Assessment of the Software Defect Prediction Cost Effectiveness in an Industrial Project

Jaroslaw Hryszko[1,2] and Lech Madeyski (✉)[1]

[1] Wroclaw University of Science and Technology, Faculty of Computer Science and Management, Wyb.Wyspianskiego 27, 50-370 Wroclaw, POLAND,
Lech.Madeyski@pwr.edu.pl
[2] Volvo Group

Abstract. Software defect prediction is a promising, new approach to increase both, software quality and development pace. Unfortunately, the cost effectiveness of software defect prediction in industrial settings is not eagerly shared by the pioneering companies. In particular, the cost effectiveness of using the DePress open source software measurement framework, developed by Wroclaw University of Science and Technology, and Capgemini software development company, for defect prediction in commercial software development projects have not been previously investigated. Thus, in this paper, we explore whether defect prediction can positively impact an industrial software development project by generating profits. To meet this goal, we conducted a defect prediction and simulated potential quality assurance costs based on the best prediction result, as well as the proposed Quality Assurance (QA) strategy. Results of our investigation were optimistic: we estimated that quality assurance costs can be reduced by almost 30% when proposed approach will be used, while estimated DePress tool usage Return on Investment (ROI) is fully 73 (7300%), and Benefits Cost Ratio (BCR) is 74. Such promising results have caused the acceptance of continued usage of the DePress-based software defect prediction for actual industrial projects run by Volvo Group.

1 Introduction

Until recently, software defect prediction process has been considered to be too complex, expensive and time-consuming, as well as there have been lack of solutions for wrapping required tools into one, universal, defect prediction framework which could be used for different software projects.

To fill this gap, Madeyski and Majchrzak [16] proposed a new, extensible (plugin-based) framework called DePress. DePress (*Defect Prediction for Software Systems*) builds upon the KNIME framework [13] and allows development of graphical workflows and uses an intuitive, user-friendly interface. Being intuitive and highly customizable, the DePress makes itself a perfect tool which can be conveniently utilized (thanks to its user-friendly interface and a wide range of plugins) in different commercial software development projects for software defect prediction. Detailed description of DePress framework and its capabilities can be found on DePress website [17] or in the article by Madeyski and Majchrzak [16].

Potential benefits of using the DePress framework for defect prediction in commercial software development projects have not been investigated [16]. To fill this gap, our study aimed to answer the following research questions:

RQ1: *What is the highest level of defect prediction, measured by F-measure, achievable by the DePress tool (using a default, non-tweaked configuration) in an industrial software project?*

The possible benefit varies, depending on the potential prediction effectiveness. This implies the need of first verifying what is the highest F-measure (harmonic mean of precision and recall [26]) value of the defect prediction process handled entirely by the DePress tool. DePress can be highly customizable thanks to its plugin-based architecture, as well as its open source nature. However, such adjustments can generate additional costs. Therefore, for the sake of simplicity, we decided to restrict the DePress usage only to its default set-up.

RQ2: *How cost effective is defect prediction using the DePress framework, in the default configuration, for defect prediction in an industrial software development project?*

The next step is to verify what will be the profit from the best prediction achievable using the default DePress' set-up. To achieve this, we used value of the *recall* measure corresponding to the highest F-measure value.

RQ3: *Will usage of the DePress framework pay off for an industrial project?*

To answer this question, we had to compare the costs of introducing and using the DePress based defect prediction to the potential benefits generated by its introduction. To achieve this, we used values such as return on investment (ROI) and benefit-cost ratio (BCR) [21].

1.1 Project Context and Target Software

Volvo Group, one of the leading automotive companies, was invited to take part in this research. The primary motivation for Volvo Group's interest was to verify, if their company can use DePress and its software defect prediction to increase quality and cost-effectiveness of quality assurance (QA) in their software development projects.

During our previous research, we recognized elements occurring in software projects that hindered or prevented completion of defect prediction [4]. A project selected finally as a research subject – an initiative which develops and maintains an application called Texas – was chosen due to absence of aforementioned elements.

Within the considered project, we can observe three stages of the software life-cycle (project phases): development, testing, and post-release phase.

1.2 Related Work

The first publication related to an industrial application of defect prediction was published in 1997 by Khoshgoftaar et al. [8]. It was a case study of quality modeling for a very large telecommunications system. Two other publications of Khoshgoftaar and Seliya from 2004 [10] and 2005 [11] continued with the previous concept and focused on commercial data analysis, but were not applied to a real-world environment. A similar approach can be found in publications by Ostrand and Weyuker [22], Ostrand et al. [24], Tosun et al. [32], Turhan et al. [34,35]. Examples of industrial applications of information gathered by using defect prediction can be found in publications by Wong et al. [37], Succi et al. [31] and Kläs et al. [12]. Complete cases describing the introduction of defect prediction in industrial environments were presented by Ostrand et al. [23], Li et al. [14] and Tosun et al. [33]. Unfortunately, none of the aforementioned works contain information on cost effectiveness of applied prediction techniques and tools. To the best of our knowledge, the only research focused on the cost effectiveness of software defect prediction in an industrial project, is conducted by Monden et al. [19]. However, they investigated cost effectiveness only from the acceptance testing effort perspective and do not use any quantitative measure of potential cost of quality assurance-focused work and cost of investment during the entire software life-cycle period. Thus, in our research we also followed approaches used when cost effectiveness of other than defect prediction quality assurance technique was investigated, such as Test-Driven Development return on investment research conducted by Müller and Padberg [21].

2 Assessment Method

To investigate the cost effectiveness of defect prediction applied to an industrial software development project using the DePress framework, we developed the following plan to follow:

1. Development of a QA effort allocation strategy, based on defect prediction provided by DePress;
2. Analysis of actual, real-life costs of quality assurance for the selected release of the Texas project (4.0.0);
3. Building software prediction models for the chosen release;
4. Selection of the highest prediction *F-measure* and the corresponding *recall* measure;
5. Usage of an effort allocation strategy, based on the prediction effectiveness characterized by *recall*, to simulate a prediction-based quality assurance scenario;
6. Results analysis.

2.1 Quality Assurance Effort Allocation Strategy

In the case investigated (the release 4.0.0 of the Texas software), developers agreed that all the modules that caused 2 and more registered defects are considered as “high risk” modules. These modules accounted for 22.4% of all modules and were responsible for 80.36% of all registered errors and the aim was to eliminate the maximum number of software defects using the available resources within a limited time period. A similar distribution of defects in the software modules was observed by different authors [2,25,27] and can be interpreted as the Pareto principle existence in software quality. Additionally, in 1976 Boehm argued that defect fixing costs are the more expensive the later defects are removed [1]. That observation, which is widely called Boehm’s Law [2], results in another important consequence of smart quality assurance efforts allocation: the earlier the QA actions will take place, the better it is from the perspective of the software development project’s budget.

Considering the above facts, we proposed a strategy which would use the prediction model to indicate as much as possible of the mentioned “high risk” software modules (22.4% in our case) responsible for most of the defects (80.36% in our case), therefore helping to integrate as much as possible the QA efforts into the coding stage of the software development, while defect fixing cost is still relatively low. Such an approach should ideally decrease the total cost of bug fixing in the project and generate savings for the total project’s budget [29].

If we denote M_{total} as the total number of testable software modules and H_{total} as the total number of discoverable defects, we can say that, in the project we analyzed, approximately $0.8H_{total}$ comes from approximately $0.22M_{total}$.

The impact of the prediction effectiveness on the overall effort allocation strategy can be reflected by using the *recall* measure (Rec) – the proportion of code units predicted as defective that were actually defective [36].

We can expect that:

$$0 < Rec < 1 \tag{1}$$

Where Rec is the measured *recall* value corresponding to highest possible *F-measure* of defect prediction performed using the DePress framework [16]. Then, expected number of predicted modules M_i, responsible for 80% of discoverable defects, should be:

$$M_i = 0.22 \times Rec \times M_{total} \tag{2}$$

Accordingly, we should expect that if the machine learning mechanism will be able to point out the “high risk” 22% of software modules with the measured *recall* (Rec), the number of defects which can be avoided by allocation of the best quality assurance efforts on the first (development) project’s phase, shall be:

$$H'_1 = 0.8 \times Rec \times H_{total} \tag{3}$$

Number of defects expected to be detected in the second and the third phase of the project:

$$H'_{2+3} = H_{total} - H'_1 \tag{4}$$

Return on Investment To investigate if usage of the DePress will pay off, we will use Return on Investment (ROI) [21]:

$$ROI = \frac{Benefit - Investment}{Investment} \tag{5}$$

If the investment will not pay off, ROI is negative, otherwise positive. In our evaluation of defect prediction cost-effectiveness we will focus on potential benefits that method will generate:

$$Benefit = C_{total} - C'_{total} \tag{6}$$

Where C'_{total} is the simulated total quality assurance cost in the project with defect prediction applied, and C_{total} is the actual QA cost in the project, without defect prediction.

Investment is defined as the total cost of defect prediction introduction. Moreover, *NetReturn* is calculated as *Benefit* reduced by *Investment*:

$$NetReturn = C_{total} - (C'_{total} + Investment) = Benefit - Investment \tag{7}$$

Benefit Cost Ratio To analyze potential benefits from the usage of defect prediction, we will use the Benefit Cost Ratio (BCR) [21]:

$$BCR = \frac{Benefit}{Investment} \tag{8}$$

Values larger than 1 for the BCR mean a monetary gain from the DePress based defect prediction usage, while values smaller than 1 mean denote a loss.

2.2 Actual Project's Quality Assurance Costs

The Volvo Group policy did not allow us to publish the real costs of work invested in the project. For the purpose of research, we agreed that the man-hour cost of work by a software developer C_d will be marked as:

$$C_d = x \tag{9}$$

In that case, the average man-hour cost of work by a software tester C_t shall be, calculated according to current labor market data rates [30]:

$$C_t = 0.85x \tag{10}$$

That means, that when a tester and a developer are working together on bug fixing during the later stages of the project (not in the coding phase), the average cost per man-hour should be:

$$C_{d+t} = \frac{C_d + C_t}{2} = 0.925x \tag{11}$$

Other costs, such as infrastructure and hardware, will remain constant for the real-life and alternative (prediction-based) scenario, so they will be omitted.

Time spent on project work was traced by every team member using the JIRA tool. As a result of analysis of that data, we could obtain an average of the total time spent on fixing a single defect for each phase of the project (Table 1). The amount of time spent on quality assurance, together with the number of hours spent and number of defects fixed, divided by phases, are shown in Table 2.

According to the data in Table 2, the total number of defects discovered in release 4.0.0 are:

$$H_{total} = \sum H_{phase} = 190 + 383 + 264 = 837 \tag{12}$$

Accordingly, the total quality assurance cost is:

$$C_{total} = \sum C_{phase} = 190x + 1063x + 733x = 1985x \tag{13}$$

The ratio between defects fixed in testing and those fixed during the post-release stages is:

$$\frac{H_2}{H_3} = \frac{383}{264} \approx \frac{3}{2} \tag{14}$$

Table 1. Average defect fixing costs

Phase	1.Development	2.Testing	3.Post-release
Team members involved	Developer	Developer Tester	Developer Tester
Average fixing time per one defect [hours], T	1	3	3
Assumed cost of man-hour C_{hour}	C_d	C_{d+t}	C_{d+t}
Cost per one defect $C_{defect} = T \times C_{hour}$	x	2.775x	2.775x

Table 2. Actual resources consumed on defect fixing

Phase	1.Development	2.Testing	3.Post-release
Number of defects discovered H_{phase}	190	383	264
QA cost per one defect C_{defect}	x	2.775x	2.775x
QA cost per phase $C_{phase} = H_{phase} \times C_{defect}$	190x	1063x	733x

2.3 Model Construction and Prediction

Since prediction results are categorical (*faulty* or *not-faulty*), we decided to use *F-measure* and *recall* to evaluate classifiers often used in software defect prediction [3,20,9,28], which are available in the basic package of KNIME: Naive Bayes, Probabilistic Neural Network and Decision Tree.

For each classifier, four different experimental setup preparations were possible, thanks to the module-based architecture of the DePress tool.

Table 3. Prediction results: F-measure values for all experimental set-ups

Classifier	Without Feature Selection		With Feature Selection	
	Class Imbalance	Class Balance	Class Imbalance	Class Balance
Probabilistic Neural Network	0.167	0.72	0.24	0.74
Decision Tree	0.279	0.667	0.357	0.682
Naive Bayes	0.237	0.621	0.412	0.766

2.4 The Highest F-measure Value and the Corresponding Recall

Using the approach described in the previous section, defect prediction was performed and its *F-measure* collected (Table 3) for all four experimental set-ups, classifiers and samples. The best prediction results (the highest *F-measure* values) were obtained for the balanced class sample, slightly better with the feature selection step. Hence, we are able to answer **RQ1**: The highest *F-measure* (based on the Naive Bayes algorithm) was 0.766. The corresponding *recall* was:

$$Rec = 0.783 \tag{15}$$

2.5 Prediction-based costs simulation

For the purpose of cost simulation in this scenario, where defect prediction is introduced to the project using the DePress framework, we assumed that:

- The total number of discoverable defects in release 4.0.0 (Equation (12)) is a constant value;
- The defects distribution among code is preserved;
- Average fixing cost per one defect (Table 1) is also true for the considered scenario;
- Information on location of “high risk” software modules, with *recall* *Rec*, will be available in the first phase of the project;

– Ratio (Equation (14)) is preserved.

Considering the *recall* value for best prediction achieved (characterized by the highest *F-measure* value) for release 4.0.0 as a result of the prediction models development (Equation (15)) and the total number of discovered defects in that release (Equation (12)), based on the proposed strategy (Equation (3)) we should expect, that the number of software issues which can be solved by allocation of the best quality assurance practices in the first, development phase of the project is:

$$H_1' = 0.8 \times 0.783 \times 837 = 524 \tag{16}$$

Regarding the number of defects which are expected to be found in later phases of the project (Equation (4)):

$$H_{2+3}' = 837 - 524 = 313 \tag{17}$$

As we assumed that ratio in Equation (14) is preserved, the number of defects which are expected to be found in the project's second and third phase (connected) are:

$$H_2' = 313 \times 0.6 = 188 \tag{18}$$

$$H_3' = 313 \times 0.4 = 125 \tag{19}$$

Considering the above values, we simulated quality assurance costs assuming that the machine learning mechanism will be able to point out the "high risk" 22% of software modules with the measured *recall* (Equation (15)), and the best quality assurance efforts will be allocated to the development phase to avoid the calculated number of defects (Equation (16)). Results of that simulation are presented in Table 4.

Table 4. Simulated QA costs, with defect prediction used

Phase	1.Development	2.Testing	3.Post-release
Number of defects fixed H_{phase}'	524	188	125
QA cost per one defect C_{defect}	x	2.775x	2.775x
QA cost per phase $C_{phase}' = H_{phase}' \times C_{defect}$	524x	522x	347x

Total quality assurance cost in this scenario will be:

$$C_{total}' = \sum C_{phase}' = 524x + 522x + 347x = 1393x \tag{20}$$

Cost of investment Costs of defect prediction introduction were calculated as the sum of such elementary costs:

Tool acquisition and installation costs of the case investigated shall be considered as zero costs. In Volvo's organization, the DePress can be ordered and installed on user's computers without any additional costs for the project.

Training time costs – after measuring time spent on training a single person, we can state that: a developer needs to spend a maximum of 4 hours on training, 1-2 hours of general introduction plus another 1-2 hours of training in DePress tool usage.

Data collection cost is mostly the man-hour cost of exporting the proper data from data sources and code metrics generation for two selected releases. After measuring time spent on that activity, we found that it took no more than 2 man-hours.

Defect prediction preparation cost is the man-hour cost of defect prediction preparation (workflow creation) using the DePress tool. Results of time measurement say that creation of a proper workflow should not take more than one man-hour.

Summary of investment costs is presented in Table 5.

Table 5. Defect Prediction Investment Costs

Activity	Time required [hours]	Cost [man-hours]
The DePress tool acquiring and installation costs	0	0
Training time costs	4	4x
Data collection cost	3	3x
Defect prediction preparation cost	1	x
TOTAL (Investment)	8	8x

2.6 Results Analysis

Here, with respect to research questions **RQ2** and **RQ3** we summarize the results of our simulation (research question **RQ1** was answered in Section 2.4).

RQ2: *How cost effective is defect prediction using the DePress framework, in the default configuration, for defect prediction in an industrial software development project?*

As shown in Table 5, the expected total investment cost of the DePress tool-based defect prediction application in software development project is:

$$Investment = 8x \tag{21}$$

Benefit Cost Ratio (8) calculated using (6) and (21) values:

$$BCR = \frac{592x}{8x} = 74 \tag{22}$$

Such BCR value shows that we should expect a high monetary gain from the DePress tool usage for supporting quality assurance with defect prediction. Moreover, $NetReturn$ (Equation (7)) from the simulated defect prediction application is:

$$NetReturn = 592x - 8x = 584x \tag{23}$$

Answering research question **RQ2**, when the defect prediction application strategy proposed in Section 2.1 is applied and *recall* of the prediction model will be 0.783, such an approach can result in reduction of final QA costs by almost 30%:

$$1 - \frac{C'_{total}}{C_{total}} = 1 - \frac{1393x}{1985x} = 0.298 \tag{24}$$

Such result can be achieved only after fixing 62.6% of the detectable bugs (Equation (16)) by effective use of quality assurance practices in the first, developmental phase of the project, on predicted "high risk" software modules.

RQ3: *Will usage of the DePress framework pay off for an industrial project?*

Simulation shows, that we should expect $Benefit$ (Equation (6)) from the DePress usage in the project:

$$Benefit = 1985x - 1393x = 592x \tag{25}$$

Accordingly, expected Return on Investment (5):

$$ROI = \frac{592x - 8x}{8x} = 73 \tag{26}$$

As ROI is positive, we can state that investment will pay off.

3 Threats to Validity

In this paper, defects are not distinguished according to their severity (minor, major, etc.) and use an average, fixed time for each single defect. Omitting the severity measure in defect prediction studies is a frequent practice [18], however it can be important when simulated QA cost calculation will be compared to real-life values. In our simulation we assumed equal severity for each defect, which is reflected in an equal, average cost (see Table 1). However, when we apply the proposed effort allocation strategy into a real-life environment, we can deal with the situation when defects left undetected until the later phases of *testing* and *after-deployment* will be characterized by higher severities than defects resolved while within the coding phase. Such a situation would negatively impact overall quality assurance costs, when the DePress tool would be used for defect prediction purposes, in comparison to simulated values. This threat opens an interesting opportunity for further research.

4 Discussion

Cost effectiveness within the simulated scenario is strictly related to the quality of prediction when measured with recall (Rec). Based on the simulation presented, it is possible to calculate the $NetReturn$ of using a proposed quality assurance effort allocation strategy for a series of defect prediction recall values. In such a way, we can observe how $NetReturn$ depends on Rec in terms of the proposed QA effort allocation strategy.

In industrial software development projects, quality assurance (defect fixing) consumes a significant amount of time and resources. By using the defect prediction technique, project members can obtain information on possible defect-prone elements of the software, before defects will occur, to optimally plan their quality assurance process. What is proposed in this paper, is a simple effort allocation strategy which is based on the DePress framework-driven defect prediction, defects distribution and the Boehm's Law, and which eliminates most of the quality assurance work during late (after-development) project's phases. Such an approach significantly increases quality assurance costs in development phase, however overall, QA costs will decrease in comparison to actual, real-life costs observed in the investigated project, as significantly less discoverable defects are left to be fixed in the later phases, where, according to Boehm's Law, bug-fixing costs are considerably higher. At the same time, we need to mention the low investment costs, when the open source DePress framework is used for defect prediction purposes. Low investment costs and high recall of even simple defect prediction performed by default in DePress, can result with high $NetReturn$ of DePress-aided quality assurance planned on a basis of the proposed effort allocation strategy. More sophisticated prediction models, especially ones using software process metrics [15,6], may help to achieve even more impressive results. It is also worth mentioning that cross-project software defect prediction [7,5] is sometimes used to reduce costs.

References

1. Boehm, B.W.: Software Engineering. IEEE Transactions on Computers 25(12), 1226–1241 (1976)
2. Endres, A., Rombach, D.: A Handbook of Software and Systems Engineering. Addison-Wesley (2003)
3. Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.: A Systematic Literature Review on Fault Prediction Performance in Software Engineering. IEEE Transactions on Software Engineering 38(6), 1276–1304 (2012)
4. Hryszko, J., Madeyski, L.: Bottlenecks in Software Defect Prediction Implementation in Industrial Projects. Foundations and Computing and Decision Sciences 40(1), 17–33 (2015), http://dx.doi.org/10.1515/fcds-2015-0002, DOI: 10.1515/fcds-2015-0002
5. Jureczko, M., Madeyski, L.: Towards Identifying Software Project Clusters with Regard to Defect Prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering. pp. 9:1–9:10. PROMISE '10, ACM,

New York, USA (2010), http://dx.doi.org/10.1145/1868328.1868342, DOI: 10.1145/1868328.1868342

6. Jureczko, M., Madeyski, L.: A Review of Process Metrics in Defect Prediction Studies. Metody Informatyki Stosowanej 30(5), 133–145 (2011), http://madeyski.e-informatyka.pl/download/Madeyski11.pdf
7. Jureczko, M., Madeyski, L.: Cross–project defect prediction with respect to code ownership model: An empirical study. e-Informatica Software Engineering Journal 9(1), 21–35 (2015), http://dx,doi.org/10.5277/e-Inf150102, DOI: 10.5277/e-Inf150102
8. Khoshgoftaar, T.M., Allen, E.B., Hudepohl, J.P., Aud, S.J.: Application of Neural Networks To Software Quality Modelling Of a Very Large Telecommunications System. IEEE Transactions on Neural Networks 8(4), 902–909 (1997)
9. Khoshgoftaar, T.M., Pandya, A.S., Lanning, D.L.: Application of Neural Networks for Predicting Faults. Annals of Software Engineering 1(1), 141–154 (1995)
10. Khoshgoftaar, T.M., Seliya, N.: Comparative Assessment of Software Quality Classification Techniques: An Empirical Case Study. Empirical Software Engineering 9(3), 229–257 (2004)
11. Khoshgoftaar, T.M., Seliya, N.: Assessment of a New Three-Group Software Quality Classification Technique: An Empirical Case Study. Empirical Software Engineering 10(2), 183–218 (2005)
12. Kläs, M., Nakao, H., Elberzhager, F., Münch, J.: Predicting Defect Content and Quality Assurance Effectiveness by Combining Expert Judgment and Defect Data-A Case Study. In: Proceedings of the 19th International Symposium on Software Reliability Engineering. pp. 17–26 (2008)
13. KNIME.COM AG: KNIME Framework Documentation (2016), https://tech.knime.org/documentation/, accessed: 2016.05.06
14. Li, P.L., Herbsleb, J., Shaw, M., Robinson, B.: Experiences and Results from Initiating Field Defect Prediction and Product Test Prioritization Efforts at ABB Inc. In: Proceedings of the 28th International Conference on Software Engineering. pp. 413–422 (2006)
15. Madeyski, L., Jureczko, M.: Which Process Metrics Can Significantly Improve Defect Prediction Models? An Empirical Study. Software Quality Journal 23(3), 393–422 (2015), http://dx.doi.org/10.1007/s11219-014-9241-7, DOI: 10.1007/s11219-014-9241-7
16. Madeyski, L., Majchrzak, M.: Software Measurement and Defect Prediction with DePress Extensible Framework. Foundations and Computing and Decision Sciences 39(4), 249–270 (2014), http://dx.doi.org/10.2478/fcds-2014-0014, DOI: 10.2478/fcds-2014-0014
17. Madeyski, L., Majchrzak, M.: ImpressiveCode DePress (Defect Prediction for software systems) Extensible Framework (2016), https://github.com/ImpressiveCode/ic-depress
18. Menzies, T., Jalali, O., Hihn, J., Baker, D., Lum, K.: Stable Rankings for Different Effort Models. Automated Software Engineering 17(4), 409–437 (2010)
19. Monden, A., Shinoda, S., Shirai, K., Yoshida, J., Barker, M., Matsumoto, K.: Assessing the Cost Effectiveness of Fault Prediction in Acceptance Testing. IEEE Transactions on Software Engineering 39(10), 1345–1357 (2013)
20. Moser, R., Pedrycz, W., Succi, G.: 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 (2008)

21. Müller, M.M., Padberg, F.: About the Return on Investment of Test-Driven Development. In: International Workshop on Economics-Driven Software Engineering Research EDSER-5. pp. 26–31 (2003)
22. Ostrand, T.J., Weyuker, E.J.: The Distribution of Faults in a Large Industrial Software System. SIGSOFT Software Engineering Notes 27, 55–64 (2002)
23. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Predicting the Location and Number of Faults in Large Software Systems. IEEE Transactions on Software Engineering 31(4), 340–355 (2005)
24. Ostrand, T.J., Weyuker, E.J., Bell, R.M.: Programmer-Based Fault Prediction. In: Proceedings of the Sixth International Conference on Predictive Models in Software Engineering. pp. 1–10 (2010)
25. Pressman, R.: Software Engineering: A Practitioner's Approach. McGraw-Hill (2010)
26. Rijsbergen, C.J.V.: Information Retrieval. Butterworth-Heinemann Newton (1979)
27. Rizwan, M., Iqbal, M.: Application of 80/20 Rule in Software Engineering Waterfall Model. In: Proceedings of the International Conference on Information and Communication Technologies '09 (2009)
28. Selby, R.W., Porter, A.: Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis. IEEE Transactions on Software Engineering 14(12), 1743–1756 (1988)
29. Slaughter, S.A., Harter, D.E., Krishnan, M.S.: Evaluating The Cost of Software Quality. Communications of the ACM 41(8), 67–73 (1998)
30. Source of Information on Salaries in Poland (2015), http://wynagrodzenia.pl/, accessed: 2015.02.28
31. Succi, G., Pedrycz, W., Stefanovic, M., Miller, J.: 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), 1–12 (2003)
32. Tosun, A., Bener, A., Turhan, B., Menzies, T.: Practical Considerations in Deploying Statistical Methods for Defect Prediction: A Case Study within the Turkish Telecommunications Industry. Information and Software Technology 52(11), 1242–1257 (2010)
33. Tosun, A., Turhan, B., Bener, A.: Practical Considerations in Deploying AI for Defect Prediction: A Case Study within the Turkish Telecommunication Industry. In: Proceedings of the Fifth International Conference on Predictor Models in Software Engineering. p. 11 (2009)
34. Turhan, B., Kocak, G., Bener, A.: Data Mining Source Code for Locating Software Bugs: A Case Study in Telecommunication Industry. Expert Systems with Applications 36(6), 9986–9990 (2009)
35. Turhan, B., Menzies, T., Bener, A., Stefano, J.D.: On the Relative Value of Cross-Company and within-Company Data for Defect Prediction. Empirical Software Engineering 14(5), 540–578 (2009)
36. Witten, I.H., Frank, E., Hall, M.A.: Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann (2005)
37. Wong, W.E., Horgan, J., Syring, M., Zage, W., Zage, D.: Applying Design Metrics to Predict Fault-Proneness: A Case Study on a Large-Scale Software System. Software: Practice and Experience 30(14), 1587–1608 (2000)