

Providing Test Quality Feedback Using Static Source Code and Automatic Test Suite Metrics

Nachiappan Nagappan¹, Laurie Williams², Jason Osborne², Mladen Vouk², Pekka Abrahamsson³

¹ Microsoft Research, Redmond, WA 98052

nachin@microsoft.com

² North Carolina State University, Raleigh, NC 27695

lawilli3, jaosborn, vouk@ncsu.edu

³ VTT Technical Research Center of Finland, Oulu, FIN-90571

Pekka.Abrahamsson@vtt.fi

Abstract

A classic question in software development is ‘How much testing is enough?’ Aside from dynamic coverage-based metrics, there are few measures that can be used to provide guidance on the quality of an automatic test suite as development proceeds. This paper utilizes the Software Testing and Reliability Early Warning (STREW) static metric suite to provide a developer with indications of changes and additions to their automated unit test suite and code for added confidence that product quality will be high. Retrospective case studies to assess the utility of using the STREW metrics as a feedback mechanism were performed in academic, open source and industrial environments. The results indicate at statistically significant levels the ability of the STREW metrics to provide feedback on important attributes of an automatic test suite and corresponding code.

1. Introduction

Organizations spend a considerable amount of time and resources on software testing-related activities. Studies [15, 29] show that software testing activities accounts for 50% of the total software cost. Software testers need a means of receiving feedback on whether they have tested enough. Traditionally dynamic coverage metrics, such as statement and branch coverage, have been employed as indicators of the quality of a test suite. In this paper, we propose an alternative approach that utilizes static measures to provide such feedback. This static metrics based feedback can be obtained early and throughout the software development process.

Our approach is built upon the Software Testing and Reliability Early Warning (STREW) metric suite [22], a set of nine static source and test code measures. The STREW metric suite leverages automatic testing effort to estimate post-release field quality. Our research objective is to investigate the utility of the Software Testing and Reliability Early Warning (STREW) metric suite to provide a developer with indications of changes and additions to their automated unit test suite and code for added confidence that product quality will be high. Our approach is applicable for those developers who incrementally create an automatic unit test suite as code is implemented, as is done with the test-driven development [4] practice.

Case studies performed in academic, open source and large scale industrial environments indicate the efficacy of the nine STREW metrics to act, in aggregate, as early indicators of post-release field quality [22]. In this paper, we present an approach whereby the individual metrics are used to provide information to the developer on prudent actions to take to gain confidence in the quality of the system. Our approach is automated via an Eclipse plug-in, the Good Enough Reliability Tool (GERT) (<http://gert.sourceforge.net/>) [11] which is integrated into the programmer development environment. In GERT, (1) historical STREW metric standards from prior projects are saved; (2) STREW metrics for the current project are calculated and compared with these historical standards; and (3) the developer is provided with color-coded feedback on the comparison (In essence the feedback is on the difference between the individual STREW metrics for the current release compared to past releases).

The color-coded feedback, similar to prior studies [8, 16, 25], is based upon a Red-Orange-Green scheme for providing developer information on each metric relative to historical data. The Red, Orange, and Green colors are visual levels of discrimination often identified with bad, ok/acceptable, and good, respectively. The premise is that the developers are not as interested in the exact numerical difference between a current metric and historical value as they are in whether they need to take action. The color coding enables the developer to quickly understand specific actions to take on. This paper does not investigate the selection of Red, Orange and Green as appropriate colors (in terms of display factors) according to human computer interaction (HCI) studies.

In this paper, we investigate the use of these three levels of discrimination of STREW metrics. For this investigation, we ran retrospective case studies in academic, open source, and a structured industrial environment. Using statistical analysis, we combined the data from these studies to examine the efficacy of our approach.

The organization of the remainder of this paper is as follows. Section 2 discusses the related work, and Section 3 provides background on the STREW metric suite. Section 4 provides the research design. Section 5 presents the case studies and results. The limitations, conclusions and future work are discussed in Section 6 and 7.

2. Related Work

In this section we provide information previous work related to the use of color coding for metric feedback and on the relationship between metrics and software quality.

2.1 Color Coded Feedback

Providing feedback on important attributes of an automatic unit test suite allows the developers to identify areas that could benefit from design restructuring/more testing. Color coding [8, 16, 25] aids developers in understanding whether a metric is within acceptable limits. Our work on using color-coded feedback is motivated by prior studies at IBM [8] and Nortel Networks [16] that use color-coding to provide feedback on metric values based on standards (predefined or calculated). These studies also do not involve any HCI investigation but an analysis of the ability to provide in-process feedback. Their primary objectives of using color coding were also to have different levels of feedback.

The Enhanced Measurement for Early Risk Assessment of Latent Defects (Emerald) [16, 17] decision-support system at Nortel Networks combined software measurements, quality models and delivery of results to provide in-process feedback to developers to improve telecommunications software reliability. Emerald provides color-coded feedback to developers using nine categories: green, yellow and seven shades of red, based on acceptable values of non-OO metrics related to software system link volume, testability, decision complexity, and structuredness. A more detailed explanation is available in [16]. The Emerald system was shown to improve architectural integrity; establish design guidelines and limits; focus efforts on modules more likely to have faults; target the test effort effectively; identify patch-prone modules early; incorporate design strategies to account for the risk associated with defective patches; and help obtain a better understanding of field problems [16]. However, STREW differs from Emerald because STREW leverages the automated testing effort to estimate the post-release field quality based on test and source code metrics. Also STREW was developed for use by OO-languages.

Similarly at IBM, feedback on the complexity of Smalltalk methods, based on the source code allows developers to modify their code to more desirable characteristics in terms of the code complexity [8]. Color-coded feedback is presented in three levels, red, yellow and green using Smalltalk complexity metrics, such as number of blocks, number of temporary variables and arguments, number of parameterized expressions. STREW, similar to the work done at Nortel Networks and IBM, provides feedback in three ranges red, orange and green based on in-process metrics obtained from both the source and test code.

2.2 Software Metrics and Quality

The STREW metrics relate software quality with metrics. The following discussion investigates the prior work in this field. The relationship between product quality and process capability [27] and maturity has been recognized as a major issue in software engineering based on the premise that improvements in process will lead to higher quality products. The process capability is defined as the ability of a process to address the issue of stability, as defined and evaluated by trend or change [27]. A relationship between product quality and process capability should manifest itself via meaningful metrics that would exhibit trends and other characteristics that would be indicative of the stability of the process. Using the Space Shuttle software,

Schneidewind reports an assessment of long-term metrics, such as Mean Time to Failure (MTTF), total failures per thousand lines of code (KLOC) change in code (churn), total test time normalized by KLOC change in code, remaining failures normalized by KLOC, change in code, and predicted time to next failure to be indicative of the stability of the software process with respect to process capability [27].

Structural O-O measurements, such as those defined in the Chidamber-Kemerer (CK) [9] and MOOD [7] O-O metric suites, are being used to evaluate and predict the quality of software [14]. The CK metric suite consist of six metrics: weighted methods per class (WMC), coupling between objects (CBO), depth of inheritance (DIT), number of children (NOC), response for a class (RFC) and lack of cohesion among methods (LCOM). Structural object-orientation (O-O) measurements, such as those in the Chidamber-Kemerer (C-K) O-O metric suite [9], have been used to evaluate and predict fault-proneness [2, 5, 6]. Tang et al. [30] studied three real time systems for testing and maintenance defects. Higher WMC and RFC were found to be associated with fault-proneness. El Emam et al. [13] studied the effect of project size on fault-proneness by using a large telecommunications application. Size was found to confound the effect of all the metrics on fault-proneness. In addition to this, Chidamber et al.[9] analyzed project productivity, rework, and design effort of three financial services applications. High CBO and low LCOM were associated with lower productivity, greater rework, and greater design effort. To summarize, there is a growing body of empirical evidence that supports the theoretical validity of the use of these internal metrics [2, 5] as predictors of fault-proneness. The consistency of these findings varies with the programming language [28]. Therefore, the metrics are still open to debate [10].

3. STREW Metric Suite

The STREW Version 2.0 metric suite consists of nine constituent metric ratios, as shown in Table 1. The metrics are intended to cross-check each other and to triangulate upon an estimate of post-release field quality. Post-release field quality information is measured using Trouble Reports (TRs) per thousand lines of code (KLOC), an external measure obtained from users. A TR [21] is a customer-reported problem whereby the software system does not behave as the customer expects. Each STREW metric makes an individual contribution towards estimation of the post-release field quality but work best when used together. Development teams record the values of these nine

metrics and the actual TRs/KLOC of projects. These historical values from prior projects are used to build a regression model that is used to estimate the TRs/KLOC of the current project under development. For our case studies, the collected TRs were screened to remove duplicates and TRs involving documentation problems.

The use of the STREW metric suite is predicated on the existence of an extensive suite of automated unit test cases being created as development proceeds. These automated unit tests need to be structured as is done with the one of the object-oriented (O-O) xUnit testing frameworks, such as JUnit. The STREW method is not applicable for script-based automated testing because, as will be discussed, the metrics are primarily based upon the O-O programming paradigm. When these xUnit frameworks are used with O-O programming, both test code and implementation code hierarchies emerge. For each implementation source code class, there exists a corresponding test code class. Often each method/function in an implementation source code class will have one or more corresponding test code method(s)/functions(s). In industrial practice, often such perfect parallel class structure and one-to-one method/function correspondence is not observed. However, a test hierarchy which ultimately inherits from the *TestCase* class (the primary JUnit class) is created to exercise the implementation code.

The nine constituent STREW metrics (SM1 – SM9) and instructions for data collection and computation are shown in Table 1. The metrics can be categorized into three groups: test quantification metrics, complexity and O-O metrics, and a size adjustment metric.

The **test quantification metrics** (SM1, SM2, SM3, and SM4) are specifically intended to crosscheck each other to account for coding/testing styles. For example, one developer might write fewer test cases, each with multiple asserts [26] checking various conditions. Another developer might test the same conditions by writing many more test cases, each with only one assert. We intend for our metric suite to provide useful guidance to each of these developers without prescribing the style of writing the test cases. Assertions [26] are used in two of the metrics as a means for demonstrating that the program is behaving as expected and as an indication of how thoroughly the source classes have been tested on a per class level. SM4 serves as a control measure to counter the confounding effect of class size (as shown by El-Emam [13]) on the prediction efficiency. The **complexity and O-O metrics** (SM5, SM6, SM7, and SM8) examines the relative ratio of test to source code for control flow complexity and for a subset of the CK metrics.

Table 1: STREW metric elements

Test quantification	
Metric	ID
$\frac{\text{Number of Assertions}}{SLOC^*}$	SM1
$\frac{\text{Number of Test Cases}}{SLOC^*}$	SM2
$\frac{\text{Number of Assertions}}{\text{Number of Test Cases}}$	SM3
$\frac{(TLOC^+/SLOC^*)}{(\text{Number of Classes}_{Test} / \text{Number of Classes}_{Source})}$	SM4
Complexity and O-O metrics	
$\frac{\text{Cyclomatic Complexity}_{Test}}{\text{Cyclomatic Complexity}_{Source}}$	SM5
$\frac{CBO_{Test}}{CBO_{Source}}$	SM6
$\frac{DIT_{Test}}{DIT_{Source}}$	SM7
$\frac{WMC_{Test}}{WMC_{Source}}$	SM8
Size adjustment	
$\frac{SLOC^*}{\text{Minimum } SLOC^*}$	SM9
* Source Lines of Code (SLOC) is computed as non-blank, non-comment source lines of code + Test Lines of Code (TLOC) is computed as non-blank, non-comment test lines of code	

The dual hierarchy of the test and source code allows us to collect and relate these metrics for both test and source code. These relative ratios for a product under development can be compared with the historical values for prior comparable projects to indicate the relative complexity of the testing effort with respect to the source code. The metrics are now discussed more fully:

The *cyclomatic complexity* [20] metric for software systems is adapted from the classical graph theoretical cyclomatic number and can be defined as the number of linearly independent paths in a program. Prior studies have found a strong correlation between the cyclomatic complexity measure and the number of test defects [31]. Studies have also shown that code complexity correlates strongly with program size measured by lines of code [18] and is an indication of the extent to which control flow is used. The use of conditional statements increases the amount of testing required because there are more logic and data flow paths to be verified [19].

The larger the inter-object *coupling*, the higher the sensitivity to change [9]. Therefore, maintenance of the code is more difficult [9]. Prior studies have shown CBO has been shown to be related to fault-

prone [2, 5, 6]. As a result, the higher the inter-object class coupling, the more rigorous the testing should be [9]. A higher *DIT* indicates desirable reuse but adds to the complexity of the code because a change or a failure in a super class propagates down the inheritance tree. The relationship between the *DIT* and fault-prone [2, 5] was found to be strongly correlated.

The *number of methods* and the *complexity of methods* involved is a predictor of how much time and effort is required to develop and maintain the class [9]. The larger the number of methods in a class, the greater is the potential impact on children, since the children will inherit all the methods defined in the class. The ratio of the WMC_{test} and WMC_{source} measures the relative ratio of the number of test methods to source methods. This measure serves to compare the testing effort on a method basis. The relationship between the *WMC* as an indicator of fault-prone has been demonstrated in prior studies [2, 5]. The final metric is a **relative size adjustment factor**. Defect density has been shown to increase with class size [13]. We account project size in terms of *SLOC* for the projects used to build the STREW prediction equation using the size adjustment factor.

4. Research Design

Section 4.1 describes the building of the color-coded feedback standards, and Section 4.2 the evaluation of the test quality feedback standards.

4.1 Test Quality Feedback Standards

Using historical data collected from previous completed comparable projects that were successful, the lower limit (LL) of each metric ratio is calculated using Equation 1. The color coding is determined by the results of this calculation. The use of this equation is predicated on a normal distribution (The Kolmogorov-Smirnov test to check for normality) of TRs. If the TRs are not normally distributed, the Box-Cox normal transformation can be used to transform the non-normal data into normal form [24]. The mean of the historical values for each metric (μ_{SMX}) serves as the upper limit. The historical data is computed from previously-successful projects with acceptable levels of TRs/KLOC. In the absence of historical data, standard values can be used that are built from projects with similar acceptable levels of TRs/KLOC. The mean and the lower limit serve as the test quality feedback standards for the STREW metrics.

$$LL(SMX) = \mu_{SMX} - z_{1/2} \frac{S.D.SMX}{\sqrt{n}} \quad (1)$$

where μ_{SMx} is the Mean of Metric SMx (SM1, SM2 ... as shown in Table 1); n is the number of samples used to calculate μ_{SMx} ; S.D._{SMx} is the standard deviation of metric SMx; and $Z_{\alpha/2}$ is the upper $\alpha/2$ quantile of the standard normal distribution.

Using the computed values, we determine the color with which to code the metric, as shown in Table 2. SMx refers to the value of each particular STREW metric for the software system under development. This value is compared with the LL as computed by Equation 1 and with the average value (μ_{SMx}).

Table 3: STREW metric color coded feedback explanation

Metric	Meaning of RED or ORANGE	Corrective Action
SM1	The assertions/KLOC are lower compared to projects that were used to build the feedback standards.	Add more assertions. These additional assertions should be meaningful assertions (can be cross checked with increase in coverage)
SM2	The test cases/KLOC are lower compared to projects that were used to build the feedback standards.	Add more test cases. These test case density but should be meaningful test cases (can be cross checked with increase in coverage)
SM3	The assertions/test case are lower compared to projects that were used to build the feedback standards.	Add more assertions per test case.
SM4	The overall testing effort measured in terms of lines of code and classes was not comparable in terms of the projects that were used to build the feedback standards.	Add more lines of test code. Increasing the overall testing effort (controlling the source code size) that would lead to an increase in the test lines of code and the test classes.
M5	The complexity of the testing effort is not comparable in terms of the previously-acceptable projects	Decrease cyclomatic complexity ratio by decreasing the cyclomatic complexity of source code. The simplest complexity measure, the cyclomatic complexity measure, ratio of the test code and source code indicates how well the testing has taken place at the complexity level. This metric works in conjunction with metric SM4 indicating that the testing effort was not thorough enough. When metric SM4 increases, metric SM5 should also increase with respect to checking for conditionals, infinite loops, unreachable code etc., that would increase the metric SM5
SM6	The CBO ratio of the test code to source code is not comparable to the previously successful projects.	Increase the CBO ratio by reducing the coupling between objects in source code. The larger the inter-object coupling, the higher the sensitivity to change. Therefore, maintenance of the code is more difficult. As a result, the higher the inter-object class coupling, the more rigorous the testing should be [9]. The CBO of the source code should be reduced to eliminate the dependencies caused by coupling.
SM7	The DIT ratio of the test code to source code is not comparable to the previously successful projects.	Increase DIT ratio by reducing DIT in the source code. Due to DIT, a change or a failure in a super class propagates down the inheritance tree. Hence we have to reduce the DIT of the source code to acceptable levels so that the DIT ratio is comparable to the projects used to build the standards.
SM8	The WMC ratio of the test code to source code is not comparable to the previously successful projects.	Increase WMC ratio by breaking down complex classes in source code to child classes. The larger the number of methods in a class, the greater is the potential impact on children, since the children will inherit all the methods defined in the class. Hence we have to reduce the number of methods in the source code to acceptable levels calculated by the projects used to build the standard. This metric essentially measures the importance of the modularity of the source code.
SM9	SM9 is a size adjustment factor for the regression equation and does not provide corrective action direction to the programmer.	

Table 2: Color coded feedback standards

Color	Interpretation
RED	$SMx < LL (Metric)$
ORANGE	$LL (Metric) \leq SMx \leq \mu_{SMx}$
GREEN	$SMx > \mu_{SMx}$

Table 3 provides detail on the specific corrective actions the programmer can take to change a metric from red to orange or from orange to green. The explanation of all these metrics is with respect to the standards built for each metric.

4.2 Evaluation methodology

To evaluate the efficacy of the test quality feedback standards, we initially use a robust Spearman rank correlation technique with respect to the number of metrics which were coded in each of the colors and the post-release TRs/KLOC. The desired correlation results between the test quality feedback and post-release field quality that would indicate the efficacy of the color-coded mechanism is shown in Table 4. In Section 5, we investigate the overall efficacy of the test quality feedback across three development environments (academic, open source and industrial).

Table 4: Desired correlation results with the color-coded feedback

Color	Desired Spearman Correlation	Interpretation
RED + ORANGE	Positive	Increase in (RED + ORANGE) increases post-release TRs/KLOC
GREEN	Negative	Increase in (GREEN) decreases post-release TRs/KLOC

The higher the number of metrics coded as red and orange, the higher we expect the TRs/KLOC to be, as red and orange metrics denote a variation from previous testing efforts. If our color-coding works as planned, a positive correlation coefficient would exist between the number of red and orange metrics with the post-release TRs/KLOC. Inversely, the correlation coefficient between the number of green metrics and the TRs/KLOC would be negative. We analyzed the color assigned to each of the metric ratios SM1-SM8. SM9 is a size adjustment factor for the regression equation and does not provide corrective action direction to the programmer.

5 Case Studies

To investigate the effectiveness of the color-coded feedback, retrospective case studies were carried out from three environments; academic (A), open source (O) and

industrial (I). (Appendix A provides a detailed discussion of these case studies). Several linear models were considered to predict trouble reports from red-plus-orange counts and the environment. Initially, a significant positive sample correlation coefficient ($r = 0.49$; $p = 0.01$) indicates that a simple linear regression with the counts as the predictor explains 24% (R^2) of the variability in trouble report density (The coefficient of determination, R^2 , 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). Because of small sample sizes ($n_A = 7$; $n_O = 13$; $n_I = 6$), the correlation coefficient is significant only when data are pooled across environments and not when investigated separately for each. However, there is evidence of differences between the feedback measurements across the environments. A test for equality of mean red/orange counts across environments in an analysis of variance is highly significant ($F = 12.5$; $p = 0.0002$). For this reason, we also consider models which account for environment effects by allowing lines with different slopes or different intercepts across environments. In particular, we consider the following four nested models for the expected trouble reports response, $E[TRs/KLOC]$. In these models, RpO denotes the red-plus-orange count and ENV_i is an indicator variable taking the value 1 if the observation comes from environment 'I' and taking the value 0 otherwise. A point to be noted is that if the analysis is performed using greens correspondingly reverse results would be obtained as the number of greens is a linear function of the number of red's and oranges.

$$M_0 : E [TRs/KLOC] = \beta_0$$

$$M_1 : E [TRs/KLOC] = \beta_0 + \beta_1RpO$$

$$M_2 : E [TRs/KLOC] = \beta_0 + \beta_1RpO + \beta_2ENV_2 + \beta_3ENV_3$$

$$M_3 : E [TRs/KLOC] = \beta_0 + \beta_1RpO + \beta_2ENV_2 + \beta_3ENV_3 + \beta_4RpO \times ENV_2 + \beta_5RpO \times ENV_3$$

Table 5 summarizes the fits of the models. The coefficient of determination R^2 increases slightly as more and more complex models are considered. However, though models M2 and M3 explain more of the variability in TRs/KLOC, the F-tests to compare them with the nested simple linear regression model are not significant, so that environment-specific coefficients in the models do not differ significantly from 0. The models are sequentially nested, so that an F-test may be used to compare any model with any other model that comes below it in the table. The only such comparison of nested models which is found to be significant compares M_0 and M_1 , $F=7.52(p=0.011)$. Attempting to allow for environment-specific slopes and intercepts from the

limited information contained in the sample leads to over fitting the model. In conclusion, a model in which TRs/KLOC are linearly increasing in RpO counts, with the increase being constant across environments, is entirely plausible and well-determined in light of this model selection

Table 5: Model fits

Model	Description of mean TRs/KLOC	Model degrees of freedom	R ²	Root MSE
M ₀	Constant	0	.	2.18
M ₁	Simple linear regression	1	0.24	1.94
M ₂	Different intercepts	3	0.35	1.87
M ₃	Different slopes, intercepts		0.37	1.93

A scatter plot of the trouble report responses against the color-coded feedback appears in Figure 1. In the Figure, different characters are used for the different environments. The least squares regression line corresponding to model M₁ is overlaid on this plot. The linear association can be seen upon inspection of the plot, but there is also some noise.

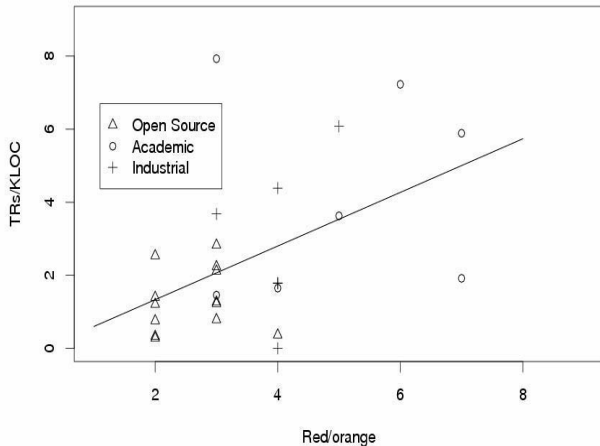


Figure 1: Scatter plot of TRs

Upon removal of one apparent outlying observation which had the largest observed trouble report ratio, the sample correlation jumps from $r = 0.49$ to $r = 0.62$, providing even stronger evidence of a strong, positive linear association between the color-coded feedback and fault proneness. Further, the assessment of variability of trouble reports also decreases from $\sqrt{MSE} = 1.94$ TRs/KLOC to $\sqrt{MSE} = 1.53$ TRs/KLOC. A summary of model M₀ with and without this extreme observation appears in Table 6. Plots of the residuals from this model do not indicate any non-normality or lack-of-fit, with the slight exception of the outlier.

Table 6: With and without outlier data fit

Regression data	Least square regression line	R ²
Outlier included	$-0.13 + 0.73$ (RpO)	0.2385
Outlier ignored	$-0.61 + 0.8$ (RpO)	0.3819

To characterize the extremity of outlying observations, diagnostics can be carried out on the residuals from the regression model. The “deleted residual” is the difference between the observed TR’s/KLOC value and the value predicted by the fitted regression where that observation is ignored, or deleted from the dataset. The “studentized” deleted residual, also called the externally studentized residual, is obtained by dividing the residual by the standard error of prediction corresponding to that value of the predictor (in this case, 3 RED/ORANGE flags). Here, the deleted residual is 6.14 (7.93-1.79), with a prediction standard error of 1.57, leading to the studentized value of 3.91. For a formal procedure to identify the observation as outlying, the studentized residual may be compared to the appropriate Bonferroni critical value from a t-distribution, $t(.025/24,22)=3.48$. Here the Bonferroni correction allows for the possibility that any of the 24 observations might be an outlier. It can be noted here that if studentized deleted residuals are similarly computed for the other 23 observations, none exceed two in absolute value and therefore none may be classified as outliers in this manner. Thus we are able to identify this externally studentized residual as an outlier.

6. Limitations

Drawing general conclusions from empirical studies in software engineering is difficult because 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 [3]. Researchers become more confident in a theory when similar findings emerge in different contexts [3]. By performing multiple case studies and/or experiments and recording the context variables of each case study, researchers can build up knowledge through a family of experiments [3] which examine the efficacy of a new practice. Replication of experiments addresses threats to experimental validity. We address these issues related to empirical studies by replicating multiple case studies through a family of experiments in three different (academic, open source and industrial) contexts. Similar results in these contexts indicate the promise of our approach.

Two other limitations in our investigation of test quality feedback are, the small size of the samples (seven academic, 13 open source, and six industrial) which make it difficult to obtain an individual statistical significance

at a high 95% confidence. Second, the analyses were all done post-mortem, i.e. the feedback results were not used to improve the testing effort as development proceeded. This post-mortem feedback explains to a certain degree the similar quantity of metrics coded in each color as shown in Figure 3 in Appendix A.

7. Conclusions and Future Work

Feedback on important attributes of a software testing effort can be useful to developers because it helps identify weaknesses and the completeness of testing phase. In this paper we have reported on the use of the STREW measures for providing such a test quality feedback in a controlled academic, open source and industrial environment. The results indicate the efficacy of the STREW metric suite to provide meaningful feedback on the quality of the testing effort.

Further, for providing test quality feedback, we have automated the collection and analysis of statement and branch coverage and an earlier version of the STREW metrics suite via an open source Eclipse plug-in GERT (Good Enough Reliability Tool) [12, 23]. We are updating the tool to reflect the current version of the STREW metric suite. We also plan to use the test quality feedback standards in-process in industrial organizations and to study the benefits of early feedback on the quality of the testing effort.

References

- [1] P. Abrahamsson, Koskela, J., "Extreme Programming: A Survey of Empirical Data from a Controlled Case Study", Proceedings of International Symposium on Empirical Software Engineering, pp. 73-82, 2004.
- [2] V. Basili, Briand, L., Melo, W., "A Validation of Object Oriented Design Metrics as Quality Indicators", *IEEE Transactions on Software Engineering*, 22(10), pp. 751 - 761, 1996.
- [3] V. Basili, Shull, F., Lanubile, F., "Building Knowledge through Families of Experiments", *IEEE Transactions on Software Engineering*, 25(4), pp. 456-473, 1999.
- [4] K. Beck, *Test Driven Development -- by Example*. Boston: Addison Wesley, 2003.
- [5] L. C. Briand, Wuest, J., Daly, J.W., Porter, D.V., "Exploring the Relationship between Design Measures and Software Quality in Object Oriented Systems", *Journal of Systems and Software*, 51(3), pp. 245-273, 2000.
- [6] L. C. Briand, Wuest, J., Ikonovskii, S., Lounis, H., "Investigating Quality Factors in Object-Oriented Designs: An Industrial Case Study", Proceedings of International Conference on Software Engineering, pp. 345-354, 1999.
- [7] F. Brito e Abreu, "The MOOD Metrics Set", Proceedings of ECOOP '95 Workshop on Metrics, pp. 1995.
- [8] S. L. Burbeck, "Real-time complexity metrics for Smalltalk methods", *IBM Systems Journal*, 35(2), pp. 204-226, 1996.
- [9] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, 20(6), 1994.
- [10] N. I. Churcher and M. J. Shepperd, "Comments on 'A Metrics Suite for Object-Oriented Design'", *IEEE Transactions on Software Engineering*, 21(3), pp. 263-5, 1995.
- [11] M. Davidsson, J. Zheng, N. Nagappan, L. Williams, and M. Vouk, "GERT: An Empirical Reliability Estimation and Testing Feedback Tool", Proceedings of International Conference on Software Reliability Engineering, Saint-Malo, France, pp. 2004.
- [12] M. Davidsson, Zheng, J., Nagappan, N., Williams, L., Vouk, M., "GERT: An Empirical Reliability Estimation and Testing Feedback Tool", Proceedings of International Symposium on Software Reliability Engineering, St. Malo, France, pp. 269-280, 2004.
- [13] K. El Emam, Benlarbi, S., Goel, N., Rai, S.N., "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics", *IEEE Transactions on Software Engineering*, 27(7), pp. 630 - 650, 2001.
- [14] R. Harrison, S. J. Counsell, and R. V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics", *IEEE Transactions on Software Engineering*, 24(6), pp. 491-496, June 1998.
- [15] M. Harrold, "Testing: A Roadmap", Proceedings of International Conference on Software Engineering, Limerick, Ireland, pp. 61-72, 2000.
- [16] J. Hudepohl, S. J. Aud, T. Khoshgoftaar, E. B. Allen, and J. Mayrand, "Emerald: Software Metrics and Models on the Desktop", *IEEE Software*, 13(5), pp. 56-59, September 1996.
- [17] J. P. A. Hudepohl, S.J.; Khoshgoftaar, T.M.; Allen, E.B.; Mayrand, J., "Integrating Metrics and Models for Software Risk Assessment", Proceedings of Seventh International Symposium on Software Reliability Engineering, pp. 93-98, 1996.
- [18] S. H. Kan, *Metrics and Models in Software Quality Engineering*. Reading, MA: Addison-Wesley, 1995.
- [19] T. M. Khoshgoftaar, Munson, J.C., "Predicting software development errors using software complexity metrics", *IEEE Journal on Selected Areas in Communications*, 8(2), pp. 253-261, 1990.
- [20] T. J. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, 2(4), pp. 308-320, 1976.

- [21] P. Mohagheghi, Conradi, R., Killi, O.M., Schwarz, H., "An Empirical Study of Software Reuse vs. Reliability and Stability", Proceedings of International Conference on Software Engineering, pp. 282-292, 2004.
- [22] N. Nagappan, "A Software Testing and Reliability Early Warning (STREW) Metric Suite," in *Computer Science Department*. PhD Thesis, Raleigh: North Carolina State University, 2005.
- [23] N. Nagappan, Williams, L., Vouk M.A., "'Good Enough" Software Reliability Estimation Plug-in for Eclipse", Proceedings of IBM-ETX Workshop, in conjunction with OOPSLA 2003, pp. 36-40, 2003.
- [24] NIST/SEMATECH, *e-Handbook of Statistical Methods*: <http://www.itl.nist.gov/div898/handbook/>.
- [25] M. C. Ohlsson, Wohlin, C., "Identification of Green, Yellow and Red Legacy Components", Proceedings of International Conference on Software Maintenance, pp. 6-15, 1998.
- [26] D. S. Rosenblum, "A practical approach to programming with assertions", *IEEE Transactions on Software Engineering*, 21(1), pp. 19-31, 1995.
- [27] N. F. Schneidewind, "An integrated process and product model", Proceedings of Fifth International Software Metrics Symposium, pp. 224-234, 1998.
- [28] R. Subramanyam, Krishnan, M.S., "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects", *IEEE Transactions on Software Engineering*, 29(4), pp. 297 - 310, 2003.
- [29] B. V. Tahat, B. Korel, and A. Bader, "Requirement-Based Automated Black-Box Test Generation", Proceedings of 25th Annual International Computer Software and Applications Conference, Chicago, Illinois, pp. 489-495, 2001.
- [30] M.-H. Tang, Kao, M.-H., Chen, M.-H., "An empirical study on object-oriented metrics", Proceedings of Sixth International Software Metrics Symposium, pp. 242-249, 1999.
- [31] J. Troster, "Assessing Design-Quality Metrics on Legacy Software," Software Engineering Process Group, IBM Canada Ltd. Laboratory, North York, Ontario 1992.

Appendix A: Case Study Details

In the following three sub-sections, details of the academic, open source and industrial case studies are provided.

A.1 Academic feasibility study

We performed a controlled feasibility with junior/senior-level students at North Carolina State University (NCSSU). The students worked on a project that involved development of an Eclipse plug-in to collect software metrics. The project was six weeks in duration

and used Java as the programming language. The JUnit testing framework was used for unit testing; students were required to have 80% statement coverage. A total of 22 projects were submitted, and each group had four to five students. The projects were between 617 LOC_{source} and 3,631 LOC_{source}. On average, the ratio of LOC_{test} to LOC_{source} was 0.35. Each project was evaluated by 45 independent test cases. Actual TRs/KLOC was estimated by test case failures because the student projects were not released to customers.

We used the technique of random splitting to build our test feedback standards from 15 randomly-selected academic projects and evaluated the built standards using the seven remaining projects.

A.2 Open source case study

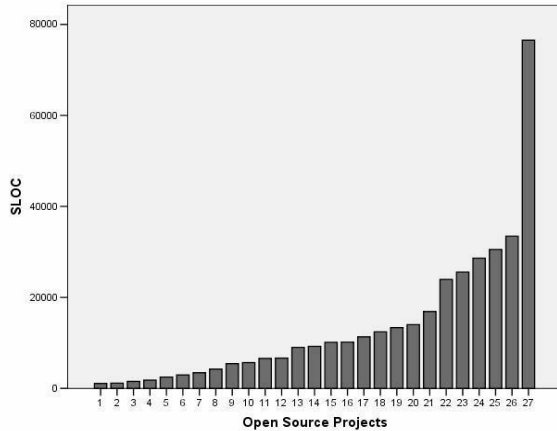
To build the standards, shown in Table 7 twenty-seven open source projects that were developed in Java were selected from Sourceforge (<http://sourceforge.net>). The following criterion was used to select the projects from Sourceforge.

- *software development tools*. All of the chosen projects are software development tools, i.e. tools that are used to build and test software and to detect defects in software systems.
- *download ranking of 85% or higher*. In Sourceforge, the projects are all ranked based on their downloads on a percentile scale from 0-100%. For example, a ranking of 85% means that a product is in the top 85% of quantity of downloads. We chose this criterion because we reasoned that a comparative group of projects with similarly high download rates would be more likely to have a similar usage frequency by customers that would ultimately reflect the post-release field quality.
- *automated unit testing*. The projects needed to have JUnit automated tests.
- *defect logs available*. The defect log needed to be available for identifying TRs with the date of the TR reported.
- *active fixing of TRs*. The TR fixing rate is used to indicate the system is still in use. The time between the reporting of a TR and the developer fixing it serves as a measure of this factor. Projects that had open TRs that were not assigned to anyone over a period of three months were not considered.
- *Sourceforge development stage of 4 or higher*. This denotes the development stage of the project (1-6) where 1 is a planning stage and 6 is a mature phase. We chose a cut-off of 4 which indicates the project is at least a successful beta release. This criterion indicates that the projects that are at a similar stage of development and are not projects too early in the development lifecycle.

Table 7: Test quality feedback standards

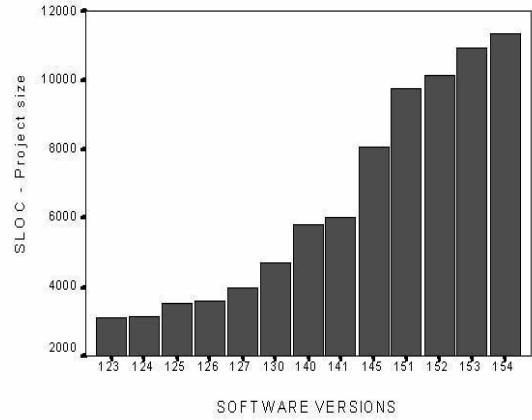
Metric	RED	ORANGE	GREEN
SM1	< 0.0545	[0.0545, 0.0822]	> 0.0822
SM2	< 0.0721	[0.0721, 0.1261]	> 0.1261
SM3	< 0.7790	[0.7790, 1.0958]	> 1.0958
SM4	< 0.7881	[0.7881, 1.0427]	> 1.0427
SM5	< 0.2474	[0.2474, 0.3376]	> 0.3376
SM6	< 0.3417	[0.3417, 0.4490]	> 0.4490
SM7	< 0.3498	[0.3498, 0.4931]	> 0.4931
SM8	< 0.2349	[0.2349, 0.3217]	> 0.3217

Figure 2 shows the names of the projects and their sizes in LOC_{source}. On average, the ratio of LOC_{test} to LOC_{source} was 0.37. The projects range from around 2.5 KLOC_{source} to 80 KLOC_{source}. The TRs are normally distributed with a range from 0.20 to 6.9 TRs/KLOC (Mean=1.42). The defect logs were screened for duplicate TRs to obtain an accurate measure of TRs/KLOC.

**Figure 2: Open source project sizes**

We used 13 versions of *htpunit*, one of the 27 open source software projects used in our earlier analysis to evaluate the test quality feedback standards. The system has a lifetime in use of three years. Each release has a time period of 2.5-3 months so that the TRs/KLOC collected are representative of equal usage. The test quality feedback standards were calculated using the 27 open source projects, as discussed earlier. The test quality feedback for the 13 versions was evaluated against the

standards built using the 27 open source projects. Figure 3 shows the size of the 13 versions as they grew over a period of three years from almost 3 KLOC to 11.5 KLOC.

**Figure 3: htpunit Project size (LOC_{source})**

A.3 Structured industrial case study

We analyzed a structured industrial case study to investigate the results under a more controlled environment. Six releases of a commercial software system “eXpert” were analyzed. A metaphor that describes the intended purpose of the system is a large sized “virtual file cabinet,” which holds a number of organized rich, i.e. annotated, links to physical or web-based resources [1]. The system has 300+ potential users and is a Java, web-based client-server solution developed by four developers at VTT Technical Research Centre of Finland [1]. The four developers were 5-6th year university students with 1-4 years of industrial experience in software development. Team members were well-versed in the Java O-O analysis and design approaches. The overall development time was 2.1 months and post-release TRs/KLOC was available from failure logs. The color categorization for the six releases of the industrial software system were calculated using the standards built from the open source projects described in Section A.2. Measures such as effort and time were collected to make sure that these factors were in a comparable level across all the releases. This data is presented in Table 8 [1].

Table 8: Industrial project data description (adapted from [1])

No.	Collected Data	Release 1	Release 2	Release 3	Release 4	Release 5	Release 6 (correction phase)	Total
1.	Calendar time	2	2	2	1	1	0.4	8.4
2.	Total work effort (h)	195	190	192	111	96	36	820
3.	LOC per release	1821	2386	1962	460	842	227	7698
4.	TRs	4	5	4	4	11	0	28
5.	TRs/KLOC	2.19	2.10	2.04	8.70	13.06	0.0	1.43