Mining Metrics to Predict Component Failures

Nachiappan Nagappan
Microsoft Research
Redmond, Washington
nachin@microsoft.com

Thomas Ball
Microsoft Research
Redmond, Washington
tball@microsoft.com

Andreas Zeller*
Saarland University
Saarbrücken, Germany
zeller@cs.uni-sb.de

ABSTRACT

What is it that makes software fail? In an empirical study of the post-release defect history of five Microsoft software systems, we found that failure-prone software entities are statistically correlated with code complexity measures. However, there is no single set of complexity metrics that could act as a universally best defect predictor. Using principal component analysis on the code metrics, we built regression models that accurately predict the likelihood of post-release defects for new entities. The approach can easily be generalized to arbitrary projects; in particular, predictors obtained from one project can also be significant for new, similar projects.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*version control.* D.2.8 **[Software Engineering**]: Metrics—*Performance measures*, *Process metrics*, *Product metrics.* D.2.9 **[Software Engineering]**: Management—*Software quality assurance (SQA)*

General Terms

Measurement, Design, Reliability.

Keywords

Empirical study, bug database, complexity metrics, principal component analysis, regression model.

1. INTRODUCTION

During software production, software quality assurance consumes a considerable effort. To raise the effectiveness and efficiency of this effort, it is wise to direct it to those which need it most. We therefore need to identify those pieces of software which are the most likely to fail—and therefore require most of our attention.

One source to determine failure-prone pieces can be their *past:* If a software entity (such as a module, a file, or some other component) was likely to fail in the past, it is likely to do so in the future. Such information can be obtained from *bug databases*—especially when coupled with version information, such that one can map failures to specific entities. However, accurate predictions require a long failure history, which may not exist for the entity at hand; in fact, a long failure history is something one would like to avoid altogether.

A second source of failure prediction is the *program code* itself: In particular, *complexity metrics* have been shown to correlate with defect density in a number of case studies. However, indiscriminate use of metrics is unwise: How do we know the chosen metrics are appropriate for the project at hand?

In this work, we apply a *combined* approach to create accurate failure predictors (Figure 1): We mine the archives of major software systems in Microsoft and map their post-release failures back to individual entities. We then compute standard complexity metrics for these entities. Using principal component analysis, we determine the combination of metrics which best predict the failure probability for new entities within the project at hand. Finally, we investigate whether such metrics, collected from failures in the past, would also good predictors for entities of other projects, including projects be without a failure history.

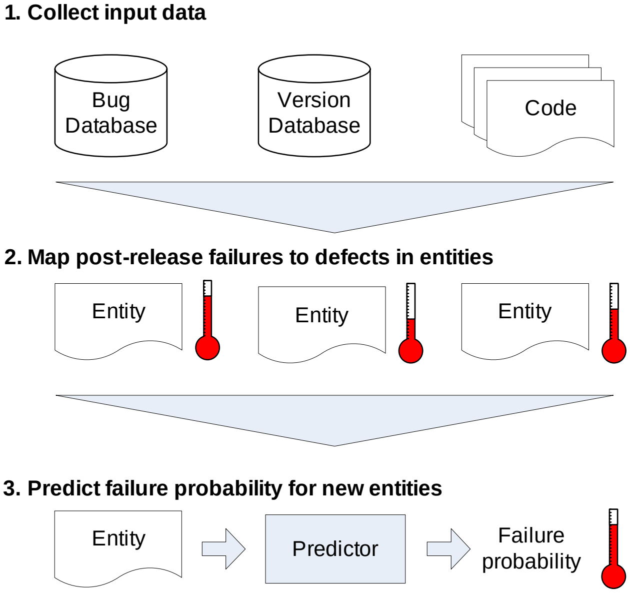

Figure 1. After mapping historical failures to entities, we can use their complexity metrics to predict failures of new entities.

ICSE'06, May 20–28, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005...$5.00.

* Andreas Zeller was a visiting researcher with the Testing, Verification and Measurement Group, Microsoft Research in the Fall of 2005 when this work was carried out.

This paper is organized in a classical way. After discussing the state of the art (Section 2), we describe the design of our study (Section 3). Our results are reported in Section 4. In Section 5, we discuss the lessons learned, followed by threats to validity (Section 6). Section 7 closes with conclusion and future work.

2. RELATED WORK

2.1 Defects and Failures

In this paper, we use the term *defect* to refer to an error in the source code, and the term *failure* to refer to an observable error in the program behavior. In other words, every failure can be traced back to some defect, but a defect need not result in a failure[1].

Failures can occur *before* a software release, typically during software testing; they can also occur *after* a release, resulting in failures in the field. If a defect causes a pre-release failure, we call it a *pre-release defec*t; in contrast, a *post-release defect* causes a failure after a release.

It is important not to confuse these terms. In particular, a study conducted by Adams [1] found that only 2% of the defects in eight large-scale software systems lead to a mean time to failure of less than 50 years—implying that defect density cannot be used to assess reliability in terms of failures [10]. Only one study so far has found that a large number of fixed pre-release defects raised the probability of post-release failures [5].

For the user, only post-release failures matter. Therefore, our approach is exclusively concerned with *post-release defects*, each of them uncovered by at least one failure in the field.

2.2 Complexity Metrics

Over the years, a number of software metrics have been proposed to assess software effort and quality [11]. These "traditional" metrics were designed for imperative, non-object-oriented programs. The object-oriented metrics used in our approach were initially suggested by Chidamber and Kemerer [8]. Basili et al. [3] were among the first to validate these metrics. In an experiment with eight student teams, they found that OO metrics appeared to be useful for predicting defect density. The study by Subramanyam and Krishnan [21] presents a survey on eight more empirical studies, all showing that OO metrics are significantly associated with defects. In contrast to this existing work, we do not predict pre-release defect density, but post-release defects, and hence actual failures of large-scale commercial software.

Empirical evidence that metrics can predict *post-release defects* (rather than pre-release defects) and thus *post-release failures* is scarce. Binkley and Schach [4] found that their coupling dependency metric outperformed several other metrics when predicting failures of four academia-developed software systems. Ohlsson and Alberg [18] investigated a number of traditional design metrics to predict modules that were prone to failures during test as well as within operation. They found that 20% of the modules predicted as most failure-prone would account for 47% of the failures. Their problem was, however, that "it was not possible to draw generalizable conclusions on the strategy for selecting specific variables for the model"—which is why we rely on failure history to select the most suitable metrics combination.

2.3 Historical Data

Hudepohl et al. [13] successfully predicted whether a module would be defect-prone or not by combining metrics and historical data. Their approach used *software design metrics* as well as *reuse information*, under the assumption that new or changed modules would have a higher defect density. In our approach, historical data is used to select appropriate metrics first, which can then be applied to arbitrary entities; also, we focus on *post-release* rather than pre-release defects.

Ostrand et al. [19] used historical data from two large software systems with up to 17 releases to predict the files with the highest defect density in the following release. For each release, the 20% of the files with the highest predicted number of defects contained between 71% and 92% of the defects being detected. Again, our approach focuses on *post-release* rather than pre-release defects; it also goes beyond the work of Ostrand et al. by not only identifying the most failure-prone entities, but also determining their common features, such that entities of other projects can be assessed.

2.4 Mining Software Repositories

In recent years, researchers have learned to exploit the vast amount of data that is contained in software repositories such as version and bug databases [16, 17, 19, 22]. The key idea is that one can map *problems* (in the bug database) to *fixes* (in the version database) and thus to those locations in the code that caused the problem [9, 12, 20]. This mapping is the base of automatically associating metrics with post-release defects, as described in this work.

2.5 Contributions

This work extends the state of the art in four ways:

1. It reports on how to systematically build predictors for post-release defects from failure history found in the field by customers.
2. It investigates whether object-oriented metrics can predict post-release defects from the field.
3. It analyzes whether predictors obtained from one project history are applicable to other projects.
4. It is one of the largest studies of commercial software—in terms of code size, team sizes, and software users.

3. STUDY DESIGN

3.1 Researched Projects

The goal of this work was to come up with failure predictors that would be valid for a wide range of projects. For this purpose, we analyzed the project history of five major Microsoft project components, listed in Table 1.

These projects were selected to form a wide range of product types. All of them have been released as individual products; they thus do not share code. All use object-oriented programming languages like C++ or C#. Finally, all of these projects are large—not only in terms of code or team size (> 250 engineers), but also in terms of user base. DirectX, for instance, is part of the Windows operating system, which has an estimated 600 million users. (The team sizes are normalized in Table 1 below).

[1] The term *fault* is usually used as a synonym for *defects*, but some authors (e.g. [18]) use it as a synonym for *failures*. In this paper, we thus avoid the term.

Table 1. Projects researched

Project	Description	Components	Code size	Team size
Internet Explorer 6	Web browser	HTML rendering	511 KLOC	14.3X
IIS W3 Server core	Web server	Application loading	37 KLOC	6.3X
Process Messaging Component	Application communication and networking	all	147 KLOC	3.4X
DirectX	Graphics library	all	306 KLOC	18.5X
NetMeeting	A/V Conferencing	all	109 KLOC	X

Let us now give a high level outline of each project.

- *Internet Explorer 6 (IE6)* is the standard Web browser shipped with most versions of Microsoft Windows. Since only a part of IE6 is written in object-oriented languages, we focus upon the HTML rendering part as an object-oriented component.
- *Internet Information Services (IIS)* is the standard Web server shipped with Microsoft Server. Again, we focus on an object-oriented component responsible for loading applications into IIS.
- *Process Messaging Component* is a Microsoft technology that enables applications running at different times to communicate across heterogeneous networks and systems that may be temporarily offline.
- *Microsoft DirectX* is an advanced suite of multimedia application programming interfaces (APIs) built into Microsoft Windows. DirectX is a Windows technology that enables higher performance in graphics and sound when users are playing games or watching video on their PC.
- Microsoft *NetMeeting* is used for both voice and messaging between different locations.

In the remainder of the paper, we shall refer to these five projects as projects A, B, C, D, and E. For reasons of confidentiality, we do not disclose which letter stands for which project.

3.2 Failure Data

Like any company, Microsoft systematically records all problems that occur during the entire product life cycle. In this study, we were interested in *post-release failures*—that is, failures that occurred in the field within six months after the initial release. For each of the projects, we determined the last release date, and extracted all problem reports that satisfied three criteria:

- The problem was submitted by customers in the field,
- The problem was classified as non-trivial (in contrast to requests for enhancement), and
- The problem was fixed in a later product update.

The location of the fix gave us the location of the *post-release defect*. We thus could assign each entity the number of post-release defects. The likelihood of a post-release defect is also what we want to predict for new entities—that is, entities without a failure history. Since each post-release defect is uncovered by a post-release failure, predicting the likelihood of a post-release defect in some entity is equivalent to predicting the likelihood of at least one post-release failure associated with this entity.

3.3 Metrics Data

For each problem report, Microsoft records fix locations in terms of *modules*—that is, a binary file within Windows, built from a number of source files. Thus, we chose *modules* as the entities for which we collected the failure data and for which we want to predict the failure-proneness.

For each of the modules, we computed a number of source code metrics, described in the left half of Table 3. These metrics apply to a module *M*, a function or method *f*(), and a class *C*, respectively.

Here is some additional information on the metrics in Table 3:

- The *Arcs* and *Blocks* metrics refer to a function's control flow graph, which is also the base for computing McCabe's cyclomatic complexity (separately measured as *Complexity*).
- The *AddrTakenCoupling* metric counts the number of instances where the address of some global variable is taken in a function—as in the C++ constructs `int *ref = &globalVar` or `int& ref = globalVar`.
- The *ClassCoupling* metrics counts the number of classes coupled to a class *C*. A class is "coupled" to *C* if it is a type of a class member variable, a function parameter, or a return type in *C*; or if it is defined locally in a method body, or if it is an immediate superclass of *C*. Each class is only counted once.

In order to have all metrics apply to modules, we summarized the function and class metrics across each module. For each function and class metric *X*, we computed the *total* and the *maximum* number per module (henceforth denoted as *TotalX* and *MaxX*, respectively). As an example, consider the *Lines* metric, counting the number of executable lines per function. The *MaxLines* metric indicates the length of the largest function in *M*, while *TotalLines,* the sum of all *Lines,* represents the total number of executable lines in *M*. Likewise, *MaxComplexity* stands for the most complex function found in *M*.

3.4 Hypotheses

So, what do we do with all these metrics? Our hypotheses to be researched are summarized in Table 2:

Table 2. Research hypotheses

	Hypothesis
H_1	Increase in complexity metrics of an entity *E* correlates with the number of post-release defects of *E*.
H_2	There is a common subset of metrics for which H_1 applies in all projects.
H_3	There is a combination of metrics which significantly predicts the post-release defects of new entities within a project.
H_4	Predictors obtained using H_3 from one project also predict failure-prone entities in other projects.

Table 3. Metrics and their correlations with post-release defects. For each module *M*, we determine how well the metrics correlate with *M*'s post-release defects. Bold values indicate significant correlation.

Metric	Description		Correlation with post-release defects of *M*				
			A	B	C	D	E
Module metrics — correlation with metric in a module *M*							
Classes	# Classes in *M*		**0.531**	**0.612**	**0.713**	0.066	0.438
Function	# Functions in *M*		0.131	**0.699**	**0.761**	0.104	**0.531**
GlobalVariables	# global variables in *M*		0.023	**0.664**	**0.695**	0.108	0.460
Per-function metrics — correlation with maximum and sum of metric across all functions *f*() in a module *M*							
Lines	# executable lines in *f*()	Max	-0.236	**0.514**	**0.585**	**0.496**	**0.509**
		Total	0.131	**0.709**	**0.797**	0.187	**0.506**
Parameters	# parameters in *f*()	Max	-0.344	0.372	**0.547**	0.015	0.346
		Total	0.116	**0.689**	**0.790**	0.152	0.478
Arcs	# arcs in *f*()'s control flow graph	Max	-0.209	0.376	**0.587**	0.527	0.444
		Total	0.127	**0.679**	**0.803**	0.158	**0.484**
Blocks	# basic blocks in *f*()'s control flow graph	Max	-0.245	0.347	**0.585**	0.546	0.462
		Total	0.128	**0.707**	**0.787**	0.158	0.472
ReadCoupling	# global variables read in *f*()	Max	-0.005	**0.582**	**0.633**	0.362	0.229
		Total	-0.172	**0.676**	**0.756**	0.277	0.445
WriteCoupling	# global variables written in *f*()	Max	0.043	**0.618**	**0.392**	0.011	0.450
		Total	-0.128	**0.629**	**0.629**	0.230	0.406
AddrTakenCoupling	# global variables whose address is taken in *f*()	Max	0.237	**0.491**	**0.412**	0.016	0.263
		Total	0.182	**0.593**	**0.667**	0.175	0.145
ProcCoupling	# functions that access a global variable written in *f*()	Max	-0.063	**0.614**	**0.496**	0.024	0.357
		Total	0.043	**0.562**	**0.579**	0.000	0.443
FanIn	# functions calling *f*()	Max	0.034	**0.578**	**0.846**	0.037	**0.530**
		Total	0.066	**0.676**	**0.814**	0.074	**0.537**
FanOut	# functions called by *f*()	Max	-0.197	0.360	**0.613**	0.345	0.465
		Total	0.056	**0.651**	**0.776**	0.046	**0.506**
Complexity	McCabe's cyclomatic complexity of *f*()	Max	-0.200	0.363	**0.594**	0.451	**0.543**
		Total	0.112	**0.680**	**0.801**	0.165	**0.529**
Per-class metrics — correlation with maximum and sum of metric across all classes *C* in a module *M*							
ClassMethods	# methods in *C* (private / public / protected)	Max	0.244	**0.589**	**0.534**	0.100	0.283
		Total	**0.520**	**0.630**	**0.581**	0.094	0.469
InheritanceDepth	# of superclasses of *C*	Max	**0.428**	**0.546**	0.303	0.131	0.323
		Total	**0.432**	**0.606**	**0.496**	0.111	0.425
ClassCoupling	# of classes coupled with *C* (e.g. as attribute / parameter / return types)	Max	**0.501**	**0.634**	**0.466**	-0.303	0.264
		Total	**0.547**	**0.598**	**0.592**	-0.158	0.383
SubClasses	# of direct subclasses of *C*	Max	0.196	**0.502**	**0.582**	-0.207	0.387
		Total	0.265	**0.560**	**0.566**	-0.170	0.387

As a first step, we examine whether there are any significant correlations between complexity metrics and post-release defects (H_1). We then want to find whether there is some common subset of these metrics that is correlated with post-release defects across different projects (H_2). As a third step, we evaluate whether we can predict the likelihood of post-release defects in new entities by combining multiple metrics (H_3). Finally, we evaluate whether predictors obtained from one project are also good predictors of failure-proneness for another project (H_4).

4. RESULTS

Let us now discuss the results for the four hypotheses. Each hypothesis is discussed in its individual section.

4.1 Do complexity metrics correlate with failures in the field?

To investigate our initial hypothesis H_1, we determined the correlation between the complexity metrics (Section 3.3) for each module M with the number of post-release defects (Section 3.2). The resulting standard Spearman correlation coefficients[2] are shown in Table 3. Correlations that are significant at the 0.05 level is shown in bold; the associated metrics thus correlate with the number of post-release defects. For instance, in project A, the higher the number of classes in a module (*Classes*), the larger the number of post-release defects (correlation 0.531); other correlating metrics include *TotalClassMethods*, both *InheritanceDepth* and both *ClassCoupling* measures. Clearly, for project A, the more classes we have in a module, the higher its likelihood of post-release defects. However, none of the other metrics such as *Lines* correlate, implying that the length of classes and methods has no significant influence on post-release defects.

Projects B and C tell a different story: Almost all complexity metrics correlate with post-release defects. In project D, though, only the *MaxLines* metric correlates with post-release defects, meaning the maximum length of a function within a module. Why is it that in project B and C, so many metrics correlate, and in project D, almost none? The reason lies within the project nature itself, or more precisely within its process: The team of project D routinely uses metrics like the ones above to identify potential complexity traps, and refactors code pieces which are too complex. This becomes evident when looking at the distribution of post-release defects across the modules: In project D, the distribution is much more homogeneous than in project B or C, where a small number of modules account for a large number of post-release defects. These modules also turn out to be the more complex ones—which is what makes all the metrics correlate in B and C.

Nonetheless, one should note that we indeed found correlating metrics for each project. This confirms our hypothesis H_1:

For each project, we can find a set of complexity metrics that correlates with post-release defects—and thus failures.

4.2 Is there a single set of metrics that predicts post-release defects in all projects?

As already discussed, each of the projects comes with its own set of predictive metrics. It turns out that there is not a single metric that would correlate with post-release defects in all five projects.

All in all, this rejects our hypothesis H_2, which has a number of consequences. In particular, this means that it is unwise to use some complexity metric and assume the reported complexity would imply anything—at least in terms of post-release defects. Instead, correlations like those shown in Table 3 should be used to *select* and *calibrate* metrics for the project at hand, which is what we shall do in the next steps.

There is no single set of metrics that fits all projects.

4.3 Can we combine metrics to predict post-release defects?

If there is no universal metric to choose from, can we at least exploit the failure history and its correlation with metrics? Our basic idea was to build predictors that would hold *within a project.* We would combine the individual metrics, weighing the metrics according to their correlations as listed in Table 3.

However, one difficulty associated with combining several metrics is the issue of *multicollinearity*. Multicollinearity among the metrics is due to the existence of inter-correlations among the metrics. In project A, for instance, the *Classes, InheritanceDepth, TotalMethods,* and *ClassCoupling* metrics not only correlate with post-release defects, but they also strongly correlated with each other. Such an inter-correlation can lead to an inflated variance in the estimation of the dependent variable—that is, post-release defects.

To overcome the multicollinearity problem, we used a standard statistical approach, namely *principal component analysis* (PCA) [14]. With PCA, a smaller number of uncorrelated linear combinations of metrics that account for as much sample variance as possible are selected for use in regression (linear or logistic). These principal components are independent and do not suffer from multicollinearity.

We extracted the principal components for each of the five projects that account for a cumulative sample variance greater than 95%. Table 4 gives an example: After extracting five principal components, we can account for 96% of the total variance in project E. Therefore, five principal components suffice.

Table 4. Extracted principal components for project E

Principal Component	Initial Eigenvalues		
	Total	% of Variance	Cumulative %
1	25.268	76.569	76.569
2	3.034	9.194	85.763
3	2.045	6.198	91.961
4	.918	2.782	94.743
5	.523	1.584	96.327

[2] The Spearman rank correlation is a commonly-used robust correlation technique [11] because it can be applied even when the association between elements is non-linear.

Table 5. Regression models and their explanative power

Project	Number of principal components	% cumulative variance explained	**R^2**	Adjusted R^2	F - test
A	9	95.33	**0.741**	0.612	5.731, $p < 0.001$
B	6	96.13	**0.779**	0.684	8.215, $p < 0.001$
C	7	95.34	**0.579**	0.416	3.541, $p < 0.005$
D	7	96.44	**0.684**	0.440	2.794, $p < 0.077$
E	5	96.33	**0.919**	0.882	24.823, $p < 0.0005$

Using the principal components as the independent variable and the post-release defects as the dependent variable, we then built multiple regression models. We thus obtained a predictor that would take a new entity (or more precisely, the values of its metrics) and come up with a *failure estimate.* The regression models built using all the data for each project are characterized in Table 5. For each project, we present the *R^2 value* which is the ratio of the regression sum of squares to the total sum of squares. As a ratio, it takes values between 0 and 1, with larger values indicating more variability explained by the model and less unexplained variation. In other words: The higher the R^2 value, the better the predictive power.

The *adjusted R^2 measure* also can be used to evaluate how well a model will fit a given data set [7]. It explains for any bias in the R^2 measure by taking into account the degrees of freedom of the independent variables and the sample population. The adjusted R^2 tends to remain constant as the R^2 measure for large population samples. The *F-ratio* is to test the null hypothesis that all regression coefficients are zero at statistically significant levels.

How does one interpret the data in Table 5? Let us focus straight away on the R^2 values of the regression models. The R^2 values indicate that our principal components explain between 57.9% and 91.9% of the variance—which indicates the efficacy of the built regression models. The adjusted R^2 values indicate the lack of bias in our R^2 values—that is, the regression models are robust.

To evaluate the predictive predictors, we ran a standard experiment: For each project, we randomly split the set of entities into 2/3 and 1/3, respectively. We then built a predictor from the 2/3 set. The better the predictor, the stronger the correlations would be between the actual and estimated post-release defects; a correlation of 1.0 would mean that the sensitivity of the predictor is high and vice versa.

The results of our evaluation are summarized in Table 6. Overall, we performed five random splits to build five models for each project to evaluate the prediction efficacy. We repeated the same process using different random splits, overall leading to 25 different models and predictions. Again, positive correlations are shown in bold. We present both the Spearman and Pearson correlations for completeness; the Pearson bivariate correlation requires the data to be distributed normally and the association between elements to be linear. In three of the five projects, all but one split result in significant predictions. The exceptions are projects C and E, which is due to the small number of binaries in these projects: In random splitting, a small sample size is unlikely to perform well, simply because one single badly ranked entity is enough to bring the entire correlation down.

What does this predictive power mean in practice? In Figure 2, we show two examples of ranking modules both by estimated and actual number of post-release defects. The left side shows one of the random split experiments from Table 6 with a Pearson correlation of >0.6. The project shown had 30 modules; the history and metrics of 2/3 of these were used for predicting the ranking of the remaining ten modules. If a manager decided to put more testing effort into, say, the top 30% or three of the predicted modules, this selection would contain the two most failure-prone modules, namely #4 and #8. Only one selected module (#6) would receive too much testing effort; and only one (#3) would receive too little.

Table 6. Predictive power of the regression models in random split experiments

Project	Correlation type	Random split 1	Random split 2	Random split 3	Random split 4	Random split 5
A	Pearson	**0.480**	**0.327**	**0.725**	-0.381	**0.637**
	Spearman	**0.238**	**0.185**	**0.693**	-0.602	**0.422**
B	Pearson	-0.173	**0.410**	**0.181**	**0.939**	**0.227**
	Spearman	-0.055	**0.054**	**0.318**	**0.906**	**0.218**
C	Pearson	**0.559**	-0.539	-0.190	**0.495**	-0.060
	Spearman	**0.445**	-0.165	**0.050**	**0.190**	**0.082**
D	Pearson	**0.572**	**0.845**	**0.522**	**0.266**	**0.419**
	Spearman	**0.617**	**0.828**	**0.494**	**0.494**	**0.494**
E	Pearson	-0.711	**0.976**	-0.818	**0.418**	**0.007**
	Spearman	-0.759	**0.577**	-0.883	**0.120**	**0.152**

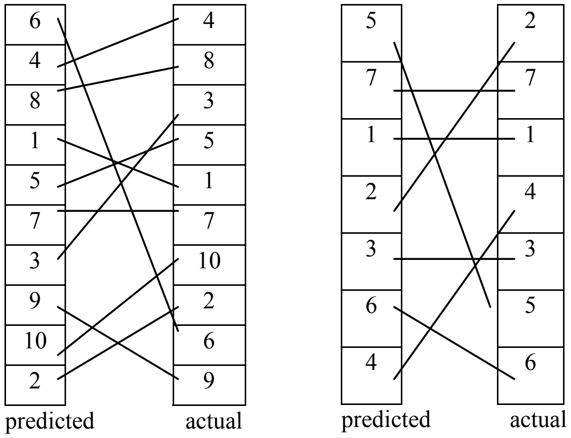

Figure 2. Comparing predicted and actual rankings

On the right side of Figure 2, we see another experiment from Table 6 with a Pearson correlation of <0.3. Here, the inaccurate ranking of module #5 in a small sample size is the reason for the low correlation. However, for any top *n* predicted modules getting extra effort, one would never see more than one module not deserving that effort, and never more than one of the top *n* actual modules missed.

All in all, both the R^2 values in Table 5 and the sensitivity of the predictions in Table 6 confirm our hypothesis H_3 for all five projects, illustrated by the examples in Figure 2. In practice, this means that within a project, the past failure history of a project can successfully predict the likelihood of post-release defects for new existing entities; therefore, the predictors can also be used after a change to estimate the likelihood of failure. The term "new entities" also includes new versions of existing entities; therefore, the predictions can also be used after a change to estimate the likelihood of failure.

Predictors obtained from principal component analysis are useful in building regression models to estimate post-release defects.

4.4 Are predictors obtained from one project applicable to other projects?

Finally, our hypothesis H_4 remains: If we build a predictor from the history and metrics of one project, would it also be predictive for other projects? We evaluated this question by building one predictor for each project, and applying it to the entities of each of the other four projects. Once more, we checked how well the actual and predicted rankings of the entities would correlate.

Our findings are summarized in Table 7. The entry "yes" indicates a significant correlation, meaning that the predictor would be successful; "no" means no or insignificant correlation.

Table 7. Prediction correlations using models built from a different project

Project used to build the model	**Sensitivity correlations between actual and predicted**					
		A	**B**	**C**	**D**	**E**
A	Pearson		No	No	No	No
	Spearman		No	No	No	No
B	Pearson	No		**Yes**	No	No
	Spearman	No		No	No	No
C	Pearson	No	**Yes**		No	**Yes**
	Spearman	No	**Yes**		No	**Yes**
D	Pearson	No	No	No		No
	Spearman	No	No	No		No
E	Pearson	No	No	No	No	
	Spearman	No	No	**Yes**	No	

As it turns out, the results are mixed—some project histories can serve as predictors for other projects, while most cannot. However, after our hypothesis H_2 has failed, this is not too surprising. Learning from earlier failures can only be successful if the two projects are similar—from the failure history of an Internet game, one can hardly make predictions for a nuclear reactor.

What is it that makes projects "similar" to each other? We found that those project pairs which are cross-correlated share the same heterogeneous defect distribution across modules which would also account for the large number of defect-correlated metrics, as observed in Section 4.1. The cross-correlated projects B and C, for instance, both share a heterogeneous defect distribution,

In essence, this means that one can learn from code that is more failure-prone to predict other entities which are equally failure-prone. For projects which are already aware of failure-prone components, one should go beyond simple code metrics, and consider the goals, the domain, and the processes at hand to find similar projects to learn from. This, however, is beyond the scope of this paper.

To sum up, we find our hypothesis H_4 only *partially confirmed:* Predictors obtained from one project are applicable *only to similar projects*—which again substantiates our word of caution against indiscriminate use of metrics. Ideas on how to identify similar projects are discussed in Section 7.

Predictors are accurate only when obtained from the same or similar projects.

5. LESSONS LEARNED

We started this work with some doubts about the usefulness of complexity metrics. Some of these doubts were confirmed: Choosing metrics without a proper validation is unlikely to result in meaningful predictions—at least when it comes to predict post-release defects, as we did. On the other side, metrics proved to be useful as *abstractions* over program code, capturing similarity between components that turned out to be a good source for predicting post-release defects. Therefore, we are happy that the failure history of the same or a similar project can indeed serve to validate and calibrate metrics for the project at hand.

Rather than predicting post-release defects, we can adapt our approach to arbitrary measures of quality. For instance, our measure might involve the cost or severity of failures, risk considerations, development costs, or maintenance costs. The general idea stays the same: From earlier history, we select the combination of metrics which best predicts the future. Therefore, we have summarized our approach in a step-by-step guide, shown in Figure **3**. In the long term, this guide will be instantiated for other projects within Microsoft, using a variety of code and process metrics as input for quality predictors.

- ➢ ***DO NOT*** *use complexity metrics without validating them for your project.*
- ➢ ***DO*** *use metrics that are validated from history to identify low-quality components.*

6. THREATS TO VALIDITY

In this paper, we have reported our experience with five projects of varying goal, process, and domain. Although we could derive successful predictors from the failure history in each of the projects, this may not generalize to other projects. In particular, the specific failure history, the coding and quality standards, or other process properties may be crucial for the success. We therefore encourage users to evaluate the predictive power before usage—for instance, by repeating the experiments described in Section 4.3.

Even if our approach accurately predicts failure-prone components, we advise against making decisions which are based uniquely upon such a prediction. To minimize the damage of post-release defects, one must not only consider the number of defects, but also the severity, likelihood, and impact of the resulting failures, as established in the field. Such estimations, however, are beyond the scope of this paper.

While the approach easily generalizes, we would caution against drawing general conclusions from this specific empirical study. In software engineering, any process depends to a large degree on a potentially large number of relevant context variables. For this reason, we cannot assume a priori that the results of a study generalize beyond the specific environment in which it was conducted [2]. Researchers become more confident in a theory when similar findings emerge in different contexts [2]. Towards this end, we hope that our case study contributes to strengthening the existing empirical body of knowledge in this field.

Building quality predictors: A step-by-step guide

1. Determine a software E from which to learn. E can be an earlier release of the software at hand, or a similar project.
2. Decompose E into entities (subsystems, modules, files, classes…) $E = \{e_1, e_2, ...\}$ for which you can determine the individual quality.
   In this paper, we decomposed the software into individual binaries—i. e. Windows components—simply because a mapping between binaries and post-release failures was readily available.
3. Build a function *quality*: $E \to \mathbb{R}$ which assigns to each entity $e \in E$ a quality. This typically requires mining version and bug histories (Section 2.4).
   In our case, the "quality" is the number of defects in an entity e that were found and fixed due to post-release failures.
4. Have a set of *metric functions* $M = \{m_1, m_2, ...\}$ such that each $m \in M$ is a mapping m: $E \to \mathbb{R}$ which assigns a metric to an entity $e \in E$. The set of metrics M should be adapted for the project and programming language at hand.
   We use the set of metrics M described in Table 3.
5. For each metric $m \in M$ and each entity $e \in E$, determine $m(e)$.
6. Determine the correlations between all $m(e)$ and *quality*(e), as well as the inter-correlations between all $m(e)$.
   The set of correlations between all m(e) and quality(e) is shown in Table 3; the inter-correlations are omitted due to lack of space.
7. Using principal component analysis, extract a set of principal components $PC = \{pc_1, pc_2, ...\}$, where each component $pc_i \in PC$ has the form $pc_i = \left\langle c_1, c_2, ..., c_{|M|} \right\rangle$.
   An example of the set PC is given in Table 4.
8. You can now use the principal components PC to build a predictor for new entities $E' = \{e_1', e_2', ...\}$ with $E' \cap E = \varnothing$. Be sure to evaluate the explanative and predictive power—for instance, using the experiments described in Section 4.3.
   We used PC to build a logistic regression equation, in which we fitted the metrics m(e') for all new entities $e' \in E'$ and all metrics $m \in M$. The equation resulted in a vector $P = \left\langle p_1, p_2, ..., p_{|E'|} \right\rangle$ where each $p_i \in P$ is the probability of failure of the entity $e_i' \in E'$.

Figure 3. How to build quality predictors

7. CONCLUSION AND FUTURE WORK

In this work, we have addressed the question "Which metric is best for me?" and reported our experience in resolving that question. It turns out that complexity metrics can successfully predict post-release defects. However, there is no single set of metrics that is applicable to all projects. Using our approach, organizations can leverage failure history to build good predictors which are likely to be accurate for similar projects, too.

This work extends the state of the art in four ways. It is one of the first studies to show how to systematically build predictors for post-release defects from failure history from the field. It also investigates whether object-oriented metrics can predict post-release defects. It analyzes whether predictors obtained from one project history are applicable to other projects, and last but not least, it is one of the largest studies of commercial software—in terms of code size, team sizes, and software users.

Of course, there is always more to do. Our future work will concentrate on these "more" topics:

- **More metrics.** Right now, the code metrics suggested are almost deceptively simple. While in our study, McCabe's cyclomatic complexity turned out to be an overall good predictor, it does not take into account all the additional complexity induced by method calls—and this is where object-oriented programs typically get complicated. We plan to leverage the failure data from several projects to evaluate more sophisticated metrics that again result in better predictors.
- **More data.** Besides only collecting source code versions and failure reports, we have begun to collect and recreate run-time information such as test coverage, usage profiles, or change effort. As all of these might be related to post-release defects, we expect that they will further improve predictive power—and provide further guidance for quality assurance.
- **More similarity.** One important open question in our work is: What is it that makes projects "similar" enough such that predictions across projects become accurate? For this purpose, we want to collect and classify data on the process and domain characteristics. One possible characterization would be a *polar chart* as shown in Figure 4, where we would expect similar projects to cover a similar space. As a side effect, we could determine which process features correlate with quality.
- **More automation.** While we have automated the extraction and mapping of failure and version information, we still manually use third-party statistical tools to obtain the predictors. We want to automate and integrate this last step as well, such that we can automatically obtain predictors from software archives. The next step would be to integrate these predictors into development environments, supporting the decisions of programmers and managers.
- **More projects.** Given a fully automated system, we shall be able to apply the approach on further projects within and outside of Microsoft. This will add more diversity to the field—and, of course, help companies like Microsoft to maximize the impact of their quality efforts.

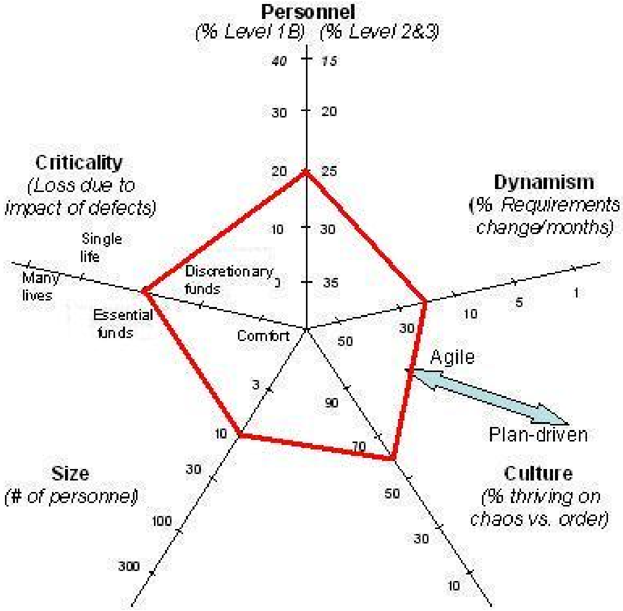

Figure 4. A Boehm-Turner polar chart [6] which characterizes the software process [15]

All in all, modern software development produces an abundance of recorded process and product data that is now available for automatic treatment. Systematic empirical investigation of this data will provide guidance in several software engineering decisions—and further strengthen the existing empirical body of knowledge in software engineering.

Acknowledgments. Andreas Zeller's work on mining software archives was supported by Deutsche Forschungsgemeinschaft, grant Ze 509/1-1. We thank Melih Demir, Tom Zimmermann and many others for their helpful comments on earlier revisions of this paper. We would like to acknowledge all the product groups at Microsoft for their cooperation in this study.

REFERENCES

[1] E. N. Adams, "Optimizing Preventive Service of Software Products", *IBM Journal of Research and Development*, 28(1), pp. 2-14, 1984.

[2] V. Basili, Shull, F.,Lanubile, F., "Building Knowledge through Families of Experiments", *IEEE Transactions on Software Engineering*, 25(4), pp. 456-473, 1999.

[3] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators", *IEEE Transactions on Software Engineering*, 22(10), pp. 751-761, 1996.

[4] A. B. Binkley, Schach, S., "Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures", Proceedings of International Conference on Software Engineering, pp. 452 - 455, 1998.

[5] S. Biyani, Santhanam, P., "Exploring defect data from development and customer usage on software modules over multiple releases", Proceedings of International Symposium on Software Reliability Engineering, pp. 316-320, 1998.

[6] B. Boehm and R. Turner, “Using Risk to Balance Agile and Plan-Driven Methods”, *IEEE Computer*, 36(6), pp. 57-66, June 2003.

[7] F. Brito e Abreu, Melo, W., “Evaluating the Impact of Object-Oriented Design on Software Quality”, Proceedings of Third International Software Metrics Symposium, pp. 90-99, 1996.

[8] S. R. Chidamber and C. F. Kemerer, “A Metrics Suite for Object Oriented Design”, *IEEE Transactions on Software Engineering*, 20(6), pp. 476-493, 1994.

[9] D. Čubranić, Murphy, G.C., “Hipikat: recommending pertinent software development artifacts”, Proceedings of International Conference on Software Engineering, pp. 408-418, 2003.

[10] N. E. Fenton, Neil, M., “A critique of software defect prediction models”, *IEEE Transactions in Software Engineering*, 25(5), pp. 675-689, 1999.

[11] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*: Brooks/Cole, 1998.

[12] M. Fischer, Pinzger, M., Gall, H., “Populating a Release History Database from version control and bug tracking systems”, Proceedings of International Conference on Software Maintenance, pp. 23-32, 2003.

[13] J. P. Hudepohl, Aud, S.J., Khoshgoftaar, T.M., Allen, E.B., Mayrand, J., “Emerald: software metrics and models on the desktop”, *IEEE Software*, 13(5), pp. 56 - 60, 1996.

[14] E. J. Jackson, *A User's Guide to Principal Components*. Hoboken, NJ: John Wiley & Sons Inc., 2003.

[15] L. Layman, L. Williams, and L. Cunningham, “Exploring Extreme Programming in Context: An Industrial Case Study”, Proceedings of Agile Development Conference, Salt Lake City, UT, pp. 32-41, 2004.

[16] A. Mockus, Zhang, P., Li, P., “Drivers for customer perceived software quality”, Proceedings of International Conference on Software Engineering (ICSE), St. Louis, MO, pp. 225-233, 2005.

[17] N. Nagappan, Ball, T., “Use of Relative Code Churn Measures to Predict System Defect Density”, Proceedings of International Conference on Software Engineering (ICSE), St. Louis, MO, pp. 284-292, 2005.

[18] N. Ohlsson, Alberg, H., “Predicting fault-prone software modules in telephone switches”, *IEEE Transactions in Software Engineering*, 22(12), pp. 886 - 894, 1996.

[19] T. Ostrand, Weyuker, E., Bell, R.M., “Predicting the location and number of faults in large software systems”, *IEEE Transactions in Software Engineering*, 31(4), pp. 340 - 355, 2005.

[20] J. Sliwerski, Zimmermann, T., Zeller, A., “When Do Changes Induce Fixes?” Proceedings of Mining Software Repositories (MSR) Workshop, 2005.

[21] R. Subramanyam and M. S. Krishnan, “Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects”, *IEEE Transactions on Software Engineering*, 29(4) pp. 297-310, April 2003.

[22] T. Zimmermann, Weißgerber, P., Diehl, S., Zeller, A., “Mining Version Histories to Guide Software Changes”, *IEEE Transactions in Software Engineering*, 31(6), pp. 429-445, 2005.