDoclet(1 decision) show one or more bug fixes for instances that hold values below a given limit. Additionally, we found out that projects ArgoUML (66.7%), Cache (50.0%), Spring (50.0%), and XDoclet(1 decision) show no bug fixes for instances that hold values above a given limit. Project Liferay has no entries for this feature, concerning author related *section models*.

There is no clear trend, since the analyzed projects show a diverging behavior.

Thus, we conclude:

| It is not possible for us to say, whether it is necessary that the same author, who made the last revision refactors the code, with respect to software defect appearance. |
|---|

# 6 Summary and Conclusion

In this research work the interrelationship of evolution activities such as refactoring is investigated to predict software defect appearance in the near future. Our case study is based on five open source projects belonging to different domains to support generality. Our work extends the knowledge in the field of software quality estimation, since as far as we know, this is the first try to quantify the interrelationship between refactoring and other evolution activities to evaluate the impact on software defect prediction. In this context, information that brings about a decision where and when to apply refactoring can help to calculate expenses. Ultimately, the main goal is to support software engineers and project managers in timely choosing the best refactorings to improve software quality, especially in relation to understandability and maintainability of source code.

We use versioning and issue tracking systems 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 size and complexity measures, relational aspects, time constraints, or team related aspects. At this, size measures are extended by relational aspects using information of co-change coupling of software entities to better catch coherences. Complexity measures are of great importance, since software continuously becomes more complex, so that changes are more difficult to add, without violating existing contracts. In relation to time constraints, especially peak month related facets have proven important to estimate error proneness. Author related measures regard, besides an overall view, close-up views of the influence of bug fix and refactoring authors. Every time an author has to work on software parts that another team member edited earlier, problems affecting source code comprehension might occur. On this account, expert knowledge and project-internal communication are cornerstones of successful software development.

We found out that refactoring related features, as well as non-refactoring related features produce high quality prediction models. Moreover, the quality of prediction models is generally kept constant and sometimes slightly improved by combining features of both categories. Refactoring should be associated with other software activities in a certain way to reduce the appearance of software defects: In general, the number of bug fixes in the target period decreases, if the number of refactorings increases, the number of bug fixes decreases, and the number of overall changes decreases within

the corresponding feature period. The same behavior can be observed, if the number of bug fix lines changed per transaction decreases, and the number of refactoring lines changed per transaction increases, and if refactoring peak month and bug fix peak month are related in a certain way. We found out that certain subsets of refactoring features are of major importance for the prediction models of each project, even though the composition of the subsets varies with respect to the investigated project. In contrast, we could not determine a subset of refactoring features that is important for all projects under investigation. The analysis of author related features showed variable results, even so certain interrelationships could be revealed: We found out that the number of authors negatively correlates to the number of bug fixes in the target period. In contrast, the number of people switches regarding the bugtracking system and the number of bug fix author switches positively correlate to the number of bug fixes in the target period.

In our future work we will concentrate on the following topics:

- **Integration of new Refactoring Features.** In this thesis we implemented 47 refactoring related features to model certain aspects of relevant information areas. This set of features could be further extended to cover more characteristics of refactoring. Especially, features relating refactoring attributes to software defects of diverse severity could provide a deeper insight into that complex topic.

- **Variation of Time Frames.** In this work we base our analysis on time frames for feature and target period for six months each. It would be interesting to what extent a variation of both time frames could influence the prediction results.

- **Improvement of SQL Queries.** We plan to further improve our SQL queries to reduce the false positive and false negative rates with respect to refactoring, as well as bug fix detection. Beyond it, we intend to use additional constraints, besides lines of code measures to restrict the total set of investigated instances.

- **Automation of Model Analysis.** Many processing steps of our approach, like the import of versioning and issue data, and the computation of commit transactions are already automated. This also applies to connecting issues and revisions, relating accounts of versioning and

issue data, and finally calculating data mining features. It would be desirable to automatize the analysis of prediction models generated by the datamining tool Weka. The most important features, e.g. the three topmost features in a model tree, or decision rules that concern a minimum of instances would be interesting facts for developers, as well as project managers. In the long run, a tool that outputs classes, which are in need for refactoring with respect to software defect reduction, is the aim.

# References

[1] Deepak Advani, Youssef Hassoun, and Steve Counsell. Refactoring trends across n versions of n java open source systems: an empirical study. Technical report, University of London, 2005.

[2] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An automatic approach to identify class evolution discontinuities. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 31–40, Kyoto, Japan, 2004.

[3] IEEE Standards Association. http://standards.ieee.org/.

[4] Victor R. Basili, Lionel C. Briand, and *Walcélio* Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, October 1996.

[5] Lionel C. Briand, Jürgen Wüst, Stefan V. Ikonomovski, and Hakim Lounis. Investigating quality factors in object-oriented designs: An industrial case study. In *Proceedings of the International Conference on Software Engineering*, 1999.

[6] Fernando Brito e Abreu and *Walcélio* Melo. Evaluating the impact of object-oriented design on software quality. In *Proceedings of the International Software Metrics Symposium*, pages 90–99, Berlin, Germany, March 1996.

[7] Andrea Capiluppi, Maurizio Morisio, and Patricia Lago. Evolution of understandability in oss projects. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 58–66, Tampere, Finland, March 2004.

[8] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications*, pages 197–211, Phoenix, Arizona, USA, 1991.

[9] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[10] Marco *D'ambros* and Michele Lanza. Software bugs and evolution: A visual approach to uncover their relationship. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, March 2006.

[11] MySQL: Open Source Database. http://mysql.com.

[12] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 166–177, 2000.

[13] Serge Demeyer and Tom Mens. Evolution metrics. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 83–86, Vienna, Austria, 2001. Session 4A: Principles.

[14] Giovanni Denaro and Mauro Pezzè. An empirical evaluation of fault-proneness models. In *Proceedings of the International Conference on Software Engineering*, pages 241–251, 2002.

[15] XDoclet: Open Source Code Generation Engine. http://xdoclet.net.

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

[17] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32, Amsterdam, Netherlands, September 2003. IEEE Computer Society Press.

[18] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *Proceedings of the International Conference on Program Comprehension*, page to appear, Athen, Greece, June 2006. IEEE Computer Society Press.

[19] Beat Fluri, Harald C. Gall, and Martin Pinzger. Fine-grained analysis of change couplings. In *SCAM '05: Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 66–74, Washington, DC, USA, 2005.

[20] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, June 1999.

[21] Spring Framework: Open Source Java/JEE Application Framework. http://www.springframework.org.

[22] Harald Gall, Mehdi Jazayeri, and Jacek Ratzinger (former Krajewski). CVS release history data for detecting logical couplings. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 13–23. IEEE Computer Society Press, September 2003.

[23] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, July 2000.

[24] L. Hatton. Re-examining the fault density-component size connection. *IEEE Software*, 14(2):89–97, March/April 1997.

[25] JBoss Cache: Open Source Performance Improvement. http://de.jboss.com/products/jbosscache.

[26] Weka 3: Data Mining Software in Java. http://www.cs.waikato.ac.nz/ml/weka.

[27] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of the International Conference on Software Maintenance*, pages 576–585, October 2002.

[28] Taghi M. Khoshgoftaar, Ruqun Shan, and Edward B. Allen. Using product, process, and execution metrics to predict fault-prone software modules with classification trees. In *Proceedings of the International Symposium on High Assurance Systems Engineering*, pages 301–310, Albuquerque, USA, 2000.

[29] Taghi M. Khoshgoftaar, Xiaojing Yuan, Edward B. Allen, Wendell D. Jones, and John P. Hudepohl. Uncertain classification of fault-prone software modules. *Empirical Software Engineering*, 7(4):297–318, December 2002.

[30] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Whitehead, Jr. Automatic identification of bug-introducing changes. In *Proceedings of the International Conference on Automated Software Engineering*, pages 81–90, Tokyo, Japan, September 2006.

[31] Sunghun Kim, Thomas Zimmermann, E. James Whitehead, Jr., and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the International Conference on Software Engineering*, pages 489–498, Minneapolis, Minnesota USA, May 2007.

[32] Patrick Knab, Martin Pinzger, and Abraham Bernstein. Predicting defect densities in source code files with decision tree learners. In *Proceedings of the International Workshop on Mining Software Repositories*, page to appear, Shanghai, China, May 2006. ACM Press.

[33] Manny Meir Lehman and Laszlo A. Belady. *Program Evolution - Process of Software Change*. Academic Press, London and New York, 1985.

[34] Manny Meir Lehman and Juan Fernandez Ramil. Software evolution and software evolution processes. *Annals of Software Engineering, special issue on Software Process-based Software Engineering*, 14:275–309, 2002.

[35] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126 – 139, 2004.

[36] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance*, page 120. IEEE Computer Society, 2000.

[37] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, April-June 2000.

[38] K.H. Moeller and D. Paulish. An empirical investigation of software fault distribution. In *Proceedings of the International Software Metrics Symposium*, pages 82–90, 1993.

[39] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the International Conference on Software Engineering*, pages 580–586, St. Louis, MO, USA, May 2005.

[40] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the International Conference on Software Engineering*, pages 284–292, St. Louis, MO, USA, May 2005.

[41] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the International Conference on Software Engineering*, pages 452–461, Shanghai, China, May 2006.

[42] Oscar Nierstrasz. Software evolution as the key to productivity. In *Proceedings Radical Innovations of Software and Systems Engineering in the Future*, Venice, Italy, October 2002.

[43] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[44] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 55–64, Rome, Italy, July 2002.

[45] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *Proceedings on the International Symposium on Software Testing and Analysis*, pages 86–96, Boston, Massachusetts, USA, July 2004.

[46] Liferay Portal: Open Source Portal Platform. http://www.liferay.com.

[47] Jacek Ratzinger, Michael Fischer, and Harald Gall. Evolens: Lens-view visualizations of evolution data. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 103–112, Lisbon, Portugal, September 2005.

[48] Jacek Ratzinger, Michael Fischer, and Harald Gall. Improving evolvability through refactoring. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 69–73, St. Louis, USA, May 2005.

[49] Jacek Ratzinger, Martin Pinzger, and Harald Gall. Eq-mine: Predicting short-term defects for software evolution. In *Proceedings of the Fundamental Approaches to Software Engineering*, pages 12–26, Braga, Portugal, March 2007. Springer.

[50] Jacek Ratzinger, Thomas Sigmund, Peter Vorburger, and Harald Gall. Mining software evolution to predict refactoring. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, page to appear, Madrid, Spain, September 2007.

[51] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 18–27, Rio de Janeiro, Brazil, September 2006.

[52] ArgoUML: Open Source UML Modeling Tool. http://argouml.tigris.org.

[53] Nikolaos Tsantalis, Alexander Chatzigeorgiou, and George Stephanides. Predicting the probability of change in object-oriented systems. *IEEE Transactions on Software Engineering*, 31(7):601–614, July 2005.

[54] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, USA, 2 edition, 2005.

[55] Annie T. T. Ying, Gail C. Murphy, Raymong Ng, and Mark Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, September 2004.

[56] Thomas Zimmermann and Peter Weißgerber. Preprocessing cvs data for fine-grained analysis. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburgh, Scotland, May 2004.

[57] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the International Conference on Software Engineering*, volume 00, pages 563–572, Edinburgh, Scotland, UK, May 2004.

# Index