Predicting Defects for Eclipse

[Revised for Dataset Version 2.0a]

Thomas Zimmermann
Saarland University
tz@acm.org

Rahul Premraj
Saarland University
premraj@cs.uni-sb.de

Andreas Zeller
Saarland University
zeller@acm.org

Abstract

We have mapped defects from the bug database of Eclipse (one of the largest open-source projects) to source code locations. The resulting data set lists the number of pre- and post-release defects for every package and file in the Eclipse releases 2.0, 2.1, and 3.0. We additionally annotated the data with common complexity metrics. All data is publicly available and can serve as a benchmark for defect prediction models.

1. Introduction

Why is it that some programs are more failure-prone than others? This is one of the central questions of software engineering. To answer it, we must first know *which* programs are more failure-prone than others. With this knowledge, we can search for properties of the program or its development process that commonly correlate with defect density; in other words, once we can measure the effect, we can search for its causes.

One of the most abundant, widespread, and reliable sources for failure information is a *bug database,* listing all the problems that occurred during the software lifetime. Unfortunately, bug databases frequently do not directly record how, where, and by whom the problem in question was fixed. This information is hidden in the *version database,* recording all changes to the software source code.

In recent years, a number of techniques have been developed to relate bug reports to fixes [3, 6, 17]. Since we thus can relate bugs to fixes, and fixes to the locations they apply to, we can easily determine the *number of defects* of a component—simply by counting the applied fixes.

We have conducted such a work on the code base of the Eclipse programming environment. In particular, we have computed the mapping of packages and classes to the number of defects that were reported in the first six months *before* and *after* release. In previous work, we made our *Eclipse bug data set* freely available [15]. For this paper, we extended our data with common complexity metrics and the counts of syntactic elements (obtained from abstract syntax trees). With this new data, many predictor models can be built out of the box which we demonstrate in this paper.

Project:	Eclipse (eclipse.org)
Content:	Defect counts (pre- and post-release) Complexity metrics
Releases:	2.0, 2.1, and 3.0
Level:	Packages and files
URL:	http://www.st.cs.uni-sb.de/softevo/bug-data/eclipse
More data:	Eclipse source code (for archived releases): http://archive.eclipse.org/eclipse/downloads/

Figure 1. Summary of our data set.

We invite readers to use our data for research purposes and to build their own models. We hope that the public availability of data sets like ours will foster empirical research in software engineering, just like the public availability of open source programs fostered research in program analysis.

2. State of the art

Predicting which components are more failure-prone than others has been addressed by a number of researchers in the past. This work, discussed below, used either complexity metrics or historical data to predict failures.

2.1. Complexity metrics

Typically, research on defect-proneness defines metrics to capture the complexity of software and builds models that relate these metrics to defect-proneness [4]. Basili et al. [1] were among the first to validate that OO metrics are useful for predicting defect density. Subramanyam and Krishnan [18] presented a survey on eight more empirical studies, all showing that OO metrics are significantly associated with defects. Post-release defects are the defects that actually matter for the end-users of a program. Only few studies ad-

Table 1. Metrics in the Eclipse data set.

		Metric	File level	Package level
methods	FOUT	Number of method calls (fan out)	avg, max, total	avg, max, total
	MLOC	Method lines of code	avg, max, total	avg, max, total
	NBD	Nested block depth	avg, max, total	avg, max, total
	PAR	Number of parameters	avg, max, total	avg, max, total
	VG	McCabe cyclomatic complexity	avg, max, total	avg, max, total
classes	NOF	Number of fields	avg, max, total	avg, max, total
	NOM	Number of methods	avg, max, total	avg, max, total
	NSF	Number of static fields	avg, max, total	avg, max, total
	NSM	Number of static methods	avg, max, total	avg, max, total
files	ACD	Number of anonymous type declarations	value	avg, max, total
	NOI	Number of interfaces	value	avg, max, total
	NOT	Number of classes	value	avg, max, total
	TLOC	Total lines of code	value	avg, max, total
packages	NOCU	Number of files (compilation units)	N/A	value

dressed post-release defects so far: Binkley and Schach [2] developed a coupling dependency metric and showed that it outperforms several other metrics; Ohlsson and Alberg [13] investigated a number of metrics to predict modules that fail during test or operation. Schröter et al. [16] showed that design data such as import relationships also can predict post-release failures.

The MetriZone project at Microsoft Research investigates how to make early estimates of software quality to predict post-release failures. Nagappan and Ball [11] showed that relative code churn predicts software defect density; (absolute) code churn is the number of lines added or deleted between versions. Additionally, Nagappan et al. [12] carried out the largest study on commercial software so far: Within five Microsoft projects, they identified metrics that predict post-release failures and reported how to systematically build predictors for post-release failures from history. Nagappan and Ball [10] also showed that the ratio between the number of dependencies within a component and the number of dependencies across a component can predict post-release failures.

2.2. Historical data

Several researchers used historical data without taking bug databases into account. Khoshgoftaar et al. [9] classified modules as defect-prone whenever the number of lines of code added or deleted exceeded a threshold. Graves et al. [7] used the sum of contributions to a module in its history to predict defect density. Ostrand et al. [14] used historical data from up to 17 releases to predict the files with the highest defect density in the next release. Hudepohl et al. [8] predicted whether a module would be defect-prone by combining metrics and historical data. From several software metrics, Denaro et al. [5] learned logistic regression models for Apache 1.3 and verified them against Apache 2.0.

3. Eclipse data set

This section presents details on how we computed the *Eclipse bug data set* and a description of its contents. Additionally, we point out research problems that can be investigated with our data set.

3.1. Data collection

How do we know which components failed and which did not? This data can be collected from version archives like CVS and bug tracking systems like BUGZILLA in two steps:

1. We identify corrections (or fixes) in version archives: Within the commit messages, we search for references to bug reports such as “Fixed 42233” or “bug #23444”. Basically every number is a potential reference to a bug report; however such references have a low trust at first. We increase the trust level when a message contains keywords such as “fixed” or “bug” or matches patterns like “# and a number”. This approach was previously used in research [3, 6, 17].
2. We use the bug tracking system to map bug reports to releases. Each bug report contains a field called “version” that lists the release for which the bug was reported; however, since the values of this field may change during the life cycle of a bug (e.g., when a bug is carried over to the next release), we only use the first reported release.

We distinguish two different kinds of defects: *pre-release defects* are observed during development and testing of a program, while *post-release defects* are observed after the program has been deployed to its users.

Since we know the location of every defect that has been fixed, it is easy to count the number of defects per location and release.

For the computation of complexity metrics, we used the Java parser of Eclipse. We implemented *visitors* that compute standard metrics (see Table 1) for methods, classes, and files (compilation units) and *aggregators* that combine the metric values into single values for the levels we were interested in (files and packages). For aggregation we used the average, total, and maximum values of the metrics; we omitted minimum values because they are zero in most cases. The source code metrics were computed on the archived builds of Eclipse (see the URL in Figure 1). Note that file level is different from class level since in Java one file can contain several classes.

3.2. Data description

Our data consists of six files in total—one file for each level (files, packages) and release (2.0, 2.1, 3.0). Table 2 summarizes the total number of cases per file. Each case contains the following information:

- ***name:*** The name of the file or package, respectively, to which this case corresponds. It can be used to identify the source code in the release and may be needed for additional data collection.
- ***pre-release defects:*** The number of non-trivial defects that were reported in the *last six months before release.*
- ***post-release defects:*** The number of non-trivial defects that were reported in the *first six months after release.*
- ***complexity metrics:*** We computed for each case several complexity metrics (see Table 1). Metrics that are computed for classes or methods are aggregate by using average (avg), maximum (max), and accumulation (sum) to file and package level.
- ***structure of abstract syntax tree(s):*** For each case, we list the size (=number of nodes) of the abstract syntax tree(s) of the file or package, respectively. Abstract syntax trees also consist of different types of nodes (see Figure 2). In addition to size, we also list the frequency of each of these nodes. These counts allow constructing new metrics without any additional processing of the source code.

Table 2. Number of cases

	Number of	
Release	Files	Packages
2.0	6729	377
2.1	7888	434
3.0	10593	661

AnnotationTypeDeclaration	*MethodInvocation*
AnnotationTypeMemberDeclaration	*MethodRef*
AnonymousClassDeclaration	*MethodRefParameter*
ArrayAccess	*Modifier*
ArrayCreation	*NormalAnnotation*
ArrayInitializer	*NullLiteral*
ArrayType	*NumberLiteral*
AssertStatement	*PackageDeclaration*
Assignment	*ParameterizedType*
Block	*ParenthesizedExpression*
BlockComment	*PostfixExpression*
BooleanLiteral	*PrefixExpression*
BreakStatement	*PrimitiveType*
CastExpression	*QualifiedName*
CatchClause	*QualifiedType*
CharacterLiteral	*ReturnStatement*
ClassInstanceCreation	*SimpleName*
CompilationUnit	*SimpleType*
ConditionalExpression	*SingleMemberAnnotation*
ConstructorInvocation	*SingleVariableDeclaration*
ContinueStatement	*StringLiteral*
DoStatement	*SuperConstructorInvocation*
EmptyStatement	*SuperFieldAccess*
EnhancedForStatement	*SuperMethodInvocation*
EnumConstantDeclaration	*SwitchCase*
EnumDeclaration	*SwitchStatement*
ExpressionStatement	*SynchronizedStatement*
FieldAccess	*TagElement*
FieldDeclaration	*TextElement*
ForStatement	*ThisExpression*
IfStatement	*ThrowStatement*
ImportDeclaration	*TryStatement*
InfixExpression	*TypeDeclaration*
Initializer	*TypeDeclarationStatement*
InstanceofExpression	*TypeLiteral*
Javadoc	*TypeParameter*
LabeledStatement	*VariableDeclarationExpression*
LineComment	*VariableDeclarationFragment*
MarkerAnnotation	*VariableDeclarationStatement*
MemberRef	*WhileStatement*
MemberValuePair	*WildcardType*
MethodDeclaration	

Figure 2. Abstract syntax tree nodes.

3.3. Data relevance

The Eclipse bug data set can be used to build and assess models for defect prediction. Figure 3 shows a histogram of the number of defects for packages in Eclipse 3.0. Most packages have no observed defects; some packages have up to 65 defects reported.

This distribution calls for two interesting research questions: Which files/packages have defects (a *classification* problem)? And which are the files/packages with the most defects (a *ranking* problem)? Having reliable predictions for both supports allocation of resources for quality assurance (such as testing) to parts of a system that are most defect-prone.

3.3.1. Classification

Classification tries to predict whether a file or package will have at least one defect reported. When applied to all files/packages, the outcome is a classification table such as the following:

		Defects are observed.	
		True	False
Model predicts defects.	Positive	True Positive (TP)	False Positive (FP) → **Precision**
	Negative	False Negative (FN)	True Negative (TN)
		↓ **Recall**	↘ **Accuracy**

For assessing the quality of a classification model we recommend to use precision, recall, and accuracy:

- ***Precision.*** The precision relates the number of true positives (predicted and observed as defect-prone) to the number of files/packages predicted as defect-prone.

  $$precision = TP / (TP + FP)$$

  A value close to one is desirable and would mean that every file/package that was predicted to have defects actually had defects.

- ***Recall.*** The recall relates the number of true positives (predicted and observed as defect-prone) to the number of files/packages that actually had defects.

  $$recall = TP / (TP + FN)$$

  A value close to one is best and would mean that every file/package that had defects observed was predicted to have defects.

- ***Accuracy.*** The accuracy relates the number of correct classifications (true positives and true negatives) to the total number of files/packages.

  $$accuracy = (TP + TN) / (TP + TN + FP + FN)$$

  A value of one is best and would mean that the model classified perfectly, i.e., made not a single mistake.

In order to interpret these measures correctly, one additionally needs to know the percentage of files (or packages) that have defects. Assume that 80% of all files have defects and a model classifies every file as defect-prone. In this case, the model has a precision of 80%, recall of 100%, and accuracy of 80%. Still such a model is not helpful for classification purposes.

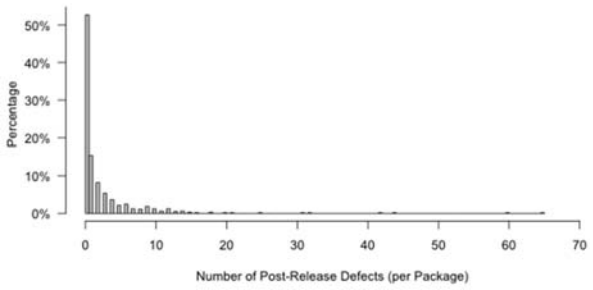

Figure 3. Histogram of post-release defects.

3.3.2. Ranking

Ranking tries to predict an order of files/packages where the files/packages with most defects come first. To measure the quality of such a ranking, we recommend using the Spearman correlation.

- ***Spearman correlation.*** The Spearman correlation coefficient measures the correlation between a predicted and observed ranking. High correlations and thus a high quality of the predicted ranking are indicated by values close to 1 and -1: values of 1 indicate a identical ranking and values of -1 indicate an opposite ranking. Values close to 0 indicate no correlation.

4. Experiments

In this section, we present a few experiments using the *Eclipse bug data set*. We do not attempt to give definite answers for defect prediction, but merely highlight the potential of bug data when it comes to address this problem.

4.1. Correlations

At first, we computed the Spearman correlation between the number of **pre-release and post-release defects** and the complexity metrics in the data set. Table 3 lists the correlation values for both, file and package level (release 3.0). Correlations significant at the 0.01 level are marked with (*).

The high correlation value of 0.768 between the number of pre-release and post-release defects on package level indicates that the packages having the most pre-release defects are likely to also have the most post-release defects and vice versa. This effect is not as strong on file level (correlation on 0.390).

Most correlations with metrics are positive and significant—the more complex a file/package the more defects it will have. However, on **file level** only the accumulated number of method calls (FOUT_sum), the

total lines of code (TLOC_sum), and the closely related sum of method lines of code (MLOC_sum) have correlation values above 0.400. This indicates that the size of files and number of methods calls seems to be a good indicator for defect-prone files.

On **package level**, most correlations are above the 0.400 threshold: fan out as measured by the number of calls in a method (FOUT), lines of code (MLOC and TLOC) nested block depth (NBD), number of non-static fields (NOF and NSF) and methods (NOM), complexity (VG), and the number of files in a package (NOCU). Most accumulated metrics (_sum) have higher correlations values than the averaged metrics. In many cases the maximum metrics (_max) show correlation values comparable to the accumulated ones. The number of interfaces (NOI) is the only metric for which we observe no correlation at all.

Finding a single indicator or predictor for the number of defects is extremely unlikely. In the next subsections, we will combine input features by building regression models (linear for ranking, and logistic for classification).

4.2. Classification

The previous section showed that individual code metrics correlate with the number of defects. But how can we use metrics to predict whether a file/package will have defects? One solution is to build statistical models that classify files/packages as defect-prone (*has_defects=1*) or not (*has_defects=0*) based on the values of the code metrics.

We built logistic regression models for the *Eclipse bug data set* to predict whether files/packages have **post-release defects.** Logistic regression models typically predict likelihoods (between 0 and 1); when the predicted likelihood was above 0.5, we classified a file/package as defect-prone, otherwise as defect-free. In total we built six models for two levels of granularity (files, packages) and three releases (2.0, 2.1, 3.0). We tested the models across releases of Eclipse, but always on the level they were built from.

Table 4 lists the precision, recall, and accuracy values for the **file level**. The recall values are low for all tests, meaning that only few of the defect-prone files were correctly identified as defect-prone. However, the precision values are above 0.500 in all but one cases, suggesting that there are only few false positives, i.e., when a file is classified as defect-prone, this decision is most likely to be correct. Most of the precision, recall, and accuracy values remain comparable across releases. This means that a model learned from one release, say 2.0, can be applied to a later release, say 3.0, without losing too much predictive power (only a decrease of 0.015 in accuracy).

Table 3. Spearman correlation between pre- and post-release defects and metrics.

Release 3.0	File level		Package level	
	pre	post	pre	post
pre	1.000	.390 (*)	1.000	.768 (*)
post	.390 (*)	1.000	.768 (*)	1.000
FOUT_avg	.313 (*)	.242 (*)	.258 (*)	.266 (*)
FOUT_max	.375 (*)	.291 (*)	.429 (*)	.413 (*)
FOUT_sum	.400 (*)	.319 (*)	.537 (*)	.523 (*)
MLOC_avg	.314 (*)	.243 (*)	.242 (*)	.282 (*)
MLOC_max	.380 (*)	.293 (*)	.429 (*)	.455 (*)
MLOC_sum	.403 (*)	.322 (*)	.545 (*)	.544 (*)
NBD_avg	.303 (*)	.237 (*)	.241 (*)	.280 (*)
NBD_max	.368 (*)	.290 (*)	.487 (*)	.508 (*)
NBD_sum	.392 (*)	.320 (*)	.552 (*)	.546 (*)
NOF_avg	.242 (*)	.191 (*)	.268 (*)	.264 (*)
NOF_max	.256 (*)	.201 (*)	.456 (*)	.417 (*)
NOF_sum	.260 (*)	.204 (*)	.507 (*)	.480 (*)
NOM_avg	.296 (*)	.255 (*)	.241 (*)	.273 (*)
NOM_max	.314 (*)	.266 (*)	.418 (*)	.417 (*)
NOM_sum	.319 (*)	.268 (*)	.502 (*)	.491 (*)
NSF_avg	.174 (*)	.162 (*)	.256 (*)	.216 (*)
NSF_max	.186 (*)	.170 (*)	.397 (*)	.354 (*)
NSF_sum	.186 (*)	.170 (*)	.459 (*)	.414 (*)
NSM_avg	.197 (*)	.176 (*)	.250 (*)	.175 (*)
NSM_max	.202 (*)	.179 (*)	.391 (*)	.323 (*)
NSM_sum	.202 (*)	.179 (*)	.448 (*)	.371 (*)
PAR_avg	.094 (*)	.064 (*)	.111 (*)	.122 (*)
PAR_max	.257 (*)	.209 (*)	.399 (*)	.378 (*)
PAR_sum	.350 (*)	.283 (*)	.554 (*)	.526 (*)
VG_avg	.300 (*)	.234 (*)	.254 (*)	.268 (*)
VG_max	.359 (*)	.279 (*)	.435 (*)	.430 (*)
VG_sum	.389 (*)	.315 (*)	.546 (*)	.538 (*)
NOCU			.514 (*)	.461 (*)
ACD	.258 (*)	.180 (*)		
ACD_avg			.316 (*)	.301 (*)
ACD_max			.416 (*)	.389 (*)
ACD_sum			.442 (*)	.414 (*)
NOI	-.160 (*)	-.129 (*)		
NOI_avg			-.021	-.037
NOI_max			.118 (*)	.094
NOI_sum			.129 (*)	.110 (*)
NOT	.160 (*)	.129 (*)		
NOT_avg			.029	.043
NOT_max			.190 (*)	.174 (*)
NOT_sum			.518 (*)	.470 (*)
TLOC	.421 (*)	.333 (*)		
TLOC_avg			.354 (*)	.377 (*)
TLOC_max			.527 (*)	.505 (*)
TLOC_sum			.581 (*)	.559 (*)

Correlations significant at the 0.01 level marked with (*).

Table 5 presents the results for the **package level**. The results improve substantially; precision values are now between 0.741 and 0.892, recall values between 0.588 and 0.789. Intuitively, it is easier to predict defect-prone packages, as already one defect-prone file makes a package defect-prone.

4.3. Ranking

In order to predict the files/packages that have most **post-release defects** we used linear regression models. Using these models we predicted for each file/package the number of expected post-release defects and compared the resulting ranking to the observed ranking using Spearman correlation. Again, we built models for each level (file, package) and release (2.0, 2.1, 3.0) and tested them across releases.

Table 6 shows the results for **file level.** In addition to Spearman, we also list the R^2 value of the trained model and the Pearson correlation. The R^2 value measures the variability in a data set that is accounted for by a statistical model. In our case only up to 37.4% of variance are explained by complexity metrics. Pearson correlation assumes a linear relation between the correlated variables; we report Pearson values only for completeness. The Spearman correlation values ρ are low, reaching 0.398 at most, but positive. Therefore, files having a higher predicted rank are more likely to have a high observed rank, too.

The results for **package level** are listed in Table 7. The R^2 values substantially improve; for release 3.0 86.5% of variability is explained by the linear model. The Spearman correlation values increase substantially to up to 0.704. Again the increase in correlation and R^2 values demonstrates that it is easier to make predictions for packages than for files. Also, models learned from earlier releases can be used to predict for future releases; for instance the model trained from release 2.1 showed a correlation of 0.704 on release 3.0.

5. Conclusions

Where do bugs come from? By mapping failures to components, our *Eclipse bug data set* offers the opportunity to research this question. The experiments in this paper showed that the combination of complexity metrics can predict defects, suggesting that the more complex code it, the more defects it has.

However, our predictions are far from being perfect. They therefore raise follow-up questions: Are there better indicators for defects than complexity metrics? How applicable are models across projects and over time? How do we integrate prediction models into the development process?

Table 4. Classification of files. (Precision *P*, Recall *R*, and Accuracy *Acc*).

Training	Testing	Defects	P	R	Acc
2.0	2.0	0.145	0.692	0.265	0.876
	2.1	0.108	0.478	0.191	0.890
	3.0	0.148	0.613	0.171	0.861
2.1	2.0	0.145	0.664	0.203	0.870
	2.1	0.108	0.668	0.160	0.900
	3.0	0.148	0.717	0.139	0.864
3.0	2.0	0.145	0.578	0.277	0.866
	2.1	0.108	0.528	0.220	0.894
	3.0	0.148	0.675	0.224	0.869

Table 5. Classification of packages. (Precision *P*, Recall *R*, and Accuracy *Acc*).

Training	Testing	Defects	P	R	Acc
2.0	2.0	0.504	0.853	0.763	0.814
	2.1	0.447	0.741	0.634	0.737
	3.0	0.474	0.786	0.588	0.729
2.1	2.0	0.504	0.806	0.700	0.764
	2.1	0.447	0.857	0.742	0.829
	3.0	0.474	0.861	0.674	0.794
3.0	2.0	0.504	0.760	0.784	0.767
	2.1	0.447	0.782	0.758	0.797
	3.0	0.474	0.892	0.789	0.855

Table 6. Ranking files with linear regression. (Pearson *r* and Spearman ρ correlation)

Files		Testing		
		2.0	**2.1**	**3.0**
Training	**2.0**	r=0.569	r=0.430	r=0.544
	R^2=0.324	ρ=0.398	ρ=0.286	ρ=0.340
	2.1	r=0.517	r=0.489	r=0.562
	R^2=0.239	ρ=0.377	ρ=0.311	ρ=0.359
	3.0	r=0.518	r=0.434	r=0.611
	R^2=0.374	ρ=0.383	ρ=0.305	ρ=0.362

All correlations are significant at the 0.01 level.

Table 7. Ranking packages with linear regression. (Pearson *r* and Spearman ρ correlation)

Packages		Testing		
		2.0	**2.1**	**3.0**
Training	**2.0**	r=0.890	r=0.870	r=0.733
	R^2=0.793	ρ=0.647	ρ=0.492	ρ=0.546
	2.1	r=0.833	r=0.930	r=0.731
	R^2=0.865	ρ=0.641	ρ=0.704	ρ=0.704
	3.0	r=0.727	r=0.782	r=0.882
	R^2=0.779	ρ=0.655	ρ=0.558	ρ=0.691

All correlations are significant at the 0.01 level.

The above questions indicate the potential of future empirical research based on bug data. To support this very research, we are happy to make the *Eclipse bug data* set publicly available.

Overall, we would like this dataset to become both a challenge and a benchmark: Which factors in programs and processes are predictors of future bugs, and which approach gives the best prediction results? The more we learn about past mistakes, the better are our chances to avoid them in the future—and build better software at lower cost.

For access to the Eclipse bug data set, as well as for ongoing information on the project, see

http://www.st.cs.uni-sb.de/softevo/

Acknowledgments. Our work on mining software repositories is funded by Deutsche Forschungsgemeinschaft, grant Ze 509/1-1. Thomas Zimmermann is additionally funded by the DFG-Graduiertenkolleg-Leistungsgarantien für Rechnersysteme. Thanks to Adrian Schröter for computing the initial version of the *Eclipse bug data set*.s

6. References

[1] 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* vol. 22, pp. 751-761, 1996.

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

[3] D. Cubranic and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts." in *25th International Conference on Software Engineering (ICSE)*, Portland, Oregon, 2003, pp. 408-418.

[4] G. Denaro, S. Morasca, and M. Pezzè, "Deriving models of software fault-proneness." in *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering* Ischia, Italy, 2002 pp. 361 - 368.

[5] G. Denaro and M. Pezzè, "An empirical evaluation of fault-proneness models." in *Proceedings of the International Conference on Software Engineering (ICSE 2002)*, Orlando, Florida, USA, 2002, pp. 241-251.

[6] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems." in *Proc. International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, 2003.

[7] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history." *IEEE Transactions on Software Engineering,* vol. 26, 2000.

[8] J. P. Hudepohl, S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand, "Emerald: Software metrics and models on the desktop." *IEEE Software,* vol. 13, pp. 56-60, September 1996.

[9] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan, "Detection of software modules with high debug code churn in a very large legacy system." in *ISSRE '96: Proceedings of the The Seventh International Symposium on Software Reliability Engineering (ISSRE '96)*, Washington, DC, USA, 1996, p. 364.

[10] N. Nagappan and T. Ball, "Explaining failures using software dependences and churn metrics," Microsoft Research, Redmond, WA 2006.

[11] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density." in *Proceedings of the International Conference on Software Engineering (ICSE 2005)*, St. Louis, Missouri, USA, 2005, pp. 284-292.

[12] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures." in *Proceedings of the International Conference on Software Engineering (ICSE 2006)*, Shanghai, China, 2006.

[13] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches." *IEEE Trans. Software Eng.,* vol. 22, pp. 886-894, 1996.

[14] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems." *IEEE Trans. Software Eng.,* vol. 31, pp. 340-355, 2005.

[15] 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*, 2006, pp. 18-20.

[16] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting failure-prone components at design time." in *Proceedings of the 5th International Symposium on Empirical Software Engineering (ISESE 2006)*, Rio de Janeiro, Brazil, 2006.

[17] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes? On fridays." in *Proc. International Workshop on Mining Software Repositories (MSR)*, St. Louis, Missouri, U.S., 2005.

[18] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects." *IEEE Trans. Software Eng.,* vol. 29, pp. 297-310, 2003.

A. Appendix: Replication guide in GNU R

Step 0: Read the files

```
files_20 <- read.table("eclipse-metrics-files-2.0.csv", header=T, sep=";")
files_21 <- read.table("eclipse-metrics-files-2.1.csv", header=T, sep=";")
files_30 <- read.table("eclipse-metrics-files-3.0.csv", header=T, sep=";")

packages_20 <- read.table("eclipse-metrics-packages-2.0.csv", header=T, sep=";")
packages_21 <- read.table("eclipse-metrics-packages-2.1.csv", header=T, sep=";")
packages_30 <- read.table("eclipse-metrics-packages-3.0.csv", header=T, sep=";")
```

Step 1: Count the files and packages (Table 2)

```
nrow(files_20)
nrow(files_21)
nrow(files_30)
nrow(packages_20)
nrow(packages_21)
nrow(packages_30)
```

Step 2: Build the histogram (Figure 3)

```
par(mar=c(5, 5, 2, 1) + 0.1)
hist(packages_30$post, freq=T, breaks=100, xlim=c(0,70), axes=F, main="", xlab="Number
of Post-Release Defects (per Package)", ylab="Percentage", col="darkgray")
axis(1)
axis(2, at=c(0,66.1,66.1*2,66.1*3,66.1*4,66.1*5,66.1*6), la-
bels=c("0%","10%","20%","30%","40%","50%","60%"), las=1)
```

Steep 3: Compute the correlations including significance at 0.01 (Table 3)

Note that the data begins only in column 3 and we therefore have to start the for loop at 3 and use i-2 to access pre.p and post.p which start with index 1.

```
pre.p <- rep (-1, 33)
post.p <- rep (-1, 33)
for (i in 3:35) {
     pre.p[i-2] <- cor.test(files_30[,i], files_30$pre, method="spearman", ex-
act=FALSE)$p.value
     post.p[i-2] <- cor.test(files_30[,i], files_30$post, method="spearman", ex-
act=FALSE)$p.value
}

cbind(cor(files_30[,3:35], files_30$pre, method="spearman"), cor(files_30[,3:35],
files_30$post, method="spearman"), (pre.p<0.01), (post.p<0.01))

pre.p <- rep (-1, 42)
post.p <- rep (-1, 42)
for (i in 3:44) {
     pre.p[i-2] <- cor.test(packages_30[,i], packages_30 $pre, method="spearman", ex-
act=FALSE)$p.value
     post.p[i-2] <- cor.test(packages_30[,i], packages_30 $post, method="spearman",
exact=FALSE)$p.value
}

cbind(cor(packages_30[,3:44], packages_30$pre, method="spearman"),
cor(packages_30[,3:44], packages_30$post, method="spearman"), (pre.p<0.01),
(post.p<0.01))
```

Step 4: Run the classification experiments (Section 4.2, Table 4 and 5)

```
test_classification <- function (train, test)
{
     model.glm <- glm((post>0) ~ pre + ACD + FOUT_avg + FOUT_max + FOUT_sum + MLOC_avg
+ MLOC_max + MLOC_sum + NBD_avg + NBD_max + NBD_sum + NOF_avg + NOF_max + NOF_sum +
NOI + NOM_avg + NOM_max + NOM_sum + NOT + NSF_avg + NSF_max + NSF_sum + NSM_avg +
NSM_max + NSM_sum + PAR_avg + PAR_max + PAR_sum + + + TLOC + VG_avg + VG_max + VG_sum,
data=train, family = "binomial")
     test.prob <- predict(model.glm, test, type="response")
     test.pred <- test.prob>=0.50

     outcome <- table(factor(test$post>0, levels=c(F,T)), factor(test.pred, le-
vels=c(F,T)))
     TN <- outcome[1,1]
     FN <- outcome[2,1]
     FP <- outcome[1,2]
     TP <- outcome[2,2]
     precision <- if (TP + FP ==0) { 1 } else { TP / (TP + FP) }
     recall <- TP / (TP + FN)
     accuracy <- (TP + TN) / (TN + FN + FP + TP)
     defects <- (TP + FN) / (TN + FN + FP + TP)
     return (c(defects, precision, recall, accuracy))
}

test_classification_pkg <- function (train, test)
{
     model.glm <- glm((post>0) ~ pre + ACD_avg + ACD_max + ACD_sum + FOUT_avg +
FOUT_max + FOUT_sum + MLOC_avg + MLOC_max + MLOC_sum + NBD_avg + NBD_max + NBD_sum +
NOCU + NOF_avg + NOF_max + NOF_sum + NOI_avg + NOI_max + NOI_sum + NOM_avg + NOM_max +
NOM_sum + NOT_avg + NOT_max + NOT_sum + NSF_avg + NSF_max + NSF_sum + NSM_avg +
NSM_max + NSM_sum + PAR_avg + PAR_max + PAR_sum + TLOC_avg + TLOC_max + TLOC_sum +
VG_avg + VG_max + VG_sum, data=train, family = "binomial")
     test.prob <- predict(model.glm, test, type="response")
     test.pred <- test.prob>=0.50

     outcome <- table(factor(test$post>0, levels=c(F,T)), factor(test.pred, le-
vels=c(F,T)))
     TN <- outcome[1,1]
     FN <- outcome[2,1]
     FP <- outcome[1,2]
     TP <- outcome[2,2]
     precision <- if (TP + FP ==0) { 1 } else { TP / (TP + FP) }
     recall <- TP / (TP + FN)
     accuracy <- (TP + TN) / (TN + FN + FP + TP)
     defects <- (TP + FN) / (TN + FN + FP + TP)
     return (c(defects, precision, recall, accuracy))
}


test_classification(files_20, files_20)
test_classification(files_20, files_21)
test_classification(files_20, files_30)
test_classification(files_21, files_20)
test_classification(files_21, files_21)
test_classification(files_21, files_30)
test_classification(files_30, files_20)
test_classification(files_30, files_21)
test_classification(files_30, files_30)
```

```
test_classification_pkg(packages_20, packages_20)
test_classification_pkg(packages_20, packages_21)
test_classification_pkg(packages_20, packages_30)
test_classification_pkg(packages_21, packages_20)
test_classification_pkg(packages_21, packages_21)
test_classification_pkg(packages_21, packages_30)
test_classification_pkg(packages_30, packages_20)
test_classification_pkg(packages_30, packages_21)
test_classification_pkg(packages_30, packages_30)
```

Step 5: Run the ranking experiments (Section 4.3, Table 6 and 7)

```
test_ranking <- function (train, test)
{
     model.lm <- lm(post ~ pre + ACD + FOUT_avg + FOUT_max + FOUT_sum + MLOC_avg +
MLOC_max + MLOC_sum + NBD_avg + NBD_max + NBD_sum + NOF_avg + NOF_max + NOF_sum + NOI
+ NOM_avg + NOM_max + NOM_sum + NOT + NSF_avg + NSF_max + NSF_sum + NSM_avg + NSM_max
+ NSM_sum + PAR_avg + PAR_max + PAR_sum + + + TLOC + VG_avg + VG_max + VG_sum, da-
ta=train)
     test.pred <- predict(model.lm, test)

     r.squared <- summary(model.lm)$r.squared
     pearson <- cor(test$post, test.pred, method="pearson")
     spearman <- cor(test$post, test.pred, method="spearman")
     pearson.p <- cor.test(test$post, test.pred, method="pearson")$p.value
     spearman.p <- cor.test(test$post, test.pred, method="spearman", ex-
act=FALSE)$p.value

     return (c(r.squared, pearson, spearman, pearson.p<0.01, spearman.p<0.01))
}

test_ranking_pkg <- function (train, test)
{
     model.lm <- lm(post ~ pre + ACD_avg + ACD_max + ACD_sum + FOUT_avg + FOUT_max +
FOUT_sum + MLOC_avg + MLOC_max + MLOC_sum + NBD_avg + NBD_max + NBD_sum + NOCU +
NOF_avg + NOF_max + NOF_sum + NOI_avg + NOI_max + NOI_sum + NOM_avg + NOM_max +
NOM_sum + NOT_avg + NOT_max + NOT_sum + NSF_avg + NSF_max + NSF_sum + NSM_avg +
NSM_max + NSM_sum + PAR_avg + PAR_max + PAR_sum + TLOC_avg + TLOC_max + TLOC_sum +
VG_avg + VG_max + VG_sum, data=train)
     test.pred <- predict(model.lm, test)

     r.squared <- summary(model.lm)$r.squared
     pearson <- cor(test$post, test.pred, method="pearson")
     spearman <- cor(test$post, test.pred, method="spearman")
     pearson.p <- cor.test(test$post, test.pred, method="pearson")$p.value
     spearman.p <- cor.test(test$post, test.pred, method="spearman", ex-
act=FALSE)$p.value

     return (c(r.squared, pearson, spearman, pearson.p<0.01, spearman.p<0.01))
}


test_ranking(files_20, files_20)
test_ranking(files_20, files_21)
test_ranking(files_20, files_30)
test_ranking(files_21, files_20)
test_ranking(files_21, files_21)
test_ranking(files_21, files_30)
test_ranking(files_30, files_20)
test_ranking(files_30, files_21)
test_ranking(files_30, files_30)
```

```
test_ranking_pkg(packages_20, packages_20)
test_ranking_pkg(packages_20, packages_21)
test_ranking_pkg(packages_20, packages_30)
test_ranking_pkg(packages_21, packages_20)
test_ranking_pkg(packages_21, packages_21)
test_ranking_pkg(packages_21, packages_30)
test_ranking_pkg(packages_30, packages_20)
test_ranking_pkg(packages_30, packages_21)
test_ranking_pkg(packages_30, packages_30)
```