Mining Software Evolution to Predict Refactoring

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

Peter Vorburger, Harald Gall
University of Zurich
Department of Informatics
CH-8050 Zurich, Switzerland
{vorburger,gall}@ifi.unizh.ch

Abstract

Can we predict locations of future refactoring based on the development history? In an empirical study of open source projects we found that attributes of software evolution data can be used to predict the need for refactoring in the following two months of development. Information systems utilized in software projects provide a broad range of data for decision support. Versioning systems log each activity during the development, which we use to extract data mining features such as growth measures, relationships between classes, the number of authors working on a particular piece of code, etc. We use this information as input into classification algorithms to create prediction models for future refactoring activities. Different state-of-the-art classifiers are investigated such as decision trees, logistic model trees, propositional rule learners, and nearest neighbor algorithms. With both high precision and high recall we can assess the refactoring proneness of object-oriented systems. Although we investigate different domains, we discovered critical factors within the development life cycle leading to refactoring, which are common among all studied projects.

1 Introduction

Refactoring is a state-of-the-art practice in software development to improve the design of existing software systems without changing the external behavior. Developers often use this technique to prepare object-oriented systems for further improvements and extensions of functionality. The identification of hot-spots where refactorings will take place improves the effectiveness of engineers in focusing on the relevant classes that will undergo changes in future [11]. For project managers it is interesting to know which locations are likely to demand refactoring. Refactoring improves the understandability of the code, but on the other hand requires development time [6]. The prediction of future refactorings allows project managers a better coordination of software development and project management, a more accurate budgeting, and efficient manpower utilization. With limited time and resources project managers want to focus on the most important refactorings. The right time for a general refactoring can be better determined with our approach.

A source to determine required refactorings of software entities such as classes [1] can be their past development and maintenance history. An entity, which undergoes many changes in the past, bears a certain probability for refactoring. Such information can be gained from versioning systems such as CVS (Concurrent Versions System) and issue tracking systems such as Jira. Certainly, source code inspection can be used to reveal the demand of refactorings, too. However, this work is very labor intensive and must be carried out by specialists. As a result refactoring is expensive. Therefore, we describe an automation of this approach by utilizing machine learning.

In this work we screen evolution data from versioning systems of two open source projects, namely ArgoUML and the Spring framework, both open source frameworks where refactoring is recognized as important engineering activity for software evolution. They are developed in Java and consist of 5000 and 10000 classes each. Each class is usually placed in a separate file in Java, thus we use files equivalent to classes and focus on files for our analysis. Several preprocessing steps are taken to derive the key attributes from the data. We use these attributes to generate models via an open-source data mining tool called WEKA [20].

In the remainder of this paper, we first describe related work (Section 2) and propose our research question (Section 3) followed by the foundations of our prediction task (Section 4). Afterwards, we present the results of our case study (Section 5) and finish with conclusions and future work (Section 7).

[1] in the entire document classes are a synonym for Java classes

2 Related Work

A number of publications show the usefulness of refactoring for software development. Capiluppi et al. [3] found that understandability was increased by refactorings in several projects. Kataoka et al. [11] try to measure the maintainability enhancement effect of program refactorings based on coupling metrics. In a larger experiment the developer's subjective evaluations match the results of the author's approach to determine the effectiveness of refactorings. Najjar et al. [15] investigate refactoring of constructors, because constructors do not ideally communicate the developer's intention, and secondary produce duplicated code. The study investigated several Java systems and found that the code of two systems could be economized. A survey paper, which discusses extensively research in the field of software refactoring was written by Mens and Tourwé [14]. It discusses refactoring activities and numerous techniques. Demeyer et al. [5] validate several metrics for identifying if refactorings took place for a given file. In contrast, we preform a prediction task based on evolution data.

Antoniol et al. [2] analyze the evolution of object-oriented source code at the class level. The focus is on a limited number of refactoring events, which are identified based on a vector space model. The application of the proposed approach to an open source domain name service produced a list of class refactoring operations. Advani et al. [1] analyzed a range of open source systems, regarding whether a refactoring occurred and if so, which were the most common. They found that simple refactorings, like basic operations on fields and methods occurred more frequently, than more interweaved refactorings, such as those, requiring manipulation of the inheritance hierarchy.

A number of metrics have been suggested to evaluate diverse kinds of software engineering activities, in recent years. Chidamber and Kemerer's metrics [4] are often used to detect faulty classes. Tsantalis et al. [19] try to quantify the change probability of each class in future releases with the help of these metrics. Forward and backward logistic regression was performed with the result, that only some proposed measures such as class size where relevant in the analyzed case studies. Their work identifies classes that have a high probability of change and at the same time can effect a large number of classes. In contrast, in our work we want to predict the probability of future refactorings using evolution data instead of source code based metrics.

In a previous work [17] we investigated the design of a software system based on evolution data, extracted from source code management systems. This information is used to detect architectural shortcomings in the design of the analyzed software and how refactoring can support the evolvability of software systems.

3 Research Question

In this paper we describe our prediction of refactorings in software projects. In previous research activities several studies analyzed the predictability of quality measures such as error proneness, defect densities, and time to failure [9, 16, 12]. We create models for the prediction of another type of event within software engineering projects such as refactorings. Similar to defects, refactorings provide an insight into the necessary rework we have to expect in future development activities. Table 1 summarizes our hypotheses to guide our research.

4 Prediction Foundations

To predict refactorings we make use of data mining algorithms. The classical KDD (knowledge discovery in databases) process distinguishes between several stages: 1. Data understanding, 2. Pre-processing, 3. Application of machine learning algorithms, 4. Post-processing, 5. Analysis of the results. The remainder of this section is organized based on the KDD process, where the analysis will be discussed separately in Section 5.

4.1 Data Understanding: Descriptive Statistics

For our approach we use two different time periods from which the attributes are obtained:

- *Learning Period* is a time frame where attributes of software evolution are accumulated as input to the prediction. We take all modifications to source code during this time into account and compute for each file a condensed history.
- *Target Period* is the time frame immediately after the learning period, where we count the number of refactorings. These numbers are used to define our data mining targets such as refactoring proneness.

Table 2 describes the number of refactorings of the two independent software projects taken from the open source community grouped by learning and target periods. These two projects are known for their good engineering practices and use refactoring as state-of-the art in their development. Especially, ArgoUML exhibits a large number of refactorings within our target periods, which were chosen to cover two months. In these months we counted the number of refactorings done by the developers in these projects. Additionally, for the first target period we analyze two learning periods of different length to investigate hypothesis H2 in Section 5.2. We provide data from the Spring framework, which is a J2EE application server for Java. In both projects

	Hypotheses
H1	Evolution data is a good predictor of future refactoring.
H2	It is possible to predict refactoring on short time frames.
H3	We can accurately predict the number of future refactorings for each file.
H4	There is a common subset of features essential for predictions in different projects.

Table 1. Research hypotheses

Project	Learning Period	Target Period	Refactoring=0	Refactoring=1	Refactoring>=2	$\sum$ Files
ArgoUML	Oct.04 - Dec.04	Jan.05 - Feb.05	603	181	129	913
	Jul.04 - Dec.04	Jan.05 - Feb.05	603	181	129	913
Spring	Aug.05 - Oct.05	Nov.05 - Dec.05	750	110	35	895
	Mai.05 - Oct.05	Nov.05 - Dec.05	750	110	35	895

Table 2. Refactoring distribution for analyzed periods by project

we selected the target periods to cover months with many refactorings, as we aim for refactoring prediction. Table 2 summarizes the number of classes exhibiting a certain number of refactorings. Classes having no refactorings in the target period we call non-refactoring prone. The number of classes that were refactored once is much smaller than the number of classes with no refactoring (i.e. we are dealing with unbalanced class distributions). Even smaller is the number of classes with two or more refactorings.

4.2 Pre-processing: Evolution Data

For mining of software development projects we use versioning systems like CVS, which allow handling of different versions of files in cooperating teams. Developers check out certain classes of the system, edit, and commit the changes to the versioning system, which merges the modifications.

Versioning systems log every action to keep the history of a file, which provides the necessary information about the history of a software system. The underlying data for our mining approach is retrieved via standard command line tools, which create log-files about all modifications in the past. These log-files are parsed and stored in a relational database [21]. In the Java programming language files are (roughly) equal to classes, because usually one class is defined in one Java file.

4.2.1 Data Preparation.

The data model of the evolution database is built on information extracted from versioning systems in the following way. Regarding versioning systems, revisions of files are strongly related to modification reports. They provide historical data about files such as change dates, change size measured on the basis of lines, and the author of a change. Revisions are related to each other, as the code of one revision replaces the code of the previous one. Releases are defined as collections of revisions of all files maintained by the particular versioning system. When a developer checks in changes to several files at once, the versioning system only stores the dates of the new revisions, but does not maintain the transaction information. As a result we have to reconstruct transactions when files were changed together in a post-processing phase.

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 messages within a short time-frame—typically a few minutes. To capture the entire transaction, possibly lasting several minutes we use a dynamic time adaptation approach. Each transaction is initially set to last for 60 seconds. Every change of a file with equal author and commit message within the transaction window is added to the transaction. The window is expanded to last 60 seconds after the last detected change event related to this transaction. Transactions are later on used for the evaluation of change couplings between software entities. We refer to change coupling as: *Two entities (e.g. files) are coupled, if a modification of 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 couplings and $|C|$ is the intensity of coupling.

4.2.2 Feature Generation.

In data mining the input attributes used by the algorithms are called *features*. Our evolution features are gathered on file basis, whereby data from all revisions of a file within a predefined time period is summarized (learning and target period). To build a prediction model we created features to represent several important information areas [18]. Prediction models have to regard different aspects of software development like the complexity of the designed solution, programmer/analyst skill, process used for development, etc. [7]. As a result we present the following summary of features for each file, containing changes within the inspection period. We group our features into different categories:

Size.
This category contains size measures such as lines of code from an evolution perspective: *linesAdded*, *linesModified*, or *linesDeleted* relative to the total *LOC* (lines of code) of a file. This represents a certain aspect of clean-up mentality, where developers remove code they regard as unnecessary. Other features of this category are *linesType*, which defines whether there are more *linesAdded* or *linesModified*. Additionally, we regard *largeChanges* as double of the *LOC* of the average change size and *smallChanges* as half of the average *LOC* of a specific file. We expect this value to be an important feature for data mining, as other studies have found that small modules are more defect-prone than large ones. [10]

Team.
The number of authors of files influences the way software is developed. We expect that the more authors are working on the changes the higher the probability of rework and mistakes. We define a feature for the *authorCount* relative to the *changeCount*. Further, the interrelation of people work is interesting. We investigate work rotation between the authors involved in the changes of each file as the feature *authorSwitches*. The number of people assigned to an issue and the authors contributing to the implementation of this issue is another feature we use (*authorMatch*) for our prediction models.

Work habits.
To get an estimation for the work habits of the developers, we inspect the number of *addingChanges*, *modifyingChanges*, and *deletingChanges* per author and per file. This information provides input to the defect prediction of files.

Complexity of existing solution.
According to the laws of software evolution [13], software continuously becomes more and more complex. Changes are more difficult to add as the software is more difficult to understand and the contracts between existing parts have to retain. As a result we investigate the *changeCount* in relation to the number of changes during the entire history of each file. The *changeActivityRate* is defined as the number of changes relative the lifetime of the file measured in months. The *linesActivityRate* describes the number of lines of code relative to the age of the file in months.

We approximate the quality of the existing solution by the *bugfixCountBefore*, which are the number of bug fixes before our prediction period relative to the general number of changes. The *bugfixCount* is computed by enumerating all changes to source code, where an bug tracking issue is attached with type "fix". We expect that the higher the fix rate is before the inspection period the more difficult it is to get a better quality later on. The *bugfixCount* is used as well as *bugfixLinesAdded*, *bugfixLinesModified*, and *bugfixLinesDeleted* in relation to the base measures such as the number of lines of code added, modified, and deleted for this file. For bug fixes not much new code should be necessary, as most code is added for new requirements. Therefore, *linesAddPerBugfix*, *linesModifiedPerBugfix*, and *linesDeletedPerBugfix* are also interesting indicators, which measure the average lines of code for bug fixes.

Difficulty of problem.
In software development projects usually new classes are added to object-oriented systems when new requirements have to be satisfied. We use the information whether a file was newly introduced during the prediction period as feature for data mining: To measure how often a file was involved during development with the introduction of other new files we use *coChangeNew* as a second indicator. Co-changed files are identified as described in [8].

Relational Aspects.
In object-oriented systems the relationship between classes is important. We use the co-change coupling between files to estimate their relationship. The first feature of this category are couplings such as the number of changes/revisions where other files have been committed with. We use the number of co-changed files relative to the change count of the learning period as feature *coChangedFiles*.

Additionally, we quantify co-changed couplings with features based on commit transactions similar to the size measures for single files: *tLinesAdded*, *tLinesModified*, and *tLinesDeleted*, *tLinesType*, *tChangeType*. These features are defined for transactions (and all the involved files) equivalent to the ones regarding only the file itself (e.g. *linesAdd*).

For file relations we also use bug fix related features: *tLinesAddedPerBugfix* and *tLinesChangedPerBugfix* are two representatives. Additionally, we use *tBugfixLinesAdded*, *tBugfixLinesModified*, and *tBugfixLinesDeleted* relative to the *linesAdded*, *linesModified*, and *linesDeleted*.

Time constraints.
As software processes stress the necessity of certain activities and artifacts, we believe that the time constraints are important for software predictions. The *avgDaysBetweenChanges* feature is defined as the average number of days between revisions. The number of days per line of code added or changed is captured as *avgDaysPerLine*.

Peaks and outliers have been shown to give interesting events in software projects [8]. For the *relativePeakMonth* feature we measure the location of the peak month, which contains most revisions, within the prediction period. The *peakChangeCount* feature describes the number of changes happening during the peak month normalized by the overall

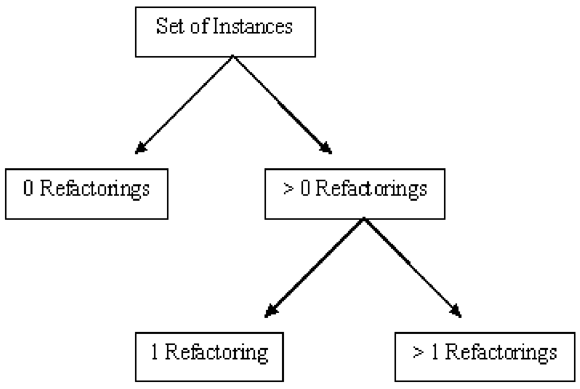

Figure 1. Analysis setup

number of changes. The number of changes is measured based on the months in the prediction period with feature *changeActivityRate*. For more fine grained data the lines of code added and changed relative to the number of months is considered for feature *linesActivityRate*.

4.2.3 Prediction Target: Refactoring Proneness.

With the features described in Section 4.2.2 we predict the number of refactorings. Our prediction models are based on two class problems, where in each application we group files in one of the two classes: having refactoring vs. without refactoring, having one refactoring vs. having several refactorings (see Figure 1). In the field of data mining this procedure is called *classifier stacking*.

Identification of refactorings. The number of refactorings is obtained from the commit messages of the versioning system. We use evolution data not only for the computation of data mining features, but also for the identification of change events as refactorings. For our prediction models we do not distinguish different types of refactorings (e.g. create super class, rename of method/class, extract method, etc.). We only assess the fact that developers try to improve the design. For our investigated projects we identify refactorings based on the commit message, which are provided by developers as comments for their modifications of source code. 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. With several refinements we used for each project 15-20 SQL queries to mark modifications as refactorings.

Evaluation of refactoring identification. With our SQL queries based on the commit messages we labeled 7758 of 60369 changes as refactorings for ArgoUML (13%) and 6251 of the 56050 changes for the Spring framework (11%). To estimate the number of refactorings we marked correctly with our method, we used a statistical evaluation. For each project we randomly selected a subset of 100 modifications and checked whether or not it is a refactoring and if we labeled them as refactoring. Table 3 shows that we obtained a high rate of correct labels. For ArgoUML only one modification (in the random set of 100) was labeled as refactoring, which turned out to be not a refactoring (false positive) and two refactorings were missed (false negative). For the Spring framework we received even better results. Although Spring exhibits a very unbalanced distribution of only 11% refactorings in our random selection, we missed only one refactoring in our labeling with the help of SQL queries and identified only one modification wrong as refactoring.

4.3 Classifiers: Data Mining Algorithms for Prediction Models

For the generation of prediction models we use several data mining algorithms:

- *J48* [2] This classifier builds its decision nodes based on entropy information. It includes improvements for dealing with numeric attributes, missing values, and noisy data (pruning). The great advantage of decision tree compared to other algorithms is that they can be easily interpreted by humans.
- *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. Thus it is a classification algorithm where first regression is built and the result is converted into classes of elements.
- *Rip* Repeated Incremental Pruning is a propositional rule learner. 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* is a Nearest Neighbor generalization. In this case a nearest-neighbor algorithm is used to build rules using non-nested generalized exemplars.

4.3.1 Evaluation of Classification.

To evaluate our prediction models we use 10-fold cross validation. In our analysis of prediction models for refactoring

[2]J48 is the WEKA implementation of the state-of-the-art decision tree learner C4.5

Project	Modifications	Identified Refactorings	Other Type	False Positives	False Negatives
ArgoUML	100	16	84	1	2
Spring	100	11	89	1	1

Table 3. Evaluation of classifying modifications as refactorings

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

Table 4. Outcome of prediction of two groups

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 refactorings = 0), but the classification is wrong (e.g. number refactorings = 1 or ≥ 2). The number of elements that is correctly classified not to belong to the given group forms the true negatives. False negatives are elements that belong to the group of interest, but are erroneously classified to belong be outside of the group. (see Table 4)

- *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 by the formula.

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

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

  We use the F-measure to compare the performance of our prediction models.

5 Results

To investigate our hypotheses in Table 1 we carried out several trials. For the investigation of the following research questions we focus particularly on the results of ArgoUML. Tables 5,6, and 7 show that the values of ArgoUML and Spring are comparable.

5.1 Hypothesis H1: Is evolution data a good predictor of future refactorings?

To answer this question we have a look at Table 5, which describes the quality of prediction models for two open source projects, ArgoUML and Spring framework. We analyze the prediction of two groups of object-oriented classes, the ones having no refactoring in the target period (defined in Table 2) and classes that have one or more refactorings. For both open source projects we list four different classification algorithms: J48, LMT, Rip, and NNge.

We can see that the prediction of classes that are non-refactoring prone have better quality indicators than classes exhibiting refactorings. For ArgoUML both precision and recall are about 0.8, which results in a high f-measure of also 0.8. For classes with refactorings the value range is 0.5 to 0.67, which results in f-measures of approximately 0.6 for the ArgoUML project. These values express that classes with refactorings are more difficult to predict than classes that are not prone to refactoring. One possible explanation is that we do not distinguish between different types of refactorings. Thus, changes to variables, methods and classes are weighted equally. We are solely interested in the fact that refactoring takes place. Refactoring is defined as the activity of improving design of existing code without changing observable behavior. However, the discrepancy between the prediction measures of classes with zero refactoring and classes subject to refactoring is due to the fact that the distribution of these two groups is not equal.

As in both projects the precision is above 0.5 (except the NNge model for Spring) the number of correctly predicted instances is high, which is important for practical application of our approach. When the developer wants to be proactive and to take care of those classes that are prone for refactoring, she has a high probability (in most cases more than 2/3) to investigate relevant files.

Table 2 shows that for the first period of ArgoUML that we investigate the number of classes not being refactored is 603 and the ones with refactoring is 310 (181 + 129).

Project	Classifier	Refactoring = 0			Refactoring >= 1		
	Algorithm	Prec.(%)	Recall(%)	F-measure	Prec.(%)	Recall(%)	F-m.
ArgoUML	J48	0.819	0.834	0.827	0.666	0.642	0.654
	LMT	0.81	0.801	0.806	0.621	0.635	0.628
	Rip	0.768	0.844	0.804	0.624	0.503	0.557
	NNge	0.804	0.849	0.826	0.67	0.597	0.631
Spring framework	J48	0.884	0.937	0.91	0.53	0.366	0.433
	LMT	0.874	0.961	0.916	0.586	0.283	0.381
	Rip	0.876	0.975	0.923	0.689	0.29	0.408
	NNge	0.893	0.913	0.903	0.492	0.434	0.462

Table 5. Predicting non refactoring prone vs refactoring prone classes

The algorithms are biased towards the dominant class distribution (prior) and therefore overestimate classes with no refactoring. To assess the algorithms on equally distributed data sets, we adjust the number of classes by randomly ignoring non-refactoring prone files. Now, the prediction algorithms then perform even better: both refactoring and non-refactoring prone classes are predicted very well with a f-measure better than 0.85 for ArguUML and 0.75 for the Spring framework. Both are high values and we therefore confirm:

It is possible to predict refactoring with evolution data with a high accuracy.

5.2 Hypothesis H2: Is it possible to predict refactorings on short time frames?

To answer the question, whether we can predict refactorings based on short time frames we compare Table 5 with Table 6. The first describes the prediction of refactorings happening in two months based on features taken from three previous months, and the second describes the prediction of the same two months based on features from six months (for exact period definition see Table 2). The prediction with the help of three months shows even better results than the prediction based on six months. Why do we obtain these interesting results? Most open source project, also ArgoUML, rely on agile development practices, where refactoring is used to improve design of source code that has been introduced lately. Therefore, the last few months before refactoring takes place are the ones with the most relevant attributes.

Thus, we conclude:

It is possible to predict refactoring of the next two months based on the last three months of development time.

5.3 Hypothesis H3: Is it possible to distinguish between different groups of files: Without refactoring, with just one refactoring, and with several refactorings?

We investigate this research question with the help of two classification tasks: First we distinguish between classes without refactoring and classes having refactoring. Then we take the second group and examine if we can distinguish classes with just one refactoring from classes with several refactorings (see Figure 1. Table 5 shows the quality values for the prediction non-refactoring prone vs. refactoring prone. We obtain high values for the f-measure, which indicates the overall performance of the prediction models. In Section 5.1 we describe that we could get even better measures, if the number of classes in each group is similar. As a result we can distinguish classes with/without refactoring very well.

Table 7 shows the results of the prediction models distinguishing classes with one refactoring vs. classes with several refactorings. The f-measures are not as high as the ones for the prediction of refactoring-proneness. An f-measure of 0.75 and 0.65 for the two groups of files having refactoring are still good, as the number of classes is much lower than for the prediction models of Table 5, which we can see in Table 2.

When developers take care of classes that are highly refactoring prone, they would investigated the group with >= 2 refactorings. In this group the precision is quite high being close to or above 0.6, which is important developers have a high probability to look at relevant classes. As the recall is also around 2/3, developers have the opportunity to analyze many refactoring prone classes to assess their design quality and their impact on the software architecture.

We come to the following conclusions:

Refactoring prone/non refactoring prone classes can be identified very accurate.

Project	Classifier	Refactoring = 0			Refactoring >= 1		
	Algorithm	Prec.(%)	Recall(%)	F-measure	Prec.(%)	Recall(%)	F-m.
ArgoUML	J48	0.811	0.826	0.818	0.581	0.556	0.568
	NNge	0.799	0.849	0.823	0.593	0.507	0.547
Spring framework	J48	0.874	0.912	0.893	0.514	0.349	0.416
	NNge	0.887	0.899	0.893	0.481	0.413	0.444

Table 6. Predicting refactoring proneness based on a larger time frame (6 months)

Project	Classifier	Refactoring = 1			Refactoring >= 2		
	Algorithm	Prec.(%)	Recall(%)	F-measure	Prec.(%)	Recall(%)	F-m.
ArgoUML	J48	0.747	0.735	0.741	0.636	0.651	0.644
	NNge	0.767	0.746	0.756	0.657	0.682	0.669
Spring framework	J48	0.694	0.718	0.708	0.624	0.617	0.62
	NNge	0.713	0.725	0.719	0.593	0.638	0.615

Table 7. Predicting classes with one refactoring vs. classes with several refactorings

Distinction between classes with one, or several refactorings is possible.

5.4 Hypothesis H4: Is there a common subset of attributes for different projects?

To answer this question we take a look at the decision trees of the two projects in our case study (Figure 2). The trees represent the result of the classification of instances containing no refactoring vs. instances with one or more refactorings. The higher the nodes in the tree the more relevance they have for the prediction. We restrict our trees in Figure 2 to five levels out of twelve to investigate only the most important features.

ArgoUML
The topmost attributes of model ArgoUML, starting from the root are: *linesChangePerChange*, *linesActivityRate*, *coChangedFiles*, *changeFrequencyBefore*, *coChangedNew*, *relNumberChanges*, and *tLinesType*.

Spring framework
The topmost attributes of model Spring, starting at the root are: *tChangesType*, *coChangedNew*, *lastChangeMonth*, *linesActivityRate*, *tLinesChangePerChange*, *tLinesAddPerBugfix*, *coChangedFiles*, *largeChanges*, *relativePeakMonth*, *largeTransactions*, *lastChangeMonth*, and *linesChange*.

Common Both tree models of ArgoUML and Spring framework have the following attributes in common: *coChangedNew*, *linesActivityRate*, and *coChangedFiles*. The first one *coChangedNew* describes the number of files that are created (newly introduced) together with changes to the inspected instance. This feature indicates that new functionality is added, because new classes are introduced together with modifications of the inspected class. If *linesActivityRate* describes that lines are changed often during the entire lifetime of the class, then also the probability for the number of refactorings rises. The number of classes changed together with the inspected one is described by *coChangedFiles*, which takes into account the importance of interrelationships in object-oriented software systems.

The trees have more commonalities than just these features. *linesChangePerChange* is the topmost feature in the tree of ArgoUML, which describes the average number of altered lines within change events, which is measured for each predicted instance. A similar measure appears in the Spring framework where *tLinesChangePerChange* is located on the third level in the second half of the tree, which describes the number of altered lines within the files of the entire transaction where the file of interest was changed. It is surprising that people related features like the *number of authors* is not represented in the trees of our case study.

We conclude that:

There is a common subset of attributes for different projects.

6 Limitations

We found that we can predict refactoring for the Spring framework quite well, but could get even better results for ArgoUML. These could be based on the projects, as they have different project histories. ArgoUML is an older project and started in 1998, whereas Spring framework followed later and started in 2003. Spring exhibits a dynamic evolution based on his young development history. As a result the results of ArgoUML are slightly better, but we still get predictions for the Spring framework with a precision

```
        ArgoUML decision tree
------------------
linesChangePerChange
linesChangePerChange
|   linesActivityRate
|   |   coChangedFiles
|   |   |   coChangedNew
|   |   |   |   tLinesType
|   |   |   |   |   . . .
|   |   |   |   tLinesType
|   |   |   |   |   . . .
|   |   |   coChangedNew
|   |   coChangedFiles
|   |   |   relNumberChanges
|   |   |   |   changeFrequencyBefore
|   |   |   |   changeFrequencyBefore
|   |   |   relNumberChanges
|   linesActivityRate
|   |   changeFrequencyBefore
|   |   changeFrequencyBefore

Number of Leaves  :     41
Size of the tree :      81




        Spring decision tree
------------------
tChangesType
|   coChangedNew
|   coChangedNew
|   |   linesActivityRate
|   |   |   tLinesAddPerBugfix
|   |   |   |   relativePeakMonth
|   |   |   |   |   . . .
|   |   |   |   relativePeakMonth
|   |   |   |   |   . . .
|   |   |   tLinesAddPerBugfix
|   |   linesActivityRate
|   |   |   coChangedFiles
|   |   |   |   largeTransactions
|   |   |   |   largeTransactions
|   |   |   coChangedFiles
|   |   |   |   lastChangeMonth
|   |   |   |   lastChangeMonth
|   |   |   |   |   . . .
tChangesType
|   lastChangeMonth
|   lastChangeMonth
|   |   tLinesChangePerChange
|   |   tLinesChangePerChange
|   |   |   largeChanges
|   |   |   largeChanges
|   |   |   |   linesChange
|   |   |   |   linesChange

Number of Leaves  :     28
Size of the tree :      55
```

Figure 2. Decision trees based on classifier J48 for classification 0 vs. $>=$ 1 refactoring

between 0.53 and 0.89.

We identify refactorings based on the commit messages of revisions entered by developers, when committing changes to files. To assess the quality of our identification technique, we tested our labeling of refactoring with randomly selected revisions. As described in Section 4.2.3 the number of false positives as well as the number of false negatives is very low.

We did not distinguish between the type of refactorings such as class refactorings or method refactorings. Instead we only tried to predict the number of future refactorings based on the past, independently from their nature. As a result simple refactorings such as *rename* are treated equally other refactorings such as *extract super-class* or *introduce new parameter*.

7 Conclusions and Future Work

We have carried out a study of refactoring activities in open source projects. Our work contributes to the understanding of the nature of software projects in several ways.

In contrast to previous studies where quality measured by the number of defects is predicted (e.g. [18]), we created classification models for refactoring, which is an important activity in state-of-the-art software projects. Refactoring is an essential element to keep the quality high and to allow further evolution based on new customer needs.

We described how refactoring prediction models can be built on short time frames. In our case study we used three months of development time to predict the number of refactorings for each file within the following two months. This information can be used by project managers to plan time and budget for future development activities.

We demonstrated that several features such as *lines activity rate* and *number of lines altered per commit* provide much information for the assessment of refactorings. But also the structure of the system is crucial for refactorings, as the *number of co-changed files* and the *number of files introduced during the maintenance* are relevant features. Both ArgoUML and Spring framework have these common features in their decision trees for refactoring prediction although they cover different domains as ArgoUML is a UML tool and Spring framework is a J2EE application server.

Our future work will continue to develop an ECLIPSE plug-in to generate models and discover the classes, in need for the most refactorings, or at least estimates the number of refactorings needed in near future.

References

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

[2] G. Antoniol, M. Di Penta, and E. 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] A. Capiluppi, M. Morisio, and P. 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.

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

[5] S. Demeyer, S. Ducasse, and O. 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.

[6] S. Demeyer, F. Van Rysselberghe, T. Girba, J. Ratzinger, R. Marinescu, T. Mens, B. Du Bois, D. Janssens, S. Ducasse, M. Lanza, M. Rieger, H. Gall, and M. El-Ramly. The lan-simulation: A research and teaching example for refactoring. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 123–131, September 2005.

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

[8] H. Gall, M. Jazayeri, and J. 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, Lisbon, Portugal, September 2003. IEEE Computer Society Press.

[9] 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, 2000.

[10] L. Hatton. Re-examining the fault density-component size connection. *IEEE Software*, 14(2):89–98, March/April 1997.

[11] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proceedings of the International Conference on Software Maintenance*, pages 576–585, October 2002.

[12] P. Knab, M. Pinzger, and A. Bernstein. Predicting defect densities in source code files with decision tree learners. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 119–125, Shanghai, China, May 2006. ACM Press.

[13] M. M. Lehman and L. A. Belady. *Program Evolution - Process of Software Change*. Academic Press, London and New York, 1985.

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

[15] R. Najjar, S. Counsell, G. Loizou, and K. Mannock. The role of constructors in the context of refactoring object-oriented systems. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 111–120, March 2003.

[16] T. J. Ostrand, E. J. Weyuker, and R. 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.

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

[18] J. Ratzinger, M. Pinzger, and H. 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.

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

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

[21] T. Zimmermann, P. Weißgerber, S. Diehl, and A. 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.