Preprint

QuASoQ 2020

8th International Workshop on
Quantitative Approaches to Software Quality

co-located with APSEC 2020
Singapore, December 1st , 2020

Editors:

Horst Lichter, RWTH Aachen University, Germany
Selin Aydin, RWTH Aachen University, Germany
Thanwadee Sunetnanta, Mahidol University, Thailand
Toni Anwar, University Petronas, Malaysia

Table of Content

Augmenting Window Contents with Transfer Learning for Effort Estimation

Sousuke Amasaki[a]

[a]Okayama Prefectural University, 111 Kuboki, Soja, 719-1197, Japan

Abstract

BACKGROUND: Some studies showed filtering out old completed projects with a window was effective for preparing a training dataset of an effort estimation model. Other studies showed selecting completed projects similar to a target project was also effective. The application of the similarity-based selection after the windowing approach was failed to synthesize their effects. The shortage of similar projects in the windowed pool was a potential cause of the failure. AIMS: To examine whether augmenting the window pool is effective to improve the estimation accuracy. METHOD: The moving windows approach was used for preparing a window pool. The similarity-based selection was applied to augment the pool. The selection assumes that projects in the pool form a set of virtual target projects. Old projects outside the pool were assumed to form a set of cross-company projects to be selected. The empirical study with a single-company ISBSG data was conducted to evaluate the effect. RESULTS: A positive synergistic effect was observed. The augmented window could synthesize the windowing approach and the similarity-based selection. It could also be combined with the similarity-based selection without performance degradation. CONCLUSIONS: Practitioners should consider adding projects similar to recently completed projects when effort estimation is based on historical data.

Keywords

effort estimation, moving windows, augmenting windows

1. Introduction

The success of software projects relies on many factors. The accuracy of software effort estimation is an serious influential factor at early project phase. Overestimation and underestimation have caused serious consequences for decades. Researchers have studied data-driven software effort estimation models while experts' judgment is still a primary choice in actual. The accuracy of the software effort estimation models is considered insufficient among not a few managers.

Software effort estimation models are affected by the adequacy of historical data from past projects. For instance, an organization's productivity is not stationary nor monotonic due to changes in the environment and the organization itself. Inaccurate effort estimation models would be obtained with the historical data that might not reflect the present productivity. A key to accurate software effort estimation is to prepare historical data that reflect the characteristics of a target project to be estimated.

A past study [1] examined two filtering techniques, namely, chronological filtering and relevancy filtering. The chronological filtering [2] removes too old project data. The relevancy filtering [3] removes dissimilar project data regarding metrics used for estimation. The study found that the combination of those techniques might be worse than the independent application.

The negative synergistic effect can be reasoned, at least, in two aspects. First, the relevancy filtering was applied after applying the chronological filtering. The chronological filtering does not care about feature variables and may select a subset that does not hold enough projects similar to a target project. It would be better to augment the subset with old but resemble projects using the relevancy filtering. Second, the simple average and median were used as effort estimation models as discussed in [1]. The simple models only used the effort variable for estimation and were insensitive to the change in the distribution of feature variables after the relevancy filtering.

This paper proposed an augmented chronological filtering based on the chronological filtering and the relevancy filtering. Its effects were investigated with a software effort estimation model using feature variables, in addition to the simple average and median models. The augmented filtering was also evaluated as alternative chronological filtering in the past combination method. The following questions were asked:

RQ1: Does augmenting moving windows with a relevancy filtering affects the estimation accuracy?

RQ2: Does using the augmentation as a chronological filtering affect the estimation accuracy of the past combination method?

QuASoQ 2020: 8th International Workshop on Quantitative Approaches to Software Quality, December 01, 2020, Singapore
EMAIL: amasaki@cse.oka-pu.ac.jp (S. Amasaki)
ORCID: 0000-0001-8763-3457 (S. Amasaki)

2. Related Work

2.1. Chronological Filtering

Although research in software effort estimation models has a long history, relatively few studies have taken into consideration the chronological order of projects. Therefore, chronological filtering has not been studied well compared with other topics in effort estimation.

To our knowledge, Kitchenham et al. [4] were first to suggest the use of chronological filtering. They built four linear regression models with four subsets, each of which comprised projects from different ranges of time duration. As the coefficients of the models were different from each other, they allowed to drop out older project data. Lokan and Mendes [2] were the first to study the effect of using moving windows in detail. They used linear regression (LR) models and a single-company dataset from the ISBSG repository. Training sets were defined to be the N most recently completed projects. They found that the use of a window could affect accuracy significantly; predictive accuracy was better with larger windows; some window sizes were particularly effective. Amasaki and Lokan also investigated the effect of using moving windows with Estimation by Analogy [5] and CART [6]. They found that moving windows could improve the estimation accuracy, but the effect was different than with LR.

Recent studies showed the effect and its extent could be affected by windowing policies [7] and software organizations [8]. Lokan and Mendes [7] investigated the effect on accuracy when using moving windows of various ranges of time duration to form training sets on which to base effort estimates. They also showed that the use of windows based on duration could affect the accuracy of estimates, but to a lesser extent than windows based on a fixed number of projects [8].

2.2. Relevancy Filtering

Relevancy filtering is a type of transfer learning approach. While many filtering approaches have been proposed for cross-project defect prediction (e.g., [9]), a few studies on cross-company effort estimation have evaluated the effects of relevancy filtering approaches.

Turhan and Mendes [3] applied brings a so-called NN-filter [10] to cross-company effort estimation of web projects. They showed that an estimation model based on raw cross-company data was worse than that based on within-company data but was improved as comparable one by using the NN-filter. Kocaguneli et al. [11, 12, 13] also introduced a transfer learning approach called TEAK for improving cross-company effort estimation. They applied it to transfer old project to a new project and found that TEAK was effective not only for cross-company effort estimation but also for cross-time effort estimation [14].

NN-filter is based on a nearest neighbor algorithm. In that sense, a study by Amasaki and Lokan [5] can be considered an evaluation study of the combination of the relevancy filtering and the chronological filtering. In that study, the combination worked well to improve estimation accuracy for a narrow range of window sizes. While that study used a wrapper approach for feature selection and logarithmic transformation in addition to the nearest neighbor algorithm, our study aims to explore the effects of the combination without such complicated factors. For that purpose, we adopted two simple estimation techniques that were not adopted in [5], described in the next section.

3. Methodology

3.1. Effort Estimation Techniques

In [1], average and median were used as software effort estimation models. The average was adopted because it uses the whole training set and is sensitive to the distribution of effort values in the training set. The median was adopted because it is robust to the distribution and contrasts with the average. These models estimate efforts without adjustments based on feature variables of projects.

To examine the difference in the use of feature variables in software effort estimation, we also adopted Lasso [15] for our experiment. Lasso is a kind of penalized linear regression models. Past studies on the chronological filtering used Lasso and showed that the chronological filtering was effective with it. Our experiment used `LassoLarsIC` of scikit-learn library.

3.2. Chronological Filtering

This study adopted fixed-size moving windows [2] and fixed-duration moving windows [8]. The latest N finished projects were selected as a training set by the fixed-size moving windows. The fixed-duration moving windows selected the latest projects finished within N months. As N influences on the effectiveness of moving windows, we explored various values as well as past studies.

3.3. Relevancy Filtering

This study used a nearest neighbor algorithm as a relevancy filtering approach. It is also called NN-filter [10].

The procedure of NN-filter is as follows:

1. Select k closest instances of history data to each instance of target project data in terms of unweighted Euclidean distance.
2. Combine the selected instances without duplication.

Note that each feature of project data was normalized with min-max normalization before the distance calculation.

As the synergistic effect could be observed with effective filtering, the relevancy filtering had to be configured as effective. For average and mean effort estimation models, we roughly fixed $k = 3$, which is the smallest number which can make average and median estimations give distinct efforts. For lasso, we roughly fixed $k = 10$, half of the minimum of the window sizes we explored. In general, increasing k would lead to worse estimation if NN-filter works well. Hence, these values could not be the best but were expected more reasonable than larger ks.

3.4. Augmentation

The augmentation adds old projects selected by the relevancy filtering into a subset obtained by the chronological filtering as follows:

1. Recently completed projects are selected with the moving windows approach.
2. NN-filter is applied to select projects from the remained old projects. The most similar project to each project of the recently completed projects is selected. The set of selected projects has no duplicate.
3. The selected projects and the recently completed projects are combined.
4. The combined projects are used to train a software effort estimation model.

Note that NN-filter uses the effort variable in addition to feature variables. As efforts of the past projects are known, it is possible to use the effort variable in the augmentation process.

The augmentation shares the same assumption as the chronological filtering that the recently completed projects resemble a target project to be estimated. Results of NN-filter are also expected to pretend to be as fresh as the recently completed projects. Therefore, the selected projects are considered to keep the similarity to the target project.

3.5. Combination

The combination of the chronological filtering and the relevancy filtering was investigated in [1]. The chronological filtering and the relevancy filtering were combined as follows:

1. Recently completed projects are selected with the moving windows approach. The remained old projects are discarded.
2. NN-filter is applied to select projects from the recently completed projects. The selected projects resemble a target project to be estimated.
3. The selected projects are used to train a software effort estimation model.

The combination method was found less effective than each of the filtering methods in [1] with mean and median models.

The augmentation can be considered a variation of moving windows approach while it is a way to combine moving windows and NN-filter. In this paper, this combination was also examined using a subset obtained by the augmentation. As the augmentation has more projects, NN-filter might bring better neighbors from an augmented subset.

3.6. Experiment procedure

As the chronological filtering relies on the time proximity, our experiment needs to assume a situation that a development organization needs to respond to continuously coming new projects. The size of windows influences on where our experiment starts. As same as the past studies, our experiment with a specific window size was conducted as follows:

1. Sort all projects by starting date.
2. For a given window size N, find the earliest project p_0 for which at least $N + 1$ projects were completed prior to the start of p_0 (projects from p_0 onwards are the ones whose training set is affected by using a window, so they form the set of evaluation projects for this window size. For example, with a window of 20 projects, at least 21 projects must have finished for the window to differ from the growing portfolio.)
3. For every project p_i in chronological sequence, starting from p_0, form a training set using moving windows and the growing portfolio (all completed projects).
    - For no filtering, the training set is all projects that finished before p_i started.

Table 1
Summary statistics for ratio-scaled variables in data from single ISBSG organization

Variable	Min	Mean	Median	Max	StDev
Size	10	496	266	6294	699
Effort	62	4553	2408	57749	6212
PDR	0.53	16.47	8.75	387.10	31.42

- For fixed-size moving windows, the training set is the N most recent projects that finished before p_i started. If multiple projects finished on the same date, all of them are included.
- For fixed-duration, the training set is the most recent projects whose whole life cycle had fallen within a window of D months prior to the start of p_i.

4. Estimate an effort of a target project based on past project data.
   - For no filtering, the training set from the previous step is used.
   - For relevancy filtering, a subset selected by a nearest neighbor from the training set is used.
   - For the augment method, an augmented set of the training set with the projects not selected in the previous step is used.
5. Evaluate the estimation results.

This study used the single-company subset of the ISBSG dataset that was analyzed in [2, 7, 8, 5, 6, 16]. Table 1 shows summary statistics. We explored window sizes from 20 to 120 projects for the size-based moving windows and from 12 to 84 months for the duration-based moving windows as well as the past study [17]. No filtering, called the growing portfolio in past studies, was used as a baseline for comparing the filtering methods.

3.7. Performance Measures

The accuracy statistics that we used to evaluate the effort estimation models are based on the difference between estimated effort and actual effort. We used Mean Absolute Error (MAE), which is widely used to evaluate the accuracy of effort estimation models, as it is an unbiased measure that favours neither under- nor over-estimates.

We concentrate first on the statistical significance of differences in accuracy that arise from using the filtering approaches. To test for statistically significant differences between accuracy measures, we use the two-sided Wilcoxon signed-rank test (`wilcoxon` function of the `scipy` package for Python) and set the statistical significance level at $\alpha = 0.05$. The setting of this study is a typical multiple testing, and the p-values of the tests must be controlled. Bonferroni correction is a popular method for this purpose. However, the adoption of this simple correction results in the lack of statistical power, especially for not large effects. We thus controlled the false discovery rate (FDR) of multiple testing [18] with the "`multipletests`" function of the `statsmodels` package in Python. FDR is a ratio of the number of falsely rejected null hypotheses to the number of rejected null hypotheses.

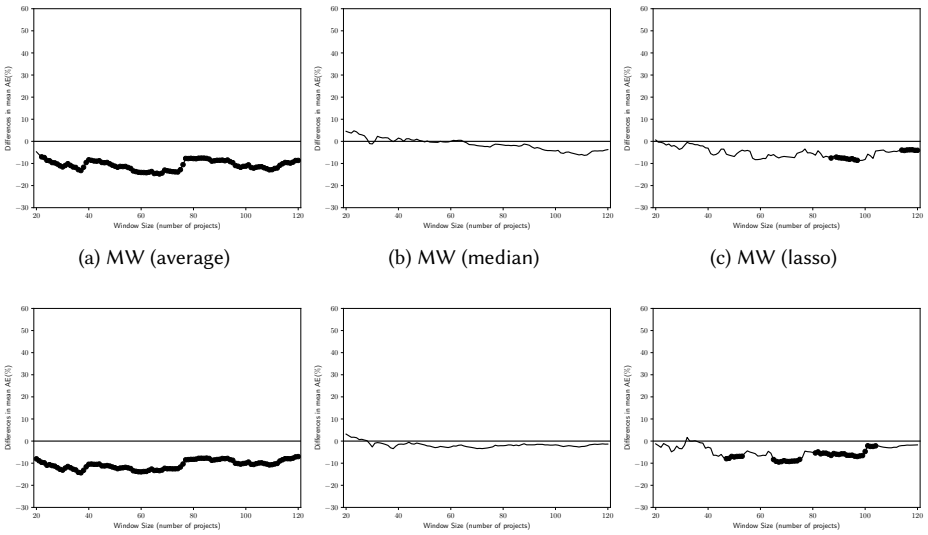
4. Results and Discussion

4.1. Comparisons between Moving Windows and Augmentation

Figure 1 has 6 plots showing the difference in mean absolute error against window sizes using the fixed-size moving windows (baseline) and the augmentation with it. The x-axis of each figure is the size of the window, and the y-axis is the subtraction of the accuracy measure value with the growing approach from that with the moving windows at the given x-value. The moving windows and the augmentation with it were advantageous where the line is below 0. Circle points mean a statistically significant difference, with the moving windows or the augmentation with it, being better than the growing portfolio. At these points, the corresponding FDR-controlled p-value was below $\alpha = 0.05$.

Figure. 1 revealed the effect of using the fixed-size moving windows and the augmentation, compared to always using the growing portfolio as follows:

- With average effort estimation, statistically significant differences were found for almost all window sizes. The augmentation did not bring clear changes except for small window sizes, where additional statistically significant differences were found.
- With median effort estimation, no statistically significant difference was found for all window sizes. The augmentation improved the performance a bit for smaller window sizes but worsened it a bit for larger window sizes. The ef-



(a) MW (average) (b) MW (median) (c) MW (lasso)

(d) Augmentation(average) (e) Augmentation (median) (f) Augmentation (lasso)

Figure 1: The difference in mean absolute error against moving windows (growing portfolio vs. fixed-size MW and Augmentation)

fects never caused a statistically significant difference.

- With lasso, statistically significant differences were found when window size is between 85 and 95 or is more than 110. The augmentation made the advantages in other window sizes statistically significant. The significant differences in larger window sizes disappeared instead. Note that lasso was more accurate than the others even when used with the growing portfolio.

These observations suggested that the augmentation could bring a positive synergistic effect on the estimation accuracy when the augmentation was applied to fixed-size windows with average or lasso.

Figure 2 plotted the same comparisons but using the fixed-duration moving windows. In the figure, square points mean a statistically significant difference, with the fixed-duration moving windows being worse than the growing portfolio. These figures revealed the effects of the fixed-duration moving windows and the augmentation with it, compared to always using the growing portfolio as follows:

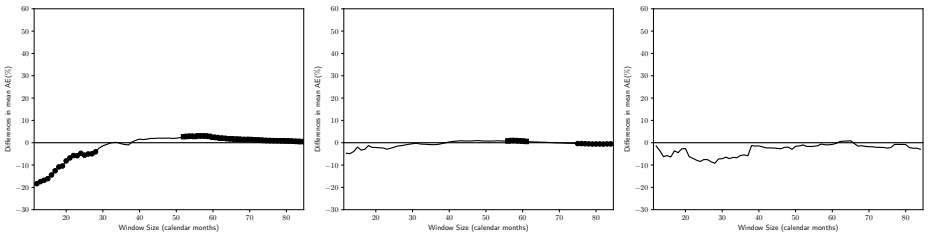
- With average effort estimation, the effective window range was between 20 months and less than 30 months. The growing portfolio got advantageous for more than 53 months. The augmentation extended the advantageous range to more than 40 months. The growing portfolio was no longer advantageous for larger window sizes.
- With median effort estimation, the effective window range was more than 60 months. Disadvantageous window sizes are between 55 months and 60 months. The augmentation made the statistically significant differences disappeared.
- With lasso, there was no significant difference. There was no clear advantage nor disadvantage. The augmentation made no statistically significant difference while the difference got closer a bit.

These observations suggested that the augmentation could improve the estimation accuracy when the augmentation was applied to fixed-duration windows with average effort estimation.

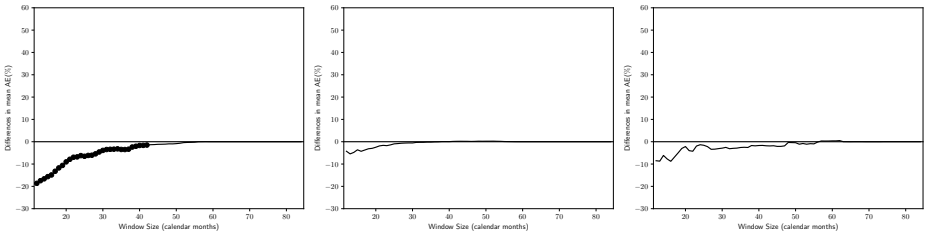
The answer to RQ1 is *yes*: Augmenting moving windows with a relevancy filtering was useful. It did not cause an apparent negative synergistic effect, at least. It sometimes made positive synergistic effects.

4.2. Evaluation of Combination of Augmented MW and NN-filter

The combination of the augmented moving windows and the NN-filter was evaluated under the same situations. The number of neighbors was set to 3 for av-

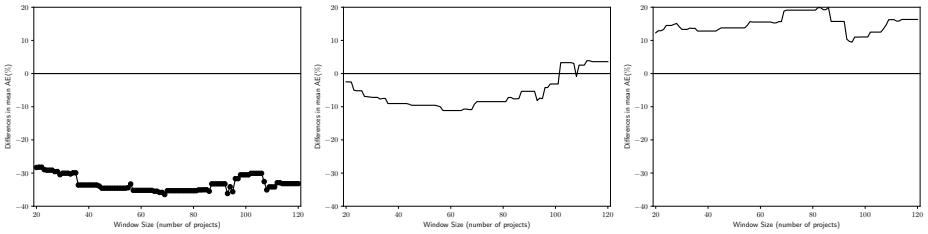


(a) MW (average) (b) MW (median) (c) MW (lasso)

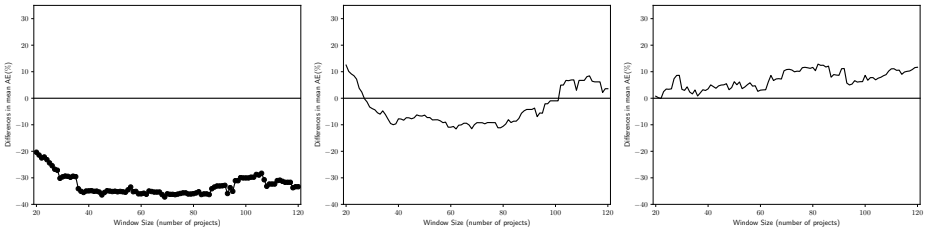


(d) Augmentation(average) (e) Augmentation (median) (f) Augmentation (lasso)

Figure 2: The difference in mean absolute error against moving windows (growing portfolio vs. fixed-duration MW and Augmentation)



(a) NN (average) (b) NN (median) (c) NN (lasso)



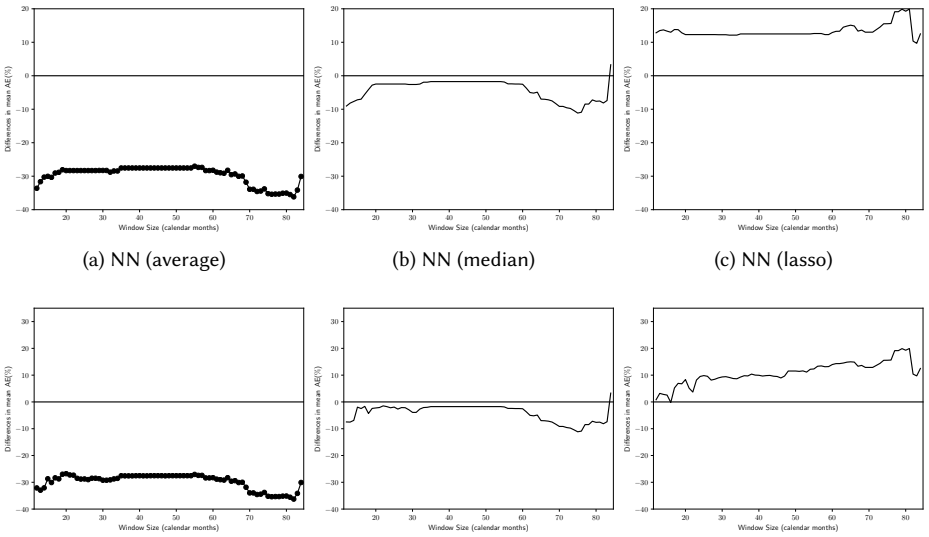
(d) AG + NN (average) (e) AG + NN (median) (f) AG + NN (lasso)

Figure 3: The difference in mean absolute error against moving windows (growing portfolio vs. fixed-size MW + Augmentation + NN-filter)

erage and median effort estimation models and 10 for lasso because lasso models, as described in Section 3.3.

Figure 3 has 6 plots showing the difference in mean absolute error against fixed-size window sizes using the NN-filter and using the combination of the augmented windows and the NN-filter. These figures revealed the effects of using the NN-filter and the augmented moving windows with it, compared to always using the growing portfolio as follows:

- With average effort estimation, the NN-filter made statistically significant differences for almost all window sizes. Combining the augmented moving windows with the NN-filter made no clear change except for small window sizes, where the



(a) NN (average) (b) NN (median) (c) NN (lasso)

(d) AG + NN (average) (e) AG + NN (median) (f) AG + NN (lasso)

Figure 4: The difference in mean absolute error against moving windows (growing portfolio vs. fixed-duration MW + Augmentation + NN-filter)

differences got smaller from about -30% to -20%. The significance of the differences was retained, though.

- With median effort estimation, the NN-filter made no clear change while it caused positive effects as depicted by the line running below the zero line for a wide window range. Combining the augmented moving windows with the NN-filter made no clear change except for small window sizes. No statistically significant difference appeared.
- With lasso, the NN-filter made no statistically significant change though it worsened the performance. Combining the augmented moving windows with the NN-filter mitigated the degradation. Note that the augmentation made the significant improvement as shown in Fig. 1(f). The NN-filter canceled the improvement.

In [1], the combination of the moving windows and NN-filter caused a negative synergistic effect. For example, less than half of the window sizes could achieve the improvement of -30% or more where mean effort estimation was applied. The augmentation made the performance improvement of -30% or more for more than a half of the range as shown in Fig. 3(d). Therefore, these observations suggested that the augmented moving windows did not result in a negative synergistic effect caused by the combination of fixed-size moving windows and NN-filter.

Figure 4 plotted the same comparison but using the fixed-duration moving windows. These figures revealed the effects of using the NN-filter and the augmented moving windows with it, compared to always using the growing portfolio as follows:

- With average effort estimation, NN-filter made statistically significant differences for almost all window sizes. Combining the augmented moving windows with the NN-filter made no clear change.
- With median effort estimation, NN-filter made no clear change while it caused positive effects as depicted by the line running below the zero line for a wide window range. Combining the augmented moving windows with the NN-filter made no clear change.
- With lasso, NN-filter made no statistically significant change though it worsened the performance. Combining the augmented moving windows with the NN-filter mitigated the degradation by NN-filter.

Therefore, these observations suggested that the augmentation did not result in a negative synergistic effect caused by the combination of fixed-size moving windows and NN-filter. Rather the degradation by NN-filter could be mitigated.

The answer to RQ2 is as follows: The combination of the augmented chronological filtering and the relevancy filter did not bring a negative synergistic effect except for small window sizes. Rather, the negative effect caused by the relevancy filtering was mitigated by the augmentation.

5. Conclusion

We explored the effects of the augmentation and its combination with a relevancy filtering for effort estimation. We confirmed the augmentation was a useful way to bring a positive synergistic effect of the chronological filtering and the relevancy filtering. Combining the augmented windows with the relevancy filtering, as well as in [1] also diminished the negative synergistic effect caused by the combination of the moving windows and NN-filter found in a past study. We thus concluded that the augmentation can be a good way to combine the two filtering approaches and also a good extension of the moving windows, which can be safely combined with the relevancy filtering.

Further investigation considering other transfer learning approaches is in future work. The NN-filter used for augmentation is a type of transfer learning, it is interesting to examine the effects of other approaches such as [14] for augmentation. Some transfer learning approaches for cross-project defect prediction [19] can also be applied. The threat to external validity can be mitigated with additional project data.

Acknowledgments

This work was partially supported by JSPS KAKENHI Grant #18K11246.

References

[1] S. Amasaki, Exploring Preference of Chronological and Relevancy Filtering in Effort Estimation, in: Proc. of Profes 2019, Springer, 2019, pp. 247–262.

[2] C. Lokan, E. Mendes, Applying moving windows to software effort estimation, in: Proc. of ESEM 2009, 2009, pp. 111–122.

[3] B. Turhan, E. Mendes, A Comparison of Cross-Versus Single-Company Effort Prediction Models for Web Projects, in: Proc. of SEAA, IEEE, 2014, pp. 285–292.

[4] B. Kitchenham, S. Lawrence Pfleeger, B. McColl, S. Eagan, An empirical study of maintenance and development estimation accuracy, The Journal of Systems & Software 64 (2002) 57–77.

[5] S. Amasaki, C. Lokan, The Effects of Moving Windows to Software Estimation: Comparative Study on Linear Regression and Estimation by Analogy, in: Proc. of IWSM-MENSURA 2012, IEEE, 2012, pp. 23–32.

[6] S. Amasaki, C. Lokan, The Effect of Moving Windows on Software Effort Estimation: Comparative Study with CART, in: Proc. of IWESEP 2014, IEEE, 2014, pp. 1–6.

[7] C. Lokan, E. Mendes, Investigating the Use of Duration-Based Moving Windows to Improve Software Effort Prediction, in: Proc. of APSEC 2012, 2012, pp. 818–827.

[8] C. Lokan, E. Mendes, Investigating the use of duration-based moving windows to improve software effort prediction: A replicated study, Inf. Softw. Technol. 56 (2014) 1063–1075.

[9] S. Herbold, CrossPare: A tool for benchmarking cross-project defect predictions, in: Proc. of 30th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), IEEE, 2016, pp. 90–95.

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

[11] E. Kocaguneli, T. Menzies, How to Find Relevant Data for Effort Estimation?, in: Proc. of ESEM, IEEE, 2011, pp. 255–264.

[12] E. Kocaguneli, G. Gay, T. Menzies, Y. Yang, J. W. Keung, When to use data from other projects for effort estimation, in: Proc. of ASE, ACM, 2010, pp. 321–324.

[13] E. Kocaguneli, T. Menzies, A. B. Bener, J. W. Keung, Exploiting the Essential Assumptions of Analogy-Based Effort Estimation, IEEE Transactions on Software Engineering 38 (2012) 425–438.

[14] E. Kocaguneli, T. Menzies, E. Mendes, Transfer learning in effort estimation, Empirical Software Engineering 20 (2015) 813–843.

[15] R. Tibshirani, Regression shrinkage and selection via the lasso, J. Roy. Statist. Soc. Ser. B (1996) 267–288.

[16] S. Amasaki, C. Lokan, Evaluation of Moving Window Policies with CART, in: Proc. of IWESEP 2016, IEEE, 2016, pp. 24–29.

[17] S. Amasaki, C. Lokan, A Replication of Comparative Study of Moving Windows on Linear Regression and Estimation by Analogy, in: Proc. of PROMISE, ACM Press, 2015, pp. 1–10.

[18] Y. Benjamini, D. Yekutieli, The control of the false

discovery rate in multiple testing under dependency, Annals of statistics 29 (2001) 1165–1188.
[19] S. Herbold, Training data selection for cross-project defect prediction, in: Proc. of PROMISE '13, ACM, 2013, pp. 6:1–6:10.

On the Evolutionary Properties of Fix Inducing Changes

Syed Fatiul Huq[a], Md. Aquib Azmain[b], Nadia Nahar[c] and Md. Nurul Ahad Tawhid[d]

Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh

Abstract

A major aspect of maintaining the quality of software systems is the management of bugs. Bugs are commonly fixed in a corrective manner; detected after the code is tested or reported in production. Analyzing Fix-Inducing Changes (FIC) — developer code that introduces bugs — provides the opportunity to estimate these bugs proactively. This study analyzes the evolution of FICs to visualize patterns associated with the introduction of bugs throughout and within project releases. Furthermore, the association between FICs and complexity metrics, an important element of software evolution, is extracted to quantify the characteristics of buggy code. The findings indicate that FICs become less frequent as the software evolves and more commonly appear in the early stages of individual releases. It is also observed that FICs are correlated to longer Commit intervals. Lastly, FICs are found to be more present in codes with fewer lines and less cyclomatic complexity, which corresponds with the law of growing complexity in software evolution.

Keywords

software evolution, fix-inducing changes, data mining,

1. Introduction

Software projects evolve over time [1] to introduce new features while fixing bugs [2] that appear in parallel. The conventional way of handling the bugs is by detecting faulty codes with test cases [3] based on user reports and writing patches [4] that eliminate the fault. In this way, a bug is fixed only after it is written. Another way of managing bugs is proactively understanding how bugs occur in systems. In this preventive process, Fix-Inducing Changes (FICs) — code that introduces bugs which induce a later fix [5] — are analyzed. FICs can be tracked from a project's change history by looking for instances of bug fixes and the code changed in these fixes. An FIC provides information about the code changes, the developer writing the bug, and the state of the development process at the time of introducing the bug. These can unveil important characteristics of the project, processes and developers that potentially cause bugs.

Studies analyzing FICs have observed how these are related to or affected by properties of the software development lifecycle. For instance, Sliwerski et al. [5], apart from coining the term, related FICs with two developer activities: the day of coding and the amount of code in a single Commit. Yin et al. [6] observed how bug fixes themselves can introduce new bugs. Other studies include relations with code smells[7], code coupling [8], developer sentiment [9, 10] and more.

Since FICs are a component of the software's history, these hold information about the software's evolution. As software evolves, so does its management, personnel, design and code. This changing environment can affect the introduction of bugs and vice versa. The existence and properties of such correlations can be established by analyzing the evolution of FICs. Observing the evolution of FICs can also help uncover its relation with other evolutionary factors, for instance, the system's complexity [11]. With this aim, this study answers the following Research Questions (RQs):

RQ1: How do FICs evolve with the software? This RQ observes how FICs change in frequency and ratio as the software system evolves. The evolution of the software is measured with its releases.

RQ2: How do FICs exist within releases? A single release depicts the software team's complete flow of activities. The flow starts with the team taking in new requirements to update the features of the software to their finalization, testing and deployment. This RQ observes how FICs appear and change in this flow.

RQ3: How do FICs relate to Commit interval? The interval between Commits signify the amount of tasks assigned to developers, along with gaps between activities. This RQ answers whether FICs behave differently than regular Commits in terms of these intervals.

RQ4: How do FICs relate to system complexity? According to Lehman's law of evolution, system complexity is a vital part of a software's evolution [11]. The law dictates that complexity increases as the software evolves. Since FICs are instances where bugs are introduced, and bugs can be affected by system complexity, this RQ observes the relation between the two entities. Specifically, in this RQ, FIC is correlated to Lines of Code (LoC) and Cyclomatic Complexity (CC) as commonly used metrics to quantify complexity [12, 13].

QuASoQ 2020: International Workshop on Quantitative Approaches to Software Quality, 1st December 2020, Singapore

EMAIL: bsse0732@iit.du.ac.bd (S. F. Huq); bsse0718@iit.du.ac.bd (Md. A. Azmain); nadia@iit.du.ac.bd (N. Nahar); tawhid@iit.du.ac.bd (Md. N. A. Tawhid)

For this study, eight Java repositories from GitHub with a total of 142,555 Commits have been analyzed. From the Commits, FICs are detected, release information are extracted and relevant metrics are calculated. The findings show that FICs decrease as the software evolves, while remaining more prevalent early in release cycles. Statistically analyzing the data shows that FICs contain larger intervals and their code reduced LoC and CC than regular Commits.

2. Related Work

Sliwerski et al.[5] introduced the term Fix-Inducing Changes (FIC), providing a process that detects FICs in version history from Concurrent Versions System (CVS) with bug reports from Bugzilla. Moreover, they showed a relation between FICs and number of files changed. Antoniol et al.[14] showed that FICs create adverse effect and produce unexpected results. They presented a robust approach to detect groups of co-changing files in CVS repositories.

Yin et al.[6] identified and analyzed incorrect bug fixes which introduce new ones instead. They analyzed the code of operating systems namely, FreeBSD, Linux and OpenSolaris. This approach also combined version control systems and bug repository to categorize changes. They proved that Fix Inducing Fix (FIF) can cause crashes, hangs, data corruption or security problems.

Bavota et al.[7] showed that 15% refactoring tasks induce bugs, analyzing 52 kinds of refactoring on 3 Java projects. They detected inheritance related refactoring as the most error-prone refactoring.

In order to analyze FICs, various works focused on different properties of change that would induce the bugs. Levin et al.[15] and Menzies et al.[16] focused on source code changes of affected files. Fukushima et al. [17] introduced developer experience, time of day, time interval of commit and some other properties of change that would induce bugs. Sadiq et al. [8] related FICs with change couplings to find that recent change couples provide better insight on new errors. Huq et al. [9] showed that developer sentiment is related with FICs, where positive comments and reviews in Pull Requests can lead to buggy Commits.

Weicheng et al.[18] explored the relation between developer Commit patterns in GitHub and software evolution. They used four metrics to measure the Commit activity of developers and code evolution: changes, interval, author and source code dependency. Moreover, this paper showed techniques to visualize these metrics for a given project. They developed a tool named Commits Analysis Tool (CAT) that finds that the changes in previous versions can affect the file which is dependent on it in the next version.

Osman et al.[19] extracted bug-fix patterns by mining change history of 717 open source projects. They manually inspected the patterns to retrieve the context and reasons that cause those bugs.

So far, FICs have been analyzed to derive relationships with different project metrics. While the evolution of Commits has been observed, the evolutionary properties of FICs have yet been studied.

3. Methodology

This study observes how Fix-Inducing Changes (FIC) evolve throughout the lifetime of software projects and how these relate to complexity metrics. The methodology of the study is divided into three parts, described as follows.

3.1. Fix-Inducing Change (FIC) Detection

FICs are changes to code that causes problems to the software system. FICs are the introduction of bugs or errors to the software, inducing fixes in the future. Hence, these can be detected from the changes that fix bugs and errors.

This study utilizes Commits, the documented changes in software projects that are managed through version controlling. Commits contain the exact lines and files where changes are made along with information of and message from the developer who posted these. The detection of FICs through Commits is conducted in the following steps, influenced by the process of [7]

1. All Commits are fetched from GitHub repositories.
2. Commit messages are extracted to detect terms such as “bug”, “fix” or “patch”. These terms signify that the aim of the Commit is related to the management of bugs.
3. Now the changes in these Commits are analyzed. Since the study deals with Java projects, it is first checked whether the changes occur in “.java” files. Commits with no changes to such files indicate that the Commits dealt with non-code entities of the software (configuration files, documentation etc.). Furthermore, the changes made in “.java” files are analyzed to see whether the changes were code comments, which also signifies the absence of code entities.
4. Then, the type of the edit made by the Commit is checked. There are three types of edit: Insert, Delete and Replace. An Insert edit means that a patch code is added onto the existing code base. However, it does not help to track which part of the previous code was buggy. There is no way of tracking back to a Commit that introduced a

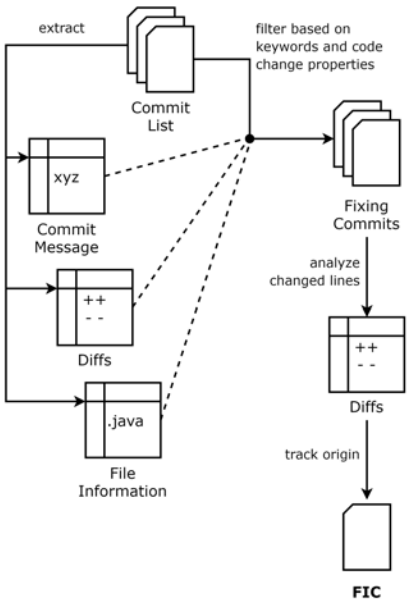


Figure 1: The methodology for identifying FICs from Commits

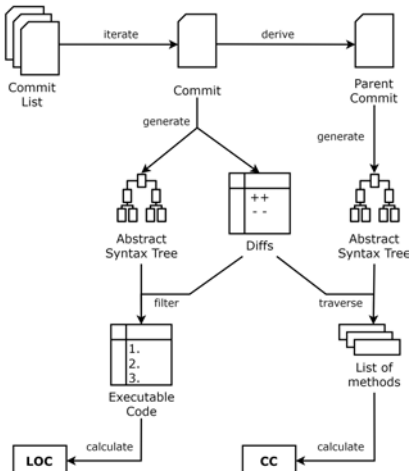


Figure 2: The methodology for extracting complexity metrics from Commits

bug. Hence, Commits with Insert edit types are discarded from further consideration.

5. After the filtering process, the remaining Commits are labeled as "Fixing Commits" or "Fixes". These are the Commits that removed buggy code.
6. Next, origin of each legitimate change in the Fix is tracked using the blame function, which returns the Commit where a specified changed line was last added or modified. These Commits are labeled as FICs.

3.2. Evolution Analysis

To understand the evolution of FICs, the projects' release tags are analyzed. Release tags define iterative final versions of the software in the project's lifetime. Since Commits are assigned these tags, it is possible to categorize Commits based on releases.

To analyze releases, first non-release tags are filtered out based on naming structures. Usually the release tags in most projects abide by the pattern: *"v #.#.#"*. The rest are tags depicting other information like branches or experiments. However, the structure of naming tags can vary with projects. For instance, patterns in projects like ElasticSearch or Commons-lang are *"Elasticsearch #.#.#"* and *"commons-lang-#.#.#"* respectively. Hence, tags are manually analyzed for each repository. Additionally, versions that are release candidates are discarded since these do not depict final releases.

Next, Commits are labeled based on their assigned tags. However, since Commits are automatically assigned all future tags, the labeling was conducted in two parts. First, Commits are extracted from all tags. Then, for every tag, only those Commits that were posted after the previous release tag are assigned to the current one.

As Commits are segmented into releases, analysis for Research Question (RQ) 1 is conducted. For each release, the number of FIC and non-FIC is calculated based on section 3.1. This segmentation is further elaborated for RQ2, by dividing each release into three equal parts. The three divisions are extracted to better understand the early, middle and late stages of a single release.

3.3. Metrics Extraction

To analyze the relation between FICs and complexity metrics — Line of Code (LoC) and Cyclomatic Complexity (CC) — first the changes to code are extracted. This includes the content of the changed files and numbers of lines which are modified or deleted. In the case of file contents, along with that of the current Commit, contents of its parent Commit are also extracted. Parent Commit is referred to the Commit directly prior to the current Commit. The contents of the parent Commit provides information of the state of code before the current Commit's changes. For FICs, their parents retains the properties of the code where the bug was introduced. With the contents of the current and parent Commits, the two metrics are calculated in the following manner:

1. **LoC**: To calculate the line of code, without considering comments, first the Abstract Syntax Tree (AST) [20] of a program is generated from the

changed files using JavaParser[1]. It is verified whether the changes have been conducted on executable code or not. Therefore, blank spaces are eliminated. Next, the modified lines in the current content are checked to identify whether these are comments based on the AST. If none of the changed lines are executable code, then the file is not further considered. Otherwise, the LoC of the parent content is calculated.

2. **CC**: To calculate cyclomatic complexity, the method in which change was introduced is first identified. This is done by taking each changed line of the current content and tracing its state back in the parent content. The generated AST of the parent content is traversed for each line. Each changed line of executable code is individually assessed and provided an associated method. This provides a list of changed methods for a single file. Each of their cyclomatic complexities are calculated, aggregating all possible paths (If-else, loops, switch statements).

4. Experimentation and Findings

Description of the dataset and the results observed for the 4 Research Questions (RQs) are described as follows.

4.1. Dataset

To conduct this research, eight well known Java projects are chosen from GitHub's repositories. These projects are open source and use GitHub as their primary medium of code storage and version control, enabling the extraction of all necessary Commits. Details of the repositories are displayed in Table 1. The projects comprise of a total of 142555 Commits to analyze. All eight repositories are used to analyze Research Questions (RQ) 1, 2 and 3. RQ4, which requires the source code, utilizes the first five repositories.

4.2. RQ1: Evolution of FICs

The 1st RQ aims to understand how FICs evolve, in terms of frequency and ratio, throughout the lifetime of software projects. The graphs in Figure 3 showcase the evolution of FICs in the eight software repositories analyzed. The different repositories show different types of patterns. In the majority of patterns, as seen in Figures 3(b, c, d, g, h) for projects Guava, Mockito, Commons-lang, Elasticsearch and Spring-framework respectively, FICs appear in the early stages of the projects' lifetime and decrease in newer versions. This indicates that earlier changes to the system tend to contain more instances of bug introduction. This can be due to a rapidly changing and volatile initial requirement, formative and incomplete development processes, lack of collaborative experience among the developers, or an insufficiency of reviewing and testing resources. But as the software evolves, the FICs get reduced, as an indication of bolstered testing and quality assurance processes, and project maturity.

On the other hand, projects Tomcat and Hadoop in Figures 3(a, e) show the opposite trend, where FICs are more predominant in later versions. This could happen due to a decreased level of scrutiny in reviewing efforts, a overhaul of new requirements, or other project and personnel related events. Only Figure 3(f) showcases a slightly more uniform pattern of FICs for the project Selenium. Although there are spikes of FICs occurring in specific versions, there is no apparent progression in the appearance of FICs.

Such visualizations of the evolution of FICs help in observing the history of the project in terms of buggy changes. This can be related to other aspects of projects that coincide with the decrease and increase of FICs to understand what affects the introduction of bugs from a high level perspective.

Table 1
Repository description of the eight projects

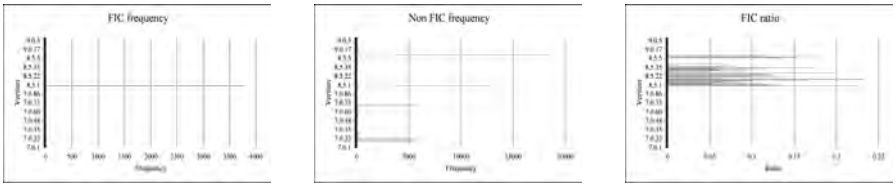
Project Name	Commits	Lifetime (Years)	Contri--butors
Apache Tomcat	19360	8.5	21
Google Guava	4798	5	187
Mockito	5019	6.7	155
Commons-lang	5396	10	115
Apache Hadoop	21435	4.8	191
Selenium	23550	9.3	435
Elastic Search	44975	8.5	1216
Spring Framework	18022	6.4	369
Total	142555	59.1	2689

4.3. RQ2: FICs in Releases

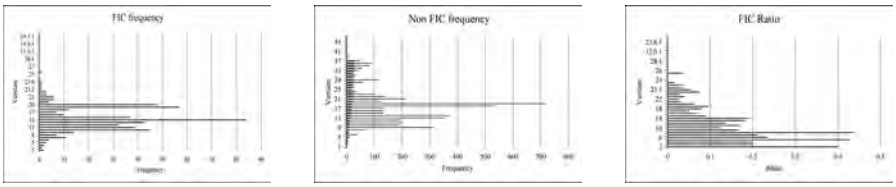
In RQ2, the pattern of FICs within individual releases is observed. In Figure 4, the appearance of FICs within releases is displayed as black circles, where the size of the circle is determined by the proportion of FIC on the total number of Commits in that stage. The releases are divided into three stages: early, middle and late, and for some projects, versions are merged for visibility.

It can be seen that for almost all the projects, FICs are more predominant in early and middle stages of releases compared to late ones. The exceptions are Tomcat and Spring framework, where FICs are similarly or more prevailing in the late stages. The high level of appearance of FICs in early and middle stages of a release can be con-

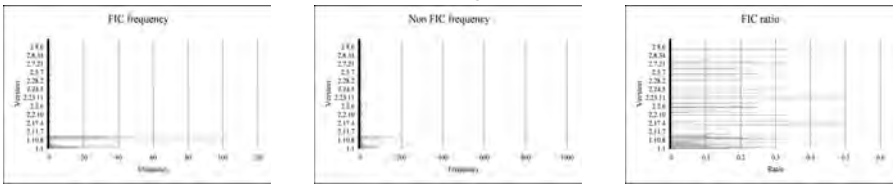
[1] https://javaparser.org/



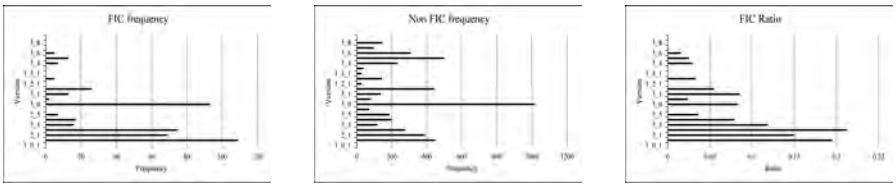
(a) Project: tomcat



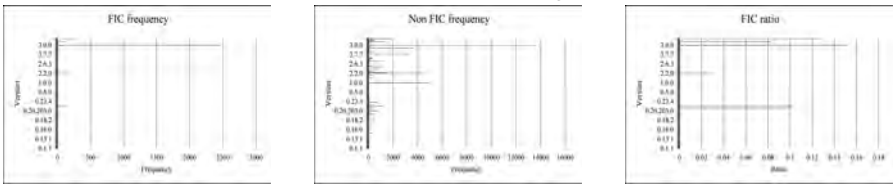
(b) Project: guava



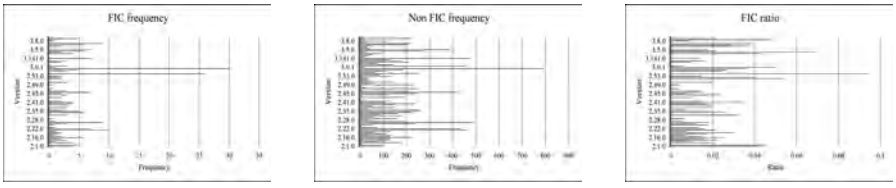
(c) Project: mockito



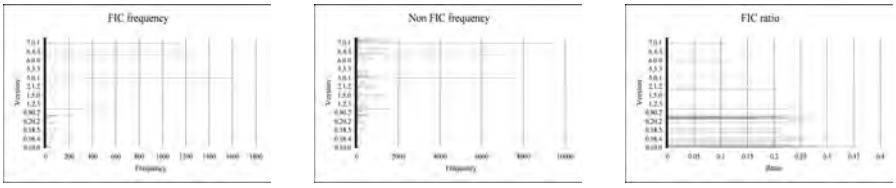
(d) Project: commons-lang



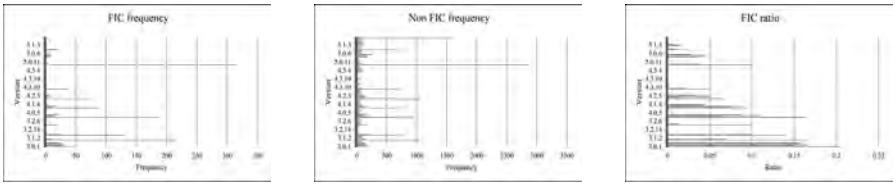
(e) Project: hadoop



(f) Project: selenium



(g) Project: elasticsearch



(h) Project: spring-framework

Figure 3: Evolution of FICs: FIC frequency, Non-FIC frequency and FIC ratio

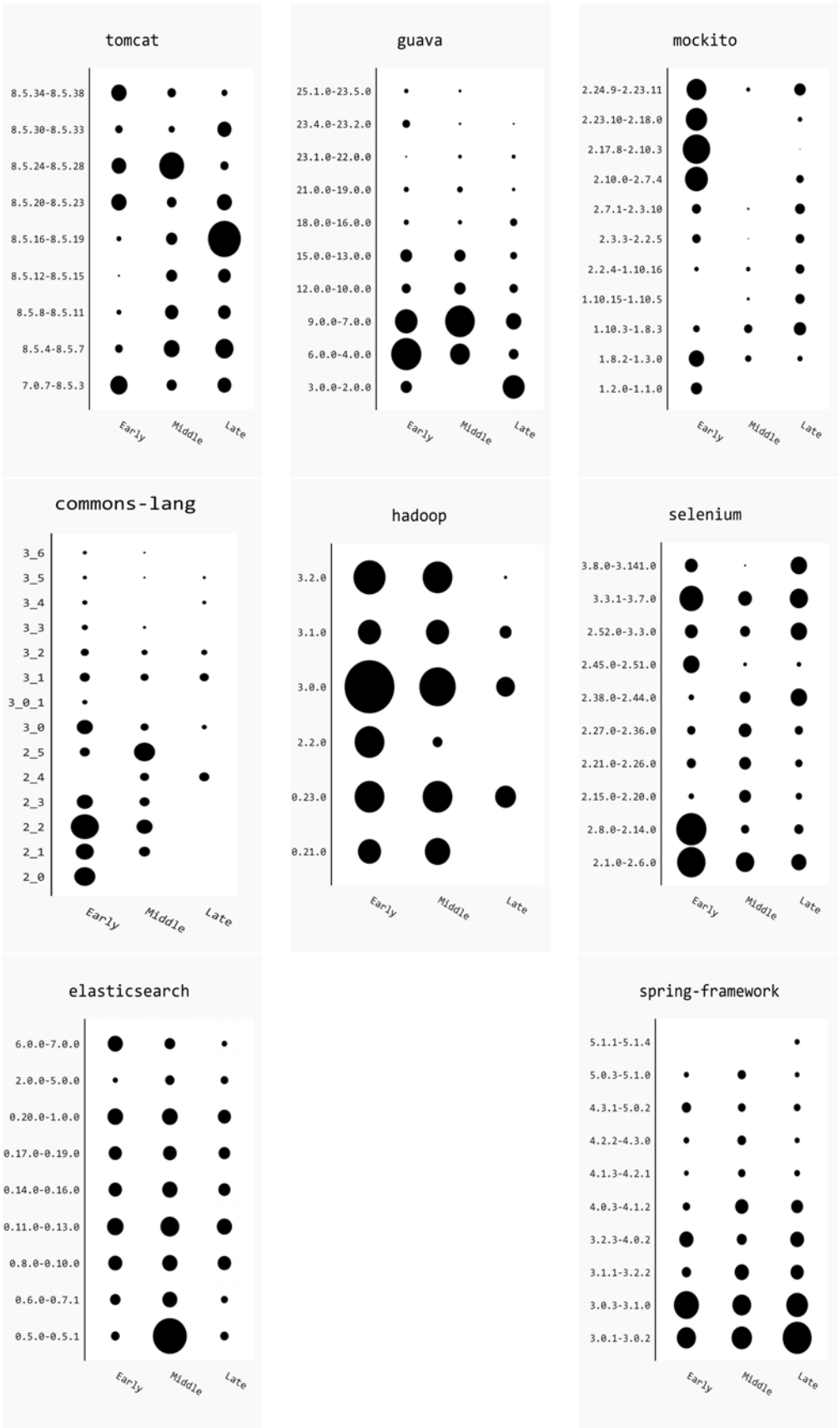


Figure 4: FICs in releases: early, middle and late stages

tributed to a higher emphasis on adding and updating features in those stages. The late stages are more focused on debugging and deployment efforts.

4.4. RQ3: FIC Interval

The 3rd RQ deals with the relation between Commit intervals and FICs. Table 2 shows that, on average, the interval (in minutes) for FICs is longer than that for regular Commits. The p-value of $< 2.2e^{-16}$ solidifies this difference as significant. This indicates that either a large amount of work relates to FICs (as shown by [5]), or that the developer introduces bug when they are away from the code for a long time.

The finding can help in preemptively detecting buggy code. Commits posted after a longer period than average can be given extra emphasis in reviews. Additionally, developers should be suggested not to disassociate from the code for a long time.

4.5. RQ4: FIC and Complexity Metrics

The last RQ observes whether FICs are related to the complexity metrics of software evolution: Line of Code (LoC) and Cyclomatic Complexity (CC). It can be seen in Table 2 that the average LoC of code where FICs occur is lower than that of regular Commits, with a p-value of $< 2.2e^{-16}$, making this difference significant. The result says that a lower LoC relates to buggy code. Hence the smaller, less busy components need to be given more emphasis in coding correctly and reviewing for bugs.

Next, as seen in Table 2, FICs occur in methods with significantly less CC than regular Commit, based on the $6.48e^{-10}$ p-value. A lower CC means that the tasks in methods are logically simpler. And yet, bugs, as statistically shown, tend to be introduced in such methods. Similar to LoC, this result prompts for a higher level of scrutiny when dealing with smaller and simpler methods.

Both of these findings support the evolution of FICs compared to complexity metrics. As described by Lehman's law of evolution[11], complexity rises as software evolves, hence increasing LoC and CC. Similarly, based on RQ1, FICs decrease in ratio in most cases, which is solidified by its inverse relation with the complexity metrics.

Table 2
Results for RQ3 and RQ4

Metric	Commit Type	Average	Standard Deviation	P-value
Interval	FIC	285.54	2027.59	$< 2.2e^{-16}$
	Regular	177.13	1113.37	
LoC	FIC	508.63	507.94	$< 2.2e^{-16}$
	Regular	636.6	760.61	
CC	FIC	3.49	3.64	$6.48e^{-10}$
	Regular	4.04	4.42	

5. Result Discussion

From the findings generated in this study, the following interpretations and applications can be estimated:

1. **Early bugs**: From both Research Questions (RQs) 1 and 2, it can be seen that bugs appear mostly in the early stages of versions and release cycles. This finding solidifies the intuition that early code tends to cause more bugs than later ones when maintenance starts to outrank development, and the software stabilizes. With this intuition proven through data, the finding can be applied to change the way software is developed. A more test-driven approach can be adopted in software projects from the beginning to mitigate the large influx of bugs.
2. **Comparative history**: By graphically extracting the evolution of FICs in software projects, the appearance of bugs can be historically analyzed. This history can unearth valuable insight, for example periods of time or certain releases where FICs peaked in number. These exceptions can be comparatively analyzed with other metrices related to the project. The metrices can range from code properties like components developed, design patterns used etc, or project aspects like type of assignment, assigned developer, developer turnover etc. The proponents may vary from project to project, hence the historical data of FICs can be used as a constant reference to such differing metrices.
3. **Intervals and bugs**: RQ3 provides insight into the correlation between Commit interval and FICs, showing the tendency of larger intervals causing bugs. This finding can be applied in project management, by monitoring the absences of developers. Developers who have been absent from the development process for longer periods should be assigned to tasks that are less sensitive and their work be reviewed more intensely. Furthermore, as also observed by Sliwerski et al. [5], large amount of changes in a single Commits that cause higher time for completion should be regulated for FICs.
4. **Software evolution and complexity**: The last finding demonstrates how FICs are correlated with line of code (LoC) and cyclomatic complexity (CC). These metrices, referred to as complexity metrices in the domain of software evolution, are important in understanding the evolution of FICs. As graphically shown in RQ1, FICs tend to decrease as the software evolves. On the other

hand, complexity increases with the software's age [11]. This indicates an inverted relationship between the two metrices, which is proven in RQ4 where FICs are found to be related with a larger LoC and CC in the code.

6. Conclusion

This study analyzes GitHub repositories to extract Fix-inducing Changes (FICs) — changes that introduce buggy code — and observes its evolution and characteristics. It is seen that FICs tend to occur in earlier versions and stages of releases. There is also a significant delay in posting FICs than regular Commits. Lastly, when relating with complexity metrics, FICs show up in code with less LoC and less CC than regular Commits. This corresponds with the decreasing FIC and increasing complexity of software evolution.

References

[1] M. M. Lehman, L. A. Belady, Program evolution: processes of software change, Academic Press Professional, Inc., 1985.

[2] M. Monperrus, Automatic software repair: a bibliography, ACM Computing Surveys (CSUR) 51 (2018) 1–24.

[3] N. Chauhan, Software Testing: Principles and Practices, Oxford University, 2010.

[4] T. Ackling, B. Alexander, I. Grunert, Evolving patches for software repair, in: Proceedings of the 13th annual conference on Genetic and evolutionary computation, 2011, pp. 1427–1434.

[5] J. Śliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes?, in: ACM sigsoft software engineering notes, volume 30, ACM, 2005, pp. 1–5.

[6] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, L. Bairavasundaram, How do fixes become bugs?, in: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ACM, 2011, pp. 26–36.

[7] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, O. Strollo, When does a refactoring induce bugs? an empirical study, in: 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, IEEE, 2012, pp. 104–113.

[8] A. Z. Sadiq, M. J. I. Mostafa, K. Sakib, On the evolutionary relationship between change coupling and fix-inducing changes (2019).

[9] S. F. Huq, A. Z. Sadiq, K. Sakib, Understanding the effect of developer sentiment on fix-inducing changes: An exploratory study on github pull requests, in: 2019 26th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2019, pp. 514–521.

[10] S. F. Huq, A. Z. Sadiq, K. Sakib, Is developer sentiment related to software bugs: An exploratory study on github commits, in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2020, pp. 527–531.

[11] M. M. Lehman, Laws of software evolution revisited, in: European Workshop on Software Process Technology, Springer, 1996, pp. 108–124.

[12] C. F. Kemerer, S. Slaughter, An empirical approach to studying software evolution, IEEE transactions on software engineering 25 (1999) 493–509.

[13] G. Xie, J. Chen, I. Neamtiu, Towards a better understanding of software evolution: An empirical study on open source software, in: 2009 IEEE International Conference on Software Maintenance, IEEE, 2009, pp. 51–60.

[14] G. Antoniol, V. F. Rollo, G. Venturi, Detecting groups of co-changing files in cvs repositories, in: Eighth International Workshop on Principles of Software Evolution (IWPSE'05), IEEE, 2005, pp. 23–32.

[15] S. Levin, A. Yehudai, Boosting automatic commit classification into maintenance activities by utilizing source code changes, in: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, ACM, 2017, pp. 97–106.

[16] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE transactions on software engineering 33 (2006) 2–13.

[17] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, N. Ubayashi, An empirical study of just-in-time defect prediction using cross-project models, in: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, 2014, pp. 172–181.

[18] Y. Weicheng, S. Beijun, X. Ben, Mining github: Why commit stops–exploring the relationship between developer's commit pattern and file version evolution, in: 2013 20th Asia-Pacific Software Engineering Conference (APSEC), volume 2, IEEE, 2013, pp. 165–169.

[19] H. Osman, M. Lungu, O. Nierstrasz, Mining frequent bug-fix code changes, in: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), IEEE, 2014, pp. 343–347.

[20] I. Neamtiu, J. S. Foster, M. Hicks, Understanding source code evolution using abstract syntax tree matching, in: Proceedings of the 2005 international workshop on Mining software repositories, 2005, pp. 1–5.

Using Rule Mining for Automatic Test Oracle Generation

Alejandra Duque-Torres[a], Anastasiia Shalygina[a], Dietmar Pfahl[a] and Rudolf Ramler[b]

[a]Institute of Computer Science, University of Tartu, Tartu, Estonia
[b]Software Competence Center Hagenberg GmbH, Hagenberg, Austria

Abstract

Software testing is essential for checking the quality of software but it is also a costly and time-consuming activity. The mechanism to determine the correct output of the System Under Test (SUT) for a given input space is called test oracle. The test oracle problem is a known bottleneck in situations where tests are generated automatically and no model of the correct behaviour of the SUT exists. To overcome this bottleneck, we developed a method which generates test oracles by comparing information extracted from object state data created during the execution of two subsequent versions of the SUT. In our initial proof-of-concept, we derive the relevant information in the form of rules by using the Association Rule Mining (ARM) technique. As a proof-of-concept, we validate our method on the Stack class from a custom version of the Java Collection classes and discuss the lessons learned from our experiment. The test suite that we use in our experiment to execute the different SUT version is automatically generated using Randoop. Other approaches to generate object state data could be used instead. Our proof-of-concept demonstrates that our method is applicable and that we can detect the presence of failures that are missed by regression testing alone. Automatic analysis of the set of violated association rules provides valuable information for localizing faults in the SUT by directly pointing to the faulty method. This kind of information cannot be found in the execution traces of failing tests.

Keywords

Software testing, test oracle, association rule mining, test oracle automation, machine learning methods in software testing

1. Introduction

Software testing is an essential activity for quality assurance in software development process as it helps ensure the correct operation of the final software [1]. However, software testing has historically been recognised to be a time-consuming, tedious, and expensive activity given the size and complexity of large-scale software systems [2]. Such cost and time involved in testing can be managed through test automation. Test automation refers to the writing of special programs that are aimed to detect defects in the System Under Test (SUT) and to using these programs together with standard software solutions to control the execution of test suites. It is possible to use test automation to improve test efficiency and effectiveness.

Software testing, automated or not, has four major steps: test case generation, predicting the outcomes of the test cases, executing the SUT with the test cases to obtain the actual outcome, and comparing the expected outcome against the actual outcome to obtain a verdict (pass/fail) [3]. In these steps there are two major challenges: find effective test inputs, *i.e.*, inputs that can reveal faults in the SUT, and determine what should be the correct output after execution of the test cases. The second challenge refers to the the *test oracle problem*. A test oracle is a mechanism that determines the correct output of SUT for a given input [4]. Although substantial research has been conducted to provide test oracle automatically, apart from model-driven testing, the oracle problem is largely unsolved.

Motivated by the above, we developed a method to derive test oracles based on information contained in object state data produced during the execution of the SUT. Object state data is the set of the values of all defined attributes of an object at a certain point of time. We assume that most programs have objects with a mutable state, and the execution of methods can modify the state of the program. The idea of using the state information roots in the assumption that relations contained in the state data when testing a new version of the SUT should remain unchanged as compared to a previous version.

Our proposed method employs Association Rule Mining (ARM). In our context, the purpose of ARM is to mine interesting relations in the state data of the SUT. ARM is an unsupervised machine learning (ML) method [5]. The algorithms used in ARM attempt to find relationships or associations between categorical variables in large transactional data sets [6]. In particular, we were interested in understanding whether the information provided by the resulting model can help to verify the correct operation of new versions of the SUT to which we normally apply existing tests for regression testing. More specifically, we wonder if we can detect and locate faults in new ver-

QuASoQ'20: 8th International Workshop on Quantitative Approaches to Software Quality, December 1, 2020, Singapore
duquet@ut.ee (A. Duque-Torres);
anastasiia.shalygina@gmail.com (A. Shalygina);
dietmar.pfahl@ut.ee (D. Pfahl); rudolf.ramler@scch.at (R. Ramler)
0000-0002-1133-284X (A. Duque-Torres); 0000-0003-2400-501X (D. Pfahl); 0000-0001-9903-6107 (R. Ramler)

sions of the SUT. We tackle our goal by answering the following research questions:

RQ1: *How effective is the rule mining approach?*. This research question investigates to what extent ARM is able to detect failures.

RQ2: *What information regarding fault localisation can the method offer?* This research question explores to what extend the information contained in association rules helps developers locate faults in the code.

In our experiments, we use the *Stack Class* of the Java Collection framework as SUT. This class was chosen as its state behaviour is well known and easy to manipulate.

2. Association Rule Mining

ARM is a rule-based unsupervised ML method that allows discovering relations between variables or items in large databases. ARM has been used in other fields, such as business analysis, medical diagnosis, and census data, to find out patterns previously unknown [6]. The ARM process consists of at least two major steps: finding all the frequent itemsets that satisfy minimum support thresholds and, generating strong association rules from the frequent derived itemsets by applying minimum confidence threshold.

A large variety of ARM algorithms exist. [7]. In our experiments, we use the Apriori algorithm from Python3 Efficient-Apriori library [8]. It is well known that the Apriori algorithm is exhaustive, that is, it finds all the rules with the specified support and confidence. In addition, ARM doesn't require labelled data and is, thus, fully unsupervised. Below we define important terminology regarding ARM:

Itemset: Let $I = \{X, \ldots, Y, Z\}$ be a set of different items in the dataset D. Itemset is a set of k different items.

Association rule: Consider a dataset D, having n number of transactions containing a set of items. An association rule exposes the relationship between the items.

Support: The support is the percentage of transaction in the dataset D that contains both itemsets X and Y. The support of an association rule $X \rightarrow Y$:

$support(X \rightarrow Y) = support(X \cup Y) = P(X \cup Y)$

Confidence: The confidence is the percentage of transactions in the database D with itemset X that also contains the itemset Y. The confidence is calculated using the conditional probability which is further expressed in terms of itemset support: $confidence(X \rightarrow Y) = P(Y|X) = support(X \cup Y)/support(X)$

Lift: Lift is used to measure frequency X and Y together if both are statistically independent of each other. The lift of rule $(X \rightarrow Y)$ is defines as $lift(X \rightarrow Y) = confidence(X \rightarrow Y)/support(Y)$.

A lift value one indicates X and Y appear as frequently together under the assumption of conditional independence.

3. Method

Figure 1 presents an overview of the method for rule-mining based tests oracles generation. Overall, the proposed method comprises two phases. *Phase I* is responsible for the ruleset generation, *i.e.*, the rule mining part. The output of this phase is the ruleset. *Phase II* is in charge of applying the ruleset to the new SUT versions. Thus, the output of this phase could be seen as a fault report for new SUT versions. Below we describe them in detail:

3.1. Phase I - Rule Set Generation

Phase I starts with the extraction of the state data. Then, feature selection and encoding are performed so that all the features become appropriate to use for the rule mining. The features should be encoded as categorical if there is a need. When all the required operations with the raw data are performed, the state data from the first version of SUT is received, and the rule mining process starts. After that, one gets a set of rules. All the process can be split into three main steps which are detailed below:

Step 1.1 - State data acquisition: This step comprises two activities:

Activity 1.1.1 (Produce test) is responsible for the generation of tests. To perform this activity, we use Randoop to generate unit tests automatically. Randoop is a popular random unit test generator for Java[1]. Randoop creates method sequences incrementally by randomly selecting a method call to apply and using arguments from previously constructed sequences. When the new sequence has been created, it is executed and then checked against contracts. Two important parameters that we use in our experiments are test limit and a random-seed. The test limit parameter helps to limit the number of tests in the test suite. The random seed parameter allows us to produce multiple different test suites since Randoop is deterministic by default. Therefore, these two parameters allow us to generate many test suites of different size containing various test cases.

Activity 1.1.2 (Execute the test suite) is responsible for state tracking and saving raw data. To track the states of the SUT while running the test suite and save it to the text file for later analysis, we built a special program that helps to track and save the information of the state of the SUT. We call this program *Test Driver*. The idea behind the Test Driver is that the methods of the SUT

[1] https://randoop.github.io/randoop/

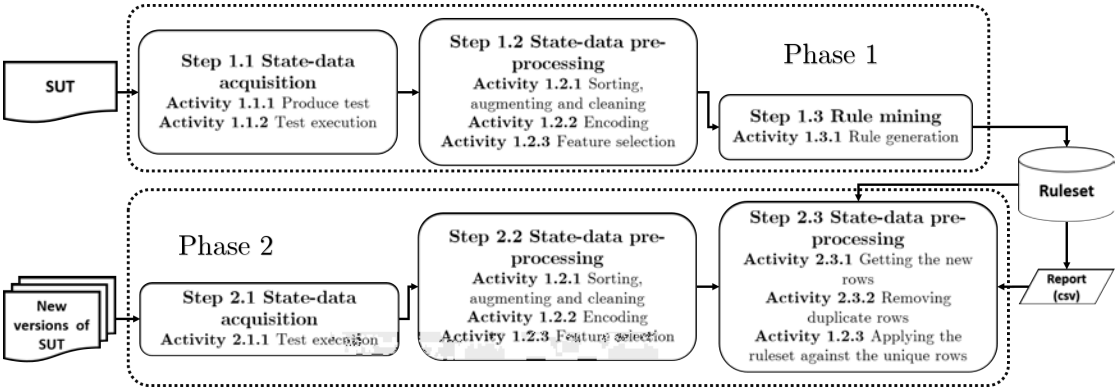


Figure 1: Overview of the method for rule-mining based test oracles generation

can be logically divided into two categories: methods that change a state of the class instance of the SUT, and methods that reflect the state. These methods are so-called state methods. The test driver tracks and stores the information returned by state methods if they are called immediately after the test case execution. The information is saved in a CSV file.

Step 1.2 - State data pre-processing: This data pre-processing step is made up of three activities:

Activity 1.2.1 (Sorting, augmenting and cleaning) is responsible for ensuring that the data is correct, consistent and useable. This activity has three main functions: *sort*, *aug*, and *clean*. The *sort* function is responsible for sorting the dataset based on the TestId and InstanceId, this is done to find interesting sequences in the data, and be able to model those sequences. When the dataset is ordered, it is possible to add more information. *e.g.*, it is possible to add characteristics that indicate the previous state. This is made trough *aug* function. The *clean* function removes the rows that are no needed or inconsistent rows.

Activity 1.2.2 (Encoding) is in charge of preparing the data according to the requirements of the rule mining algorithm. For example, Apriori [9], which is the algorithm used in this paper, works only with categorical features. Thus, Activity 1.2.2 categorises and generalises the numerical inputs into string representations.

Activity 1.2.3 (Feature selection) is an extra activity which allows to explore the different performance of the method when different features are used. For instance, in this paper we used five different datasets which contain different numbers of features.

Step 1.3 - Rule mining: This step is responsible for generating the set of rules by using the Apriori ARM algorithm.

3.2. Phase II

Phase II is in charge of applying the ruleset to the new versions. Like Phase I, Phase II comprises three steps

Step 2.1 - State data acquisition: Unlike Step 1.1, this step has only one activity, the test execution activity. In phase one, we assume that the first version of the SUT is correct; then, we build test suites using Randoop and the test driver. In Step 2.1, the same tests generated in Activity 1.1.1 are used to test the new versions of the SUT.

Step 2.2 - State data pre-processing: This step performs the same activities as Step 1.2 to prepare the state dataset of the new SUT version.

Step 2.3 - Apply the ruleset to the new SUT version: This step comprises three activities. *Activity 2.3.1* is in charge of comparing and selecting the rows that are exclusively different from the first version of the SUT. *Activity 2.3.2* removes duplicate rows. This is done for optimisation. If two or more rows are the same, then they will have the same results.

Activity 2.3.3 (apply the ruleset against the new unique rows) is responsible for applying the rule-set against the new unique rows. A rule will have two sides, *e.g.*, let's consider the rule $(X \rightarrow Y)$, in this rule, X is a left-hand side (LHS) of the rule, and Y is a right-hand side (RHS). The procedure for applying the ruleset against the new unique rows works as follows: $i)$ pick a rule from the set of rules, $ii)$ use LHS, $iii)$ select values which match LHS, $iv)$ check whether these values match the RHS, $v)$ save the values which don't match, and $vi)$ repeat steps $i - v$ for every rule. In the end, we know that whenever the state dataset contains values that violate rules, the new version of the SUT is not correct. Since only those rows in the state dataset corresponding to the modified SUT that are different from rows in the state dataset corresponding to the unmodified (correct) SUT have the potential to violate rules, it makes sense to only analyze the different rows.

4. Results

The full set of data generated during our experiments as well as all scripts can be found in our GitHub repo[2]. In our experiment, we use the Stack Class of the Java Collection framework as SUT. This class was chosen as its state behaviour is well known and easy to manipulate. The Stack implementation contain seven public methods: *push()* which puts an item on the top of the stack, *pop()* removes and returns an item from the top, *clear()* clears the stack, *peek()* returns an item from the top of the stack without deleting it, *isEmpty()* tests if stack is empty, *size()* returns size of the stack and, finally, *toString()* returns a string representation of the object.

The methods *push(), pop()* and *clear()* modify instances of the Stack class when they are called. On the other hand, *peek(), isEmpty(), size()* and *toString()* provide information about the Stack object state when they are called and, thus, *peek(), isEmpty(), size()* and *toString()* are state methods. The test driver for Stack class creates an instance of the Stack class and contains public methods *push(), pop()* and *clear()* which call the original *push(), pop()* and *clear()* using the instance of the Stack class. Additionally, the driver implements *peek(), isEmpty(), size()* and *toString()* but these methods are private. Furthermore, the driver has a method for writing states to the CSV file. This method is used whenever *push(), pop()* or *clear()* methods are called during test execution. This set-up allows us to run Randoop test suite generation not on the original Stack class but on the driver class, where the public methods are the ones that modify the Stack objects. Thus, only *push(), pop(),* and *clear()* methods are called in the test sequences, and the state data captured by *peek(), isEmpty(), size()* and *toString()* is saved to a CSV file.

4.1. Phase I - Rule Set Generation

Step 1.1 - State data acquisition: Two different reports are the output of this step, *Test Report (pass / failed)* and *State Report*. The regression testing generates the *Test Report* . Test Driver generates the *State Report*. The *State Report* provides seven main features *testID*, *intanceID*, *size*, *isEmpty*, *peek_obj_type*, *pushInput*, and *calledMethod*. The features *testID* and *instanceID* provide an identification of the test generate by Randoop. One test can have multiples instances. Thus, *instanceID* is the identification of those instance belonging to the same test. The feature *size* tells the size of the Stack, and is a numerical feature. *isEmpty* feature contains the values "True" or "False". *isEmpty* is "True" when the Stack is empty, otherwise it will be "False". *peek_obj_type* tells us the element at the top of the Stack. *calledMethod* tells us which method was called, *i.e.,* push, pop, or clear.

Table 1
Summary of the support and lift metrics of the rule-set extracted from the Stack class data using ARM with different number of feature

DS†	NR†	Support			Lift		
		Max	Mean	Min	Max	Mean	Min
FS-3	14	0.403	0.381	0.283	2.539	2.391	1.946
FS-4	36	0.337	0.273	0.225	2.539	2.475	2.243
FS-8	439	0.269	0.227	0.205	3.808	3.404	2.545
FS-9	676	0.305	0.231	0.201	4.392	3.237	2.169
FS-10	1450	0.279	0.225	0.203	4.404	3.492	2.442

†Data Set, †Number of rules

Step 1.2 - State data pre-processing: In this step, we sorted the dataset based on the *testID* and *intanceID*, this is done to find the sequence of Stack size, and be able to model those sequences. When the dataset is ordered, it is possible to add more information. For example, it is possible to add characteristics that indicate the previous state. To distinguish from the original features, we add a *_p* at the end of the feature name, this means "previous". Then, the previous states are named as: *size_p*, *isEmpty_p*, *peek_obj_type_p*, *calledMethod_p*. Then, we removed the unnecessary rows, this is, rows whit not state information. For instance, the *Test Driver* writes a rows with the name *"Constructor"* in the feature *calledMethod* which indicates that a new Stack was created. This information it is not related to the state, thus, should be dropped. Finally, we encoded the features should be encoded as categorical if there is a need. In the context of our data, we encoded the feature *size*, and *size_p* since they are not categorical features.

We create five different datasets which contain different numbers of features. The created dataset are named with the prefix FS, which stands for *Feature Selection*. To distinguish the different dataset, they have been named with the number of characteristics that were used, *i.e.,* FS-X where X is the number of features. The datasets created are the following: ***FS-3*** comprises the features *size*, *isEmpty*, and *peek_obj_type*. ***FS-4*** contains the features used in FS-3 plus *called_Method*. ***FS-8*** comprises the features used in *FS-4* and their previous values, which are *size_p*, *isEmpty_p*, *peek_obj_type_p*, and *calledMethod_p*. ***FS-9*** uses the FS-8 features and the feature *pushIputh*. Finally, ***FS-10*** uses all the features.

Step 1.3 - Rule mining: We apply the Apriori algorithm with minimal support and maximum confidence thresholds, *i.e.*, 0.2 and 1, to each dataset. Table 1 provides a comparison between the number of rules, support and lift values for each dataset. As per Table 1, we can observe that the average support ratio decrease when the number of features used is increased. The average is closer to the threshold value set up. Table 1 also shows

[2]https://github.com/aduquet/Using-Rule-Mining-for-Automatic-Test-Oracle-Generation

the number of rules that can be generated does not have linear behaviour since it depends on the number of items belonging to each feature. From Table 1 column lift, we can observe that the lift ratio is increasing when the number of features used is increased. This is the opposite of the support ratio. A lower lift ratio means that the probability of occurrence of a determinate rule is weak since the LHS and RHS are near to be independent between themselves.

4.2. Phase II

Step 2.1 - State data pre-processing: In this step, we created new versions of the Stack class. We introduce defects to the class under test. For example, in the Stack class we use three state methods: *peek()*, *size()* and *isEmpty()*. We modify each of them individually and also make all possible combinations of these modifications. Table 2 summarises the main modifications made to the SUT, and it provides the meaning of the terminology used to refers to the different version of the SUT. These modifications are done on purpose and manually, as we want to understand the potential of ARM to detect and locate faults using the state information of the SUT.

Table 2
Summary of the modifications injected to the Stack class

Modification	$peek()$	$isEmpty()$	$size()$
Mod_{1-p}	✓	–	–
Mod_{2-e}	–	✓	–
Mod_{3-s}	–	–	✓
Mod_{4-pe}	✓	✓	–
Mod_{5-ps}	✓	–	✓
Mod_{6-es}	–	✓	✓
Mod_{7-pes}	✓	✓	✓

Some of the modifications affect the state such that it will be quite easy to detect that something is wrong. For example, *isEmpty()* method is modified such that it returns *isEmpty=True* in the cases when size of the Stack class instance is 2 or 0. Thus, we would get an obviously faulty state *size="2.0"*, *is_empty="true"*. The modification of *peek()* will not return the object on the top of a Stack class instance but the previous one in the cases when stack size is greater or equal than 2. Modification of *size()* would return incorrect size for the Stack class instances that contain one or more objects. Thus, the states would look like the correct ones, and the dataset would not contain faulty-looking rows.

Step 2.2 - State data pre-processing: In this step, the same process of Step 1.2 was performed on the new data.

Step 2.3 - Apply the rule-set to the new SUT versions: We applied the ruleset against the state data of new versions by following the activities described in Section 3.2.

4.3. RQ1: How effective is the rule mining approach?

Table 3 summarises the not modified version and the seven modifications of the Stack class regarding the results obtained by the regression test, columns Regression test (pass / failed), and the information obtained during the test execution generated by the Test Driver, *i.e.,* the State Data. We notice from Table 3 that some datasets that correspond to modifications, *e.g.,* Mod_{2-e}, Mod_{4-pe}, Mod_{6-es}, and Mod_{7-pes}, do not have the same number of rows as in the No-Mod dataset. This is because some tests from the test suite are failed during the test execution the state in these cases will not be written to the CSV file. When no tests from the suite are failed, all the states will be written to the file. Therefore, the number of rows will be equal to the Not-Mod data since we execute the same test suite both for the Not-Mod and for the modified data extraction, *e.g.,* Mod_{1-p}, Mod_{3-s}, and Mod_{5-ps}.

As Table 3 column "Regression test" shows, only four of seven modifications have failed tests (Mod_{2-e}, Mod_{4-pe}, Mod_{6-es}, and Mod_{7-pes}). We see that the number of failed tests is the same for Mod_{2-e} and Mod_{4-pe} datasets. Also, the number of failed tests are the same for Mod_{6-es} and Mod_{7-pes}. The state data columns also shows the same behaviour, but regarding the number of rows generated. The common aspect between all these datasets is *isEmpty* method modification. In particular, Mod_{2-e} and Mod_{4-pe} have 1452 failed test. The Mod_{4-pe} modification is the combination of the modified methods *peek* and *isEmpty*. However, it seems that the regression test spots the fault related to *"isEmpty"* only. Furthermore, the Mod_{1-p}, which is the modification of Peek, none regression tests failed. This fact confirms the regression tests failed of Mod_{4-pe} are belonging to *isEmpty* modification only.

Same as Mod_{1-p}, none regression test failed in Mod_{3-s}. In this modification *"size"* method is modified. The number of tests failed in Mod_{7-pes} and Mod_{6-es} are different from Mod_{2-e} and Mod_{4-pe}. *Are the regression tests spotting faults in the other modified methods, i.e., "peek" and "size" when combined in this way?* The modification of *size()* returns the incorrect size for the Stack class instances that contain one or more objects, *i.e.,* when the size of the Stack is greater than 0, the modification would return the correct size plus one. For instance, if the $size = 1$, the modified version will return $size = 2$. That is why Mod_{6-es} increase the number of failed tests because the size modified method increases the number of $size = 2$, then triggers the modified *isEmpty()* when the size of the Stack class is 2 or 0. From the Table 3 we can ask ourself, *why the regression tests are failing only in the isEmpty() method?* When analysing in detail the report provided by regression test, we can find that during the execution of the test, there is an exception

Table 3
Summary of the no modified version and the seven modifications of the Stack class regarding the results obtained by the regression test, and the information obtained during the test execution generated by the test driver

Dataset	Regression test		State data																	
	Pass	Failed	Total # of rows	Total # of different rows	Total # of UR†					Total # of different UR† (hereinafter, new unique rows)					Total # of new UR† that violate rules				
					FS-3	FS-4	FS-8	FS-9	FS-10	FS-3	FS-4	FS-8	FS-9	FS-10	FS-3	FS-4	FS-8	FS-9	FS-10
No-Mod	2037	0	71604	0	47	76	320	320	320	0	0	0	0	0	0	0	0	0	0
Mod_{1-p}	2037	0	71604	11882	45	73	232	308	398	28	43	97	166	189	0	0	0	38	38
Mod_{2-e}	585	1452	37786	6390	25	33	108	108	108	1	2	20	20	20	1	2	20	20	20
Mod_{3-s}	0	0	71604	43405	47	76	320	320	320	46	74	237	237	237	0	0	19	19	19
Mod_{4-pe}	585	1452	37786	7792	26	33	106	119	133	14	20	46	59	66	1	2	31	42	49
Mod_{5-ps}	2037	0	71604	43405	44	73	232	308	398	44	71	156	232	279	0	0	19	50	57
Mod_{6-es}	25	2012	7160	5497	5	9	17	17	17	5	7	8	8	8	1	2	6	6	6
Mod_{7-pes}	25	2012	7160	5497	3	7	14	16	17	3	5	6	8	8	1	2	4	7	7

†Unique rows

that checks if the Stack is empty or not. By making the modification in the *IsEmpty()* method, we generate the situation where the exception is generated, thus allowing the test to not finish its execution and report it as a failure.

Table 4
Percentage of new unique rows (among all-new unique rows) that violate at least one rule

Dataset	% of new unique rows that violate rules				
	FS-3	FS-4	FS-8	FS-9	FS-10
No-Mod	-	-	-	-	-
Mod_{1-p}	0	0	0	22.89	20.11
Mod_{2-e}	100	100	100	100	100
Mod_{3-s}	0	0	8.02	8.02	8.02
Mod_{4-pe}	7.14	10	67.39	71.19	74.24
Mod_{5-ps}	0	0	12.18	21.55	20.43
Mod_{6-es}	20	28.57	75	75	75
Mod_{7-pes}	33.33	40	66.67	87.50	87.5

In Table 3, the column "Total # of unique rows" indicates that the number of unique rows increases when the number of features increases. This is because by adding more features, we increase the heterogeneity in the data. The number of new unique rows refers to those rows that are different from the unmodified SUT version. The idea of using state information is based on the assumption that the relationship in state when testing a new version of the SUT must remain unchanged or must not change significantly. Therefore, we can conclude that the rows of the modified versions that are different from the unmodified version are failures. Up to this point our method is capable of detecting failures without the need to use ARM.

We are interested in understanding whether the information provided by the resulting ARM model would have confirmed the failure detection that is already given when identifying new rows in the state dataset of a modified SUT. Therefore, we measured the proportion of rules that are violated by new rows. In Table 3, the column "Total # of new unique rows that violate rules" shows the total number of new unique rows that violate at least one rule. It turns out that only for the *isEmpty()* modification always all rows also trigger a rule violation. It is less often the case when other methods are modified (either individually or in combination). Only when the largest feature sets (FS-9 and FS-10) are used, there is always at least one new row in the dataset that also violates at least one rule. This result is weaker than what we can already see by just looking at new rows but, recalling that none of the regression tests failed when only methods *size()* and *peek()* were modified, rule violations seem to occur in a more balanced way. This let us hope that we might be able to exploit this when answering research question RQ2.

4.4. RQ2: What information regarding fault localisation can the method offer?

Regarding the first research question, we concluded that failure detection effectiveness improves by comparing state data even without using ARM. However, neither analyzing the traces of failing tests (all of them failed when executing the *pop()* nor inspecting the information provided in the new unique rows provided any helpful information that would guide fault localization. Therefore, we set our hopes in a systematic analysis of the rules that are violated by new unique rows.

As a starting point for fault localization, it is necessary that at least one rule be violated by at least one new single row. Table 5 summarises the total number of rules generated per dataset, and the number of rules violated among all the rules generated. As Table 5 shows, from FS-8 to FS-10, more than a hundred rules need to be analysed to be able to localise the fault. To reduce and optimize the number of rules, we construct a rule hierarchy. Let us consider the following set of rules: *(i)* $A, B, C \rightarrow D$ *(ii)* $A, B \rightarrow D$ *(iii)* $C, B \rightarrow D$, and *(iv)* $C \rightarrow D$. The rule *(i)* contains the same items of rules *(ii)*, *(iii)*, *(iv)* in implicitly. Therefore, having only rule *(i)* is sufficient for

Table 5
Total number of rules generated per-each dataset, and the number of rules violated among all the rules generated

DS	RG†	Total # of rules Violated (the row match with LHS but NOT with RHS)													
		Mod_{1-P}		Mod_{2-e}		Mod_{3-s}		Mod_{4-pe}		Mod_{5-ps}		Mod_{6-es}		Mod_{7-pes}	
		RV$^{\perp}$	OSR*	RV$^{\perp}$	OSR*	RV$^{\perp}$	OSR*	RV$^{\perp}$	OSR*	RV$^{\perp}$	OSR*	RV$^{\perp}$	OSR*	RV$^{\perp}$	OSR*
FS-3	14	0	0	3	-	0	0	3	-	0	0	3	-	3	-
FS-4	36	0	0	4	-	0	0	4	-	0	0	4	-	4	-
FS-8	439	0	0	73	16	95	95	73	56	95	95	113	98	113	98
FS-9	676	12	10	142	61	95	95	224	191	307	297	297	269	297	272
FS-10	1450	12	10	224	65	285	285	236	191	297	397	314	269	325	272

†Rules generated, $^{\perp}$ Number of rules violated * Optimal set of rules

interpretation purposes.

Once the number of rules is reduced, we can analyse them more efficiently. The key to being able to analyse the set of rules violated is to answer the following question: *Which item is violating the rule?;* A rule violation occurs when some items in a row match the LHS of the rule, but at least one items in the same row do not match the RHS of the same rule. Based on the above, the item that generate the violation of the rule must be present more frequently in the RHS than in the LHS of the set of violated rules. Then, to locate the fault, we quantify the occurrence of the main methods in the LHS with respect to their occurrence in RHS. If the LHS/RHS ratio approaches zero, the fault has been located. Let's consider Table 6 the set of rules violated by Mod_{1-p} of **FS-9**, the total number of rules violated are 10. Please note the following nomenclature: *A: peek, B: isEmpty, C: size. D: calledMethod,* and *E: inputPush.* Let's compute the relation LHS/RHS per each item, *i.e., A, B, D* and *E.* $A = 3/7 = 0.428$, $B = 2/2 = 1$, $D = 6/2 = 3$, and $E = 6/3 = 3$. The value closest to zero is $A = 3/7 = 0.428$, which corresponds to the peek method. We can conclude that the fault has been located, and that it corresponds to the peek method.

Table 7 reports modifications located using our approach. According to the results shown in the table, six out of seven modifications could be located when using nine or ten features. An interesting aspect that stands out from the results is the localization of the modifications where the *isEmpty* method is involved. When using fewer features, *e.g.,* FS-4, the *isEmpty* modification is always located, which is not the case when using more features.

Note that our method can only point out one fault per analysis. Therefore, one must apply the analysis repeatedly. Once a fault has been located, it must be removed, the tests must be run again, and if there are still new unique rows in the state dataset, we must try to spot the next fault. Thus, if we had started out with the case where all three methods were modified, and we had used models with 8, 9, and 10 features, we would have first located the fault in *(*with all feature sets), then in *peek()* (with FS-9 and FS-10), and then in *isEmpty()* (with FS-8).

Table 6
Set of violated rules of Mod_{1-p} with FS-9

LHS	RHS
D = push, E = objType	***A = objType***
E = objType	*D = push,* ***A = objType***
B = True, E = objType	***A = objType***
D = push	*B = True,* ***A = objType***
D = push, B = False, E = objType	***A = objType***
E = objType	*D = push, B = False,* ***A = objType***
D = push, ***A = objType***	*E = objType*
D = push, B = False, ***A = objType***	*E = objType*
D = push, ***A = objType***	*E = objType, B = False*
E = objType	***A = objType***

Table 7
Fault localised using ARM approach

Modification "Bug"	Modification localised by ARM				
	FS-3	FS-4	FS-8	FS-9	FS-10
Mod_{1-p}	✗	✗	✗	p	p
Mod_{2-e}	-	e	e	pushInput	pushInput
Mod_{3-s}	✗	✗	s	s	s
Mod_{4-pe}	✗	e	method	p	p
Mod_{5-ps}	✗	✗	s	s	s
Mod_{6-es}	-	e	s	s	s
Mod_{7-pes}	-	e	s	s	s

5. Threats to Validity

In the context of our proof-of-concept validation, two types of threats to validity are most relevant: threats to internal and external validity.

With regards to internal validity, it is not fully clear in which situations the iteratively applied heuristic that we propose to localize faults is successful, i.e., points to a rule element that carries useful information. It is clear that our method only can capture behaviour of the program that has not been changed and, thus, behaves equal at the level of granularity at which we capture object states. In addition, our positive results might depend on the number and type of faults injected. A more systematic and more comprehensive analysis is required to explore the limitations of our method with regards to both failure exposure and fault localization.

With regards to external validity our study is rather limited since we only use one well-understood class in our experiments. Thus, the actual scope of effectiveness of our proposed method for object-oriented programs is yet to be determined.

6. Related Work

A good overview of methods proposed to automatically generate test oracles can be found in [2] and [4]. As far as we know, ARM has not yet been used for this purpose by other researchers. A field that has recently received attention in the context text oracle generation is metamorphic testing as metamorphic relations can be used as test oracles[10].

7. Conclusion

We presented a new ARM-based method for the detection and localization of faults using the state data from SUT executions. In our proof-of-concept, we used the Stack class from the Java collection framework as the SUT. To test our method, we generated seven faulty versions of the SUT. The results obtained have shown that our approach is capable of detecting failures and locating faults. The ARM-approach mainly benefits fault localization.

An advantage of our method is that our method tested not on the SUT with only integer inputs and outputs but on the class under test where we can have any type of inputs, and the state data consequently is also of mixed types of data. Thus, our method can be generalized for inputs of any type, not only for integers. It removes some limitations on the type of SUT that can be analyzed.

One of the weaknesses of our method is the need of a test driver that we use to extract state data during the test suite execution. To generate the test driver for the SUT, we have to identify the state extracting methods manually. For efficiency reasons, it would be better to have an automatic identification of state extracting methods. Unfortunately, there is no simple way to do this. Also, in the case of the manual identification, for some classes, it may not be so clear what methods should be marked as state extracting methods.

Given the limitations of our study, more experiments have to be conducted to empirically test our proposed method for fault detection and localization. We are currently focusing on extending our experiments in two directions. First, we will add more kinds of fault injections to test the sensitivity of our method with regards to the type of faults in a program. We will systematize this by using mutation. Second, we will apply our proposed method to more classes in the Java collections framework and beyond.

Acknowledgments

This research was partly funded by the Estonian Center of Excellence in ICT research (EXCITE), the IT Academy Programme for ICT Research Development, the Austrian ministries BMVIT and BMDW, and the Province of Upper Austria under the COMET (Competence Centers for Excellent Technologies) Programme managed by FFG, and by the group grant PRG887 of the Estonian Research Council.

References

[1] T. M. Abdellatif, L. F. Capretz, D. Ho, Software analytics to software practice: A systematic literature review, in: 2015 IEEE/ACM 1st Int'l Workshop on Big Data Software Engineering, 2015, pp. 30–36.

[2] R. Braga, P. S. Neto, R. Rabêlo, J. Santiago, M. Souza, A machine learning approach to generate test oracles, in: Proc. of the XXXII Brazilian Symp. on Softw. Eng., SBES '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 142–151.

[3] K. Patel, R. M. Hierons, A partial oracle for uniformity statistics, Softw. Quality Journal 27 (2019) 1419–1447.

[4] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, S. Yoo, The oracle problem in software testing: A survey, IEEE Trans. on Softw. Eng. 41 (2015) 507–525.

[5] L. Zhou, S. Yau, Efficient association rule mining among both frequent and infrequent items, Computers and Mathematics with Applications 54 (2007) 737 – 749.

[6] S. K. Solanki, J. T. Patel, A survey on association rule mining, in: 2015 Fifth Int'l Conf. on Advanced Computing Communication Technologies, 2015, pp. 212–216.

[7] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: Proc. of the 20th Int'l Conf. on Very Large Data Bases, VLDB '94, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994, p. 487–499.

[8] G. McCluskey, Efficient-apriori documentation, 2018.

[9] A. Bhandari, A. Gupta, D. Das, Improvised apriori algorithm using frequent pattern tree for real time applications in data mining, Procedia Computer Science 46 (2015) 644 – 651.

[10] B. Zhang, et al., Automatic discovery and cleansing of numerical metamorphic relations, in: Proc. 35th IEEE International Conference on Software Maintenance and Evolution (ICSME 2019), 2019, pp. 235–245.

An Industrial Case Study on Fault Detection Effectiveness of Combinatorial Robustness Testing

Konrad Fögen, Horst Lichter

Research Group Software Construction, RWTH Aachen University, Aachen, Germany
https://www.swc.rwth-aachen.de

Abstract
Combinatorial robustness testing (CRT) is an extension of combinatorial testing (CT) to separate test suites with valid and strong invalid test inputs. Until now, only one controlled experiment using artificial test scenarios was conducted to compare CRT with CT. The results indicate advantages of CRT when much exception handling is involved. But, it is unclear if these advantages are also valid in the real-world. In this paper, we present the results of a case study conducted to compare the fault detection effectiveness of CRT and CT by testing an industrial system with 31 validation rules and 13 injected faults.

Keywords
Software Testing, Combinatorial Testing, Robustness Testing

1. Introduction

Robustness is an important property of software. It describes "the degree to which a system [...] can function correctly in the presence of [invalid inputs]" [1]. Invalid inputs are caused by external faults, i.e. faults in other systems or made by users interacting with a system. Examples are inputs to the system under test (SUT) that contain invalid values like a string value when a numerical value is expected, or invalid value combinations like a begin date which is after the end date. When invalid inputs remain undetected, they can propagate to failures in the SUT resulting in abnormal behavior or crashes [2].

Developers attempt to improve robustness of systems by implementing exception handling (EH) to detect and recover from invalid inputs. Unfortunately, EH is itself a significant source of faults (cf. [3, 4]). Therefore, it is important to test the exceptional behavior as well.

Combinatorial testing (CT) is a black-box test method that is based on an input parameter model (IPM) [5]. When considering the exceptional behavior, an IPM must describe invalid values and invalid value combinations that trigger EH. Unfortunately, invalid values and invalid value combinations can cause *input masking* (cf. [6, 7, 8]). When a SUT is stimulated with an invalid input, the EH is expected to detect it, to respond with an error message, and to terminate the SUT without resuming the normal behavior. Consequently, the remaining values and value combinations of the test input remain untested as they are masked.

To avoid input masking, combinatorial robustness testing (CRT) is developed as an extension to CT using a robustness input parameter model (RIPM) being an extension of an IPM with additional semantic information to annotate values and value combinations as invalid [7]. With this semantic information, valid test inputs can be selected which do not cover any invalid value or invalid value combination. Further on, strong invalid test inputs can be selected which contain exactly one invalid value or one invalid value combination.

Due to the separation of valid and strong invalid test inputs, the input masking effect can be avoided when testing the normal behavior and the exceptional behavior. However, in comparison to CT which does not separate valid and strong invalid test inputs, CRT requires effort to model the additional semantic information.

Despite the presence of input masking, CT can still be effective in detecting faults as a previous controlled experiment indicates [8]. Nevertheless, the fault detection effectiveness (FDE) of CT decreases for systems with much EH. Even for high testing strengths and large test suites, the FDE of CT deteriorates. For systems with much EH, CRT is a promising approach that can achieve a higher FDE while requiring fewer test inputs than CT [7]. For systems with little EH, CRT is at least as effective as CT.

Although, the current assessment is solely based on one controlled experiment with artificial test scenarios (cf. [7]). Therefore, our objective is to further compare CRT with CT guided by the following two research questions.

RQ 1 Is the CRT test method applicable in real-world test scenarios?

RQ 2 How does the CRT test method compare with CT in real-world test scenarios?

QuASoQ 2020: 8th International Workshop on Quantitative Approaches to Software Quality, December 01, 2020, Singapore
foegen@swc.rwth-aachen.de (K. Fögen);
lichter@swc.rwth-aachen.de (H. Lichter)
0000-0002-3440-1238 (H. Lichter)

To answer these research questions, we conducted a case study. According to Kitchenham et al. [9], a case study helps to evaluate the benefits of methods and tools in industrial settings. When applied to compare methods and tools, a case study is of explanatory nature "seeking an explanation of a situation or a problem" [10]. As Runeson & Höst state, a case study "will never provide conclusions with statistical significance" [10]. But it can provide sufficient information to help you judge if specific technologies will benefit your own organization or project" [9]. Since a case study has, by definition, a higher degree of realism than a controlled experiment [10], a case study that compares CRT with CT can provide additional insights that complement and extend the findings of the previously conducted controlled experiment.

The paper is structured as follows. Section 2 introduces basic concepts of CT and CRT. Related work is discussed in Section 3. Next, the design of the case study is introduced (Section 4) and its results are presented (Section 5). Afterwards, threats to validity are discussed (Section 6) before the paper is concluded in Section 7.

2. Background

In the following, CT and CRT are briefly introduced. For more information, please refer to [11, 5, 7].

2.1. Combinatorial Testing

CT is a black-box test method [5]. It is based on an **input parameter model** (IPM) which declares n parameters and each parameter is associated with a non-empty set of values. A **schema** is a set of parameter-value pairs for d distinct parameters [12]. A schema with $d = n$ parameter-value pairs is a **test input**. A schema a covers another schema b if and only if schema a includes all parameter-value pairs of schema b.

Real-world systems are often constrained and certain values should not be combined to schemata and test inputs [5]. These schemata are irrelevant because they are not of any interest for the test. Test inputs that cover irrelevant schemata are irrelevant as well and their test results have no informative value. Hence, they should be excluded from testing.

Constraint handling is often used to exclude irrelevant schemata [13]. Therefore, irrelevant schemata are explicitly modeled by a set of logical expressions (called **exclusion-constraints**). A schema is **relevant** if it satisfies all exclusion-constraints. A schema is **irrelevant** if at least one exclusion-constraint remains unsatisfied.

A **coverage criterion** is a condition that must be satisfied by a test suite. A **test selection strategy** describes how values are combined to test inputs such that a given coverage criterion is satisfied [11]. Test suites resulting from a test selection strategy that supports constraint handling, e.g. `IPOG-C` [13], satisfy the **t-wise relevant coverage criterion**. This criterion is satisfied if the relevant test inputs of a test suite cover all relevant schemata of degree $d = t$ that are described by an IPM [11, 5].

2.2. Combinatorial Robustness Testing

To avoid input masking, CRT is developed as an extension to CT that separates valid and invalid test inputs [7]. To better separate the concepts, we say that CT relies on IPMs while CRT relies on **robustness input parameter models** (RIPM). A RIPM contains additional **error-constraints** which is another set of constraints to annotate relevant schemata as invalid. A relevant schema is also a **valid schema** if it satisfies all error-constraints. A relevant schema is an **invalid schema** if at least one error-constraint remains unsatisfied. Further on, an invalid schema is a **strong invalid schema** if exactly one error-constraint remains unsatisfied.

Test selection strategies like `ROBUSTA` [7] not only consider exclusion-constraints to exclude irrelevant schemata, they also consider error-constraints and exclude invalid schemata from valid test inputs. Further on, strong invalid test inputs are selected such that each invalid value and invalid value combination that is modeled by error-constraints appears in strong invalid test inputs.

Valid test inputs are selected to satisfy **t-wise valid coverage**. The t-wise valid coverage criterion is an extension of the t-wise relevant coverage criterion. It is satisfied if all valid schemata with a degree of $d = t$ that are described by a RIPM are covered at least once by a valid test input.

Strong invalid test inputs are selected to satisfy **b-wise strong invalid coverage** where b denotes the robustness interaction degree. Without robustness interaction ($b = 0$), the coverage criterion is called single error coverage (cf. [11, 7]). It is satisfied if each invalid schema that is described by an error-constraint appears in a strong invalid test input. With robustness interaction ($b \geq 1$), each described invalid schema is combined with all valid schemata of degree $d = b$. The coverage criterion is satisfied if all combinations of invalid schemata and b-sized valid schemata are covered by strong invalid test inputs.

Following these brief introductions of CT and CRT, the conceptual difference between the two approaches should become clear. CT and CRT use the same parameters and values. But CT does not distinguish between valid and invalid schemata. Instead, both types of schemata are mixed and the FDE purely relies on the combinatorics, i.e. different testing strengths t. In contrast, CRT distinguishes valid and invalid schemata to avoid the effect of input masking. Here too the FDE relies on combinatorics but the avoidance of input masking has an additional influence.

CRT requires the effort to model error-constraints. Test selection strategies that consider error-constraints also become more complex. This raises the question whether the avoidance of input masking outweighs the additional effort and complexity of CRT. Until now, only artificial test scenarios are used to compare CT with CRT (cf. [7]) and it remains unclear if indicated advantages of CRT can be transferred to real-world scenarios. Therefore, this case study was conducted.

3. Related Work

To the best of our knowledge, Sherwood [6] first mentioned invalid values in the context of CATS which is a test selection strategy and tool for CT. Cohen et al. [14] and Czerwonka [15] also acknowledged the necessity to separate valid and strong invalid test inputs. They also published test selection strategies and tools and the IPMs contain semantic information to distinguish relevant from irrelevant schemata and to distinguish valid from invalid values. However, invalid value combinations are not directly supported. Therefore, we proposed ROBUSTA and the structure of RIPMs with error-constraints [7].

Many studies exist that demonstrate the usefulness and effectiveness of CT (cf. [16, 17, 18]). But most studies do not distinguish between relevance and validness and focus on testing the normal behavior.

One case study by Wojcıak & Tzoref-Brıll [19] reports on applying CT and also considers testing with invalid inputs. They report that single error coverage was not sufficient because EH depended on interactions between invalid and valid values. In particular, "the same [exception] would often be handled differently depending on the firmware in control [...] or depending on the configuration of the system". A further remark is concerned with the ratio of valid versus invalid test inputs: "Since a lot of attention was given to [robustness] testing [...] where full recovery in the presence of [exceptions] was expected, the [test suite] contained a ratio of up to 2:1 [invalid test inputs vs. valid test inputs]."

4. Case Study Design

In this section, the case under analysis and the data collection procedure are introduced.

4.1. Case Under Analysis

The case is a development project conducted by an IT service provider of an insurance company, where a new software was developed to manage the life-cycle of life insurance contracts. One subsystem of the software is concerned with the validation of insurance application data according to a set of validation rules and with forwarding the data when it satisfies the validation rules. It is the same project which we analyzed in a previous case study (cf. [18]).

Altogether, 31 validation rules are defined to check insurance application data. The order of the validation rules is predefined and all validation rules are traversed for each insurance application data. Whenever a validation rule is not satisfied by an insurance application, a corresponding error code is returned and the remaining validation rules are skipped. If all validation rules are satisfied, the subsystem returns SUCCESS and the insurance application data is further processed. Although, the further processing is out of scope for this case study.

Each validation rule is built as an implication consisting of two parts:

$$\text{isApplicable(application)} \Rightarrow \text{isValid(application)}$$

The first part determines whether a given validation rule is applicable to the insurance application data or not. If a rule is applicable, the insurance application must not violate the rule, i.e. isValid(application). Otherwise, the validation rule is ignored.

Because details of the case are confidential, a generic example is given to provide further illustration of validation rules. The example depicts two validation rules to define maximum sums that can be insured depending on the permissions of the insurance agents. The first validation rule is applicable to all applications created by insurance agents with the highest level of permission. The second validation rule is applicable to all applications that are created by insurance agents with lower permission level.

The distinction between the two validation rules is made by the first part of the implication:

Rule 1: $\texttt{isApplicable}(\texttt{application}) :$
$\texttt{application.agent.permission} = \texttt{highest\_level}$

Rule 2: $\texttt{isApplicable}(\texttt{application}) :$
$\texttt{application.agent.permission} \neq \texttt{highest\_level}$

The second part of the implication is used to enforce the maximum insured sum. As an application may consist of several partial contracts, the individual insured sums of all partial contracts are collected first. Afterwards, it is checked whether the total sum exceeds the threshold. While the structure of both rule's isValid() parts is the same, different values for the maximum_insured_sum constant are used:

$$\begin{aligned}&\texttt{isValid}(\texttt{application}) :\\ &\quad \texttt{total\_sum} = \sum_{\texttt{partial} \in \texttt{application}} \texttt{partial.insured\_sum}\\ &\quad \texttt{total\_sum} \leq \texttt{maximum\_insured\_sum}\end{aligned}$$

This example shows that many parameters may be involved in a validation rule, that intermediate calculations may be required, and that intermediate calculations may be reused in different validation rules. Therefore, all validation rules should be tested thoroughly.

For this case study, we consider the current set of validation rules as correct and treat them as our specification. By browsing the source code repository, we have identified 13 changes that have been made to the validation rules in order to correct them. Each change documents a fault that existed previously but is fixed prior to release. Based on these 13 changes, we reconstructed 13 implementation versions of which each contains one fault.

The 13 faults can also be classified according to our robustness fault classification (cf. [7]). Five faults can only be detected by invalid test inputs, while eight faults can be detected by both valid and invalid test inputs. Two of these five faults can be classified as *faults in error-signaling*. To reveal them, invalid test inputs must trigger EH which responds with an incorrect error code. The other three faults can be classified as *faults in error-detection conditions*. The conditions are too weak and do not detect invalid test inputs. Hence, the SUT incorrectly continues with its normal behavior.

The remaining eight faults can be detected by both valid and invalid test inputs. They are *faults in error-detection conditions*. Four of theses faults have conditions that are too strong and therefore incorrectly detect exception occurrences for valid test inputs. The other four faults have characteristics of being too weak and too strict at the same time because wrong parameters with similar characteristics are used in the exception condition. As a consequence, an invalid test input may not violate the condition (too weak) while a valid test input may not satisfy the condition (too strong).

4.2. Data Collection Procedure

Data collection refers to the measurement and calculation of metric values from test execution. Therefore, metrics are defined in this section. Furthermore, the modeling of the IPM and RIPM as well as the selection and execution of test inputs is described.

4.2.1. Metrics

The resources available from the software development project are not directly analyzed and compared. Instead, they are used to reconstruct the implementation versions for test execution and to create a RIPM and an IPM that represent variations of insurance application data.

Based on the RIPM and IPM, test inputs are selected using a CT and a CRT test selection strategy. Then, the test inputs are executed on the 13 reconstructed implementations to assess the effectiveness.

A common metric to assess the effectiveness is **fault detection effectiveness** (FDE) [11, 16]. A test suite T is denoted as *failing* for a test scenario SC if at least one of the test inputs $\tau \in T$ detects the fault in SC.

$$\text{failing}(T, SC) = \begin{cases} 1 \text{ if } \exists \tau \in T \text{ that fails for } SC \\ 0 \text{ otherwise} \end{cases}$$

Using the failing function, FDE is defined as the ratio between the number of test suites T of a test suite family T^* that fail for a test scenario SC and the number of all test suites in the family T^*. In this case study, the family of test suites contains 20 different variants. In other words, the FDE is based on 20 randomized test suites that all satisfy the same coverage criterion for the same IPM or RIPM. They all test the same test scenario.

$$\text{FDE}(T^*, SC) = \frac{\sum_{T \in T^*} \text{failing}(T, SC)}{|T^*|}$$

Further on, the **average fault detection effectiveness** (AFDE) denotes the average FDE over a family of test scenarios SC^*. In our case study, the family of test scenarios SC^* consists of the 13 reconstructed implementations. The AFDE represents the average effectiveness of CRT and CT equally distributed over the 13 faults.

$$\text{AFDE}(T^*, SC^*) = \frac{\sum_{SC \in SC^*} \text{FDE}(T^*, SC)}{|SC^*|}$$

4.2.2. Modeling of IPM and RIPM

Since the FDE and AFDE metrics highly depend on the quality of the RIPM and IPM, a systematic modeling approach is necessary. We model the IPM first and later extend it with error-constraints to get a RIPM.

The IPM is modeled iteratively for one validation rule at a time. In each iteration, parameters and values are added to ensure that test inputs with the following three characteristics can be detected: (1) test inputs that are not applicable; (2) test inputs that are applicable and valid; (3) test inputs that are applicable but not valid. In addition, some exclusion-constraints are introduced to ensure syntactic correctness of selected test inputs. The IPM is considered as complete once the IPM contains all parameters and values necessary to satisfy branch coverage of each validation rule.

For the RIPM, the modeling of additional error-constraints is required. The error-constraints are modeled iteratively and we add new or update existing ones until the separation of valid and strong invalid test inputs conforms to the responses of the SUT, i.e. the SUT returns SUCCESS for each valid test input and the SUT returns an error code for each strong invalid test input.

In total, the IPM and RIPM consist of 32 parameters and 106 values. Most parameters have two, three, or four values each. But two parameters have six values each

and one parameter has even nine values. Three exclusion-constraints of which each restricts combinations of two parameters are required to ensure syntactical correctness of the insurance applications. Furthermore, the RIPM contains 31 error-constraints. 15 error-constraints annotate single values as invalid. The remaining 16 error-constraints annotate schemata with 2, 3, or 5 values.

The complete IPM and RIPM are described below in exponential notation. For parameters and values, x^y refers to y parameters with x values. For exclusion- and error-constraints, x^y refers to y constraints with x parameters.

$$\text{Parameters \& Values: } 9^1 6^2 5^1 4^8 3^8 2^{12}$$

$$\text{Exclusion-Constraints: } 2^3$$

$$\text{Error-Constraints: } 5^2 3^6 2^8 1^{15}$$

4.2.3. Selecting and Executing Test Inputs

After creating the IPM and RIPM, both models are used to select sets of test inputs. Since we compare CRT with CT, two different test selection strategies are used. ROBUSTA is used to select test inputs for the RIPM and IPOG-C is used to select test inputs for the IPM.

To compare the FDE and AFDE of CRT with CT, test suites that satisfy different coverage criteria are used. We apply IPOG-C to select test suites that satisfy t-wise relevant coverage for $t \in \{1, ..., 5\}$. Furthermore, we apply ROBUSTA to select test suites that satisfy t-wise valid coverage with $t \in \{1, ..., 3\}$ and that satisfy b-wise strong invalid coverage with $b \in \{0, 1\}$.

To reduce the effect of accidental fault detection caused by ordering, the order of parameters and values of the input parameter models is randomly reordered and 20 different model variants are used to select test suites for each coverage criteria.

Table 1 depicts the average sizes of test suites that satisfy the different coverage criteria. Since ROBUSTA encompasses two coverage criteria (t-wise valid coverage and b-wise strong invalid coverage), the test suites are considered both, separately and combined.

The largest test suite is selected by IPOG-C which is required to satisfy t-wise relevant coverage with $t = 5$ (15023.70 test inputs). The second-largest test suite is also selected by IPOG-C to satisfy t-wise relevant coverage with $t = 4$ (2813.45 test inputs). The third-largest test suite is selected by ROBUSTA and satisfies t-wise valid coverage with $t = 3$ and b-wise strong invalid coverage with $b = 1$ (2224.30 test inputs).

When comparing the test suite sizes of t-wise relevant coverage of IPOG-C with t-wise valid coverage of ROBUSTA, it can be seen that the error-constraints drastically reduce the number of valid test inputs.

After test input selection, the test suites are used to stimulate the SUT in 13 different versions. Therefore, the 13 reconstructed implementations of which each contains one fault are tested to determine which test suite is able to detect which fault. The results are discussed in the following section.

Table 1
Test suite sizes of test suites for different coverage criteria

Coverage Criteria	t	b	Size
t-wise relevant coverage	1	-	9.00
	2	-	68.10
	3	-	480.10
	4	-	2813.45
	5	-	15023.70
t-wise valid coverage	1	-	7.00
	2	-	48.30
	3	-	267.95
b-wise strong invalid coverage	-	0	301.00
	-	1	1956.35
t-wise valid coverage and b-wise strong invalid coverage	1	0	308.00
	1	1	1963.35
	2	0	349.30
	2	1	2004.65
	3	0	568.95
	3	1	2224.30

5. Results & Discussion

In this section, the case study results regarding the computed FDE and AFDE values are reported and discussed.

5.1. Fault Detection Effectiveness

Table 2 lists the FDE values of all test suites families applied to all 13 implementations. For better readability, + is used to indicate an FDE value of 1.00. The faults nos. 1 to 8 can all be detected by both valid and invalid test inputs, while the faults nos. 9 to 13 can only be detected by invalid test inputs. Again, the shown FDE value is an average value for one test suite family with 20 different test suites that are created by randomizing the order of parameters and values before selecting test inputs. As an example, in the first row for fault no. 3, an FDE value of 0.05 means that one out of 20 test suites detected the fault at least once per test suite.

As can be observed, t-wise relevant coverage is not able to detect all faults reliably. The FDE values increase when testing strength t grows. But even with $t = 5$ (15023.70 test inputs), only 7 faults are detected reliably (FDE value of 1.00). Further on, fault no. 10 remains undetected (FDE value of 0) and faults nos. 9 and 13 are only detected by one out of 20 test suites (FDE value of 0.05).

The CRT coverage criteria are characterized by avoiding the invalid input masking effect. Since all invalid schemata are excluded by t-wise valid coverage, the faults

Table 2
FDE values for different coverage criteria

Coverage Criteria	t	b	FDE values for faults nos. 1 to 13													AFDE values
			1	2	3	4	5	6	7	8	9	10	11	12	13	
t-wise relevant coverage	1	-	0	0	0.05	0.05	0	0	0	0	0	0	0.25	0.05	0	0.03
	2	-	0.10	0.10	0.45	0.20	0.10	0	0	0	0	0	0.65	0.20	0	0.14
	3	-	0.75	0.75	+	+	0.65	0.05	0.10	0.05	0.05	0	+	0.65	0	0.47
	4	-	+	+	+	+	+	0.15	0.10	0.05	0	0	+	+	0	0.56
	5	-	+	+	+	+	+	0.50	0.35	0.15	0.05	0	+	+	0.05	0.62
t-wise valid coverage	1	-	0.75	0.75	+	+	0.50	0.50	+	0.80	0	0	0	0	0	0.48
	2	-	+	+	+	+	+	+	+	+	0	0	0	0	0	0.62
	3	-	+	+	+	+	+	+	+	+	0	0	0	0	0	0.62
b-wise strong invalid	-	0	+	+	+	+	+	+	0.90	0.80	+	+	+	+	+	0.98
	-	1	+	+	+	+	+	+	+	+	+	+	+	+	+	+
t-wise valid coverage and b-wise strong invalid coverage	1	0	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	1	1	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	2	0	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	2	1	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	3	0	+	+	+	+	+	+	+	+	+	+	+	+	+	+
	3	1	+	+	+	+	+	+	+	+	+	+	+	+	+	+

nos. 9 to 13 cannot be detected. But for all other faults, *t*-wise valid coverage has higher FDE values for the same testing strength t when compared to *t*-wise relevant coverage. Because invalid input masking is avoided, a testing strength of $t = 2$ is sufficient to detect faults nos. 1 to 8 reliably (FDE values of 1.00).

Using *b*-wise strong invalid coverage with $b = 0$, 11 out of 13 faults can already be detected reliably and the two remaining faults have high FDE values of 0.90 and 0.80. The effectiveness of robustness interactions is even higher and all faults can be detected reliably with $b = 1$.

Four faults that have too strong error detection conditions and that actually require valid test inputs to be detected are also reliably detected by *b*-wise strong invalid coverage. We could observe that a strong invalid test input that is expected to violate the error detection condition of the l-th validation rule is also expected to satisfy all prior validation rules from 1 to $l-1$. Therefore, strong invalid test inputs can be considered as "partially-valid" test inputs that are able to accidentally detect faults that require valid test inputs. This effect is strengthened by robustness interactions because more test inputs are selected and more interactions are covered by them.

ROBUSTA combines *t*-wise valid coverage and *b*-wise strong invalid coverage and the FDE values show that test suites for both coverage criteria complement each other. Since valid and strong invalid test inputs are able to detect faults nos. 1 to 8, the FDE values are complemented by the combination of both test suites. For faults nos. 9 to 13, the FDE values are not complemented by the combination of both test suites. This is because test suites that only satisfy *t*-wise valid coverage cannot detect these faults. Therefore, the FDE values of the combined test suites are the same as the FDE values of the test suites that satisfy *b*-wise strong invalid coverage.

In order to detect all faults reliably, the *b*-wise strong invalid coverage must be selected because faults nos. 9 to 13 remain undetected otherwise. Either robustness interaction ($b > 0$) or the combination of *b*-wise strong invalid coverage with *t*-wise valid coverage is required to reliably detect faults nos. 1 to 8. Even though $t = 1$ is only sufficient to detect three of the first eight faults reliably, the combination with *b*-wise strong invalid coverage improves the FDE and all faults can be detected reliably.

The discussion of the FDE shows which coverage criteria are appropriate to reliably detect different types of faults. Next, we discuss the AFDE over all 13 faults.

5.2. Average Fault Detection Effectiveness

Because AFDE values are average values over a set of faults, AFDE allows making general statements about both the effectiveness and the efficiency of coverage criteria. First, we discuss the effectiveness in terms of AFDE values of different coverage criteria. Therefore, Table 2 lists the AFDE values for test suites that satisfy different coverage criteria. Afterwards, we discuss the efficiency in terms of AFDE values in relation to test suite sizes (listed in Table 1).

The AFDE values reflect what we discussed before since they aggregate FDE values. Because of the invalid input masking effect, test suites that satisfy *t*-wise relevant coverage only reach an AFDE value of 0.62.

In direct comparison, test suites that satisfy *t*-wise valid coverage reach a maximum AFDE value of 0.62 as well. The same AFDE value can be reached because they prevent invalid input masking. However, the AFDE value cannot be further improved by increasing the testing

strength because faults nos. 1 to 8 are already detected reliably and faults nos. 9 to 13 cannot be detected by valid test inputs. Comparing the two coverage criteria for each testing strength individually shows that the AFDE value of t-wise valid coverage is always higher than the AFDE value of t-wise relevant coverage.

For b-wise strong invalid coverage, the lowest AFDE value is 0.98 (no robustness interactions) which is always higher than the AFDE values of t-wise relevant and valid coverage. Furthermore, b-wise strong invalid coverage with robustness interactions has an AFDE value of 1 and therefore detects all faults reliably.

Overall, the combination of t-wise valid coverage and b-wise strong invalid coverage performs the best and always detects all faults reliably.

When putting the AFDE values in relation to test suite sizes, it can be noted that t-wise relevant coverage has the worst efficiency as it requires 15023.70 test inputs for an AFDE value of 0.62. In contrast, t-wise valid coverage only requires 48.30 test inputs for an AFDE value of 0.62.

The best efficiency is offered by the combination of t-wise valid coverage with $t = 1$ and b-wise strong invalid coverage with $b = 0$ which requires 308.00 test inputs for an AFDE value of 1.00. When using an AFDE value of 0.92 as a lower boundary (12 out of 13 faults), b-wise strong invalid coverage with $b = 0$ is sufficient and only requires 301.00 test inputs for an AFDE value of 0.98.

This discussion about efficiency is, of course, influenced by the characteristics of the 13 faults and cannot be generalized. But as more general statements, it can be observed that t-wise relevant coverage requires more test inputs to reach a similar AFDE value than t-wise valid coverage, b-wise strong invalid coverage, or the combination of both. At the same time, the combination of t-wise valid coverage and b-wise strong invalid coverage always has an AFDE value of 1.00 while at most 2224.30 test inputs are used. This finding is also consistent with our prior experimental evaluation (cf. [7]).

Therefore, we draw the conclusion that t-wise valid coverage, b-wise strong invalid coverage, and the combination of both perform as well as or better than t-wise relevant coverage in terms of effectiveness and efficiency. Although, the findings are only derived from one particular case. Therefore, we do not consider this to be true for all SUTs but for SUTs with many validation rules.

6. Threats to Validity

We compare the effectiveness of CRT using an implementation of the ROBUSTA test selection strategy with CT using an implementation of the IPOG-C test selection strategy. To ensure an unbiased implementation, both implementations follow the guidelines of Kleine & Simos [20]. Further on, the source code of the test selection strategies is published as part of the `coffee4j` open-source test automation framework[1].

The effectiveness of CRT and CT highly depend on the IPM and RIPM. Furthermore, the effectiveness depends on the faults that are considered in this case study.

Unfortunately, details of the case, i.e. source code of the validation rules and detailed descriptions of the faults, are confidential. To improve transparency and reproducibility, we describe the faults and make the characteristics of the IPM and RIPM explicit.

To avoid any bias, both the IPM and RIPM are modeled systematically and share the same set of parameters and values. To prevent falsified results due to accidental fault triggering, the orders of parameters and values are randomized and 20 different variants are used in test input selection. All presented FDE values are average values.

Since this is a case study with only one case, it is difficult to generalize the findings [10]. Further on, it has to be noted that the archival data of this case study is only a snapshot and the ground truth, i.e. the existing and previously existing faults, is unknown. Hence, the data can be biased towards simpler faults that are easier to detect. To prevent too far-reaching conclusions, we describe the characteristics of the SUT and also limit our conclusions to similar systems with many validation rules.

7. Conclusion

CRT extends CT to generate separate test suites with valid and strong invalid test inputs in order to avoid input masking that is caused by EH. Therefore, CRT requires additional effort to model error-constraints and introduces additional complexity to test selection strategies because error-constraints must be considered. This raises the question about the usefulness of CRT and whether the avoidance of input masking outweighs the additional effort and complexity. Until now, only artificial test scenarios are used to compare CT with CRT and it remains unclear if indicated advantages of CRT can be transferred to real-world scenarios.

In this paper, we therefore present the results of a case study based on a real-world system with 31 validation rules and 13 previously existing faults. To compare CT with CRT, we construct a IPM and a RIPM, select test inputs, and stimulate 13 implementations of the real-world system of which each implementation contains one of the 13 previously existing faults. For the subsequent discussion, we introduce the FDE and AFDE metrics.

To summarize the findings of this case study, we discuss both research questions individually.

Research Question 1: Our results indicate that the CRT test method is applicable in real-world test scenarios. This case study demonstrated that RIPMs with 32

[1] See https://coffee4j.github.io for more information.

parameters and 31 error-constraints can be constructed. Further on, the ROBUSTA test selection strategy is capable of selecting test suites for RIPMs with 32 parameters and 31 error-constraints.

Research Question 2: The comparison of CRT with CT is consistent with the findings of our previously conducted controlled experiment with artificial test scenarios (cf. [7]). Since the case under analysis has much EH, CRT performs better than CT in terms of FDE. Further on, it requires fewer test inputs to achieve better AFDE values than CT.

Therefore, we draw the conclusion that *t*-wise valid coverage, *b*-wise strong invalid coverage, and the combination of both perform as well as or better than *t*-wise relevant coverage in terms of effectiveness and efficiency.

Although, the FDE and AFDE values are influenced by the characteristics of the 13 faults and cannot be generalized. Therefore, we do not consider this to be true for all SUTs but for SUTs with much EH.

In future work, we plan to conduct further case studies to learn more about the FDE of CRT and CT.

References

[1] IEEE, IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990 (1990).

[2] A. Avižienis, J. Laprie, B. Randell, C. E. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Trans. Dependable Sec. Comput. 1 (2004) 11–33.

[3] C. Marinescu, Are the classes that use exceptions defect prone?, in: Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution, EVOL/IWPSE 2011, Szeged, Hungary, September 5-6, 2011., 2011, pp. 56–60.

[4] P. Sawadpong, E. B. Allen, B. J. Williams, Exception handling defects: An empirical study, in: 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering, 2012, pp. 90–97.

[5] C. Nie, H. Leung, A survey of combinatorial testing, ACM Comput. Surv. 43 (2011) 11:1–11:29.

[6] G. B. Sherwood, Effective testing of factor combinations, in: Proceedings of the Third International Conference on Software Testing, Analysis and Review, Washington, DC, 1994, pp. 151–166.

[7] K. Fögen, H. Lichter, Combinatorial robustness testing with negative test cases, in: Proceedings of the 19th IEEE International Conference on Software Quality, Reliability and Security, QRS 2019, Sofia, Bulgaria, July 22-26, 2019, 2019, pp. 34–45.

[8] K. Fögen, H. Lichter, An experiment to compare combinatorial testing in the presence of invalid values, in: Proceedings of the 7th International Workshop on Quantitative Approaches to Software Quality co-located with 26th Asia-Pacific Software Engineering Conference (APSEC 2019), Putrajaya, Malaysia, December 2, 2019., 2019, pp. 27–36.

[9] B. A. Kitchenham, L. Pickard, S. L. Pfleeger, Case studies for method and tool evaluation, IEEE Softw. 12 (1995) 52–62.

[10] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering 14 (2009) 131–164.

[11] M. Grindal, J. Offutt, S. F. Andler, Combination testing strategies: a survey, Softw. Test., Verif. Reliab. 15 (2005) 167–199.

[12] C. Nie, H. Leung, The minimal failure-causing schema of combinatorial testing, ACM Trans. Softw. Eng. Methodol. 20 (2011) 15:1–15:38.

[13] L. Yu, Y. Lei, M. N. Borazjany, R. Kacker, D. R. Kuhn, An efficient algorithm for constraint handling in combinatorial test generation, in: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013, 2013, pp. 242–251.

[14] D. M. Cohen, S. R. Dalal, M. L. Fredman, G. C. Patton, The AETG system: An approach to testing based on combinatiorial design, IEEE Trans. Software Eng. 23 (1997) 437–444.

[15] J. Czerwonka, Pairwise testing in real world, in: 24th Pacific Northwest Software Quality Conference, volume 200, Citeseer, 2006.

[16] J. Petke, M. B. Cohen, M. Harman, S. Yoo, Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection, IEEE Trans. Software Eng. 41 (2015) 901–924.

[17] H. Wu, n. changhai, J. Petke, Y. Jia, M. Harman, An empirical comparison of combinatorial testing, random testing and adaptive random testing, IEEE Transactions on Software Engineering (2018) 1–1.

[18] K. Fögen, H. Lichter, A case study on robustness fault characteristics for combinatorial testing - results and challenges, in: Proceedings of the 6th International Workshop on Quantitative Approaches to Software Quality co-located with 25th Asia-Pacific Software Engineering Conference (APSEC 2018), Nara, Japan, December 4, 2018., 2018, pp. 22–29.

[19] P. Wojciak, R. Tzoref-Brill, System level combinatorial testing in practice - the concurrent maintenance case study, in: Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA, 2014, pp. 103–112.

[20] K. Kleine, D. E. Simos, An efficient design and implementation of the in-parameter-order algorithm, Mathematics in Computer Science 12 (2018) 51–67.

An Evaluation of Machine Learning Methods for Predicting Flaky Tests

Azeem Ahmad[a], Ola Leifler[a] and Kristian Sandahl[a]

[a]Linköping University, 581 83 Linköping, Sweden

Abstract

The quality of the product is uncertain if the test cases change their outcome(i.e., from pass to fail or vice versa) without modifications in the codebase. Tests that change their outcome without any modification in the code base are called flaky tests. The common method to detect test flakiness is to re-run the test cases to ensure if test cases outcomes are deterministic. The cost of re-running tests is often high. In addition to re-running tests, developers put effort and time to investigate the root causes of test flakiness. The need for prevention of flaky tests is evident before adding it to a test suite. In this paper we have investigated as a means of prevention the feasibility of using machine learning (ML) classifiers for flaky test prediction in project written with `Python`. This study compares the predictive accuracy of the three machine learning classifiers (Naive Bayes, Support Vector Machines, and Random Forests) with each other. We compared our findings with the earlier investigation of similar ML classifiers for projects written in `Java`. Authors in this study investigated if test smells are good predictors of test flakiness. As developers need to trust the predictions of ML classifiers, they wish to know which types of input data or test smells cause more false negatives and false positives. We concluded that RF performed better when it comes to precision (> 90%) but provided very low recall (< 10%) as compared to NB (i.e., precision < 70% and recall >30%) and SVM (i.e., precision < 70% and recall >60%).

Keywords

Improve Software Quality, Flaky Test Detection, Machine Learning Classifiers, Experimentation, Test Smells

1. Introduction

Developers need to ensure that their changes to the code base do not break existing functionality. If test cases fail, developers expect test failures to be connected to the changes. Unfortunately, some test failures have nothing to do with the code changes. Developers spend time analyzing changes trying to identify the source of the test failure, only to find out that the cause of the failure is test flakiness (TF). Many studies [1, 2, 3, 4] have been conducted to determine the root causes of test flakiness. These studies concluded that the main root cause of TF is the test smells. Test smells are poorly written test cases and their presence negatively affect the test suites and production code or even the software functionality [5]. Another definition is *"poor design or implementation choices applied by programmers or testers during the development of test cases"* [2] . Asynchronous wait, input/output calls, and test order dependency are some of the test smells that have been found to be the most common causes of TF [1]. The results presented by Luo et al. [1] were partially replicated by Palomba and Zaidman [2], leading to the conclusion that the most prominent causes of TF are test smells such as asynchronous wait, concurrency, and input output issues. There is strong evidence that the main reasons for test flakiness are specific test smells. Luo et al. suggested that *"developers should avoid specific test smells that lead to test flakiness"*. Authors in [2] investigated the question: *"To what extent can flaky tests be explained by the presence of test smells?"* They concluded that the *"cause of 54% of the flaky tests can be attributed to the characteristics of the co-occurring test smell"*.

Mapping test smells to flaky test resemble the problem of mapping words to spam/ham email. Certain words (i.e., sale, discount etc.) are more frequent in spam emails. Many studies [6, 7, 8, 9, 10, 11, 12, 13, 14, 15] have been conducted to predict email class (i.e., spam or ham) based on email contents. We adopted a similar approach in this study to determine the flakiness of test cases based on the test case code. Machine Learning approaches have been widely studied and there are lots of algorithms that can be used in e-mail classification including Naive Bayes [16][17], Support Vector Machines [18][19][15, 14], Neural Networks [20][21], K-nearest neighbor [22].

Recently, Pinto et al. evaluated five machine learning classifiers (Random Forest, Decision Tree, Naive Bayes, Support Vector Machine, and Nearest Neighbour) to generate flaky test vocabulary [23]. They con-

8th International Workshop on Quantitative Approaches to Software Quality in conjunction with the 27th Asia-Pacific Software Engineering Conference (APSEC 2020) Singapore, 1st December 2020

azeem.ahmad@liu.se (A. Ahmad); ola.leifler@liu.se (O. Leifler); kristian.sandahl@liu.se (K. Sandahl)

0000-0003-3049-1261 (A. Ahmad)

cluded that Random Forest and Support Vector Machine provided best prediction of flaky tests. The investigated test cases were written in `Java` and the authors concluded that: *"future work will have to investigate to what extent their findings generalize to software written in other programming languages* [23].

In this study, we implemented supervised ML classifiers to detect if the test case is flaky or not based on the contents of a test cases written in `Python`. We compared our findings with what was presented by Pinto et al. [23]. We looked for evidence if machine learning classifiers are applicable in predicting flaky tests and the results can be generalized to test cases written in other languages. In addition to this, our unique contribution is to investigate if test smells are good predictors of test flakiness. Through manual investigation of false positives and false negatives, we concluded a list of test smells that are strong and weak predictors of test flakiness. We investigated the following research questions in this study.

RQ1: What are the predictive accuracy of Naive Bayes, Support Vector Machine and Random Forest concerning flaky test detection and prediction?

RQ2: To what extent the predicting power of machine learning classifiers vary when applied on software written in other programming language?

RQ3: What can we learn about the predictive power of test smells using machine learning classifiers mentioned in RQ1?

2. Data Set Description and Prepossessing

We wrote a script to extract the contents of all test cases from open-source projects, mentioned in Table 1. After the test case content's extraction, we checked which of the test cases, in our database, has been mentioned in [24] as flaky. After this mapping, we finalized a database with the project name, test case name, test case content and a label. There are many keywords in the test case code that are irrelevant for the identification of test flakiness. We performed extensive data cleaning such as removing punctuation marks, digits and specific keywords (i.e., int, string, array, assert*) as well as converting text to lower case.

2.1. Classifiers:

An NBC, first proposed in 1998, is a probabilistic model which can determine the outcome (i.e., flaky or not flaky) of an instance (i.e., test case) based on the contents of its features (i.e., test case code). In our case, the outcome of NBC is binary. NBC is widely applied in classification and known to obtain excellent results. [25].

The attractive feature of SVM is that it eliminates the need for feature selections, which makes spam classification easy and faster [14]. SVM deals with the dual categories of classification and can find the best hyperplane to partition a sample space [15].

RF is an ensemble classification method (a technique that combines several base models to produce an optimal predictive model) suitable for handling problems that involve grouping data into different classes. RF predicts by using decision trees. Trees are constructed during training which can later be used for class prediction. There is a vote associated with each tree and once the class vote has been produced for all individual trees, the class with the highest vote is considered to be the output.

2.2. Performance Metrics and Parameters Tuning

To evaluate the predictive accuracy of classifiers, accuracy as the only performance indices is not sufficient [16]. We must consider precision, recall, F1-score, ROC curve, false positives and false negatives [16]. There is always some cost associated with false positives and false negatives. When a non flaky test wrongly classified as flaky, it gives rise to a some what insignificant problem, because an experienced user can bypass the warning by looking at test case code. In contrast, when a flaky test is wrongly classified as non flaky test, this is obnoxious, because it indicates the test suite still have test cases whose outcome cannot be trusted.

The experiment started with the implementation of simple NB without Laplace smoothing. The results did not provide good accuracy or precision, because without Laplace smoothing, the probability of appearing a rare test smell (i.e., test smell that was not in the training set) in the test set is set to 0, given the formula

$$\theta_j = n_{jc}/n_c$$

where the θ is the probability that an individual test smell is present in a flaky test, n_{jc} represents the number of times that particular test smell appeared in a test case and n_c represents the number of times that test smell appeared in any test case. Laplace smoothing refers to the modification in the equation:

$$\theta_j = (n_{jc} + \alpha)/(n_c + \alpha)$$

Table 1
Open-source project names provided by [24] with number of total test cases and flaky tests

Project Name	Total Number of TCs	Flaky Tests
apache-qpid-0.18	2357	284
hibernate 4	3231	273
apache-wicket-1.4.20	1250	216
apache-karaf-2.3	163	102
apache-struts 2.5	2346	60
apache-derby-10.9	3832	40
apache-lucene-solr-3.6	764	7
apache-cassandra-1.1	523	4
apache-nutch-1.4	7	4
apache-hbase-0.94	29	2
apache-hive-10.9	23	2
jfreechart-1.0.18	2292	0

where we set the $\alpha = 1$ so that classifier adds 1 to the probability of rare test smells that were not present in the training set. Another step is to identify the threshold (i.e., 0.0 - 1.0) which will increase the predictive accuracy of the outcome. As far as SVM was concerned, although the feature data set space was linear, we decided to use both kernels (i.e., linear and poly) for the sake of experiment. For random forest, we used `ntree` between 300 - 700 as well as restricting number of variables available for splitting at each tree node known as `mtry` between 25 and 100.

3. Results

This section discusses the performance of NBL, SVM and RF with different parameters. We compared our results with the findings of Pinto et al. to discuss how results vary between Java and Python projects. We also discussed why some classifiers do not perform as expected and what can we learn about the predictive power of test smells for test flakiness detection and prediction.

3.1. RQ1: Performance of Naive Bayes Classifier, Support Vector Machine and Random Forest

Table 2 shows the 20 features with the highest information gain together with their frequency with respect to flaky and non-flaky tests. We assigned the features to the categories presented by Luo et al. in [1]. We manually traversed the code of flaky and non-flaky tests to understand the context and how features were used in the tests to assign categories. The top feature "conn" appeared in 1361 flaky tests and only 15 non-flaky tests. This feature is associated with external connection to input/output devices and lies under the category of "IO", presented by Luo et. al in [1]. The second top feature is "double" which appeared in 1190 flaky tests and 12 non-flaky tests assigned to the category of "IO" followed by "floating points operations". The top 3rd feature "tabl" was related to table creation during runtime for databases queries and appeared 1150 times in flaky tests and 52 times in non-flaky tests.

Figure 1 (A) represents the ROC curve [26] concerning NBC with Laplace smoothing denoted as NBL with different threshold (i.e., from 0.0 to 1.0). We conducted different experiments with different training and test data sets such as 50/50, 60/40, 70/30, 80/20 and 90/10. We found similar values for *k-fold* cross validation. ROC curve provides a comparison between sensitivity and specificity helping in organizing classifiers and visualizing their performance [26]. Sensitivity also known as the true positive rate represents a benefit of predicting flaky tests correctly and specificity also known as false positive rate represents the cost of predicting non flaky tests as flaky tests. In the case of false positive, developers need to spend effort and time, just to find out that this is a classifier mistake and the test case is not flaky. The optimal target, in the ROC curve, is to rise vertically from origin to the top left corner (higher true positive rate) as soon as possible because then the classifier can achieve all true positives with the cost of committing a few false positive. The diagonal line, in Figure 1 (A), represents the strategy of randomly guessing the outcome. Any classifier that appears in the lower right triangle performs worse than a random guessing and we can see that NBL lies in the upper left triangle. Looking at 1 (A), NBL with 70/30 data partition is suitable to proceed further with 0.4 probability score. NBL, as shown in 1 (A), has stopped issuing positive classification (i.e., flaky test prediction) around 0.76 - 0.87 threshold. After 0.87, it commits more false positive rate.

We tuned different parameters in NBL, SVM and RF before conducting further experiments. We do not intend to provide the results of all experiments because those experiments were only conducted to find the optimal parameters. The rest (i.e., simple NB, SVM with radial and sigmiod kernels) were not included in further experiments and discarded. Figure 1 (A-E) provides comparisons of NBL, SVM-Linear and SVM-Poly (i.e., different kernels) for accuracy, precision, recall and F1-score. All classifiers have achieved good accuracies ranging from 93% - 96%. NBL outperformed SVM although the difference between them is not dramatic. Looking only at the accuracy results of classi-

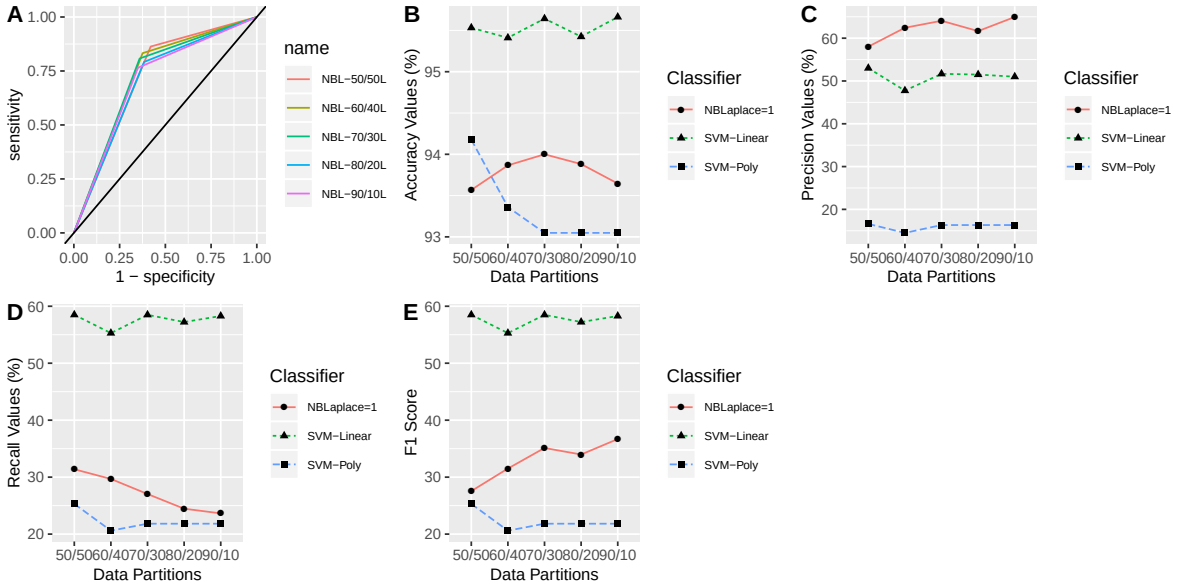


Figure 1: Performance comparison among classifiers. (A) represents the ROC curve of NBL classifier with different data partition and probability score. (B-E) represents the accuracy, precision, recall and F1-score of different classifiers with a different data partition, respectively.

fiers can be deceiving. The important factor for classifier selection is to ask the right question and motivate the choice of using specific classifier such as **are we interested in detecting flaky tests correctly (i.e., precision) or marking a non flaky test as flaky is not cost effective (i.e, recall)**. It is important to look at precision, recall and accuracy all together for classifier selection. We can assume that practitioners are more interested in precision than recall because the test suite size, in many organizations, is very large and they cannot inspect all test cases. In this particular case, any classifier that correctly flag flaky tests will be encouraged. Precision can answer the question; ***"If the filter says this test case is flaky, what's the probability that it's flaky?"***. Figure 1 (C,D) provides precision and recall values for NBL and SVM. It can be noticed that NBL precision is increasing (in C) with the gradual decrease in recall (in D). NBL precision of 65% dictates that 35% of what was marked as flaky was not flaky. Recall is also lower in NBL as compared to SVM-Linear. SVM-Poly performs worst in terms of precision and recall as expected due the fact that the input data set is not polynomial and is well suited for image processing whereas linear kernel performs better for text classification.

F1-score, as presented in Figure 1 (E), is the harmonic mean of precision and recall. F1-score is useful and informative because of prevalent phenomenon of class imbalance in text classification [27]. NBL is a suitable candidate although it has a lower F1-score as compared to SVM-Linear because NBL performs better with short documents as in our case, the training test case consists of 6-15 lines of code [28]. NBL provides higher precision and lower recall as compared to SVM-linear. Another disadvantage of SVM is that it requires high computation and are very sensitive to noisy data [29].

RF provides lesser classification error and better F1-scores as compared to decision trees, NBL and SVM. The precision, in which we are most interested, is usually better than that of SVM and NBL. Authors in [16] also concluded that RF performs better than NBL and SVM. The class outcomes are based on *"votes"* which are calculated by each tree in the forest. The outcome (i.e., flaky or not flaky) is selected based on the higher votes. Figure 2 presents the performance of RF with respect to selected metrics. *mtry* represents the number of variables randomly sampled as candidates at each split while *ntree* is the number of trees to grow. There is no way to find an optimal *mtry* and *ntree*, so we experimented with different settings, as shown in Figure 2. The *mtry* has a direct effect on precision and recall as shown in Figure 2. With an increase in *mtry*, the precision is decreasing and recall in increasing; an un-

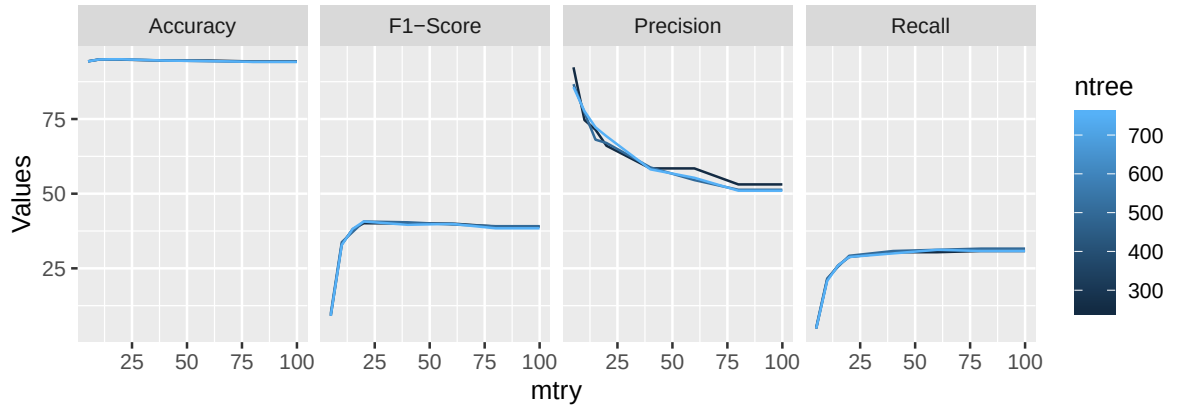


Figure 2: Performance of RF with different parameters (i.e., number of trees and mtry).

Table 2
Top 20 features and assigned category

Features	#FT	#Non-FT	Assigned category from Luo et. al [1]
conn	1361	15	IO
double	1190	12	floating point
tabl*	1150	54	-
rsnext	500	22	Unordered collections
for	241	15	-
jdbcassertfullresultsetr	900	0	IO
messag	1101	87	concurrency
null	334	88	-
sclose	360	0	IO
select	1080	15	IO
sgettransactioncommit	700	15	IO
expr	162	2	-
tcommit	134	5	-
true	700	11	-
epsilon	383	0	floating point
fail	269	13	-
jdbcassertcolumnnamesr	366	0	IO
throw*	592	49	-
rsclose	300	0	IO
row	161	3	-

wanted situation. The optimal value of *mtry* is 5 where precision is higher and recall is lower regardless of the number of trees. The change in *mtry* did not affect the accuracy but as we discussed earlier, we are not only interested in accuracy but precision too.

We performed several experiments to find optimal parameters within a classifier before comparing it to other classifiers. After these experiments, we identified three unique classifiers with unique and optimal parameters. Since, we are most interested in higher precision, we can see that RF with *mtry = 5* and *ntree=250* outperforms all other classifiers only for precision. RF has achieved more than 90% precision with less than 10% recall. We did not achieve high precision (i.e., >90%) in all classifiers. NBL provides unexpected results although it holds a good reputation in terms of detecting spam emails [29]. As compared to NBL and SVM, RF have distinct qualities such as 1) it can work with thousands of different input features without any feature deletion 2) it calculates approximation of important features for classification and 3) it is very robust to noise and outliers [30]. Caruana in [17] compared 10 different ML classifiers and concluded that decision trees and random forest outperform all other classifiers for spam classification.

3.2. RQ2: Predicting Power of ML Classifiers with Respect to Other Languages

In comparison of our findings with what was presented by Pinto et al. [23], we observed two differences. First, the top 20 features are very different in both studies. Only two features such as "tabl" and "throw" marked as star (*) in Table 2 were similar in both the findings. However, we noticed that most of the our features were related to "IO" output category, as presented in Table 2, which complemented the findings of Pinto et al. stating "that all projects manifesting flakiness are IO-intensive" [23]. Second, we have a very lower precision, recall and f1-score as compared to Pinto et al. except at a instance where random forest provided 0.92 precision. Table 3 provides detail statistics of precision, recall, and f1-score of three algorithms for com-

Table 3
Comparison of Precision, Recall and F1-Score between our findings (A) and Pinto et al. (B)

	Precision		Recall		F1-Score		Diff
Algo.	A	B	A	B	A	B	
Random Forest	0.92	0.99	0.4	0.91	0.09	0.95	↘
Naive Bayes	0.62	0.93	0.15	0.8	0.24	0.86	↘
Support Vector	0.51	0.93	0.61	0.92	0.57	0.93	↘

parison. The algorithms on `Python` language continuously performed worst contrary to what pinto et al. claimed: *"Although the studied projects are mostly written in Java, we do not expect major differences in the results if another object-oriented programming language is used instead, since some keywords maybe shared among them" [23].*

We speculate that there could be several reasons associated with these performance reduction such as (1) We implemented the code ourselves using R libraries for aforementioned classifiers whereas pinto et al. used Weka [31] which is an open source machine learning software that can be accessed through a graphical user interface, standard terminal applications [32], (2) Number of features were very high in the training samples and in these cases other models should be considered (i.e., regularized linear regression) that might performed better, (3) the versatility offered by parameter tunning can become problematic and require special considerations that can impact the classifiers, etc.

3.3. RQ3: Test Smells Analysis and their Predictive Power for Test Flakiness Detection and Prediction

We investigated manually different cases of true positives (i.e., correct flaky test prediction), false positive (i.e., flaky test cases marked as non flaky) and false negative (i.e., non flaky test cases marked as flaky) and true negatives (correct non flaky test prediction) to answer RQ3. We observed that it is not only the frequency of test smell that makes a test case flaky but its co-existence with the class code or external factors such as operating systems or specific product. For example, The test smell *'Conditional Test Logic'* as mentioned in [3] refers to nested and complex *'if-else'* structure in the test case. Depending on which branch of *'if-else'* is executed, the system under test may require specific environment settings. Failing to set the environment, during different executions, will flip the test case outcome, thus making it flaky.

After manual investigation of all true/false positives and true/false negatives, we come up with a list of test smells that are strong or weak predictors of test flakiness, as shown in Table 4. Strong predictors refer to those test smells that existed in true positives and true negatives cases whereas weak predictors only existed in false negatives and false positives. Test smells that are classified as weak predictors in this study are still useful and can help in identification of test flakiness, but they are not useful with machine learning classifiers because they require additional information such as what operating system they are running on and whether or not specific configurations should be deployed. Test smells that are classified as strong predictors are very useful with machine learning classifiers because they only exist in test case function as one unit and do not require additional information.

4. Lesson Learned

ML and AI algorithms in recent years have established a good reputation for predicting diseases based on symptoms, spam emails based on email contents and many more. We believe that given a proper input data set which clearly distinguishes between flaky and non flaky tests, ML and AI can provide high prediction capabilities saving effort, time and resources. We strongly believe that practitioners, during training of data set, should not consider complete test cases as an input but only the test codes (i.e., only few lines) that reveal test flakiness.

It is inconclusive that predicting power of machine learning vary with respect to software written in another languages. Investigation on `Java` test cases [23] revealed good results while findings for `Python` test cases performed unexpected, thus requiring more investigations whether lexical information can be traced to flakiness.

Async wait, precision, randomness and IO test smells are string predictors can be predicted by machine learning classifiers with 100% precision because they only exist in test case code and do not require additional information from test class or operating system. Whereas all other test smells mentioned in Table 4 are weak predictors of test flakiness and require additional sources of information. We are only aware of test smells that are investigated in open-source repositories and literature on test smells in closed-source software is scarce.

5. Discussion and Implication

Valuable Indicators for Testers These classifiers can increase the awareness about flaky test vocabulary among

Table 4
Test Smells as Strong and Weak Predictors Together with Source of their Existence

Test Smell Category	Prediction Category	Test Case	Test Class	Operating System	External Libraries	Hardware/Product
Async wait	Strong	[✓]	-	-	-	-
Precision (float operations)	Strong	[✓]	-	-	-	-
Randomness	Strong	[✓]	-	-	-	-
IO	Strong	[✓]	-	-	-	-
Unordered Collection	Weak	[✓]	[✓]	[✓]	-	-
Time	Weak	[✓]	[✓]	[✓]	-	[✓]
Platform	Weak	[✓]	[✓]	[✓]	[✓]	-
Concurrency	Weak	[✓]	[✓]	[✓]	-	-
Test order dependency	Weak	[✓]	[✓]	[✓]	[✓]	[✓]
Resource Leak	Weak	[✓]	[✓]	[✓]	-	-

testers. When a new test is added to a test suite, it will be easy to identify whether this test case contains specific test smells that were known to increase test flakiness during previous executions. Testers can take advantage of these types of information to reduce test flakiness. Testers can easily identify test smells that are independent of their environment with the help of Table 4.

Precision Depends on Data Set: In the literature of ML, particularly with spam detection, it is acknowledged that precision is a function of the combination of the classifier and the data set under investigation. Classifier's precision, in isolation of data set, does not make sense. The right question is ***"how precise a classifier is for a given data set"***. Unfortunately, there is no data available that provides test case contents and an associated label thus, limiting the use of advanced ML and AI algorithms. In addition to lack of flaky test data, all research has been conducted with open-source software and we know a little about what test smells are present in closed-source software. Ahmad et. al. concluded that there are specific test smells that are associated with the nature of the product [33] known as *'company-specific'* test smells. The classifier which are trained on a specific data set or a domain cannot be generalized to be used with another data set or domain. There is a long road ahead to explore the best classifier given different data sets.

Beyond Static Analysis of Test Smells and their Frequency: ML is capable of incorporating different sources of information to increase predictive accuracy as compared to the limited experiment in this study where we only utilized the frequency of test smells in the test case. During the investigation of the cases of *'false negative'* and *'false positive'*, it has been observed that the frequency of test smells in the test case will not be sufficient for prediction. Some test case code (i.e., seeds()) will cancel the effect of test smell (i.e., random()), no matter how frequent the random() function appears in the test case. Some test smell, even with single appearance, will weight more than a test smell for higher frequency.

Precision Vs Recall: When a test suite grows in size, developers would like any indications of tests that are more likely to be flaky rather than adopting an approach of re-run which of-course is not cost effective in terms of time and resources. Developers like to increase precision at the expense of recall. When encountering *'false negative'*, an experienced developer, having sufficient knowledge of the test smells, will bypass the outcome, however, with *'false positive'*, developers are unaware of the fact that test suite still contains flaky tests. The motivation of employing ML classifiers (i.e., higher precision - low recall vs balances precision and recall) should be made clear before proceeding with implementation.

Multi-Factor Input Criteria for Flaky Test Detection: We observed that the ML algorithm should include different sources of information to increase predictive accuracy. These sources may include 1) assigning specific weight (i.e., in numbers) to specific test smells or test code, 2) developer's experience (i.e., new developer, unaware of the test design guidelines are more likely to write flaky tests), 3) company-specific test smells.

6. Related Work

Luo et al., in [1], investigated 52 open-source projects and 201 commits and categorized the causes of test case. Asynchronous wait (45%), concurrency (20%), and test order dependency (12%) were found to be the most common causes of TF. Palomba and Zaidman in [2] partially replicated the results presented by Luo et al. concluding that the most prominent causes of TF

are asynchronous wait, concurrency, and input output and network issues. Authors investigated, in [3], the relationship between smells and TF. Another empirical study of the root causes of TF in Android Apps was conducted by Thorve et al. [4] by analyzing the commits of 51 Apache open-source projects. Thorve et al. [4] complement the results of Luo et al. and Palomba and Zaidman, but they also report two additional test smells (user interface and program logic) that are related to TF in Android Apps. Bell et al. in [34] and proposed a new technique called DeFlaker, which monitors the latest code coverage and marks the test case as flaky if the test case does not execute any of the changes. Another technique called PRADET [35] does not detect flaky tests directly, rather it uses a systematic process to detect problematic test order dependencies. These test order dependencies can lead to flakiness. King et al. in [36] present an approach that leverages Bayesian networks for flaky test classification and prediction. This approach considers flakiness as a decease mitigated by analyzing the symptoms and possible causes. Teams using this technique improved CI pipeline stability by as much as 60%. To best of our knowledge, no study has been conducted to evaluate the predictive accuracy of machine learning classifiers that can help developers in flaky test case prediction and detection.

Dutta et al. [37] and Sjobom [38] investigated projects written in `Python` language to classify test smells that increase test flakiness. Their study is limited to list the test smells and their effect on test flakiness. Our study worked with the test smells identified in [38]. Pinto et al. evaluated five machine learning classifiers (Random Forest, Decision Tree, Naive Bayes, Support Vector Machine, and Nearest Neighbour) to generate flaky test vocabulary written in `Java` [23]. The concluded that Random Forest and SVM performed very well with high precision and recall. They concluded that features such as "job", "action", and "services" were commonly associated with flaky tests. We replicated the similar experiment with different programming language and extended the current knowledge by answering RQ2 and RQ3.

7. Validity Threats

The authors in this study selected only those ML classifiers which have established a good reputation of high accuracy in spam detection thus reducing the selection bias.

The authors in this study reduced the experimenter bias by performing several experiments with different thresholds (i.e., probability scores, kernels, number of trees, etc.) before selecting a champion.

External validity refers to the possibility of generalizing the findings, as well as the extent to which the findings are of interest to other researchers and practitioners beyond those associated with the specific case being investigated. Since the precision strongly depends on the data set under investigation, we have an external validity threat. We cannot generalize the findings of this study for other data set.

8. Conclusion

At the moment of writing this paper, literature is scarce on test flakiness (i.e., root causes, challenges, mitigation strategies, etc.) which requires significant attention from researchers and practitioners. We extracted flaky and non flaky test case contents from open source repositories. We implemented three ML classifiers such as Naive Bayes, Support Vector Machine and Random Forest to see if the predictive accuracy can be increased. The authors concluded that only RF performs better when it comes to precision (i.e., > 90%) but the recall is very low (< 10%) as compared to NBL (i.e., precision < 70% and recall >30%) and SVM (i.e., precision < 70% and recall >60%). The authors concluded that predicting accuracy of ML classifiers are strongly associated with the lexical information of test cases (i.e., test cases written in `Java` or `Python`). The authors investigated why other classifiers failed to produce expected results and concluded that; 1) it is a combination of the test smell and an external environment that makes a test case flaky, and in this study, the external environment was not taken into consideration, 2) ML classifiers should not only consider the frequency of test smells in the test case but other important test codes that have an ability to cancel the effect of test smells.

References

[1] Q. Luo, F. Hariri, L. Eloussi, D. Marinov, An Empirical Analysis of Flaky Tests, in: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, ACM, New York, NY, USA, 2014, pp. 643–653. URL: http://doi.acm.org/10.1145/2635868.2635920. doi:10.1145/2635868.2635920, event-place: Hong Kong, China.

[2] F. Palomba, A. Zaidman, Does Refactoring of Test Smells Induce Fixing Flaky Tests?, in: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 1–12. doi:10.1109/ICSME.2017.12.

[3] F. Palomba, A. Zaidman, The smell of fear: on the relation between test smells and flaky tests, Empirical Software En-

gineering 24 (2019) 2907–2946. URL: https://doi.org/10.1007/s10664-019-09683-z. doi:10.1007/s10664-019-09683-z.

[4] S. Thorve, C. Sreshtha, N. Meng, An Empirical Study of Flaky Tests in Android Apps, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 534–538. doi:10.1109/ICSME.2018.00062.

[5] V. Garousi, B. Küçük, Smells in software test code: A survey of knowledge in industry and academia, Journal of Systems and Software 138 (2018) 52–81. URL: http://www.sciencedirect.com/science/article/pii/S0164121217303060. doi:10.1016/j.jss.2017.12.013.

[6] R. Shams, R. E. Mercer, Classifying Spam Emails Using Text and Readability Features, in: 2013 IEEE 13th International Conference on Data Mining, 2013, pp. 657–666. doi:10.1109/ICDM.2013.131, iSSN: 2374-8486.

[7] S. K. Tuteja, N. Bogiri, Email Spam filtering using BPNN classification algorithm, in: 2016 International Conference on Automatic Control and Dynamic Optimization Techniques (ICACDOT), 2016, pp. 915–919. doi:10.1109/ICACDOT.2016.7877720, iSSN: null.

[8] E. Sahın, M. Aydos, F. Orhan, Spam/ham e-mail classification using machine learning methods based on bag of words technique, in: 2018 26th Signal Processing and Communications Applications Conference (SIU), 2018, pp. 1–4. doi:10.1109/SIU.2018.8404347, iSSN: null.

[9] K. Mathew, B. Issac, Intelligent spam classification for mobile text message, in: Proceedings of 2011 International Conference on Computer Science and Network Technology, volume 1, 2011, pp. 101–105. doi:10.1109/ICCSNT.2011.6181918, iSSN: null.

[10] A. B. M. S. Ali, Y. Xiang, Spam Classification Using Adaptive Boosting Algorithm, in: 6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007), 2007, pp. 972–976. doi:10.1109/ICIS.2007.170, iSSN: null.

[11] R. K. Yin, Case study research design and methods, 4th ed ed., Thousand Oaks, Calif Sage Publications, 2009. URL: https://trove.nla.gov.au/work/11329910.

[12] A. A. Alurkar, S. B. Ranade, S. V. Joshi, S. S. Ranade, P. A. Sonewar, P. N. Mahalle, A. V. Deshpande, A proposed data science approach for email spam classification using machine learning techniques, in: 2017 Internet of Things Business Models, Users, and Networks, 2017, pp. 1–5. doi:10.1109/CTTE.2017.8260935, iSSN: null.

[13] S. Vahora, M. Hasan, R. Lakhani, Novel approach: Naïve Bayes with Vector space model for spam classification, in: 2011 Nirma University International Conference on Engineering, 2011, pp. 1–5. doi:10.1109/NUiConE.2011.6153245, iSSN: 2375-1282.

[14] M. R. Islam, W. Zhou, M. U. Choudhury, Dynamic Feature Selection for Spam Filtering Using Support Vector Machine, in: 6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007), 2007, pp. 757–762. doi:10.1109/ICIS.2007.92, iSSN: null.

[15] T.-Y. Yu, W.-C. Hsu, E-mail Spam Filtering Using Support Vector Machines with Selection of Kernel Function Parameters, in: 2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC), 2009, pp. 764–767. doi:10.1109/ICICIC.2009.184, iSSN: null.

[16] E. G. Dada, J. S. Bassi, H. Chiroma, S. M. Abdulhamid, A. O. Adetunmbi, O. E. Ajibuwa, Machine learning for email spam filtering: review, approaches and open research problems, Heliyon 5 (2019) e01802. URL: http://www.sciencedirect.com/science/article/pii/S2405844018353404. doi:10.1016/j.heliyon.2019.e01802.

[17] R. Caruana, A. Niculescu-Mizil, An empirical comparison of supervised learning algorithms, in: Proceedings of the 23rd international conference on Machine learning, ICML '06, Association for Computing Machinery, Pittsburgh, Pennsylvania, USA, 2006, pp. 161–168. URL: https://doi.org/10.1145/1143844.1143865. doi:10.1145/1143844.1143865.

[18] C.-Y. Chiu, Y.-T. Huang, Integration of Support Vector Machine with Naïve Bayesian Classifier for Spam Classification, in: Fourth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2007), volume 1, 2007, pp. 618–622. doi:10.1109/FSKD.2007.366, iSSN: null.

[19] Z. Jia, W. Li, W. Gao, Y. Xia, Research on Web Spam Detection Based on Support Vector Machine, in: 2012 International Conference on Communication Systems and Network Technologies, 2012, pp. 517–520. doi:10.1109/CSNT.2012.117, iSSN: null.

[20] A. S. Katasev, L. Y. Emaletdinova, D. V. Kataseva, Neural Network Spam Filtering Technology, in: 2018 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM), 2018, pp. 1–5. doi:10.1109/ICIEAM.2018.8728862, iSSN: null.

[21] M. K., R. Kumar, Spam Mail Classification Using Combined Approach of Bayesian and Neural Network, in: 2010 International Conference on Computational Intelligence and Communication Networks, 2010, pp. 145–149. doi:10.1109/CICN.2010.39, iSSN: null.

[22] L. Firte, C. Lemnaru, R. Potolea, Spam detection filter using KNN algorithm and resampling, in: Proceedings of the 2010 IEEE 6th International Conference on Intelligent Computer Communication and Processing, 2010, pp. 27–33. doi:10.1109/ICCP.2010.5606466, iSSN: null.

[23] G. Pinto, B. Miranda, S. Dissanayake, What is the Vocabulary of Flaky Tests? (2020) 11.

[24] W. Lam, R. Oei, A. Shi, D. Marinov, T. Xie, iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests, in: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), 2019, pp. 312–322. doi:10.1109/ICST.2019.00038, iSSN: 2159-4848.

[25] M. Sasaki, H. Shinnou, Spam detection using text clustering, in: 2005 International Conference on Cyberworlds (CW'05), 2005, pp. 4 pp.–319. doi:10.1109/CW.2005.83, iSSN: null.

[26] T. Fawcett, An introduction to ROC analysis, Pattern Recognition Letters 27 (2006) 861–874. URL: http://www.sciencedirect.com/science/article/pii/S016786550500303X. doi:10.1016/j.patrec.2005.10.010.

[27] D. Zhang, J. Wang, X. Zhao, Estimating the Uncertainty of Average F1 Scores, in: Proceedings of the 2015 International Conference on The Theory of Information Retrieval, ICTIR '15, Association for Computing Machinery, Northampton, Massachusetts, USA, 2015, pp. 317–320. URL: https://doi.org/10.1145/2808194.2809488. doi:10.1145/2808194.2809488.

[28] Wang, Baselines and bigrams | Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Short Papers - Volume 2, ???? URL: https://dl-acm-org.e.bibl.liu.se/doi/10.5555/2390665.2390688.

[29] S. Abu-Nimeh, D. Nappa, X. Wang, S. Nair, A comparison of machine learning techniques for phishing detection, in: Proceedings of the anti-phishing working groups 2nd annual eCrime researchers summit on - eCrime '07, ACM Press, Pittsburgh, Pennsylvania, 2007, pp. 60–69. URL: http://portal.acm.org/citation.cfm?doid=1299015.1299021. doi:10.1145/1299015.1299021.

[30] L. Breiman, Random Forests, Machine Learning 45 (2001) 5–32. URL: https://doi.org/10.1023/A:1010933404324. doi:10.1023/A:1010933404324.

[31] I. H. Witten, E. Frank, Data mining: practical machine learning tools and techniques with Java implementations, ACM SIGMOD Record 31 (2002) 76–77. URL: https://doi.org/10.1145/

507338.507355. doi:10.1145/507338.507355.

[32] Weka 3 - Data Mining with Open Source Machine Learning Software in Java, ???? URL: https://www.cs.waikato.ac.nz/ml/weka/index.html.

[33] A. Ahmad, O. Leifler, K. Sandahl, Empirical Analysis of Factors and their Effect on Test Flakiness - Practitioners' Perceptions, arXiv:1906.00673 [cs] (2019). URL: http://arxiv.org/abs/1906.00673, arXiv: 1906.00673.

[34] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, D. Marinov, DeFlaker: Automatically Detecting Flaky Tests, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), 2018, pp. 433–444. doi:10.1145/3180155.3180164.

[35] A. Gambi, J. Bell, A. Zeller, Practical Test Dependency Detection, in: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), 2018, pp. 1–11. doi:10.1109/ICST.2018.00011.

[36] T. M. King, D. Santiago, J. Phillips, P. J. Clarke, Towards a Bayesian Network Model for Predicting Flaky Automated Tests, in: 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), IEEE Comput. Soc, Lisbon, 2018, pp. 100–107. doi:10.1109/QRS-C.2018.00031.

[37] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, S. Misailovic, Detecting flaky tests in probabilistic and machine learning applications, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, Association for Computing Machinery, New York, NY, USA, 2020, pp. 211–224. URL: https://doi.org/10.1145/3395363.3397366. doi:10.1145/3395363.3397366.

[38] A. Sjöbom, Studying Test Flakiness in Python Projects : Original Findings for Machine Learning, 2019. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-264459.

Towards the Identification of Process Anti-Patterns in Enterprise Architecture Models

Barry-Detlef Lehmann[a], Peter Alexander[a], Horst Lichter[a] and Simon Hacks[b]

[a]RWTH Aachen University, Research Group Software Construction, Aachen, Germany
[b]KTH Royal Institute of Technology, Network and Systems Engineering, Stockholm, Sweden

Abstract

IT processes constitute the backbone of an integrated enterprise architecture (EA). The model thereof sustains the development and management of the EA. Nevertheless, the quality of such models tends to degrade over time due to, e.g. improper modeling practices or ineffective evaluation. In this regard, the knowledge of relevant modeling anti-patterns can help identify, mitigate, and prevent the occurrence of sub-optimal or adverse constructs in the model. In the field of business process modeling (BPM), a plethora of BPM anti-patterns has been defined and compiled in various taxonomies. However, these BPM anti-patterns mostly focus on technical issues, which thus are applicable for evaluating workflows but not EA-level processes. We strongly argue that the concept of process anti-pattern in EA domain can facilitate EA analyses on process-related issues. To address this gap, this paper presents a catalogue of 18 EA process modeling anti-patterns, which we derived from the existing BPM anti-patterns. Our result should serve as food for thought and motivation for future research in this context.

Keywords

enterprise architecture, process anti-pattern, model quality

1. Introduction

IT processes transform fragmented capabilities within an enterprise architecture (EA) into consolidated business assets. The models thereof often need to be consulted or even adapted in the efforts of managing and evolving the EA. Nevertheless, these process models are often developed with less consideration of quality due to, e.g. time pressure, little awareness of good modeling practices, or inadequate evaluation of the models. The uncontrolled development in this manner will eventually render the process models useless or even misleading [1]. This situation may hamper the sustainability of EA practices within the organization.

To avoid this, the development and evaluation of process models must be guided by the knowledge of relevant patterns and anti-patterns. In general, the *modeling pattern* is defined as a proven solution to a recurring modeling problem whereas the *modeling anti-pattern* is defined as a modeling solution that is known to pose risks [2]. The understanding of these concepts can help identify, mitigate, and prevent the occurrence of sub-optimal or adverse constructs within the models [1]. In this study, we focus on the concept of process anti-pattern in the context of EA modeling, specifically the application thereof in the evaluation of EA models.

In the field of business process modeling (BPM) research, a plethora of BPM anti-pattern taxonomies have been proposed [1, 3, 4]. However, these BPM anti-patterns mostly address rather technical aspects like the use of syntax or layout in the process model, which are very specific to the modeling notations in use. Moreover, discussions of BPM anti-patterns have mostly been presented in workflow modeling notations [5] (e.g. BPMN). These situations hinder the application of BPM anti-pattern to EA practices, in which processes are viewed from rather strategic perspectives and modeled in EA modeling notations. We strongly argue that transferring the existing BPM anti-patterns into process anti-patterns in the domain of EA can help improve the development of processes and their quality that underlie the EA.

The effort of transferring an existing concept into the domain of EA is not new. Salentin and Hacks introduced the concept of *EA smell*, which is defined as a hint to a bad habit that impairs the quality of the EA [6]. In their work, they transferred the concept of code smell into the context of EA through conceptual derivation and transformation methodology. Inspired by their work, the same methodology is applied in this study to answer the following research question (RQ):

RQ What process anti-patterns can be defined to support EA modeling activities through the analysis of published process anti-patterns?

The remainder of this paper is structured as follows: Section 2 gives an overview of previous studies on anti-pattern or other related concepts (e.g. smell) in the fields

QuASoQ 2020: 8th International Workshop on Quantitative Approaches to Software Quality, December 01, 2020, Singapore
barry.lehmann@rwth-aachen.de (B. Lehmann); alexander@swc.rwth-aachen.de (P. Alexander); lichter@swc.rwth-aachen.de (H. Lichter); shacks@kth.se (S. Hacks)
0000-0001-6534-278X (P. Alexander); 0000-0002-3440-1238 (H. Lichter); 0000-0003-0478-9347 (S. Hacks)

of BPM and EA modeling research; section 3 describes our methodology for obtaining process anti-patterns for EA modeling problems; section 4 elaborates our findings and the analysis thereof; section 6 demonstrates our results and discusses the implications as well as threats to the validity thereof; and section 7 motivates future research directions and concludes this paper.

2. Related Work

The concept of anti-pattern was coined in 1995 by Koenig [7] to describe a common solution to a recurring problem which poses risks of being counterproductive. Although an anti-pattern may serve as a practical short-term solution, the use of it sets a context in which certain changes may become more expensive or impossible. The (unintentional) use of anti-pattern is highly influenced by, e.g. time pressure, inadequate knowledge of best practices, or unforeseen changes.

The spectrum of studies about anti-patterns covers a wide-range of software engineering topics, such as software development and modeling. In the domain of BPM, a number of taxonomies of process anti-patterns have been proposed, each of the taxonomies addresses a specific area of concern. In 2019, a bibliography of all these taxonomies was published based on a literature review study [1]. Therein, the collected taxonomies are divided into seven categories based on the addressed modeling problems. Furthermore, this study suggests several rules of thumb in documenting process anti-patterns. The authors of this study advocate the use of this bibliography in the efforts to increase the quality of BP models.

Analogous with the concept of process anti-pattern, the concept of *process anomaly* is also known in BPM research. Vidacic and Strahonja present a literature review of this concept in which the collected process anomalies are divided into three categories: structural, semantical, and syntactical anomalies [8]. They also suggest approaches to the mitigation or prevention thereof. Suchenia et al. provide a brief overview of BPM anti-patterns, present these in BPMN models, and categorize these into three categories: syntactic, structural, and control flow anti-patterns [9]. Trcka et al. present data-flow anti-patterns and an approach to identifying these [10]. Further in this topic, Sadiq et al. identify seven common data-flow anti-patterns and provide the basic algorithm to address these [11]. Finally, Döhring and Heublein present a taxonomy of control-flow, rule-based, and data-flow anti-patterns; demonstrate examples of such anti-patterns in BPMN; and suggest detection as well as prevention mechanisms thereof [12].

In the domain of EA modeling, the concept of anti-pattern remained unknown until the recent suggestion of an EA smell taxonomy by Salentin and Hacks. Therein, the authors provide a catalog of 45 EA smells that originated from code smells. In their approach, they transform a catalog of well-known code smells into EA smells and categorize the EA smells based on the three concerns of EA: business, application, and technology. Furthermore, they present a tool that can detect 14 EA smells. As an extension to their work, this study explores the current knowledge about process anti-pattern to obtain a new understanding thereof in the EA domain.

Finally, the idea of looking at different abstraction levels in processes to address different stakeholders is not new. Several studies have been conducted to decompose processes into different abstraction levels. Giachetti proposes to divide the process hierarchy into functions, processes, sub-processes, activities, and tasks [5]. This study argues that the natural hierarchical attribute of the process should be used. Viljoen, on the other hand, decomposes it into enterprise model, macro, business process, sub-process, activity, and task [13]. Koschmider and Blanchard propose a semiautomatic detection for different process abstraction levels with processes modeled with Petri Nets [14]. Their goal is to detect a process hierarchy in a process model. All these studies commonly advocate that the levels of abstraction applied to processes should meet the goals of relevant stakeholders.

3. Methodology

To transfer the existing knowledge about process anti-pattern into the EA domain, this study follows the methodologies proposed by Peffers et al. and Hevner et al. for the Design Science Research (DSR) [15], [16]: a type of research which aims to devise an artifact that addresses a "heretofore unsolved and important business problem" by drawing on the existing knowledge. The resulting artifact must be rigorously evaluated in terms of its "utility, quality, and efficacy" and effectively communicated to the relevant audience.

This study is performed as follows: At first, we collected knowledge about the 336 already-published BPM anti-patterns in scientific literature, which have been compiled in [1] and are publicly accessible on [17]. Based on a mapping of modeling notations between BPMN and ArchiMate, we processed these BPM anti-patterns and finally derived 18 process anti-patterns of EA relevance, which are then documented in a structured template and exemplified in ArchiMate models. A closer look into this procedure is provided in the following subsections.

3.1. Notation Mapping

Most BPM anti-patterns have been analyzed and visualized in BPMN. While BPMN provides a full-fledged framework to create graphical business processes mod-

BPMN	ArchiMate	Source
Business Process Diagram, Pool, Lanes	Process	[21, 19]
Activities, Task, Sub-Process	Function	[21]
Collaboration Diagram	Interaction	[21]
Event	Event	[21, 18, 19]
Data Object	Object	[21, 19]
Lane	Business Role, Business Actor, Application Component	[21, 19]
Sequence flow	Triggering, Flow	[20]
Data association	Access	[20]
Inclusive and parallel gateways	And-Junction	[20, 18]
Exclusive and event-based gateways	Or-Junction	[20, 18]

Table 1
Mapping from BPMN elements to ArchiMate

els, BPMN is intended for rather detailed business process modeling, such as the modeling of conversation, choreography, and collaboration models. However, such business process models constitute only a small area within the broad view of EA. Therein, processes are analyzed from rather strategic perspectives, e.g. the connection of all high-level processes to the surrounding organization units to achieve strategic targets. The modeling of such perspectives has been facilitated by a number of EA modeling languages; the most popular one is ArchiMate. We strongly argue that method supports for analyzing EA processes must be built on top of an EA modeling language. Therefore, to transfer the existing BPM anti-patterns into the EA domain, we first need to create a mapping between BPMN and ArchiMate to figure out the possibilities of deriving something of EA relevance.

Previous studies have suggested several mappings between BPMN and ArchiMate [18, 19, 20, 21]. They show that both modeling languages share some conceptual similarities. Firstly, both provide similar notations for connecting process elements (e.g. the sequence, default, and conditional flows) and regulating these with gateways or junctions. Secondly, both support the modeling of similar relationships between processes and the related elements [19]. For example, BPMN's support for creating relationships between e.g. activities and data objects is similar to ArchiMate's support for creating relationships between e.g. business processes and business objects. In table 1, we list the mappings of notations which we use as a basis to conduct the next steps.

3.2. Transformation Design

From our analysis of related studies in BPM research, we found several taxonomies which collect in total of 336 BPM anti-patterns and classify these based on the characteristics thereof [1] [17]. From the perspective of our study, we argue that the depth of these taxonomies stretches beyond the context of EA analysis. For example, the various situations of deadlock are addressed by different BPM anti-patterns. While such specialized modeling problems are indeed identifiable within BPMN models, such detailed problems are not visible through the high-level notations of ArchiMate. Therefore, we pruned such detailed branches of classification within the selected taxonomies, thereby reducing the size of our analysis to 200 BPM anti-patterns.

Next, this study follows a defined procedure to derive process anti-patterns of EA relevance, which is as follows: Firstly, BPMN elements found in the names and descriptions of the BPM anti-patterns are translated into ArchiMate elements based on the notation mapping shown in table 1. Secondly, each translated description is evaluated to determine whether it is comprehensible within the context of EA. Through this step, we derived 18 process anti-patterns of EA relevance, while excluding the rest because the translated descriptions thereof do not describe relevant EA modeling problems (e.g. anti-patterns related to syntax errors such as the *sequence flow crosses process boundary* [3]). Among the derived process anti-patterns, some are directly applicable in the EA domain only after applying the notation mapping (e.g. the *Layout Deficit* [1]), while some other require broad modification in the description thereof to depict valid EA modeling problems (e.g. *Useless Test*).

4. An initial catalogue of process anti-patterns in EA

To support the usability of our contribution, we have organized the resulting 18 process anti-patterns in a catalogue, which is publicly accessible on our [22]. Since this catalogue is still in its initial stages, we encourage further extensions and improvements to it. A detailed discussion on future research directions is given in section 7.

In this section, we first introduce the categorization used in the catalogue. Next, we provide a deeper look

into one process anti-pattern under each category. Lastly, we describe the template used to document some general attributes of the process anti-patterns.

4.1. Categorizing process anti-patterns

To present the proposed catalogue in an organized and systematic manner, we divide the contained process anti-patterns into the following five categories that we derived from some selected categories in [1].

The category of **semantic error** includes process anti-patterns that address the semantically error or inconsistent parts of the model under analysis. These problems are commonly caused by the improper or deficient use of modeling notations, which may distort the understanding of the model. It is worth to mention that such semantic errors should not be confused with syntax errors as the latter addresses the violation against the rules of assembling the notations, whereas the earlier addresses the false or ambiguous impression conveyed by the model [3]. Due to the gap between the syntactic rules of BPMN and ArchiMate, we derived no process anti-patterns of EA relevance under the syntax error category.

The category of **control-flow problem** includes process anti-patterns that address the flawed concurrency of process flows which raises unpredictability in the final outcome. Such issue occurs when the modeled processes are split or joined without properly considering the influence thereof to the final outcome.

The category of **understandability problem** includes process anti-patterns that address the excessive or lack of complexity within the model, thereby requiring excessive efforts to understand the process under analysis in its full context. Such an issue is commonly caused by the improper/inconsistent level of information granularity or inadequate/imprecise coverage of the actual process in reality.

The category of **rule-related defect** includes process anti-patterns that address the contradictions among the rules specified within the model. Such an issue may occur when e.g. the actual process is not well-defined, well-communicated, or well-understood among some contributors to the rules specification in the model.

The category of **data-flow related defect** includes process anti-patterns that address the proneness to conflicts when the same data object is concurrently used for different kinds of transactions. These issues are commonly caused by the overlapping data responsibilities among different processes or the centralization of too many data in a single data object.

4.2. Demonstrating process anti-patterns

Next, we name the 18 process anti-patterns and map these under the aforementioned categories, as listed in table 2.

Table 2
Categories of EA anti-pattern

Category	Anti-pattern
Semantic Error	End event missing
	Start event missing
Control-flow Problem	Dead Element
	Deadlock
	Infinite loop
	Lack of synchronization
	Undefined junction condition
Understand-ability Problem	Junction named as element
	Layout deficit
	Language deficit
	Missing negative case
	The word And in element name
	The word Or in element name
	Useless test
Rule-related Defect	Contradiction in input
Data-flow related Defect	Inconsistent data
	Mismatched data
	Missing data

Furthermore, to give a closer look into the catalogue, we elaborate one process anti-pattern under each category and provide an example thereof.

Under the category of *semantic error*, the **end event missing** occurs when the modeled process does not clearly specify the end events [3] [23], which then causes confusion or misinterpretation about the valid conditions to finalize or abort the process execution. An example of this problem is when multiple high-level processes of different organization bodies are integrated within the EA model without specifying the points when the collaborative outcomes have been achieved or any disruptions have to be handled. A solution to this anti-pattern would be to simply introduce *end event elements* that clearly signal all possible ways to end the process.

Under the category of *control-flow problem*, the **lack of synchronization** refers to the situation when the modeled process does not specify a proper synchronization among concurrent process flows, thereby showing no predictable outcome. An example of such a situation is when the modeled process is split by an *AND junction* and later joined by an *OR junction* [24, 25]. A solution to this anti-pattern is to ensure the highest test coverage of all points of synchronization specified in the model.

Under the category of *understandability problem*, the **useless test** is identified when the modeled process fulfills only some of all the possible cases in reality [26], thereby making it impossible to identify and test the real extent of the supported problem domain. An example thereof is when the handling of possible mistakes or disruption during the process execution is not specified within the model. As a solution, the model should be

Attribute	Meaning
Name	Gives the anti-pattern a meaningful designator
Problem	Describes why the anti-pattern leads to problems
Consequences	Describes what the consequences of the anti-pattern are
Solution	Describes a solution to the anti-pattern
Graphical Definition	Shows a graphical representation of the anti-pattern to reduce misinterpretations

Table 3
Documentation attributes of EA process anti-pattern (adopted from [1]).

incrementally and iteratively developed along with the continuous identification of relevant test cases until it reaches a reasonable level of complexity and covers the complete problem domain [27].

Under the category of *rule-related defect*, the **contradiction in input** occurs when some rules applied to the modeled process may contradict with each other, thereby hindering the process execution to continue as intended [12]. An example thereof is when a certain data object passes the input validations specified on a junction despite being actually invalid for the supplied process. A solution to this anti-pattern is to continuously perform a rigorous combinatorial testing of all possible input types and all the rules applied to the modeled process.

Under the category of *data-flow-related defect*, the **inconsistent data** describes the situation in which data objects (e.g. customer records or insurance claim) are accessed by concurrent process flows, thereby making it prone to data handling mistakes. An example thereof is when multiple processes work on duplicates of the same data object, and a (manual) synchronization procedure between the duplicates is required after every modification on one side. To mitigate this, the strategy of handling the data must be carefully defined and implemented.

4.3. Documenting process anti-patterns

In general, the documentation of *modeling anti-patterns* includes many attributes of *modeling patterns* [28] together with some other attributes like *cause* and *detection* [29]. To document the process anti-patterns identified in this study, we derive some attributes from the templates for documenting BPM anti-patterns introduced in [1], as shown in table 3. Please note that, at the time of this writing, not all attributes have been completed for each process anti-pattern due to the need of further information and analysis.

5. Applying process anti-patterns in EA

In order to illustrate the concept and the usage of process anti-pattern in the context of EA, we analyze a slightly modified ArchiMate EA model and annotate it with anti-pattern information. The model used, depicted in fig. 1, is contained in a publicly accessible collection of ArchiMate example EA models [30]. This model defines how new orders are processed. After a new order is received, planning the order and evaluating the customers' credibility are done in parallel. After an approved proposal is available, the customer signs the respective contract to accept the proposal.

When analysing the model, it can be observed that the processes Evaluate customer credit and Plan order do not wait for each completion before continuing to the Develop approved proposal process, which may lead to undesired results (e.g. the contract does not consider the customer's credibility). These are consequences of the *lack of synchronization* anti-pattern.

When analyzing the usage of the Order Data element, we identify the *inconsistent data* anti-pattern because this data can be changed without rerunning dependent processes (e.g. Evaluate customer credit).

Furthermore, the Evaluate customer credit process exhibits the *useless test* anti-pattern, as only the positive test result is modeled.

Next, we can identify the anti-pattern *end event missing* because no clear termination is defined for this EA process model. The process could end in Refuse proposal or in Accept proposal.

Finally, we detect the *contradiction in input* anti-pattern at the Or-Junction that splits the control flow after Develop approved proposal. There, the incorrect condition will never lead to the Refuse proposal process and therefore makes it a dead process.

6. Discussion

Some of the main goals of applying the EA discipline within an enterprise is to ensure the business-IT align-

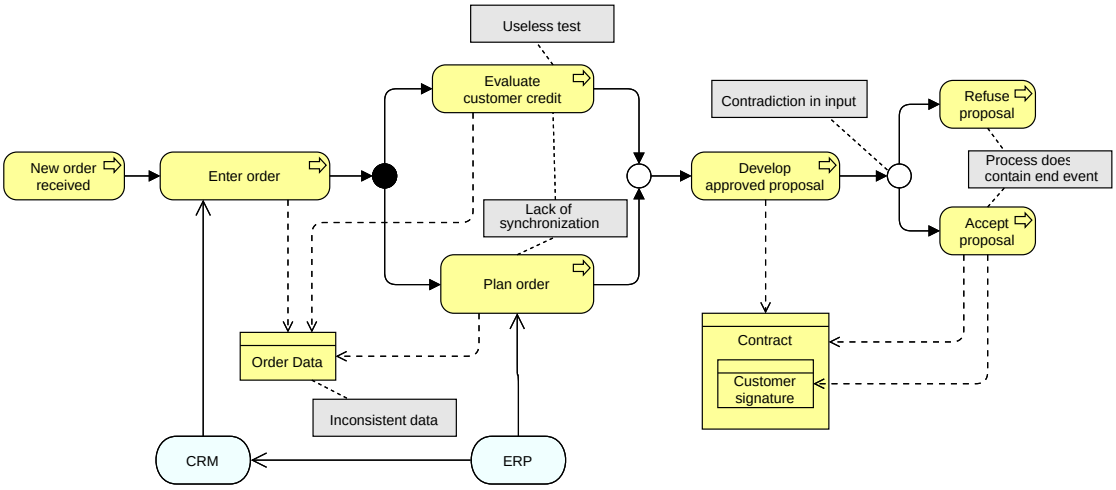


Figure 1: An example EA ArchiMate model with annotated process anti-patterns

ment [31] and to develop solid IT strategies that can help achieve strategic targets [32]. For this reason, this section seeks to answer the ultimate question of this study: "how can enterprise architects benefit from the contribution of this study?" In this section, we describe the use of the proposed process anti-patterns to support both research and practice of EA methods. Following to this, we discuss the threats to the validity of our results.

6.1. Implication for researchers & practitioners

The concept of *EA debt* has recently been introduced as the deviation between the current state and the hypothetical ideal state of the enterprise [33]. Factors to such deviation in EA (e.g. sub-optimal or adverse solution design) are likely to be identifiable within the EA models created or used during the planning, development, evaluation, or communication of the EA [6]. Therefore, the ability to recognize the existence of such deviation in EA models is needed, and the concept of process anti-pattern proposed in this study is intended to support such ability with sharp focus on processes. In this case, practitioners can use the process anti-patterns catalogue to scan the EA models for potential *EA Debt*. In any case, it is beneficial that the practitioners are aware of the potentially vulnerable parts of the EA Model as further development could be hindered if these remain ignored [6].

Furthermore, we also intend to impart food for thought into the EA research community and provide a basis for further research works in this topic. For example, the process anti-patterns identified in this study might help researchers to extend the automatic detection of EA anti-patterns in an EA model, as has been drafted in a program that currently detects 14 EA smells [6].

6.2. Threats to validity

The results of this study have to be seen in the light of some limitations. The limitations that affect the results of this paper are the lack of previous research and bias during the anti-pattern transformation.

Lack of previous research. There is little research on transforming low-level process methods to be applicable for high-level processes and even much less on bringing together the modeling notations for such processes. In addition, the already suggested mappings between the modeling notations for low-level and high-level processes are rather described as informal and lack of theoretical foundations, thereby leaving room for interpretation or different mapping solutions. This might reduce the validity of our results because our approach relies on the existing mapping between BPMN and ArchiMate.

Bias during the anti-pattern transformation. To reduce bias when selecting the relevant BPM anti-patterns to be transformed, we first need to establish objective selection criteria. Thus, we defined a mapping of notations and apply it on the collected BPM anti-patterns to prune the ones that do not fit in the new domain. Despite this, subjective assessment is still inevitably involved during the process, thereby leaving room for interpretation and may not produce unique results.

7. Conclusion & Future Works

The concept of anti-pattern has been long known to help recognize common solutions that are not sustainable for the future development. However, little emphasis has been put on studying this concept in the context of EA. The first step in this direction has recently been made to transfer the existing code smells into the EA domain, out of which an initial catalogue of 45 EA smells has been developed and proposed [6]. To pursue a meaningful extension to this result, this study focuses on transferring the existing BPM anti-patterns [1] into process anti-patterns for EA modeling problems, with reference to a mapping between the notations of BPMN and ArchiMate.

The process anti-patterns identified in this study are compiled in a catalogue that is publicly accessible on our [22]. Therein, the process anti-patterns are categorized and documented in a well-known template to ease the use or extension thereof by EA practitioners and researchers. The practical use of this catalogue covers a broad range of topics, starting from the identification of flaws in EA models to the identification of EA debt. Nevertheless, this catalogue is still in its initial stages. Much more information and analyses are needed before this catalogue can be evaluated in real industrial contexts. Therefore, to motivate further research in this topic, the rest of this section outlines some ideas of future research directions.

The future research directions in this topic can be divided into three main topics: to pursue different methods for defining new EA anti-patterns, to perform empirical studies for improving both conceptual and practical knowledge in this context, and to develop tool supports for the automatic detection of the anti-patterns in EA models. In terms of analyzing more methods to find EA anti-patterns, we suggest to investigate new domains (e.g. documentation, data, or requirement anti-patterns) to extend the catalogue with adapted or new EA anti-patterns. Also, the investigation of cause-and-effect relationships among the identified EA anti-patterns may provide insights on the possible propagation of impacts thereof. In terms of empirical studies, evaluations using real EA models from different business domains can help to verify and improve the quality of the catalogue and the proposed method supports. Finally, tool supports can be developed to support the continuous detection of EA anti-patterns or the (early) signs of their occurrences.

References

[1] A. Koschmider, R. Laue, M. Fellmann, Business process model anti-patterns: a bibliography and taxonomy of published work, in: J. vom Brocke, S. Gregor, O. Müller (Eds.), 27th European Conference on Information Systems - Information Systems for a Sharing Society, ECIS 2019, Stockholm and Uppsala, Sweden, June 8-14, 2019, 2019. URL: https://aisel.aisnet.org/ecis2019_rp/157.

[2] M. Fellmann, A. Koschmider, R. Laue, A. Schoknecht, A. Vetter, Business process model patterns: State-of-the-art, research classification and taxonomy, Business Process Management Journal (2018). doi:10.1108/BPMJ-01-2018-0021.

[3] T. Rozman, G. Polančič, R. V. Horvat, Analysis of most common process modelling mistakes in bpmn process models, in: EuroSPI 2007, Industrial Proceedings, 2007, pp. 1.21 – 1.31. URL: https://2020.eurospi.net/images/proceedings/EuroSPI2007-ISBN-978-3-9809145-6-7.pdf.

[4] S. Von Stackelberg, S. Putze, J. Mülle, K. Böhm, Detecting data-flow errors in bpmn 2.0, Open Journal of Information Systems (OJIS) 1 (2014) 1–19.

[5] R. E. Giachetti, Design of Enterprise Systems: Theory, Architecture, and Methods, 1st ed., CRC Press, Inc., USA, 2010.

[6] J. Salentin, S. Hacks, Towards a catalog of enterprise architecture smells, in: N. Gronau, M. Heine, H. Krasnova, K. Poustcchi (Eds.), Entwicklungen, Chancen und Herausforderungen der Digitalisierung: Proceedings der 15. Internationalen Tagung Wirtschaftsinformatik, WI 2020, Potsdam, Germany, March 9-11, 2020. Community Tracks, GITO Verlag, 2020, pp. 276–290. doi:10.30844/wi_2020_y1-salentin.

[7] A. Koenig, Patterns and antipatterns, J. Object Oriented Program. 8 (1995) 46–48.

[8] T. Vidacic, V. Strahonja, Taxonomy of Anomalies in Business Process Models, 2014, pp. 283–294. doi:10.1007/978-3-319-07215-9_23.

[9] A. Suchenia, T. Potempa, A. Ligęza, K. Jobczyk, K. Kluza, Selected Approaches Towards Taxonomy of Business Process Anomalies, volume 658, 2017, pp. 65–85. doi:10.1007/978-3-319-47208-9_5.

[10] N. Trcka, W. M. P. van der Aalst, N. Sidorova, Data-flow anti-patterns: Discovering data-flow errors in workflows, in: P. van Eck, J. Gordijn, R. J. Wieringa (Eds.), Advanced Information Systems Engineering, 21st International Conference, CAiSE 2009, Amsterdam, The Netherlands, June 8-12, 2009. Proceedings, volume 5565 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 425–439. doi:10.1007/978-3-642-02144-2_34.

[11] S. W. Sadiq, M. E. Orlowska, W. Sadiq, C. Foulger, Data flow and validation in workflow modelling, in: K. Schewe, H. E. Williams (Eds.), Database Technologies 2004, Proceedings of the Fifteenth Australasian Database Conference, ADC 2004, Dunedin, New Zealand, 18-22 January 2004, volume 27 of *CRPIT*, Australian Computer Society, 2004, pp. 207–

214. URL: http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV27Sadiq.html.

[12] M. Döhring, S. Heublein, Anomalies in rule-adapted workflows - a taxonomy and solutions for vbpmn, in: 2012 16th European Conference on Software Maintenance and Reengineering, 2012, pp. 117–126.

[13] S. Viljoen, Reflections on business process levelling, white paper, 2012. URL: https://realirm.com/sites/default/files/whitepapers/reflections_on_business_process_leveling_0.pdf.

[14] A. Koschmider, E. Blanchard, User assistance for business process model decomposition, in: Proceedings of the 1st IEEE International Conference on Research Challenges in Information Science, 2007, pp. 445–454.

[15] K. Peffers, T. Tuunanen, M. Rothenberger, S. Chatterjee, A design science research methodology for information systems research 24 (2007) 45–77. doi:10.2753/MIS0742-1222240302.

[16] A. R. Hevner, S. T. March, J. Park, S. Ram, Design science in information systems research, MIS Q. 28 (2004) 75–105. URL: http://misq.org/design-science-in-information-systems-research.html.

[17] Business process model patterns classification, 15.11.2020. URL: http://www.bpmpatterns.org/.

[18] Archimate® 3.1 specification, 27.06.2020. URL: https://pubs.opengroup.org/architecture/archimate3-doc/.

[19] D. Orlovskyi, A. Kopp, Enterprise architecture modeling support based on data extraction from business process models, in: S. Subbotin (Ed.), Proceedings of The Third International Workshop on Computer Modeling and Intelligent Systems (CMIS-2020), Zaporizhzhia, Ukraine, April 27-May 1, 2020, volume 2608 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2020, pp. 499–513. URL: http://ceur-ws.org/Vol-2608/paper38.pdf.

[20] M. Lankhorst, Combining archimate® 3.0 with other standards – bpmn, 6.9.2016. URL: https://bizzdesign.com/blog/combining-archimate-3-0-with-other-standards-bpmn/.

[21] L. Penicina, Linking bpmn, archimate, and BWW: perfect match for complete and lawful business process models?, in: J. Grabis, M. Kirikova, J. Zdravkovic, J. Stirna (Eds.), Short Paper Proceedings of the 6th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modeling (PoEM 2013), Riga, Latvia, November 6-7, 2013, volume 1023 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2013, pp. 156–165. URL: http://ceur-ws.org/Vol-1023/paper15.pdf.

[22] B.-D. Lehmann, Enterprise process anti-pattern, 05.10.2020. URL: https://ea-anti-pattern.pages.rwth-aachen.de/process-anti-patterns/.

[23] C. Dias, V. Stein Dani, J. Mendling, L. Thom, Anti-patterns for Process Modeling Problems: An Analysis of BPMN 2.0-Based Tools Behavior, 2019, pp. 745–757. doi:10.1007/978-3-030-37453-2_59.

[24] Z. Han, P. Gong, L. Zhang, J. Ling, W. Huang, Definition and detection of control-flow anti-patterns in process models, in: 2013 IEEE 37th International Computer Software and Applications Conference Workshops (COMPSACW), IEEE Computer Society, Los Alamitos, CA, USA, 2013, pp. 433–438. doi:10.1109/COMPSACW.2013.111.

[25] R. Laue, A. Awad, Visualization of business process modeling anti patterns, ECEASST 25 (2010). doi:10.14279/tuj.eceasst.25.344.

[26] R. Laue, W. Koop, V. Gruhn, Indicators for open issues in business process models, in: International Working Conference on Requirements Engineering: Foundation for Software Quality, Springer, 2016, pp. 102–116.

[27] P. Desfray, G. Raymond, Chapter 5 - key modeling techniques, in: P. Desfray, G. Raymond (Eds.), Modeling Enterprise Architecture with TOGAF, The MK/OMG Press, Morgan Kaufmann, Boston, 2014, pp. 67 – 91. doi:https://doi.org/10.1016/B978-0-12-419984-2.00005-7.

[28] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, volume 47 of *Addison Wesley Professional Computing Series*, 1995.

[29] J. Bogner, T. Boceck, M. Popp, D. Tschechlov, S. Wagner, A. Zimmermann, Towards a collaborative repository for the documentation of service-based antipatterns and bad smells, in: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), 2019, pp. 95–101. doi:10.1109/ICSA-C.2019.00025.

[30] P. Beauvoir, A collection of archimate® models., 27.06.2020. URL: https://github.com/archimatetool/ArchiModels.

[31] J. Gøtze, The changing role of the enterprise architect, in: 2013 17th IEEE International Enterprise Distributed Object Computing Conference Workshops, 2013, pp. 319–326. doi:10.1109/EDOCW.2013.42.

[32] C. Strano, Q. Rehmani, The role of the enterprise architect, Inf. Syst. E-Business Management 5 (2007) 379–396. doi:10.1007/s10257-007-0053-1.

[33] S. Hacks, H. Höfert, J. Salentin, Y. C. Yeong, H. Lichter, Towards the definition of enterprise architecture debts, in: 23rd IEEE International Enterprise Distributed Object Computing Workshop, EDOC Workshops 2019, Paris, France, October 28-31, 2019, 2019, pp. 9–16. doi:10.1109/EDOCW.2019.00016.

Evaluating the Impact of Inter Process Communication in Microservice Architectures

Benyamin Shafabakhsh[a], Robert Lagerström[b] and Simon Hacks[b]

[a]School of Electrical Engineering and Computer Science,
KTH Royal Institute of Technology,
Stockholm, Sweden

[b]Division of Network and Systems Engineering,
KTH Royal Institute of Technology,
Stockholm, Sweden

Abstract

With the substantial growth of cloud computing over the past decade, microservice architectures have gained significant popularity and have become a prevalent choice for designing cloud-based applications. Microservices based applications are distributed and each service can run on a different machine. Due to its distributed nature, one of the key challenges when designing applications is the mechanism by which services communicate with each other. There are several approaches for implementing inter process communication (IPC) in microservices; each comes with different advantages and trade-offs. While theoretical and informal comparisons exist between them, this paper has taken an experimental approach to compare and contrast the popular forms of IPC communications. Several load test scenarios have been executed to obtain quantitative data related to performance efficiency, and availability of each method. The evaluation of the experiment indicates that, although there is no universal IPC solution that can be applied in all cases, the asynchronous pattern offers various advantages over its synchronous rival.

Keywords

Microservices, Inter Process Communication, IPC, Inter-Service Communication, Distributed Systems, gRPC, RabbitMQ

1. Introduction

Over the past few years, microservices have earned enormous attention and gained popularity from the industry. They helped large organisation such as Amazon and Netflix to serve millions of requests per minutes [1]. Microservice architecture is a style of developing software as a collection of independent services. Each service is running on its own process that is independent from other processes and can be deployed separately from other services [2]. Designing a software based on microservices involves answering questions and overcoming technical challenges that often do not exist in monolithic architecture, like inter process communication (IPC) [3], service discovery [4], decomposition strategy [5], or managing ACID transactions [6].

Despite the growth and importance of microservices in industry, there has not been sufficient research on microservices, partly due to lacking a benchmark system that reflects the characteristics of industrial microservice systems [7]. IPC is one of the important challenges of microservice architectures [8]. In monolithic based systems, components can call each other at the language-level while in microservices each component is running on its own process and possibly on a different machine than other services. The choice of IPC mechanism is an important architectural decision which can impact the software's non-functional requirements [8].

As of today, there are no concrete explanations or any standardized approach that can help to decide the right IPC method when designing microservice based applications. Due to this reason, there is an abundant confusion around the question of when to use which method and what are the trade-offs for choosing that method. Deciding between a synchronous and asynchronous approach is an important decision to take in regards to how services collaborate with each other [9].

There are two questions this paper is working towards answering:

1. From performance efficiency standpoint, what are the implications for utilizing available synchronous and asynchronous methods for implementing IPC in microservice architectures?
2. How does the IPC method choice impact availability of the system?

Woodstock'20: Symposium on the irreproducible science, June 01–05, 2020, Woodstock, NY
EMAIL: bensha@kth.se (B. Shafabakhsh); robertl@kth.se (R. Lagerström); shacks@kth.se (S. Hacks)
ORCID: 0000-0003-3089-3885 (R. Lagerström); 0000-0003-0478-9347 (S. Hacks)

The motive behind selecting the performance efficiency, and availability as the two criteria for this research is that the choice of IPC method directly impacts these two non-functional requirements in a microservices based system, while other non-functional requirements such as security [10] and maintainability [11] can span over few other areas and goes beyond IPC. Being able to measure these qualities in the system are critical in order to achieve an efficient management of any software system [12, 13]. Moreover, the chosen quality attributes are among the top priorities for most modern applications [14, 15].

In this work, we describe a systematic approach for selecting IPC method when designing microservices based software. The remainder of this paper focuses on state of the art in identifying different IPC models in section 2. Next, in section 3, we discuss the development of the prototypes built for the purpose of discovering the relationship between each IPC method and its impact on performance efficiency and availability. We then run a test against each prototype to investigate its outcome and discuss previous work conducted in this domain. Finally, we draw a conclusion in section 6.

2. State of the Art

When designing IPC mechanism, there are two type of interaction style to choose from: synchronous and asynchronous, which we will shortly introduce next.

2.1. Synchronous Communication

Synchronous communication is often regarded as request/response interaction style. One microservice makes a request to another service and waits for the services to process the result and send a response back. In this style, it is common that the requester blocks its operation while waiting for a response from the remote server. Representational state transfer (REST) application programming interfaces (API) [16] and gRPC[1] are the most common framework for implementing Synchronous form of communication in microservices [8].

- **REST API**: REST is an architectural style that is commonly used for designing APIs for modern web services [17]. In a system that uses REST API for its IPC communication, each service typically has its own web-server up and running on a specific port such as 8080 or 443, and each service exposes a set of endpoints to enable the interactions with other microservices and exchange of information between them. The server interacts directly with client through its interface also known as Web API.
- **gRPC**: gRPC is an open source high performance RPC framework designed an developed by Google. Remote procedure call (RPC) is a mechanism used in many distributed applications to facilitate inter process communication. RPC was first implemented by Birrell and Nelson [18] and it has been regarded as a protocol that enables a message exchange between two process with characteristics of low overhead, simplicity and transparency [19]. By default, when a client sends a request to a server it halt the process and waits for the results to be returned. RPC is therefore considered as synchronous form of communication [20]. Figure 1 presents the operational process between client and server in gRPC. In this model, the client implements the same method as its corresponding server through local objects also known as stubs.

2.2. Asynchronous Communication

The asynchronous form of communication can be implemented in microservices when services exchange messages with each other through a message broker. In this form of interaction, the message broker acts as an intermediary between services to coordinate the request and responses [8]. One of the fundamental differences in asynchronous communication as compared to the synchronous mode is that in asynchronous communication the client no longer makes a direct call to the server and expect an immediate answer. Instead, other services subscribe to the same broker to pick-up the available requests and process them further before placing them back to the message queue.

Figure 2 provides an example of the asynchronous pattern. In this sample, when a new order is created, the *customer service* publishes a request to the broker with some metadata such as *customer id*, *customer email address*, etc. Other services such as *loyalty*, *post*, and *email service* subscribe to that broker and take the request from there without having to communicate with Customer service directly.

[1] https://grpc.io

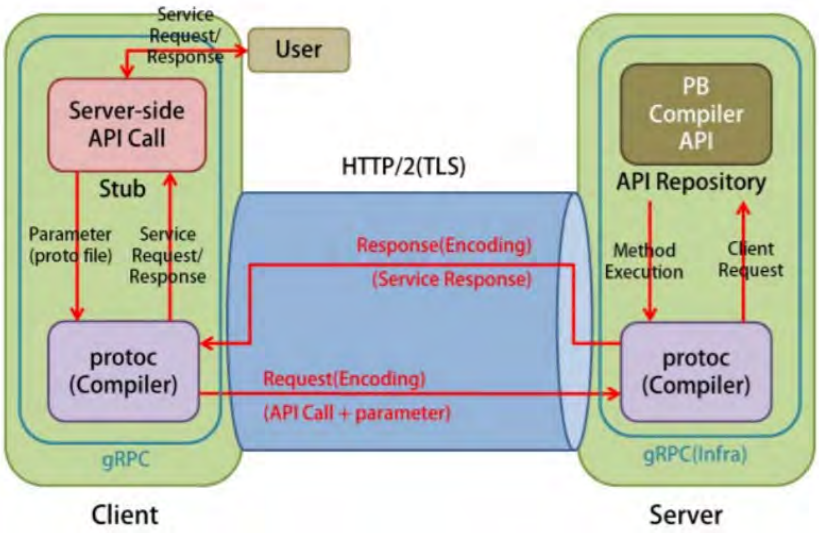


Figure 1: gRPC architecture [21].

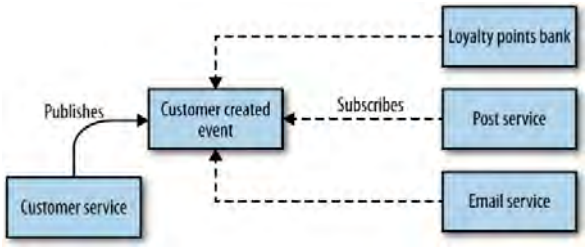


Figure 2: High-level architecture of Asynchronous pattern [9].

3. Implementation

To identify the quality attributes of each IPC method, we have designed and developed a set of microservices for an e-commerce scenario. In this scenario, the goal is to simulate fetching all the information required to display a product page of an e-commerce website. A client requests a product page to be displayed on his/her device and behind the scenes the following microservices work together to serve that request:

- **Product Information Service**: This microservice is responsible for fetching the primary metadata associated with the requested product. Information such as product name, price, description, color, and image are stored in this microservice database.
- **Product Review Service**: This microservice is responsible for fetching the customer reviews associated with the requested product from its database.
- **Product Recommendation Service**: This microservice is responsible for fetching the product recommendations based on the requested productId from its database.
- **Product Shipping Service**: This microservice is responsible for fetching available shipment options and the delivery estimates based on the given product from its database.
- **Customer Shopping Cart Service**: This microservices is responsible for fetching the existing items in the customer's shopping cart in order to display them to the customer.

All the microservices have been developed using NodeJS[2]. A non-relational database system, MongoDB[3], has been used as the database solution for all the microservices except for the service responsible for providing shipment information. Due to the nature of data required by shipping service, the shipping service uses MYSQL[4]. Docker[5] has been utilized to containerize all the microservices. In order to run the test system, the services have been deployed to Microsoft Azure Kubernetes Cluster Service[6]. Table 1 shows the

[2]https://nodejs.org/en/
[3]https://www.mongodb.com/
[4]https://www.mysql.com/
[5]https://www.docker.com/
[6]https://azure.microsoft.com/sv-se/services/kubernetes-service/

Table 1
Kubernetes Cluster specification of the test system.

Instance Type	Azure DS2-v2
vCPU	2
Memory	7 GiB
Storage	8 GiB, SSD, 6400 IOPS
Kubernetes Version	1.14.8
Node Count	3

hardware specification of the testing system used for this research.

In the synchronous mode both REST API, and gRPC have an identical architecture; in both methods, there is a direct communication between API Gateway[7] and each microservice. Each microservice acts as server, and the API Gateway acts as a client of those server. The key difference between REST API and gRPC is the underlying communication protocol as well as the format of the messages they exchange. gRPC has adopted protocol buffer[8] as its proprietary message format, while the REST API uses JSON [22] format to exchange data.

The asynchronous architecture uses RabbitMQ as message broker. In this pattern, the communication between API gateway and other services does not take place directly, rather it goes through a mediator also known as message queue. In both synchronous and asynchronous methods, the API Gateway is the entry point to the system, which receives a request with specific *product id* from client's device such mobile app or web browser over HTTPS protocol. The gateway then communicates back and forth with each microservice depending on the IPC method the system uses.

4. Results and Evaluation

4.1. Performance Efficiency

Three test cases have been designed and executed using Apache JMeter[9]. All test cases aim to measure the throughput of each IPC method. *Throughput* is an essential attribute for calculating performance efficiency. In all three test experiments, the test duration was 180 seconds, while the number of concurrent virtual users that continuously send requests to the system and wait for response has been varied. The motive behind having test duration as a constant variable and number of virtual users as the controlled variable is to understand how each IPC method reacts differently when the concurrent requests and traffic to the system increase or decrease.

Throughput is calculated by the total number of requests and responses the method managed to make within the specified duration of 180 seconds; the higher the number, the higher the throughput and the better it is.

The results are presented in figure 3. The data indicates that gRPC has outperformed REST API, and RabbitMQ in the first case with 50 users by being able to process 43 requests higher than REST API, and 147 requests more than RabbitMQ; this signifies that synchronous form of communication can offer higher throughput than the asynchronous method in the situation when the load to the system is relatively low. Meanwhile, the result of the first case also reveals that synchronous form of communication can process requests slightly faster than asynchronous form and, therefore, has lower latency when the number of concurrent threads[10] in the system is low.

The second case has double the number of virtual users as compared to the first one. Increasing the number of virtual users causes the number of concurrent threads in the system to grow and results in longer processing time. The same data imply that gRPC has the highest throughput by processing a higher number of requests compared to RabbitMQ and REST API; however, the gap between gRPC and RabbitMQ is now more narrowed than in the first case. In this test, gRPC managed to score the best average response time than REST API and RabbitMQ by 200 milliseconds. The processing time between REST API and RabbitMQ are equal to each other; however, RabbitMQ managed to process extra 25 requests than its synchronous rival.

The number of virtual users in the third case has increased four times as compared to the first case. The outcome of the third testing experiment implies considerable difference between synchronous versus asynchronous form of communication both in throughput and latency when the number of parallel requests increases. In this test, asynchronous form of communication using RabbitMQ has outperformed the other two methods by being able to process a total of 4480 requests within the given period while gRPC managed to process 132 requests lower than RabbitMQ, and REST API processed 146 less requests than its asynchronous rival. What makes the asynchronous pattern to operate better in the third test case is that, in asynchronous form the performance decline take place more gradually while in the synchronous

[7] https://microservices.io/patterns/apigateway.html
[8] https://developers.google.com/protocol-buffers
[9] https://jmeter.apache.org/

[10] Each virtual user occupies one thread in the system.

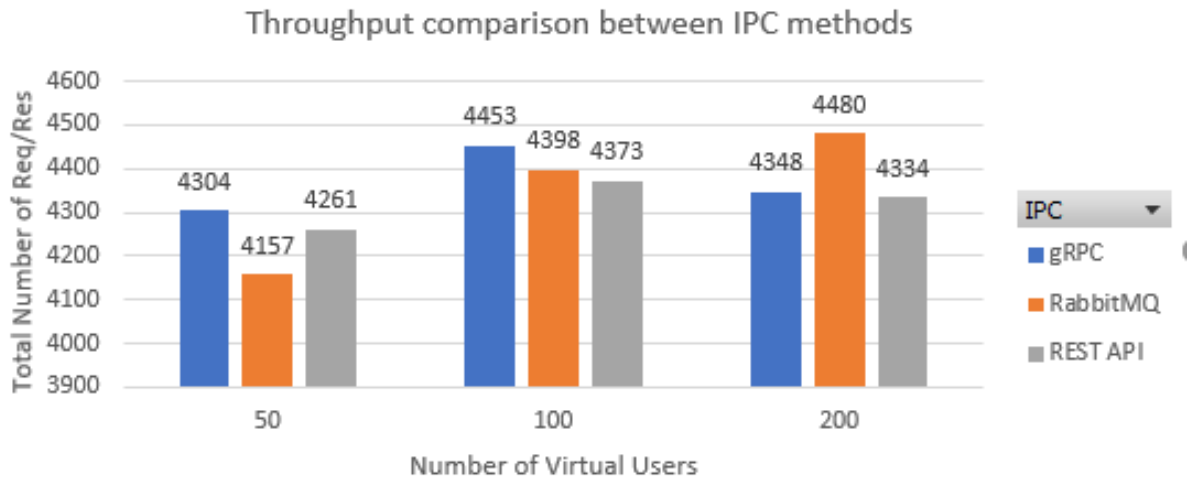


Figure 3: Throughput comparison.

pattern the performance begins to drop radically as soon as the load to the system intensifies.

4.2. Availability

There are variety of parameters that can affect availability of a system –even hardware components can play a role in determining the availability rate of a system. For this measurement, all the parameters outside IPC has been ignored. The availability of each IPC method has been calculated by using the following equation [23]:

$$Availability = \frac{MTTF}{MTTF + MTTR},$$

with MTTF standing for "Mean Time to Failure," and MTTR for "Mean Time to Recovery." MTTF represents the duration that the system is expected to last in operation before failure occurs. In contrast, MTTR represents the duration the system requires to return to operation after a failure has occurred. The higher the MTTR, the longer it takes for the system to recover from a failure, which consequently reduces the availability of the system.

Based on this formula, three other tests were executed using Apache Jmeter against all the three different IPC methods to discover which one offers higher availability. Unlike the previous test cases that had a fixed duration, these test cases had no specific duration. They ran as long as the services became unavailable due to the high number of requests coming to the system. Further, in this test, the average response time, and the number of requests/responses were not been tracked since they do not contribute to determining the availability of the method. The first case ran with 200 virtual users, the second with 300 virtual users, and third with 400 virtual users. Without having a high number of parallel users measuring availability becomes more challenging as the system remains operational for a significantly longer duration.

Figure 4 provides a summary of the conducted tests. During the first test, it took about seven minutes for the services to become unavailable using RabbitMQ, while gRPC went down after about five minutes, and the REST API took approximately four and a half minutes. These numbers were then dropped in each method in the subsequent tests as the number of parallel requests were doubled. After the services became unavailable, the Kubernetes cluster has been manually restarted. From that moment, both gRPC and REST API took about 20 seconds only to become available again, while RabbitMQ took ten extra seconds. The main reason behind RabbitMQ taking longer than synchronous form to return back to operation is the fact that it has an extra component known as a message broker that requires to be refreshed and establish a new connection with each service. From this experiment, it is possible to infer that an asynchronous approach offers higher availability than its synchronous opponents.

Consequently, if microservices use a synchronous based communication both client and server must be responsive at all time, otherwise the request will fail after a specific duration depending on the configura-

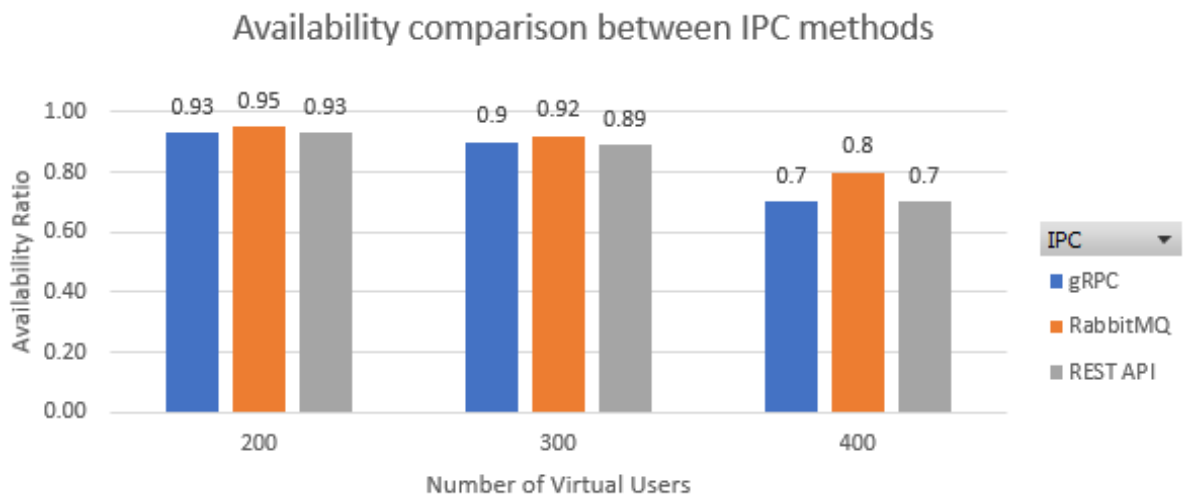


Figure 4: Availability comparison.

tion. In contrast, a temporary outage of the server in an asynchronous setting causes minimal to no impact to the consumer, since the consumer is loosely coupled with the server. The requests can stay in the message queue and be processed at the later timing when the server is back to operation. The asynchronous pattern offers capabilities that can help the system to improve its availability and resiliency from outage. It allows continuous operation even if there is a failure in one of the system's components without compromising the availability of the entire system.

4.3. Discussion and Threats to Validity

In addition to the two non-functional requirements that have been evaluated throughout this work, it is important to take into account the functional requirements for which microservices are being developed for. It is essential to distinguish whether the scenario requires an immediate response back from services or not. To elaborate further on this, in the proof of concept scenario that was built during this work, displaying a product page for an e-commerce was simulated. In this scenario, the client sends a request to load the product page and expects an immediate result back. The result of the request can either be the product page or an error that indicating the request was failed. The key point in this scenario is that the client expects an immediate result. In such scenarios the synchronous form of communication can be more suitable as these scenarios cannot take advantage of the features that an asynchronous form can offer.

Furthermore, our research incorporates some threats to validity. First, we performed our experiments just with single technologies as representatives for certain principles (synchronous vs. asynchronous). Therefore, our results can just indicate certain advantages of these principles. Second, we simulated no complete system but just a small part of a bigger system, e.g., there is no communication between the microservices during our requests. However, this ensures that we are not testing other effects, but only the interaction between the gateway and the microservices. Third, we were using technologies that are highly configurable, thus a completly different configuration could lead to other results. However, as we just changed configurations where necessary, we assume that others can reproduce our results, especially as they are in accordance with our theoretical expectations.

5. Related Work

Sufficient work has been done to benchmark the performance of microservices, and compare and contrast it with other architectures such as service oriented architecture (SOA) [24], or with the monolithic architecture [25, 26, 27, 28, 29].

Ueda et al. [30] conducted research at IBM that aimed to design an infrastructure that is optimized for running microservice architectures. The team built two versions of the sample application. One based on monolithic and the other based on microservices. The team discovered a significant performance overhead and higher hardware resource consumption in the mi-

croservices version of the application as compared to monolithic one. The paper has marked poor design of process communication in microservice architectures as one of the significant performance degradations, and, therefore, unleashed the potential for further research and improvements in this topic. The paper has also pointed out that network virtualization techniques, which are often used in a microservice architectures, is another non-negligible reason behind the performance gap of monolithic versus microservice architecture. The paper, however, has not prescribed any specific solution or suggestion as to how to overcome these challenges but rather pointed out the potential future work for it.

Fernandes et al. [31] compared REST API performance versus advanced message queuing protocol (AMQP) [32], which is one of the protocols used in message-based communication that falls under asynchronous category. The study has been done by measuring the averaged exchanged messages for a period of time using the REST API and AMQP. The authors performed the experiments by setting up two independent software instances that constantly received messages for a 30 minutes period with an average 226 request per second. Each instance processed the received input message and stored them into a persistent database. After executing the experiments, the authors concluded that for scenarios where there is a need to receive and process-intensive amount of data, AMQP performs far better than REST API as it has a better mechanism for data loss prevention, better message organization, and utilize lower hardware resources.

In contrast to Fernandes et al., check we in our work the behavior of the systems with different loads. We recognize that synchronous approaches perform good with low loads while asynchronous approaches scale better at higher loads.

Meanwhile, Dragoni et al. [28] have conducted a migration for a real-world mission-critical case study in the banking industry by transforming a monolithic software into a microservice architecture. They observed how availability and reliability of the system changed as a result of the new architecture. The solution consists of decomposing several large components to which some of them requires to communicate with third-party services. The services in the new architecture use message-based asynchronous communication as its IPC model to exchange data with each other. The authors believe that aiming to have a simple and decouple integration between services and following principle to handler failure will eventually lead to higher reliability in microservice architecture.

Further, the authors argue that microservice architectures lead to a higher availability as the new system is broken down into several components and decoupled from each other, which makes it possible to load-balance individual services as needed. This was particularly not possible in the legacy monolithic based system. At the same time, the new architecture offers higher reliability and can better cope with failures. This is due to the fact that in the new system the communication relies on a message-broker that can be configured to ensure all messages get delivered eventually.

6. Conclusion

When developing a microservices based system, the choice of IPC method is an important decision to make. In this paper, we compared synchronous and asynchronous IPC methods with regards to performance efficiency and availability. The outcome of our evaluation indicates that on average asynchronous approach provides better performance efficiency and higher availability. We also discussed a scenario where synchronous methods are more suitable to be utilized. Therefore, both synchronous and asynchronous type of communication has to be adopted according to the functional and non-functional requirements of the specific components.

References

[1] J. Thönes, Microservices, IEEE software 32 (2015) 116–116.

[2] D. Namiot, M. Sneps-Sneppe, On micro-services architecture, International Journal of Open Information Technologies 2 (2014) 24–27.

[3] L. L. Peterson, N. C. Buchholz, R. D. Schlichting, Preserving and using context information in interprocess communication, ACM Trans. Comput. Syst. 7 (1989) 217–246. doi:10.1145/65000.65001.

[4] S. Haselböck, R. Weinreich, G. Buchgeher, Decision guidance models for microservices: service discovery and fault tolerance, in: Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems, 2017, pp. 1–10.

[5] J. Fritzsch, J. Bogner, A. Zimmermann, S. Wagner, From monolith to microservices: A classification of refactoring approaches, in: J.-M. Bruel, M. Mazzara, B. Meyer (Eds.), Software Engineering Aspects of Continuous Development

and New Paradigms of Software Production and Deployment, Springer International Publishing, Cham, 2019, pp. 128–141.

[6] C. K. Rudrabhatla, Comparison of event choreography and orchestration techniques in microservice architecture, Int J Adv Comput Sci Appl 9 (2018) 18–22.

[7] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, W. Zhao, Poster: Benchmarking microservice systems for software engineering research, in: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), IEEE, 2018, pp. 323–324.

[8] C. Richardson, Microservices patterns: with examples in Java, Manning Publications, 2019.

[9] S. Newman, Building microservices : designing fine-grained systems, first edition.. ed., 2015.

[10] P. Johnson, D. Gorton, R. Lagerström, M. Ekstedt, Time between vulnerability disclosures: A measure of software product vulnerability, Computers & Security 62 (2016) 278–295.

[11] R. Lagerström, P. Johnson, M. Ekstedt, Architecture analysis of enterprise systems modifiability: a metamodel for software change cost estimation, Software quality journal 18 (2010) 437–468.

[12] P. Närman, P. Johnson, R. Lagerström, U. Franke, M. Ekstedt, Data collection prioritization for system quality analysis, Electronic Notes in Theoretical Computer Science 233 (2009) 29–42.

[13] M. Ekstedt, U. Franke, P. Johnson, R. Lagerström, T. Sommestad, J. Ullberg, M. Buschle, A tool for enterprise architecture analysis of maintainability, in: 2009 13th European Conference on Software Maintenance and Reengineering, IEEE, 2009, pp. 327–328.

[14] U. Franke, M. Ekstedt, R. Lagerström, J. Saat, R. Winter, Trends in enterprise architecture practice–a survey, in: International Workshop on Trends in Enterprise Architecture Research, Springer, 2010, pp. 16–29.

[15] P. Johnson, R. Lagerström, P. Närman, M. Simonsson, Extended influence diagrams for system quality analysis, Journal of Software 2 (2007) 30–42.

[16] R. T. Fielding, R. N. Taylor, Architectural styles and the design of network-based software architectures, volume 7, University of California, Irvine Irvine, 2000.

[17] M. Masse, REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces, " O'Reilly Media, Inc.", 2011.

[18] A. D. Birrell, B. J. Nelson, Implementing remote procedure calls, ACM Transactions on Computer Systems (TOCS) 2 (1984) 39–59.

[19] J.-K. Lee, A group management system analysis of grpc protocol for distributed network management systems, in: SMC'98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 98CH36218), volume 3, IEEE, 1998, pp. 2507–2512.

[20] R. A. Olsson, A. W. Keen, Remote Procedure Call, Springer US, Boston, MA, 2004, pp. 91–105. doi:10.1007/1-4020-8086-7_8.

[21] S. G. Du, J. W. Lee, K. Kim, Proposal of grpc as a new northbound api for application layer communication efficiency in sdn, in: Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication, 2018, pp. 1–6.

[22] C. Severance, Discovering javascript object notation, Computer 45 (2012) 6–8.

[23] P. Johnson, R. Lagerström, M. Ekstedt, M. Österlind, It management with enterprise architecture, KTH, Stockholm (2014).

[24] T. Erl, Service-oriented architecture: concepts, technology, and design, Pearson Education India, 1900.

[25] T. Cerny, M. J. Donahoo, J. Pechanec, Disambiguation and comparison of soa, microservices and self-contained systems, in: Proceedings of the International Conference on Research in Adaptive and Convergent Systems, RACS '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 228–235. doi:10.1145/3129676.3129682.

[26] D. Taibi, V. Lenarduzzi, C. Pahl, Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation, IEEE Cloud Computing 4 (2017) 22–32.

[27] R. Chen, S. Li, Z. Li, From monolith to microservices: A dataflow-driven approach, in: 2017 24th Asia-Pacific Software Engineering Conference (APSEC), 2017, pp. 466–475.

[28] N. Dragoni, S. Dustdar, S. T. Larsen, M. Mazzara, Microservices: Migration of a mission critical system, arXiv preprint arXiv:1704.04173 (2017).

[29] Z. Kozhirbayev, R. O. Sinnott, A performance comparison of container-based technologies for the cloud, Future Generation Computer Systems 68 (2017) 175 – 182. doi:10.1016/j.future.2016.08.025.

[30] T. Ueda, T. Nakaike, M. Ohara, Workload characterization for microservices, in: 2016 IEEE international symposium on workload characterization (IISWC), IEEE, 2016, pp. 1–10.

[31] J. L. Fernandes, I. C. Lopes, J. J. Rodrigues, S. Ullah, Performance evaluation of restful web services and amqp protocol, in: 2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN), IEEE, 2013, pp. 810–815.

[32] S. Vinoski, Advanced message queuing protocol, IEEE Internet Computing 10 (2006) 87–89.

Understanding the Involvement of Developers in Missing Link Community Smell: An Exploratory Study on Apache Projects

Toukir Ahammed, Moumita Asad and Kazi Sakib

Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh

Abstract

Missing link smell occurs when developers collaborate in source code without communication. This can affect software maintenance by the means of lacking mutual awareness, mistrust and knowledge gap. Existing studies have investigated the relationship of missing link smell with different socio-technical factors and code smell. This study aims to understand how many developers are involved with missing link smell, by calculating the percentage of smelly developers for a project. The study also investigates the relationship between the number of contributions and the number of missing link involvements of a developer. The result shows that the percentage of smelly developers involved with missing link smell is 8.7% on average. The result also suggests a moderate positive correlation between the contribution of a developer to the project and the involvement in smell.

Keywords

missing link smell, community smell, software engineering, empirical analysis

1. Introduction

Community smells are the organizational and social anti-patterns in a development community [1]. Community smells may lead to the emergence of social debt which indicates unforeseen project costs connected to a sub-optimal software development community. Community smells may not be an immediate obstacle for software development but these can affect software maintenance negatively in the long run [2]. Missing link is one of the common community smells. It refers to the condition when two co-committing developers show uncooperative behavior by not communicating [3].

Missing link community smell decreases communication activities in the development community. The lack of communication and cooperation negatively affects mutual awareness and trust among developers [3]. A software product can be thought of as the combined effort of all developers. So, the collaboration along with proper communication is necessary among developers. It is important to know how many developers are involved in missing link smell as they may affect the whole project. Identifying these developers and analyzing their characteristics is important. This will help the management to take steps such as task reassigning, team reformation, increasing awareness about communication etc. to keep communication issues lower in the community.

The detection of missing link smell and its impact on software artifacts have been analyzed in previous studies. S. Magnoni proposed the identification pattern of missing link community smell [3]. Tamburri et al. examined the relationship between community smells and different socio-technical factors, e.g., socio-technical congruence, turnover etc [4]. This study considered missing link, organizational silo, black cloud and radio silence community smell. Palomba et al. investigated the impact of missing link and four other community smells on code smell intensity [2]. Catolino et al. analyzed the role of four community smells including missing link smell on gender diversity and women participation in open-source community [5].

However, a little is known about developers and how developer contributions in the project relate to missing link smell. This study aims to focus on these by addressing the following Research Questions (RQs).

RQ1: How many developers are involved in missing link community smell?

In an open-source project, there can be many developers contributing to the project. All developers may not be involved in missing link smell. This RQ aims to find how many developers are involved in missing link smells in a community. This is important to know the collective contribution of developers to the number of missing link smells in a project. The project managers will get insight about the project health. This finding can be used in mitigating community smells by focusing on these developers and their communication issues.

RQ2: How does missing link smell relate with a developer contribution?

This RQ focuses on the involvement of individual developer in missing link smell. This RQ relates an impor-

QuASoQ 2020: 8th International Workshop on Quantitative Approaches to Software Quality

EMAIL: bsse0806@iit.du.ac.bd (T. Ahammed); bsse0731@iit.du.ac.bd (M. Asad); sakib@iit.du.ac.bd (K. Sakib)

CEUR Workshop Proceedings (CEUR-WS.org)

tant characteristic of a developer, i.e., contribution, to missing link smell. This finding will help project managers understanding which type of developers involve more in missing link smell. This information can be used to decide which developers can be monitored to control missing link smell in the community from the beginning of a project.

In this study, missing link smells are analyzed on seven open-source projects of *Apache* ecosystem. First, the instances of missing link smell are detected in each project. Then the developers associated with each smell are identified by extracting the instance of smell. The fraction of developers involved with missing link smell is calculated to check whether a subset of developers are involved with this type of smell. Then the correlation is investigated between the contribution of developers and their involvement in missing link smells.

The results of the study show that a small part of total developers involved with missing link community smell. On average, 8.7% of the total developers of a project are involved with missing link smell. This study also finds a significant moderate positive correlation between the developer contribution and their involvement in missing link smell ($\tau_b = 0.612$, $p < 0.01$).

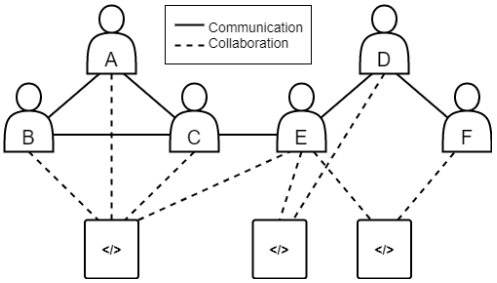


Figure 1: Developer Social Network

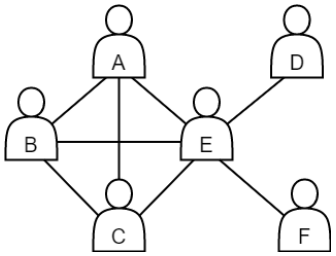


Figure 2: Collaboration Network

2. Background

This section provides some important terminologies to better understand the missing link community smell.

Developer Social Network (DSN): A network of a software development community where a node represents developer and relationships, e.g., communication, coordination, between developers are represented by an edge.

Collaboration Network: A specific type of DSN which indicates the collaboration in a development community. Here, a node represents a developer who contributes to the project in the version control system. Two developers are connected through an edge if they contribute to the same part of source code within a given time frame [3]. Figure 2 represents an example of collaboration network.

Communication Network: A specific type of DSN which indicates the communication within the defined communication channel of a development community. Here, a node represents developers who communicate in the defined communication channel, i.e., mailing list. Two developers are connected through an edge if they replied in the same e-mail within a given time frame [3]. A communication network is illustrated in Figure 3.

Missing Link Community Smell: A missing link community smell is occurred when a couple of developers collaborate with each other but show uncooperative behaviors by not communicating. This smell can be identified detecting collaboration between two developers that do not have the communication counterpart in defined communication channel e.g., development mailing list [3].

An example of DSN is illustrated in Figure 1. The upper part of the graph represents communication and the lower part represents the collaboration among developers. The developers are connected with solid line if they communicate with each other. The developers are connected to the file icon through a dashed line if they contribute to that source code file.

The collaboration and communication network can be generated separately from this DSN. Figure 2 and Figure 3 represent the collaboration and the communication network respectively. The missing link smell can be identified comparing the collaboration network with communication network. There is a link between developer *E* and *F* in the collaboration network (Figure 2) but there is no corresponding link between these two developers in the communication network (Figure 3). Developer *E* and *F* are collaborating on the same part of source code but they are not connected through any communication link. Thus, this is considered as an instance of missing link between developer *E* and *F*.

3. Related Work

In recent years, community smells are studied to incorporate organizational and social aspect of developer community in software engineering research. Some studies

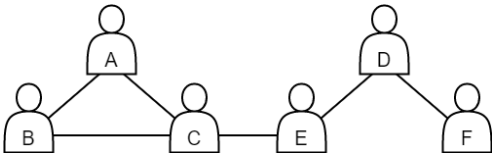


Figure 3: Communication Network

focused on defining different community smells that can lead to unforeseen project costs. On the other hand, some studies investigated the impact of community smells on different software artifacts.

Tamburri et al. first introduced the concept of social debt in software engineering [6]. Later, in an industrial case study, they improved and elaborated the definition of social debt. In the same study, they defined nine different community smells which are connected to social debt [1]. They also suggested a list of possible mitigations of community smells such as learning community, cultural conveyors, stand-up voting etc., to avoid the negative effects.

Magnoni proposed the identification pattern of four out of nine community smells [3] defined in [1]. He developed an open-source tool *CODEFACE4SMELLS*[1] as an extension to *CODEFACE* [7]. This tool is capable of detecting community smells from the change history in the version control system and the communication history in development mailing list.

Tamburri et al. analysed the distribution of community smells in open-source projects [4]. They also assessed the relation between community smells and existing socio-technical quality factors, e.g., socio-technical congruence, communicability, turnover etc.

Palomba et. al examined the relationship between social and technical debt [2] [8]. They assessed the impact of community smells on code smells. They found community smells significantly influencing code smell intensity. They also proposed a community-aware code smell intensity model in which both technical and community related factors were considered.

Catolino et al. analysed the role of gender diversity and women participation on community smell [5]. They considered four types of community smell i.e., organizational Silo, Lone Wolf, Black Cloud and Radio Silence. They found that gender diverse team had lower number of community smells than non-gender diverse team. They also showed that gender diversity and women participation were important factors for Black Cloud and Radio Silence whereas organizational Silo and Lone wolf were found partially related.

The existing studies have focused on community smells and the impact of these smells on software artifacts. The phenomenon of community smells is surrounded with developers in a development community. But a little is known about the developers involved with community smells. The relation between missing link smell and developer characteristics is not yet investigated. So, the developers involved with community smells and their characteristics, e.g., contribution, need to be explored.

4. Methodology

This study aims to understand how many developers of a project are involved in missing link smell. This study also wants to assess the relationship between a developer's contribution and involvement in missing link smell. First, the missing link smell is detected for all the selected projects. Then the percentage of smelly developers is retrieved for each project. Later, the correlation analysis is performed between a developer's contribution and involvement in missing link smell.

4.1. Dataset

In this work, 9 large open-source projects belonging to *APACHE* ecosystem are selected for analysis. Table 1 provides the list of analysed projects with their name, source code link, development mailing list and analysis period. All projects are hosted in online version control system *GitHub* and the development mailing list archives are available on *Gmane*[2].

The selected projects are large enough in terms of community members and the number of commits. The projects have 668 community members on average. All the projects have a substantial number of commits, with an average of 10359. Thus the study has enough collaboration and communication data for analysis.

4.2. Missing Link Smell Detection

The selected projects are analysed using a six-month analysis window. The analysis period of a project starts from when both communication in mailing list and change history in repository are available. A few more months are excluded to make the analysis period divisible by six months. The analysis period for each project is given in Table 1.

For every analysis window of a project, a communication network and a collaboration network is built. The communication network is generated extracting communication data from development mailing list and the collaboration network is generated extracting collaboration data from the project repository. After having both communication and collaboration networks, the instances of missing link smell are identified by comparing

[1] https://github.com/maelstromdat/CodeFace4Smells

[2] http://gmane.io

Table 1
List of Analysed Projects

#	Project Name	Source Code	Mailing List	Analysis Period
1	Apache Cassandra	github.com/apache/cassandra	gmane.comp.db.cassandra.devel	Oct-2009 - Sep-2020
2	Apache Cayenne	github.com/apache/cayenne	gmane.comp.java.cayenne.devel	Nov-2007 - Aug-2020
3	Apache CXF	github.com/apache/cxf	gmane.comp.apache.cxf.devel	Nov-2010 - Sep-2020
4	Apache Jackrabbit	github.com/apache/jackrabbit	gmane.comp.apache.jackrabbit.devel	Dec-2005 - Sep-2020
5	Apache Jena	github.com/apache/jena	gmane.comp.apache.jena.devel	Oct-2012 - Sep-2020
6	Apache Mahout	github.com/apache/mahout	gmane.comp.apache.mahout.devel	Oct-2008 - Aug-2020
7	Apache Pig	github.com/apache/pig	gmane.comp.java.hadoop.pig.devel	Oct-2010 - Aug-2020

every collaboration link with communication networks. If any collaboration link does not have its communication counterpart, this link is identified as a missing link instance.

An open-source tool, *CODEFACE4SMELLS*, is used to detect missing link community smell in this study. This tool is capable of detecting missing link smell in the aforementioned way from project repository and development mailing list. The tool requires the link of source code repository and mailing list archive as input. Then the tool returns a list of missing link instances for each window of the project. A missing link instance is represented by a pair of developers. For example, (a, b) represents a missing link instance between developer a and b.

4.3. Smelly Developers Identification

A developer involved with a missing link smell is considered as a smelly developer. An instance of missing link smell consists of two collaborating developers who do not communicate with each other. Thus for every missing link smell, there are two smelly developers. *CODEFACE4SMELLS* outputs a missing link instance as a pair of developers. So, the smelly developers can be obtained by extracting all missing link instances of a project. The smelly developers of a project x can be denoted by a set SD_x. The number of smelly developers of the project will be the number of elements in SD_x.

To calculate the percentage of smelly developers in a project, the total number of developers of that project is required. The total number of developers is defined as the sum of the number of developers who contribute to source code and the number of members who communicate on mailing list [3]. The total number of developers of a project is obtained by counting the number of members present in either collaboration or communication network generated by *CODEFACE4SMELLS*. The percentage of smelly developers of a project is calculated using the following formula (Equation 1),

$$percSD_x = \frac{numSD_x}{totalDev_x} \times 100\%, \tag{1}$$

where $numSD_x$ is the number of smelly developers in project x and $totalDev_x$ is the number of total developers in project x.

4.4. Correlation Analysis

RQ2 aims to understand the relationship between a developer's contribution and involvement in missing link smell. To address this RQ, the correlation between following two measures is analysed:

1. how many commits a developer has in the project repository
2. how many times a developer is involved in missing link smell

In open-source projects, commits are the most representative form of coding contribution [9]. So, the contribution of a developer in a project is measured by the number of commits done in that project repository. The number of commits done by every individual developer is retrieved from the source code repository.

The number of involvement in missing link smells can be obtained from the list of missing link instances of a project. First, the developers are extracted from all missing link instances of the project. Then the number of involvement is calculated counting how many times a developer occurs in the list.

Both the number of commits and the number of involvement in smells of a developer are converted into percentage to achieve the relative measurement. The following two formulas, Equation 2 and Equation 3 are used to calculate number of commits and involvement is missing link smell in percentage respectively:

$$percentCommit = \frac{numCommit_i}{\sum_{i=1}^{n} numCommit_i} \times 100\% \tag{2}$$

where $numCommit_i$ is the number of commits of developer i and n is the total number of smelly developers.

$$percentMissingLink = \frac{numMissingLink_i}{\sum_{i=1}^{n} numMissingLink_i} \times 100\% \tag{3}$$

Table 2
Correlation coefficient interpretation

Correlation Coefficient (Negative)	Correlation Coefficient (Positive)	Interpretation
-0.4 $< \tau_b \leq$ 0.0	0.0 $\leq \tau_b <$ 0.4	Weak
-0.7 $< \tau_b \leq$ -0.4	0.4 $\leq \tau_b <$ 0.7	Moderate
-0.9 $< \tau_b \leq$ -0.7	0.7 $\leq \tau_b <$ 0.9	Strong
-1.0 $\leq \tau_b \leq$ -0.9	0.9 $\leq \tau_b \leq$ 1.0	Very Strong

where $numMissingLink_i$ is the number of commits of developer i and n is the total number of smelly developers.

Finally, the correlation analysis is performed between *percentCommit* and *percentMissingLink* for each project individually. Kendall's tau-b is used to assess the degree of association between these two variables. Both *percentCommit* and *percentMissingLink* have tied values in the dataset. As Kendall's tau-b can handle tied ranks, this is used for the correlation analysis. The correlation coefficient is considered significant if the p-value is less than 0.01. The correlation coefficient is interpreted according to Table 2.

5. Result Analysis

This section presents the result analysis of this study. All missing link smells found from selected projects are analysed to answer the research questions. Analysis and discussion for both research questions are provided as follows.

5.1. RQ1: How many developers involved with missing link smell?

To answer this RQ, all missing link smells of a project are considered. For every project, the number of total developers and the number of smelly developers are calculated. Then the percentage of smelly developers is obtained for each project.

Table 3 demonstrates the percentage of smelly developers for each project. It is observed that, on average 10.5% of total developers of a software community are involved in missing link smells. *Apache Cayenne* community has the highest percentage of smelly developers (21.1%). This is also the smallest community among 7 communities. Tamburri et. al. found that the number of community smell grows quadratically with the number of community members until the threshold of 200 community members [4]. The occurrences of community smell tend to stabilize after this threshold. As the number of total developers in *Apache Cayenne* community is less than 200, the number of missing link smell has not stabilized yet. So, this project has relatively more missing link smell and consequently more smelly developers. Excluding *Apache Cayenne* project, the rest six projects have 8.7% smelly developers on average.

These results suggest that only a small portion of developers in an open-source software community are involved with missing link smells. They do not communicate appropriately with their co-committing or collaborative developers. Thus, they contribute to the total number of community smells in a software community.

5.2. RQ2: How does community smell relate with a developer contribution?

To answer this RQ, the correlation between a developer's contribution and involvement in missing link smell is analyzed. Kendall's tau-b is used as a correlation technique since it can handle tied values.

First, the correlation analysis is performed individually for each development community. The Kendall's tau-b coefficients and p-values are provided in Table 4. All seven projects show a moderate positive correlation between number of commits and number of smells which is statistically significant with $p<0.01$.

Another correlation analysis is performed combining all the projects. The value of the correlation coefficient is slightly increased but still falls under the range of moderate positive correlation. This result is also statistically significant with a p-value less than 0.01.

These results suggest that a developer who contributes more in a project tends to have more missing link smells. This can happen because a developer, who contributes more, have to communicate more with other developers. The overload of communication may be the reason for involving in more missing link smells than others. From another point of view, a developer having more contribution to a project is likely to be more familiar and experienced with that project. As he knows most of the aspects of that project, he may take the communication with co-committers lightly while contributing. However further analysis is required to find out the causes of involving in more smells.

Table 3
Percentage of Smelly Developers

#	Project Name	Total Developers	Smelly Developers	Smelly Developers(%)	Average
1	Apache Cassandra	1380	205	14.9%	**8.7%**
2	Apache CXF	972	94	9.7%	
3	Apache Jena	244	34	13.9%	
4	Apache Mahout	615	28	4.6%	
5	Apache Pig	668	22	6.0%	
6	Apache Jackrabbit	927	28	3.0%	
7	Apache Cayenne	175	37	21.1%	
	Average	**668**	**64**	**10.5%**	

Table 4
Correlation Analysis

#	Project Name	Tau-b	p-value
1	Apache Cassandra	0.508	< 0.01
2	Apache Cayenne	0.543	< 0.01
3	Apache CXF	0.528	< 0.01
4	Apache Jackrabbit	0.589	< 0.01
5	Apache Jena	0.452	< 0.01
6	Apache Mahout	0.409	< 0.01
7	Apache Pig	0.513	< 0.01
	Overall	**0.612**	**< 0.01**

6. Threats to Validity

This section discusses the potential threats that may affect the validity of this study.

Threats to external validity: Threats to external validity concern the generalization of obtained results. In this study, seven projects from *Apache* are analysed. Thus the generalisation requires more projects belonging to different systems. However, to mitigate this threat large and diverse projects are selected that have a long change history - 11 years on average.

Threats to internal validity: Threats to internal validity concern the factors that can influence our results but are not accounted for. In this study, *CODEFACE4SMELLS* tool is used for the detection of missing link smell. The outputs of *CODEFACE4SMELLS* are directly incorporated in this study without checking whether there is any defect in the tool. However, the capability of this tool of identifying missing link smell was evaluated [3]. This tool is also used in other studies in detecting community smells [2] [5] [10].

Moreover, this tool relies on mailing list to detect communication among developers. The contribution guidelines of the selected projects in this study suggest that mailing list represents the main communication channel for these projects. But there may exist other communication channels, e.g., Skype, Facebook etc., where developers communicate with each other. However, retrieving communication data from these channels may not be practical as this data is not publicly available. So, it can be said that mailing lists provide sufficiently accurate information on communication among developers in this study.

7. Conclusion

This study explores the percentage of developers in a software development community involved in missing link smells. Furthermore, the relationship between developer contribution and involvement in missing link smell is examined. At first, missing link smells are detected for all the projects. Next, the smelly developers are identified by extracting missing link instances. The percentage of smelly developers are calculated for every project. The number of appearances of a developer in missing link smell is counted. The contribution of a developer to a project is measured by the number of commits. Finally, correlation analysis is done between contribution and involvement in smell.

This study analyses seven open-source projects of *Apache*. The result shows that the number of developers involved in missing link smells is 8.7% on average. This study also founds that there is a positive correlation between the number of commits of a developer and the number of involvement in missing link smells. The developers who contribute more tend to involve in more missing link smell.

In future, other types of community smell, e.g., organizational silo, radio silence, can be examined to find their association with developers contribution.

Acknowledgments

The virtual machine facility used in this research is provided by Bangladesh Research and Education Network (BdREN).

References

[1] D. A. Tamburri, P. Kruchten, P. Lago, H. Van Vliet, Social debt in software engineering: insights from industry, Journal of Internet Services and Applications 6 (2015) 10.

[2] F. Palomba, D. A. A. Tamburri, F. A. Fontana, R. Oliveto, A. Zaidman, A. Serebrenik, Beyond technical aspects: How do community smells influence the intensity of code smells?, IEEE transactions on software engineering (2018).

[3] S. Magnoni, An approach to measure community smells in software development communities (2016).

[4] D. A. Tamburri, F. Palomba, R. Kazman, Exploring community smells in open-source: An automated approach, IEEE Transactions on software Engineering (2019).

[5] G. Catolino, F. Palomba, D. A. Tamburri, A. Serebrenik, F. Ferrucci, Gender diversity and women in software teams: How do they affect community smells?, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS), IEEE, 2019, pp. 11–20.

[6] D. A. Tamburri, P. Kruchten, P. Lago, H. van Vliet, What is social debt in software engineering?, in: 2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), IEEE, 2013, pp. 93–96.

[7] M. Joblin, W. Mauerer, S. Apel, J. Siegmund, D. Riehle, From developer networks to verified communities: a fine-grained approach, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, volume 1, IEEE, 2015, pp. 563–573.

[8] F. Palomba, D. A. Tamburri, A. Serebrenik, A. Zaidman, F. A. Fontana, R. Oliveto, Poster: How do community smells influence code smells?, in: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), IEEE, 2018, pp. 240–241.

[9] S. Daniel, R. Agarwal, K. J. Stewart, The effects of diversity in global, distributed collectives: A study of open source project success, Information Systems Research 24 (2013) 312–333.

[10] F. GIAROLA, Detecting code and community smells in open-source: an automated approach (2018).

Detection and Correction of Android-specific Code Smells and Energy Bugs: An Android Lint Extension

Iffat Fatima[a], Hina Anwar[b], Dietmar Pfahl[b] and Usman Qamar[a]

[a] *College of Electrical and Mechanical Engineering, National University of Sciences and Technology,Islamabad, Pakistan*
[b] *Institute of Computer Science, University of Tartu, Tartu, Estonia*

Abstract

Context: While Android applications suffer from code smells and energy drain issues there is still a lack of tools that help developers improve energy consumption and maintainability of Android applications. Objective: Our research aims to provide tool support to Android developers helping them to create greener and more maintainable applications by eliminating Android-specific code smells/energy bugs. The proposed tool support integrates routine code smell detection with energy bug detection so that developers can do both at the same time. Method: We extend 'Android Lint' (AL) with custom rules to detect and correct 12 code smells (nine are new and three are improved) and three energy bugs (two are new and one is improved). In addition, for the improved and newly introduced code smells, we compared the performance of our tool with the open version of the 'PAPRIKA' tool. Result: We evaluated our tool on nine open-source Android applications. Our tool detects the specified code smells and energy bugs with an average precision, average recall and F1 score of 0.93, 0.96, and 0.94, respectively. It accurately corrects 84% of selected code smells and energy bugs. The performance of the new and improved code smell detection is better than that achieved by 'PAPRIKA'. Conclusion: Our tool is a useful extension to the existing 'AL' tool with better performance than 'PAPRIKA'.

Keywords

Green Software Development, Android, Energy Optimization, Code Smell, Energy Bug, Android Lint, Detection, Refactoring, Static Analysis

1. Introduction

Recently, the focus of research has shifted towards sustainable and green software development with a focus on energy optimized programming and energy optimization at the application level [1]. Software is now being built not only keeping performance, dependability, and maintainability in mind but also the principles of green software engineering aiming at the development of sustainable software with less negative impact on the environment. With the fast-paced emergence of mobile technologies in the past decade, mobile applications are being widely used. 3.5 billion people use smartphones around the world [2], and Android has 75% of the market share.

Code smells and energy bugs have been identified as causes of abnormal energy consumption in Android applications. Significant research has been carried out on the impact of object-oriented smells in Java applications [3, 4, 5, 6]. In our previous work [1] we identified support tools that aid green Android development. We further identified the coverage of code smells and energy bugs by those tools and identified their limitations. We concluded that there is a lack of guidelines for Android developers to write sustainable software. Current state of the art tools lack in providing complete coverage for Android code smells and energy bugs. Moreover, they lack in usability, IDE integration and effective refactoring approach. The aim of this research is to create a tool that solves these issues by aiding developer to solve energy related problems during development of the application for improved performance and maintainability.

'Android Lint' (AL) is the default static analysis tool in Android Studio IDE, hence used by most Android developers. Code smells detected and prioritized by 'AL' tend to disappear faster from code base as compared to other code smells detection tools. Moreover, a lint tool integrated in Android Studio IDE not only encourages the developers to correct code smells on the go but also plays a role in developer education [7]. We chose a custom implementation of 'AL' API to 1) maximize coverage of Android code smells/energy bugs, 2) provide recommendations to developers for refactoring, 3) provide a preview of the detected and corrected code, and

QuASoQ 2020: 8th International Workshop on Quantitative Approaches to Software Quality, December 1st, 2020, Singapore
iffat.fatima@ce.ceme.edu.pk (I. Fatima); hina.anwar@ut.ee (H. Anwar); dietmar.pfahl@ut.ee (D. Pfahl); usmanq@ceme.nust.edu.pk (U. Qamar)
0000-0002-4725-4636 (H. Anwar); 0000-0003-2400-501X (D. Pfahl)

4) provide an interface consistent with the Android Studio IDE.

The 'extended Android Lint' (xAL) tool is evaluated on open-source Android applications to detect and correct code smells and energy bugs. The current evaluation has resulted in average precision, average recall and F1 score of 0.93, 0.96, and 0.94, respectively. Whereas, 84% of the suggested corrections applied to the applications under test, resulted in the smooth functioning of applications. However, results cannot be generalized based on these statistics due to the small scale of the evaluation setup. Our tool also provides better code smell/energy bug coverage, and usability compared to 'PAPRIKA' tool.

Section 2 provides related work. Section 3 contains the tool design and implementation details. Section 4 presents the evaluation plan and results. Section 5 discusses threats to validity. Section 6 concludes the study.

2. Related Work

There are several publications in which Android application analysis tools have been presented that detect or correct Android-specific code smells and energy bugs. The aim of this study is to provide a solution at the development stage hence we look into only those tools that perform static analysis, with the aim to optimize applications in terms of their energy consumption.

The 'HOT-PEPPER' toolkit [8] (based on 'PAPRIKA' tool [9]) detected and refactored a set of Android-specific code smells and produced a corrected version of the APK without developer intervention.

'Statedroid' tool [10] performed a taint-like analysis using specified resource protocols to detect energy leaks caused by Wakelock Bugs and Resource Leaks.

In [11], the authors used a combination of 'Eclipse Refactoring API', 'PMD', and 'AL' to build a tool that optimizes Android applications for CPU usage. Their rule set covered only a limited number of Android code smells. However, the tool offered developers the flexibility to add their own rules.

The 'aDOCTOR' tool [12] detected 15 code smells causing energy drains by traversing the abstract syntax tree. The code smells were removed manually by authors and correlated to energy consumption. Energy estimation was done using 'PETrA'. The 'aDOCTOR' tool has a precision and recall of 98%.

Jiang et al.[13] used 'SAAF' for resource leak analysis and 'AL' for layout defect analysis in Android applications. They detected energy bugs like Camera Leak, Memory leak, Multimedia Leak, sensor Leak, and layout defects.

Olivier Le Goaër [14] presented an automated tool based on 'AL' which detected 11 energy greedy Android patterns such as Draw Allocation, Wakelock, Recycle, Obsolete Layout Parameter, HashMap Usage, Member Ignoring Method, Excessive Method Calls and some Resource Leaks. The tool 'AutoRefactor' was used for refactoring and the impact of those refactoring on energy consumption of open-source Android applications was measured.

The 'E-Debitum' [15] tool (based on 'SonarQube') detected six energy code smells and calculated their energy debt.

Comprehensive coverage of Android-specific code smells and energy bugs within one single tool is missing in existing tools. In the tools mentioned above, most commonly detected code smells were Member Ignoring Method, Internal Getter and Setter methods whereas most commonly detected energy bugs were Wakelock Bug and Resource Leaks. Existing tools have low usability due to lack of Android Studio IDE integration. Tools in studies [9, 8, 13] provided a command line interface while in the study [11] integration with Eclipse IDE was provided instead of Android Studio (which is the official IDE for Android development [7]). Tools compatible with Android Studio [14, 16], were not open source. Tools in studies [9, 13, 12, 11, 16] did not refactor applications while tools in studies [8, 15] provided completely automated refactoring hence reducing the control of the developer during refactoring process.

3. Design

In this section, we describe how we selected the baseline tool for extension and how we enhanced it.

3.1. Baseline Tool Selection

The aim of this study is to create a tool that covers the limitations of previously developed tools in terms of providing a comprehensive coverage of the Android code smells and energy bugs and solving the usability issues such as IDE integration, flexibility and ease of use for developer, open source availability etc. Based on this objective, we set the following criteria for selecting a tool for an extension:

- The tool should be open source or provide the ability to extend or customize it to add

rules for detection and refactoring of code smells/energy bugs.

- The tool should be able to perform static analysis.
- The tool should be integrate-able with Android studio IDE as it is the official IDE for Android development [7].
- The tool should provide an inline warning on code smell/energy bug detection inside Android Studio code editor.
- The tool should provide a mechanism to list detected code smells/energy bugs along with their description.

The above criteria were applied on industry-standard tools such as 'AL[1]' (AL), 'PMD', 'SpotBugs[2]' (SB), 'SonarLint[3]' (SL), 'SonarQube[4]' (SQ), and 'SOOT[5]' (ST) (see Table 1). A similar tool, 'FindBugs', was not considered as its successor is 'SpotBugs'. 'Eclipse refactoring engine' was also excluded as it is not integratable with Android Studio. Tools that perform static analysis but focus on code styling such as 'CheckStyle' were excluded. We also considered detection and optimization tools identified in our previous work [1]. Even though many of these tools were built on top of open-source static analysis tools such as 'SOOT', 'SPARK', 'SAAF', 'ASM', 'PMD', and 'Lint', we did not select them for an extension because they were not designed to be integrated with Android Studio IDE. The tools using dynamic analysis were also excluded as they require the application to be built every time. Long average build time for Android applications is a known and significant issue among the development community[6]. Table 1 shows a comparison of tools in terms of selection criteria. After this comparison, 'SpotBugs', 'SonarQube', and 'SOOT' were excluded as they built the application every time a code smell/energy bug needs to be detected.

Next, we compared the three shortlisted tools: 'AL', 'PMD', and 'SL' for Android-specific code smell and energy bug coverage (See Table 2 and 3). 'AL', 'PMD', and 'SonarLint' can cover many different types of issues in code (code smell, bug or error is referred to as 'issue' in these tools). For example, 'AL' can detect 261 different types of Android-specific issues[7]. Majority of these issues are related to syntax and styling of the code. We could not find any evidence in the literature about the energy impact of the issues already covered by the above shortlisted tools, therefore, we only compared them for the coverage of 25 Android-specific code smells and nine energy bugs listed in [1].

Table 1
Comparison of tools in terms of selection criteria

Criteria	AL	PMD	SB	SL	SQ	ST
Open Source	✓	✓	✓	✓	✓	✓
Customization API	✓	✓	X	✓	✓	X
Source Code Analysis	✓	✓	X	✓	X	X
Byte Code Analysis	✓	X	✓	X	✓	✓
Android Studio Integration	✓	✓	✓	✓	✓	X
Inline issue warning/hint	✓	X	X	✓	✓	X
List of detected code smells/energy bugs	✓	✓	✓	✓	✓	✓
Allows adding refactoring rules	✓	X	X	X	X	X

AL = Android Lint, SB = SpotBugs,SL= SonarLint, SQ = SonarQube, ST= SOOT

In Tables 2 and 3, '*' represents that the code smell/energy bug is detected but based on the definition of code smell/energy bug[8] the detection coverage has room for improvement. ✓represents that code smell/energy bug is detected, × represents that code smell/energy bug is not covered by the tool yet. From Tables 2 and 3, we can see that 'AL' already covers 13 code smells and six energy bugs. Therefore, an effort towards improvement in the small number of undetected code smells/energy bugs will result in a single tool with maximum coverage. In addition, 'AL' provides offline documentation for rules and allows the developer to choose whether to correct a specific code smell/energy bug or not. Based on the above data, 'AL' is a feasible tool for an extension.

3.2. Android Lint Extension

In this section, we explain the 'AL' API and implementation details of our new 'extended Android Lint' (xAL) tool.

API Overview. 'AL' provides an embedding API that allows adding custom rules. In order to create custom rules, the 'AL' embedding API pro-

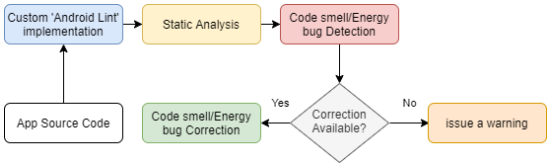


Figure 1: Overview of 'AL' API

[1] http://tools.android.com/tips/lint-custom-rules
[2] https://spotbugs.github.io/
[3] https://www.sonarlint.org/features/
[4] https://docs.sonarqube.org/latest/extend
[5] https://github.com/Sable/soot/
[6] https://developer.android.com/studio/build/optimize-your-build
[7] http://tools.android.com/lint/overview
[8] https://figshare.com/s/84ae49a21551e6302d41

Table 2
Code smell coverage by tools 'AL', 'PMD' and 'SL'

	DTWC	DR	LIC	IDFP	ISQLQ	IDS	IGS	LT	MIM	NLMR	PD	RAM	SL	UC	LC	LWS	UHA	BFU	UIO	IWR	HAT	HSS	HBR	IOD	ERB
AL	X	✓	✓	✓	✓	✓	✓	*	✓	X	X	✓	✓	✓	X	✓	X	*	✓	X	X	X	X	✓	*
PMD	X	X	X	X	✓	X	✓	*	X	X	X	X	✓	✓	X	X	X	X	X	X	X	X	X	X	*
SL	X	✓	X	X	✓	X	✓	*	✓	X	✓	X	✓	✓	X	✓	X	X	X	X	X	X	✓	X	✓

DTWC=Data Transmission Without Compression, DR=Debuggable Release, LIC=Leaking Inner Class, IDFP=Inefficient Data Format and Parser, ISQLQ=Inefficient SQL Query, IDS=Inefficient Data Structure, IGS=Internal Getter and Setter, , LT=Leaking Thread, MIM=Member Ignoring Method, NLMR=No Low Memory Resolver, PD=Public Data, RAM=Rigid Alarm Manager, SL=Slow Loop, UC=Unclosed Closeable, LC=Lifetime Containment, LWS= Long Wait State, UHA=Unsupported Hardware Acceleration, BFU= Bitmap Format Usage, UIO=UI Overdraw, IWR=Invalidate Without Rect, HAT=Heavy AsyncTask, HSS=Heavy Service Start, HBR=Heavy Broadcast Receiver, IOD=Init ONDraw, ERB=Early Resource Binding

Table 3
Energy bug coverage by tools 'AL', 'PMD' and 'SL'

	RL	WB	VBS	IB	TMV	TDL	NCD	UL	UP
AL	*	✓	X	X	✓	✓	✓	✓	✓
PMD	*	X	X	X	X	X	X	X	X
SL	*	X	X	X	X	X	X	X	X

RL=Resource Leak, WB=Wake-lock Bug, VBS=Vacuous Background Services, IB= Immortality Bug, TMV=Too Many Views, TDL= Too Deep Layout, NCD=Not Using Compound Drawables, UL= Useless Leaf, UP=Useless Parent

vides many class APIs. In the 'AL' API, each code smell/energy bug has the following properties: Id, summary, explanation, category, severity, priority, and additional links[9]. This information is shown to the developer when a code smell/energy bug is detected. Each code smell/energy bug is registered in an issue registry class and is detected by a detector class. The functionalities of detector class used by our implementation are given in additional material[10]. Fig. 1 gives an overview of the 'AL' API (version 26.5.2 is used for the implementation of detectors). The complexity calculation for the code smells HAA, HSS and HBR are done using 'Metrics Reloaded' plugin for Android Studio[11].

Inclusion of New Code Smells/Energy Bugs. For all undetected and partially covered code smells/ energy bugs (see Table 2 and 3), detection and refactoring rules are defined based on the definitions provided in additional materials and Android development best practice guides[12] provided by Google. Table 4 shows a list of Android code smells and energy bugs that are implemented in our 'extended Android Lint' (xAL) tool. In 'Implementation' column 'novel' refer to code smells/energy bugs that are not already present in the 'AL' tool. 'Improvement' refers to code smells /energy bugs that are partially covered by the 'AL' tool and can be improved by inclusion of additional APIs/conditions in our new 'xAL' tool. The last column of Table 4 shows whether a correction is suggested by 'xAL' for the detected Android code smells and energy

Table 4
Android code smells and energy bugs implemented in 'xAL'

Abbr.	Implementation	Detection	Correction offered
Code smells			
DTWC	novel	yes	No, just warning is shown
LT	improvement	yes	yes
NLMR	novel	yes	yes
PD	novel	yes	yes
LC	novel	yes	yes
UHA	novel	yes	yes
BFU	improvement	yes	No, just warning is shown
IWR	novel	yes	No
HAT	novel	yes	No, just warning is shown
HSS	novel	yes	No, just warning is shown
HBR	novel	yes	No, just warning is shown
ERB	improvement	yes	No, just warning is shown
Energy Bugs			
RL	improvement	yes	yes
VBS	novel	yes	yes
IB	novel	yes	yes

DTWC=Data Transmission Without Compression, LT=Leaking Thread, NLMR=No Low Memory Resolver, PD=Public Data, RAM=Rigid Alarm Manager, LC=Lifetime Containment, UHA=Unsupported Hardware Acceleration, BFU= Bitmap Format Usage, IWR=Invalidate Without Rect, HAT=Heavy AsyncTask, HSS=Heavy Service Start, HBR=Heavy Broadcast Receiver, ERB=Early Resource Binding, RL=Resource Leak, VBS=Vacuous Background Services, IB= Immortality Bug

bugs. Table 5 shows Android code smells/energy bugs that are partially covered by the original 'AL' tool and the improvements we implemented in our new 'xAL' tool for each of them. Pseudo-code for implemented code smells and energy bugs are given in additional material[13]. For most of the detected code smells/energy bugs we offer corrections. In cases where we do not provide corrections to the developer, a warning is issued. Each implemented code smell/energy bug is tested on sample classes which contains possible variations in which a selected code smell/energy bug can be present in the code. In addition, we provide a description for each of the detected code smell/energy bug, with the aim to help developers in refactoring. We made our tool open-source [16]. The 'xAL' tool is compiled

[9] http://tools.android.com/tips/lint-custom-rules
[10] https://figshare.com/s/84ae49a21551e6302d41.
[11] https://github.com/BasLeijdekkers/MetricsReloaded
[12] https://developer.android.com/topic/performance
[13] https://figshare.com/s/84ae49a21551e6302d41
[16] https://figshare.com/s/63c5b3e957f390432edf

Table 5
Proposed improvements in the Android Code Smells and Energy bugs detection

CS/ EB	API/Class already detected by 'AL' tool	API/Class detected by 'xAL' tool as improvement
LT (CS)	Detects thread class leak only.	Detection of classes like android.os.Handler and java.lang.Runnable that can lead to a thread leak[14].
BFU (CS)	Detects bitmap duplication, creation in onDraw() and usability issues.	Detection of Bitmap using Bitmap.create(params. . .)[15].
ERB (CS)	Detects creation of objects during DrawAllocation which can be performed by lazy initialization	Detection of heavy APIs (of type: Android .location, android.media, android.database, android.hardware) that should be initialized in lazy fashion1.
RL (EB)	Detects resources such as IO, JDBC, static fields, wifi manager, StringBuffer etc.	Detection of resources like Camera, Media Player etc. [?]

CS =Code Smell, EB =Energy Bug, LTLeaking Thread, BFU= Bitmap Format Usage, ERB=Early Resource Binding, RL=Resource Leak

as a Jar file. The Jar file is placed in the .android/lint folder of the Android Studio installation, typically located in USER-HOME. Android Studio is restarted for new detectors to take effect. Applications can be analyzed for Android code smells and energy bugs in two ways[17], i.e., in-line analysis and whole-application analysis.

4. Evaluation

4.1. Evaluation Plan

We evaluated the 'xAL' tool in two steps. First, we evaluated the tool on a selection of open source apps, then we compared the tool's performance to that of the 'PAPRIKA' tool.

Evaluation on Open Source Apps. The evaluation includes testing on nine real-world applications[18] chosen from the F-Droid[19] repository. The applications were chosen if the source code was in Java and the number of line of code was less than 30,000 (for ease of manual verification). We checked that each application can be compiled and executed on a device without errors.

Comparison with 'PAPRIKA'. We considered state of the art tools such as 'PAPRIKA', 'aDOCTOR', 'SAAD', 'Chimer' and 'AL' (implementation by Goaër [14]) as possible comparison candidates. 'Chimer' and the 'AL' tool presented in [14] were not available in open source. We chose 'PAPRIKA' as it had the largest number of overlapping code smells/energy bugs with our 'xAL' tool. These are Heavy Async Task (HAT), Heavy Broadcast Receiver (HBR), Heavy Start Service (HSS), Invalidate without Rect (IWR), No Low Memory Resolver (NLMR) and Unused Hardware Acceleration (UHA) and Resource Leak (RL). JAR file for Paprika was downloaded from their open-source repository[20].

4.2. Evaluation Results

This section presents the results of testing the tool on open source projects and compares its coverage with PAPRIKA tool.

4.2.1. Evaluation on Open Source Apps

The tool was tested on the nine applications selected from the F-Droid repository. Table 6 shows the results of the evaluation on open source applications. It lists the application name against the code smells detected in it, true positives (TP), false positives (FP), false negatives (FN), precision (P), recall (R), total corrections available (TCA) and total applied correction (TAC) .

Some of the code smells/energy bugs (such as 'Early Resource Binding' (ERB), 'Heavy Async Task' (HAT) and 'Heavy Broadcast Receiver' (HBR)) did not appear in any of the applications under test.

In the case of application 'Kolabnotes' two false negatives were detected, i.e. two instances of code smell 'Lifetime Containment' (LC) code smell. A possible reason could be declarations of interfaces in non-lifecycle classes, which were not included in the LC code smell definition. In the case of application 'Sound Recorder' two false positives were detected, i.e. two instances of code smells 'No Low Memory Resolver' (NLMR). A possible reason could be that the class used by this application is deprecated, hence not covered by the implementation of our 'xAL' tool. In the case of applications 'Kolabnotes' and 'Camera Roll' one false positive (i.e., one instance of the 'Data Transmission without Compression' (DTWC) code smell) was detected for each application. In the case of application 'Calorie Scope' one false positive was detected, i.e., one instance of 'Resource Leak (RL) for a camera instance energy bug. In the case of application, 'Privacy-Friendly Weather' five false positives (i.e. Lifetime Containment (LC) code smell (4 instances) and Resource Leak (RL) energy bug (1 instance)) were detected. In the case of application, 'Reminders' two false positives (i.e. two instances of Lifetime Containment (LC) code smell) were detected. In the case of code smell Lifetime Containment (LC), a possible reason for false positive could be that it flags abstract classes as interfaces as well. In

[17] https://figshare.com/s/84ae49a21551e6302d41 (See Tool Walk-through)
[18] https://figshare.com/s/84ae49a21551e6302d41 (See Table 1)
[19] https://f-droid.com
[20] https://github.com/GeoffreyHecht/paprika

Table 6
Results of evaluation on open source app using 'xAL' tool

ID	App	CS/EB Detected	Detection Results					TCA	TAC
			TP	FP	FN	P	R		
1	Odyssey	UHA, NLMR, HSS, LC, IWR	11	0	0	1	1	6	6
2	Kolabnotes	UHA, BFU, LT, DTWC, IWR, NLMR	18	1	2	1	0.9	7	7
3	Calorie Scope	LC, NLMR, RL, VBS	12	1	0	0.9	1	13	13
4	Camera Roll	UHA, BFU, DTWC, IWR, LC, NLMR, PD	21	1	0	1	1	16	6
5	Bipol Alarm	UHA, HSS, DTWC, NLMR	4	0	0	1	1	4	4
6	Sound Recorder	UHA, PD	7	0	2	1	0.8	7	7
7	CameraColorPicker	UHA, BFU, NLMR, IWR, LC, RL	12	5	0	0.7	1	17	12
8	Privacy-Friendly Weather	UHA, LT, LC, NLMR	13	2	0	0.9	1	15	13
9	Reminders	UHA, DTWC, NLMR	5	0	0	1	1	5	5
		TOTAL	103	10	4	–	–	90	73

CS = Code Smell, EB = Energy Bug, TP = True Positive, FP = False positive, FN = False Negative, P= Precision, R= Recall, TCA = Total Corrections Available, TAC= Total Applied Corrections

the case of code smell Data Transmission without Compression (DTWC), a possible reason could be that the 'xAL' tool does not track instances that were compressed in another class. In the case of energy bug Resource Leak (RL), a possible reason for false positives could be that code smell/energy bug was handled pro-actively by the developer. For example, for Resource Leak (RL), if the camera instance was closed proactively by the developer in a method other than onStop(). In this case, check on onStop() method is no longer required.

Corrections were available for 66.3% of the detected Android code smells and energy bug instances, out of which 84% of corrections were applied. 16% corrections were not applied, which include false positives and instances of Public Data (PD) code smell (correction of this code smell altered the functionality of the application under test). These numbers are dependent on the type of code smell/energy bug and the frequency of its instances in the application.

4.2.2. Comparison with PAPRIKA

The 'PAPRIKA' tool did not work on applications 1 to 4 that had AndroidX[21] dependencies. Due to this inherent limitation of 'PAPRIKA', it was only tested on applications 5 to 9. Refactoring of code smells/energy bugs was not applied in any test application as the accessible version of 'PAPRIKA' tool does not offer to refactor.

Table 7 shows the results of the evaluation on open source applications using 'PAPRIKA'. It lists the application name against the code smells detected in it, true positives (TP), false positives (FP), false negatives (FN), precision and recall. 'PAPRIKA' was able to detect three types of code smells namely: No Low Memory Resolver (NLMR), Heavy Start Service (HSS), Heavy Broadcast Receiver (HBR). For these smells, no false positives were detected. 'PAPRIKA' did not detect any instance of the code smells/energy bugs Unused Hardware Acceleration (UHA), Resource Leak (RL) and Invalidate without Rect (IWR), which could be seen in 'FN' column.

Table 8 shows a comparison of the code smells detected by both 'xAL' and 'PAPRIKA'. ✓represents that a code smell is detected in a test application, × represents that a code smell was not detected in a test application and empty cells show that the code smell was not present in a test application. Code smell Heavy Async Task (HAT) was not present in any of the test applications hence not detected by 'PAPRIKA' and our 'xAL' tool. 'xAL' was able to detect almost all the code smell instances that were detected by 'PAPRIKA' tool. However, two instances of 'No Low Memory Resolver' (NLMR) code smells were missed in application 'Sound Recorder' (as they used a deprecated class API). The 'Heavy Broadcast Receiver' (HBR) code smell was also missed by our tool as the value for an upper limit of computational complexity is set to five in 'PAPRIKA' and ten (as per McConnel [17]) in our 'xAL' tool.

'PAPRIKA' detected seven instances of false negatives for the code smells: Unused Hardware Acceleration (UHA) (5 instances) and Invalidate without Rect (IWR) (1 instance) and energy bug Resource Leak (RL) (1 instance). As compared to 'PAPRIKA', our tool was able to detect Invalidate without Rect (IWR) code smell and Resource Leak (RL) energy bug in 'Sound Recorder' test application. Unused Hardware Acceleration (UHA) code smell was also detected in all five applications by our tool. Table 9 shows a compatibility comparison between 'PAPRIKA' and 'xAL' to gain a better understanding of the support offered by both tools as well as their limitations.

The average precision, recall and F1-score for our 'xAL' were 0.93, 0.96 and 0.94, respectively. The average precision, recall and F1-score for 'PAPRIKA' were 1.0, 0.74 and 0.85, respectively. Hence, our 'xAL' tool not only provides better Android code smell/energy bug coverage but also improves upon the usability aspects of the tool in comparison to 'PAPRIKA' tool.

5. Threats to Validity

Our 'xAL' tool only performs static source code analysis of Android applications. Since static source code analysis could be done during development re-

[21] https://developer.android.com/jetpack/androidx

Table 7
Results of evaluation on open source apps using 'PAPRIKA'

ID	Test App	CS/EB detected	TP	FP	FN	Precision	Recall
5	Bipol Alarm	NLMR, HSS	2	0	1	1	0.67
6	Sound Recorder	NLMR	2	0	3	1	0.67
7	CameraColor Picker	NLMR	6	0	1	1	0.67
8	PrivacyFriendly Weather	NLMR	11	0	1	1	0.92
9	Reminders	NLMR, HBR	4	0	1	1	0.8
	TOTAL		25	0	7	–	–

CS = Code Smell, EB = Energy Bug, TP = True Positive, FP = False positive, FN = False Negative, NLMR=No Low Memory Resolver, SS=Heavy Service Start, HBR=Heavy Broadcast Receiver

Table 8
Code smell detection comparison between 'PAPRIKA' and 'xAL'

App	Bipol Alarm		Sound Recorder		CameraColor Picker		PrivacyFriendly Weather		Reminders	
CS	P	xAL	P	xAL	P	xAL	P	xAL	P	xAL
HAT										
HBR							✓	×		
HSS	✓	✓								
IWR					×	✓				
NLMR	✓	✓	✓	×	✓	✓	✓	✓	✓	✓
UHA	×	✓	×	✓	×	✓	×	✓	×	✓
RL					×	✓				

CS= Code smell P='PAPRIKA', xAL= 'extended Android Lint', HAT=Heavy AsyncTask, HSS=Heavy Service Start, HBR=Heavy Broadcast Receiver, IWR=Invalidate Without Rect, NLMR=No Low Memory Resolver, UHA=Unsupported Hardware Acceleration, RL=Resource Leak

Table 9
Compatibility comparison between 'PAPRIKA' and 'xAL'

Compatibility criteria	PAPRIKA	xAL
Java version support	Java 7 only	versions >= Java7
Support for apps with Android X	No	Yes
In-line warnings in Code Editor	No	Yes
Navigation to LOC in Code Editor	No	Yes
Individual code smell analysis	No	Yes
Disabling detection of CS/EB	No	Yes

LOC=Line of code, CS/EB=CodeSmell/EnergyBug, xAL='extended Android Lint'

peatedly, the support provided by our tool could benefit developers. Implementation of a functionality may vary based an application's architecture/design and the coding style of a developer. Hence, our definitions of code smells/energy bugs might not cover every scenario related to a particular code smell, leading to false positives and false negatives in the results. For example, we only consider lifecycle classes, i.e. Activity and Fragment for lifecycle dependent issues like Resource Leak and Leaking Thread. However, depending on the application architecture, lifecycle dependent components might be called in non-lifecycle classes which may go undetected. Moreover, some corrections for code smells, and energy bugs might clash with the functional requirements of the application. For example, correction for Public Directory (PD) code smell changes public directory to a private directory. However, for an application that requires access to a public directory like Gallery, if it is changed to a private directory, the functional requirement will be in contradiction. But as the corrections can only be applied after the developer's consent, such issues are less likely to occur. Correction/recommendations are offered for 66% of detected code smells/energy bugs. For the rest of 34% code smells/energy bugs, only a warning is shown with hints for correction (cf section 3.2.2). The decision of applying the suggested correction is left for the developers. Our tool does not cover third-party libraries used in Android applications. During development, we tested our tool against sample classes, which were injected with code smells/energy bugs by the authors of this study. Hence there might be researcher bias in the introduction of those code smells/energy bugs (in terms of coding style, location and variety), leading to higher accuracy in results. To mitigate this threat, the tool was evaluated on nine open-source applications that already contained some of the code smells and energy bugs. During the evaluation, we did not physically measure the changes in energy consumption of the applications under test due to refactoring of code smells/energy bugs. The assumption that refactoring the selected code smells/energy bugs lead to energy optimization of Android applications is based on the related work such as [11, 14, 16].

6. Conclusion

We extended the tool 'AL' to detect and correct Android-specific code smells and energy bugs that may lead to energy optimization in Android applications. On top of the 261 issues already covered by 'AL', our extended tool 'xAL' provides coverage for 12 Android-specific code smells (nine new and three improved) and three energy bugs (two new and one improved). Moreover, 'xAL' integrates directly in Android Studio IDE and gives control to the developer for refactoring code smell/energy bugs, which was missing in other state of the art tools. We evaluated 'xAL' on nine open-source applications; it detects code smells and energy bugs with an average precision, average recall and F1 score of 0.93, 0.96, and 0.94 respectively. It accurately corrects 84% of selected code smells and energy bugs. Our tool offers better code smell and energy bug detection coverage as compared to 'PAPRIKA'. In the future, we aim to evaluate 'xAL' on a large data set of applications, which will also help in analyzing the

correlation between the frequency of occurrences of code smells/energy bugs, and impact on energy consumption due to their refactoring.

Acknowledgments

This work is supported by the Estonian Center of Excellence in ICT research (EXCITE), and group grant PRG887 funded by the Estonian Research Council.

References

[1] I. Fatima, H. Anwar, D. Pfahl, U. Qamar, Tool Support for Green Android Development: A Systematic Mapping Study, in: 5th Int. Conf.on Softw. Technologies - ICSOFT, 2020, pp. 409–417.

[2] A. Turner, How many people have smarphones worldwide (Apr 2020), 2020. URL: https://www.bankmycell.com/blog/how-many-phones-are-in-the-world.

[3] R. Verdecchia, R. Aparicio Saez, G. Procaccianti, P. Lago, Empirical Evaluation of the Energy Impact of Refactoring Code Smells (2018) 345–365. doi:10.29007/dz83.

[4] H. Anwar, D. Pfahl, S. N. Srirama, Evaluating the Impact of Code Smell Refactoring on the Energy Consumption of Android Applications, in: 45th Euromicro Conf.on Softw. Eng. and Advanced Applications, SEAA, 2019, pp. 82–86. doi:10.1109/SEAA.2019.00021.

[5] A. V. Rodríguez, C. Mateos, A. Zunino, M. Longo, An analysis of the effects of bad smell-driven refactorings in mobile applications on battery usage, in: Modern Softw. Eng. Methodologies for Mobile and Cloud Environments, 2016. doi:10.4018/978-1-4666-9916-8.ch009.

[6] C. Sahin, L. Pollock, J. Clause, How do code refactorings affect energy usage?, Int'l Symposium on Empirical Softw. Eng. and Measurement - ESEM (2014) 1–10. doi:10.1145/2652524.2652538.

[7] S. Habchi, R. Rouvoy, N. Moha, On the survival of android code smells in the wild, Proceedings - 2019 IEEE/ACM 6th Int'l Conf. on Mobile Softw. Eng. and Systems, MOBILESoft 2019 (2019) 87–98. doi:10.1109/MOBILESoft.2019.00022.

[8] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, R. Rouvoy, Investigating the energy impact of Android smells, 24th IEEE Int'l Conf. on Softw. Analysis, Evolution, and ReEng. - SANER (2017) 115–126. doi:10.1109/SANER.2017.7884614.

[9] G. Hecht, R. Rouvoy, N. Moha, L. Duchien, Detecting Antipatterns in Android Apps, 2nd ACM Int'l Conf. on Mobile Softw. Eng. and Systems - MOBILESoft (2015) 148–149. doi:10.1109/MobileSoft.2015.38.

[10] Z. Xu, C. Wen, S. Qin, State-taint analysis for detecting resource bugs, Science of Computer Programming (2018) 93–109. doi:10.1016/j.scico.2017.06.010.

[11] V. N. Huynh, M. Inuiguchi, B. Le, B. N. Le, T. Denoeux, Improve the Performance of Mobile Applications Based on Code Optimization Techniques Using PMD and Android Lint, LNCS (including subseries LNAI and LNB) 9978 LNAI (2016) V–VI. doi:10.1007/978-3-319-49046-5.

[12] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia, Lightweight detection of Android-specific code smells: The aDoctor project, SANER 2017 - 24th IEEE Int'l Conf. on Softw. Analysis, Evolution, and ReEng. (2017) 487–491. doi:10.1109/SANER.2017.7884659.

[13] H. Jiang, H. Yang, S. Qin, Z. Su, J. Zhang, J. Yan, Detecting Energy Bugs in Android Apps Using Static Analysis, LNCS (including subseries LNAI and LNB) 10610 LNCS (2017) 192–208. doi:10.1007/978-3-319-68690-5_12.

[14] O. L. Goaër, Enforcing green code with android lint, in: 34th IEEE/ACM Int. Conf. on Automated Softw. Eng. Workshop - ASEW, 2019. doi:10.1109/ASEW.2019.00018.

[15] D. Maia, M. Couto, J. Saraiva, E-Debitum : Managing Softw. Energy Debt (2020) 162–169.

[16] M. Couto, J. Saraiva, J. P. Fernandes, Energy Refactorings for Android in the Large and in the Wild (2020) 217–228. doi:10.1109/saner48275.2020.9054858.

[17] S. McConnell, Code Complete: A Practical Handbook of Softw. Construction 9 (2011).

Comparison of Code Smells in iOS and Android Applications

Kristiina Rahkema[a], Dietmar Pfahl[a]

[a]Institute of Computer Science, University of Tartu, Tartu, Estonia

Abstract

Code smells are patterns indicating bad practices that may lead to maintainability problems. For mobile applications most of the research has been done on Android applications with very little research on iOS applications. Our goal is to compare the variety, density, and distribution of code smells in iOS and Android applications. We analysed 273 open source iOS and 694 open source Android applications. We used PAPRIKA and GraphifySwift to find 19 object oriented code smells. We discovered that the distributions and proportions of code smells in iOS and Android applications differ. More specifically, we found: a) with the exception of one code smell (DistortedHierarchy) all code smells that could be observed in Android apps also occurred in iOS apps; b) the overall density of code smells is higher on iOS than on Android with LazyClass and DataClass particularly sticking out; c) with regards to frequency, code smells are more evenly distributed on iOS than on Android, and the distributions of code smell occurrences on class level are more different between the platforms than on app level.

Keywords

Mobile applications, Android, iOS, Code smells

1. Introduction

Code smells are patterns indicating bad practices that often lead to maintainability problems [1]. Code smells have been studied extensively for desktop applications (shortened to "apps" in the following). For mobile apps most of the analysis has been done on the Android platform.

Mannan et al. [2] analyzed 21 object oriented code smells in open source Android apps. They compared code smell occurrences on Android and Java desktop apps looking at differences in variety, density and distribution of code smells. They discovered that the variety of code smells is the same, but density and distribution of code smells in desktop Java and Android apps differ. They mention that other mobile platforms should have the same variety of code smells but do not discuss possible differences in density or distribution.

Habchi et al. [3] used the tool PAPRIKA [4] to analyse iOS and Android apps and compared proportions of code smells on these platforms. They analysed iOS apps for four object oriented, three iOS specific and Android apps for four object oriented and two Android specific code smells. They discovered that code smell proportions were higher in Android apps.

Our goal is to compare the variety, density and distribution of code smells in iOS and Android apps. First we will check if variety, density and distribution of code smells differ in iOS and Android apps to see if the results are similar to differences found between Android and desktop Java apps by Mannan et al. [2]. Second we extend the analysis done by Habchi et al. [3] by comparing the densities and distributions of more code smells in iOS and Android apps, to see if Android apps are in general more prone to code smells and if different platforms are more prone to different code smells. In this study we aim to answer the following research questions:

RQ 1: Are all types of object-oriented code smells present in both iOS and Android apps?

To answer this and the following research questions, we used the tool GraphifySwift[1] [5] to analyse iOS Apps and the tool PAPRIKA [4] to analyse Android apps. To make the code smell definitions (and calculations) comparable across plat-

QuASoQ 2020: 8th International Workshop on Quantitative Approaches to Software Quality, December 1st, 2020, Singapore
kristiina.rahkema@ut.ee (K. Rahkema);
dietmar.pfahl@ut.ee (D. Pfahl)
0000-0003-2400-501X (D. Pfahl)

[1]https://github.com/kristiinara/GraphifySwift

forms, we adapted the code smell queries defined by Rahkema et al. [5] when searching for code smells in Android apps. In total, we identified 19 code smell types that could potentially occur in apps on both platforms. We took under consideration that the variety of code smells depends on the programming language used. For example, the code smell RefusedParentBequest is not applicable to Swift because Swift lacks the *protected* keyword. Therefore, we did not include it in our analyses.

Our analysis showed that 18 of the 19 identified code smells occurred in apps on both platforms, i.e., Android and iOS. Code smell DistortedHierarchy never occurred in iOS apps.

To better understand whether the frequency of occurrence is similar, we formulated our second research question.

RQ 2: Do code smells occur with the same density in iOS and Android apps?

To answer this question, we calculated the overall density of all code smells and the densities of each of the 19 code smells over all apps on both iOS and Android. It turned out that, contrary to what Habchi et al. [3] expected, the overall density of code smells is higher in iOS apps than in Android apps. Code smells LazyClass, DivergentChange, PrimitiveObsession and DataClass had a particularly high density in iOS apps. On the other hand, code smells LongMethod, LongParameterList and ShotgunSurgery were clearly more frequent in Android apps. In addition, we found that the code smell densities per code smell type were sometimes higher and sometimes smaller in iOS apps as compared to Android apps. This might be explained by the fact that Android apps tend to have more of the code smells that correspond to more complex classes whereas iOS apps tend to have more of the code smells that correspond to more simple classes.

To better understand the distributions of code smells in apps on the two platforms iOS and Android, we formulated our third research question.

RQ 3: Do code smell distributions differ between iOS and Android apps?

To answer this question we first compared the proportions of code smell occurrences across all iOS and Android apps. The results confirmed what we had seen when we compared code smell densities: the proportions of code smells differ between platforms. In addition, we saw that code smells are more evenly distributed in iOS apps as compared to Android apps.

Then we analyzed how large the share of smelly apps on each platform is and how large the share of smelly classes is on each platform. We did these analyses for each code smell type separately. It turned out that the percentages of smelly apps are relatively similar between platforms. Only the code smell DataClass is much more prominent in iOS apps than in Android apps.

In addition, we found that the distributions of code smell occurrences on class level are more different between the platforms than on app level. This result might, again, be explained by the fact that Android apps usually have larger classes and, thus, tend to have more of the code smells that correspond to more complex classes whereas iOS apps tend to have more compact classes and, thus, tend to have more of the code smells that correspond to more simple classes. This effect is more prominent when doing the analysis on class level than on app level.

2. Related Work

Code smells in desktop applications: Fowler [1] defined 22 object oriented code smells and provided refactorings for these code smells. Khomh et al. [6] studied the impact of code smells. They found that code smells affect classes negatively and that classes with more code smells were more prone to changes [6]. Olbrich et al. [7] studied the evolution and impact of code smells based on two open source systems. Their findings confirmed that code smells affect the way how code changes in a negative way. They were also able to identify different phases of evolution in code smells [7]. Linares et al. [8] made a large scale analysis of Java Mobile apps and discovered that anti-patterns negatively impact software quality metrics such as fault-proneness [8].

Tufano et al. [9] studied the change history of 200 open source projects and found that most code smells are introduced when the corresponding code is created and not when it is changed. They also found that when code does become

smelly through evolution then it can be characterized by specific code metrics. Contrary to common belief [10] they discovered that most code smells are not introduced by newcomers, but by developers with high work loads and high release pressure [9].

Code smells in Android applications: Different kinds of code smells have been researched for Android, such as object-oriented, Android-specific, security-related and energy-related code smells. Gottschalk et al. proposed an approach to detect energy related code smells on mobile apps and validated this approach on Android and showed that it is possible to reduce energy consumption by refactoring the code [11]. Ghafari et al. [12] studied security-related code smells and discovered that most apps contain at least some security-related code smells.

Hecht [4] proposed an approach to detect code smells and anti-patterns on Android systems and implemented this approach in a tool called PAPRIKA. This tool analyses the Android APK, creates a model of the code and inserts this model into the neo4j database. Code smells are then defined as database queries which makes it possible to query code smells on a large number of apps at the same time. He analysed 15 popular apps for the occurrences of four object oriented and three Android code smells. Hecht et al. [13] tracked, the software quality of 106 popular Android apps downloaded from the Google Play Store along their evolution. They calculated software quality scores for different versions of these apps and tracked their evolution. There were different evolution graphs, such as constant decline, constant rise, stability or sudden change in either direction depending on the programming practices of the team [13]. This shows that code quality is not necessary linked to app size, but the programming practices of the developers. Mateus et al. [14] used PAPRIKA to analyze Android apps written in Java and kotlin. They compared code smell occurrences in both languages and concluded that apps that were initially written in Java and later introduced kotlin were of better quality than other Android apps [14]. They analysed a set of 2167 open source Android apps combining different databases of open source Android apps.

In these papers using PAPRIKA the number of code smells studied was limited due to the number of code smells PAPRIKA is able to detect and ranged from three to four object oriented code smells and four to six Android specific code smells [15][13][14]. Mannan et al. [2] decided to broaden this scope and studied 21 object oriented code smells using the commercial tool InFusion. They analyzed open source Android and Java desktop apps for these 21 code smells and compared their occurrences. Mannan et al. detected that the variety of code smells was the same and most code smells occur in both systems in a similar frequency with major differences only for a couple of code smells. They concluded that studying code smells on mobile platforms can be done with tools meant for desktop apps. They also found that the code smells that have been researched so far are not the same ones that occur most and that the focus should change to code smells that are more relevant [2]. They suggest that other mobile platforms will have the same code smells, but do not give any suggestions towards the possible differences in density or distribution. They analysed 500 Android and 750 Java desktop apps randomly selected from GitHub. Unfortunately, the tool Infusion used by Mannan et al. does not seem to be available anymore. Therefore a direct comparison using Infusion for code smell analysis on iOS is no longer possible.

Code smells in iOS applications: Habchi et al. [3] used PAPRIKA to detect code smells in iOS apps. They used ANTLR4 grammars to generate parsers for Swift and Objective-C code. They created the apps graphs that could then be used by PAPRIKA. They analysed 176 Swift and 103 Objective-C apps from a collaborative list of open source iOS apps. In their study they analysed four object oriented, three iOS specific and two Android specific code smells. They compared smell proportions in iOS and Android apps and discovered that the proportions of code smells were higher in Android apps. On the other hand proportions of code smells in Objective-C and Swift were similar [3].

Rahkema et al. [5] introduced a tool called GraphifySwift that analyses Swift code and detects 34 object oriented code smells. Similarly to PAPRIKA, GraphifySwift enters data about the analysed app into the neo4j database. The database structures used by PAPRIKA and GraphifySwift

are similar, but slightly different. In their analysis they used the same collaborative list of open source iOS apps but did not compare the results to Android.

In the following, we extend the research in [2, 3, 5]. We adapted the queries defined in [5], where possible, so that they could be applied to a database populated by PAPRIKA. We used PAPRIKA to analyse Android apps and GraphifySwift to analyse Swift apps. Then we compared the two platforms with regards to variety, density, and distribution of 19 code smells.

3. Methods

In Section 3.1, we present the tools used for code smell analysis. In Section 3.2, we cover the choice of apps and in Section 3.3 we describe the analysis performed.

3.1. Code Smell Analysis

In previous research a tool called PAPRIKA has been used to find code smells in Android applications [4, 15, 13, 3, 14]. PAPRIKA analyses the Android APK, enters data about the applications into a neo4j database and defines queries for each code smell. For analysing iOS applications Habchi et al. [3] used PAPRIKA to query code smells, but populated the neo4j database using ANTLR grammars. Rahkema et al. [5] introduced a new tool called GraphifySwift that extends the functionality of PAPRIKA. It analyses Swift code, enters data about the iOS applications into a neo4j database and defines database queries to find code smells. PAPRIKA is able to find four object oriented code smells. Since the queries for these four code smells are implemented identically in GraphifySwift, it produces the same results as PAPRIKA for them. In GraphifySwift additional code smell queries are defined. Overall, GraphySwift is able to find 34 object oriented code smells.

For the analysis of iOS apps we used the tool GraphifySwift. We used the same thresholds as in Rahkema et al. [5]. Note that we focused on Swift code as Swift has replaced Objective-C and not many differences between the two languages are to be expected according to Habtchi et al. [3].

For Android apps we used PAPRIKA to populate the neo4j database. We then took the queries defined by Rahkema et al. for GraphifySwift to find code smells. Since GraphifySwift was originally developed to analyse iOS apps we had to adapt the code smell queries so that they could be used on the database produced by PAPRIKA. We made the following changes to the code smell queries:

We removed references to Module nodes, i.e., the relationship

```
(app)-APP_OWNS_MODULE->(module)-
        MODULE_OWNS_CLASS->(class)
```

was substituted by the relationship

```
(app)-APP_OWNS_CLASS->(class)
```

We removed references to argument type or substituted them with argument name. Argument names are not accessible in Java bytecode and therefore the argument name provided by PAPRIKA is actually the argument type.

Finally, we added the relationship

```
(variable|argument)-IS_OF_TYPE
                    ->(class)
```

by finding classes whose name matched the argument name or variable type.

After these modifications of the database and queries, 19 of the 34 GraphifySwift code smell queries could be used on the Android app database produced by PAPRIKA.

The code smell queries that had to be excluded contained metrics or attributes that were not provided by PAPRIKA. We excluded for example queries referring to code duplication, maximum nesting depth, number of switch statements and number of comments.

For the analysis of Android apps we calculated new thresholds based on the apps that we analysed. The list of iOS and Android thresholds is included in the thresholds table[2].

3.2. Choice of Applications

For analysis of iOS apps we used the same collaborative list of open source iOS apps as was used by

[2] https://figshare.com/articles/conference_contribution/Thresholds_for_iOS_and_Android_code_smell_analysis/13102991

Rahkema et al. [5] and whose older version was used by Habchi et al. [3]. The final set of successfully analysed apps was the same as in [5] and included 273 open source iOS apps.

For analysis of Android apps we took the list of apps provided by Habchi et al. [3]. Since the list only included app package names, we queried AllFreeAPK api[3] to find and download these apps. We decided to search AllFreeAPK instead of GitHub, as PAPRIKA uses APKs for analysis and this way we were able to skip the step of compiling these apps. Later during the analysis we needed to discard some of the very big apps due to performance issues. In total we included 694 open source Android apps in our analysis.

3.3. Data Analysis

To answer RQ1, we checked whether any of the 19 identified code smells occurred in at least one app on each platform.

To answer RQ2, we calculated the densities of code smells for both iOS and Android apps and compared these. Code smell density was calculated by counting the number of code smells (total and per code smell type) and dividing by the number app instructions.

To answer RQ3, we had to perform several calculations. To calculate the relative frequencies of code smells per code smell type on each platform, we counted the code smells of a type in all apps and divided by the total code smell count. We did this per platform. To calculate the code smell distributions on app and class levels per platform, we counted how many apps (and classes) contain at least one code smell of a certain type and then divided by the total number of apps (and classes).

4. Results

We analysed 273 open source iOS apps using GraphifySwift and 694 open source Android apps using PAPRIKA and modified code smell queries from GraphifySwift to answer our research questions. We analyzed the apps with regards to 19 code smells: BlobClass, ComplexClass, CyclicClassDependency, DataClass, DataClumpFields, DistortedHierarchy, DivergentChange, InappropriateIntimacy, LazyClass, LongMethod, LongParameterList, MiddleMan, ParallelInheritanceHierarchies, PrimitiveObsession, SAPBreaker, ShotgunSurgery, SpeculativeGeneralityProtocol, SwissArmyKnife and TraditionBreaker.

Below, we present and discuss the results for each research question separately.

RQ 1: Are all types of object-oriented code smells present in both iOS and Android apps?

When comparing the occurrence of code smells on each platform, we found that 18 of the 19 identified code smells occurred in apps on both platforms, i.e., Android and iOS. Code smell DistortedHierarchy never occurred in iOS apps.

Our result does not fully support Mannan et al.'s expectation that mobile apps on other platforms than Android should exhibit the same code smells [2].

RQ 2: Do code smells occur with the same density in iOS and Android apps?

The results of our code smell density analysis is shown in Figure 1. Accumulated over all code smells it turned out that the apps on the iOS platform had a density of 41.7 smells/kilo-instructions while the apps on Android only had a density of 34.4 smells/kilo-instructions. This result is contrary to what Habchi et al. [3] expected.

Moreover, it can be seen from Figure 1 that the code smell densities differ between iOS and Android. Code smells LazyClass, DivergentChange, PrimitiveObsession and DataClass had a particularly high density in iOS apps. On the other hand, code smells LongMethod, LongParameterList and ShotgunSurgery were clearly more frequent in Android apps. The fact that code smell densities were sometimes higher and sometimes lower in iOS apps as compared to Android apps might be explained by the fact that Android apps tend to have more of the code smells that correspond to more complex classes whereas iOS apps tend to have more of the code smells that correspond to more simple classes.

RQ 3: Do code smell distributions differ between iOS and Android apps?

Figure 2 shows the relative frequency of code smell occurrences over all apps on the Android platform (blue bars) and the iOS platform (red bars). The results confirm what we had seen when

[3] https://m.allfreeapk.com/api/

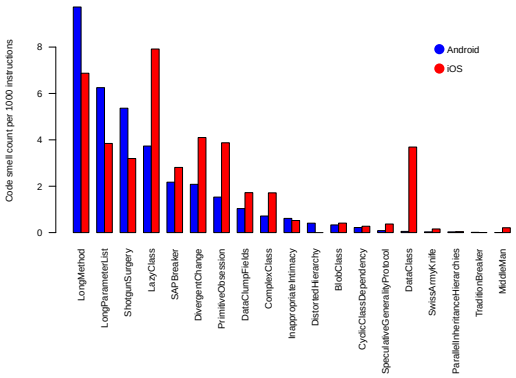


Figure 1: Comparison of code smell densities between Android (blue) and iOS (red) apps

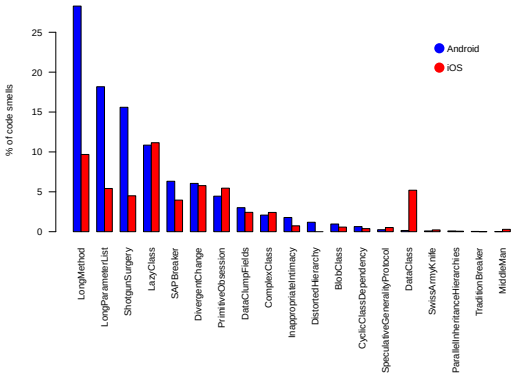


Figure 2: Code smell proportions on Android (blue) and iOS (red)

we compared code smell densities: the proportions of code smells differ between platforms. In addition, we see that code smells are more evenly distributed in iOS apps as compared to Android apps.

Then we analyzed how large the share of smelly apps on each platform is and how large the share of smelly classes is on each platform. We did these analyses for each code smell type separately.

Figures 3 and 4 show the percentages of apps and classes, respectively, containing code smells of a certain type.

We found that the percentages of smelly apps are relatively similar between platforms. The biggest differences occurs for code smell DataClass (79% of iOS apps have at least one affected class

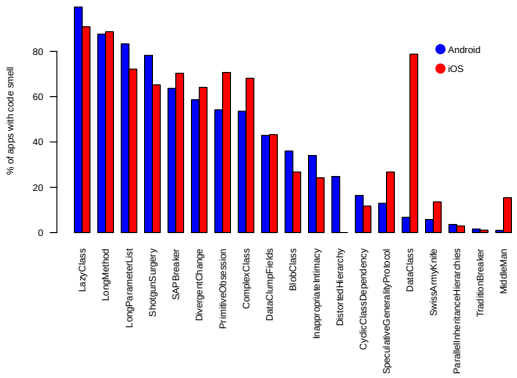


Figure 3: Comparison of code smell frequencies on app level between Android (blue) and iOS (red)

while only 7% of Android apps are affected), MiddleMan (15% of iOS apps are affected but only 1% of Android apps), and DistortedHierarchy (25 % of Android apps are affected but none of the iOS apps is).

We found that the distributions of code smell occurrences on class level are more different between the platforms than on app level. This result might, again, be explained by the fact that Android apps usually have larger classes and, thus, tend to have more of the code smells that correspond to more complex classes whereas iOS apps tend to have more compact classes and, thus, tend to have more of the code smells that correspond to more simple classes. This effect is more prominent when doing the analysis on class level than on app level.

In addition, we analyzed the occurrence of the method-based code smells LongMethod and LongParameterList separately. We found that in iOS apps 9% of methods are considered LongMethod while this is the case for 14% of the methods in Android apps. In iOS apps 5% of the methods have a LongParameterList while this is the case for 9% of methods in Android apps.

5. Threats to Validity

Internal Validity: In our case internal validity might be affected by how code smells are detected by the tools used. PAPRIKA has been used in multiple studies [15, 13, 3, 14]. GraphifySwift was in-

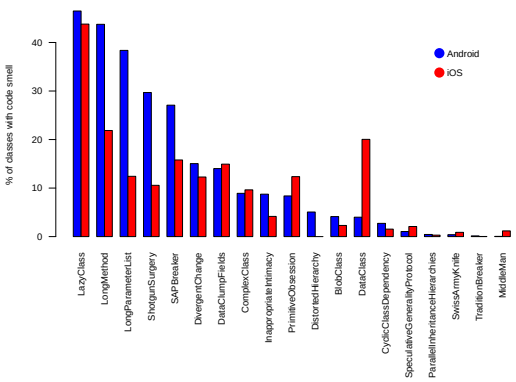


Figure 4: Comparison of code smell frequencies on class level between Android (blue) and iOS (red)

troduced in [5] and validated by replicating results in [3]. We adapted code smell queries defined in [5], but did so by not changing the code smell definitions themselves.

External Validity: We analysed open source apps. For swift the analysis can only be performed if the code of the app is accessible. For Android the analysis could be performed on apps from the app store. Therefore for both platforms open source apps were chosen. Previously [5] it was shown that although there are some differences between apps that are on the app store the differences are small. On the iOS platform we only analyzed apps written in Swift. Given that Objective-C and Swift code is quite similar, we assume our results extend to apps written in Objective-C.

Construct Validity: GraphifySwift uses standard definitions of code smells found in literature [5]. In code smell queries we use thresholds calculated based on the app set analysed. Using thresholds is a common approach for detecting code smells. We used the same method to determine thresholds as was used by Hecht et al. [13], Habchi et al. [3] and [5]. Thresholds might differ between languages, but since they are calculated based on the current set of apps analysed language specific differences should be resolved.

Reliability: For iOS analysis we used the same collaborative list of open source iOS apps written in Swift as was used in [5]. All these apps are available on GitHub. The list of successfully analysed apps can be found on the tool GitHub page. GraphifySwift is open source and also available on the tool GitHub page. For Android analysis we used the list of apps analysed by [3], the list of successfully analysed apps can be found in the list of apps[4]. PAPRIKA is open source and also available on GitHub. The adapted code smell queries used for Android analysis can be found in the list of Android code smell queries[5].

6. Conclusion

Mannan et al. [2] analysed the density and distribution of code smells in Android apps. We calculated a similar density and distribution for iOS and Android apps and saw that these densities and distributions were different. Additionally we discovered that one of the code smells analysed by Mannan et al. was not present in iOS apps.

Habchi et al. [3] compared ratios of code smell occurrences on iOS and Android. We extended their research by adding additional code smells to the analysis and found that code smell occurrences are not always higher in Android apps. For some code smells they were higher in iOS apps. This shows that Android apps are not necessarily smellier, but different kinds of code smells are more prevalent depending on the platform.

These results can be interesting for developers moving from one platform to the other. It can also be useful for developers of tools for these platforms. We see that the emphasis on which code smells to look at is different depending on the platform.

Acknowledgments

This research was partly funded by the Estonian Center of Excellence in ICT research (EXCITE), the IT Academy Programme for ICT Research Development, the Austrian ministries BMVIT and BMDW, and the Province of Upper Austria under the COMET (Competence Centers for Excellent Technologies) Programme managed by FFG,

[4] https://figshare.com/articles/dataset/iOS_and_Android_app_analysis_data/13103012

[5] https://figshare.com/articles/conference_contribution/GraphifySwift_queries_adapted_for_PAPARIKA_for_Android_code_smell_analysis/13102994

and by the group grant PRG887 of the Estonian Research Council. We thank Rudolf Ramler for the thorough review of a previous version of this paper.

References

[1] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.

[2] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, C. Jensen, Understanding code smells in android applications, in: 2016 IEEE/ACM Int'l Conf. on Mobile Softw. Eng. and Systems (MOBILESoft), IEEE, 2016, pp. 225–236.

[3] S. Habchi, G. Hecht, R. Rouvoy, N. Moha, Code smells in ios apps: How do they compare to android?, in: 2017 IEEE/ACM 4th Int'l Conf. on Mobile Softw. Eng. and Systems (MOBILESoft), IEEE, 2017, pp. 110–121.

[4] G. Hecht, An approach to detect android antipatterns, in: Proc. of the 37th Int'l Conf. on Software Engineering-Volume 2, IEEE Press, 2015, pp. 766–768.

[5] K. Rahkema, D. Pfahl, Empirical study on code smells in ios applications, in: Proc. of the IEEE/ACM 7th Int'l Conf. on Mobile Softw. Eng. and Systems, MOBILESoft '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 61–65.

[6] F. Khomh, M. Di Penta, Y.-G. Gueheneuc, An exploratory study of the impact of code smells on software change-proneness, in: 2009 16th Working Conf. on Reverse Engineering, IEEE, 2009, pp. 75–84.

[7] S. Olbrich, D. S. Cruzes, V. Basili, N. Zazworka, The evolution and impact of code smells: A case study of two open source systems, in: 2009 3rd Int'l symposium on empirical software engineering and measurement, IEEE, 2009, pp. 390–400.

[8] M. Linares-Vásquez, S. Klock, C. McMillan, A. Sabané, D. Poshyvanyk, Y.-G. Guéhéneuc, Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps, in: Proc. of the 22nd Int'l Conf. on Program Comprehension, ACM, 2014, pp. 232–243.

[9] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk, When and why your code starts to smell bad, in: Proc. of the 37th Int'l Conf. on Software Engineering-Volume 1, IEEE Press, 2015, pp. 403–414.

[10] T. Sharma, Extending Maintainability Analysis Beyond Code Smells, Ph.D. thesis, 2019.

[11] M. Gottschalk, J. Jelschen, A. Winter, Saving energy on mobile devices by refactoring., in: EnviroInfo, 2014, pp. 437–444.

[12] M. Ghafari, P. Gadient, O. Nierstrasz, Security smells in android, in: 2017 IEEE 17Th Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM), IEEE, 2017, pp. 121–130.

[13] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, L. Duchien, Tracking the software quality of android applications along their evolution (t), in: 2015 30th IEEE/ACM Int'l Conf. on Automated Softw. Eng. (ASE), IEEE, 2015, pp. 236–247.

[14] B. G. Mateus, M. Martinez, An empirical study on quality of android applications written in kotlin language, Empirical Software Engineering (2018) 1–38.

[15] G. Hecht, R. Rouvoy, N. Moha, L. Duchien, Detecting antipatterns in android apps, in: Proc. of the Second ACM Int'l Conf. on Mobile Softw. Eng. and Systems, IEEE Press, 2015, pp. 148–149.