Getting Defect Prediction into Industrial Practice: The ELFF Tool

David Bowes*, Steve Counsell†, Tracy Hall†, Jean Petric* and Thomas Shippey*
* University of Hertfordshire, Hatfield, United Kingdom
Email: {d.h.bowes,j.petric,t.shippey}@herts.ac.uk
† Brunel University London, London, United Kingdom
Email: {steve.counsell, tracy.hall}@brunel.ac.uk

***Abstract*—Defect prediction has been the subject of a great deal of research over the last two decades. Despite this research it is increasingly clear that defect prediction has not transferred into industrial practice. One of the reasons defect prediction remains a largely academic activity is that there are no defect prediction tools that developers can use during their day-to-day development activities. In this paper we describe the defect prediction tool that we have developed for industrial use. Our ELFF tool seamlessly plugs into the IntelliJ IDE and enables developers to perform regular defect prediction on their Java code. We explain the state-of-art defect prediction that is encapsulated within the ELFF tool and describe our evaluation of ELFF in a large UK telecommunications company.**

I. Introduction

A great deal of research has been done over the last 20 years on software defect prediction. Our previous study identified 204 published papers reporting on defect prediction between the years 2000 and 2012 [3] with subsequent studies showing a continued growth in the number of defect prediction studies [5]. Defect prediction is a popular and thriving area of academic interest. Despite this huge research effort it is increasingly clear that defect prediction has not transferred into industrial practice [6, 5]. Consequently companies are not enjoying the benefits that defect prediction is reported to deliver. These benefits include being able to identify defects early and therefore repair more economically [8], as well as enabling the effective targeting of testing on potentially problematic code. One of the reasons that defect prediction remains largely an academic area of interest is that there are no good tools for defect prediction readily available to developers [5, 7]. In response to this problem we present our *Ensemble Learning for Fault Finding* (ELFF) defect prediction tool[1].

Lewis et al. [7] has shown that software developers are enthusiastic about the idea of defect prediction, however there are a number of issues that have prevented the adoption of defect prediction. ELFF addresses reported reasons why developers have not taken up defect prediction, including:

- Developers do not like a high number of false positive predictions [7], as these false positives waste their time. ELFF allows the developer to control over the number of false positives, by allowing the developer to chose the prediction threshold. For example, developers can choose to only see predictions with a probability of at least 95% likely to be defective.
- The granularity of predicting defective code is problematic for developers [7]. Most defect prediction models make predictions at the class or file level. This high level prediction granularity is of limited value to developers, as it takes developers significant effort to manually hone in on a predicted defect in a class that may well contain 20 methods and be hundreds of lines of code long. Such predictions are not been practically useful to developers. ELFF predicts at both the method level and class level.
- The lack of actionable results from defect predictions hampers the uptake of defect prediction [7]. Our interactions with our industry collaborators show that the developers want to understand what code features have motivated a defective prediction and how they could potentially solve the problem. ELFF allows the developer to access full information on the features of particular code that has been predicted as defective.

ELFF is made up of two parts, a backend that deals with the entire process of defect prediction, and an an IntelliJ plugin, which visualises the results to the developer. ELFF is being used successfully in a real industrial systems and has been well received by developers from a large UK-based telecommunication company.

This paper presents background about the research underpinning the development of ELFF in Section 2. In Section 3 we describe the main features of ELFF and provide some screenshots of the tool. Feedback from our industrial evaluations and how we plan to extend ELFF is presented in Section 4.

II. Background

There are many tools being used by developers in a commercial setting to identify where defects may be,

[1]The tool will be available upon acceptance of the paper at http://www.elff.org.uk/data_tools/ELFFDefectPredictionTool

for example static analysis tools FindBugs[2], PMD[3] or SonarQube[4]. These tools look for known suspicious code and have rules to detect these violations [1, 9]. Software defect prediction is different to static analysis, as it does not have any pre-defined violations, instead it uses the systems defect history to develop models that predict based on defects the system has had in the past. This creates a tailored prediction for each system individually. The reasoning being, that if this problem has occurred in the past during the development of the system, then it could happen again in the future.

There are tools available that allow the use of defect prediction, e.g. DePress, AgenaRisk, Prest and Dione. DePress, developed by Hryszko and Madeyski [4], is a framework which allows for software measurement. Hryszko and Madeyski [4] have used their framework in the Volvo Group and estimated that DePress could reduce quality assurance costs by around 30%. Another defect prediction tool is Agena[5]. Agena uses Bayesian networks to model where defects may be inserted at the module (method) level. These results are displayed in various risk maps or tables. Dione is a web application which automatically collects software code metrics and then performs defect prediction on the collected metrics [2]. Our ELFF tool is different to DePress, Agena and Dione as it an out of the box software defect prediction tool that requires little setup. These tools require extensive set up and show the results of defect prediction in a separate window or separate program, a known deterrent to the wider use of software defect prediction [7]. Our ELFF tool highlights potential defects to a developer in their coding environment, during development.

III. The ELFF Tool

The ELFF tool is made up of two main parts - a back end that conducts the software defect prediction, and a front end that deals with the visualisation of the results from the back end within a developer's integrated development environment (IDE).

A. The back end

The back end of the ELFF tool gathers previous defect information, collects source code metrics, combines these two pieces of information and then performs defect prediction.

The back end has three main parts. The first two parts gather the historical defects and collect the source code metrics. These two parts are explained in more detail in Shippey et al. [10]. The third part manages the defect prediction on the metrics collected in parts one and two.

1. The first part collects historical defect information. To do this, we created our own implementation of the popular SZZ algorithm [11]. The process involves matching commit messages from software versioning systems to bugs reported in bug tracking systems. This process allows the identification of defect insertion points and fault fix points. This information allows us to identify at what point in time there was a defect in a particular class or method. The ELFF tool provides support for both Git and SVN software repositories and a wide range of bug databases (Bugzilla, Jira, SourceForge issues and GitHub issues).

2. Currently we collect the source code metrics using the JHawk[6] tool. The JHawk output is parsed to collect metrics at both class and method level[7]. The tool can extract metrics from various sources, and is not limited to using JHawk output. The metrics are combined with the defect information gathered in part one to form the datasets to be used in our defect prediction models.

3. The final part performs the defect prediction. Using the metrics gathered above (or other datasets if preferred) we can create defect prediction models. Currently we have support for four different machine learning algorithms - J48, Random Forest, Logistic Regression and Naive Bayes, but more can be added easily in the future. To perform the defect prediction on the current code, we create models using a previous snapshot. A snapshot is a specific commit or date during the lifetime of the software product where we have identified which methods are defective. The models are saved and can be used again and again. Each model predicts on the current code base, giving a probability of between 0 and 1 as to how likely the class or method is to contain a defect. In the visualisation of the tool, these probabilities of the four models are aggregated to give an average defective probability of a method.

B. The IDE plugin

To make the output from the backend of the tool more accessible to developers, we developed an IntelliJ IDEA[8] plugin. To start, the developer only has to select a small amount of options to start the defect prediction process. A commit tag, where the repository is and the source files on which to perform predictions. The prediction model created in this process can be used on future versions of the code, or at each time defect prediction is performed, the models can be recreated. If the project is a relatively new project and has no/limited amount of previous faults, then the developer can use models that have been created using open source project metrics available from Shippey et al. [10].

The developer has a choice of viewing the defect predictions in two ways - a file view or a system overview. The file view will put a marker in the gutter of the code (where the line numbers are or other warnings) as seen in Figure 1 next to each method that has a probability of

[2]http://findbugs.sourceforge.net/
[3]https://pmd.github.io/
[4]https://www.sonarqube.org/
[5]http://www.agenarisk.com/agenarisk/
[6]http://www.virtualmachinery.com/jhawkprod.htm
[7]For a full list of the metrics created see here - http://www.virtualmachinery.com/Jhawkmetricslist.htm
[8]https://www.jetbrains.com/idea/

over 50% being defective. At this level, the marker is quite small and not intrusive, however as the probability of the method being defective increases, the size of the marker increases, until it goes over a set threshold when the method code is highlighted yellow, seen also in the Figure 1. This threshold has a default defectiveness probability of 95% but this threshold can be changed to reduce the number of false positives to a level that the developer is happy with.

The system overview shows the developer the packages, classes or methods that contain the most probable defects quickly and easily. When the system view is opened, it lists the packages with the highest number of methods that are over the chosen probability threshold (this threshold is changed by moving the slider at the bottom of the system view). The developer can drill down into the package and find the classes with the most probable defects, and then drill down again to see the most probable defective methods. Once the developer has found the method they want to potentially want to look at, the system view can display the code of this method below the search list or open the methods file in the code editor pane.

In both the developer and system views, there is a chance for the developer to gain deeper insight into why that particular method may be predicted as being defective over the other methods, by viewing the metrics used to determine the probability of defectiveness. Figure 2 shows an example of the metrics for a method. Each metric has its own normalised bar with three pieces of information. A black bar that displays the acceptable value ranges determined by JHawk, a green bar that displays the company's normal range as calculated by ELFF on the metrics provided and a dot to show that methods actual metric value. The dot is coloured red, if it falls outside the normal range for the company, or blue if it is inside the normal range. Hovering the mouse over the metric name offers a chance for the developer to discover what that metric is and what a potential step is to correct a metric value that is outside the normal range 3. As you can see in Figure 2 the dot for HDIF or Halstead difficulty is coloured red as it is out of range of both the system and JHawk normal range. If we hovered over the HDIF button, it will tell us what HDIF is, and that to fix this value, the developer may want to reduce the number of unique operators.

An important feature of the tool, similar to online discussion forums like StackOverflow, allows developers to vote on predictions. Whilst using the tool, developers are able, by interacting with the marker in the gutter, or in the system view, to vote up if they think that method does actually contain a defect, or vote down, if they believe that the method does not contain a defect. The developer can also get rid of warnings they do not want to see again by clicking on an option that will hide the prediction for ever, or they can say that this method does contain a defect, which raises the probability instantly to 100%. All the interaction and votes are kept in a log specific for that particular software project and can be used in future defect prediction as metrics. The voting up and down helps reduce the amount of warnings, and over time, if the developers are good at voting, may improve future predictions.

IV. Evaluation

ELFF is being used by four developers in a large UK telecommunications company. The developers are using ELFF as part of their everyday development. ELFF was introduced to the whole team in an introduction session and volunteers invited to use and evaluate the tool. The four volunteers had ELFF installed on their machines and were given a short training session. We have started the evaluation by emailing weekly questionnaires to the four volunteers. These questionnaires are very simple and include only three multiple choice questions. The questions are:

1) Have you used the ELFF tool this week? - Every day, A few times, Never
2) Has it found any defective methods? - Lots of them, A few, Just one, None
3) How usable is the ELFF tool? - Not at all, Enough, A lot.

Overall the developers have welcomed the information provided by ELFF. They particularly liked having data about the features of code predicted to be defective, i.e. the metric values for that code. The main feedback from developers was that they wanted more information on what to do about the code predicted to be defective. In response to this we are now extending the tool to provide suggestions on how to fix the code that is being predicted defective.

V. Conclusion

In this paper we have presented our defect prediction tool - ELFF. We believe that our tool helps solve some of the problems that have limited the impact of software defect prediction in industry. ELFF has been developed alongside our industrial partner, who are using the tool to predict defects on their systems. We have plans to improve this tool further by implementing new features which are desired by industry, such as fix suggestions and show instances where similar code has been fixed in the past.

Acknowledgements

This work was partly funded by a grant from the UK's Engineering and Physical Sciences Research Council under grant number: EP/L011751/1.

References

[1] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE software*, 25(5), 2008.

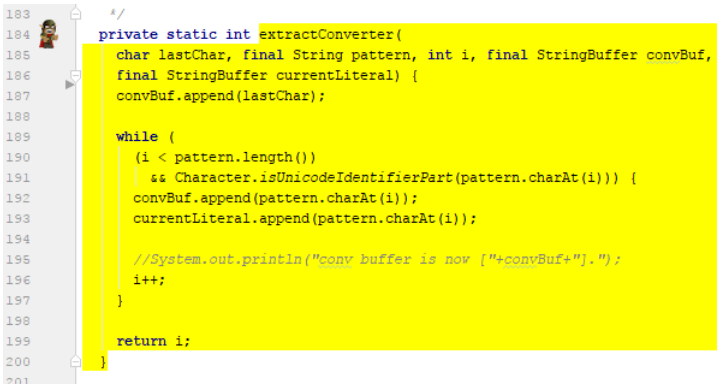

Fig. 1. Preview of the code turning yellow due to the high probability of that method being defective.

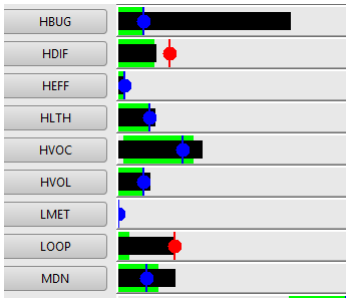

Fig. 2. Visualisation of the metrics of a method

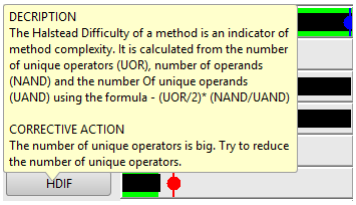

Fig. 3. Example of the help text when a developer will hover over a metric in the metric view

[2] B. Caglayan, A. T. Misirli, G. Calikli, A. Bener, T. Aytac, and B. Turhan. Dione: An integrated measurement and defect prediction solution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 20:1–20:2, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. doi: 10.1145/2393596.2393619. URL http://doi.acm.org/10.1145/2393596.2393619.

[3] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6):1276–1304, 2012. ISSN 0098-5589.

[4] J. Hryszko and L. Madeyski. *Assessment of the Software Defect Prediction Cost Effectiveness in an Industrial Project*, pages 77–90. Springer International Publishing, Cham, 2017. ISBN 978-3-319-43606-7. doi: 10.1007/978-3-319-43606-7_6. URL http://dx.doi.org/10.1007/978-3-319-43606-7_6.

[5] Y. Kamei and E. Shihab. Defect prediction: Accomplishments and future challenges. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 33–45, March 2016. doi: 10.1109/SANER.2016.56.

[6] M. Lanza, A. Mocci, and L. Ponzanelli. The tragedy of defect prediction, prince of empirical software engineering research. *IEEE Software*, 33(6):102–105, Nov 2016. ISSN 0740-7459. doi: 10.1109/MS.2016.156.

[7] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr. Does bug prediction support human developers? findings from a google case study. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 372–381, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486838.

[8] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001. ISBN 0072496681.

[9] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ISSRE '04, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2215-7. doi: 10.1109/ISSRE.2004.1. URL http://dx.doi.org/10.1109/ISSRE.2004.1.

[10] T. Shippey, T. Hall, S. Counsell, and D. Bowes. So you need more method level datasets for your software defect prediction?: Voilà! In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2016, Ciudad Real, Spain, September 8-9, 2016*, pages 12:1–12:6, 2016. doi: 10.1145/2961111.2962620. URL http://doi.acm.org/10.1145/2961111.2962620.

[11] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005. ISSN 0163-5948.