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.