EQ-Mine: Predicting Short-Term Defects for Software Evolution

Jacek Ratzinger[1], Martin Pinzger[2], and Harald Gall[2]

[1] Distributed Systems Group
Vienna University of Technology, Austria
ratzinger@infosys.tuwien.ac.at
[2] s.e.a.l. – software evolution and architecture lab
University of Zurich, Switzerland
{pinzger,gall}@ifi.unizh.ch

Abstract. We use 63 features extracted from sources such as versioning and issue tracking systems to predict defects in short time frames of two months. Our multivariate approach covers aspects of software projects such as size, team structure, process orientation, complexity of existing solution, difficulty of problem, coupling aspects, time constrains, and testing data. We investigate the predictability of several severities of defects in software projects. Are defects with high severity difficult to predict? Are prediction models for defects that are discovered by internal staff similar to models for defects reported from the field?
We present both an exact numerical prediction of future defect numbers based on regression models as well as a classification of software components as defect-prone based on the C4.5 decision tree. We create models to accurately predict short-term defects in a study of 5 applications composed of more than 8.000 classes and 700.000 lines of code. The model quality is assessed based on 10-fold cross validation.

Keywords: Software Evolution, Defect Density, Quality Prediction, Machine Learning, Regression, Classification

1 Introduction

We want to improve the evolvability of software by providing prediction models to assess quality as soon as possible in the product life cycle. When software systems evolve we need to measure the outcome of the systems before shipping them to customers. Software management systems such as the concurrent versioning system (CVS) and issue tracking systems (Jira) capture data about the evolution of the software during development. Our approach, EQ-Mine uses this data to compute a number of features, which are computed for source file revisions in the pre- and post-release phases. Based on these evolution measures we then set up a prediction model.

To evaluate the defect density prediction capabilities of our evolution measures we apply three data mining algorithms and test 5 specified hypotheses.

Results clearly underline that defect prediction models have to take into account different aspects and measures of the software development and maintenance [1]. In extension to our previous work on predicting defect density of source files [2] we use detailed evolution data from an industrial software project and include team structure and process measures.

The remaining paper is structured as follows. It starts with the formulation of our research hypotheses (section 2). Related work is discussed in section 3. In section 4 we describe the evolution measures used to build our defect prediction model. Our approach is evaluated on a case study in section 5. We finalize this paper with our conclusions and intent for future work in section 6.

2 Hypotheses

To guide the metrics selection for defect prediction and our evaluation with a case study, we set up several hypotheses. In contrast to previous research approaches (e.g. [3]) EQ-Mine aims at a fine-grained level. Our hypothesis are used to focus on different aspects of our fine grained analysis such as the severity of defects, the timing of predictions around releases, and the type of defect discovered (internal vs. external):

– *H1: Defect density can be predicted based on a short time-frame.* Previous research focused on the prediction of longer time-frames such as releases [4, 5]. In our research we focus on months as time scale and use two months of development time to predict defect densities for the following two months.
– *H2: Critical defects with high severity have a low regularity.* Prediction models build on the regularity of the underlying data and can predict events better that correspond to this regularity. We expect that defects that are critical are more difficult to detect as they "hide better" during the testing and product delivery.
– *H3: Quality predictions before a release are more accurate than after a release.* Project quality can be estimated in different stages of the development process. Some stages are more difficult to assess than others. Previous studies already indicated that the accuracy of data mining in software engineering varies over time (e.g. [5]). We expect that defects that are detected before a release date are easier to predict than defects that are reported afterwards.
– *H4: Defects discovered by internal staff have more regularity than defects reported by the customer.* For prediction model creation it is an important input to know where the defect comes from. Was it recognized by the internal staff (e.g. during testing) or does the defect report come from customer sites? We expect that internally and externally detected defects have different characteristics. As a result one group can be easier predicted than the other one.
– *H5: Different aspects of software evolution have to be regarded for an accurate defect prediction.* We use a large amount of evolution indicators for defect prediction. These indicators can be grouped into several categories such as

size and complexity measures, indicators for the complexity of the existing solution and team related issues. For defect prediction we expect that data mining features from many different categories are important.

3 Related Work

Many organizations want to predict software quality before their systems are used. Fenton and Neil provide a critical review of literature that describes several software metrics and a wide range of prediction models [1]. They found out that most of the statistical models are based on size and complexity metrics with the aim to predict the number of defects in a system. Others are based on testing results, the testing process, the "quality" of the development process, or take a multivariate approach.

There are various techniques to identify critical code pieces. The most common one is to define typical bug patterns that are derived from experience and published common pitfalls in a certain programming language. Wagner et al. [6] analyzed several industrial and development projects with the help of bug detection tools as well as with other types of defect-detection techniques.

Khoshgoftaar et al. [7] use software metrics as input to classification trees to predict fault-prone modules. One release provides the training dataset and the subsequent release is used for evaluation purpose. They claim that the resulting model achieved useful accuracy in spite of the very small proportion of fault-prone modules in the system. Classification trees generate partition trees based on a training data set describing known experiences of interest (e.g. characteristics of the software). The tree structure is intuitive and can be easily interpreted. Briand et al. [8] try to improve the predictive capabilities by combining the expressiveness of classification trees with the rigor of a statistical basis. Their approach called OSR generates a set of patterns relevant to the predicted object estimated based on the entropy H.

There are different reasons for each fault: Some faults exist because of errors in the specification of requirements. Others are directly attributable to errors committed in the design process. Finally, there are errors that are introduced directly into the source. Nikora and Munson developed a standard for the enumeration of faults based on the structural characteristics of the MDS software system [9]. Changes to the system are visible at the module level (i.e. procedures and functions) and therefore this level of granularity is measured. This fault measurement process was then applied to a software system's structural evolution during its development. Every change to the software system was measured and every fault was identified and tracked to a specific line of code. The rate of change in program modules should serve as a good index of the rate of fault introduction. In a study the application of machine learning (inductive) technique was tested for the software maintenance process. Shirabad et al. [10] present an example of an artificial intelligence method that can be used in future maintenance activities. An induction algorithm is applied to a set of pre-classified training examples of the concept we want to learn. The large size and complexity of systems, high staff turnover, poor documentation and the long periods of time these systems must be maintained leads to a lack of knowledge in how to proceed the maintenance of software systems.

Only a small number of empirical studies using industrial software systems are performed and published. Ostrand and Weyuker, for example, evaluated a large inventory tracking system at AT&T [4]. They analyzed how faults are distributed over different releases. They discovered that faults are always heavily concentrated in a relatively small number of releases during the entire life cycle. Additionally the number of faults is getting higher as the product matures and high-fault modules tend to remain high fault in later releases. So it would be worthwhile to concentrate fault detection on a relatively small number of high fault-prone releases, if they can be identified early.

4 Data Measures

To mine software development projects we use the data obtained from versioning system (CVS) and issue tracking system (Jira). CVS enable the handling of different versions of files in cooperating teams. This tool logs every change event, which provides the necessary information about the history of a software system. The log-information for our mining approach—pure textual, human readable information—is retrieved via standard command line tools, parsed and stored in the release history database [11].

Jira manages data about project issues such as bug reports or feature requests. This system give a historical overview of the requirements and their implementations. We extract the data based on its backup facility, where the entire issue data can be exported into XML files. These files are processed to import the information into our database. In a post-processing step we link issues from Jira to log information from CVS using the comments of developers in commit messages by searching for issue numbers. In addition we distinguish between issues created by developers and issues created by customers by linking issue reporters to CVS authors. Issues are counted as reported by internal staff when the issue reporter can be linked to a CVS author otherwise the issue is defined to be external (e.g. hotline).

4.1 Features

From the linked data in the release history database we compute 63 evolution measures that are considered as features for data mining. These features are gathered on file basis, where data from all revisions of a file within a predefined time period is summarized. To build a balanced prediction model we create features to represent several important aspects of software development such as the complexity of the designed solution, process used for development, interrelation of classes, etc. As previous studies [2, 3] discovered that relative features provide better performance in prediction than absolute ones, we decided that all our 63 features have to be relative. For EQ-Mina we set up the following categories of features for each file containing changes within the inspection period:

Size. This category groups "classical" measures such as lines of code from an evolution perspective: *linesAdded*, *linesModified*, or *linesDeleted* relative to the total *LOC* of a file. For example a file had three revisions within the learning period adding 3, 5, and 4 lines and this file had 184 lines before the learning period, we feed into the data mining: (*linesAdded* = (3+5+4)/184) => ($\sum$ defects).

Other features of this category are *linesType*, which defines if 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*. We expect that this number is an important feature in the data mining, as other studies have found out that small modules are more defect-prone than large ones. [12, 13]

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 is the possibility of rework and mistakes. We define a feature for the *authorCount* relative to the *changeCount*. Further, the interrelation in 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 for our prediction models.

Process orientation. In this category we assemble features that define how disciplined people follow software development processes. For source code changes developers have to include the issue number in their commit message to the versioning system. We define a feature regarding *issueCount* relative to *changeCount*. The developer is requested to also provide some rationale in the commit message. Thus, we use *withNoMessage* measuring changes without any commit comment as a feature for prediction.

In each project the distribution between different priorities of issues should be balanced. Usually, the number of issues with highest priority is very low. A high value may indicate problems in the project that have effects on quality and re-work amount. Accordingly, we investigate *highPriorityIssues* and *middlePriorityIssues* relative to the total number of issues. Also the time to close certain classes of issues provides interesting input for prediction and we use *avgDaysHighPriorityIssues* and *avgDaysMiddlePriorityIssues* in relation to the average number of days that are necessary to close an issue.

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

Complexity of existing solution. According to the laws of software evolution [14], software continuously becomes 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 during the entire lifetime of the file relative to the months of the lifetime. 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* before our prediction period relative to the general number of changes before the prediction period. 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 interesting indicators, which measure the average lines of code for bug fixes.

Difficulty of problem. New classes are added to object-oriented systems when new features and 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 *cochangeNewFiles* as a second indicator. Co-changed files are identified as described in [15].

The amount of information necessary to describe a requirement is also an important source of information. The feature *issueAttachments* identifies the number of attachments per issue.

Relational Aspects. In object-oriented systems the relationship between classes is an important metrics. We use the co-change coupling between files to estimate their relationship. We use the number of co-changed files relative to the change count 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* relative to lines of code added, modified, and deleted. The *TLinesType* describes if the transactions contained more lines added or lines modified. *TChangeType* is a coarser grained feature that describes if this file was part of transactions with more adding revisions or more modifying revisions.

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 constrains 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 captured as *avgDaysPerLine.*

Peaks and outliers have been shown to give interesting events in software projects [15]. 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 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 regarded for feature *linesActivityRate*.

Testing. We use testing metrics as an input to prediction models, because they allow estimating the remaining bug number. The number of bug fixes initiated by the developers itself provides insight into the quality attentiveness of the team and are covered by feature *bugfixesDiscoveredByDeveloper*.

4.2 Data Mining

For model generation and evaluation we use the data mining tool called Weka [16]. It provides algorithms for different data mining tasks such as classification, clustering, and association analysis. For our prediction and classification models we selected linear regression, regression trees (M5), and classifier C4.5. The regression algorithms are used to predict the number of defects for a class from its evolution attributes.

The following metrics are used to assess the quality of our numeric prediction models:

- *Correlation Coefficient* (C. Coef.) ranges from -1 to 1 and measures the statistical correlation between the predicted values and the actual ones in the test set. A value of 0 indicates no correlation, whereas 1 describes a perfect correlation. Negative correlation indicates inverse correlation, but should not occur for prediction models.
- *Mean Absolute Error* (Abs. Error) is the average of the magnitude of individual absolute errors. This assessment metrics does not have a fixed range like the correlation coefficient, but is geared to the values to be predicted. In our case the number of defects per file is predicted, which ranges from 1 to 6 and 16 respectively (see Table 1 and Table 2). As a result, the closer the mean absolute error is to 0 the better. A value of 1 denotes that on average the predicted value differs from the actual number of defects by 1 (e.g. 3,5 instead of 4).
- *Mean Squared Error* (Sqr. Error) is the average of the squared magnitude of individual errors and it tends to exaggerate the effect of outliers – instances with larger prediction error – more than mean absolute error. The range of the mean squared error is geared to the ranges of predicted values, similar to the mean absolute error. But this time the error metrics is squared, which overemphasize predictions that are far away of the actual number of defects. The quality of the prediction model is good, when the mean squared error is close to the mean absolute error.

The quality of our prediction models is assessed through 10-fold cross validation. For this method the set of instances is splitt randomly into 10 sub-sets (folds) and the model is build 10 times and validated 10 times. For each turn the classification model is trained on nine folds and the remaining one is used for testing. The resulting 10 quality measures are averaged to yield an overall quality estimation. Therefore, 10-fold cross validation is a strong validation technique.

5 Case Study

For our case study with EQ-Mine we analyzed a commercial software system from the health care environment. The software system is composed of 5 applications such as a clinical workstation or a patient administration system. This object-oriented system is built in Java consisting of 8.600 classes with 735.000 lines of code. For the clinical workstation a plug-in framework similar to the one of Eclipse is used and currently 51 plug-ins are implemented. The development is supported by CVS as the versioning system for source files and Jira as the issue tracking system. We analyzed the last two releases of this software system: One in the first half of 2006 and the other one in the middle of 2005.

Number of defects per file (all severities)	Number of files	Number of defects per file (high severity)	Number of files
1	46	1	10
2	11	2	2
3	5	3	1
4	7	4	0
5	2	5	0
6	1	6	0

Table 1. Pre-release: Number of files distinguishing between the ones with defects of all severities and files where defects with high severity were found.

5.1 Experimental Setup

For our experiments we investigated 8 months of software evolution in our case study. We use two months of development time to predict the defects of the following two months, which builds up a 4 months time frame. We compare the predictions before the release date with the predictions after it, which results in a period of 8 months. Before the release we create prediction models for defects in general and for defects with high severity. These models can be compared to the ones after. After the release date we additionally distinguish defects discovered by internal staff vs. defects reported from the field (customer). With this experimental set up we test our hypotheses from Section 2.

Number of defects with severity reported by	No. of files severity=all int. & ext.	No. of files severity=all internal staff	No. of files severity=all external customer	No. of files severity=high int. & ext.
1	46	30	32	21
2	21	12	7	1
3	8	6	1	0
4	6	4	1	0
5	5	4	0	0
7	1	1	0	0
12	1	1	0	0
16	1	1	0	0

Table 2. Post-release: Number of files distinguishing different types of defects.

5.2 Results

Short Time Frames. Our analysis focuses on short time frames. To evaluate H1 of Section 2 we use two months of development time to predict the following two months. Table 3 shows several models predicting defects before the release where the two months period for defect counting are laid directly before the release date and the other two months before this two target months are taken to collect feature variables for the prediction models. In the first Table 3(a) we can see that we obtain a correlation coefficient larger than 0.5, which is a quite good correlation. The mean absolute error is low with 0.46 for linear regression and 0.36 for M5 and the mean squared error is also low with 0.79 for linear regression and 0.67 for M5. In order to assess these prediction errors, Table 1 describes the defect distribution of the two target months. As mean squared error emphasizes outliers, we can state that the overall error performance of the prediction of all pre-release defects is very good.

	C. Coef.	Abs. Error	Sqr. Error
Lin. Reg.	0.5031	0.4604	0.7881
M5	0.6137	0.3602	0.6674

(a) All defects

	C. Coef.	Abs. Error	Sqr. Error
Lin. Reg.	-0.0424	0.1352	0.3173
M5	0.0927	0.0792	0.2589

(b) High severity defects

Table 3. Prediction pre-release defects

To confirm our first hypothesis Table 4(a) lists the quality measures for the prediction of post-release defects. There the values are not as good as for pre-release defects, but the correlation coefficients are still close to 0.5. Therefore, we confirm H1:

> We can predict short time frames of two months based on feature data of two months.

High Severity. Table 3(b) shows the results for the prediction models on pre-release defects with high severity. We get the severity level of each defect from the issue tracking system, where the defect reporter assigns severity levels. The quality measures for high severity defects differ from the prediction of all defects, because the number and distribution of high severity defects have other characteristics (see Table 1). It is interesting that linear regression has only a negative correlation coefficient. But also M5 can only reach a very low correlation coefficient of 0.10. The overall error level is low because of the small defect bandwidth of 0 up to 3.

	C. Coef.	Abs. Error	Sqr. Error
Lin. Reg.	0.5041	0.9443	1.5285
M5	0.4898	0.7743	1.4152

(a) All defects

	C. Coef.	Abs. Error	Sqr. Error
Lin. Reg.	0.4464	0.9012	1.5151
M5	0.5285	0.688	1.3194

(b) Defects discovered internally (through test + development)

	C. Coef.	Abs. Error	Sqr. Error
Lin. Reg.	0.253	0.3663	0.5699
M5	0.4716	0.2606	0.4574

(c) Defects discovered externally (through customer + partner companies)

	C. Coef.	Abs. Error	Sqr. Error
Lin. Reg.	0.1579	0.1973	0.3175
M5	0.087	0.1492	0.3048

(d) High severity defects

Table 4. Prediction post-release defects

For the post-release prediction of high severity defects in Table 4(d) the correlation coefficient of 0.16 is slightly better. The prediction errors are slightly worse, but this is due to the fact that there are more post-release defects with high severity than pre-release. However, we can conclude:

Defects with high severity cannot be predicted with such a precision as overall defects.

Before vs. After Release. Our hypothesis H3 states that pre-release defects can be better predicted than the post-release ones. When we compare Table 3(a) with Table 4(a) we see that our hypothesis seems to be confirmed. The correlation coefficients of linear regression are very similar, but the prediction errors are higher for pre-release defects. This situation is even more remarkable for M5, as the pre-release correlation coefficient reaches 0.61 whereas the post-release remains at 0.49. For these prediction models also the two error measures are much higher for post-release. While comparing the defect distribution between pre-release in Table 1 with post-release in Table 2, we could believe that the high error rate is due to the fact that we discovered more files with many defects that occur post-release than pre-release. But when we repeat the model creation of post-release defects with a similar distribution to pre-release, we get still a mean absolute error of 0.68 and a mean squared error of 1.06, which is still clearly larger than for pre-release.

What about high severity defects? Are they still better predictable before a release than after? When we look at Table 3(b) and Table 4(d) we see a similar picture for this subgroup of defects. Only the correlation coefficient for linear regression is higher for post-release defects than for pre-release, because there are many more high severity defects after the release. This could be because the defects reported from customers are ranked higher than when they are discovered internally, in order to stress the fact that the defects from customers have to be fixed fast. When we repeat the model creation with similar distributions of pre-release and post-release we get similar correlation coefficients but higher prediction errors for post-release. Therefore, we can conclude that:

> Predictions of post-release defects have higher errors than for models generated for pre-release.

Discovered Internally vs. Externally. We show the difference between prediction of defects discovered by internal staff (testers, developers) vs. defects discovered externally (e.g. customer, partner companies) in Table 4(b) and Table 4(c). For internal defects the correlation coefficient is larger than 0.5, which is produced by the M5 predictor. Although it seems that the prediction error is lower for external defects than for internal ones, this result may be caused by the fact that there are no files with many externally discovered defects (see post-release defect distribution in Table 2). However, when we redo the prediction for internal defects with a similar distribution as for external defects, we get a mean absolute error of 0.48 and a mean squared error of 0.86 with a correlation coefficient of 0.47. As a result, we can partly reject H4 and conclude that:

> Defects discovered externally by customers and partner companies can be predicted with lower absolute and squared error than defects discovered internally by testers and developers.

Aspects of Prediction Models. To analyze the aspects of prediction models in more detail we created two cases using the C4.5 tree classifier: The first model distinguishes between files that are defect-prone vs. files without defects. The second tree model separates the files with just one defect from the ones with several defects. At each node in the tree, a value for the given feature is used to divide the entities into two groups: files with a feature value large/smaller than the threshold. The leafs of the decision trees provide a label for the entities (e.g. predicted number of defects). For each file such a tree has to be traversed according to its features to obtain the predicted class. If a node has no or only one successor than it is defined to be a leaf node for a part of the tree.

Tree 1 describes that the feature bearing the most information concerning defect-proneness is the location of the peak month, where the peak month is defined as the one containing the most change events for the analyzed file. Features on the second level are change activity rate and author count. Relative peak month and change activity rate represent the category of time constraints. Nevertheless, the tree is composed of features from many different categories.

Author count and author switches belong to the team category. The number of resolved issues in relation to all issues referenced by source code revisions is an indicator for the process category, similar to the number of source adding changes in relation to the overall change count. Also the ratio of revisions without a commit message describes the process orientation of the development. The number of lines added per bug fix provides insight into the development process itself. *We conclude that not size and complexity measures dominate defect-proneness, but many people-related issues are important.*

tree root
relativePeakMonth
— changeActivityRate
— — resolvedIssues
— — — bugfixLinesAdded
— — — — withNoMessage
relativePeakMonth
— authorCount
— — addingChanges
— — — authorSwitches

Tree 1: Pre-release with/without defects

Tree 2 describes the prediction model evaluating the defect-prone files (one vs. several defects). This classification tree is much smaller than the previous one for prediction of defect-prone files. Nevertheless, it contains data mining features from many categories. The top level and the bottom level both regard lines edited during bug fixing, but on the first level the lines added to the file are of interest whereas at the bottom the relational aspect is central with lines deleted in all files of common commit transactions. Additionally, the team aspect plays an important role, as the number of author switches is the feature on the second level. The model is completed by features indicating the ratio of adding and changing modifications.

tree root
linesAddPerBugfix
— authorSwitches
— — addingChanges
— — — modifyingChanges
— — — — TBugfixLinesDel

Tree 2: Pre-release one vs. several defects

From these classifications we conclude that:

Multiple aspects such as time constraints, process orientation, team related and bug-fix related features play an important role in defect prediction models.

5.3 Limitations

Our mining approach is strongly related with the quality of our data for the case study. As a result, validity of our findings is related with the data of the versioning and issue tracking system. Versioning systems register single events such as commits of developers, where the event recording depends on the work habits of the developers. However, we could show that an averaging effect supports statistical analysis [17] in general.

Our data rely strongly on automated processing. On one hand this ensures constancy, but on the other hand it is a source of blurring effects. In our case we extracted issue numbers from commit messages to map the two information systems. To improve the situation we could try to map from bug reports to code changes based on commit dates and issue dates as described in [5]. In our case this approach does not provide any valuable mappings, which we discovered on a random sample of 100 discovered matches.

We can only identify locations of defects corrections based on change data from versioning systems and derive from this information prediction models for components. Bug fixes can take place at locations different to the source of defects. Similar approaches are used by other researchers [5, 4, 3]. With predicting defect corrections, we provide insight into improvement efforts, as defect fixes could be places being in urgent need of code stabilization.

For our empirical study we selected software applications of different types such as graphical workstations, administrative consoles, archiving and communication systems, etc. We still cannot claim generalization of our approach on other kinds of software systems. Therefore, we need to evaluate the applicability of EQ-Mine on each specific software project. Nevertheless, this research work contributes to the existing empirical body of knowledge.

6 Conclusions and Future Work

In this work we have investigated several aspects of defect prediction based on a large industrial case study. Our research contributes to the body of knowledge in the field of software quality estimation in several ways. We conducted one of the first studies dealing with fine grained predictions of defects. We estimate the defect proneness based on a short time-frame. With this approach project managers can decide on the best time-frame for release and take preventive actions to improve user satisfaction. Additionally, we compare defect prediction before and after releases of our case study and discovered that in both cases an accurate prediction model can be established. In contrast to other studies, we investigated the predictability of defects of different severity. We could show that prediction of defects with high severity has lower precision. We also analyzed customer perceived quality, where defects reported by customers need other prediction models than defects discovered by internal staff such as testing.

In order to create accurate prediction models we inspected different aspects of software projects. Although size was already used in many other studies it is still an important input for prediction. We extend size measures with relational aspects, where we use the data about evolutionary co-change coupling of software entities. We can show that, for example, the number of lines added to all classes on common changes is as important for defect prediction of a class as the number of lines added to this particular class. Other aspects of our approach are the complexity of the existing solution and the difficulty of the problem in general, as they are causes of software defects. We include people issues of different types in our analysis to cover another important cause of defects. When a developer has to work on software that somebody else has initially written mistakes can occur, because she has to understand the design of her colleague. Factors such as author switches are covered by our team group of data mining features. The discipline of a developer does also influence defect probability. As a result we use indicators for process related issues. Finally, we include time constrains and testing related features into our defect prediction models. The models were created based on 63 data mining features from the 8 categories described.

In our future work we focus on the following topics:

- *Software Structure.* As we currently use evolution measures for quality estimations, we intend to enrich our models with information about software structures. Object-oriented inheritance hierarchies as well as data and control flow information provide many insights into software systems, which we will include in our quality considerations.
- *Automation.* Our analysis relies on automated data processing such as information retrieval, mapping of defect and version information, and feature computation. The model creation relies on scripts using the Weka data mining tool [16]. Integrated tools providing predictions and model details such as the most important features can help different stakeholders. On the one hand, developers could profit from this information best, when it is available in the development environment. On the other hand, project managers need a lightweight tool separated from development environments to base their decisions on.

7 Acknowledgments

We thank Peter Vorburger for his valuable input and thoughts to this research work. Special thanks go to André Neubauer and others for their comments on earlier versions of this paper.

References

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

2. Knab, P., Pinzger, M., Bernstein, A.: Predicting defect densities in source code files with decision tree learners. In: Proceedings of the International Workshop on Mining Software Repositories, Shanghai, China, ACM Press (2006) 119–125
3. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings of the International Conference on Software Engineering, St. Louis, MO, USA (2005) 284–292
4. Ostrand, T.J., Weyuker, E.J.: The distribution of faults in a large industrial software system. In: Proceedings of the International Symposium on Software Testing and Analysis, Rome, Italy (2002) 55–64
5. Schröter, A., Zimmermann, T., Zeller, A.: Predicting component failures at design time. In: Proceedings of the International Symposium on Empirical Software Engineering, Rio de Janeiro, Brazil (2006) 18–27
6. Wagner, S., Jürjens, J., Koller, C., Trischberger, P.: Comparing bug finding tools with reviews and tests. In: Proceedings of the International Conference on Testing of Communicating Systems, Montreal, Canada (2005) 40–55
7. Khoshgoftaar, T.M., Yuan, X., Allen, E.B., Jones, W.D., Hudepohl, J.P.: Uncertain classification of fault-prone software modules. Empirical Software Engineering **7**(4) (2002) 297–318
8. Briand, L.C., Basili, V.R., Thomas, W.M.: A pattern recognition approach for software engineering data analysis. IEEE Transactions on Software Engineering **18**(11) (1992) 931–942
9. Nikora, A.P., Munson, J.C.: Developing fault predictors for evolving software systems. In: Proceedings of the Software Metrics Symposium, Sydney, Australia (2003) 338–350
10. Shirabad, J.S., Lethbridge, T.C., Matwin, S.: Mining the maintenance history of a legacy software system. In: Proceedings of the International Conference on Software Maintenance, Amsterdam, The Netherlands (2003) 95–104
11. Fischer, M., Pinzger, M., Gall, H.: Populating a release history database from version control and bug tracking systems. In: Proceedings of the International Conference on Software Maintenance, Amsterdam, Netherlands, IEEE Computer Society Press (2003) 23–32
12. Moeller, K., Paulish, D.: An empirical investigation of software fault distribution. In: Proceedings of the International Software Metrics Symposium. (1993) 82–90
13. Hatton, L.: Re-examining the fault density-component size connection. IEEE Software **14**(2) (1997) 89–98
14. Lehman, M.M., Belady, L.A.: Program Evolution - Process of Software Change. Academic Press, London and New York (1985)
15. Gall, H., Jazayeri, M., Ratzinger (former Krajewski), J.: CVS release history data for detecting logical couplings. In: Proceedings of the International Workshop on Principles of Software Evolution, Lisbon, Portugal, IEEE Computer Society Press (2003) 13–23
16. Witten, I.H., Frank, E.: Data Mining: Practical machine learning tools and techniques. 2 edn. Morgan Kaufmann, San Francisco, USA (2005)
17. Ratzinger, J., Fischer, M., Gall, H.: Evolens: Lens-view visualizations of evolution data. In: Proceedings of the International Workshop on Principles of Software Evolution, Lisbon, Portugal (2005) 103–112