Software Engineering Research - From Cradle to Grave

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

ABSTRACT

Although this is a talk about the design of predictive models to determine where faults are likely to be in the next release of a large software system, the primary focus of the talk is the process that was followed when doing this type of software engineering research. We follow the project from problem inception (cradle) to productization (grave), describing each of the intermediate stages to try to give a picture of why such research takes so long, and also why it is necessary to perform each of the steps.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging – *Debugging aids*

General Terms: Experimentation

Keywords: software faults, fault-prone, prediction, regression
model, empirical study, software testing

1. INTRODUCTION

One of the primary concerns in the software engineering research community in recent years has been the relevancy and the impact of the work we do. We worry that practitioners either don't know about our research, do not care about it or even find it irrelevant, and we wonder what we can do differently. In some fields of mathematics and science, theory for its own sake is enough and even desirable. Afterall, there are others who will build on the theoretical foundations, and hopefully move the research towards more applied uses. Sometimes newly developed techniques or theoretical approaches will facilitate new ways of thinking or new ways of solving problems. But as the name of our field indicates, software engineering is an engineering field and so there should be a path from successful research to application. Of course, the path is often long and difficult and non-obvious, but still that path must exist.

And let me be clear. I am not talking about the distinction between pure and applied research. Even when doing pure research in software engineering, I believe we must provide analytical evidence that the idea works or empirical evidence and ultimately both sorts of evidence if we hope to convince software engineers, i.e. those who build software systems, that our research is worthwhile. And the empirical evidence has to include the application of our research to the type and scale of systems that are the realities that they have to deal with. We also need to do reasonable assessments of the effectiveness of our newly-proposed techniques in terms that practitioners can understand and consider relevant.

Because of these convictions, I have been involved in a research project at AT&T Labs for the past six years, along with my colleagues Tom Ostrand and Bob Bell, to develop a technique that will allow large software projects to accurately predict where faults will reside in the next release of their software systems. By identifying files that are likely to be particularly fault-prone, projects will know how to best allocate their time, personnel and tool resources to ensure the construction of systems that are highly dependable, highly reliable and economical to build, evolve and maintain.

This research project represents an example of what I like to think of as the *cradle to grave research* approach. Beginning in 2001, Tom Ostrand and I developed a conceptual model of what we hoped to accomplish, not knowing at the time, of course, whether or not it would be feasible. The central process would require us to decide on a statistical model that could be used to identify the most fault-prone files.

Before deciding on what sort of statistical model would be most appropriate however, we first had to make sure that the common wisdom, embodied in the software engineering folklore, was correct: that faults were very non-uniformly distributed in practice. Afterall, if all files were equally likely to be faulty, it was not even meaningful to talk about "the most fault-prone files".

This observation led Ostrand and me to perform an initial empirical study to see whether we observed the non-uniform distribution for a large industrial software system with several years of field exposure. This involved first finding a development project that was willing to let us carefully study their development process and fault distribution database, and that was willing to answer questions when necessary. Of course we tried to be as non-intrusive as possible, but we recognized that realistically, if we did not have the project's cooperation, we could not perform even the initial empirical study.

We were lucky to find a project that was building and maintaining an inventory system which was about 500,000 lines of code (LOC) and had been in the field for about three years with twelve successive quarterly releases. We observed in that preliminary study that, as expected, faults were distributed very non-uniformly in each of the twelve initial releases. [16]

This project used a commercially-available integrated change management/version control system which has a database that contains information about every modification that is made to the system. Since the change management and version control functions are integrated into a single system, we are assured that every change is actually recorded. That is a frequent problem for researchers in this area: changes might be made without being recorded. Using this tool, this cannot happen. The only way to modify a file requires that a *modification request* (MR) be written that describes in detail the reason why the modification is to be made. Once it is approved, a great deal of information in entered into the MR database including how the system will be modified, which files are to be modified, the actual lines of code to be added, deleted or modified, the stage of development during which this MR was initiated, in which release or releases the changes will be made, and the programmer making the changes.

2. FINDING A SYSTEM TO STUDY

Finding a project is often an insurmountable hurdle for both academics and industrial researchers. Because of the issues mentioned earlier, practitioners often consider researchers more of an impediment than an asset. There is something of a paradox here. On the one hand, practitioners often feel that there is little or no evidence that the proposed research technique, process or tool will be able to scale and provide the levels of performance required by the production project. They are rightfully wary because they haven't seen sufficient evidence and adoption of an unproven approach is just too much of a risk. On the other hand, without a real large production project to try the approach on, how will we be able to provide the required evidence to the next project?

And that is why it is necessary to create a relationship between researchers and practitioners that is based on trust, mutual respect and a sense that there is a potential win-win situation for both sides of the equation. But how can this be initiated?

It requires that researchers really understand the demands that practitioners face and recognize that when a practitioner, development or test manager of a project agrees to participate in a research project, they are taking a risk and are unlikely to see a direct reward for *this* system that is being developed. They are participating generally to help the *greater good* and in the hope that the research will pay dividends on later projects. This is very similar to the medical patient who is a subject of a drug test. They are warned that it is unlikely that *they* will be helped during the trial. However, they are contributing to the future of science and medicine.

We have found the following things make it much more likely that project personnel will agree to let researchers use their projects as study subjects as well as giving them access to their artifacts:

- Don't make grandiose claims. It is unlikely that this work will revolutionize the software development process in the near term.
- Promise to be minimally intrusive in their process and follow through on that promise. They have enough demands on their time and any significant additional burdens will likely sink the partnership.
- Recognize their realities and deadlines.
- Be a team player and agree to help them if asked. This help might involve anything from running test cases when they are short-staffed to collecting and analyzing metrics for them, even if this is not the direct goal of your research.
- Recognize that in many cases, they have more practical experience building systems than you do and so *listen* to their advice and assessments of your ideas. An attitude that comes across as: "I am smart and you are not", is not likely to get you the help you need.
- Remember that they are helping *you*. Make sure they know you appreciate their time and help.

3. IDENTIFYING RELEVANT SYSTEM PROPERTIES

Having found a suitable project to be the subject of our initial empirical study, and having verified that faults were non-uniformly distributed in this system and therefore that the basic goal of our research was meaningful, we then needed to identify which properties were most closely associated with the most fault-prone files.

Our original thought had been that the best way to assess the fault-proneness of files was in terms of the *fault density* of the file which we measured in terms of faults per thousand lines of code (faults/KLOC). Using this measure, and the inventory system, we identified a number of properties that appeared to be relevant. After a year of this preliminary work, we published our first paper [16] and thought we were ready to begin creating prediction models.

However, before embarking on the statistical model development, we presented our initial findings at an in-house practitioner conference. We wanted to get their feedback on the perceived value of our research once we were able to make these predictions, and also to identify other projects that might volunteer to serve as subjects of later empirical studies.

One of the comments we received was that they believed that identifying the files that were likely to contain the largest numbers of faults rather than having the highest fault density was more valuable. Since their arguments were cogent, we decided we would build models to predict which files were likely to contain the largest numbers of faults in the next release of the software. We therefore had to repeat our initial study to identify properties closely associated with fault-prone files when fault-proneness was assessed in terms of the numbers of faults rather than the fault density.

We were also contacted by a second project which was of similar size to the inventory project but very different in both function and the programming languages in which it was written. While the inventory project was largely written in Java, with this single language accounting for roughly three quarters of the code, the second project was a service provisioning system which was much more heterogeneous in terms of languages used. For the provisioning system, the largest amount of code in a single language was written in

SQL and that accounted for less than one quarter of the code.

At this point we had been working for close to two years and we were ready to enlist the help of a statistician to help us select appropriate prediction models. Our colleague Bob Bell then joined our research team. At the end of another year we had decided on using a negative binomial regression model [13] and were ready to make the first predictions for the inventory system. This was 2004, almost three years after we started the research project.

In addition to the necessary preliminary studies that were quite time-consuming, a substantial part of the reason why it took so long for us to get to the point of making predictions was that the MR database that holds the necessary data is not organized in a way the facilitates the data extraction. In addition, although the database contains a very large amount of information associated with every MR, one thing that it does *not* explicitly indicate is whether a change was made because a failure was observed indicating a fault was present in the code, or whether the change was being initiated for some other reason such as an enhancement to the system or a specification change. Therefore, it was necessary to develop proxies to determine whether or not a proposed change was made because of a fault or other reason. We went back to the inventory project to get their advice on whether they could suggest a reasonable approximation of this. Using their suggestion, we wrote scripts to do the necessary data extraction from the database and were ready to begin making predictions for our first project.

4. MAKING PREDICTIONS

We used the properties we had identified in our initial study of the inventory system, and found that when we used the negative binomial regression model we developed to identify the 20% of the files that it predicted to contain the largest numbers of faults, those files contained, on average, 83% of the faults. These results were described in our paper [18] with more details of the model and extensions described in [19]. For our initial predictions we used the SAS [23] statistical package to create the models.

We were extremely pleased with these results but wondered whether we would observe similar behavior for other systems. Afterall, it was the inventory system that was the subject of the case study in which we identified the properties used to build the model. Maybe other properties would be more appropriate for other systems, using different languages, different personnel, and different development paradigms.

We therefore contacted the second project that had volunteered as a result of hearing about our work during the in-house practitioner conference. This was a service provisioning database system that also contained several hundreds of thousands of lines of code. At this point it had been in the field for more than two years.

We repeated the empirical study using the new subject system, and observed almost identical results - the 20% of the files identified by the negative binomial regression model as likely to contain most of the faults contained, on average, 83% of the faults. This project had a relatively small number of changes to the system, enabling us to manually read each MR to decide whether or not it represented a fault. This meant we used a different definition of what it meant to be a *fault* for the second subject system. While we needed a proxy for the first system, for this one we were able to directly identify faults.

The fact that we got the same results using this subject system which was entirely independent of the initial project that had been used to select properties was very encouraging. We felt we could use this information to enlist other projects to be subjects of further empirical studies. We were particularly interested in finding a new project with some substantially different characteristics.

The next project we approached was developing an automated voice response system. This was a relatively new project and we felt that was desirable so that we could see whether our predictions were applicable to a less mature system. More importantly, we discovered that this project did not have regularly-scheduled releases; instead they had what they called *continuous releases*. In contrast, each of the two earlier study subjects had roughly quarterly release schedules. We were excited by the prospect of studying this project because many practitioners from other companies who had heard us speak about our prediction methods had asked whether we thought it would work with projects without regularly-scheduled releases.

We were ready to begin working with this project more than four years after beginning this line of research. The first issue we had to address was how to deal with this lack of releases. We considered different ways of defining synthetic releases and finally designed a way of aggregating information in three month long periods that simulated quarterly releases, the most common length we had observed for projects.

We found that our predictions were again quite accurate, but not quite as good as we observed for the first two subject programs. For this system we found that the 20% of the files selected by the model contained, on average, 75% of the actual faults in the system. Considering that our models were designed to be used with an entirely different development paradigm, this was an extremely positive result showing the robustness of the prediction models. Details of this third study appear in [4].

5. AUTOMATING

Having performed three large empirical studies to validate the usefulness of our models for three different systems written in different languages, having different functionality and using different development paradigms, we now had evidence that our technology was likely to be widely applicable and recognized that if this technology was to be useful to practitioners, we would have to build a tool that entirely automated the process. That meant we had to identify a fourth project and see whether we could create a prediction model that was based solely on what we had learned from the three earlier projects we had already studied, applying it to the fourth.

In the previous three studies we had built new custom models for each with the help of our statistician and had done all of the data extraction, analysis and pre-processing manually. In order to automate, we would have to build a tool that would do all of these phases automatically and use a prediction model that could be applied after just one or two releases relying on the commonality that we have observed across projects.

During the third study, we had begun automating the phases that preceded the prediction phase. That portion

System	Number of Releases	Years	KLOC	Pctg Faults Identified
Inventory	17	4	538	83%
Provisioning	9	2	438	83%
Voice Response	-	2+	329	75%
Maintenance Support	35	9	500	84%

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

% Files Selected	**Type II**	**% Faults Included**
5	2%	62%
10	1%	72%
15	1%	79%
20	1%	84%
25	1%	87%

Table 2: Metric Results for Different Values of N

of the tool that would function as the front-end was now designed and partly implemented, but we were not sure whether a standardized prediction model could be developed or whether custom models were always necessary. Given that the results of all three completed studies were so similar, we anticipated that there was enough commonality that the same characteristics could be used as the basis for prediction. We were therefore optimistic that a totally automated tool could be built. We believed that this automation was essential because production software projects generally do not have extra time or personnel to do the necessary data extraction and analysis or the expertise to design the statistical models.

For this fourth empirical study, we were fortunate to identify a software system that was written and maintained by a different large company that used the same commercially-available integrated change management/version control system that had been used by the subjects of our previous studies. Since the data extraction and analysis portion of the tool relied on a modification request database in this specific format, this was essential at this point. Ultimately we envision having different front end versions of the tool for different change management or fault database tools used by projects. A common intermediate form would be created that would then feed into the prediction portion of the tool.

The subject system for this empirical study was a maintenance support system. The fact that it was written and maintained by people at another company meant that in addition to the fundamentally different functionality of this system, and the different languages used, there was an entirely different corporate culture used by the projects in the development process.

For this study, we considered four different models ranging from one in which absolutely everything was pre-determined to a completely customized model. We found that the best model was one that relied entirely on the characteristics found during our earlier studies to be most relevant. Not only was this model entirely automatable, but in addition, it produced more accurate prediction results than the custom model produced specifically for this system. Details of the four studies are summarized in Table 1 and details of this fourth study are discussed in [20].

This fourth study was completed almost six years after the research began. However, now that the data extraction and preparation portion of the tool has been fully automated, we are able to prepare a system for prediction in minutes or hours rather than months or years. Obviously this is the difference between a process that is useful to practitioners and one that is not. In addition, the automated extraction process does not require that the user know or understand the format of the database, the data being extracted, or anything about how it will be used.

We have also implemented a negative binomial regression model constructor which takes the extracted data, builds the model, and presents the user with the files predicted to contain the selected percentage of faults. The files are listed in descending order of the number of faults each is predicted to contain.

6. METRICS

Having selected relevant characteristics and an appropriate type of statistical prediction model, and used them to make predictions for several large production systems, it is necessary to assess the effectiveness of the model and resulting predictions. Initially we used the percentage of faults contained in the selected files as a measure of how good our predictions were. We still believe that this is the most relevant measure. However, it is also important to look at other ways of measuring effectiveness to convince potential users of the relevancy of this work.

Since our primary goal is to help testers identify which files they should test first and most heavily because they are likely to be most problematic, a second important measure of the predictions' effectiveness is to make sure that we do not indicate that certain files are likely to be non-faulty and therefore of little need of scrutiny when in fact they contain large numbers of faults. This is what is known as a *Type II misclassification* or a *false negative.*

For the maintenance support system we considered six different ways of assessing effectiveness when we used our prediction model to select the N% of the files predicted to contain the largest numbers of faults when N varied from 5% to 25%. In addition to the percentage of faults included in the selected files and Type II misclassifications, the metrics investigated include Accuracy, Recall, Precision, and Type I misclassifications. Definitions and the underlying philosophy for each of these metrics is discussed in detail in [17] and results are shown for each of the metrics and each of the percentages of files considered.

Table 2 shows abbreviated findings for the average percentage of faults included in the selected files and Type II misclassifications for this system because we consider them most relevant to practitioners.

7. RELATED WORK

Many other research groups have looked at issues related to fault prediction. What distinguishes our work is both its depth and breadth. We have studied each of the different phases outlined above and performed multiple, multi-year empirical studies using four different large industrial systems, each with unique characteristics.

Quite a number of groups have studied the first step of the problem: identifying properties of software systems most closely associated with software entities that had the most problems. Among papers describing such research are [1, 3, 7, 8, 10, 11, 14, 15, 21].

Several groups have also developed various types of prediction models and used them in empirical studies to make predictions for software systems. We outline some of the most relevant research and identify the differences from our work.

Arisholm and Briand [2] built a custom stepwise logistic regression model for a medium-sized Java system (110 KLOCs) for which they had 17 releases created over a seven year period. They used three releases to make predictions for one other release. Their prediction was made to categorize files as likely to be either faulty or non-faulty. They assessed their work by looking at the numbers of false negatives and false positives and found roughly 20% of each.

Another group that has performed closely related research is Denaro and Pezze [6]. They used logistic regression and static software metrics to develop a family of models based on collections of these metrics. They used the open-source Apache web server software as the subject of their empirical study, along with the associated fault database. They used data from Apache version 1.3 to make predictions about Apache version 2.0.

They also predicted fault counts, but at the much coarser module level (rather than the file level at which we worked). Like us, they sorted the modules based on predicted numbers of faults and found that their best model needed the first 50% of the modules in order to include 80% of the actual faults. In contrast, we generally required less than 20% of the files to include 80% of the actual faults.

Succi et al. [22] used the size of the software plus object-oriented metrics [5] to identify fault-prone classes. They considered three different regression models, and made predictions for two small C++ projects (23 and 25 KLOCs) In order to include 80% of the faults, their models required between 43% and 48% of the classes. Rather than looking at successive releases of the software to make predictions they instead relied on a single time interval for each system.

Graves et al. [9] also performed a study similar to ours. They considered several different models aimed at predicting the number of faults. They considered a single very large system working at the coarse module level. The authors did not consider a series of releases, but instead looked at a single two year period, and made a single prediction based on that period. They did not provide details about the effectiveness of their models.

Khoshgoftaar et al. [12] used binary decision trees to classify modules as likely to be faulty or not faulty. They used a set of 24 binary static metrics in addition to four execution time metrics. They considered four releases of a single large system, using the first release to make predictions for the next three. Their Type I and Type II misclassification rates were, respectively, 32.2% and 21.7% for the last of the releases they studied.

In summary, what we see as some of the primary differences between our work and the other research groups include:

- We did four empirical studies using different systems while most of the other groups looked at a single system or at most two systems.
- We followed each of the systems for multiple years (two years through nine years) making predictions for over 60 different releases of the systems while most of the other groups made predictions for one release or at most a few releases.
- We made predictions about which files would be most likely to contain the largest numbers of faults, while many other groups just categorized files as being either likely to contain faults or unlikely to contain faults.
- All of the systems we studied were large. Although some of the groups did study a large system, many of the others looked at only small or medium-sized systems.
- All of our predictions were relatively accurate in the sense that a relatively large percentage of the faults were contained in a relatively small percentage of the files. Most of those research groups that actually made predictions of the type that we did were far less accurate, requiring larger percentages of files.
- We observed a false negative (Type II misclassification) rate of 1%. Since this type of misclassification tells testers that a file that really is faulty, is not likely to be faulty, it gives the tester a false sense of security about the file by indicating that it does not require special attention. Therefore, we consider having a very low false negative rate to be of utmost importance. None of the other groups that reported this metric had rates nearly as low as ours.
- We have built a tool to completely automate the process from data extraction to final predictions. This tool does not require user intervention or expertise. To our knowledge, none of the other groups have build such a tool which we believe is essential to making this technology usable by practitioners.

8. CONCLUSIONS AND FUTURE WORK

So where are we now? Are we at the "grave" end of the project? Well, we are close but not there yet. There are a number of things that we still need to do before the research can be handed off to practitioners and we can walk away (or stand behind a pillar and peek without intruding).

Because we have built a tool, we can now make predictions for a large system in minutes rather than months or years. That was obviously essential to make the technology usable by practitioners. We can now also make predictions without requiring any expertise, again due to the tool. And again that was essential. We believe that the experience that we can provide in the form of the four empirical studies provides some compelling evidence to encourage practitioners to be willing to try the technology, and the metrics that we can provide enhances the evidence.

So what is left to do? First we need to do a different sort of empirical study. We need to find development projects at the beginning of their lifetimes and have them use our prediction technology as part of their development process and see how the projects do. Do they believe our predictions are helpful? Secondly, does it help them produce software with fewer faults than would happen without the use of the predictions? Does it help them produce software more efficiently?

Of course, these things are very difficult to assess since different projects may have very different fault rates. These differences might depend on the skill of their development and testing personnel, the complexity of the system being developed, the size of the system, whether the system is an implementation of well-understood and long-existing functionality (perhaps a port to a new language or a new platform) or whether it is an implementation of entirely new technology.

Another interesting factor to consider is whether the regular use of these predictors will make them significantly less effective in the future. If developers are regularly pointed at the files that would be most problematic if the predictors were not used, and as a result they learn to be more careful when building or modifying these files, what are the implications? And will testers learn to identify the problematic files without using the predictors and carefully scrutinize these files removing faults? If so, will practitioners stop using the predictors because they will become less effective with time? Does this imply that new characteristics will be needed to identify the remaining faults?

In fact if developers and testers can learn to somehow anticipate which files *would be* problematic, that would be wonderful. The goal of this work is to make systems more reliable and more economical to build. However, it seems unlikely that their intuition will nullify the use of the predictors since the predictors are designed to enhance and facilitate what they already do successfully. Since almost all of the faults in the systems studied were found during system test rather than post-release, these are faults testers are already able to identify. The goal of the predictors in such cases is to make the identification of these faults more efficient.

This brings up another issue. Because the systems that were subjects of our empirical studies were all highly reliable in the sense of having very few post-release faults, the faults that we encounter are overwhelmingly faults found during system testing, prior to system deployment for a given release. For less dependable or robust systems, would different prediction models be needed?

Yet another issue we plan to study is the nature of the faults *not* identified by our models. For every release, we observed typically 5% to 20% of the faults not identified as being contained in the top 20% of the files. Of course, we could increase the percentage of files identified and decrease the percentage of faults not contained in the files. For many systems, the files predicted to be the worst 30% or 40% include all or just about all of the faults.

But for any given percentage of files selected, what is the distribution of the faults not contained in them; faults that might be considered particularly difficult or unusual because they are not in files that have the characteristics identified by our studies? Are they typically in one or a few files, or are they typically spread among many different files? Ideally we are not missing any files that contain many faults. We expect that typically we are failing to identify files that contain just one or two faults. Given that *any* file might contain a fault, we certainly should not be surprised that we miss a relatively small number of faults if we select a relatively small percentage of files like 20%, provided that we are not missing any of the really problematic files containing many faults. We plan to do additional empirical studies to make sure that this is correct.

Although we have a functioning and efficient tool at this point, if it is to be used by practitioners, it will need to have a good user interface. This is still to be designed and implemented.

Is this research project in the grave yet? Definitely not, but it is far from the cradle at this point, walking fully upright rather than crawling, and hopefully almost ready to head out on its own.

9. 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, J. Deng. Using Regression Trees to Classify Fault-Prone Software Modules. *IEEE Trans. on Reliability*, Vol 51, No. 4, Dec 2002, pp. 455-462.

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

[14] 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.

[15] 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.

[16] 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.

[17] T. Ostrand and E.J. Weyuker. How to Measure Success of Software Prediction Models. *Proc. Fourth International Workshop on Software Quality Assurance*, Dubrovnik, Croatia, Sept 2007.

[18] 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.

[19] 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.

[20] 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 (ISSTA2007)*, London, England, July 2007.

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

[22] 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.

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