Software Refactoring Under Uncertainty: A Robust Multi-Objective Approach

Wiem Mkaouer, Marouane Kessentini, Slim Bechikh
University of Michigan, USA
firstname@umich.edu

Mel Ó Cinnéide
University College Dublin, Ireland.
mel.ocinneide@ucd.ie

Kalyanmoy Deb
Michigan State University, USA.
kdeb@egr.msu.edu

ABSTRACT

Refactoring large systems involves several sources of uncertainty related to the severity levels of code smells to be corrected and the importance of the classes in which the smells are located. Due to the dynamic nature of software development, these values cannot be accurately determined in practice, leading to refactoring sequences that lack robustness. To address this problem, we introduced a multi-objective robust model, based on NSGA-II, for the software refactoring problem that tries to find the best trade-off between quality and robustness.

Categories and Subject Descriptors
D.2 [**Software Engineering**].

General Terms
Algorithms, Reliability.

Keywords
Search-based software engineering, software quality, code smells, robust optimization.

1. INTRODUCTION

Large-scale software systems exhibit high complexity and become difficult to maintain. It has been reported that the cost of maintenance and evolution activities comprises more than 80% of total software costs. In addition, it has been shown that software maintainers spend around 60% of their time in understanding the code. To facilitate maintenance tasks, one of the widely used techniques is refactoring which improves design structure while preserving the overall functionality of the software.

There has been much work on different techniques and tools for refactoring [2][3][4].The vast majority of these techniques identify key symptoms that characterize the code to refactor using a combination of quantitative, structural, and/or lexical information and then propose different possible refactoring solutions, for each identified segment of code. In order to find out which parts of the source code need to be refactored, most of the existing work relies on the notion of design defects or code smells. The generic term code smell refers to structures in the code that suggest the possibility of refactoring. Once code smells have been identified, refactorings need to be proposed to resolve them. Several automated refactoring approaches are proposed in the literature and most of them are based on the use of software metrics to estimate quality improvements of the system after applying refactorings.

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.
ACM 978-1-4503-2881-4/14/07.
http://dx.doi.org/10.1145/2598394.2598499

The existing literature on software refactoring invariably ignores an important consideration when suggesting refactoring solutions: the highly dynamic nature of software development. In this paper, we take into account two dynamic aspects as follows:

- *Code Smell Severity*: This is the severity level assigned to a code smell type by a developer. It usually varies from developer to developer, and indeed a developer's assessment of smell severity will change over time as well.
- *Code Smell Class Importance*: This is the importance of a class that contains a code smell, where importance refers to the number and size of the features that the class supports. A code smell with large class importance will have a greater detrimental impact on the software. Again, this property will vary over time as software requirements change and classes are added/deleted/split.

Existing approaches to the refactoring problem assume a static environment to the problem, i.e., that all detected code smells are of the same severity and that the importance of the class in which is the code smell is situated is not liable to change.

We believe that the uncertainties related to class importance and code smell severity need to be taken into consideration when suggesting a refactoring solution. To this end, we introduce in this paper a novel representation of the code refactoring problem, based on *robust* optimization [1] that generates robust refactoring solutions by taking into account the uncertainties related to code smell severity and the importance of the class that contains the code smell. Our robustness model is based on the well-known multi-objective evolutionary algorithm NSGA-II and considers possible changes in class importance and code smell severity by generating different scenarios at each iteration of the algorithm. In each scenario, the detected code smell to be corrected is assigned a severity score and each class in the system is assigned an importance score. In our model, we assume that these scores change regularly due to reasons such as developers' evolving perspectives on the software or new features and requirements being implemented or any other code changes that could make some classes/code smells more or less important. Our multi-objective approach aims to find the best trade-off between maximizing the quality of the refactoring solution in terms of the number of code smells corrected and maximizing its robustness in terms of the severity of the code smells corrected and the importance of the classes that contains the code smells.

The remainder of this paper is structured as follows. Section 2 provides the background required to understand our approach and the nature of the refactoring challenge. In Section 3, we describe robust optimization and explain how we formulate software refactoring as a robust optimization problem. Section 4 presents and discusses the results obtained by applying our approach to six large open-source projects. Related work is discussed in Section 5, while in Section 6 we conclude and suggest future research directions.

2. MULTI-OBJECTIVE ROBUST SOFTWARE REFACTORING

The refactoring problem involves searching for the best refactoring solution among the set of candidate ones, which constitutes a huge search space. A refactoring solution is a sequence of refactoring operations where the goal of applying the sequence to a software system *S* is typically to minimize the number of code smells in *S*. As outlined in the Introduction, in a real-world setting code smell severity and class importance are not certainties. A refactoring sequence that resolves the smells that one developer rates as severe may not be viewed as effective by another developer with a different outlook on smells. Similarly, a refactoring sequence that fixes the smells in a class that is subsequently deleted in the next commit is not of much value.

To address these issues, we propose a robust formulation of the refactoring problem that takes class importance and smell severity into account. Consequently, we have two objective functions to be maximized in our problem formulation: (1) the quality of the system to refactor, i.e., minimizing the number of code smells, and (2) the robustness of the refactoring solutions in relation to uncertainty in the severity level of the code smells and in the importance of the classes that contain the code smells. Analytically speaking, the formulation of the robust refactoring problem can be stated as follows:

Maximize

$$\begin{cases} f_1(x,S) = NCCS(x,S)/NDCS(S) \\ f_2(x,S) = \sum_{i=1}^{NCCS} [SmellSeverity(CCS_i,x,S) + Importance(CCS_i,x,S)] \end{cases}$$

$$subject\ \ to\ \ \ x = (x_1,...,x_n) \in X$$

where *X* is the set of all legal refactoring sequences starting from *S*, x_i is the *i*-th refactoring in the sequence *x*, *NCCS*(*x,S*) is the *Number of Corrected Code Smells* after applying the refactoring solution *x* on the system *S*, *NDCS* is the *Number of Detected Code-Smells* prior to the application of solution *x* to the system *S*, CCS_i is the i-th Corrected Code Smell, *SmellSeverity*(CCS_i, *x*, *S*) is the severity level of the *i*-th corrected code smell related to the execution of *x* on *S*, and *Importance*(CCS_i, *x*, *S*) is the importance of the class containing the *i*-th code smell corrected by the execution of *x* on *S*.

The smell's severity level is a numeric quantity, varying between 0 and 1, assigned by the developer to each code smell type (e.g., blob, spaghetti code, functional decomposition, etc.). We define the class importance of a code smell as follows:

$$Importance(CCS_i,x,S) = \frac{(NC/MaxNC(S)) + (NR/MaxNR(S)) + (NM/MaxNM(S))}{3}$$

such that *NC*/*NR*/*NM* correspond respectively to the *Number of Comments/Relationships/Methods* related to the CCS_i and *MaxNC*/*MaxNR*/*MaxNM* correspond respectively to the *Maximum Number of Comments/Relationships/Methods* of any class in the system *S*. There are of course many ways in which class importance could be measured, and one of the advantages of the search-based approach is that this definition could be easily replaced with a different one. In summary, the basic idea behind this work is to maximize the resistance of the refactoring solutions to perturbations in the severity levels and class importance of the code smells while maximizing simultaneously the number of corrected code smells. These two objectives are in conflict with each other since the quality of the proposed refactoring solution usually decreases when the environmental change (smell severity and/or class importance) increases. Thus, the goal is to find a good compromise between (1) quality and (2) robustness. This compromise is directly related to *robustness cost*, as discussed above. In fact, once the bi-objective trade-off front (quality, robustness) is obtained, the user can navigate through this front in order to select his/her preferred refactoring solution. This is achieved through sacrificing some degree of solution quality while gaining in terms of robustness. In this way, the user can seek his/her preferred solution based on the robustness cost metric corresponding to the loss in terms of quality for achieving robustness.

3. CONCLUSION AND FUTURE WORK

In this paper, we have introduced a novel formulation of the refactoring problem that takes into account the uncertainties related to code smell correction in the dynamic environment of software development where code smell severity and class importance cannot be regarded as fixed. Code smell severity will vary from developer to developer and the importance of the class that contains the smell will vary as the code base itself evolves.

Future work involves extending our approach to handle additional code smell types in order to test further the general applicability of our methodology. In this paper, we focused on the use of a structural metric to estimate class importance, but this can be extended to consider also the pattern of repository submits to achieve another perspective on class importance. In a similar vein, our notion of smell severity assumes each smell type has a certain severity, but a more realistic model is to allow each individual smell instance to be assigned its own severity. If further experiments confirm our observation that the knee point is indeed a trademark of the quality-robustness trade-off frontier for all software refactoring problems, then it would be interesting to apply straightway a knee-finding algorithm to the bi-objective problem and determine if it yields any computational benefit. In an interactive software refactoring tool, the potential speed-up might be critical to success. Overall the use of robustness as a helper objective in the software refactoring task opens up a new direction of research and application with the possibility of finding new and interesting insights about the quality and severity trade-off in the refactoring problem.

ACKNOWLEDGEMENT

This work was supported, in part, by the Institute for Advanced Vehicle Systems-Michigan grant and the Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre

4. REFERENCES

[1] G. Antoniol, M. Di Penta, and M. Harman. A Robust Search-Based Approach to Project Management in the Presence of Abandonment, Rework, Error and Uncertainty. 2004. In *Proceedings of the Software Metrics, 10th International Symposium* (METRICS '04). IEEE Computer Society, Washington, DC, USA, 172-183. DOI=10.1109/METRICS.2004.4 http://dx.doi.org/10.1109/METRICS.2004.4

[2] M. Harman; and L. Tratt. 2007. Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation* (GECCO '07). ACM, New York, NY, USA, 1106-1113. DOI=10.1145/1276958.1277176 http://doi.acm.org/10.1145/1276958.1277176.

[3] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni. Design Defects Detection and Correction by Example. 2011. In *Proceedings of the 19th International Conference on Program Comprehension* (ICPC'11). 81-90.

[4] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum. 2012. Maintainability Defects Detection and Correction: A Multi-Objective Approach. *Journal of Automated Software Engineering*. Springer. vol. 20, no. 1, 47-79. DOI= 10.1007/s10515-011-0098-8