Characterisation, Classification and Prediction of Software Defects

Elmar Stellnberger

Department of Informatic Systems, University of Klagenfurt, Austria
estellnb@elstel.org

Abstract. Classification and prediction of software defects is central to software quality assurance and control of the whole software engineering process. Bugs need to be classified correctly when implementing the software quality assurance process. Qualitative models are based on this data and can be used to improve process quality, customer satisfaction and to explore defect associations and propagation. Quantitative models are indispensable for resource allocation since not all units can be inspected and reviewed. Static, process, churn and entropy of source code metrics need to be compared and can be combined with pre-release defects to achieve better results.

1 Introduction

There are two different types of models for software quality prediction and assurance: Qualitative and quantitative models. While qualitative models require an appropriate bug data acquisition which needs to be implemented into the existing software development process, quantitative approaches can be applied more easily. They may operate with existing data extracted from the versioning system or changelog. Qualitative models on the other hand provide an early indicator of process problems which can not be achieved with traditional methods. We will discuss both kinds of models including their strengths and weaknesses.

Counting the mere number of bugs would be an insufficient predictor for software quality assessment as it neither accounts for the severity nor the field of impact. Even with bugs classified like this as well as an optimized software quality and process management it remains a problem to assess and control software quality in advance. We will focus on how defect data can be used for process improvements in the section about Classification and Prediction. Classifying defects correctly and usefully is indispensable when it comes to assess process quality, customer satisfaction and defect propagation.

The strength of numeric models on the other hand lies in a good resource allocation for testing and formal verification. It is desirable to have a high prediction accuracy. As many defective units as possible should be inspected (true positive rate, recall) while the additional effort to inspect clean code units should remain low (false positives). Employing cost sensitive prediction allows to trade off a certain degree of the false positives and accuracy against a higher recall which is particularly important in order not to miss important bugs (see also: ROC curves) [8]. Numeric prediction is a core issue of quality assurance.

To measure the actual impact of defects on software quality and thus the defect density in the final product most of the studies focus on post-release defects discovered about 6 month after the final release to the customer [5,6,7,8]. Not everything experienced as a failure by the customer is a defect due to an actual error in the source code.

The maximum net gain of quantitative models can be visualized with a cumulative lift chart which shows a steeply increasing curve at the beginning which becomes more flat when it later on approximates to the 100% mark [7]. It shows that 20% of the files contain 100% of all bugs while 20% of all LOC (lines of code) contain 70% of bugs. The bugfix approach which has been used as comparative measure for current bug prediction methods by Ambros et al. shows that by inspecting 20% of predicted files, 75% of bugs can be found while 20% of predicted LOCs account for 40% of all bugs. There are even better performing measures than extrapolating the past number of bug fixes.

In the following we will show how to collect and evaluate the necessary data to improve the software development process. A plethora of different measures and approaches has evolved by the time. We will present and discuss the most important metrics and how they can be deployed. This article is meant as a short introductory overview over the most important topics of software defect prediction. It should introduce the reader into the interesting and exciting field of software quality engineering.

2 Bug Classification and Data Retrieval

Multiple classification schemes have been developed [1]. IEEE standard 1044-1993 includes the *process activity (trigger)* and the *phase* during which a defect has been found as well as its *suspected cause* and the *defect type* or *taxonomy* which describes the nature of the fix. The Linux kernel Bugzilla f.i. keeps hold of the component, tree and version along with the *severity, domain dependent attributes* like hardware and operating system in which a bug has been found as well as whether it was a *regression*. Other attributes like status (resolved/fixed, open, etc.), keywords, tags and *inter-bug dependencies* serve the purpose of finding, ranking and working on bugs more easily. Some of these informations have been frequently reverse engineered from the *report* and *last modification date* of the bug tracker together with the versioning system being used[1].

The *source* of a defect may basically be any document during any phase such as requirements analysis, design or code. For means of extracting process and product related metrics it is important to localize the fix of a given report in the versioning repository i.e. finding the source of a defect.

A study by Schröter et al. which shows the impact of library usage on bug densities succeeded to reverse engineer 70% of all report-fix links committed by the same developer as the report was resolved within a time frame of 12 hours [6]. Commits resolving a certain bug typically contain a message with a string like 'fix', 'fixed' or 'resolved' and 'bug number/#number' or perhaps just the string '#number' but nothing like 'prefix' or 'postfix' [7]. A company which implements a quality engineering process should establish bug reports and fixes in the versioning repository to be linked explicitly and encourage developers to deploy the necessary means in practice; f.i. mentioning the URL of the report in the commit or attaching a patch containing a commit number to the report may be sufficient to do so.

In order to find out which bugs could be related Song et. al have employed a sliding window protocol on bugs being filed on the same project or component not being more than a day apart [4]. Methods like these may be necessary if data is missing or not designated by the classification scheme. Sometimes associations and classifications may even need to be established by hand with hindsight like it was the case for the basic study about orthogonal defect classification for which the authors started to manually classify after regression analysis [2].

Another important source of data for bug prediction is the changelog though we will not examine this in detail here. While some experiments utilize the data from the versioning system, many studies combine it with the bug database, however, still focusing on bugs that have been resolved [7,5,6].

3 Classification based Bug Prediction and the Software Development Process

Large companies like IBM, HP and Motorola have their own classification scheme. Considering the current state of the art such a classification scheme should at least include a set of orthogonal classifiers which means that bugs are classified into categories that are independent and thus do not overlap. The primary approach by

1 e.g. cvs, git, svn

IBM initially employed for defect data of an operating system development project was later on called Orthogonal Defect Classification or ODC and adopted by HP and Motorola as well due to its analytical strength.

There are three main categories of ODC: D*efect type* and *mode*, *defect trigger* and *defect impact*. The most important category is the type classification by the actual implementation properties of the applied fix. It has been shown that each type is prevalently discovered during a certain phase of development: Functional defects prevalently during design, assignment defects frequently during coding, checking defects often show up at function tests and problems such as timing or synchronization issues during system test. If the wrong type of defect shows up in the wrong phase of development this may indicate a process problem. A large number of functional defects in a testing phase indicates that the development should still undergo significant design or coding effort. It tells that these phases deserve more attention. Classifying defects by type provides a very early indication of process problems which can not be achieved with traditional methods where problems do not show up before system, integration testing or delivery.

The defect mode can be one of missing, unclear and wrong [1]. A very early study has shown that defect types are related to *symptom groups* each group having a characteristic inflection of the reliability growth curve which shows whether the discovery of a bug may depend on the discovery of previously found bugs [2]. It showed that symptom group one had a specifically high amount of missing initialization errors and thus the highest inflection. Most bugs where either easy to find and dependent or independent and hard to detect. The study stated that improvements in design and specification would have made a significant difference.

The defect trigger on the other hand can be used to improve the effectiveness of system testing. If a bug is found during the design compatibility test of an inspection then Design Compatibility would be the trigger. Triggers are grouped by the process activity that triggered the defect detection like review, inspection, unit, function, system and field test. If there is a deviation in the bug distribution between system and field/beta testing, that points to specific problems or weaknesses in system testing.

Finally the defect impact is suitable for assessing customer satisfaction.The impact dimension has an impact type being one of serviceability, availability, installability or expandability, just to name a few. The GSMBSS (GSM Base Station Systems) software development organization of Motorola has counted the number of post release defects in 1996 an found that most reported defects were due to capability, followed by serviceability and reliability [3]. That may be a process indication on whether to apply usability engineering. As usability was ranked after availability it did not seem to be a problem for the customers of GSMBSS. Merely counting the defect impact type may however be not sufficient as the severity ('blocker', 'critical':

crash/hang/data loss, 'major', .., 'enhancement') of each impact needs to be taken into account as well.

Software defect association mining does also use classification data [4]. It can discover interesting relationships such as a bug in an external interface being accompanied by a computational defect in 70% of all cases. The mined data set showed that data value errors which are neither type nor omission errors but commission errors take one hour or less to fix in 65% of all cases. The study mined over 1000 relationships with an accuracy of about 95%, a false negative rate in the area of 3% and a recall of about 87%.

4 Numeric Bug Prediction

In the following sections we will differ between source code metrics, process metrics and other approaches for predicting the number of bugs in a given module or file. While process metrics have been proven to perform better than the so called product related metrics at least when studied in isolation our introduction first focuses on the traditional approach of static code metrics. Some of the best performing metrics in a benchmark conducted by Ambros et al. are still calculated on base of source code metrics like churn of source code metrics and entropy of source code metrics.

4.1 Static Code Metrics

The most simple and long-serving software metric is the mere number of lines of code (LOC). Though increasingly diverse, elaborate and innovative metrics have been established by the time this is still a relevant metric which has been proven to contribute relevant information under any kind of circumstances. Though this approach is oversimplifying and clearly suboptimal as it just yields a straight diagonal line on the cumulative lift chart which we have discussed in the introduction it can be used to measure and compare the performance of other metrics. Early works state an approximation of 23 defects per 1000 LOC according to Akiyama's first equation and an approximate defect density of 8 to 12 after Fagan inspections [9].

The next generation of software defect predictors was function based counting the number of parameters, blocks, global variables read or written, counting how often the address of a variable has been taken, counting the call graph (FanIn, FanOut) or how many arcs a function in the control flow graph has. A procedural metric, to which predictive power is widely attributed, is McCabe`s cyclomatic complexity which measures the number of independent paths through the program. The higher the cyclomatic complexity the more test cases and possible execution paths to think of.

The most recent set of static code metrics is object oriented (OO). Examples for OO metrics are the weighted method count per class, the inheritance depth, the number of subclasses, the coupling between classes in terms of attributes, parameters and return types or the visibility of class elements. The benchmark which we will discuss in 'the performance of metrics' uses the Chidamber & Kemerer suite along with some supplementary OO metrics [7].

A fundamental study conducted by Microsoft concludes that there is no single set of static code metrics to be universally applicable. The predictive power of metrics for each investigated software project is largely different with some projects yielding good results for OO metrics, others for functional metrics or both OO and functional metrics. The set of deployable static code metrics needs to be fine tuned for each project based on its history. Notably only one project was correlated with none of these metrics except mere LOC. It was performing regular refactorings based on software metrics [5]. It has been proven that refactoring improves software quality [8].

4.2 Process, Change and Other Metrics

While the software development process is governed by many qualitative and people related issues, what can be measured best are mere change metrics. Many properties of the software development process become condensed and measurable by change metrics like the number of refactorings applied, the number of contributing authors, previous bugfixes, the number of revisions commited to the repository of the versioning system, the code churn measuring added and deleted lines of code as well as the maximum and average changeset of files commited together.

The code churn source code metric may not only be based on the mere LOCs but on any other static code metric (churn of source code metrics class). Another derived set of metrics is the entropy of source code metrics based on Shannon Entropy. While code churn metrics account for the changes of metric values over time these class counts the distribution over entities on a file or module level.

Some alternative approaches have also investigated the effect of usage relations between components. Network analysis on dependency graphs can be used to predict special 'escrow binaries'. The so called escrow binaries are critical binaries like the operating system kernel which need to undergo a special protocol on changes including more extensive testing, fault-injection, code reviews [6]. A study by Schröter et al. has additionally found that the defect proneness of a component is significantly determined by the set of components that it uses. This fact is mostly determined by the problem domain of the used component. 71% of the components using the compiler package but only 14% using the ui package needed fixing [10].

4.3 The Performance of Metrics

A key study conducted by Moser et al. shows that process metrics outperform static code metrics [8]. It uses Naïve Baise, logistic regression and decision tree learners on a file level. These results have been affirmed by an independent benchmark of metrics by Ambros et al. [7] which is mainly based on regression models. It compares metrics of five different projects[2] in three different scenarios: Binary classification, a ranking by the number of bugs per class or file and the bug-count per LOC. The set of process metrics employed by Moser as well as certain churn and entropy of source code metrics were among the top performers in all three scenarios. All three metrics were significantly better than the others for binary classification and ranking per class. They were still significantly better than code metrics in the ranking per LOC.

However, the same benchmark did also find out that static code metrics where the most frequently selected for decision tree learners, Naïve Baise learners and generalized linear logistic models [7, table 8], if these machine learners were able to select any attribute of choice. This indicates that static code metrics do still encode relevant information.

The study conducted by Moser et al. has also tried a combined approach of change and code metrics [8, table 5]. Their graphical result plot shows that code metrics may have little to add. As the results of a combined approach were not significantly better than the more simple change metric based approach the study suggests to drop the combined approach in favor of a single process related approach that is more simple to implement. It argues that both approaches were not orthogonal. Combined with the number of pre-release defects the study by Moser has achieved an accuracy of 95%, a true positive rate of 90% and a false positive rate of less than 2%.

Network analysis on dependency graphs can be used to predict special 'escrow binaries' with an accuracy of 60% versus 30% yielded by static code measures. The recall for defect prediction is 10% higher than for static code measures [10]. The mentioned alternative approaches have not yet been included in any common benchmark.

5 Conclusion, Future Work and Threats to Validity

We have investigated qualitative and quantitative approaches to bug prediction and found that both methods complement well in improving the software quality engineering and process engineering as a whole.

The right choice of bug classification mechanism is an important issue in the quality management of any company. Investigations have shown that classifiers used

2 Eclipse JDT Core, Eclipse PDE UI, Equinox framework, Mylyn, Apache Lucene

for bug tracking should include an orthogonal set of classifiers while a more elaborate set of classifiers may be desirable for root cause analysis.

The methodology for numeric bug prediction has evolved over the years and is now a statistically well founded discipline ready to be deployed on a larger scale in practice though more applied research in this field may be desirable. In a world with limited resources adequate bug prediction is essential because only a limited number of code units can be sufficiently tested and inspected.

All investigations have shown that in order to achieve predictive power the underlying set of base metrics must be independent or 'orthogonal' [2, 3, 5, 7]. The prediction of future failures or post-release defects needs to be based on historical data no matter which approach is used. Missing domain knowledge as based on the history of a project explains why some elder studies may have arrived at partially contradictory results [7].

There is still a lot of work to do. Existing approaches may need to be compared and combined while collecting further evidence and achieving better reproducibility on existing results will be necessary. There may be more powerful yet undiscovered metrics and methodologies besides the existing wealth of approaches. In certain project environments fine tuned static code measures may perform better than in our studies [5,7,8]. Over-fitting of the problem domain by an excessive number of rules or attributes selected by machine learners can be a problem under certain circumstance leading to degraded real predictive performance. Even the used base data being partially reverse engineered may be flawed or insufficient for effective bug prediction.

In the future, as this particular field evolves, software quality and defect prediction may be increasingly used to affirm or reject design decisions. A good design will yield low defect-proneness and thus less consequential charges. Applying regular refactorings has already proven to be beneficial. There have been studies on the optimal module size. The CMM (Capability Maturity Model) tried to improve software quality by bureaucratizing the development process. Whole approaches such as the CMM have at last lost reputation because there was no evidence in their effect on the residual defect density [8,9]. Basing design decisions on evidence could revolutionize software engineering though gaining statistical significance is rather hard to achieve.

We believe that the area of bug prediction is an interesting and challenging field of software engineering which deserves more attention.

References

[1] Wagner, S. (2008, July). Defect classification and defect types revisited. In *Proceedings of the 2008 workshop on Defects in large software systems* (pp. 39-40). ACM.

[2] Chillarege, R. Orthogonal Defect Classification. In M. R. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 9. IEEE Computer Society Press and McGraw-Hill, 1996.

[3] Bridge, N., & Miller, C. (1997). Orthogonal defect classification using defect data to improve software development. *Software Quality*, *3*(1), 1-8.

[4] Song, Q., Shepperd, M., Cartwright, M., & Mair, C. (2006). Software defect association mining and defect correction effort prediction. *Software Engineering, IEEE Transactions on*, *32*(2), 69-82.

[5] Nagappan, N., Ball, T., & Zeller, A. (2006, May). Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering* (pp. 452-461). ACM.

[6] Schröter, A., Zimmermann, T., & Zeller, A. (2006, September). Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering* (pp. 18-27). ACM.

[7] D'Ambros, M., Lanza, M., & Robbes, R. (2012). Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, *17*(4-5), 531-577.

[8] Moser, R., Pedrycz, W., & Succi, G. (2008, May). A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on* (pp. 181-190). IEEE.

[9] Fenton, N. E., & Neil, M. (1999). A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, *25*(5), 675-689.

[10] Zimmermann, T., & Nagappan, N. (2008, May). Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering* (pp. 531-540). ACM.