Minimizing Refactoring Effort through Prioritization of Classes based on Historical, Architectural and Code Smell Information

Aabha Choudhary
National Institute of Technology
Jalandhar, India
E-mail: aabhasgnr@gmail.com

Paramvir Singh
National Institute of Technology
Jalandhar, India
E-mail: singhpv@nitj.ac.in

***Abstract*—Improving a software system's internal structure through regular refactoring is considered vital for its long and healthy life. However, despite its amenities, refactoring is not readily adopted by software development teams in industry mainly due to strict project deadlines and limited resources. Hence, they look for optimal refactoring recommendations that would incur minimal effort overhead while outputting decent benefits in terms of enhanced software quality. To this end, we propose an approach for identifying and prioritizing object-oriented software classes in need of refactoring. Our approach first identifies the most refactoring-prone as well as architecturally relevant classes, and then generates class ranks based on the code smell information. In addition to locating classes with the most significant incremental refactoring opportunities, this work contributes through suggesting developers on estimating maximum code smell correction (paying off maximum technical debt) with minimum refactoring effort. We evaluated the proposed approach on a sample of 1621 classes and 2358 code smell instances, distributed over 28 versions of four open source java systems.**

Keywords—class prioritization; software refactoring; code smell; refactoring effort; technical debt

I. INTRODUCTION

Maintenance and evolution are lifelines for the success of a product in modern day software development. Evolving software requires its design and code to be optimized periodically in order to avoid any technical debt resulting from decaying of such artifacts [1]. The decision over a given software change is critical and requires expertise on the part of software developers. Software refactoring is a simple yet effective approach that enables developers improve the design structure of software while preserving its perceived external behavior [2]. In order to limit the maintenance cost and improve the quality of the software system, ideally software companies try to incorporate refactoring practices as an integral part of their development and maintenance processes [1].

However, the ground reality is somewhat different. Not only are the developers expected to regularly enhance the quality of software, they are also under a constant pressure to spend most of their person hours adding new features rather than refactoring the source code [4]. Some of the major hurdles in refactoring adoption in industrial projects include, getting management buy-in, deadline pressure, inadequate refactoring tool support, etc. [1]. Also, the entire process of refactoring comprises a number of distinct activities that make it a tedious and expensive phenomenon [3]. Consequently, various automated tools supporting different refactoring activities have been proposed, which help in reducing manual effort, time consumption and errors; thus bringing down the overall evolution complexity and cost. However, on the other hand, these automated tools have their own issues too. For instance, code smell detection tools yield numerous results which are quite hard to examine. This scenario demands a balanced approach from the refactoring research community to help developers introduce refactoring to the rest of stakeholders as a significant tool for continuous quality [4] without adversely affecting project deadlines and cost.

This work looks for a solution in the Law of the Vital Few, which states - "only 20% of code contains 80% of errors" [5]. We propose a class prioritization approach that is capable of identifying, at the top of the generated class priority list, a set of most crucial (decided on architectural relevance alongwith code smell information) and refactoring-prone (decided on historical information) application classes in need of urgent refactoring. We further investigate whether prioritizing classes in need of refactoring using the proposed approach might help in achieving an affordable balance between the estimated refactoring effort savings (in terms of number of classes to be refactored) and amount of code smell correction (in terms of number of smell instances to be removed).

II. RELATED WORK

Palomba et al. [19] proposed approaches that involve examining the version history of software's source code to identify the code smell instances in the current software version. Macia et al. [7] worked diligently to probe the relation between code anomalies and architectural problems. Oizumi et al. [8] carried this research further and introduced a new approach that explores the relationship between code anomaly agglomerations and architectural problems.

Tsantalis et al. [9] proposed an approach for ranking the refactoring opportunities based on historical volatility. Meananeatra [10] presented a technique that generates an optimal refactoring sequence for improving software maintenance. Ouni et al. [11] described a search-based approach for identifying the most appropriate refactorings based on chemical reaction optimization. Steidl et al. [12] introduced a prioritization scheme for two code smells, *Code Clones* and *Long Method*, based on the expected low costs involved in the correction of these code smells. Vidal et al. [13] presented a semi-automated approach for prioritizing the code smells based on three different criteria: code smell relevance, past modifications and modifiability scenarios of the software.

There is limited research work performed in the area pertaining to the prioritization of classes in need of urgent refactoring treatments. Zhao et al.[14] prioritized classes based on a weighted maintainability rank for each class containing bad smells, utilizing different class characteristics such as size, complexity, etc. In a data mining based study, Kosker et al.

[15] applied weighted Naïve Bayes (NB) algorithm to predict the classes in urgent need of refactoring. Malhotra et al. [16] prioritized the software classes based on Quality Depreciation Index Rule (QDIR) metric, which measures the quality of a class based on the number of bad smells present and a set of object-oriented design metrics values for each class.

III. Proposed Approach

Our class prioritization approach follows a three step process as explained next.

A. Analysis of Versions (AV)

We follow the hypothesis stated by Girba et al. [6] that the classes that were frequently refactored in the past are more likely to undergo refactoring in the future. These classes can be categorized as refactoring-prone classes. The remaining classes that did not experience refactoring in any of the previous versions are considered to be least harmful; so we filter out such classes at this step.

B. Analysis of Architecturally relevant Classes (AC)

Another major factor that should be considered when identifying candidates for refactoring is the extent to which they are harmful to the system's architectural design. Architectural problems have more detrimental impact on the quality and lifecycle of the system than other traditional code smells. Therefore, we select those classes that contain code smells having a direct relationship with such architectural problems [8]. The current version of the software is analyzed to locate the architecturally-relevant classes as they are considered to be the pillar classes of the software design, and a delay in their improvement can cause deteriorating effects on the system's quality.

The data from the above two steps are combined and the common set of classes, which are both architecturally relevant as well as frequently refactored, are provided as input to the third step. The rest of the classes are discarded.

C. Generation of Rank (GR)

The resulting crucial and refactoring-prone classes are then ordered according to their impact on the system's quality. The classes are ranked using class scores generated as follows:

$$Class\ Score = F \times S \times \sum (S(x_i) \times I(x_i)) \quad (1)$$

Here, F is the frequency score of a class. Every time a class is found to be refactored at least once when comparing two subsequent software versions, its frequency score is incremented by 1. S is the severity score of a class. It measures the negative impact of a class on the quality attributes of the system. It is measured by exploiting several software metrics like size, cohesion, coupling, complexity, etc. [17]. $S(x_i)$ is the severity score of a particular code smell x_i for a given class. Each code smell instance has a different effect on the system design. This score represents the relative negative impact of the code smell instances. $I(x_i)$ represents the number of instances of a code smell x_i present in a class. Here, i identifies a particular code smell.

Once the class scores are generated, all these classes are ranked in decreasing order of their scores. Hence larger the *Class Score* value, higher the rank of the class; signifying the need of refactoring for a class. A threshold value can be associated with the class ranks to leave out the lowest ranked classes present in the sorted list, thus further reducing the number of shortlisted classes.

An example for the proposed approach and additional information regarding this work is publicly available[1].

IV. Study Design

We chose four open-source Java applications to perform our experiments. Table I provides the descriptive characteristics (number of classes, Kilo Lines Of Code (KLOC), and number of code smell instances) of the selected systems, namely HealthWatcher[2], orDrumbox[3], GanttProject[4] (Gantt for short) and JHotdraw[5].

A. Methodology

The overall research methodology followed is shown in Fig. 2. The first step of the proposed approach involves analyzing different stable versions of a given input system. The classes (belonging to each current version), which have been refactored at least once in previous software versions, are recorded alongwith their frequency score values. For identifying the number of frequently refactored classes in each software system, Ref-Finder tool [18] is used. Classes that have never been refactored before are filtered out at this step, and the remaining classes are recorded. At the next step, the current version of the software system is analyzed with the help of Organic[6] tool. For our analysis, we considered those classes as architecturally-relevant, which are identified by intra-boundary and cross-boundary topologies [8]. An intersection of the class sets obtained from the aforementioned analysis steps are used as input for the final step.

In the final step, JSpIRIT[7] is used to detect code smell instances in the current versions of the sample applications. Although JSpIRIT is generally used for prioritizing code smells, we exploited its detection features only (as it does not provide the code smell severity scores as desired for this study). It supports the identification of 10 code smells using a software metric-based detection strategy. This metric-based strategy dissolves our need to calculate the object-oriented metrics values separately for predicting smelly classes, as the classes will be automatically categorized as 'smelly' according to the presence of code anomalies. Thereafter, inFusion[8] tool is used to calculate the severity index of classes and code smell instances. It supports the identification of 7 of the 10 code smells detected by JSpIRIT. Hence, we use only these 7 code smells[1] in this work.

B. Evaluation of Proposed Approach

We utilize the approach evaluation parameters: Code smells Correction Ratio (CCR) [11] and Estimated Effort (EE) [16] to evaluate our proposed approach. CCR is defined as the total number of code smell instances to be removed by refactoring the prioritized classes, divided by the total number

[1] http://www.pvsingh.com/a_choudhary
[2] http://ptolemy.cs.iastate.edu/design-study/#healthwatcher
[3] http://sourceforge.net/projects/ordrumbox
[4] http://ganttproject.biz/index.php
[5] http://jhotdraw.org
[6] http://wnoizumi.github.io/organic/plugin
[7] https://sites.google.com/site/santiagoavidal/projects/jspirit
[8] http://www.intooitus.com/products/infusion [Last Accessed - Mar 10, 2016]

TABLE I. VERSION-WISE SIZE AND CODE SMELL CHARACTERISTICS OF SAMPLE APPLICATIONS

App. ►	HealthWatcher										orDrumbox			GanttProject								JHotDraw					
Ver. no.	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***0.9.08***	***0.9.22***	***0.9.23***	***2.6.1***	***2.6.2***	***2.6.3***	***2.6.4***	***2.6.5***	***2.6.6***	***2.7***	***2.7.1***	***2.7.2***	***7.1***	***7.2***	***7.3***	***7.4***	***7.5***	***7.6***
# Classes	88	92	104	107	108	112	116	120	132	135	195	206	217	499	500	504	507	510	511	517	518	521	528	699	717	718	753	753
KLOC	8	8.5	9.1	9.3	9.6	9.7	9.8	9.9	10.5	11.5	32	34	35	64	65	65.5	66.7	67.3	68	69	69.4	69	93	123	125	126	133	135
# Smell instances	37	37	41	41	40	42	44	43	53	60	404	422	458	541	596	584	601	623	599	641	653	668	1003	855	947	986	1172	1172

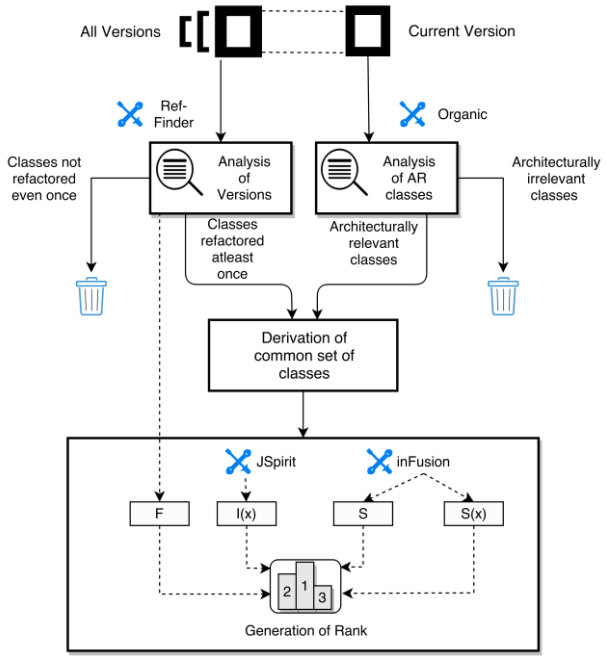

Fig.2.Methodology flow for the evaluation of the proposed approach

of code smell instances present in the software. It is given by (2).

$$\text{CCR} = \frac{\text{Numberof code smell instances to be removed}}{\text{Total numberof code smell instances in the system}} \quad (2)$$

EE is defined as the total number of classes that needs to be refactored divided by the total number of classes present in the software. After calculating the total estimated effort for refactoring, the total reduction in refactoring effort can be deduced. EE is given by (3).

$$\text{EE} = \frac{\text{Number of classes to be refactored}}{\text{Total numberof classes in the system}} \quad (3)$$

V. PRELIMINARY RESULTS AND ANALYSIS

The evolution of sample applications is quantified with the help of three version specific characteristics mentioned in Table I. The percentage of refactored classes between any two subsequent versions is always greater than 0% across all applications, which is quite normal for real world software systems. On average, 30% of the total number of classes is architecturally relevant across all four sample applications. This indicates that 30% (avg.) of the classes stand critical to the respective architectures of the current versions of these applications, and hence need immediate refactoring. Further, it is inferred that, on average, out of those 30% classes, 21% classes are refactored more than once in the previous versions. Thus, these 21% classes are chosen as the most significant and refactoring-prone classes.

Table II highlights three top ranked classes for each application, respectively. Note that in this table, the two penultimate columns are devoted to the total number of code smell instances in a class and average of code smells severity values, respectively. However, for the actual calculation of the class scores (when exercising the proposed approach), we considered the individual code smells alongwith their respective severity scores. It is conspicuous from Table II that the four class score parameters (Section III.C) contribute uniformly in generating balanced ranks for the classes. E.g. for orDrumbox, class ControlerProduct has higher severity score than classes OrTrack, Song, and Command; but due to low average number of code smell instances and frequency score, it records a low class score (and rank). Moreover, it is found that the classes having higher frequency scores have a higher number of code smell instances too. Thus it is revealed that despite of having been refactored in the earlier versions, such classes are still smelly and need further attention.

The evaluation results are summarized in Table III. On average, nearly 40% (EE) of the total number of classes needs frequent refactoring treatments. At the same time, the CCR scores indicate that more than 75% of the code defects fall within these 40% classes. On applying refactoring treatments to these classes, a significant improvement in software quality is ensured as indicated by high CCR scores. At the next step (AC), we observed that almost half of the previously refactored classes do not pose any threat to the architectural degradation of the selected systems. For instance, for JHotdraw, 158 of 278 classes (56%) are architecturally relevant. Consequently, EE score for JHotDraw dropped to 20% with a minimal decline in CCR score. Similar results are obtained for orDrumbox; a refactoring effort of 26% and smell correction of 73%. We gathered the results for those classes having ranks above a custom threshold value of 20. The developers can choose any threshold values that help them establish the desired balance between the amount of smell correction and estimated refactoring effort for their projects.

The results revealed that the search space for refactoring opportunities (number of classes) is constantly shrinking with

TABLE II. HIGH PRIORITY CLASSES OFSAMPLE APPLICATIONS

Rank	Class name	F	S	Total $I(x_i)$	Avg. $S(x_i)$	Class Score
HealthWatcher						
1	FoodComplaintStateClosed	5	32.4	7	7.1	7938.0
2	Situation	4	28.0	5	4.4	2576.0
3	ComplaintRepositoryRDB	3	30.0	2	4.0	1440.0
OrDrumbox						
1	Command	2	70.0	38	6.8	8960.0
2	Ortrack	2	49.6	32	5.6	8928.0
3	Song	2	49.4	18	4.2	2568.8
GanttProject						
1	Taskmanagerimpl	5	57.4	17	7.0	34153.0
2	GanttProject	3	50.8	12	5.7	12649.2
3	GanttOptions	4	35.2	12	7.1	11827.2
JHotDraw						
1	SVGInputFormat	5	18.0	33	9.0	26730.0
2	DefaultDrawingView	4	14.0	28	8.2	12096.0
3	Bezier	2	11.2	22	7.6	4659.2

TABLE III: EVALUATION METRICS RESULTS FOR THE FOUR SYSTEMS STUDIED

System	Step	EE	CCR
HealthWatcher	AV	42% (58|135)	70% (42|60)
	AC	19% (26|135)	65% (39|60)
	GR	**16% (22|135)**	**60% (36|60)**
OrDrumbox	AV	44% (97|217)	76% (349|458)
	AC	26% (58|217)	73% (336|458)
	GR	**18% (39|217)**	**63% (289|458)**
GanttProject	AV	35% (182|521)	79%(528|668)
	AC	17%(99|521)	75%(501|668)
	GR	**15%(79|521)**	**70%(470|668)**
JHotDraw	AV	37% (278|753)	75% (885|1172)
	AC	20% (158|753)	70% (827|1172)
	GR	**17% (132|753)**	**67% (789|1172)**

each step of the proposed approach. We observed that on average (across all sample applications) the refactoring effort required to remove 65% (avg. CCR) of code smells comes out to be 16% (avg. EE); hence saving an estimated refactoring effort up to 84%. For HealthWatcher, EE value shows a similar trend whereas its CCR value is slightly less as compared to that for other applications; but still acceptable. Overall results indicate that a major chunk of code anomalies could be targeted with a significant reduction in refactoring effort. This approach provides the developers with an option to choose any number of top ranked classes to refactor depending upon an affordable balance between the code smell corrections and estimated refactoring effort.

VI. THREATS TO VALIDITY

A construct validity threat concerns the possible errors (like some classes may be missed or may be identified as false positives) while identifying the refactored classes, architecturally relevant classes and code smell instances. To mitigate this threat, we considered widely known and fairly accurate code smell detection and refactoring detection tools that have been used in the relevant literature. While analyzing the software versions, we considered all those classes that have been refactored more than once. It would also include such classes that underwent a change just once during their evolution across various software versions. To mitigate this threat, we manually analyzed the classes, and decided whether they should be included or not. We analyzed four small and medium-sized open source java systems. This could affect the generalization of our approach when applied to large-sized commercial systems as well as systems written in other programming languages. However, the variable-sized sample applications along with the consistency in the experimental results across those applications minimize this threat to some extent.

VII. CONCLUSIONS AND FUTURE WORK

In this study, we presented an approach for the prioritization of classes in need of refactoring. The proposed approach combines three different perspectives, i.e. historical data, architectural design, and severity of the class; and generates a prioritized list of classes. By prioritizing the classes based on our approach, up to 84% of refactoring effort was expected to be saved while also eradicating 65% (approx.) of the code smells for the four selected software applications. The proposed approach and the related findings can be useful for the development teams, which are short on project time and budget, to have refactoring processes on board. Also, such a prioritization of refactoring opportunities might help the developers in doing away with having to process large sets of refactoring recommendations.

We intend to extend this work in a number of ways. Since our initial experimental study includes small and medium-sized projects, we plan to replicate our experiments to the large-sized systems and industrial projects in future. An important future work would be to aim more number of distinct code smells to further gain in terms of saved refactoring effort and increased code smell correction. We also aim to compare our results with previous class prioritization techniques targeting refactoring. Lastly, we intend to implement our approach in the form of an automated tool that provides a prioritized list of classes in need of refactoring, and let the developers opt for the right balance between the estimated refactoring effort savings and expected code smell corrections.

REFERENCES

[1] T. Sharma, G. Suryanarayana, and G. Samarthyam, "Challenges to and Solutions for Refactoring Adoption: An Industrial Perspective," *IEEE Software*, vol. 32, no. 6, pp. 44–51, Nov. 2015.

[2] M. Fowler, Refactoring: Improving the Design of Existing Code. Pearson Education India, 2009.

[3] T. Mens and T. Tourwé, "A Survey of Software Refactoring," *IEEE Trans Softw. Eng.*, vol. 30, no. 2, pp. 126–139, 2004.

[4] M. Leppänen, S. Mäkinen, S. Lahtinen, O. Sievi-Korte, A. P. Tuovinen, and T. Männistö, "Refactoring-a Shot in the Dark?," *IEEE Software*, vol. 32, no. 6, pp. 62–70, Nov. 2015.

[5] R. S. Pressman, Software Engineering: A Practitioner's Approach. Palgrave Macmillan, 2005.

[6] T. Girba, S. Ducasse, and M. Lanza, "Yesterday's Weather: guiding early reverse engineering efforts by summarizing the evolution of changes," In *Proc. Int'l Conf. on Softw. Maint.*, pp. 40–49, 2004.

[7] I. Macia, A. Garcia, C. Chavez, and A. von Staa, "Enhancing the Detection of Code Anomalies with Architecture-Sensitive Strategies," In *Europ. Conf. on Softw. Maint. and Reeng*,pp. 177–186, 2013.

[8] W. N. Oizumi, A. F. Garcia, T. E. Colanzi, M. Ferreira, and A. V. Staa, "On the relationship of code-anomaly agglomerations and architectural problems," *J. Softw. Eng. Res. Dev.*, vol. 3, no. 1, Dec. 2015.

[9] N. Tsantalis and A. Chatzigeorgiou, "Ranking Refactoring Suggestions Based on Historical Volatility," In *European Conference on Software Maintenance and Reengineering*, pp. 25–34, 2011.

[10] P. Meananeatra, "Identifying Refactoring Sequences for Improving Software Maintainability," In *Proc. of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 406–409, 2012.

[11] A. Ouni, M. Kessentini, S. Bechikh, and H. A. Sahraoui, "Prioritizing code-smells correction tasks using chemical reaction optimization," *Softw. Qual. J.*, vol. 23, no. 2, pp. 323–361, 2015.

[12] D. Steidl and S. Eder, "Prioritizing Maintainability Defects Based on Refactoring Recommendations," In *Proceedings of International Conference on Program Comprehension*, pp. 168–176, 2014.

[13] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Autom. Softw. Eng.*, pp. 1–32, Dec. 2014.

[14] L. Zhao and J. H. Hayes, "Rank-based Refactoring Decision Support: Two Studies," *Innov Syst Softw Eng*, vol. 7, no. 3, pp. 171–189, 2011.

[15] Y. Kosker, B. Turhan, and A. Bener, "An expert system for determining candidate software classes for refactoring," *Expert Syst. Appl.*, vol. 36, no. 6, pp. 10000–10003, Aug. 2009.

[16] R. Malhotra, A. Chug, and P. Khosla, "Prioritization of Classes for Refactoring: A Step Towards Improvement in Software Quality," *Proc. Int'l Symp. Women in Computing and Informatics*, pp. 228–234, 2015.

[17] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg, "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems," *Int'l Conf. Softw. Maint.*, pp. 1–10, 2010.

[18] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-Finder: A Refactoring Reconstruction Tool Based on Logic Query Templates," *Proc.ACM Int'l Symp. Found. of Softw. Eng.*, pp. 371–372, 2010.

[19] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk and A. De Lucia, "Mining Version Histories for Detecting Code Smells,"In *IEEE Trans. on Softw. Eng.*, vol. 41, no. 5, pp. 462-489, May 1 2015.