Interactive Code Smells Detection: An Initial Investigation

Mohamed Wiem Mkaouer(✉)

Department of Computer and Information Science,
University of Michigan, 4901 Evergreen Road,
Dearborn, MI 48128, USA
mmkaouer@umich.edu

Abstract. In this paper, we introduced a novel technique to generate more user-oriented detection rules by taking into account their feedback. Our techniques initially generate a set of detection rules that will be used to detect candidate code smells, these reported code smells will be exposed in an interactive fashion to the developer who will give his/her feedback by either approving or rejecting the identified code smell in the code fragment. This feedback will be fed to the GP as constraints and additional examples in order to converge towards more user-preferred detection rules. We initially investigated the detection of three types of code smells in four open source systems and reported that the interactive code smell detection achieves a precision of 89 % and recall on average when detecting infected classes. Results show that our approach can best imitate the user's decision while omitting the complexity of manual tuning the detection rules.

1 Introduction

Code smells have been known as bad programming behavior that can be introduced during the initial software design or during its maintenance. The existence of these smells is a strong indicator for poor software quality as the infected code tends to be more difficult to understand and to update. As a consequence, the risk of introducing errors while committing regular software updates becomes alarming.

There has been much work resulting in different techniques and tools for code smells detection [1–4]. These techniques deploy different detection strategies using various structural metrics due to the inconsistency in the definition of code smells and due to the subjectivity of the code smell interpretation by the software engineers [5]. In fact, the source code used measurements, i.e., metrics, may vary from one technique to another. Also, two detection strategies using the same rules may give different results based on various thresholds that can be used when interpreting metric values. One of the main limitations of these strategies is that they impose a pre-defined definition of what is seen as bad symptoms in the code although it should be subject to the developer's interpretation.

To cope with the above mentioned limitations, we propose a novel interactive code smells detection that dynamically adapts the developers' preference by deploying detection rules that have been tuned based on their feedback. This approach starts by

using three state-of-art code smells detection techniques that each one generates a list of code smells along with their location in the code. One of the challenges is how to choose the most suitable detection technique for a given smell type. To this end, this approach starts by finding the overlapping code smells (type and location) among the detection techniques. Based on this analysis, the infected code fragments are ranked based on their frequency and suggested to the developer for each smell type. The developer can approve or reject each suggestion. This feedback is then used to evaluate the performance of the detection techniques using the accepted/rejected suggestions and rank them. In the next stage, this feedback is also used as a training set to refine the detection rules of the best-ranked detection technique. This approach was evaluated it on four open source systems.

2 Interactive Code Smells Detection

The general structure of this approach is sketched in Fig. 1. Our detection framework starts by generating, for an input software system, a list of detected code smells, for each detection strategy. Any detection strategy can be used as part of the initial detection stage as long as it is based on semi-automated or fully automated rules-based detection and its rules are defined using a set of structural metrics that can be easily computed using the code parsing and statistical analysis.

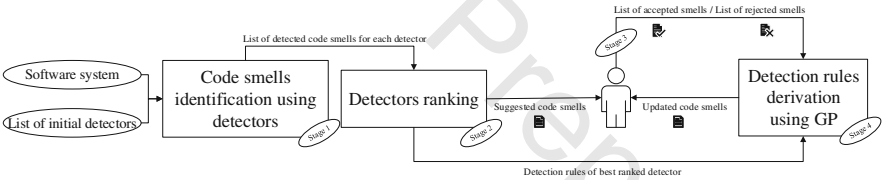

Fig. 1. The interactive Detection four main stages.

The generated lists, as outcomes of the first step, are firstly clustered per smell type. Each type is associated with a pool of possibly infected code fragments that are also classified by their originated detector. At the second stage, for each pool, the code fragments are sorted based on their occurrences among the classes of detectors, and so, for each smell type, a list of candidate code fragments to investigate is generated. In other terms, fragments are obviously sorted based on their overlap between detectors. More generally, any common feature among different strategies could be beneficial in search for more meaningful results that may achieve a tradeoff between these techniques [6].

The third stage suggests the top candidate fragments to analyze for each smell type. The developer can interactively confirm the existence of the smell in the fragment or report it as false positive. The developer does not need to evaluate the whole list of fragments, only with few evaluations, the ranking of detectors can still be effective, but the higher the number of evaluations is, per smell type, the more effective will be the

generation of detection rules using the GP that is conducted after the interactive session with the developer.

The last step takes the developer's feedback along with the highest ranked detector's rules as input to the GP. A GP algorithm is a population-based evolutionary algorithm that uses natural selection to generate an optimal solution. GP encoding is optimized for trees structure, where the internal nodes are functions (operators) and the leaf nodes are terminal symbols. Both the function set and the terminal set must contain symbols that are appropriate for the target problem which matches, for instance, the detection rules representation. During the evolution, a training set is still applied to assess the learning process. The following pseudo-code highlights the adaptation of GP for the problem of detection rules generation.

Algorithm1. Rules generation using GP

```
Input: Software System (S)
Input: Detection rules (R)
Input: Set of Accepted (SA) and Rejected (SR) code smells
Output: Derived Detection rules
1: initial_population(P, Max_size)
2: P:= set_of(I)
3: I := rules(R, Smell_Type)
4: repeat
5:     for all I ∈ P do
6:       detected_smells := execute_rules(R,S)
7:       fitness(I) := compare(detected_smells, SA, SR)
8:     end for
9:     best_solution := best_fitness(I);
10:    P := generate_new_population(P)
11:    it:=it+1;
12: until it=max_it
13: return best_solution
```


3 Initial Evaluation Study

3.1 Research Questions

We defined two research questions to address in our experiments.

RQ1: To what extend can the interactive detection assist developers in the process of smells detection?

RQ2: Can the generated rules be generalized and used in the detection of code smell instances in software systems?

The answer to RQ1 is conducted through recording the number of accepted suggestions compared to the overall suggested fragments per smell type after the execution of all the stages of the interactive detection. A group of two Ph.D. students was asked

to evaluate, manually, whether the suggested code fragments do contain the reported smell. Eventually, the number of meaningful suggestions per all suggestions constitutes the Manual Correctness (MC):

$$MC = \frac{|\text{accepted suggestion s}|}{|\text{all suggestion s}|}$$

To answer RQ2, a cross-fold validation has been conducted using the four open source systems used for in the experiment through four iterations. Precision and recall scores are calculated based on the ratio of the reported smells out of those suggested manually:

$$\text{PR}_{precision} = \frac{|\text{suggested smells} \cap \text{expected smells}|}{|\text{suggested smells}|} \in [0, 1]$$

$$\text{RC}_{recall} = \frac{|\text{suggested smells} \cap \text{expected smells}|}{|\text{expected smells}|} \in [0, 1]$$

3.2 Experimental Setting

We used a set of well-known open-source Java projects that were mainly chosen because they were the subject of several extensive studies in detection and comparison between code smells detection tools. We used two state of art code smell detectors namely InCode [7], Mäntylä et al. [5], as initial detectors for the first stage of the interactive detection. The choice of these techniques is based on the fact of their tree-based rules representation, Fig. 2 illustrates the example of the God Class detection rule based on [7]. The tree leaves are a composition of structural metrics and their ordinal values (Very_High, High, Medium, Low and Very_Low), the ordinal values are statistically interpreted using Box-Plot [8] in order to replace them with actual values extracted from the software system.

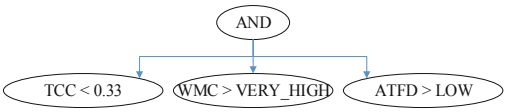

Fig. 2. Tree representation of the God Class rule in [11].

We applied our approach to four open-source Java projects: Xerces-J, JFreeChart, GanttProject, and JHotDraw. Table 1 provides some descriptive statistics about these four programs. We compared the performance of our approach with two deterministic detectors [5, 7] (previously used during the first stage) and one search-based detection rules generator [4].

Table 1. Statistics of the studied systems.

Systems	Release	# of classes/KLOC	# of flawed classes	Overlap % between detectors	# of interactive sessions with subjects	Average subjects' actions (accepted/rejected combined)
Xerces-J	v2.7.0	991/240	61	66 %	1	29
JHotDraw	v6.1	585/21	14	73 %	1	21
JFreeChart	v1.0.9	521/170	34	84 %	1	17
GanttProject	v1.10.2	245/41	19	89 %	1	12

During this study, we use the same parameter setting for all executions of the GP. The parameter setting is specified in Table 2.

Table 2. Parameter tuning for GP.

GP parameter	Values
Population size/Max Tree Depth	100/2
Selection/Survival/K	Roulette-Wheel/K-Tournament/2
Crossover/Crossover rate	Single-point/0.9
Mutation/Mutation rate	Sub-tree/0.1
Max iterations	1000/2500/5000

3.3 Results and Discussions

As an answer to RQ1, Fig. 3 reports the results of the empirical qualitative evaluation of the detection rules in terms of the MC ratio.

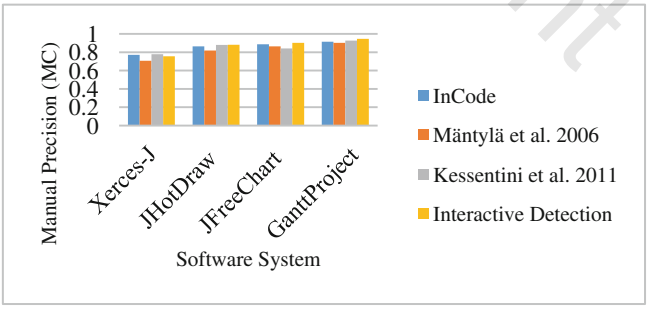

Fig. 3. Median of *MC* on all four software systems using different rules detection techniques.

Table 3. Median values of precision and recall for the detection of God Class, BLOB and Data Class in 4 systems over 30 runs.

Software	God Class		BLOB		Data Class	
	Precision (%)	Recall (%)	Precision (%)	Recall (%)	Precision (%)	Recall (%)
InCode	91	96	84	85	97	99
Mäntylä et al.	86	89	78	82	94	96
Kessentini et al.	88	97	82	96	89	97
Interactive detection	89	98	85	87	95	98

As reported in Fig. 3, the majority of the code smells detected our approach gained the satisfaction of the subjects. It is clear that the least performance of our approach in terms of median of accepted code smells among all reported ones over all the three smell types is with Xerces-J, which is the largest software used in our experiment, this can be explained by the fact that our approach may need a larger number of interactive sessions especially that the ratio of the number of interactions per number of flawed classes is relatively low compared to the other projects. For medium to small projects, the interactive detection performance was relatively acceptable.

In addition to the qualitative evaluation, we automatically evaluate our approach in terms of precision and recall to give more quantitative evaluation and answer RQ2. It is notable that we used the same training process for our approach as well as the By-Example approach of Kessentini et al. [4]. Since InCode [7], Mäntylä et al. [5] use pre-defined detection rules, no fold training was necessary for them and since they were deterministic approaches, no multiple runs were required as well. Then, we compare the proposed detected smells with some expected ones defined manually by the different groups for several code fragments extracted from the four systems. Table 3 summarizes our finding.

4 Conclusion and Future Work

We proposed, in this paper a novel interactive recommendation tool, for the problem of code smells detection rules' generation. The empirical study shows promising results as well as several further investigations to be conducted as part of the future work. Future work should also validate our approach with additional smells types, larger systems and especially a threshold that defines the maturity of the generated rules in order to draw conclusions about the general applicability of our methodology. We are planning on automating the whole smell management process through the combination of this approach as a first phase with the correction phase that has been the subject of a previous study [9].

References

1. Mäntylä, M., Vanhanen, J., Lassenius, C.: A taxonomy and an initial empirical study of bad smells in code. In: Proceedings of Conference Name, Conference Location, pp. 381–384 (2003)
2. Marinescu, R.: Detection strategies: metrics-based rules for detecting design flaws. In: Proceedings of Conference Name, Conference Location, pp. 350–359 (2004)
3. Moha, N., Gueheneuc, Y.-G., Duchien, L., Le Meur, A.-F.: DECOR: a method for the specification and detection of code and design smells. IEEE Trans. Softw. Eng. **36**(1), 20–36 (2010)
4. Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., Ouni, A.: Design defects detection and correction by example. In: Proceedings of Conference Name, Conference Location, pp. 81–90, 22–24 June 2011
5. Mäntylä, M.V., Lassenius, C.: Subjective evaluation of software evolvability using code smells: an empirical study. Empirical Softw. Eng. **11**(3), 395–431 (2006)
6. Deb, K., Srinivasan, A.: Innovization: innovating design principles through optimization. In: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, Seattle, Washington, USA (2006)
7. Marinescu, R., Ganea, G., Verebi, I.: inCode: continuous quality assessment and improvement. In: Proceedings of Conference Name, Conference Location, pp. 274–275 (2010)
8. Williamson, D.F., Parker, R.A., Kendrick, J.S.: The box plot: a simple visual method to interpret data. Ann. Intern. Med. **110**(11), 916–921 (1989)
9. Mkaouer, M.W., Kessentini, M., Bechikh, S., Deb, K., Ó Cinnéide, M.: Recommendation system for software refactoring using innovization and interactive dynamic optimization. In: Proceedings of Conference Name, Conference Location, pp. 331–336 (2014)