Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models

Article *in* Empirical Software Engineering · October 2008

DOI: 10.1007/s10664-008-9082-8 · Source: DBLP

CITATIONS
153

READS
765

3 authors, including:

Elaine J. Weyuker
University of Central Florida
194 PUBLICATIONS **10,713** CITATIONS

SEE PROFILE

Thomas J. Ostrand
Mälardalen University
70 PUBLICATIONS **4,654** CITATIONS

SEE PROFILE

Empir Software Eng manuscript No.
(will be inserted by the editor)

Do Too Many Cooks Spoil the Broth? Using the Number of Developers to Enhance Defect Prediction Models

Elaine J Weyuker · Thomas J Ostrand · Robert M Bell

the date of receipt and acceptance should be inserted later

Abstract Fault prediction by negative binomial regression models is shown to be effective for four large production software systems from industry. A model developed originally with data from systems with regularly scheduled releases was successfully adapted to a system without releases to identify 20% of that system's files that contained 75% of the faults. A model with a pre-specified set of variables derived from earlier research was applied to three additional systems, and proved capable of identifying averages of 81, 94 and 76% of the faults in those systems. A primary focus of this paper is to investigate the impact on predictive accuracy of using data about the number of developers who access individual code units. For each system, including the cumulative number of developers who had previously modified a file yielded no more than a modest improvement in predictive accuracy. We conclude that while many factors can "spoil the broth" (lead to the release of software with too many defects), the number of developers is not a major influence.

Keywords software faults · negative binomial model · empirical study · developer counts

1 Background

We have been using negative binomial regression models to predict which executable files of a large multi-release industrial software system are likely to contain the most 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, allowing them to find faults more efficiently and, perhaps, giving them time to do more testing than would otherwise be possible. In addition, alerting developers about potentially fault-prone files might help them determine files that are good candidates for re-architecture.

Common folklore in software engineering asserts that "too many cooks spoil the broth" when it comes to code development and maintenance. That is, code units will be less fault-prone if they are written and maintained by only a few, or even just one,

AT&T Labs-Research

developer. Another common belief is that a developer who works for the first time on a file that has previously been written or maintained by others is more likely to introduce faults into the software than programmers who have prior experience with the code. Intuitively, a developer who is familiar with the history of a file's functionality and code changes is much more likely to make reliable changes than one who is encountering the code for the first time.

To assess this belief, we expanded previous fault prediction models to include information about the number of developers who worked on specific files in previous releases, and applied the modified models to three large stand-alone subsystems of a Maintenance Support System. For each subsystem, a higher total number of distinct developers changing an individual file over the course of its past history was associated with more faults for that file in subsequent releases, even holding fixed the number of changes. However, inclusion of developer variables in the prediction model yielded at most a modest improvement in predictive accuracy. We know of only one previous study, by Mockus and Weiss [16], that has investigated the use of 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.

In an earlier paper [24], we presented an empirical study showing that a preliminary model made quite accurate predictions for two large industrial software systems with many distinct characteristics. We refer to these as the Inventory system and the Provisioning system. The predictions of the models for these systems are based on readily available characteristics such as a file's language, size, age, and history of faults and changes. We evaluated the models by counting the faults actually discovered and corrected in the 20% of the files predicted to contain the largest numbers of faults in the next release. Although these systems had substantially different characteristics, were written by different development teams, and used different programming languages, the identified 20% of files contained, on average, 83% of the faults for each of the systems.

The current paper provides several new contributions. In addition to the investigation of whether information about the number of software developers who have changed a file can further improve the accuracy of fault predictions, we present the results of empirical studies of four different large production software systems, two of which have not been included in any earlier study. The first of these four, introduced in [4], will be referred to as the Voice Response system. Because this system did not have regularly scheduled releases, we could not apply our original prediction models in a straightforward way, as the models relied on the existence of regular releases.

The other three subject systems are interesting because they were all written by a different company with its own corporate culture that might affect the usefulness or effectiveness of our prediction algorithms. Each of these three is actually a stand-alone subsystem of a very large multi-function system; each subsystem is comparable in size to the systems we studied previously. In the sequel, we refer to the overall system as the Maintenance Support system, and to the three subsystems as S, W, and E. The overall system is very mature, with more than 35 releases over a period of nine years. We compare our prediction results for the three subsystems to the results of our earlier studies for systems that ranged in age from two years to four years of field exposure.

To our knowledge, this is the most comprehensive set of empirical studies performed to assess the effectiveness of proposed fault prediction models. In addition, of the few other research groups that have investigated fault prediction models, most have worked at a coarser level of granularity than we have. While we have used our models to identify individual files that are likely to be fault-prone, other groups have looked at

the larger module level, where a module is a collection of files. As these researchers have acknowledged, it is considerably easier to identify fault-prone modules than individual files.

The paper is organized as follows. Section 2 describes the change reporting database from which system and fault information is extracted, techniques for determining which change reports represent software faults, and the structure of the prediction models. Section 3 describes each of the large industrial systems for which we present results in this paper. In Section 4, we summarize our past empirical studies and prediction results. Section 5 explores incorporating information about the number of distinct developers who have changed a file into the prediction models. Subsection 5.1 considers models using several different developer counts, while Subsection 5.2 presents results for the three Maintenance Support systems. We discuss related work in Section 6 and present our conclusions in Section 7.

2 Prediction Methodology

Our prediction methodology is designed for large industrial systems with a succession of releases over years of development and maintenance. These systems typically include hundreds of thousands of lines of code (KLOC) in several hundreds, or even thousands of files. The prediction models utilize readily available information about the files at any point in time.

2.1 The MR System

Each of the systems we report on in the current and earlier papers used a variant of a commercially available, integrated version control/change management system. All changes to the system's code are managed through a *modification request* (MR), which is initially created by the person requesting the change, and subsequently updated by the person who actually makes the change. An MR describing a problem with the system's behavior is typically created by a tester and eventually sent on to the developer responsible for the part of the system that is believed to contain the fault. The process of deciding which part of the system is actually at fault may not be straightforward. When the problem-causing fault or faults are eventually identified, the software is updated, and the MR is augmented with information describing the change. The following information is typically included in every MR:

- an English description of the reason for the proposed change, written by the person who creates the MR.
- an English description of the change, written by the person who modifies the software.
- a list of the files changed or added to the system by the MR.
- identification of the specific lines of code that were added, deleted or modified because of the MR.
- the stages of the development process during which the MR was created and the changes were made.
- dates of all the significant activities, including MR creation, software updates, and acceptance of the change via retesting.
- id's of the testers and developers who created the MR and updated the software

We have used the information in the MR database to identify relevant predictive factors such as the file size; whether this was the first release in which the file appeared; whether there were faults in earlier versions of this file; the number of faults in this file during earlier releases; how many changes were made to the file in previous releases; and the programming language in which the file was written.

2.2 Fault Identification

Despite all the detailed information that appears in an MR, the normal MR form does not include a simple checkbox that would indicate whether the MR was created to address a detected software system failure (i.e., a fault), or some other type of issue, such as functionality enhancement, performance improvement, a requirements change, an interface change, etc. The only way to positively determine whether an MR represents a fault is to read its natural language descriptions, and sometimes even these are unclear. For the projects we have studied, the very large number of MRs, many of which do not represent faults, makes it unrealistic to read all the descriptions. Further, use of the English descriptions is ruled out for our eventual goal, an automated fault prediction tool. As a result, in our first two studies we used heuristics, described in [24], to determine which MRs should be counted as faults.

When we began work with the Voice Response project (described in more detail in Section 3.2), during its 18th month of development, we asked the team to modify its MR form to include an explicit indication of whether the MR was being initiated due to evidence of a fault. This fault identification field became part of the MR form while we were collecting data for month 21. Users were able to choose among three alternatives for an MR: *bug*, *not a bug*, and *don't know*.

We used this field, together with the existing *MR-Category* field of the MR, to classify each MR as either a fault or non-fault. The MR-Category field can be assigned values including developer-found, system-test-found, customer-found, new-code, and new-feature. We restricted our attention to MRs categorized as being either system-test-found or customer-found. Our rationale was that the system tester's goal is to find pre-release faults and that any MRs initiated by customers were likely to be faults.

Starting with Release 21, we categorized as a fault any MR that was labeled either system-test-found or customer-found and whose Bug-id field was either identified as a *bug* or was left blank. If the Bug-id field was marked *not a bug*, the MR was not considered to be a fault regardless of the value of the MR-Category. For the few MRs for which the Bug-id field's value had been assigned *don't know*, we based our decision about fault status on a manual reading of the MR's detailed description. All of them turned out to be faults. For MRs created before the project added the bug-identification field, we categorized any system-test-found or customer-found MRs as faults. For the other three systems discussed in this paper, this field was not available, and we used the MR-Category alone to decide whether or not an MR represents a fault.

2.3 Prediction Models

In all of our studies, we use negative binomial regression (NBR) [14] to model the logarithm of the expected number of faults in a file during a particular release, with various combinations of attributes as independent variables. Table 1 lists the set of all

Table 1 Attributes

Attribute
1. KLOC
2. File age
3. Programming language
4. Faults in Release n-1
5. Faults in Release n-2
6. Changes in Release n-1
7. Changes in Release n-2
8. Exposure
9. Developers in Release n-1
10. New developers in Release n-1
11. Cumulative developers, Releases 1:n-1

attributes we considered; each model is based on a subset of these. *File age* means the total number of consecutive prior releases in which the given file appeared. A file that is new in the current release, or that did not exist in the immediately prior release even if it did exist in earlier releases, has an age of 0. *Changes in Release n-1* or *n-2* means the total number of times the file was modified for any reason. Its value is obtained by counting the number of times the file was checked in during the release in question. Each check-in has to be related to a previously created MR. *Faults in Release n-1* or *n-2* means the number of Changes whose MR was determined to be a fault. *Exposure* is defined in Section 3.2, and the three developer-related attributes are defined in Section 5.1.

The combination of a file and release served as the unit of analysis for our models, using a linear combination of the explanatory variables to model the logarithm of the expected number of faults. Specifically, let y_i equal the number of faults observed in file i and x_i be a vector of characteristics for that file. NBR specifies that y_i, given x_i, has a Poisson distribution with mean λ_i. To allow for additional dispersion compared with Poisson regression in the number of faults for each file, the conditional mean of y_i is given by $\lambda_i = \gamma_i e^{\beta' x_i}$, where γ_i is itself a random variable drawn from a gamma distribution with mean 1 and unknown variance $\sigma^2 \geq 0$. The variance σ^2 is known as the *dispersion parameter*. We used Version 9.1 of SAS [27] to estimate the model parameters.

In our previous work, we have developed models using various combinations of attributes 1-8. In the work described in this paper, we extend those models with variables 9-11, and evaluate the effectiveness and utility of the augmented models that use these developer-related attributes.

We have typically used our models to identify the 20% of the files predicted to contain the largest numbers of faults. There has been some question as to whether making and using such predictions are cost-effective activities. Arisholm and Briand [2] argue 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.

We believe that it is questionable that the cost of testing is directly proportional to the size of a file, especially for black-box testing techniques, and note that for each of the systems that we have investigated to date, the percentage of lines of code included in the identified files was always substantially smaller than the percentage of faults contained in them (see, e.g., [24]). In addition, if black-box testing techniques are

Table 2 System Information for Case Study Subjects and Percentage of Faults in Top 20% of Files

System	Number of releases	Years	LOC in last release	Mean faults per release	Percentage of faults identified
Inventory	17	4	538,000	342	83%
Provisioning	9	2	438,000	34	83%

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.

3 Systems Studied in Past and Current Research

3.1 Inventory System and Provisioning System

Our first prediction model was based on a preliminary study of faults in the Inventory system. We collected data from 17 successive releases that were developed over a period of four years. During that time, the system grew in size from an initial 584 files containing 146,000 lines of code to 1950 files containing 538,000 lines. The number of faults detected per release varied from 988 in the first release, to 127 during a release after 3 years, and back up to 253 in the last release we studied. We noted a strong Pareto effect for the faults reported for this system: in the first release, all faults were located in 40% of the files, and for every subsequent release, 100% of the faults were located in fewer than 20% of the system's files.

We continued our study with the Provisioning system, for which we collected data on nine releases over two years of development and maintenance. This system was nearly stable in size over the two years it was observed. Practically all changes made were maintenance updates, and relatively few defects were reported.

Three-quarters of the Inventory system was written in Java, with 3-8% written in each of shell, Perl, and C, and much smaller percentages in a variety of other languages. The Provisioning system's main language was SQL (about 25%), with about 10% in each of Java, shell, and C. Table 2 summarizes information about these two systems, and further details are reported in [24].

3.2 Voice Response System

The Voice Response system was similar in size and duration to the two previous subjects, but its development process was quite different. Over a period of 29 months, this project grew from 61 files and 15,700 LOC to over 1900 files and 329,000 LOC, the same order of magnitude as the two earlier systems. The main language used was Java, about 56% of the code, followed by shell (9%), xml (7%), sql (6%), and 34 other languages that make up the remaining 22%.

The distinguishing feature of this project is that it used a *continuous release* process, with changes being made and tested continuously. A new version of the software was made available to customers as soon as it had satisfactorily passed the testing process for each new modification. In contrast, the Inventory and Provisioning systems had

a *discrete release* development process, with new versions being released at roughly regular intervals (typically every 3 months).

It is important to understand that it is not just the lack of releases that distinguishes the development paradigm used by the Voice Response system from that of the Inventory system and the Provisioning system. In the discrete release process, there is generally a predesignated release date and earlier dates when developers have to have completed their code modifications and additions, leaving time for a substantial system test period during which the code is frozen. During that period, system testers rigorously test the code, and developers fix any problems identified by system testers. Of course, any new or modified code written during this fix period must also be retested. Under the continuous release paradigm, there is no designated system test period.

Because the model developed in [24] was based on the regular release schedules of those systems, we had to modify the way we assessed both the fault and change histories and to determine what it should mean when we speak about "faults for the next release", given that there were no explicit next releases. To apply our prediction methodology to the Voice Response system, we arbitrarily defined one-month intervals to be *synthetic releases* for the system. Given the project's history for months 1 to $(N-1)$, fault predictions were made for month N. However, neither the developers nor the testers were aware of the synthetic releases, and nothing they did was in any way affected by their definition. Consequently, we were not confident that the prediction models that were developed previously would be effective at making accurate predictions.

Because a project that follows a discrete release schedule introduces most of the new code at the start of each release, we wanted to account for the fact that new files might enter a synthetic release at any time and therefore be in that release for very different lengths of time. Files that entered the system during the last few days of a month had much less opportunity to cause a failure during that month than those that entered the system at the beginning of the month. Because of this, we treated files that entered during the second half of a month as part of the following month's release. For example, if a new file entered the system on any day from June 16-30, it was considered to be part of the July release. On the other hand, if it entered the system on June 1-15, it was considered to be part of the June release.

To inform the model about the fraction of the month that a given new file has been in the system, we introduced the notion of *Exposure*. The Exposure for a new file can vary from 0.5 to 1.5, depending on whether the file has been in the system for half of the month (0.5), half of the previous month plus all of the current month (1.5), or something in between. The exposure for all non-new files is always 1. A full discussion of these issues is included in [4] along with a complete discussion of the model variants considered.

3.3 Maintenance System

With the Maintenance Support system, we again have a project that uses a discrete release process with regular schedules for development, testing, modification, and release. The three subsystems of the Maintenance Support system all schedule releases at approximately 3-month intervals. The systems are each rather similar in size and composition to the three earlier systems. Each of the subsystems was large, containing, respectively 668, 1413, and 584 files and 442 KLOCs, 384 KLOCs, and 329 KLOCs

during their last release. Subsystems S and W existed for 35 releases and nine years when we began studying them, while Subsystem E's first release occurred when the other two were already ten releases old. We followed E for 28 releases representing about seven years in the field. The number of faults detected per release varied widely for each subsystem, from a minimum of zero to a maximum of 130. As with the earlier systems studied, almost all of these faults were found during testing done prior to field release for customer use.

These systems were written by a different company than the three earlier subject systems, although both companies use the same version management/change control system. Thus, we can compare prediction results across different corporate cultures and presumably different design, development and testing paradigms.

4 Fault Prediction Findings without Using Developer Variables

The negative binomial regression model proved very successful at prospectively identifying fault prone files for both the Inventory and Provisioning systems [24]. No developer variables were tested for either system. Across all four years of the Inventory system, the top 20% of the system's files identified by the model contained an average of 83% of the faults reported for the last 15 of the 17 quarterly releases. For the Provisioning system, practically all changes were maintenance updates, and relatively few defects were reported. Consequently, we combined groups of four successive releases into an *aggregated release* for purposes of modeling and assessment. Using the same predictor variables as for the Inventory system, the top 20% of files identified by the model again contained 83% of the faults in the aggregated release that was tested.

Figure 1 shows the prediction results for the Voice Response system. Models were estimated for each month based on data for all prior months, using the model that was closest to the ones used during the earlier studies. The predictions start with month 7, providing the model with an initial base training period that corresponds roughly to two regular releases of 3 months duration. Note that during several months, particularly late in the study period, relatively few faults were observed. These low numbers can lead to highly variable prediction results. For example, during month 27 only 2 faults were detected in the 1906 files, and the files containing them were not included in the top 20% predicted by the model. On the other hand, for months 28 and 29, the model's top 20% included all the files responsible for the few detected faults.

Over the 23 months for which predictions were made, the top 20% of predicted files contained an average of 75% of the detected faults. Although less than the 83% correctly identified for the Inventory and Provisioning systems, this is still an extremely positive finding given that the models were not designed for this type of development paradigm.

After building custom-designed models for the first two systems analyzed in [24], we recognized that it was essential to develop a way of automating the prediction process so that practitioners with no modeling experience could benefit from the technology. This requires developing a common prediction model that could be applied to any system, with possible tuning based on observations made during the first few releases of a new subject system. Our ultimate goal is a totally automated tool that will perform data extraction and fault prediction without intervention, and therefore not require the user to possess any expertise in data extraction, analysis, or statistics.

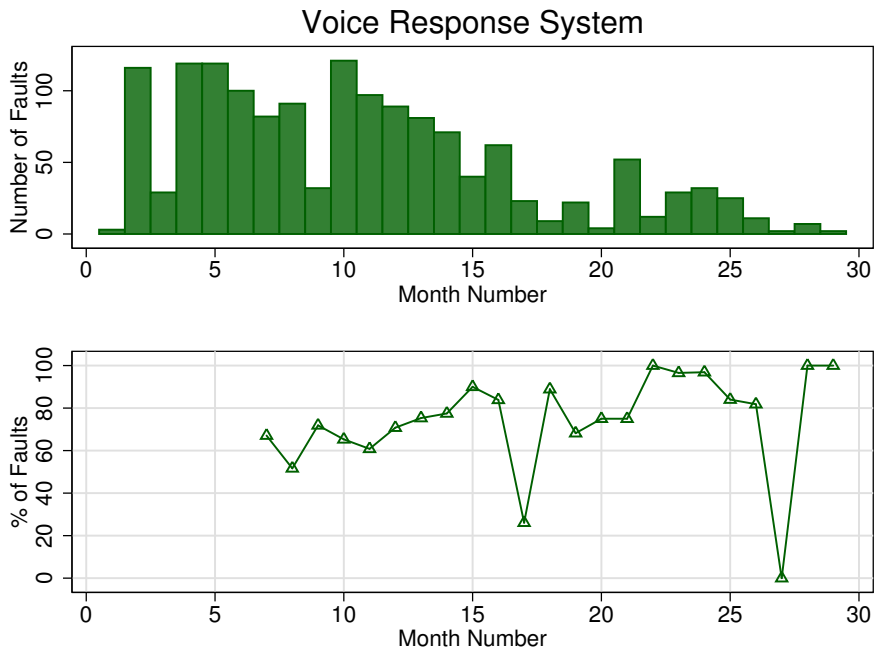

Fig. 1 Number of Detected Faults and Percentage of Faults in Top 20% of Files, by Release Number for the Voice Response System

With this goal in mind, we assessed, for Subsystem S, four distinct negative binomial regression models having different degrees of automatability [25]. Two of these were entirely turnkey models, for which both the predictors and coefficients were pre-specified, and based entirely on observations made during the earlier studies of other systems - i.e., independent of any data from Subsystem S, for which the predictions were being made. We also assessed a fully customized model developed from data of the first five releases of Subsystem S.

It turns out that the best prediction results occurred for a fourth intermediate model, with pre-specified predictor variables based on results from the previous systems, and with coefficients that were estimated using Subsystem S's data. That is, we were able to use experience from completely different systems studied previously to identify a successful set of predictor variables, some of which were transformed to improve the model fit. Consequently, application of the model requires no particular statistical expertise, only software to fit a negative binomial regression model and to compute predictions. This model's predictor variables were log(KLOC); number of previous versions of the file in the system, categorized as 0, 1, 2-4, or more than 4; 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.

When we applied this intermediate model to Releases 1 to 20 of Subsystems W and E, results were similar to those for Subsystem S. For each subsystem, the most

powerful single predictors were log(KLOC), the square root of the number of changes in the previous release, and the indicator for new files, always in that order (based on Wald chi square statistics). Fuller details of the intermediate model and its prediction results appear in [25].

5 The Influence of Developer Variables

To evaluate the impact of including developer variables in the prediction models, we collected and incorporated counts of individual developers who interacted with each file of the three Maintenance Support subsystems, over the course of all releases.

5.1 Developer Metrics

Because the MR database identifies the developer who inputs 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. We used 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 can only be non-zero for files with at least one change in the prior release. The second one will be zero even for prior-release changed files if all of the developers modifying the file had also modified the file during previous releases. Because the square root of the number of prior release changes is already part of the model, and will reflect whether or not a file has been changed, these two metrics are specifically designed to improve prediction accuracy for changed files.

Figure 2 shows the distribution of the cumulative number of developers who changed files during the first 20 releases of Subsystem S (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 40% had only one or two developers. However, 10% of the changed files in Subsystem S during these releases have ten or more developers, with the maximum reaching 31.

Figure 3 shows that the mean of the cumulative number of developers for Subsystem S grew rather steadily with the number of releases. The reason is that for files with changes at a release, between 40 and 80% were touched by at least one new developer (see Figure 4). 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.

Between 20 and 40 percent of files changed at any given release of Subsystem S were touched by more than one developer (without regard to whether the developers were new to the file). Figure 5 shows this proportion as a function of a file's age.

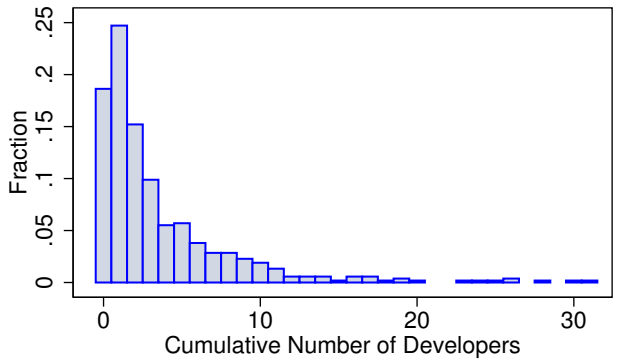

Fig. 2 Histogram of Cumulative Number of Developers after 20 Releases

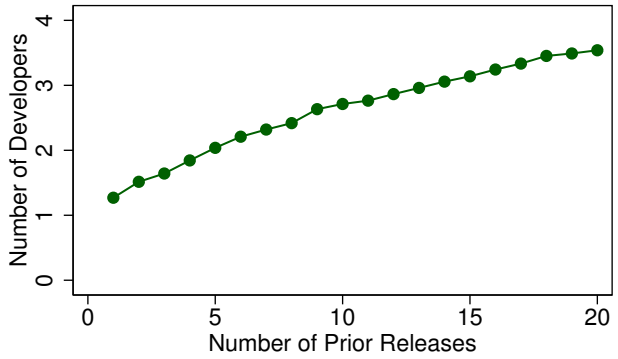

Fig. 3 Mean Cumulative Number of Developers, by File Age

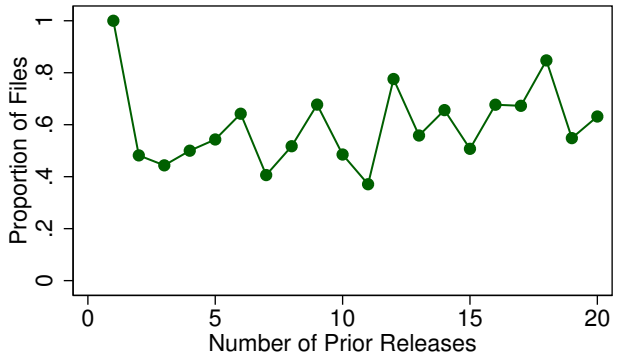

Fig. 4 Proportion of Changed Files with at Least One New Developer, by File Age

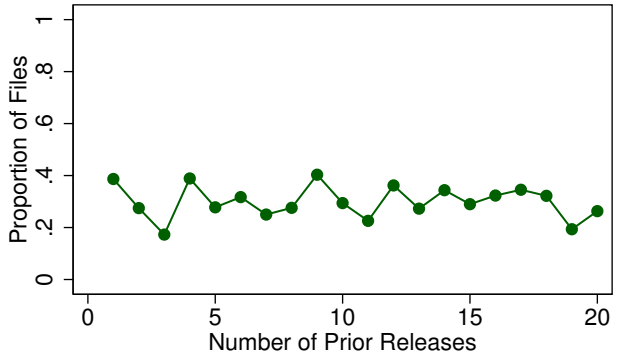

Fig. 5 Proportion of Changed Files with Multiple Developers, by File Age

5.2 Fault Prediction Results

We evaluated the predictive power of the developer metrics using data from each of the studied subsystems of the Maintenance Support system. Starting from the fully automatable model used to make predictions for Subsystems S, W, and E, (see Section 4), we added the developer metrics singly and in various combinations. To account for the skewness of the developer metrics, each was transformed by taking the square root.

Except for the number of new developers for Subsystem E, all three developer metrics had positive, statistically significant coefficients when included as the only developer metric for Releases 1 to 20 of each subsystem. The strongest relationship was always observed with the cumulative number of developers. After including the cumulative number of developers, the other metrics were statistically significant only for Subsystem S.

Figures 6, 7, and 8 present information about Subsystems S, W, and E respectively. In each case, the bar graph shown in the upper portion of the figure presents the number of faults found during each release. For all releases of each subsystem, faults were concentrated in a relatively small proportion of files—generally less than 10% of files, with five exceptions in the 10% to 15% range across subsystems.

The lower portion of each graph shows the percentages of faults included in the 20% of the files predicted to have the most faults by release, starting at Release 3, for the corresponding subsystems. Results are shown both when no developer information is included in the model, and when information is included about the cumulative number of developers who changed the file. Release numbers are aligned across the upper and lower portions of each figure to make it easy to see whether a lower than usual percentage of faults included in the top 20% of the files corresponds to many faults being missed or relatively few faults being missed.

For the model without developer information, the average percentages of faults identified in the top 20% of files across all releases were 81.1%, 93.8%, and 76.4% (with standard errors of 2.0%, 1.6%, and 5.2%), respectively, for Subsystems S, W, and E. However, performance was far from uniform across releases for Subsystems S and E. Early performance of the model was particularly poor for Subsystem E, apparently because there were very few faults to train on during the early releases. When

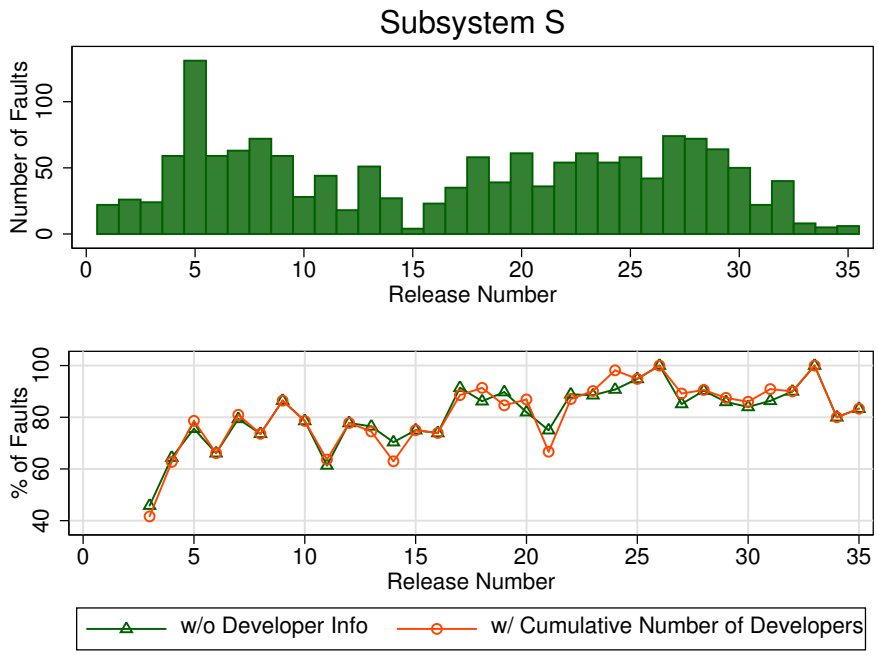

Fig. 6 Number of Faults and Percentage of Faults in Top 20% of Files, by Release Number for Subsystem S

restricted to Releases 6 and later, the corresponding averages are 83.0%, 93.8%, and 84.7% (standard errors = 1.7%, 1.8%, and 2.7%), comparable to or better than the results observed in earlier studies. This is particularly encouraging because these results are from a pre-specified model based on analysis of totally separate systems.

For all 3 subsystems, inclusion of the cumulative number of developers who had previously modified a file yielded at most a modest improvement in predictive accuracy. For Subsystem W, including this variable increased the percentage of identified faults by 1.0% (from 93.8% to 94.8%) across Releases 3 to 35. However, the improvement was only 0.2% for Subsystem S and there was no change for Subsystem E. The same changes in percentage of identified faults occurred across Releases 6 to 35 when the cumulative developers variable was included in the model.

For Subsystem W, notice that Release 24 is an exception to the otherwise uniformly high predictive performance for that subsystem. Almost all the faults not identified in the top 20% of files for this release were associated with a single, atypical fault MR that resulted in the modification of 29 executable files. In contrast, no other MR for Release 24 led to the modification of more than four executable files, and over the entire 35 releases of Subsystem W, only 14 out of 1909 fault MRs caused modification of 10 or more files. All 29 files in the single MR had been in the subsystem since Release 16, none had any faults or changes in the prior Release 23, and only three of them had a single change in Release 22. Thus, all these files had values on key predictor variables that placed them at low risk for faults at Release 24.

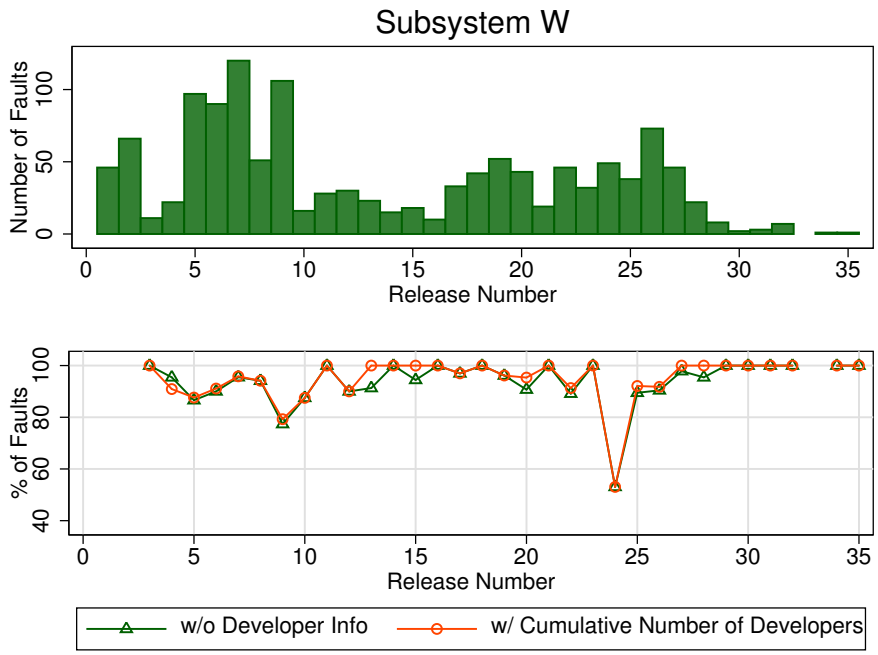

Fig. 7 Number of Faults and Percentage of Faults in Top 20% of Files, by Release Number for Subsystem W

This single anomalous point in the prediction results for Subsystem W illustrates that surprises can always occur, and that the model is not foolproof. It serves as a reminder that the prediction results should be used as an aid to prioritize testing, thereby helping to locate faults quickly. They are *not* intended to tell testers that the rest of the files do not need to be tested.

6 Related Work by Other Groups

In recent years there has been a growing interest in fault localization and predictive models. Much of the work has been similar in nature to our preliminary study [22] and identifies properties of software entities that seem to be closely associated with fault-prone code, without making actual predictions of which files are likely to be particularly fault-prone. These include [1,3,7,8,10,11,17,18,22,26].

Several groups have also developed models that predict 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 [2,6,9,12,13,16,21,28]. None of the above-cited papers, however, describes results with the same goal as ours: to identify files likely to contain the largest numbers of faults in the next release of a system. Some of the most closely related research is described in this section. points of departure.

In some ways the paper by Mockus and Weiss [16] is the most relevant to this paper, because they studied the relationship between faults and four developer variables.

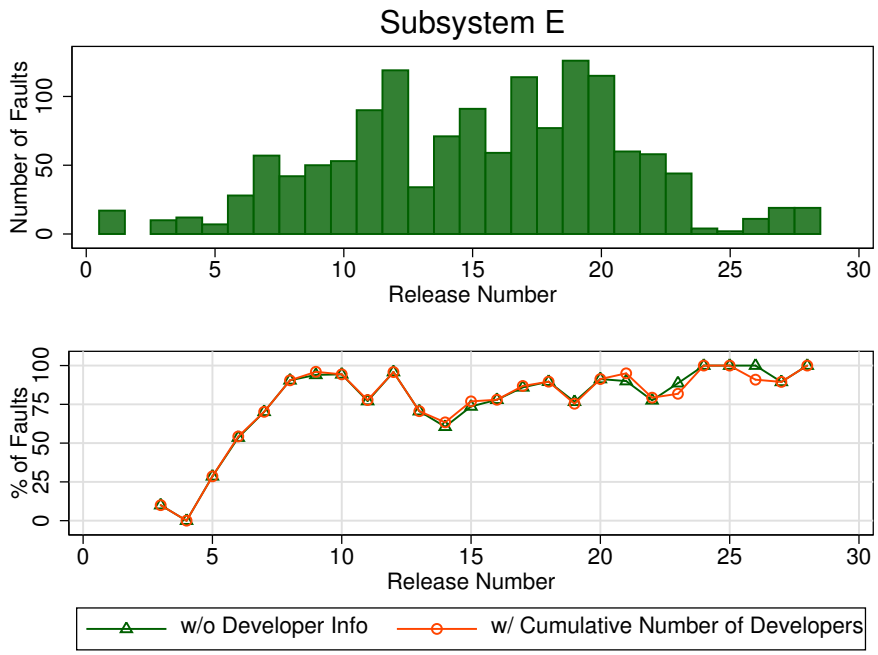

Fig. 8 Number of Faults and Percentage of Faults in Top 20% of Files, by Release Number for Subsystem E

They used ten years worth of data from a large telephone switching system with 15,000 MRs to design a model for predicting the probability that changes made in response to an MR would cause a failure in the resulting system. Their logistic regression model predicts whether or not a given change will cause a failure, but they did not assess the accuracy of their model's predictions. 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. 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, which counted the number of developers involved in completing an MR, was closest to the types of variables we considered.

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 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 addition, they only reported their experience with a single system.

Arisholm and Briand [2] created a prediction model 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 regression, they made predictions for just one future release. In our studies, we made predictions for each of the successive releases for five systems, noting how the prediction accuracy changed as the system matured. For the sixth system, we had insufficient data to allow predictions for each successive release, and so we aggregated releases. Arisholm and Briand did not use developer information in making their predictions.

Menzies et al. [15] used a naive Bayes data miner to build predictors to categorize modules as faulty or non-faulty. They found it to outperform their early prediction attempts using J48 and OneR from the WEKA toolset [29]. There are several differences from our work. First, their artifacts were much smaller than the ones we studied, containing, on average, approximately 20,000 LOC. In contrast, the systems we considered averaged over 400,000 LOC. Second, they did not look at successive releases of systems to make repeated predictions as the software matured. Across the six systems we studied, we made predictions for over 130 different releases, with individual systems being studied from two to nine years. Menzies et al. assessed their predictions using 10-fold cross validation of the data from a single release. Like many other researchers, their goal was to categorize modules as defect prone or not, rather than predicting the numbers of faults likely to be contained in each file and sorting them to select the likely worst files.

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 [22]. 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 and then selected the most accurate models based on how closely each one came to correctly determining all of the system's most fault-prone modules. The selected models were not as accurate as ours, and they did not make use of developer information. Their work was based on data collected from Apache version 1.3 and evaluated on Apache version 2.0. The best of the selected models required 50% of the identified modules to be included in order to cover 80% of the actual faults. As mentioned above, for five of the six systems we studied, the model we used identified 20% of the files containing, on average, at least 83% of the actual faults, with the files identified for Subsystem W containing almost 94% of the 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. Their study considered four releases of an industrial telecommunications system, using the first release to build the tree and the remaining three releases for evaluation. Suc-

cess was measured based on misclassification rates. More details about this work and its relevance to ours is included in [4].

Succi et al. [28] used software size as well as object-oriented metrics proposed in [5] to identify fault-prone classes in two small C++ projects (23 and 25 KLOCS). For one of the projects, 80% of the total faults were actually contained in less than 30% of the classes, and for the other project 80% of the faults were reportedly contained in 2% of the classes. They evaluated a number of different models not containing developer information, and used them to find the smallest set of classes that contained 80% of the project's faults. Over all of their models, their predictions required between 43% and 48% of all classes to include 80% of the faults. Although the goal of this work is closely related to ours, there are important differences. The sizes of their subject systems were less 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 the predictions seem to be significantly less accurate than ours. They do not describe the relation of classes to files for their subject systems, making it difficult to directly compare our findings for files to theirs for classes. In general, a C++ class can be contained in a single file, or spread across multiple files, while a single file can contain several classes, a single class, or part of a single class.

Nagappan et al. [19] investigated the ability of various code metrics to predict post-release failures for several large Microsoft software systems. For each system, they were able to use a specific set of metrics to estimate the most likely modules where future failures occurred, but found that the metrics for one system were usually not helpful in predicting failures for a different system. Because the variables in our prediction model have been consistently successful across a variety of different systems, it is worthwhile examining their results to attempt to see how and why they differ from ours.

There are several major differences between the studies in [19] and our work. First, Nagappan et al. evaluate the fault-proneness only of entities that are newly inserted into the system, while our models have been applied equally to new and previously existing files. While we have found that new files generally are more likely to be faulty than old ones, we have also found that faultiness tends to persist for certain files across releases. Second, all the inputs to their prediction models are code complexity metrics, evaluated on the static code of the program units, while our models make use of the past history of changes and faults that have been recorded for files. These two differences are obviously related–new code units have no past history, and therefore change and fault information from the past does not exist. Although Nagappan et al. relate the fault history of existing units to the complexity metrics of those units in order to create a prediction model for new units, the models themselves cannot make use of the non-existent fault history of the new units.

Third, Nagappan et al. confine their attention to post-release faults only, remarking that "for the user, only post-release failures matter". While this is true for the end-users, defects that escape unit and integration testing must still be discovered during system testing. Otherwise they will become post-release defects that can cause failures for the end-user. In our development environment, very comprehensive pre-release testing is the norm, and we believe this explains why most of the systems we studied contain either few or no post-release faults. The vast majority of the faults identified for the systems we study are found during system test. The fact that we are trying to identify different sets of faults may partly account for the differences in our results.

It is not surprising that a single model based solely on code complexity does not apply equally well to different projects that are developed by different teams. Devel-

opment teams can have significantly different styles of programming, leading to widely different values of metrics, and widely different ways of introducing errors into their code. We believe that a major reason for the success of our standard model across a variety of systems is because it is based not only on static code metrics, but also includes project and code unit history information.

A second study by Nagappan and Ball [20] analyzed the ability of software dependencies and software churn to predict post-release failures, and found that increases in the former generally indicated an increase in the latter. None of the metrics analyzed in either of these papers involved developers.

In summary, the only other related paper that we are aware of that used developer information is [16], and the goal of its predictions is different from ours.

7 Conclusions and Future Work

The experiments described in this paper lead to three main conclusions. First, the prediction model developed originally with data from systems with regularly scheduled releases, and oriented to make predictions for future releases, was successfully adapted to a system without releases. The prediction results were somewhat less accurate than for the systems with regular releases, but the top 20% of identified files still contained an average of 75% of the system's actual faults.

Second, the standardized prediction model that we developed for system S, based on our experience with the earlier Inventory, Provisioning, and Voice Response systems, performed very well on two additional systems. The standardized model's structure is fixed, and only the coefficients of terms in the model equation are adjusted according to data extracted from the history of the system whose faults are being predicted.

Third, we investigated the addition of developer access information to the model equation, and found only a slight improvement in the prediction results when these data were included. Without the developer information, the standardized model was able to correctly identify 20% of the files containing, on average, 81.1%, 93.8%, and 76.4% of the faults in Subsystem S, W and E respectively starting with Release 3. Inclusion of developer information in the model yielded 81.3%, 94.8%, and 76.4% of the faults, at best a negligible increase. Omitting Releases 3 through 5, with their relatively low fault recall, yields the same small increase in fault detection. However, the incremental cost of collecting the additional information, and of including those variables in the prediction model is also negligible, and we intend to evaluate their contribution to prediction accuracy for any additional systems that we examine.

Another issue that we expect to study is whether regular use of our prediction technology will affect its usefulness. If testers use a tool that identifies the files likely to be most problematic in the next release of the system to prioritize their testing efforts, will that affect the faults observed? Will they become so complacent about the effectiveness of the model that they ignore other parts of the software so that the use of the tool leads to perhaps a quicker identification of most faults but more residual faults that make their way to the field rather than being found during system testing? We do not expect this to be the case; however, until an automated tool is available, and development projects begin using it to prioritize their testing efforts, we will not know whether there are any unanticipated ancillary consequences. We expect to follow production projects that start using the tool once it is available.

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. T. Menzies, J. Greenwald, and A. Frank. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Trans. on Software Engineering*, Vol 33, No 1, Jan 2007.
16. A. Mockus and D.M. Weiss. Predicting Risk of Software Changes. *Bell Labs Technical Journal*, April-June 2000, pp. 169-180.
17. 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.
18. 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.
19. N. Nagappan, T. Ball, and A. Zeller. Mining Metrics to Predict Component Failures *Proc. Int. Conf. on Software Engineering*, Shanghai, China, May 2006, pp. 452-461.
20. N. Nagappan and T. Ball. Using Software Dependencies and Churn Metrics to Predict Field Failures *Int. Symp. on Software Engineering and Measurement* Madrid, Sep. 21-22, 2007.
21. 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.
22. 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.
23. 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.
24. 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.
25. T.J. Ostrand, E.J. Weyuker, and R.M. Bell. Automating Algorithms for the Identification of Fault-Prone Files. *Proc. ACM/International Symposium on Software Testing and Analysis (ISSTA07)*, London, England, July 2007.

26. M. Pighin and A. Marzona. An Empirical Analysis of Fault Persistence Through Software Releases. *Proc. IEEE/ACM ISESE 2003*, pp. 206-212.
27. SAS Institute Inc. *SAS/STAT 9.1 User's Guide*, SAS Institute, Cary, NC, 2004.
28. 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.
29. I.H. Witten and E. Frank, *Data Mining*, second edition Morgan Kaufmann, 2005.