Predicting Defects in SAP Java Code: An Experience Report

Tilman Holschuh[*]
SQS AG
tilman.holschuh@sqs.de

Markus Päuser
SAP AG
markus.paeuser@sap.com

Kim Herzig[*]
Saarland University
kim@cs.uni-sb.de

Thomas Zimmermann[*]
Microsoft Research
tz@acm.org

Rahul Premraj[*]
Vrije Universiteit Amsterdam
premraj@cs.uni-sb.de

Andreas Zeller
Saarland University
zeller@acm.org

Abstract

Which components of a large software system are the most defect-prone? In a study on a large SAP Java system, we evaluated and compared a number of defect predictors, based on code features such as complexity metrics, static error detectors, change frequency, or component imports, thus replicating a number of earlier case studies in an industrial context. We found the overall predictive power to be lower than expected; still, the resulting regression models successfully predicted 50–60% of the 20% most defect-prone components.

1. Introduction

To get the most out of quality assurance, it is important to *allocate quality assurance resources wisely*—i.e., to decide how much test and reviewing effort goes into the individual components of a large software system. This task is challenging, because neither defects nor their potential damage is distributed equally across the system. One therefore needs to focus on where to put the most effort—for instance, the components with the highest complexity, the components most critical for essential functionality, or the components most frequently or most recently changed.

To support such allocation decisions at SAP, we wanted to leverage actual *facts* from the *project history*—in particular, the *defect history* of a project, telling which defect had been fixed when and by whom in which component. For instance, if we knew which components have shown to be especially defect-prone in the past, we could justify spending extra effort on these in the future. If we additionally could determine which *features* of these components correlate with proneness to defects, we could check new components for such features—and predict their defect likelihood.

In the past, such historic facts had been exploited a number of times. Weyuker and Ostrand [23] had shown that 80% of the defects in a large AT&T system are located in just 20% of the files. In a study at Microsoft, Nagappan et al. [21] used complexity metrics to successfully predict the most defect-prone components. Schröter et al. [26] showed that *imports* could be used as defect predictors.

All these works rely on the same principal scheme, summarized in Figure 1: By *mining* defect and change histories, one determines the number of past defects fixed in a component. The resulting defect counts can then be related to other component features (such as complexity metrics, dependencies, change frequency. . .) to learn which features typically correlate with defect counts. This allows to build *predictor models* which predict the defect count for a new component based on these features.

We were curious to learn whether such approaches

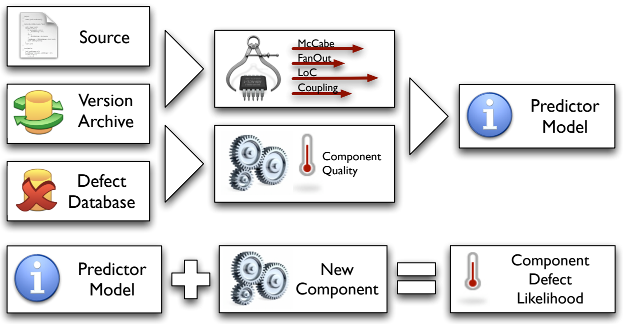

Figure 1. Predicting Defects in a Nutshell. We map previous defects to components and relate the resulting quality to component features. These features can then be used to predict future quality.

[*]This work was conducted while *Tilman Holschuh* and *Kim Herzig* were research interns with SAP, and while *Rahul Premraj* and *Thomas Zimmermann* were research associates with Saarland University.

would be applicable to SAP's code base as well. We therefore decided to *replicate* these earlier case studies on one of SAP's larger systems. In this experience report, we make the following contributions:

- We *replicated* and compared the studies of Nagappan et al. [21] and Schröter et al. [26] on a much larger Java system in a different industrial context;
- We *extended* the study to additionally consider dependency and code smell metrics;
- We *experienced* that mining software archives provided very valuable data;
- We *learned* that predictors obtained from one project are hard to generalize to other projects, or a system of several projects;
- We *expect* that more features, in particular process features, may be required to differentiate between projects and further improve prediction.

This paper starts with the context at SAP (Section 2), followed by the related work (Section 3). We then discuss how we obtained defect data (Section 4) and metrics (Section 5), followed by the predictor models we used (Section 6). The results we obtained are discussed in Section 7, closing with the lessons we learned on the benefits and limitations of mining and predicting defects (Section 8). Section 9 provides a conclusion and an outlook to future work.

2. Context and Goal

Since 1997, SAP uses the Java programming language for producing software systems. Java is primarily used for interfaces or engines on top of base systems such as SAP ERP, which are mostly written in the ABAP programming language. Overall, at SAP, several million lines of Java are used in production code.

In this paper, we describe our experience on a very large Java system consisting of several hundred individual *projects*. Projects with coherent tasks are grouped into so-called *tracks*.

In each of these projects, the production process is composed of a *development* and a *maintenance* phase. During development, the product is built and enhanced as well as tested locally. At the end of development, a maintenance branch is created from the main development trunk; in this branch, only corrections are applied, but no new features are added. From this branch, individual versions (called *service packs*) are assembled, tested, packaged, and deployed to the customer. Tracking the individual changes, versions, and branches is being handled by standard software configuration management (SCM) systems such as Perforce.

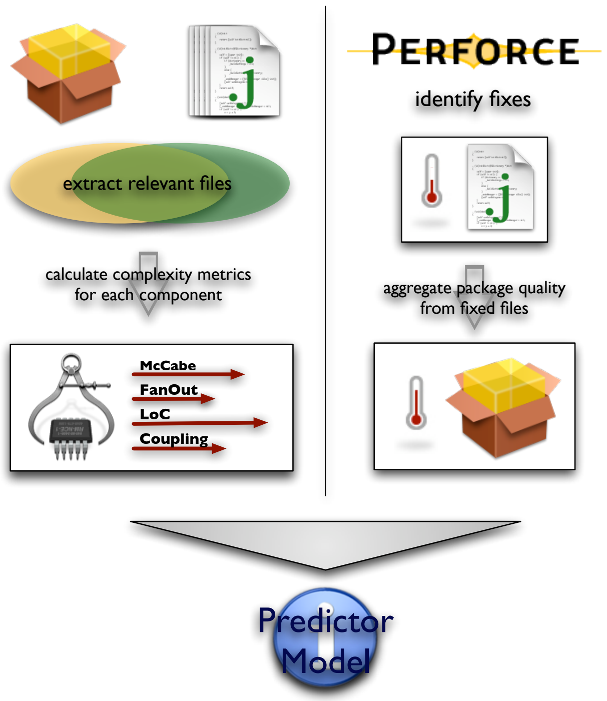
PERFORCE
identify fixes
extract relevant files
calculate complexity metrics for each component
aggregate package quality from fixed files
McCabe
FanOut
LoC
Coupling
Predictor Model

Figure 2. Collecting metrics and defect counts per component

As post-release fixes are applied only in the maintenance branch, it is easy to relate these fixes to individual service packs. Thus, for each service pack, we can exactly tell *which* fixes were applied, and *where* they were applied. By counting the fixes per artifact, we wanted to have a *defect distribution*, showing us in which artifacts the most defects occurred—or, more precisely, the most defects were fixed.

At the same time, we wanted to extract appropriate *features* from software components—in particular, *complexity metrics*—and relate these to defect counts, as sketched in Figure 2. Our overall goal was to build statistical *predictor models* that would allow predicting the defect counts for new components, based on their features—and thus assist in deciding where to allocate quality assurance resources.

3. Background

We started this project by studying the related work.

Defects. More than 20 years ago, Basili et al. [5] explored data on defect distribution, module size and complexity, as well as fixing effort in a medium-sized software project. One of their key results was that defect density decreased with increasing module size—a result that contrasted with common assumptions about modular programming. To collect the data, developers filled out forms whenever they changed a file. Similar to the work of Koru et al. [16] that undermines their results, we aimed for collecting such data automatically.

Fenton and Ohlsson [8] could not replicate the results of Basili et al. [5] and pointed out that defects found during testing and in production are not related. Consequently, we exclusively focused on defects fixed during production.

Software Archives. Recently, researchers have learned to leverage the enormous amount of process data as contained in software archives [25, 27, 29, 31, 32]. By mapping defects to components, one can compute the defect density [10] and identify defect-inducing changes [27]. This fine-grained mapping forms the base for our prediction.

Metrics. Software quality management makes use of *metrics* to measure the quality of a product as well as the effort for its development and maintenance. The product metrics compiled by Fenton and Pleeger [9] measure the size, complexity, and the programming style of a software; the best known are the number of non-comment non-blank code lines (LOC), McCabe's complexity measure [18] as well as the Halstead metrics [13].

As these early metrics were unsuited for object-oriented design, researchers have explored better alternatives. Chidamber und Kemerer's OO metrics [6] were successfully used by Basili [4] to predict defect densities. The cohesion metrics by Henderson-Sellers [14] refine Chidamber and Kemerer's cohesion metric; see [12] for a discussion. In [17], Robert C. Martin presented dependency metrics to measure the quality of an object-oriented design in terms of it being reusable and conforming to OO design principles.

Defect prediction. Researchers have long recognized the potential value of defect prediction. Ostrand et al. [24] used historic data from version archives to predict defects in two large software systems. Their predictor model considered whether a file had been changed in the previous release; the size and age of the file; whether it was present in the previous release; and its number of defects in this release. For each release, they could successfully predict 71%–92% of the 20% most defect-prone files. Graves et al. [11] added software metrics as additional features and found that the number of changes of a file was a stronger predictor than its size.

While the set of code metrics to be used appeared to be quite stable over the past few years, the variety of techniques used to train and to improve defect accuracy is wide. Lately, Kamei et al. [15] presented interesting over and under sampling techniques to prevent heavily unbalanced training sets. Menzies et al. [19] showed that similar techniques can also be used to reduce the size of training sets without loosing predictive accuracy.

Most related to our current approach is the work of Nagappan et al. [21], who used *complexity metrics* to successfully predict the most defect-prone components. Another important related work is the work of Schröter et al. [26], who showed that *imports* could be used as defect predictors. Imports were also used by Neuhaus et al. [22] to predict a subset of defects, namely security vulnerabilities. Dependencies between components, as well as change data, were also used by Nagappan and Ball [20] to predict post-release defects.

In [7], Fenton and Neil criticize potential issues of earlier software defect prediction models. In particular, they point out that earlier studies using linear regression models did not adequately consider the problem of *multi-collinearity.* This can be addressed by principal component analysis, as applied in [21] as well as in the present work.

4. Collecting Defect Data

As laid out in Section 2, post-release fixes are applied only in the maintenance branch. Thus, they can easily be retrieved directly from the SCM system. For each service pack, we could thus exactly tell *which* fixes were applied, and *where* they were applied. For each artifact, we could thus count how many times it was fixed—or, in hindsight, how many defects it still contained as it was released in the last service pack.

In related work on extracting defect densities in open source software, establishing defect counts is listed as challenging: First, one has to establish a mapping between defect database and version archive [10]; second, one must distinguish between fixes and non-corrective changes [28]. Due to the particular organization of all fixes in one single branch, we experienced no such issues.

As shown in Figure 3, each Java *project* at SAP can be decomposed into several *packages*, which in turn is composed of *classes.* In addition to aggregate defect counts per Java class, we also aggregated defect counts per package and per project. (As a single fix may encompass multiple classes, the sum of defect counts across all parts is different from the defect count for a whole.) We thus could tell precisely where the defects were, and how they were distributed.

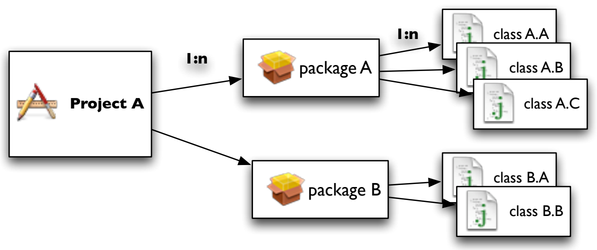

Figure 3. Levels of granularity

5. Collecting Metrics

As stated initially, we wanted to build a *predictor* by relating defect counts to other program features, such as complexity metrics. To extract metrics, we used external tools, working on Java source as well as on Java binaries. Extracting metrics was surprisingly difficult, since we had to keep track of sources and binaries, as well as their interrelations; also, not every source in the version archive and not every binary produced actually ends up in a service pack.

Overall, we collected 78 metrics from our Java system:

Complexity metrics. There is a common assumption that the more complex a component is, the more defect prone it is. To quantify complexity, we used a set of 40 source code complexity metrics such as *total lines of code, McCabe complexity, number of attributes* or *number of methods.* These metrics also included Halstead metrics such as *vocabulary, difficulty,* or *number of distinct operands.*

Additionally, we also used the eight Chidamber and Kemerer byte code metrics [6] such as *weighted methods per class*, *depth of inheritance tree*, or *number of children.*

Dependency metrics. Another common assumption is that the number and kind of dependencies affects the defect-proneness of a component. We used the JDepend tool to measure Martin's eight dependency metrics [17] such as *efferent couplings* or *abstractness* (the ratio between abstract and concrete classes).

Code smell metrics. If a piece of code exhibits code smells and questionable coding style, chances are that it contains actual defects. We used the PMD [2] and FindBugs [1] tools to search for error patterns. We counted the number of PMD warnings for each of the 15 PMD rule categories such as *UnusedCodeRules* or *SecurityCodeGuidelines*, as well as the number of FindBugs warnings for six FindBugs categories such as *Performance* or *Correctness.*

Change metrics. The more frequently a component is changed, the more likely it is to contain a defect introduced during one of these changes. From the SCM system, we extracted for each artifact the *total number of changes* it had gone through.

All these metrics were determined both at the class and package level; for each entity, we computed the sum, the average, and the maximum of each metric (where applicable).

6. Predictor Models

The purpose of the statistical analysis in this paper is to predict defects for software components, but also to reveal defect distributions to better understand failure-prone components. In this section, we explain the different prediction models that we used for our experiments. The analysis was conduced with the freely available tool *GNU R*.[1]

Prediction models are trained with data for which the defect-proneness is known. For the studies in this paper, we used two different training sets (see also Figure 4).

1. To assess *long-term predictions*, we took the eight months after the release of version V1 as training data. We tested our predictions on the eight months after version V2 (denoted as V1→V2 throughout the paper).

2. For *short-term predictions*, we took the two months after the release of version V2 as training data to predict defects in version V3 (denoted as V2→V3).

Our evaluation setting has the advantage that it closely reflects the potential usage of prediction models at SAP: a model is learned from one version of a software, and then applied to predict the defects of a later version of the same software. In contrast, techniques such as data splitting or n-fold cross-validation operate on a single dataset and neglect the temporal aspect.

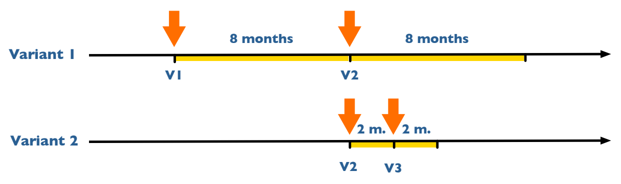

Figure 4. Long-term and short-term prediction.

6.1. Linear Regression

Linear regression is a standard technique for prediction models [30] and can reveal relationships between one dependent variables (here: number of defects, see Section 4)

[1]http://www.r-project.org

and one or more independent variables (metrics, see Section 5). A crucial prerequisite is that independent variables are not inter-correlated with each other. Partly because we used different tools to collect metrics, some metrics showed high correlations in our data set, for example *number of type declarations* and the *total number of classes* within a package. When neglected, such inter-correlations can increase the variability of the dependent variable and thus decrease the quality of predictions. This phenomenon is called *multi-collinearity.*

To remove multi-collinearity, we used *principal component analysis* (PCA). PCA reduces complex datasets by projecting the data points into a subspace with lower dimensionality, while keeping most of the information. The resulting principal components are independent of each other, which satisfies the requirements of linear regression. Specifically for our prediction models, we use the number of defects as the dependent variable and the principal components resulting from PCA as independent variables.

The quality of a linear regression model is expressed using the *coefficient of determination* (R^2). It expresses the proportion of variance in a data set that is accounted for by a statistical model. The R^2 coefficient can take values between 0 and 1, where a higher value implies a higher predictive power. The *adjusted* coefficient of determination ($\overline{R}^2$) expresses the robustness of the model with respect to the number of used regressors.

6.2. Support Vector Machines

A *support vector machine* (SVM; see [30], for example) is a classification method used for pattern recognition, such as recognition of spam e-mail. The data points of the objects to be classified are represented in a vector space. In this space, the SVM fits a multi-dimensional separating hyperplane maximizing the margin between the data sets. The SVM thus clusters objects into groups and can decide which group new objects are in. SVMs can be used for classification as well as for regression.

6.3. Regression vs. Classification

We present the prediction results in two ways: regression and classification.

6.3.1. Regression

A regression analysis predicts actual defect counts for individual components. Such an analysis can be conducted with linear regression as well as with SVMs.

To express the predictive power of a regression, Spearman's *rank correlation coefficient* is widespread and useful. A high positive Spearman correlation indicates a precise prediction regarding both ranking as well as the actual defect counts. A Spearman correlation of -1 means that the ranking was predicted in reverse order. For our purposes, we were most interested in the Spearman correlation for the 5%, 10%, and 20% most defect-prone components, as these would be focused upon in quality assurance allocation based on the prediction.

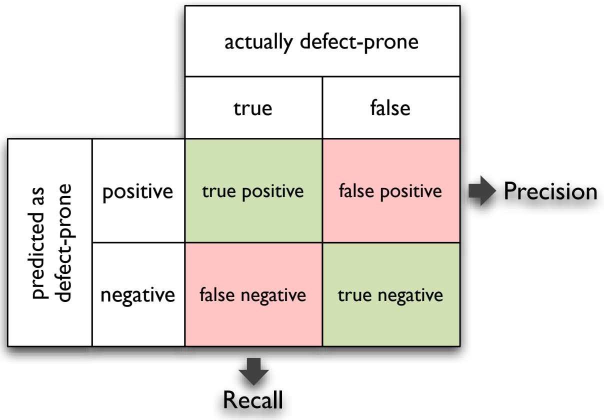

Figure 5. Precision and Recall of a Classification

6.3.2. Classification

As an alternative to regression, one can build predictors that *classify* components whether they are defect-prone or not. For classification, we exclusively used SVMs; as with regression analysis, we used the most 5%, 10%, and 20% most defect-prone components for classification.

To measure the quality of a classification, we use *precision* and *recall,* as shown in Figure 5:

Precision expresses whether the components classified as defect-prone are correctly classified. A value close to 1 means that almost all reported components are indeed defect-prone; all positives are *true positives.*

$$Precision = \frac{true\ positives}{true\ positives + false\ positives}$$

Recall expresses the percentage of defect-prone components that are classified as such. Again, a value close to 1 is the best, and means that almost all defect-prone components are classified as such; all negatives are *true negatives.*

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives}$$

7. Results

In this section, we present the results of our experiments. We first explain the distribution of defects in the studied software system. We then discuss the correlations between metrics and the number of defects; finally, we show the performance of different defect prediction models.

7.1. Defect Distribution

For both package and class level, we observed a Pareto effect. For the eight months after the release V1, we found that 20% of all packages contain more than 70% of all defects. For the eight months after release V2, we found that 20% of packages contain 80% of defects. A similar effect has been observed in other industrial studies. For example, Fenton and Ohlsson report 60% of all defects in the 20% most-defect prone modules [8] and Andersson and Runeson found between 63% and 70% of all defects in the 20% most defect-prone modules [3].

The presence of a Pareto effect indicates that it is worthwhile to prioritize resources for quality assurance (QA), as long as one knows the most defect-prone parts of a software system.[2] Managers often rely on their experience to identify these parts. Having access to prediction models helps them to cross-check their judgments, explain the decisions, and to partly automate the process of QA prioritization.

The Pareto effect additionally provides an upper bound for the maximal possible recall of prediction models. On the SAP data, any prediction of the 20% most defect-prone components, will catch at most 70% (for V1) or 80% of the defects (for V2).

7.2. Correlation

Before building prediction models, we checked whether there is a relation between the number of defects and the metrics computed in Section 5. In this section, we report the Spearman correlations between defects and metrics. Correlation values close to 1 indicate that two variables are almost perfectly related; values close to -1 indicate a strong relation, but in opposite direction; finally, values close to 0 indicate that the two variables are independent.

We first correlated the number of defects in components across different versions of the system. This answers the questions "Are the most defect-prone components in V1 the same as in V2?" For this analysis, we considered only components that existed in both releases. In Table 1, we show the results on package and class level.

[2]In contrast, if defects would be distributed equally, there would be no clear benefit in prioritizing QA resources.

	V1→V2		V2→V3	
	Packages	Classes	Packages	Classes
Overall	0.451	0.309	0.471	0.412
Project P1	0.464	0.332	0.328	0.181
Project P2	0.641	0.450	0.491	0.277
Project P3	0.361	0.350	0.519	0.382
Project P4	0.664	0.440	0.413	0.222
Project P5	0.529	0.191	0.366	0.407
Project P6	0.679	0.374	0.388	0.225

Table 1. Spearman correlation between number of defects for subsequent versions.

In most cases, the correlations are stronger on package level than on class level; as a consequence one can expect better prediction results for packages than for classes. Furthermore, there is a difference between the correlations for V1→V2 and for V2→V3, which might be because of the different time intervals (eight months vs two months). Most importantly, the correlations are only weak to medium (0.3–0.6), which indicates that the components that are most defect-prone changes are not exactly the same in two different versions.

A prerequisite for making reliable predictions is that the independent variables (here: metrics) are correlated with the dependent variable (here: number of defects). In Table 2, we show the correlations between a selection of metrics and the number of defects—for the entire software system ("Overall"), Track T1, and Project P3 on package level and for Project P4 on class level. Out of the 78 collected metrics, we include only the most promising metrics (with the highest correlations) in Table 2.

In almost all cases, the metrics correlated positively with the number of defects, i.e., the higher the value of a metrics, the more defects a component is likely to have. However, for the packages of the entire system only the number of changes (*N_p4changes*) had a correlation of above 0.400. If we focus our analysis on single tracks or packages (for example Track T1 or Project P3), the correlations increase for several metrics to values above 0.500. On class level the correlations are generally lower; for example, for Project P4 all correlations are below 0.400, except for the number of changes and number of violated *ControversialRules*.

7.3 Regression Analysis

In this section, we present the results from defect prediction with linear regression models and principal component analysis. Table 3 shows general information about the models which we built with Variant 1 (V1→V2, eight months of defect data) and Variant 2 (V2→V3, 2 months of defect

Metric		Level: Package			Class
		Overall	T1	P3	P4
Complexity Metrics					
TLOC	sum	0.281	**0.565**	**0.583**	0.377
Total Lines Of Code	max	0.274	**0.518**	**0.587**	–
MPC	sum	0.272	**0.561**	**0.592**	0.304
Method Lines of Code	max	0.268	**0.529**	**0.582**	0.273
UWCS	sum	0.264	**0.504**	**0.526**	0.396
Unweighted Class Size	max	0.257	0.462	**0.561**	–
VG	sum	0.262	**0.554**	**0.583**	0.299
McCabe Complexity	max	0.264	**0.512**	**0.588**	0.261
NBD	sum	0.256	**0.545**	**0.564**	0.297
Nested Block Depth	max	0.255	0.497	**0.585**	0.265
N_casts	sum	0.253	**0.535**	**0.586**	0.317
Number of Casts	max	0.240	0.474	**0.582**	0.295
N_comments	sum	0.312	**0.521**	**0.557**	0.363
Number of Comments	max	0.310	0.485	**0.582**	–
Dependency Metrics					
Ce		0.316	**0.542**	**0.608**	–
Efferent Couplings					
Chidamber and Kemerer Metrics					
CBO	sum	0.307	**0.572**	**0.586**	0.299
Coupling Between Objs	max	0.313	**0.554**	**0.593**	–
RFC	sum	0.269	**0.550**	**0.581**	0.368
Response for Class	max	0.278	**0.509**	**0.595**	–
WMC	sum	0.246	0.485	**0.514**	0.377
Weigted Mthds per Cls	max	0.245	0.443	**0.523**	–
LCOM	sum	0.233	0.460	**0.500**	0.268
Lack of Cohesion	max	0.229	0.437	0.491	–
Change Metrics					
N_p4changes	sum	0.393	**0.503**	0.308	**0.403**
Number of Changes	max	**0.402**	0.379	0.240	–
Code Smell Metrics					
DesignRules	sum	0.262	**0.527**	**0.557**	0.264
	max	0.254	0.479	**0.578**	–
Optimization-	sum	0.271	**0.563**	**0.546**	0.315
Rules	max	0.270	**0.526**	**0.535**	–
Controversial-	sum	0.266	**0.532**	**0.570**	**0.401**
Rules	max	0.261	**0.503**	**0.568**	–
NamingRules	sum	0.264	0.493	**0.531**	0.381
	max	0.259	0.454	**0.521**	–
CodeSize-	sum	0.282	**0.523**	**0.573**	0.370
Rules	max	0.281	0.496	**0.572**	–

Table 2. Spearman correlations between metrics and number of defects for the entire Java system, track T1, and projects P3 and P4.

data) for the packages of the entire software system, Track T1 and Project P2. The quality of each model is reflected by the R^2-value; as a rule of thumb, trustworthy models should have values above 0.500, which means that they can explain the majority of the training data. For the entire software system, both variants fail to meet this criteria, only if one learns a model for a single track or project, the R^2-values reach an acceptable level.

Model		R^2	adj. R^2	p-value
Overall	V1→V2	0.341	0.336	$2.2e^{-16}$
	V2→V3	0.244	0.238	$2.2e^{-16}$
Track T1	V1→V2	**0.681**	**0.637**	$2.2e^{-16}$
	V2→V3	0.436	0.365	$2.2e^{-16}$
Project P2	V1→V2	**0.668**	**0.583**	$2.2e^{-16}$
	V2→V3	**0.523**	0.420	$6.9e^{-16}$

Table 3. Prediction models with linear regression on package level. The selected principal components explain 99% of the cumulated variance.

Note that the R^2-values only describe how well a model fits its training data. In particular, they do not make any statements about the testing data and high R^2-values do not automatically result in good predictions. To measure how well a model performs on the testing data, we use Spearman correlation and hit rates. The Spearman correlations for all packages and the top 5%, 10%, and 20% of most-defect prone packages are listed in Table 4. In addition, we report the hit rate for the 5%, 10%, and 20% of most-defect prone packages. The hit rate describes how many of the observed top n% of most defect-prone packages are actually in the predicted top n%.

For the entire software system, the Spearman correlation values range between 0.3 and 0.5, the hit rate is between 45% and 55%. When focussing on tracks and projects, the results improve: correlation values rise up to 0.7 and the hit rate increases above 60%.

7.4 Prediction on Class Level

We repeated our experiments on class level for Project P4 using Variant 1 (V1→V2). To build regression models we used linear regression with principal component analysis and a support vector machine. We also used support vector machines to build a classification model. The linear regression model had an acceptable R^2- value above 0.5. Table 5 shows the results of this experiment.

For the regression, the results for linear regression and support vector machines were similar. They had Spearman correlations between 0.4 and 0.5 and hit rates around 0.5.

		V1→V2		V2→V3	
		Spearman	Hit Rate	Spearman	Hit Rate
Overall	total	0.464	–	0.376	–
	5%	0.409	45.8%	0.323	47.7%
	10%	0.425	51.8%	0.442	48.3%
	20%	0.507	54.7%	0.586	46.7%
Track T1	total	0.596	–	0.554	–
	5%	0.430	47.6%	0.648	45.4%
	10%	0.679	39.5%	0.543	62.7%
	20%	0.283	62.7%	0.487	62.7%
Project P2	total	0.603	–	0.425	–
	5%	0.735	41.6%	0.341	30.7%
	10%	0.456	60.0%	0.156	42.3%
	20%	0.420	64.0%	0.171	52.9%

Table 4. Prediction results of linear regression on package level for the entire system.

(a) Regression

		Spearman	Hit Rate
LinReg	top 5%	0.508	53.6%
	10%	0.480	49.3%
	20%	0.455	44.8%
SVM	top 5%	0.515	46.3%
	10%	0.428	46.9%
	20%	0.431	46.0%

(b) Classification (SVM)

	Precision	Recall
top 5%	0.489	0.575
10%	0.432	0.532
20%	0.542	0.493

Table 5. Results for Project P4 on class level.

For classification with support vector machines, both precision and recall were roughly 0.50. This means that every second file predicted as defect-prone had actually defects, and every second file with defects was correctly predicted as defect-prone.

7.5. Prediction for New Components

We also tested whether the models can predict defects for new components. For this experiment, we trained a model from version V1 of Track T2 and tested the model on the new packages in version V2.[3] Table 6 shows the results for regression and classification, again with linear regression and support vector machines.

[3] We identified new components based on their names, i.e., the packages that exist in version V2 but not in V1. We did not account for refactorings.

(a) Regression

		Spearman	Hit Rate
LinReg	top 5%	0.057	33.3%
	10%	-0.143	50.0%
	20%	0.200	52.1%
SVM	top 5%	-0.086	33.3%
	10%	-0.108	50.0%
	20%	0.112	47.8%

(b) Classification

	Precision	Recall
top 5%	0.222	0.400
10%	0.352	0.666
20%	0.289	0.785

Table 6. Results for new packages in Track T2.

For both linear regression and support vector machines, the Spearman correlations are weak and in some cases even negative (-0.1 to 0.2); the hit rates reach 33.3%–52.1%. For classification, the precision and values are low compared to the results for the previous models. Thus it is unlikely to get reliable predictions for new components.

7.6. Prediction with Metrics vs. Dependencies

In a last experiment, we compared classification with metrics data against classification with dependency data (as proposed by Schröter et al. [26]). For the classification, we used a support vector machine with Variant 1 (V1→V2).

Table 7 shows the results for Track T1, Project P1, Project P2, and Project P6. In some cases, we can observe high values for precision (>0.7) and recall (>0.7); however, both models fail to reach good predictions in all situations. It is noteworthy that the results are very similar, thus we cannot recommend which approach should be preferred over the other.

8. Lessons Learned

Software archives such as change and defect histories provide lots of valuable data about the quality of the product. As quality assurance managers, we of course have an idea of what the most defect-prone components or what the most risky components are. However, the abundance of facts extracted from software archives allows us to look at a far more complete and detailed picture. Quoting one of the SAP team leaders: "You always remember the extremes, the

(a) Track T1

		Precision	Recall
Metrics	5%	0.750	0.272
	10%	0.521	0.369
	20%	0.428	0.552
Dependencies	5%	0.708	0.265
	10%	0.568	0.328
	20%	0.430	0.434

(b) Project P1

		Precision	Recall
Metrics	5%	0.227	0.322
	10%	0.314	0.493
	20%	0.412	0.600
Dependencies	5%	0.277	0.384
	10%	0.344	0.505
	20%	0.481	0.509

(c) Project P2

		Precision	Recall
Metrics	5%	0.866	0.433
	10%	0.477	0.583
	20%	0.488	0.764
Dependencies	5%	0.818	0.360
	10%	0.648	0.558
	20%	0.606	0.465

(d) Project P6

		Precision	Recall
Metrics	5%	0.363	0.533
	10%	0.419	0.433
	20%	0.531	0.539
Dependencies	5%	0.583	0.466
	10%	0.392	0.366
	20%	0.469	0.492

Table 7. Classification results of models using metrics vs. dependencies as input features.

best and the worst in quality, and you learn from them. This data, however, gives you all the shades of gray, for hundreds of projects, and thus far more than any individual could ever remember or summarize." This was our first lesson learned:

Software archives are reliable, easily accessible, and complete source for defect data.

As it came to *predicting* defects, we were generally satisfied with the results; yet, we felt we could have achieved more. The study of Nagappan et al. [21] showed a much higher predictive power for complexity metrics and a number of well-known Microsoft products such as Internet Explorer; likewise, the study of Schröter et al. [26] showed better prediction results for imports and the Eclipse programming environment. We had expected that by incorporating the best features from both these studies, we would obtain similar predictive power, but this was not the case.

What is it that makes our experience different from the Microsoft and Eclipse environments? For us, a crucial point is that all studies mentioned above predicted defects *within a single project.* In fact, one of the key results of Nagappan et al. [21] was that predictors obtained from one project could only rarely be applied to another project. In contrast, our study encompassed and summarized Java code from several hundred large-scale, individual projects, which all differ in their domain, their maturity, their criticality, their team members, their testing, their reviewing—but few of these features, if any, actually end up in code as measurable quantities. A piece of code will have considerably fewer defects if it was thoroughly tested or reviewed. Yet, one cannot tell this from the code alone—one needs all the context.

Since our predictors eventually predict defect counts based on similarity to known artifacts with known quality, we should strive to take more of these process and quality assurance features into account. This is our second lesson learned:

Defects have many sources, and code is just one of them.

9. Conclusion and Future Work

As shown in Section 7, the predictive power increases as soon as one stays within a single project, confirming the results of Nagappan et al. [21] at Microsoft. We are currently investigating methods to create *clusters* of similar projects, with the aim of making precise predictions within these clusters. One important challenge is to identify those features by which clusters should be formed. This, however, is part of the general research question: Which features are there that allow us to predict defect counts? We are not sure whether we will ever be able to answer this question in a short sentence. What we know, however, is that any prediction of this kind will be self-defeating. In fact, we *hope* it will be self-defeating—because once we know where the defects will be, we will do our best to obliterate them.

Acknowledgments. We thank Günther Limböck of SAP Quality Governance & Production (QGP) for enabling this study and his ongoing support. Jürgen Heymann (QGP) provided lots of valuable insights during the study.

References

[1] FindBugs. http://findbugs.sourceforge.net/.

[2] PMD. http://pmd.sourceforge.net/.

[3] C. Andersson and P. Runeson. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Trans. Software Eng.*, 33(5):273–286, 2007.

[4] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering*, 22(10):751–761, 1996.

[5] V. R. Basili and B. T. Perricone. Software errors and complexity: an empirical investigation. *Commun. ACM*, 27(1):42–52, 1984.

[6] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *OOPSLA*, 1991.

[7] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25(5):675–689, 1999.

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

[9] N. E. Fenton and S. Pfleeger. *Software Metrics: A Rigorous & Practical Approach (second edition)*. International Thompson Press, 1996.

[10] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering*, pages 90–99, Victoria, B.C., Canada, November 2003. IEEE Computer Society Press.

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

[12] B. Gupta. A critique of cohesion measures in the object-oriented paradigm, 1997.

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

[14] B. Henderson-Sellers, L. L. Constantine, and I. M. Graham. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 3:143–158, 1996.

[15] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. ichi Matsumoto. The effects of over and under sampling on fault-prone module detection. *Empirical Software Engineering and Measurement, International Symposium on*, 0:196–204, 2007.

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

[17] R. Martin. OO Design Quality Metrics—An Analysis of Dependencies. In *position paper, Proc. Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA*, volume 94, 1994.

[18] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, Vol. 2(4):308–320, 1976.

[19] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang. Implications of ceiling effects in defect predictors. In *PROMISE '08: Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 47–54, New York, NY, USA, 2008. ACM.

[20] N. Nagappan and T. Ball. Explaining failures using software dependences and churn metrics. In *Proceedings of the 1st International Symposium on Empirical Software Engineering and Measurement*, 2007.

[21] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *28th International Conference on Software Engineering (ICSE)*, November 2005.

[22] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.

[23] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 55–64, New York, NY, USA, 2002. ACM Press.

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

[25] A. Schröter, T. Zimmermann, R. Premraj, and A. Zeller. If your bug database could talk.... In *Proceedings of the 5th International Symposium on Empirical Software Engineering, Volume II: Short Papers and Posters*, pages 18–20, September 2006. Available at `http://www.st.cs.uni-sb.de/softevo/`.

[26] A. Schröter, T. Zimmermann, and A. Zeller. Predicting component failures at design time. In *Proceedings of the 5th International Symposium on Empirical Software Engineering*, pages 18–27, September 2006.

[27] J. Śliwerski, T. Zimmermann, and A. Zeller. Hatari: Raising risk awareness (research demonstration). In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 107–110. ACM, September 2005.

[28] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[29] D. Čubranić and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society.

[30] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, second edition, June 2005.

[31] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings of the First International Workshop on Mining Software Repositories*, pages 2–6, May 2004.

[32] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572. IEEE Computer Society, May 2004.