A Tool for Mining Defect-Tracking Systems to Predict Fault-Prone Files

Thomas J. Ostrand
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
ostrand@research.att.com

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

Abstract

In earlier research we identified characteristics of files in large software systems that tend to make them particularly likely to contain faults. We then developed a statistical model that uses historical fault information and file characteristics to predict which files of a system are likely to contain the largest numbers of faults. Testers can use that information to prioritize their testing and focus their efforts to make the testing process more efficient and the resulting software more dependable. In this paper we describe a proposed new tool to automate this prediction process, and discuss issues involved in its design and implementation. The goal is to produce an automated tool that mines the project defect tracking system and that can be used by testers without requiring any particular statistical expertise or subjective judgements.

Keywords: Prediction Tool, Software Faults, Software Defects, Fault-prone, Software Testing

1 Background

Change tracking systems are commonly used within software development projects to allow the development group to record software faults (also known as defects) and the actions taken to repair them, as well as other changes made to a software system for any reason at all. If the change information is recorded accurately and kept up-to-date, the team is always aware of the system's status, knows what problems are currently outstanding, and can make sound decisions about whether the product is ready for release. Fault history also provides valuable data for analysis of project trends. A new project can be evaluated against the fault patterns of previous related projects, and changes made to the project's methods, tools, or organization if fault rates are significantly higher than expected.

Many large AT&T software projects use an integrated change tracking/version control system, which makes both the project code and the change information accessible to developers and testers. The version control part of the system maintains the code base of the development project. A system administrator sets up an initial framework of the project in the system, and programmers then write code and check it in under version control. When the programmer wants to modify code, s/he checks out the needed files, makes the changes, and then checks the code back in.

The change tracking part of the system maintains a database of *modification requests* (MRs) that have been written for the project under development. An MR can be written by any member of the project team, for reasons that include creating initial code for the project, adding code to implement new functionality, performing maintenance updates to the project, and reporting failures or incorrect behavior. Typically, developers write the first three types of MRs, and testers write the last type, although developers may also detect and report incorrect behavior. Failure-reporting MRs are sent to the developer team, which may use the description of the problem to determine the most appropriate developer to fix the problem.

In order to avoid confusion between references to a software system that is using the change tracking system, and references to the change tracking system itself, in the remainder of this paper we use the term *system* to refer to the change tracking system, and we use the term *project* to refer to a system under development.

Change tracking involves accessing a database that records relevant information about any changes that are made to the software. We are particularly interested in changes that are made because of the presence of faults, and we will therefore sometimes refer to the relevant portions of the combined change tracking/version control system simply as the *defect tracking system*. For every detected failure, an entry is made in the defect tracking system describing the problem and the change made to fix it, including a text description of the problem, the names of the MR writer and the developer who ultimately fixed the problem, and the specific files that were modified in response to the MR.

Our work has used the information in the AT&T defect tracking databases of several large software projects to develop a model that can be used to guide the testing of future releases of the projects. We analyzed the historical change information for these projects, categorizing faults and identifying code characteristics that are associated with faults. Using the fault information from past project releases, we constructed a statistical fault prediction model for the files of future releases [3]. As described in that paper, for one large AT&T project with 17 successive quarterly releases, the model identified 20% of the files in each new release that contain between 71% and 92% of the faults identified in that release. The average over all of the releases considered was 83%.

This prediction can be very useful to help system testers focus their efforts on the parts of the project where defects are most likely to be located. Of course, it does not remove the obligation to assure that all parts of the project have been adequately tested, but knowing that most of the problems will show up in a relatively small portion of the code means that it should be profitable to test that portion earlier and more intensively than other parts. Time available for testing is always limited, and testing the most fault-prone areas early should reveal more faults earlier. As a result, more of the precious testing time can be allocated to areas that may otherwise not be covered as thoroughly, possibly leading to software with higher reliability with the same use of testing resources. Another possibility is that testing time could actually be reduced, as the tester might uncover the same faults more quickly, leading to the same degree of reliability sooner and more cheaply than would otherwise be possible.

The fault prediction is done using a negative binomial regression model whose independent variables are characteristics of the individual files in a release. The variables include the number of lines of code in the file; whether the file is new, changed, or unchanged; the file's age (the number of releases it has previously appeared in); the number of faults found in the previous release; and the file's source language.

2 A Proposed Fault Prediction Tool

Once the mathematical framework of the model has been established, it is straightforward but tedious to apply it to the specific values for a particular release, to derive the fault-proneness predictions for individual files in the next release. We therefore propose constructing a testing tool that incorporates the prediction model, and that will allow testers to generate an ordered list of the most fault-prone files without requiring the intervention or assistance of a professional statistician, or any particular understanding of the statistics involved .

The tool assigns each file of the next release an expected number of faults, and then ranks the files in decreasing order of that number. In our earlier case study [3], we found that the total number of faults predicted for even relatively small sets of files became quite accurate. The tool will allow the tester to request sets of files that are specified either by the percentage of the project that they constitute, or by the expected percentage of faults that they will contain.

For example, the tester could ask for a list of the 20% of the next release's files that are predicted to have the most faults, as was shown for the sample software project analyzed in [3]. However, since the predictor ranks all the project's files in order of their number of predicted faults, the user can just as easily ask for the top T% of that listing for any value T.

Conversely, the user could request a listing of the minimum set of files that are predicted to contain at least P% of the faults in the release. In [3], we found that over the seventeen project releases, the top 20% of files selected by the model included from 71% to 92% of the actual faults in that release, with an overall average of 83%. If we had always wanted to identify a minimal set of files that were projected to contain at least 80% of the faults, then for some releases the model would have identified fewer than 20% of the files, and for other releases more than 20% of the files.

3 Tool Components

The tool will have three main components.

The first component extracts information about faults in the current release from the change tracking database. This information includes the names of specific files that were changed to repair each fault. This component requires an accurate determination of which entries in the database actually represent faults. We found that it was often difficult to accurately identify which changes were made because of faults and which were made for some other reason such as a new feature or a planned enhancement.

Although the same change control system is used by many different AT&T projects, different project teams use the system in different ways, and provide different information regarding the motivation behind a change. For example, one team we worked with uses an MR field that specifies either the group that raised the issue behind a change, or indicates that the change was due to maintenance or an enhancement. Maintenance and enhancement changes are clearly not faults. When the group that raised the issue was the system test group, the MR was almost certainly created because of a failure, and the change was almost certainly to repair a fault. In other cases, such as a customer-requested change, it was sometimes difficult to determine whether or not a change was due to a fault without reading through the entire text of the modification request entry.

Without an explicit identification of fault/no-fault in each report, the best approach is to analyze the text description of the change report. The reporter is expected to write a brief description of the original problem, which is usually (although not always) adequate to determine if the change is fault-based. If the description is ambiguous, and the reporter is accessible, then a personal interview can usually resolve the situation. But this requires a large amount of human effort and can be prohibitively expensive if there are many faults listed in the defect tracking system.

In a second study that we carried out [3], there were approximately 500 change reports to analyze, and the description analysis approach was feasible. However, in our original study [2], many thousands of MRs were entered into the change tracking system, and over 5000 of them were determined to be faults for the first twelve releases of the project. This was certainly too many to analyze individually, or to question each of the reporters. For that project, after discussions with the project test manager, we defined a heuristic that we used to determine what was a fault. We will further discuss the problem of how to decide whether a change report represents a defect in the next section.

After the faults have been identified, the tool's second component goes to the code base to extract properties of the files. The model uses properties of all files in the project, not just those that have been modified. Since only a small percentage of the files are usually modified in a given release, the tool can maintain a table of file properties for the entire project, and update only the entries for the files that have been modified as a result of fault detection. The file characteristics used by the model include the file size, the number of faults in previous releases, the file's change status, the file's age, and the programming language used. The first three of these can change during development and debugging of a release, and should be checked and updated for any files that are mentioned in a change report. The file's age simply increases by one with each successive release, and the programming language does not change.

Extracting all this data about files and faults is facilitated by the fact that the defect tracking system and the version control system are integrated in a single system.

Our current model does not use other static properties such as cyclomatic number [1] and inter-module coupling. Our original case study found that including the cyclomatic number in the model did not improve its predictive ability over that of lines of code. We have not yet assessed the predictive ability of module coupling. If additional studies determine that using these characteristics improve the model's accuracy, they may be incorporated into future versions, and can also be checked from the code base.

The tool's third component makes the fault predictions, using the fault and file information gathered in the previous steps, as well as the model's earlier calculations. Once the file characteristics are known, they can be supplied as input to the model, and the predictions for the next release produced.

4 Designing the Predictor Tool

As presently implemented, the fault-proneness prediction is a multi-step process with human intervention at several key points. These include identifying the particular project MRs that represent faults; identifying the files that have been modified to fix a fault MR; and obtaining properties of the modified files.

The testing tool will replace the human intervention with scripts that implement these required activities. In this section we discuss the issues that have to be solved to accomplish this.

4.1 What is a Defect?

As mentioned above, the tool's database extractor component needs a means to determine which change reports represent defects. We originally thought we could base this decision on the report's *category*. Different software projects use this field in different ways, but it always partitions the changes into the three classes of *enhancement, maintenance* and *modification*. One project uses the category to provide additional information about modifications, by identifying where in the development/testing process the need for a change was originally determined. The possible values are *development, system test, user acceptance test* and *customer*.

At first glance, it seems clear that enhancement and maintenance changes should not be counted as faults, while the modification changes are all faults.

Unfortunately, we have found that certain fields in the change report, including the *category*, are frequently not filled out accurately. Testers are always under pressure to complete work quickly, and testing time is always at a premium. We found that creators of modification requests often leave the default values in place for fields of the MR that they believe are not important.

Since the category field is frequently unreliable, we have begun to identify faults by examining the job category of the person who created the modification request. If this person is an integration or system tester, then the modification request invariably represents a fault. When developers initiate an MR, it may be necessary to understand the culture of the development group and the nature of the requested change, or it may be necessary to simply read all requests initiated by developers to determine whether they are new features, enhancements, or actual faults.

Developers can report two types of problems. The first type occurs when a developer writes a modification request

against his or her own code, after the developer discovers a problem during unit testing. In some projects, developers note all such faults, and the resulting changes are recorded in the database. More commonly, unit testing changes are never recorded; the developer simply keeps the code checked out during the unit testing phase, makes all changes to the single checked-out version, and only checks the code in once, when it's judged ready for integration. If reporting unit test faults is part of the development group culture, then the change reports based on them should be included as input to the predictor.

Developers can also write a modification request when they detect a problem or inconsistency in the project specification. In this case, the developer is actually testing and finding a defect in the specification. There may not be any code yet written, and hence none to be changed, or the developer might have written code originally based on a mistaken understanding of the specification. These specification MRs represent real faults that have to be corrected, and they should be included as input to the predictor.

However, many changes initiated by developers are not defect fixes, but are enhancements, new features, or modifications to keep the project consistent. Separating these from the fault-based changes requires more detailed analysis of the change description, or a face-to-face meeting with the developer.

This situation leads us to the third, and best, means of identifying a change report as originating with a discovered defect: ask the person who has written the change report. To help collect this answer as painlessly as possible, the management of one project has agreed to augment its MR creation form with an explicit fault-classification field with 3 possible values: "Fault", "no fault", "unknown". To avoid false answers generated by testers or developers in a hurry, the field has no default value. Future users of the change report system would be expected to fill out this field. This simple addition to the MRs should both improve the accuracy of the data, and hence the accuracy of the prediction, and also simplify the data mining process, making it possible to automate this portion of the data extraction.

Since the fault-classification field has been added only very recently, we do not yet have any analysis results for change reports that use it. However, we do believe that it will be an essential part in fulfilling our goal of building a tool to automate the identification of the most fault-prone files of a project.

4.2 Obtaining Properties of Files

Obtaining file properties is a two-step process: first, the fault MRs have to be interrogated to identify the proper files, and second, those files have to be located in the code base and analyzed.

The AT&T defect tracking system is a proprietary relational database that makes information available as either a text file report or an Excel spreadsheet. The Excel format allows users to extract certain large classes of change reports and feed their information directly into a spreadsheet. Once the entries are in the spreadsheet, it is quite simple to create tables that show relations between different attributes of the change reports, and that can summarize totals of different types of reports. Unfortunately, the current version of the extraction into Excel only includes those fields of the change report that have a fixed number of possible values. While this covers many of the change report fields, it does not include a list of files that have been changed, since there can be arbitrarily many of them.

The text reports are produced by a set of specialized commands that can search for data and create text files with varying levels of detail. These commands can be used to access all the database's information, but apart from specifying which details should be included, the user has little control over the format of the text report. This means that post-processing programs must be written to extract the particular details that are needed for our fault-proneness prediction. In particular, we have to search through the report to find the text strings that name the files that are associated with each fault. We presently do this search with stand-alone shell scripts that are run by a human on the change system's text reports. The tool will integrate the change system's commands with these scripts to remove the necessity for human intervention.

Once the files are identified, we again use stand-alone scripts initiated by a human to extract their static properties. The scripts are run over the latest build of the project code, and include simple line and character counts. More complex code metrics, like the cyclomatic number, have also been computed, although our current prediction model does not use them. A possible structure for the predictor tool is to run these code analyses independently, to build and maintain a table of their values. The tool can then use the table entries whenever they are needed to produce fault-proneness results.

5 Using On-Demand Fault Predictions

The original intent of the fault-proneness predictor was to provide guidance for testers when testing starts on a new release, helping them to focus their efforts on a small percentage of the files that the model predicts are most likely to contain faults. In this scenario, the fault-proneness prediction for release N+1 would be made as close as practical to the beginning of system testing for N+1, to take maximum advantage of the fault data generated by the testing of release N.

However, we can also envisage a tool that would be

able to produce the model predictions on demand at any time during the development and testing of a given release, based on the latest information that has been entered into the change database, and the latest configuration of the code.

The availability of on-demand fault-proneness prediction would allow testing guidance based on release N to be given during the testing of release N. A possible scenario would be to run the on-demand prediction every night after the system testing phase has started. This would augment all the prior fault data with the current day's failure and fault information, and would provide the testers each morning with fresh guidance on where to concentrate testing efforts. The daily prediction should not be used to completely revise the project's original test plan. Rather, its information should be viewed as supplemental, giving advice about potential serious trouble areas in the code.

6 Summary

The tool proposed in this paper is an outgrowth of our studies of defects and characteristics of files containing defects reported for large AT&T software projects. It provides an automated framework for the fault-proneness prediction model that we developed earlier, and will allow testers and developers to generate and use the prediction results without the assistance or intervention of specialists.

The issues encountered in designing the tool include how to identify modification requests that represent software defects, how to make use of data extracted from the repository to interrogate the development project's code base, and how to present the tool's capabilities to its potential users in the most useful way.

The tool design is presently at a very early stage, but the success of the prediction model in our case studies encourages us to believe that this will eventually become a highly useful element of the system tester's toolkit.

References

[1] T.J. McCabe. A Complexity Measure. *IEEE Trans. on Software Engineering*, Vol 2, 1976, pp. 308-320.

[2] 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.

[3] T. 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.