Using Developer Information as a Factor for Fault Prediction

Elaine J. Weyuker, Thomas J. Ostrand, Robert M. Bell
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
(weyuker,ostrand,rbell)@research.att.com

Abstract

We have been investigating different prediction models to identify which files of a large multi-release industrial software system are most likely to contain the largest numbers of faults in the next release. To make predictions we considered a number of different file characteristics and change information about the files, and have built fully-automatable models that do not require that the user have any statistical expertise. We now consider the effect of adding developer information as a prediction factor and assess the extent to which this affects the quality of the predictions.

1 Background

We have been using negative binomial prediction models [20, 21, 4] to identify which executable files of a large multi-release industrial software system are most likely to contain the largest numbers of faults in the next release. By identifying files that are likely to be particularly problematic in the next release, we can help software testers prioritize their testing efforts. This should in turn allow them to find faults more efficiently, and perhaps give them time to do more testing than would otherwise be possible. In addition, if we can alert developers to likely problematic files, this might help them determine which files were the best candidates for rearchitecting.

The three case studies, cited above, showed that our models were able to make quite accurate predictions for systems having different characteristics. The models identified 20% of the files predicted to contain the largest numbers of faults. For two of the systems, these files contained, on average, 83% of the faults. For the third system, the percentage of faults included in the identified files was approximately 75%. However, considering that this system did not have regularly scheduled releases, a basic assumption for the technology, the predictions for it were surprisingly accurate.

In this short paper we investigate whether the addition of information about the number of software developers who have changed a file can further improve the accuracy of fault prediction. We know of only one previous study, by Mockus and Weiss [15], which has investigated similar variables. That study demonstrated a relationship between faults and the experience of developers with the system, but no relationship with the number of developers. Our analysis shows that having more developers touch a file was associated with more faults in subsequent releases, even holding fixed the number of changes.

2 Our Previous Fault Prediction Findings

We based our prediction methodology on a preliminary study [19] that looked at which characteristics of a large software system were most closely associated with files that turned out to be particularly fault prone. All of the factors considered were objectively assessable characteristics of the file including the number of thousands of lines of code (KLOCs) in the file, the file's age, the system's maturity in terms of the overall number of releases to date, and the language in which the file was written. We also included the file's recent history of faults and changes.

In each study, we used negative binomial regression [14] to model the number of faults in a file during a particular release. The combination of a file and release serves as the unit of analysis for our model, using a linear combination of the explanatory variables. This models the logarithm of the expected number of faults. We used Version 9.1 of SAS [24]. More information on how this methodology was applied to a different system during an earlier case study can be found in [4].

We typically used our models to identify the 20% of the files predicted to contain the largest numbers of faults. Table 1 shows the results for these three studies. There has been some question as to whether making and using

Third International Workshop on Predictor Models in Software Engineering (PROMISE'07)
0-7695-2954-2/07 $20.00 © 2007 IEEE

System	Number of Releases	Years	KLOC	Pctg Faults Identified
Inventory	17	4	538	83%
Provisioning	9	2	438	83%
Voice Response	-	2+	329	75%

Table 1. System Information for Case Study Subjects and Percentage of Faults in Top 20% of Files

such predictions are cost-effective activities. Arisholm and Briand [2] argued that if the percentage of faults included in the selected files or other entities is less than or roughly equal to the percentage of lines of code included in the files identified as being fault-prone, then fault prediction is not worth the effort.

Although we believe that it is questionable that the cost of testing is directly proportional to the size of a file, especially if using black-box testing techniques, we note that for all of the case studies we performed, the percentage of lines of code included in the files were always substantially smaller than the percentage of faults identified (see, e.g., [21]. In addition, if black-box testing techniques are being used, it seems reasonable to assume that assessing the cost effectiveness based on the percent of files versus the percent of faults is, in fact, the most accurate way of making that assessment.

We recently began studying a fourth system for a new case study [25]. This system is particularly interesting because it was written by a different international company than the earlier three subject systems. This allowed us to see whether the prediction results for this system are similar to the ones observed for the three earlier systems even though there is a different corporate culture and presumably different design, development and testing paradigms. In addition, this is a very mature system with 35 releases and roughly nine years of field exposure.

For this project, a maintenance support system, we observed results that were very similar to our earlier results. In particular, the 20% of the files identified by the model as being ones likely to account for the largest numbers of faults contained, on average, 81% of the faults identified during system testing. We note that for all of the systems studied to date, the vast majority of all faults were observed during some stage of testing prior to the release of the system to the field.

The maintenance support system used the same commercially available integrated version control/change management system as was used by the systems we studied previously. This requires documentation of any change made to the system in a *modification request* (MR).

By extracting information from MRs, we based our predictions on objectively assessable factors including file size; whether the file was new to the system in the current release; the number of faults the file had in earlier releases; the number of changes made to the file in previous releases; and the programming language in which the file was written.

The models developed for the systems for which we made predictions in earlier case studies were customized using observations of these factors during the first few releases of the system. In this study, however, our goal was to develop a fully-automatable model based on what we learned during earlier studies. Therefore, we adapted the models used in previous studies by extracting data from the maintenance support system in an automatic way, as would be done by a tool being built for use without human intervention.

We assessed four distinct models for the maintenance support system having different degrees of automatability. They ranged from fully specified formulas based entirely on what we learned from our three earlier case studies, to an entirely customized model. Interestingly we found that an intermediate model which is pre-specified with coefficients estimated from data for the current system using negative binomial regression yielded the best overall results, even better than the fully customized model [25]. We were able to estimate these coefficients using our experience gathered during our earlier studies [21, 4]. From this experience we were able to determine which characteristics were likely to be of greatest importance.

More specifically, in this model, the coefficients to predict Release N depend on a model fit to Releases 2 to (N-1). We decided to exclude Release 1 from our modeling because it differed substantially from later releases. In particular it lasted twice as long as the norm and although there were many file changes, there were only 22 changes that we identified as being faults. In addition, very little system testing occurred during Release 1, which might explain the small numbers of faults detected. These differences might cause its use to be unrepresentative of later releases and therefore it was excluded.

3 The Influence of Developers

Perhaps the question most commonly asked when we have presented talks about this work has been whether we considered information about the developers among the predictive factors. The reasoning is that it is easiest to make reliable changes to code if the developer is familiar with the complete history of a file's functionality and code changes. Therefore, we might expect more faults to result

from changes by developers who are new to working on a file or who share responsibility with other developers.

3.1 Developer Metrics

Because the control/change management system identifies the developer who input each change, we are able to track both the number of developers who made recent changes and whether those developers had made changes previously. Unfortunately, we could not reliably identify the developer(s) responsible for the initial file creation. In particular, we derived the following three metrics:

- The number of developers who modified the file during the prior release.
- The number of *new* developers who modified the file during the prior release.
- The *cumulative* number of *distinct* developers who modified the file during all releases through the prior release.

Note that the first two metrics are only positive for files with at least one change in the prior release, and the second may be zero even for those file-release combinations. Because the negative binomial regression model controls for changes in the prior release (specifically, the square root of the number of changes) [25], these two metrics are designed to improve prediction for changed files.

Figure 1 shows the distribution of the cumulative number of developers after 20 releases (about five years) for 526 files that reached that point. The mean number of developers was 3.54, but values varied greatly and the distribution was very skewed. 19% of files were never changed, so their cumulative developer counts were zero. Of changed files, about one half had one or two developers. However, 10% of changed files have ten or more developers, with the maximum reaching 31.

Figure 2 shows that the mean of the cumulative number of developers grew rather steadily with the number of releases. The reason is that for files with changes at a release, generally about 60% were touched by at least one new developer (see Figure 3). Indeed, the declining growth rate over time in the mean cumulative number of developers was due entirely to an even sharper decline in the proportion of files with any changes as files matured.

About 30 percent of files changed at any given release were touched by more than one developer (without regard to whether the developers were new to the file). Figure 4 shows this proportion as a function of a file's age.

3.2 Results

We evaluated the predictive power of the developer metrics using data from the maintenance support system. Start-

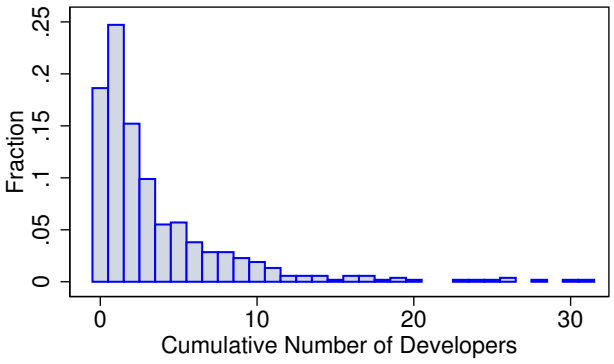

Figure 1. Histogram of Cumulative Number of Developers after 20 Releases

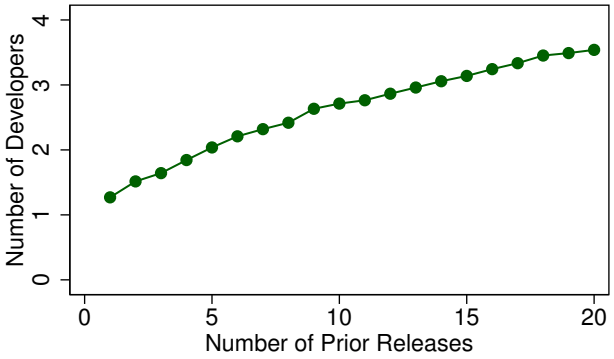

Figure 2. Mean Cumulative Number of Developers, by File Age

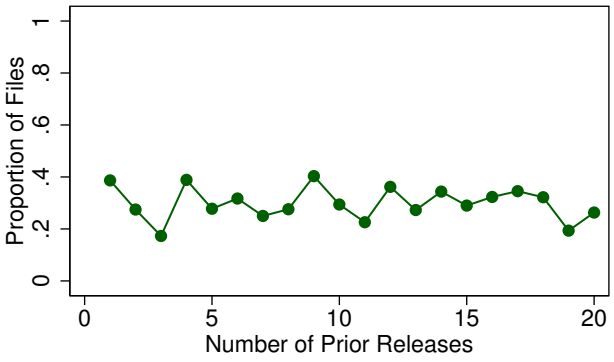

Figure 3. Proportion of Changed Files with Multiple Developers, by File Age

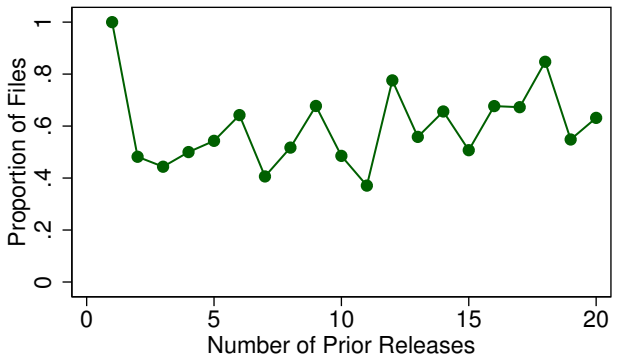

Figure 4. Proportion of Changed Files with at Least One New Developer, by File Age

ing from the best negative binomial regression model for that system, we added the developer metrics singly and in various combinations. The other explanatory variables in the model were: log(KLOC); number of previous releases in the system, categorized as 0, 1, 2-4, or 5 plus; square roots of the numbers of changes in the previous release, changes two releases ago, and faults in the previous release; dummy variables for selected programming languages; and dummy variables for each previous release [25]. To account for the skewness of the developer metrics, each was transformed by taking the square root. Regressions were fit with one observation per file-release combination for Releases 2 through N, for N = 6, 10, 18, 26, and 34.

In general, relationships with the developer variables were strongest for data including Release 18 or later. For each of the three samples extending through Release 18 or later, all three developer metrics had positive, statistically significant coefficients when included as the only developer metric. In general, the strongest relationship was with the cumulative number of developers, followed by the number for the prior release. After including those two variables, the number of new developers was never significant.

Table 2 shows the joint impact of the two best developer metrics on our predictive accuracy measure. As in [25], we divide releases into groups of five and report the percentage of faults contained in the top 20% of files. The next to last row shows mean values for the model both with and without developer metrics, over all releases from Release 3 through Release 35. Because the developer metrics were most strongly related to faults at later releases, we also include the mean values for both versions of the model over Releases 6 through 35. We see that the inclusion of the number of developers modifying files does improve the accuracy of the prediction, but only slightly.

4 Related Work by Other Groups

In recent years there has been a growing interest in fault localization and predictive models. Many of the papers are similar in nature to our preliminary study [19] and identify properties of software entities that seem to be closely associated with fault-prone code. These include [1, 3, 7, 8, 10, 11, 16, 17, 19, 22].

Several research groups have also developed models that do some sort of prediction of such things as the locations, the probability, or the quantity of faults or some related characteristic that will occur in a software system at some future time [2, 6, 9, 12, 13, 15, 18, 23].

None of the above-cited papers, however, describes results that have the same goal as ours: to identify files likely to contain the largest numbers of faults in the next release of a system [20, 21, 4]. Some of the closely related research is described in this section.

The paper by Mockus and Weiss [15] is the most relevant to this paper, because they studied the relationship between faults and four developer variables. They constructed a model to predict the probability that changes made to a system in response to an MR would cause a failure in the resulting system. They studied a large telephone switching system with ten years worth of data and used 15,000 MRs to design their model. They used logistic regression to construct their model, and instead of predicting which files are most likely to contain faults as we did, they predicted whether or not a given change will cause a failure. Besides developer variables, the characteristics they used included the size of the change; the number of distinct files, modules, and subsystems that were touched by the change; and the time duration needed to complete the change. They did not assess how accurate their model's predictions were. The four developer variables included three measures of developer experience: an overall measure, a measure of recent experience, and a measure of experience for the specific subsystem. The fourth developer variable was a count of the number of developers involved in completing an MR.

Only one of Mockus and Weiss's four developer variables–the overall experience measure–was statistically significant. More experience was associated with fewer problems in the future. However, their developer count, the measure most comparable to any of ours, was not statistically significant. Even though there is some overlap between their developer metrics and ours, there are enough differences in both the metrics and the overall study designs to potentially account for any differences in findings.

In [2], the authors created a prediction model to be used to aid testers. They designed their model for a particular moderate-size Java system that had about 110 KLOC. Although they had 17 releases available, they used data from only three of these releases, and using stepwise logistic re-

Release Number	W/O Developers	With Developers
3-5	55	49
6-10	78	79
11-15	71	73
16-20	84	86
21-25	89	88
26-30	90	91
31-35	92	92
Mean for all Releases	81.3	81.7
Mean for Rel 6-35	83.9	84.9

Table 2. Percentage of Faults in Top 20% of Files, by Model for the Maintenance Support System

gression, they made predictions for just one future release. For each of the four systems we have studied to date, we made predictions for each of the successive releases, noting how the prediction accuracy changed as the system matured. They did not use developer information in making their predictions.

Graves et al. [9] also performed research that was closely related to ours. Their study used a large telecommunications system that they followed for approximately two years. They first identified module characteristics closely associated with faults, similar to our preliminary work described in [19]. Using these findings, they built several prediction models to determine the number of faults expected in the next version of a module. Their models did not include developer information. Instead they relied entirely on the fault history of the system, working at the module level, which is far coarser than the file level that we worked at. Their system had 80 modules, containing approximately 2500 files. More details about the differences between our work and this paper are described in [4].

Denaro and Pezze [6] used logistic regression to construct their prediction models. Using sets of static software metrics, they constructed potential models. They selected the most accurate models based on how closely each model came to correctly determining all of the system's most fault-prone modules. Their models were not as accurate as ours, and they did not use developer information to define the models. Their work was based on data collected from Apache version 1.3 and evaluated on Apache version 2.0. They reported that their best models required 50% of the identified modules to be included in order to cover 80% of the actual faults. As mentioned above, for three of the four systems we studied, the model we used identified 20% of the files containing, on average, at least 83% of the actual faults. Even for the voice response system where we had to use synthetic releases, the 20% of the files identified by our model contained 75% of the faults.

In [13], Khoshgoftaar et al. described a way of using binary decision trees to classify software modules as being either fault-prone or not fault-prone. A module was considered fault-prone if it contained at least 5 faults. They used a set of 24 static software metrics as well as four execution time metrics, but none that related to developers. They considered four releases of an industrial telecommunications system, using the first release to build the tree and the remaining three releases for evaluation. They measured success based on misclassification rates. More details about this work and its relevance to ours is included in [4].

Succi et al. [23] used software size as well as object-oriented metrics proposed in [5] to identify fault-prone classes. They did not include developer information. They considered a number of different models to make predictions for two small C++ projects (23 and 25 KLOCS). For one of the projects, 80% of the total faults were contained in less than 30% of the classes, and for the other project 80% of the faults were reportedly in 2% of the classes. For each of the different models they tried, classes they predicted to contain 80% of the faults ranged between 43% and 48% of all classes. Although the goal of that paper is closely related to our work, there are important differences. The subjects of their study were smaller than 10% of the size of the systems that were subjects of our studies. While we looked at many successive releases for each system, they appear to have used a single interval, and their predictions seem to be significantly less accurate than ours.

In summary, the only other related paper that we are aware of that used developer information is [15], and the goal of their predictions was different from ours.

5 Conclusions

We modified our earlier prediction model to include number of developers that made changes to each file both in the prior release and cumulatively. We applied this new model to a very mature maintenance support system that had been in the field for over nine years and 35 releases. We observed that without the developer information, a completely automatable model was able to correctly identify

20% of the files containing, on average, 83.9% of the faults in the system starting with Release 6. When the developer information was included, we found that this rose to 84.9% of the faults.

Since our goal is to implement a fully-automatic tool that can do both the data extraction portions of this work and the prediction portions, and since the developer information can be automatically extracted from the MR database, this is very encouraging.

We intend to continue our implementation efforts and case studies to validate the usefulness of our models including the newly identified factor, developers making changes.

References

[1] E.N. Adams. Optimizing Preventive Service of Software Products. *IBM J. Res. Develop.*, Vol 28, No 1, Jan 1984, pp. 2-14.

[2] E. Arisholm and L.C. Briand. Predicting Fault-prone Components in a Java Legacy System. *Proc. ACM/IEEE ISESE*, Rio de Janeiro, 2006.

[3] V.R. Basili and B.T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, Vol 27, No 1, Jan 1984, pp. 42-52.

[4] R.M. Bell, T.J. Ostrand, and E.J. Weyuker. Looking for Bugs in All the Right Places. *Proc. ACM/International Symposium on Software Testing and Analysis (ISSTA2006)*, Portland, Maine, July 2006, pp. 61-71.

[5] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. on Software Engineering*, vol 20 no 6, June 1994, pp.476-493.

[6] G. Denaro and M. Pezze. An Empirical Evaluation of Fault-Proneness Models. *Proc. International Conf on Software Engineering (ICSE2002)*, Miami, USA, May 2002.

[7] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, A. Mockus. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Trans. on Software Engineering*, Vol 27, No. 1, Jan 2001, pp. 1-12.

[8] N.E. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Trans. on Software Engineering*, Vol 26, No 8, 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. Guo, Y. Ma, B. Cukic, H. Singh. Robust Prediction of Fault-Proneness by Random Forests. *Proc. ISSRE 2004*, Saint-Malo, France, Nov. 2004.

[11] L. Hatton. Reexamining the Fault Density - Component Size Connection. *IEEE Software*, March/April 1997, pp. 89-97.

[12] T.M. Khoshgoftaar, E.B. Allen, K.S. Kalaichelvan, N. Goel. Early Quality Prediction: A Case Study in Telecommunications. *IEEE Software*, Jan 1996, pp. 65-71.

[13] T.M. Khoshgoftaar, E.B. Allen, J. Deng. Using Regression Trees to Classify Fault-Prone Software Modules. *IEEE Trans. on Reliability*, Vol 51, No. 4, Dec 2002, pp. 455-462.

[14] P. McCullagh and J.A. Nelder. *Generalized Linear Models*, Second Edition, Chapman and Hall, London, 1989.

[15] A. Mockus and D.M. Weiss. Predicting Risk of Software Changes. *Bell Labs Technical Journal*, April-June 2000, pp. 169-180.

[16] K-H. Moller and D.J. Paulish. An Empirical Investigation of Software Fault Distribution. *Proc. IEEE First International Software Metrics Symposium*, Baltimore, Md., May 21-22, 1993, pp. 82-90.

[17] J.C. Munson and T.M. Khoshgoftaar. The Detection of Fault-Prone Programs. *IEEE Trans. on Software Engineering*, Vol 18, No 5, May 1992, pp. 423-433.

[18] N. Ohlsson and H. Alberg. Predicting Fault-Prone Software Modules in Telephone Switches. *IEEE Trans. on Software Engineering*, Vol 22, No 12, December 1996, pp. 886-894.

[19] 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.

[20] T.J. Ostrand, E.J. Weyuker, and R.M. Bell. Where the Bugs Are. *Proc. ACM/International Symposium on Software Testing and Analysis (ISSTA2004)*, Boston, MA, July 2004.

[21] T.J. Ostrand, E.J. Weyuker, and R.M. Bell. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Trans. on Software Engineering*, Vol 31, No 4, April 2005.

0-7695-2954-2/07 $20.00 © 2007 IEEE

[22] M. Pighin and A. Marzona. An Empirical Analysis of Fault Persistence Through Software Releases. *Proc. IEEE/ACM ISESE 2003*, pp. 206-212.

[23] G. Succi, W. Pedrycz, M. Stefanovic, and J. Miller. Practical Assessment of the Models for Identification of Defect-prone Classes in Object-oriented Commercial Systems Using Design Metrics. *Journal of Systems and Software*, Vol 65, No 1, Jan 2003, pp. 1 - 12.

[24] SAS Institute Inc. *SAS/STAT 9.1 User's Guide*, SAS Institute, Cary, NC, 2004.

[25] E.J. Weyuker, T.J. Ostrand, and R.M. Bell. Adapting a Fault Prediction Model to Allow Widespread Usage *2nd International Promise Workshop*, Philadelphia, PA, September 2006.

Third International Workshop on Predictor Models in Software Engineering (PROMISE'07)
0-7695-2954-2/07 $20.00 © 2007 IEEE