An Industrial Research Program in Software Fault Prediction

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

Abstract: It is often wondered why so much of the software engineering research that appears in the literature is not adopted by practitioners. After all, there are lots of exciting new ideas that could potentially improve both the quality and efficiency of software production. So why is this research ignored? Is it just a matter of ignorance or shortsightedness on the part of practitioners? In this paper we discuss our experience doing software fault prediction research in an industrial research lab, and describe the timeline, empirical studies and their results, and contrast them with a typical academic research program.

1 Research Overview

For the last five years, we have been conducting research on how to identify which files in large industrial software systems are most likely to contain large numbers of faults in future releases. The goal is to build a tool that will be able to automatically make these predictions without requiring the development team to spend extra time or have specialized expertise to extract or analyze data, build statistical models, or make the actual predictions. Instead this should all be provided seamlessly by a tool that will provide a list of the files that it has determined to be most likely to be fault-prone. In particular, this tool will provide this list in decreasing order of predicted numbers of faults. This information should allow testing practitioners to prioritize testing effort, and indicate to developers which files might be particularly problematic in the future and therefore might be candidates for re-engineering.

During these years, we have performed a number of case studies using four different large industrial systems, each with different characteristics. Our first case study [2] used an inventory system with 12 releases that had been in the field for three years. The study was designed to identify characteristics of files that proved to be problematic.

We restricted attention to file characteristics that could be objectively measured. Some of the characteristics that we determined to be most relevant were the size of the file as measured by the number of lines of code (LOC) in the file, whether the file was new to the current release, or had appeared in previous releases. If it was not new to this release, was it changed in the previous release, or unchanged? Other characteristics included the age of
the file in terms of the number of releases it had been in the system, and how mature the system was as a whole. We also considered how many faults were in the previous release and the programming language in which the file was written.

All of this information was provided by a commercially-available change management/version control system that was used by each of the projects that served as subjects in our case studies. In order for any change to be made to the system, a modification request (MR) had to be written describing why the change was to be made, what was to be changed, the actual code changes, and a great deal of other information. All of the information used for the predictions is extracted from the MR database or from the code itself.

Using what we learned in our preliminary case study based on the first 12 releases of the inventory system, we built a negative binomial regression model to make predictions for this system. By the time we were ready to make actual predictions, another year had elapsed during which there had been an additional five releases of the inventory system, for a total of 17 releases over a period of more than four years.

We found that our predictions were quite accurate which encouraged us to perform further case studies. Table 1 provides information about each of the systems used as subjects of our case studies, including the number of releases, years in the field, and thousands of lines of code (KLOCs). The inventory system and service provisioning system were the basis for case studies described in [4]. The automated voice response system was the subject of a study described in [1], and the maintenance support system is currently being studied.

Table 2 provides information about the accuracy of the predictions we were able to make, averaged over all releases of the relevant system. In each case we used the prediction model to identify the 20% of the files expected to contain the largest numbers of faults. For each system, the table entry indicates the percentage of the actual faults that were contained in those files. For example, for the inventory system, we found that averaged over the 17 available releases representing more than four years in the field, the model accurately identified 20% of the files containing 83% of the actual faults observed in that system.

<table>
<thead>
<tr>
<th>System</th>
<th>Number of Releases</th>
<th>Years</th>
<th>KLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inventory</td>
<td>17</td>
<td>4</td>
<td>538</td>
</tr>
<tr>
<td>Provisioning</td>
<td>9</td>
<td>2</td>
<td>438</td>
</tr>
<tr>
<td>Voice Response</td>
<td>-</td>
<td>2+</td>
<td>329</td>
</tr>
<tr>
<td>Maintenance Support</td>
<td>35</td>
<td>9</td>
<td>500</td>
</tr>
</tbody>
</table>

Table 1: System Information for Case Study Subjects

<table>
<thead>
<tr>
<th>System</th>
<th>Percentage Faults Identified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inventory</td>
<td>83%</td>
</tr>
<tr>
<td>Provisioning</td>
<td>83%</td>
</tr>
<tr>
<td>Voice Response</td>
<td>75%</td>
</tr>
<tr>
<td>Maintenance Support</td>
<td>84%</td>
</tr>
</tbody>
</table>

Table 2: Percentage of Faults in Top 20% of Files for Previously Studied Systems
We note that each of the four systems are of comparable size but have significantly different characteristics that made them particularly interesting to study. For example, the automated voice response system had no scheduled releases. Instead they used a “continuous release” development paradigm in which new code entered the system on a daily basis. Essentially they used a code, test, release process rather than having regularly scheduled releases.

The other three projects had regularly scheduled releases which began with a sustained coding phase including unit testing, followed by a substantial system test phase, and once the system test goals had been met, the entire new release was fielded. This is a much more typical development paradigm among projects in our environment.

The maintenance support system was interesting for at least two reasons: first it was a very mature system having been used continuously for over nine years. In addition, it was written and maintained by a different corporation from the one that produced the other three systems. By studying systems with many different characteristics, we hope to be able to determine whether or not there is sufficient commonality among system characteristics for our prediction models to be considered in some sense universal. This, in turn, should allow us to use the information gained from our case studies to build a fully-automatable model, and eventually a tool to allow users to apply our technology with no data extraction, analysis, or statistical expertise needed.

2 Research Paradigm

In order to perform this research, several things had to be accomplished. In this section we discuss them to highlight the importance of each of these tasks and to emphasize the difference between our research program that was performed in an industrial research laboratory, and many academic research programs. Our goal is to help explain our view of why many interesting results in the research literature do not get transferred into practice.

2.1 Getting Started

When we first embarked on this course of research, we needed to find a development project that was willing to allow us access to their MR database. In order to do this, we needed to convince very busy people with very demanding deadlines to be willing to answer questions, provide advice, and trust us not to disturb any data in the MR database, or intrude in some other way. Because we had in the past provided in-house testing advice to a number of projects and developed trusting relationships with several testers and high-level managers, we gained access to the inventory system and a few of their key personnel.

Once we had performed our preliminary case study, we presented our findings at an in-house symposium for practitioners, and asked for other projects to volunteer to become involved. A software test manager from the service provisioning system heard about our preliminary findings and our ultimate goals and volunteered to become our next subject.
system.

In each case we asked the advice of the system’s development personnel about how best to measure certain factors, and what they believed would be particularly relevant factors. We recognized that practitioners might have insights based on pragmatic experience that we as researchers might not necessarily think of.

We also enlisted the skills of a statistician to help with the modeling. We discussed our goals and described the factors we had determined to be most central based on our pre-
liminary study, and factors that we had considered that did not seem to be associated with faulty files. Based on the types of information we had available and types of results we hoped to be able to provide, the statistician suggested using a negative binomial regression model.

2.2 Specific Issues

Changes are made to software systems for many reasons. Although we are interested in all code changes, we are especially interested in changes that have been made because failures have occurred, either during testing or once the code has been deployed. Because the MR format for the systems that we have worked with is lacking a field that categorizes the reason for the MR’s creation, it was essential that we determine a proxy for identifying which MRs actually represented fault MRs. In each case we spoke with project personnel to get their thoughts on how best to make this determination.

The first project we worked with, the inventory project, proposed one rule of thumb: MRs that caused changes to be made to either one or two files were likely caused by faults, while MRs that either caused no files to be changed, or ones that caused more than two files to be changed were generally not caused by software faults. If nothing was changed in the code, then usually either the requirements or documentation were being changed, often to match the current functionality of the software. This would not represent a software bug. They explained that when an MR changed many files, it was often because of an interface redesign, causing every file interacting with the changed file to be modified appropriately.

In order to determine whether this was a reasonable rule of thumb to use, we performed a small informal study in which we selected about fifty MRs that each of us read carefully and evaluated. We included a number of MRs that changed three or more files, and some that changed more than twenty files. We found that each of these MRs that changed more than two files were in fact not fault MRs, while almost all of those that changed only one or two files were actually fault MRs. More information about the use of this rule of thumb is available in [4].

For the second case study, the service provisioning system, there were few enough MRs to actually read each one and assess whether it represented a fault or not, so no approximation was needed for this system.

For the automated voice response system, we again asked testers how they thought we might determine whether an MR was written because of a software fault, or in response
to some other factor. One of the test managers suggested that if an MR was written by a system tester, it would necessarily represent a fault, since that is explicitly the function of testers - to find symptoms of faults. Once pointed out, this was an obviously good rule to use. In addition, for this system, we convinced the MR database administrator to add a field to the MR form in which people could indicate explicitly whether an MR was or was not written in response to a fault.

Our rule then became that an MR entry represented a fault if it was explicitly indicated that it was a fault using the newly included field, or was initiated during system test, end-to-end test, operations-readiness test, beta test, or was found by a customer. Each of those testing stages is done by professional testers (as opposed to unit testing which is done by developers). We used the rule that testers find faults in our fourth case study as well.

3 Having a Broader Impact

Once we had prediction results to share from the first study of seventeen releases of the inventory system, we approached the service provisioning testing manager who had expressed interest in the work based on the preliminary study and showed her the results. We then began work collecting and analyzing data, and making predictions for this second subject system. We spent roughly one year working with this system before we were ready to make the actual predictions.

After completing the second case study with two years worth of data extracted from nine releases of the provisioning system, and getting results similar to those observed for the more mature inventory system (see Table 2), we began presenting our results at in-house venues, national and international research conferences [3], invited talks, and in a journal publication [4]. This provided extensive feedback from both the practitioner and research communities.

For the third system that served as the subject of one of our case studies, we used an automated voice response system that did not have regular releases. Since the fundamental unit of study and prediction for our prediction models was a release, this system provided a very interesting challenge for how to deal with this lack of releases. We developed a way of creating synthetic releases and used them to predict future behavior of the system. Details can be found in [1]. It is clear from Table 2 that although the accuracy of the predictions for this system were, on average, somewhat lower than those for the other systems, they were nonetheless surprisingly accurate.

As different factors that might further improve the quality of the predictions were proposed, we investigated whether they were indeed important positive factors. In [1], for example, we considered three different models applied to the automated voice response system.

As practitioners at other companies became familiar with our results, we were eager to see whether our approach and models would be applicable in other industrial environments. We approached people from another corporation with the help of some of our high-level management, and were offered access to a maintenance support system that had 35 releases
with nine years of field exposure written and maintained by another large international corporation. It was very interesting to see how similar the prediction results were for this system, even though there was presumably a different corporate culture and presumably different development and testing paradigms and standards used. We have been working with this data for close to two years.

Another large international corporation has now approached us directly and asked us to apply our technology to one of their very large mature software systems, and non-disclosure agreements have been signed. We are excited by this new collaboration - one of their researchers will provide the needed data, while we will apply our prediction technology.

4 What Can Be Done

We have outlined the steps that we went through to perform our research to predict which files are most likely to be fault-prone in the next release of a large industrial software system. Some of the things that we found were essential to this research which we feel are also needed in order to eventually transfer this technology to practitioners include:

- Get initial interest from practitioners so that access to necessary system resources are made available including developer and tester time for answering questions.

- Present preliminary results to practitioners to get feedback. This should also provide future case study subjects as well as assessments of the perceived value and guide any modifications that should be made.

- Perform multiple case studies on substantial systems with different characteristics in order to assess the generality of the proposed research.

- Present results of your case studies at both research and practitioner workshops, conferences, and/or journals. Listen carefully to comments, criticisms and suggestions. Consider incorporating any issues that have been raised.

- Collaborate with other researchers with complimentary expertise. We augmented our skills by inviting a statistician to join our research team. It may also be necessary to find a database expert or someone with data analysis expertise. Make your collaborations ones in which the whole is more than the sum of the parts - duplicating your own skills may speed things up somewhat but will not broaden the scope of your work.

- Be prepared for your research to take a very long time. Performing case studies on large industrial software systems can take many months or years. But it is essential that this be done in order to convince practitioners that your work is useful and worth taking a risk and using.

- Automate. It is often essential that an automated commercial-grade tool be built. If it will take a substantial amount of added effort to utilize or apply your research, very
few projects can afford to use it. A tool can provide expertise that practitioners may not have and make the use of the process cost-effective. We are currently building a tool and have completed the data extraction and formatting functions and selected a statistical system that will be integrated to make the actual predictions.

We often hear from academic researchers that they do not have access to large industrial software systems and so are unable to perform research of this scope. However, we have to recognize that as a community, if software engineering research is to be relevant to practitioners, we must conduct it the same way that every other engineering field does, and that involves doing large case studies, both to show proof of concept, and also to convince practitioners that the new ideas have merit and are worthy of adoption.

We have to recognize that change is often expensive and risky and so there is little incentive for practitioners to adopt new technology unless they are convinced that it is likely to be beneficial and cost-effective and that is what case studies should provide.

But what about the objection about academics not having access to large production software systems? First, there are now a number of large open-source projects that might readily serve as subjects of case studies. Often these systems also have open bug databases.

A second possibility is to spend some time in industry, either on a part-time basis or while taking a leave of absence from an academic position. But you have to realize that, in general, industrial hosts are doing you and your research a favor. If you are spending roughly a day a week there, with the goal to forward your research, consider offering your services free. It is very unlikely that what you are doing while you are visiting will in any way be useful in the near term. You are finding resources for your research.

If you can find a local company that is willing to support your research, that is wonderful, but do not expect that to be the norm. Recognize that doing large scale empirical studies is a necessary part of your research just like thinking up new ideas and writing up your results. Just because it is being done in an industrial environment, does not mean that you are providing work for the company - more likely they are providing resources for your research which costs them money and distracts from their mission. Hopefully granting agencies will recognize the importance of these types of “internships” and provide funding that covers the time spent doing industrial empirical studies.

Recognize that doing software engineering research is very time consuming. It is not enough to have a “neat idea”. The goal should be to have your ideas adopted by practitioners and for that to happen you must prove to them that the idea is viable in the large. A small case study can provide a proof of concept, but this is not a substitute for multiple full-scale case studies.

Try partnering with industrial researchers who have an empirical study program as part of their research. Recognize that all the work that goes into obtaining, analyzing, and organizing data takes a great deal of time and effort. Therefore, if you can find someone with appropriate data, and can convince them to collaborate with you, they are not exploiting you by expecting to be a full partner in the research. The research cannot be done without the data and empirical studies and collecting and analyzing the data is just as important as your new idea.
Until there is a wide-spread realization that doing software engineering research takes a
great deal of time and generally requires large empirical studies, it is likely that much of
our research is doomed to be treated as irrelevant.

References

Proc. ACM/International Symposium on Software Testing and Analysis (ISSTA2006), Portland,

Proc. ACM/International Symposium on Software Testing and Analysis (ISSTA2002), Rome,
Italy, July 2002, pp. 55-64.


[4] T.J. Ostrand, E.J. Weyuker, and R.M. Bell. Predicting the Location and Number of Faults in