Prediction = Power

Elaine J. Weyuker

AT&T Labs - Research
180 Park Avenue, Florham Park, NJ 07932
weyuker@research.att.com

Abstract. An argument is made that predictive metrics provide a very powerful means for organizations to assess characteristics of their software systems and allow them to make critical decisions based on the value computed. Five different predictors are discussed aimed at different stages of the software lifecycle ranging from a metric that is based on an architecture review which is done at the earliest stages of development, before even low-level design has begun, to one designed to predict the risk of releasing a system in its current form. Other predictors discussed include the identification of characteristics of files that are likely to be particularly fault-prone, a metric to help a tester charged with regression testing to determine whether or not a particular selective regression testing algorithm is likely to be cost effective to run on a given software system and test suite, and a metric to help determine whether a system is likely to be able to handle a significantly increased workload while maintaining acceptable performance levels.

Keywords: Architecture review, fault-prone, metrics, prediction, regression testing, risk, scalability.

1 Introduction

If we had a crystal ball and knew somehow that certain files in a large software system were going to be particularly problematic in some way, think about what that would allow us to do. If the nature of the problems were incorrect behavior, we could focus our functional testing resources there, increasing the chances of identifying and removing the problems, thereby having a more dependable system than we would otherwise have. It would also probably mean that we could do the testing more economically because we could target our testing efforts to just those problematic files.

Similarly, if the nature of the problems were related to system performance, we could focus our performance testing efforts to identify potential bottlenecks so that workloads could be balanced or distributed more appropriately. In this way, users would never have to wait in long queues or experience unacceptable delays. Undoubtedly you would have happier customers, and likely more customers.

What other sorts of things would we like to be able to predict? What if you were releasing a new product? Initially it would likely have a small user base. Typically when you do performance testing, you test a system's behavior for

D. Hogrefe and A. Wiles (Eds.): TestCom 2003, LNCS 2644, pp. 1–9, 2003.

the initial expected workload, and also a slightly enhanced one. But what if the product were to take off and you now had one or more orders of magnitude more customers? Could you predict how the system would behave under those circumstances and determine when and how many additional servers would be needed? Could you predict whether there was a point at which adding more servers would actually lead to decreased rather than increased throughput, and therefore the system might have to be re-architected? If we could plan for the increased workload by predicting when these significantly increased workloads were likely to occur, we could order new hardware before performance became unacceptable or begin re-architecture so that the customers never saw any negative impact of the increased workload.

When creating a new product, there is often a tradeoff between being first or early to market and improving the dependability of the system. Is it better to get there early and get a large share of the market, or offer the product later with a higher level of reliability than would be possible were you to release the software now? What are the risks associated with each of these scenarios? If we could predict the risk, or expected loss, associated with the release of a software system, we could make informed decisions about whether or not it is wise to release the product in its current state, understanding fully the consequences of whichever decision is made.

Being able to predict these and other related system characteristics can an have enormous impact on the observed dependability of a software system, the cost to deliver the system, and even whether or not the system can be viably produced. In this paper we will examine what sorts of characteristics can be reasonably predicted and how these predictions might be made. We also provide pointers to the literature describing empirical studies used to determine the predictors of interest, as well as studies that show applicability of these predictors for large, industrial software systems as well as the usefulness of these predictors in practice.

In Section 2, we look at predicting the likely quality of the ultimate software system produced by considering the results of architecture reviews that are done well before any implementation has begun. Section 3 examines ways of determining which files in a software system are likely to be particularly fault-prone, and therefore good candidates for concentrating testing resources. In Section 4, we discuss a way of predicting whether or not it is likely to be cost-effective to use a selective regression testing algorithm to try to minimize the regression test suite. If the cost of running the algorithm is high, or the reduction in test suite size is low, it may be more cost-effective to simply rerun as much of the regression test suite as possible rather than using some of those scarce resources of fine-grained analysis with little payoff. Section 5 describes ways of predicting when a system's workload is likely to increase significantly so that steps can be taken to prepare for this situation, thereby assuring that customers always experience acceptable performance, while in Section 6, we discuss how risk can be predicted so that projects can make informed decisions about whether or not it is safe to release a software system in its current state.

2 Architecture Reviews

Architecture is defined as "the blueprint of a software/hardware system. The architecture describes the components that make up the system, the mechanisms for communication between the processes, the information content or messages to be transmitted between processes, the input into the system, the output from the system, and the hardware that the system will run on." [12]

Architecture reviews are performed very early in a software system's lifecycle. [12,1,7] It is a standard part of the quality assessment process at AT&T and many other organizations that produce software systems that need to be highly dependable and always available. As soon as the system's requirements and high-level design have been completed, a review is done to assure that the architecture is complete, and that low-level design can begin. It is part of the standard wisdom of the software engineering community that the earlier in the lifecycle problems are identified, the easier and cheaper they are to rectify, and these reviews strive to help identify problems at a very early stage so that they do not negatively impact the system.

The goal of an architecture review, then, is to assess the quality and completeness of the proposed architecture. An obvious extension, therefore, is to use this assessment as a way of predicting the likely quality of the ultimately implemented system, and determining its likelihood of failure. Of course, assuring that the architecture is complete and sound does *not* guarantee that problems will not enter the system during the low-level design, coding, or even testing of the system, but it does assure that the foundation on which these later stages depend is itself appropriate.

For these reasons, Avritzer and Weyuker proposed a way of predicting likely project success based on the results of an architecture review. They developed a questionnaire-based metric that computed a score and provided five ranges indicating whether a project was at low, moderate-low, moderate, moderate-high, or high risk of failure. They used the results of 50 industrial architecture audits performed over a period of two years to create the questionnaire, and then selected seven projects for which they computed the metric. They compared the metric's assessment with the assessment done by several senior personnel familiar with the systems, which at that point had all been in production. They found that the informal and formal assessments matched for six of the seven systems. [3,4] The seventh project was assessed by the metric as being at moderate risk of failure, while knowledgeable personnel rated it as an excellent system, with very few defects identified in the field. It turned out that the review done for this project was not a true architecture review, but a review done at an even earlier stage, known as a discovery review. Although this sort of review is similar to an architecture review, it is done before the architecture is complete. Its goal is to help a project make decisions and weigh benefits and risks of potential architectural decisions. Therefore, it was appropriate at this stage that certain portions of the architecture were incomplete, and did not indicate potential problems. The results of this case study indicated that it was worthwhile applying this metric for all projects once they had completed an architecture review.

For this reason, Weyuker extended this work and did a larger empirical study limiting consideration to projects during architecture reviews (rather than including the results from projects' discovery reviews.) [19] One of the limitations of the Avritzer-Weyuker metric, however, was that it was difficult and time-consuming to compute because it involved a careful assessment of whether or not the project suffered from any of fifty-four of the most severe problems identified most frequently in the original set of 50 systems studied. For each of the problems, a score had to be assigned indicating the severity of the problem in this instance. Therefore, Weyuker proposed a highly simplified metric that did not require that a questionnaire be completed, and performed a new case study using thirty-six systems for which the results of architecture reviews were present. Of the 36 projects assessed, only one seemed to be at greater risk of failure than the simple metric indicated. It was therefore recommended that projects use this metric to predict their likelihood of success based on the results of their architecture review. In this way, if there is a prediction of a moderate to high risk of failure, there should be sufficient time to modify the architecture to correct problems or complete missing portions so that the project does not move forward with low-level design until a re-review and re-application of the metric indicates that the project has a low risk of failure.

3 Fault-Proneness

Testing software systems for correctness and functionality is an essential and expensive process. It is often estimated that testing and related activities consume as much or more resources as the implementation of the system. Even if that is an exaggeration, there is no doubt that testing a large industrial software system takes a substantial amount of time and money, and that the consequences of doing a poor or inadequate job of testing can be catastrophic. But it is also difficult to do a comprehensive job of testing. For many systems, the input domain is enormous and it is impossible to execute even a very small percentage of the possible inputs during testing. Therefore it is essential that there be a systematic approach to testing and that there be some way of prioritizing potential test cases. Similarly, a large software system may be comprised of many thousands of files, and a decision has to be made which of these files should be most thoroughly exercised, and which can be only lightly tested.

For this reason, if we could identify which files were likely to be most fault-prone, we could give testers a way to allocate testing resources and to prioritize their testing activities. With this goal in mind, a number of research teams have performed empirical studies in recent years to try to identify properties of files that make them particularly fault-prone. [2,6,8,9,10,13,14,15] Adams [2], Fenton and Ohlsson [8], Munson and Khoshgoftaar [14], and Ostrand and Weyuker [15] have all observed seeing a very uneven distribution of faults among files within a system. Generally, a very small number of files accounted for the vast majority of the faults. If those files could be identified, then testing effort could be concentrated there, leading to highly dependable systems, with less testing expense.

Given that there tends to be a very uneven distribution of faults among files, a variety of file characteristics have been examined in these empirical studies as possible identifying factors. Some of these characteristics include file size, whether a file is new or appeared in an earlier release, the age of the file, whether earlier versions of the file contained a large number of faults, whether a large number of faults were identified during earlier stages of development, and the number of changes made to a file. Some initial indicators have been identified, but significantly more carefully-done case studies must be prepared.

In a recent empirical study using 13 releases of an industrial inventory tracking system, Ostrand and Weyuker [15] found that among the predictors they considered, the best predictors of fault-proneness involved considering whether the file had been newly-written, and whether it had had a particularly large number of faults in the previous release. Both were shown be promising predictors of high fault densities in the current release.

In a more recent study, they found that if they refined the earlier question about whether the file was new or old, and distinguished between old files that had been changed and those that had not been changed, they found that new files did not always have a higher average fault density than files that appeared in earlier releases. They found, instead, that in more than half of the releases, files that had been changed since the prior release had higher average fault densities than new files, although, for every release, old files that remained unchanged from the prior release always had lower average fault densities than either newly-written files or old, changed files.

Their new research also found that the age of a file was not a particularly good predictor of fault-proneness. For every release studied, they found some cases for which newer files had lower average fault densities than older ones. They therefore concluded that age does not seem to be a good predictor of fault-proneness.

Finally, they examined the question of whether as a system matured, the average fault density of later releases was lower than earlier releases. They did observe a general downward trend, although this value did not decrease monotonically. Hopefully, many other research groups will continue to explore characteristics of files that are particularly fault-prone. If we can learn to make predictions of this nature with any degree of accuracy, there should be an enormous payoff for any organization concerned about producing dependable software systems at reasonable costs.

4 Regression Testing

Whenever a software system has been changed, whether the purpose of the change is to fix defects, enhance the functionality, change the functionality, or any other purpose, there is always the possibility that the fixes or changes have introduced new defects into the product. For this reason, whenever a change is made to the software, the system must be retested. This is known as *regression testing* and consists of rerunning previously-run test cases to make sure

that those test cases that performed correctly before the changes, still perform correctly. Since test suites for industrial software systems are often very large, and therefore it is impractical to rerun the entire test suite, *selective regression testing* algorithms have been proposed in order to have an efficient way to select a relatively small subset of the entire test suite. The goal is to select just those test cases to rerun that are relevant to the changes that have been made. By doing this, it is hoped that a significantly smaller subset of test cases can be identified, leading to substantial savings of both time and cost.

Examination of the regression testing literature indicated that several of the proposed algorithms are likely to be computationally very expensive to run, and yield test suites that were almost as large as the full test suite. In the worst case, a great deal of the limited regression testing resources would have been spent selecting the subset, only to learn that the algorithm requires the entire test suite, or nearly the entire test suite, to be rerun. In that case, the test organization is in worse shape than they were initially, since they did not have enough resources to rerun the entire test suite, spent a large portion of their budget learning that they really *should* rerun the entire test suite, and therefore are now able to rerun an even smaller fraction than they would have been able to run in the first place.

This observation led Rosenblum and Weyuker to develop predictors that are computationally very efficient, and indicate to the user whether or not a given selective regression testing strategy is likely to be cost-efficient to use for a given software system and test suite. [16,17] Additional studies of the use of this predictor appear in [11].

Thus, by using a predictor, a regression testing organization can decide whether or not it is likely to be beneficial to spend part of its limited resources to select an efficient subset of a large test suite, rather than assuming that this is a wise use of resources.

5 Scalability

When we build a new software system, we often have to make a prediction of what the customer base is likely to be. Once the system is complete, we then do performance testing and assess how the system behaves under workloads expected in both the near term and once the product has become more established. But what if the product becomes wildly successful and suddenly has to be able to handle a workload that is several times larger or even orders or magnitude larger than the current workload? Of course, this is every businesses' dream, but if the system cannot handle this increased workload while providing acceptable levels of performance to its customers, it can easily become a nightmare. In addition, many projects use costly custom-designed hardware that must be budgeted for and ordered well in advance of when it will be deployed. On the one hand, a project does not want to order expense equipment on the off-chance that things will go well and it will be needed, but they also do not want to be caught without sufficient capacity if it is needed.

For this reason, Weyuker and Avritzer considered the question of how a project could predict whether or not its software would scale under significantly increased workloads. [18] They defined a new metric, the Performance Non-Scalability Likelihood (PNL), which was designed to be used in conjunction with a workload-based performance testing technique [5].

This metric makes an assessment of the expected loss in performance under varying workloads by distinguishing between system states that provide acceptable performance behavior, and those that don't. The metric incorporates two factors:

- The probability that the system is in a given state (which is determined by the workload characterization used for performance testing).
- The degree to which the system fails to meet acceptable performance while in that state.

A case study was presented that demonstrated how to apply the PNL metric to a large, complex, industrial software system, including the steps needed to model the system, the type of data collected, and the actual computation of the metric. A description of the implications of the computation and experiences of the project of using the information derived from the application of the metric to plan for additional capacity was also presented. This prediction allowed the project to seamlessly rebalance their workloads, identify a potential bottleneck, and deploy additional capacity so that users never encountered unacceptable performance, even though the workload increased substantially over the period of study.

6 Risk

Software risk can be defined to be the expected loss due to failures caused by faults remaining in the software. This is a very important characteristic of software systems because, if we could predict the risk associated with releasing a system in its current form, we could determine whether or not it is safe to do so. In Reference [20], Weyuker discussed the limitations of using some definitions of risk that were proposed in earlier research. The problems centered around either a requirement for information that could not generally be determined, or a requirement that more data be collected than is practically feasible for a large industrial system with a large, complex input domain. Another class of limitation is the fact that many of the risk definitions do not use relevant information that *is* available. For this reason, Weyuker introduced a predictive metric that could be used to determine the likely risk associated with a software system in an industrial environment, thereby helping the development team determine whether or not it is safe to release their software.

The metric incorporated information about the degree to which the software had been tested, as well as the how the software behaved (i.e. whether or not it fails). This metric assures that testers are not rewarded for doing a poor job of testing, by treating unexecuted test cases as if they had been run and failed. For

this reason, as the amount of testing increases, the assessed risk typically goes down, because in a typical system, one expects to see a very small percentage of test cases fail. Similarly, if a fault is encountered and a failure occurs, the assessed risk will also decrease once the fault has been corrected. Another feature of this predictive metric is that it includes a mechanism to incorporate the severity of a failure. If the failure is a minor or cosmetic one, it will clearly impact the perceived reliability of the system.

7 Conclusions

We have discussed the importance and value of being able to predict characteristics of software systems. In this way, development and maintenance organizations can determine cost-effective ways of allocating scarce resources such as testing personnel and equipment or laboratory space, and determining whether it is safe to proceed with a proposed architecture or release a system. Each of these predictors have been assessed or developed with case studies performed on large industrial software systems which show both the usefulness of these predictors as well as the feasibility of using these predictors for large systems. Pointers to the literature describing these case studies are provided. Our conclusion is that being able to predict these and related characteristics of software system represent a very powerful mechanism for creating highly dependable systems.

References

1. G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrup, and A. Zaremski. Recommended best industrial practice for software architecture evaluation. Technical Report CMU/SEI 96-TR-025, available at *http://www.sei.cmu.edu/products/publications*, Jan, 1997.
2. E.N. Adams. Optimizing Preventive Service of Software Products. *IBM J. Res. Develop.*, Vol28, No1, Jan 1984, pp.2-14.
3. A. Avritzer and E.J. Weyuker. Investigating Metrics for Architectural Assessment, *Proc. IEEE/Fifth International Symposium on Software Metrics (Metrics98)*, Bethesda, Md., Nov. 1998, pp.4-10.
4. A. Avritzer and E.J. Weyuker. Metrics to Assess the Likelihood of Project Success Based on Architecture Reviews. *Empirical Software Eng. Journal*, Vol. 4, No. 3, Sept. 1999, pp.197-213.
5. A. Avritzer, J. Kondek, D. Liu, and E.J. Weyuker. Software Performance Testing Based on Workload Characterization. *Proc. ACM/Third International Workshop on Software and Performance (WOSP2002)*, Rome, Italy, July 2002.
6. V.R. Basili and B.T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, Vol27, No1, Jan 1984, pp.42-52.
7. L. Bass, P. Clements, and R. Kazman. Software architecture in practice. Addison Wesley, 1998.
8. N.E. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Trans. on Software Engineering*, Vol26, No8, Aug 2000, pp.797-814.

9. T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting Fault Incidence Using Software Change History. *IEEE Trans. on Software Engineering*, Vol 26, No. 7, July 2000, pp. 653-661.
10. L. Hatton. Reexamining the Fault Density - Component Size Connection. *IEEE Software*, March/April 1997, pp.89-97.
11. M.J. Harrold, D. Rosenblum, G. Rothermel, and E.J. Weyuker. Empirical Studies of a Prediction Model for Regression Test Selection. *IEEE Trans. on Software Engineering*, Vol 27, No 3, March 2001, pp.248-263.
12. J.P. Holtman. Best current practices: software architecture validation. AT&T, March, 1991.
13. K-H. Moller and D.J. Paulish. An Empirical Investigation of Software Fault Distribution. *Proc. IEEE First Internation Software Metrics Symposium*, Baltimore, Md., May 21-22, 1993, pp.82-90.
14. J.C. Munson and T.M. Khoshgoftaar. The Detection of Fault-Prone Programs. *IEEE Trans. on Software Engineering*, Vol18, No5, May 1992, pp.423-433.
15. T. Ostrand and E.J. Weyuker. The Distribution of Faults in a Large Industrial Software System. *Proc. ACM/International Symposium on Software Testing and Analysis (ISSTA2002)*, Rome, Italy, July 2002, pp.55-64.
16. D.S. Rosenblum and E.J. Weyuker. Predicting the Cost-Effectiveness of Regression Testing Strategies, *Proc. ACM Foundations of Software Engineering Conf (FSE4)*, Oct 1996, pp.118-126.
17. D.S. Rosenblum and E.J. Weyuker. Using Coverage Information to Predict the Cost-Effectiveness of Regression Testing Strategies, *IEEE Trans. on Software Engineering*, March, 1997, pp. 146-156.
18. E.J. Weyuker and A. Avritzer. A Metric to Predict Software Scalability. *Proc. IEEE/Eighth International Symposium on Software Metrics (Metrics2002)*, Ottawa, Canada, June 2002, pp.152-158.
19. E.J. Weyuker. Predicting Project Risk from Architecture Reviews. *Proc. IEEE/Sixth International Symposium on Software Metrics (Metrics99)*, Boca Raton, Fla, Nov. 1999, pp.82-90.
20. E.J. Weyuker. Difficulties Measuring Software Risk in an Industrial Environment. *Proceedings IEEE/International Conference on Dependable Systems and Networks (DSN2001)*, Goteberg, Sweden, July 2001, pp. 15-24.