FOUNDATIONS OF COMPUTING AND DECISION SCIENCES
Vol. 40 (2015) No. 1
DE GRUYTER OPEN
DOI: 10.1515/fcds-2015-0002
ISSN 0867-6356
e-ISSN 2300-3405

BOTTLENECKS IN SOFTWARE DEFECT PREDICTION IMPLEMENTATION IN INDUSTRIAL PROJECTS

Jarosław HRYSZKO *, Lech MADEYSKI †

Abstract.
Case studies focused on software defect prediction in real, industrial software development projects are extremely rare. We report on dedicated R&D project established in cooperation between Wroclaw University of Technology and one of the leading automotive software development companies to research possibilities of introduction of software defect prediction using an open source, extensible software measurement and defect prediction framework called DePress (Defect Prediction in Software Systems) the authors are involved in. In the first stage of the R&D project, we verified what kind of problems can be encountered. This work summarizes results of that phase.

Keywords:
software defect prediction, industrial application, depress framework

1 Introduction

Software defect prediction is a quality assurance technique in software engineering, where sophisticated methods (including machine learning) are used to predict future defects in computer programs. Such information can be used to support optimal efforts and resources allocation in the software development projects (e.g. to focus quality assurance activities on software classes which are predicted to be defect-prone).

*Wroclaw University of Technology, Faculty of Computer Science and Management, jaroslaw.hryszko@pwr.edu.pl
†Wroclaw University of Technology, Faculty of Computer Science and Management, lech.madeyski@pwr.edu.pl

1.1 State of the art

Despite the fact, that enormous amount of scientific work has been done on defect prediction in software engineering, industrial case studies reporting introduction, obstacles or benefits of defect prediction in industrial environments are quite rare. Since now, reviews of the state of the art in defect prediction research studies were published in 1999 by Fenton and Neil [2], in 2009 by Catal and Diri [1] and – in the most comprehensive form – in 2012 by Hall et al. [3]. Those works prove, that only few publicized research investigated real world, industrial application of defect prediction. First publication, related to industrial application of defect prediction was published in 1997 by Khoshgoftaar et al. [5]. It was a case study on quality modeling of a very large telecommunications system. Authors of that work used neural network to model future fault proneness of real-world system managed by telecommunication company, but finally results were not used to support company's quality assurance procedures. Two other publications of Khoshgoftaar and Seliya from 2004 [6] and 2005 [7] continue the previous concept and focus on commercial data analysis, not on a real-world application. Similar approach is common for most research utilizing industrial data - examples can be found in different publications by Ostrand [13], [15], Tosun [19] and Turhan [21], [22]. Example of industrial application of knowledge gathered by using defect prediction can be found in few publications, such as [23], and [18]. Complete cases describing introduction of defect prediction in industrial environments were presented by Ostrand [14], Li et al. [9] and Tosun [20]. Five industrial software development projects were investigated in order to build software defect prediction models by Jureczko and Madeyski [4]. The data sets of those projects, as well as a number of open source projects investigated by the authors, were made available online at available online at: `http://purl.org/MarianJureczko/MetricsRepo/` as well as in the PROMISE repository. Recent thorough investigation of both industrial and open source projects with respect to defect prediction including process metrics was described by Madeyski and Jureczko [10].

2 Objective and case description

As it was described in the previous paragraph, reports on commercial applications of defect prediction techniques in software development projects are still quite rare. Therefore, in this paper we describe a real-world endeavor of researchers from Wroclaw University of Technology who started long-term research project on introducing defect prediction techniques to real-life industrial software development projects led by the IT company, part of the globally recognized automotive group.

Plan of the the aforementioned research project assumed dividing research activities into a number of specified stages which were, to a large extent, required by the IT company project management methodology. Initial phase was divided as follows:

1. Starting cooperation with the IT company – receiving proposal, which software development projects can be used in investigation;

2. Review set of proposed projects, gather knowledge about the projects;
3. Formulate proper approach for introducing defect prediction in software development projects;
4. Recognize bottlenecks (weak points) in the projects, which can influence negatively defect prediction;
5. Propose improvements, which can be applied in the investigated software development projects.

This paper focuses on the two last stages and answering to research question: *What are main problems related to introducing software defect prediction in existing, mature commercial software development projects?*

2.1 Gathering knowledge regarding projects

Initial step of our research was to gather information describing five software development projects proposed by the company encoded as *Project B, Project P, Project T, Project V* and *Project M* as candidates for defect prediction introduction. Projects were characterized basing on nine questions, presented in Table 1.

Table 1: Assessment questions for researched projects

1. Have defects (Bug Reports – BRs) been registered in some specific tool(s)?
2. Since when BRs have been registered?
3. How the documentation of BRs looks like in project?
4. What is the estimated end date of current project?
5. How is the code coverage by unit tests in the application?
6. Is there any plan to add unit tests in the old part of the application?
7. Is there any plan to add unit tests in the new part of the application?
8. Is there test plan and test cases package created for the application?
9. How many people are currently working on the application and in what roles?

Characteristics of selected projects are collected in Appendix A.

2.2 Considered approach

In the investigated case two variations of the approach to introduce defect prediction were considered. In first, information describing first version of the application,

marked as *n*, i.e. code metrics and defects registered in defect tracker during the testing campaign were used for training purposes. Version *n+1* was used for defect prediction. Registered defects found in version *n+1*, were used for prediction precision validation and – again – for training data creation to be used for prediction in version *n+2*, and the cycle continued. In the second approach information describing first version of the application was used for training, but defect prediction based on mathematical model created in such way w applied to every software version released during development cycle.

The first approach is more precise but also more expensive. It can have various variants (e.g. ensemble learning approach can be used to cope with the so called concept drift). The second approach is cheaper but less precise.

For both alternatives, decision was made to use DePress framework proposed by Majchrzak and Madeyski from Wrocław University of Technology [11, 12]. DePress (*Defect Prediction for Software Systems*) allows for creating graphical workflows, using intuitive, user-friendly interface. Its possible applications varies from software measurement, product and process improvement, quality assurance processes, integrating data from various resources and tools for the sake of data analyses — including software defect prediction. Using standard data formats provided by popular applications used for software creation, as well as being intuitive and highly customizable, DePress makes itself a perfect tool to be utilized in different commercial software development projects for defect prediction introduction.

DePress workflow, planned to be used in the described research project, is presented in Figure 1. SVN (or GIT – to support TFS-based code repositories) plugin was used to collect information about changes in a source code repository in XML format, generated using SVN client. Second input file was generated by defect tracking software (in the analyzed software development projects JIRA and ClearQuest were used as bug and issue trackers) and used by JIRA or ClearQuest input DePress plugin to gather defect information. Third, input file was used to introduce information regarding project's code metrics, collected using additional tools, like Eclipse Metrics plugin for Eclipse Integrated Development Environment or Visual Studio Code Metrics Power Tool (more information can be found in [16]). In our workflow presented in Figure 1, code metrics information for particular software version (i.e. *n*) was combined with defect information to obtain training data or validation data. Combining information from different sources was planned to be conducted in two steps – in first, defect information was combined with list of code modules (classes, methods) modified due to defects. It is possible in most cases when developer add unique defect identifier to comment sent when submitting code to the repository. In the second step, metrics describing selected code modules are added to the final portion of data, used later for training or validation. Later, during next phases of the research, feature selection was planned to be introduced to select most effective set of metrics. Another direction of further research is to use other features not related to the source code metrics.

Conditions which have to be met in order to just perform the defect prediction (conditions 1, 3 and 4) using described approach, as well as to assure high quality of resulting predictions (condition 2 and from 5 to 8) have been listed in Table 2. As is

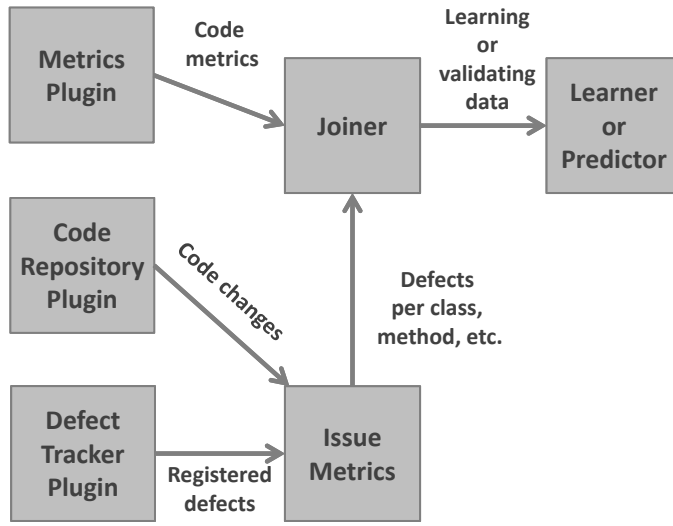

Figure 1: A simplified DePress workflow

Table 2: Requirements for performing defect prediction in selected projects, using specified workflow

1.Set of files containing defect information and code metrics has to be available;
2.Unified method for marking the software version in code repository and defect trackers has to be present;
3.Each defect report has to be registered under one, single entry in defect tracker software;
4.Each unique defect report has to be registered under unique identifier;
5.Information explaining in which version each defect was found should be available;
6.Defect fix has to be sent into the repository within single commit;
7.Defect fix has to be sent into the repository with related defect's identifier;
8.No other changes than defect fix cannot be sent within one commit into code repository;

apparent from the Table 2, set of requirements was created, regarding code repository and defect tracker contents. Next step of the research was to verify, if the mentioned requirements were fulfilled in analyzed projects, addressing the research question.

3 Results

Software development projects selected for research were verified if they fulfill requirements listed in Table 2. Observations gathered during that investigation can be divided into two areas: code repository related area, and defect tracker area.

3.1 Defect tracker area

From the requirements gathered in Table 2, points 1-5 can be applied to defect tracker area of observations. *Requirement 1* was fulfilled in all investigated software development projects. In most cases, getting files containing defect information was easy and was only limited to running proper "Export" function within the defect tracker. However, during the research, few minor datafile-related issues were met, with no impact on the future defect prediction process. First problem was strictly related to JIRA defect tracking software used in some of the analyzed software development projects. Global JIRA settings for all related projects were fixed and allowed for extraction only 1000 records in single file. This small issue was impacting total time for preparing input data for defect prediction in projects with few thousands of defects registered. To prepare input data, set of defects had to be divided manually into parts containing maximum 1000 defects each. Parts in such form can be exported to file, but after that operation, all the files have to be merged back to one defects set. That operation, although it is easy, it is also time consuming and impacts time frame of the defect prediction operation.

Similar issue was also observed in projects where different software was using for defect tracking. In that case, to prepare final data set, data from different defect trackers had to be extracted, often – to more than one file, then transformed using DePress into one, common format and merged. Extreme example of such situation was found in *Project M*, where huge part of the defects was stored as simple e-mail communication between testers and developers. Each communication had to be stored in single file, then loaded into DePress framework, parsed, re-formatted and concatenated into single data set.

Using unified naming in both – defect tracker and code repository is important: if this requirement, listed in Table 2 under point 2, is not fulfilled, it is almost impossible to connect data describing code from proper application version (release) with defect information – please notice that in regular software life cycle in considered company bug report can be registered anytime for at least three different versions of application: for example, when found by end-user, defect report will concern version n of the software (if we assume that only one single version is used by end-users, what is highly unlikely); at the same time version *n+1* is being tested by testers and version *n+2* is being written by software developers – also for these two versions bugs can be reported. Such case is analogical to situation when multiple releases are tested in parallel. For four projects, charts showing the most frequent version numbers present in defect trackers are presented in Appendix B. *Project M* was not included, as due to early phase of development only 1.0.0.0. version was applied to all changes and

fixes, thus there is no other version number assigned to reported defects. As shown on the charts attached, version naming is not consistent in some cases (see Figure 5 and Figure 6). The problem is even more serious when we oppose names found in defect trackers to naming present in code repository. In *Project P* and *Project V* defects were registered using different version naming (numbering) approach than adopted in code repository. As a result of such situation, defect information cannot be automatically combined with code-related data, such as metrics or changes log. Manual combination will be time consuming and will impact time frame of the process. On the other hand,

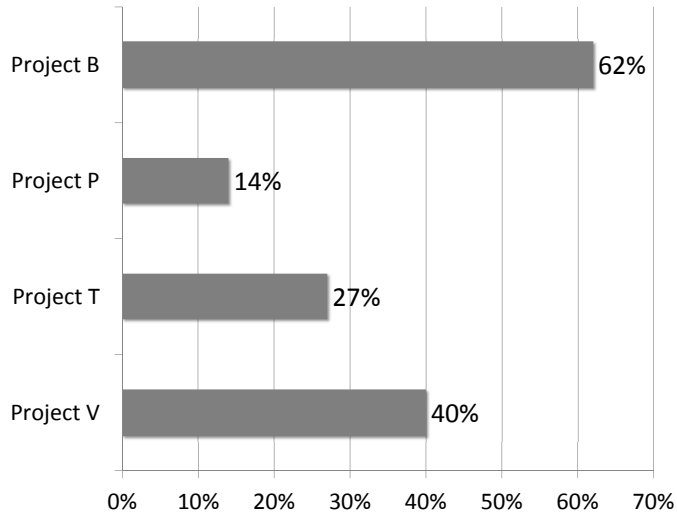

Figure 2: Percentage share of defect reports without version information

the remarkable consistency was demonstrated regarding *Requirement 3*. In every investigated project, every observed software defect was tracked using single entry in currently used tool for defect tracking. Such approach caused situation, when also *Requirement 4* was fulfilled – when using of specialized software for defect tracking, like JIRA, HP Quality Center or IBM ClearQuest, an unique identifier is assigned to each new entry.

Requirement 5 is strictly connected with the numbering approach issue. It is easy to notice on charts provided in Appendix B that for significant part of total defects registered, there is no information telling in which version (release) of software particular defect was detected. Percentage figures describing share of this kind of defects are gathered in Figure 2. In the case of the proposed solutions of software defect prediction described in paragraph 2.2, such entries without version information still can be used for defect prediction, but it should be expected that prediction quality will decrease – for example, if considered defects were found in code which was later refactored, it will cause the situation when code metrics describing refactored code will be marked as defective in data set used for learning.

3.2 Code repository area

Requirements 1, 2, 6, 7 and *8* from Table 2, belong to code repository area of observations. As it was mentioned above, *Requirement 1* was fulfilled in all investigated software development projects. Problems caused by different naming conventions used for defect trackers and code repositories were described in previous subsection.

It was observed that when best practices for version control are not mandatory in software development projects, *Requirements 6, 7* and *8* are not fulfilled. Instead of sending to the code repository each code change separately with the proper comment describing the change and using single commit (see Figure 3), it is common practice to send different changes using one commit. Such approach may affect reliability of the input data for defect prediction purposes with varying degrees, depending on case:

- Software developer sends defect's fix into the repository separately, as single "code package" but did not add proper comment, containing unique defect identifier;
- Software developer sends defect's fix into the repository separately, attaching unique defect identifier, but "package" also contains other small changes, not related to the defect, like local code refactoring ("hidden refactoring");
- Developer sends defect's fix as well as small additional changes, but did not attach defect identifier while sending code to the repository;
- Developer sends different code changes, related to different tasks (defect fixes, new functionalities, code refactoring) in a single package, attaching general comment or no comment at all (Figure 4).

Last mentioned practice makes data describing changes in code repository useless for chosen approach of defect prediction in investigated projects.

In four from five total investigated software development projects, regarding legacy systems with many releases over a long time period, best practices for version control are maintained, what results with high quality data describing code changes due to defect fixes. Exception from that rule is small and relatively new *Project M*, where significant amount of "commits", especially from initial project phase, consists of many changes of different kind, namely new functionalities, refactorization and fixes were introduced at the same time. Great amount of commits contains several dozen changed files with different adjustments, moreover those amendments are not defined in comments to commits. Comments describing commits in *project's M* repository are too vague, moreover most of them characterize main functionality only. Amendments introduced together with those functionality are often not mentioned at all. Even in cases where adjustments and fixes are accompanied with comments, they tend to be very ambiguous, e.g. "Few fixes in bindings", "Fixes in Web Interface". None of the comments consists of reference to an e-mail in which defect was reported, hence there is no possibility to indicate which defects were fixed in a given package. At the moment, thorough analysis of reported defect's date, commits' date and changes in source code is the only possibility to merge changes with reported defects.

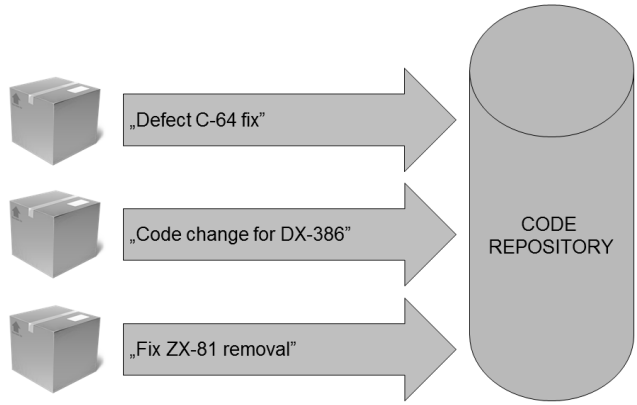

Figure 3: Best practice for sending changes to the code repository

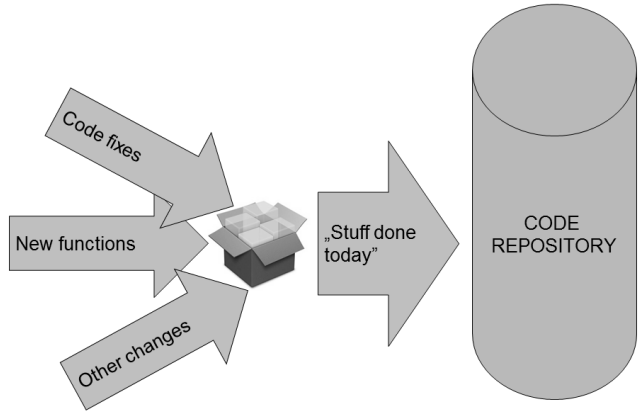

Figure 4: Improper practice of sending code to the repository

4 Improvement propositions

Similar to results of the investigation, improvement propositions can relate to specified area (defect tracker or code repository) or be applied to the whole project – like improving work culture within the projects, by persuading team members to maintain best software development practices and self discipline in that matter. Such approach, as well as clearly visible version number in application, defect tracker and code repository, will result in keeping unified version marking convention in each project.

4.1 Defect tracker area

In the investigated projects, professional defect tracking software, like JIRA or Microsoft Team Foundation Server provide unique identifier for each registered defect. Team members should register every defect found as single entry in defect tracker. Such practice will secure that every defect registered will receive its own unique identifier assigned by tracker. Identifier should be used later, during defect fixing. Reviewing defects stored in tracker to avoid duplication of defects also should be introduced as common practice.

To complete the missing version information from bug reports (as it was shown in Figure 2), additional information should be collected, describing relation between version number and time period when particular version was installed in test environment (when defect was found by tester) or production environment (when defect was found by end user). Such information can be used by proper DePress workflow to complete missing version information in data acquired from defect tracker.

4.2 Code repository area

No other changes should be send to the code repository with the defect fix – project's work culture should ensure that code is send to repository after every change, separately, together with appropriate description (Figure 3) – if it is defect fix, it should contain unique identifier assigned by defect tracker.

To follow the considered approach, instead of using lacking defect identifiers in code repository data to combine code changes with particular defects, DePress workflow can be modified by adding special nodes providing semantic or syntactic analysis of repository log entry, to link changes with bug reports stored in defect tracker. Syntactic analysis looks for pre-defined patterns in commit comments to recognize bug fixes, semantic analysis uses time relation basing on assumption, that bug report's status is changed to "fixed" after sending appropriate code to the repository. Those methods of finding change-bug relation was used with good results and then described by Śliwierski, Zimmermann and Zeller in [17].

5 Conclusions

Authors wish that observations gathered in this work as well as improvement propositions will support software development professionals who want to take advantage of defect prediction and introduce this technique in different software companies. We convince that proper work culture is crucial in software development projects and have to be maintained from the beginning. Even small derogation from best practices – like resignation from inserting defect's unique identifiers to code commits, harmless at the moment, can result that any innovative technology or technique when introduced in the future, as defect prediction in this case, will meet significant problems or at least will suffer from bad data quality. With the growth and maturation of de-

veloped applications, small discrepancies or negligences can turn into serious quality or project-related knowledge issues – like presented in this work 62% lack of version information in registered defects (Figure 5).

Important role in collecting information analyzed in described research played DePress Framework and set of its interfaces created for variety of software development tools. Highly customizable DePress allowed to combine data from various sources, stored in various formats, like different defect tracker tools used in projects (see Table 4 for example). Thus, if requirements for introducing defect prediction cannot be maintained in the project from the beginning, DePress can support particular projects in recovering lost information using its abilities to utilize and combine different data sources, as well as data mining mechanisms and semantic or syntactic analysis.

A Characteristics of the projects involved

Characteristics of software development projects which were selected for defect prediction introduction. Information gathered in tables were received from project's representatives. BR stands for Bug Report – a single entry in defect tracking software describing one software defect found in developed (maintained) application.

Table 3: Characteristics of the project V

Defect tracker used	IBM Clear Quest, JIRA
Since when BRs have been registered?	Since 2005
Documentation of BRs	Documents and Wiki stored on local server
Estimated end date	Continuous maintenance
Code coverage by unit tests	10% of LOC
Unit tests in the old part of the application	Yes (Refactoring)
Unit tests in the new part of the application	Yes
Test plan and test cases package	Yes
Size of the project team	8 developers, 3 testers

Table 4: Characteristics of the project P

Defect tracker used	IBM ClearQuest, JIRA, HP Quality Center
Since when BRs have been registered?	Since 1.09.2008
Documentation of BRs	BRs are described in defect tracker according to Test Manager specification
Estimated end date	Continuous maintenance
Code coverage by unit tests	5% - 74% 10% of LOC (depends on module)
Unit tests in the old part of the application	Yes (Refactoring)
Unit tests in the new part of the application	Yes
Test plan and test cases package	Yes
Size of the project team	7 developers, 7 testers

Table 5: Characteristics of the project T

Defect tracker used	JIRA
Since when BRs have been registered?	Since 2009
Documentation of BRs	JIRA
Estimated end date	Continuous maintenance
Code coverage by unit tests	3% of LOC
Unit tests in the old part of the application	No
Unit tests in the new part of the application	Yes
Test plan and test cases package	Yes
Size of the project team	3 developers, 1 tester

Table 6: Characteristics of the project B

Defect tracker used	JIRA
Since when BRs have been registered?	Since 18.10.2011
Documentation of BRs	JIRA
Estimated end date	Continuous maintenance
Code coverage by unit tests	17% of LOC
Unit tests in the old part of the application	No
Unit tests in the new part of the application	Yes
Test plan and test cases package	Yes
Size of the project team	3 developers, 2 testers

Table 7: Characteristics of the project M

Defect tracker used	Microsoft Team Foundation Server, E-mails
Since when BRs have been registered?	Since April 2014
Documentation of BRs	Microsoft Team Foundation Server, E-mails
Estimated end date	2016
Code coverage by unit tests	No unit tests
Unit tests in the old part of the application	No
Unit tests in the new part of the application	No
Test plan and test cases package	No
Size of the project team	2 developers

B Version numbers present in defect trackers

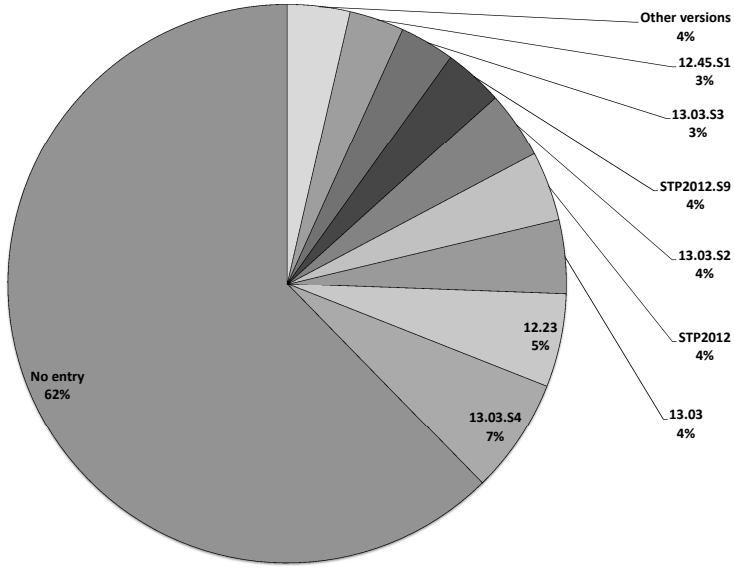

Figure 5: Defects registered for different version numbers in Project B

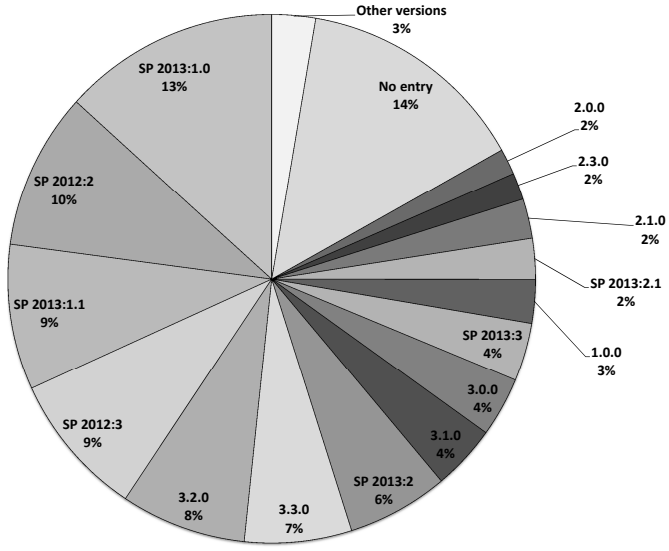

Figure 6: Defects registered for different version numbers in Project P

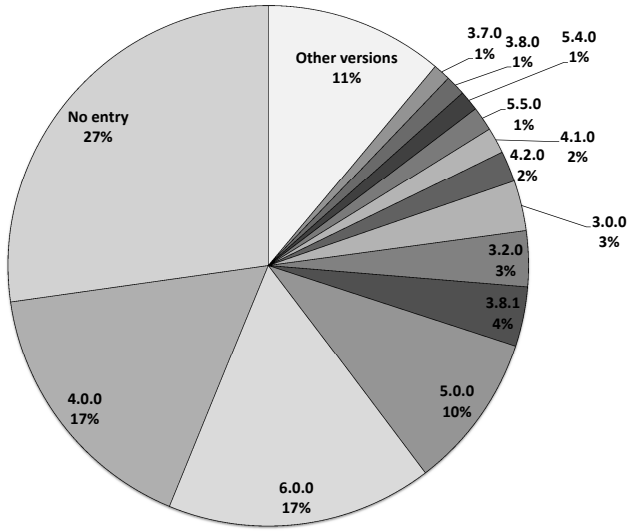

Figure 7: Defects registered for different version numbers in Project T

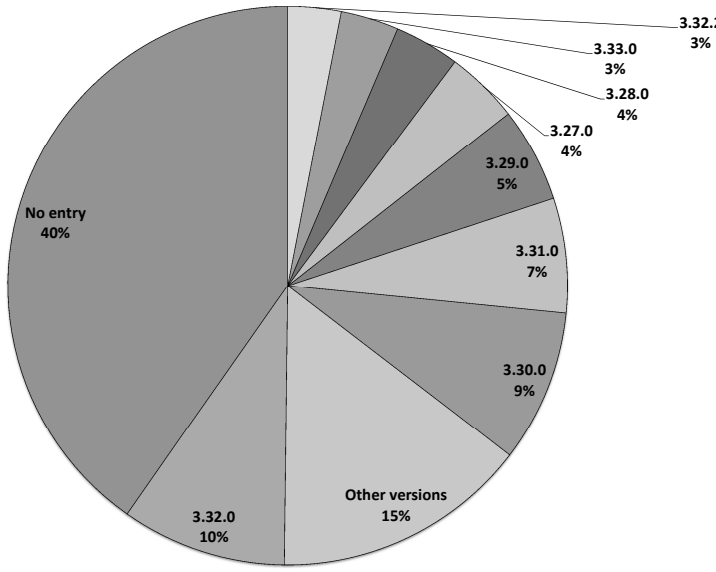

Figure 8: Defects registered for different version numbers in Project V

References

[1] Catal, C. and Diri, B. (2009). A systematic review of software fault prediction studies. *Expert Systems with Application*, 36:7346–7354.

[2] Fenton, N. and Neil, M. (1999). A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25:675–689.

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

[4] Jureczko, M. and Madeyski, L. (2010). Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 9:1–9:10, New York, NY, USA. ACM.

[5] Khoshgoftaar, T., Allen, E., Hudepohl, J., and Aud, S. (1997). Application of neural networks to software quality modelling of a very large telecommunications system. *IEEE Transactions on Neural Networks*, 8:902–909.

[6] Khoshgoftaar, T. and Seliya, N. (2004). Comparative assessment of software quality classification techniques: An empirical case study. *Empirical Software Engineering*, 9:229–257.

[7] Khoshgoftaar, T. and Seliya, N. (2005). Assessment of a new three-group software quality classification technique: An empirical case study. *Empirical Software Engineering*, 10:183–218.

[8] Klas, M., Nakao, H., Elberzhager, F., and Munch, J. (2008). Predicting defect content and quality assurance effectiveness by combining expert judgment and defect data-a case study. *Proceedings of the 19th International Symposium on Software Reliability Engineering*, pages 17–26.

[9] Li, P., Herbsleb, J., Shaw, M., and Robinson, B. (2006). Experiences and results from initiating field defect prediction and product test prioritization efforts at abb inc. *Proceedings of the 28th International Conference on Software Engineering*, pages 413–422.

[10] Madeyski, L. and Jureczko, M. (2014). Which Process Metrics Can Significantly Improve Defect Prediction Models? An Empirical Study. *Software Quality Journal*.

[11] Madeyski, L. and Majchrzak, M. (2012). ImpressiveCode DePress (Defect Prediction for software systems) Extensible Framework. Available as an open source project from GitHub: https://github.com/ImpressiveCode/ic-depress.

[12] Madeyski, L. and Majchrzak, M. (2014). Software Measurement and Defect Prediction with DePress Extensible Framework. *Foundations and Computing and Decision Sciences (accepted)*.

[13] Ostrand, T. and Weyuker, E. (2002). The distribution of faults in a large industrial software system. *SIGSOFT Software Engineering Notes*, 27:55–64.

[14] Ostrand, T., Weyuker, E., and Bell, R. (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31:340–355.

[15] Ostrand, T., Weyuker, E., and Bell, R. (2010). Programmer-based fault prediction. *Proceedings of the Sixth International Conference on Predictive Models in Software Engineering*, pages 1–10.

[16] Rudiger, L., Lundberg, J., and Lowe, W. (2008). Comparing software metrics tools. *Proceedings of The 2008 International Symposium on Software Testing and Analysis*, pages 131–142.

[17] Sliwierski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? *Proceedings of The 2005 International Workshop on Mining Software Repositories*.

[18] Succi, G., Pedrycz, W., Stefanovic, M., and Miller, J. (2003). Practical assessment of the models for identification of defect-prone classes in object-oriented commercial systems using design metrics. *Journal of Systems and Software*, 65:1–12.

[19] Tosun, A., Bener, B., Turhan, B., and Menzies, T. (2010). Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry. *Information and Software Technology*, 52:1242–1257.

[20] Tosun, A., Turhan, B., and Bener, A. (2009). Practical considerations in deploying ai for defect prediction: A case study within the turkish telecommunication industry. *Proceedings of the Fifth International Conference on Predictor Models in Software Engineering*, page 11.

[21] Turhan, B., Kocak, G., and Bener, A. (2009a). Data mining source code for locating software bugs: A case study in telecommunication industry. *Expert Systems with Applications*, 36:9986–9990.

[22] Turhan, B., Menzies, T., Bener, A., and Di Stefano, J. (2009b). On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14:540–578.

[23] Wong, W., Horgan, J., Syring, M., Zage, W., and Zage, D. (2000). Applying design metrics to predict fault-proneness: A case study on a large-scale software system. *Software: Practice and Experience*, 30:1587–1608.

Presented at the 16th KKIO Software Engineering Conference, 22-24 September 2014, Poznan, Poland