Essay: Competitive coevolutionary code-smells detection

Jingdie Liu

University of Adelaide

Abstract. Code smells is a defect that may affect the maintenance and evolution of a system. This essay discusses the novelties, comparison, future works and critiques of Boussaa *et al.*'s paper that discuss code smells detection by CCEA. The novelties include the utilization of algorithm to tackle problems and representation of individuals. When compared with related research, the solution of code-smells is widely using the manual detection data as benchmark, and there are many other algorithms are used in detecting code smells. The possible future works may be related to be able to detect more kinds of code smells and the refactoring after detection. However, the efficiency of the algorithm is compared with the algorithms that are not in the same level, making the paper less convincing. Nevertheless, the limited increase in the precision and recall is at the expense of running time.

1 Introduction

Code smells are generated in the process of programming. This kind of design anomalies may make the software difficult to be maintained [3]. In 2013, Boussaa *et al.* explored the application of competitive evolutionary algorithm in detecting code smells [1]. After designing the two populations which affect mutually based on the rules of CCEA, the validation is also discussed. The novelties including the utilization of algorithm to tackle problems and representation of individuals will be discussed in Section 2. In section 3, when compared with related research, the solution of code-smells is widely using the manual detection data as benchmark, and there are many other algorithms are used in detecting code smells. Section 4 is dedicated to the possible future works, it may be related to be able to detect more kinds of code smells and the refactoring after detection. There is the critique in section 5. In the research, the efficiency of the algorithm is compared with the algorithms that are not in the same level, making the paper less convincing. Nevertheless, the limited increase in the precision and recall is at the expense of running time.

1.1 Motivation

Code smells are not reported in the way of reporting bugs, and they are prone to be neglected by most programmers because it will not cause a failure to a system.

It is well known that even if the code is not readable or hard to be extended, the partial goal can still be achieved in specific stages successfully. Therefore, programmers may not care them because there may be someone other to be responsible for next stage. In [2], the survey about the concern to code smell showed that there were only 13% of the investigated programmers understood the concept and apply it in their coding works. However, Brown et al. indicated that the detection and removal of code smell can help developers to easily understand source code [3]. More seriously, bad-smells may cause failures indirectly, since they made a system hard to change, leading to introduce bugs. So far, there is not an application that can detect code smell 100% correctly. This may be one of the reasons that there are no many people intend to improve the quality of code by removing code smells, because it costs long time to detect all unacceptable code smells manually. The gap between the importance of code smells and the ignorance to them motivates the need for the research to an efficient way to detect code smells so that the developers can notice and remove them as soon as possible.

1.2 Background

Some researchers have already realized the disadvantages of bad smells. In 2010, Obirich *et al.* investigated two code smells, God Class and Shotgun Surgery in developing evolution, to see the impact of code smells [4]. The result shows that the classes infected with code smells have higher possibility to be changed in the following evolutions than non-infected classes. That means the cost of maintaining code-smell snippets is higher than normal snippets. Similarly, Khomh *et al.* stated that classes with code smells are more change-prone than others [5]. The changes in the development and maintains means that the developers have to put more efforts on it. Therefore, the bad smell of code should be detected and removed as soon as possible to avoid excessive efforts.

On the other hand, Khomh *et al.* put forward that the code smells can be detected because they are mostly defined in terms of thresholds on metrics [5]. However, the value of the thresholds varied as the size of system, so it is necessary to define a suitable threshold when exploring the problems related with detection of code smells. To solve this problem, Fontana *et al.* propose a data-driven method to derive threshold values for code metrics, which can be used for implementing detection rules for code smells [6]. In this way, the method is transparent and repeatable and can provide the moderate amount of observations when detecting code smells in different size of programs.

2 Novelties

2.1 2.1 A utilization of algorithm to tackle problems

In the essay, the authors explored the detection of code smells by competitive coevolution algorithm (CCEA). This algorithm helps the research to get a better result than single population algorithms do. For CCEA, there are multiple sub-populations to show the co-evolution. Meanwhile, the fitness value of a particular individual depends on the fitness values of other individuals belonging to other sub-populations, and this feature shows the competition among the sub-populations. According to [7], this kind of evolution can not only produce fitter individuals but also escape from local optima.

To apply CCEA in the detection of code smells, the Boussaa *et al.* designed two sub-populations. The first one generates detection rules after taking knowledge of code smells' examples, as the Fig.1 shows. The second one generates artificial code-smells based on the notion of deviation from well-designed code fragments.

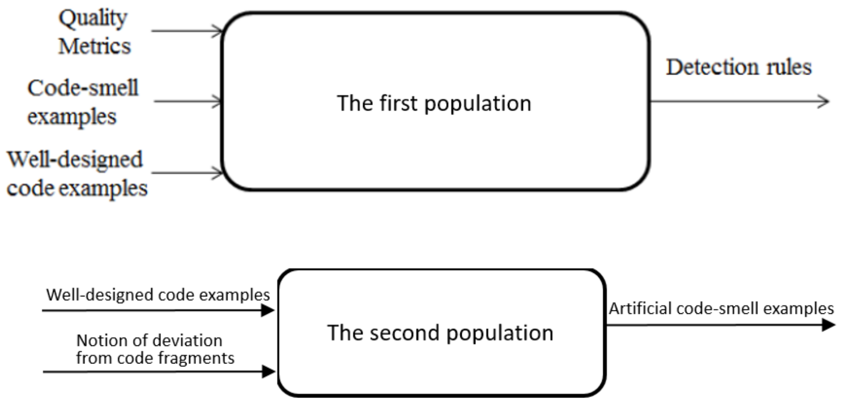

Quality Metrics
Code-smell examples
Well-designed code examples
The first population
Detection rules
Well-designed code examples
Notion of deviation from code fragments
The second population
Artificial code-smell examples

Fig. 1. The input and output of the two populations

The measurements for the two populations show the thought of the competition and coevolution. For the first (detection rules) population, the detection rules solutions are evaluated based on the coverage of the base of code-smell examples and the coverage of generated "artificial" code-smells by the second population. For the second (artificial code smell) population, the process performs not only maximize the distance between generated code-smell examples and reference code example, but also minimize the number of generated examples that are not detected by the first population. This design can firstly represent each population with traditional quality metrics, and it also implement the interaction between the two populations. The overall process of CCEA is showed in Fig 2. The evolutions of the two populations are parallel, and each population has individual ways of cross-over and mutation. The loop will not stop until the results of the two population meet the stopping criterion.

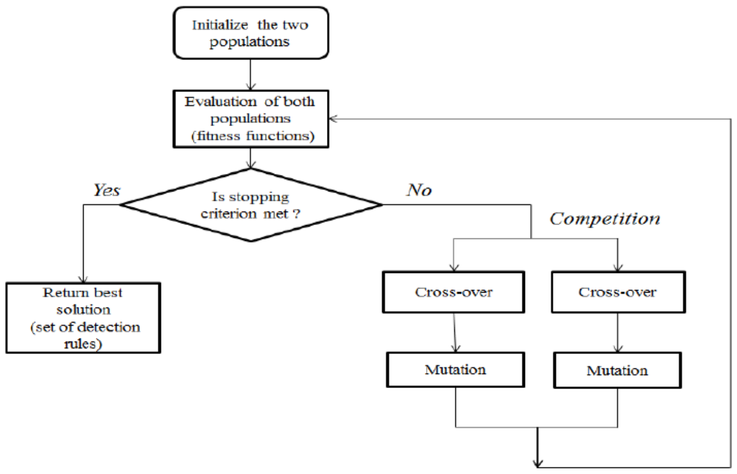

Initialize the two populations
Evaluation of both populations (fitness functions)
Is stopping criterion met ?
Yes
No
Competition
Return best solution (set of detection rules)
Cross-over
Cross-over
Mutation
Mutation

Fig. 2. Competitive co-evolution algorithm for code-smell detection

2.2 Solution representations

For the first population, the representation is a forest of AND-OR trees. Each tree is composed of terminals and functions. Terminals in a tree are represented as leaf-node, showing the metrics and the corresponding values. Functions are represented as internal-node, containing two logic operators AND and OR. One of the representations that show the bad smells is like Fig.3. In the example, BLOB is found in designs where one large class monopolizes the behaviour of a system (or part of it), and the other classes primarily encapsulate data. Spaghetti Code(SC) is a code with a complex and tangled control structure.

For the second population, the code elements in code-smell example are represented as sets of predicates. Such as the sequence of predicates CGAAMPPM means the class(C) contains a generalization link(G) with two attributes(A) and two methods(M), and the first method has two parameters(P).

3 Code smells detection in other papers

3.1 Manual detection as benchmark

To explore the efficiency of detecting code smells manually, Schumacher *et al.* chose three computer science students as candidates to participant the study containing two projects [8]. Each of them was given 90 minutes for each project to identify God Class. The result showed that the process of finding specific code smells was not too challenging for candidates, so they can detect the code smells manually in an effective fashion. The result of manual code-smells detection.

Similarly, the research conducted by Boussaa *et al.* also used the code smells detected by human as a basis to evaluate the performance of the competitive

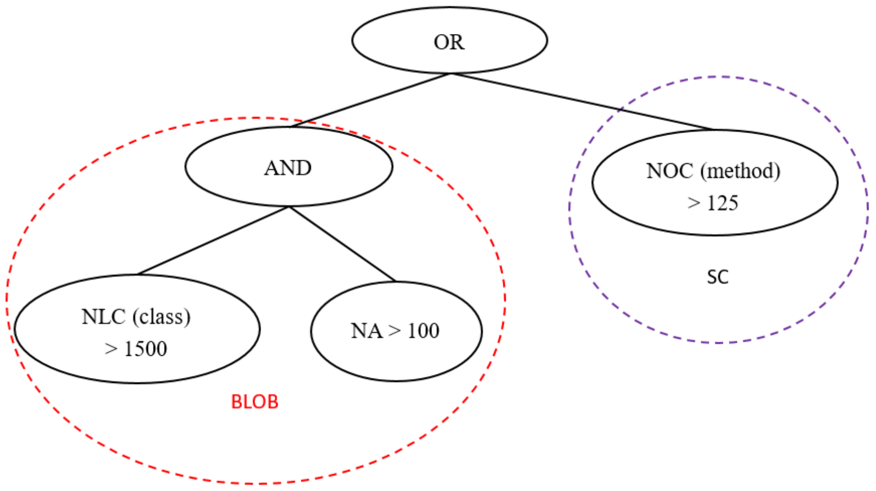


Fig. 3. The representation of the first population.NLC: Number of Lines of Code; NA: Number of Attributes; NOC: Number of Children.

co-evolution algorithm. The difference is that the Boussaa *et al.*'s data stem from others' researches.

3.2 Detection method

The algorithm of automated detection is to choose quality metrics and set thresholds for them firstly in [8]. In detailed, God classes have high complexity, low cohesion and extensive access to the data of foreign classes. To implement these features, the authors set the conditions to detect code smells as showed in Fig.4. The detection strategy is a logical composition of appropriate code metrics and corresponding thresholds that automatically detects design flaws in an application.

In comparison, the algorithm of Boussaa *et al.* can detect three types of code smells including BLOB, Spaghetti Code and Functional Decomposition. The data structure used in the research is also well-designed using sets of tree and string as described in Section 2.2. Nevertheless, the result keeps evolving, so that the final result is the most satisfactory. It may conclude that the efficiency of Boussaa *et al.*'s CCEA is better than the Schumacher *et al.*'s algorithm.

In [9], the three code-smells in Boussaa *et al.*'s research are also explored. The difference is that Kessentini *et al.* used single population evolution algorithm. However, to a same project, neither precision or recall get better results than Boussaa *et al.*'s. This shows the CCEA used in code-smell detection is promising in some degree.

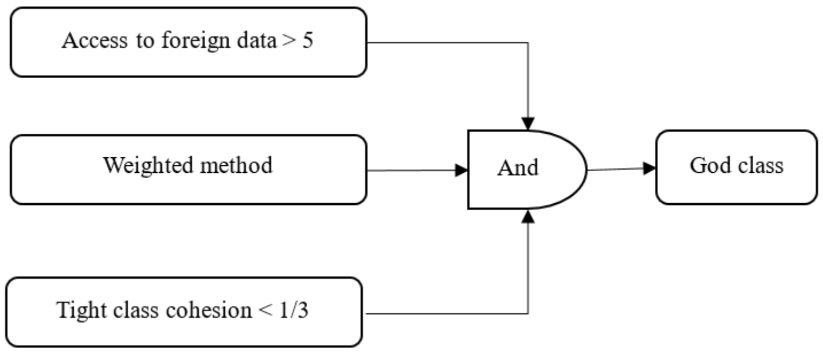


Fig. 4. Code smell detection strategy

4 Future work

4.1 Increase the types of code-smells that can be detected

In [1], only three code smells are detected. However, there are still many types of other code smells, such as Feature Envy, God Class, Shotgun Surgery, and so on. According to [10], there are some other tools to detect code-smells, and they perform better then Boussaa *et al.*'s research in variety of detection. For example, Stench Blossom can detect Data Clumps, Feature Envy, InstanceOf, Long Method, Large Class, Message Chain, Switch Statement and Typecast. The tool iPlasma can even detect up to eleven kinds of default code smells and four kinds of code smells by custom rule.

However, it is still acceptable in this essay that only detect three kinds of code smells since the importance is to explore the application of CCEA in code-smell detection. If possible, the algorithm can be applied in to code-smell tool. At that time, the variety of detection will be considered as a necessary element.

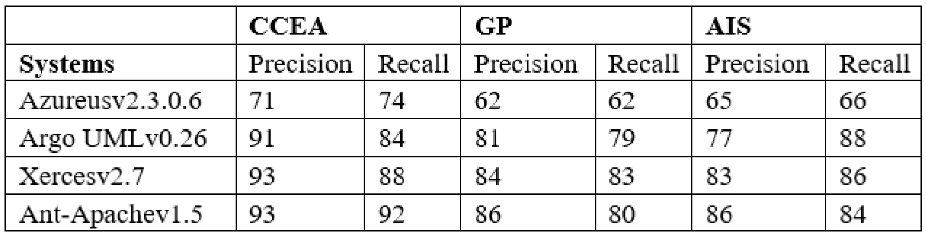
4.2 Re-factor the code-smells after detecting them

Detecting code smells automatically can replace some unnecessary cost of labor. If there is the automated refactor which can help programmers to remove those code smells, the tool or the method is handier for testing staff. Therefore, the next step of the essay can be code refactoring. There are some researches that have discussed about this topic. In [11], the researchers used NGSAII to find the trade-off from (1)minimizing the number of code smells, and (2)maximizing the use of development history while (3)preserving the construct semantics, then to give refactoring suggestions for the detected code-smells. The performance is also proven to be better than two other searching algorithms. In [12], the researches provide another idea to refactor the code smells parts. It is based on a chemical reaction optimization metaheuristic search, and prioritize the importance of code smells to find out the best solution to maximize the number of fixed riskiest code smells.

Overall, if the future work is related with the code-smells refactoring, the development history and code-smell priority can be considered. Although at current stage, the work of refactor cannot be replaced by automation, it is still promising if there is a tool can give programmers suggestions about it.

5 Critique

In [1], when the researchers tried to validate the application of their CCEA in code-smells detection, there is a comparison with other algorithms. The table shows the precision and recall rate when applying three different algorithms in four systems. Precision means the the fraction of correctly detected code-smells among the set of all detected code-smells, and recall shows how many code-smells have not been missed.



	CCEA		**GP**		**AIS**	
Systems	Precision	Recall	Precision	Recall	Precision	Recall
Azureusv2.3.0.6	71	74	62	62	65	66
Argo UMLv0.26	91	84	81	79	77	88
Xercesv2.7	93	88	84	83	83	86
Ant-Apachev1.5	93	92	86	80	86	84

Fig. 5. Comparison of CCEA, GA and AIS in code-smell detection

This kind of comparison can show the good performance of CCEA in code-smell detection, however, the other two algorithms GP and AIS are single-population approaches. It is unfair that compare a multi-population approach with single-population approach, since the result is apparent. Besides, the year that the paper was published is 2013, there were already many researches that dedicated to code-smell detection using multiple objectives. The reason that CCEA instead of other popular multi-objective algorithm is chosen to deal with the problem is not mentioned in the paper. What's more, the researches have the enough better resources to compare with, but they did not do that. Therefore, this comparison in validation is not so convincing and the reason of the choice about the algorithm is not fully discussed in the paper.

Another detail that deserved to be focused is the running time for the CCEA in code-smells detection. It costs 1h and 22 minutes for CCEA, 1h and 13 minutes for GP, and 1h and 4 minutes for AIS. From the table we can see that some of the increase in recall is little. It is hard to trade off between a higher precision or recall and higher efficiency in practice. The gap between theory and practice cannot be ignored. The exploration of this essay may be breakthrough, but it is possible that it does not contribute to the practice. If this CCEA algorithm is applied into a code-smell detection tool in the future, the trade-off should be considered. On the other hand, in theoretical research, the optimization of this trade-off may also be a new topic in searched based software engineer.

6 Conclusion

This essay summarizes the novelties of Boussaa *et al.*'s research including the utilization of CCEA in detecting code smells and representation of individuals in the evolution algorithm. When compared with related research, the solution of code-smells is widely using the manual detection data as benchmark, and there are many other algorithms are used in detecting code smells. The possible future works may be related to be able to detect more kinds of code smells and the refactoring after detection. However, the efficiency of the algorithm is compared with the algorithms that are not in the same level, making the paper less convincing. Nevertheless, the limited increase in the precision and recall is at the expense of running time.

References

1. Boussaa, M., Kessentini, W., Kessentini, M., Bechikh, S. and Chikha, S.B., 2013, August. Competitive coevolutionary code-smells detection. In *International Symposium on Search Based Software Engineering* (pp. 50-65). Springer, Berlin, Heidelberg.
2. Yamashita, A. and Moonen, L., 2013, October. Do developers care about code smells? an exploratory survey. In *Reverse Engineering (WCRE), 2013 20th Working Conference on* (pp. 242-251). IEEE.
3. Brown, W.H., Malveau, R.C., McCormick, H.W. and Mowbray, T.J., 1998. AntiPatterns: refactoring software, architectures, and projects in crisis. John Wiley & Sons, Inc..
4. Olbrich, S., Cruzes, D.S., Basili, V. and Zazworka, N., 2009, October. The evolution and impact of code smells: A case study of two open source systems. In *Empirical Software Engineering and Measurement*, 2009. ESEM 2009. 3rd International Symposium on (pp. 390-400). IEEE.
5. Khomh, F., Di Penta, M. and Gueheneuc, Y.G., 2009, October. An exploratory study of the impact of code smells on software change-proneness. In *Reverse Engineering*, 2009. WCRE'09. 16th Working Conference on (pp. 75-84). IEEE.
6. Fontana, F.A., Ferme, V., Zanoni, M. and Yamashita, A., 2015, May. Automatic metric thresholds derivation for code smell detection. In *Proceedings of the Sixth international workshop on emerging trends in software metrics* (pp. 44-53). IEEE Press.
7. Stanley, K.O. and Miikkulainen, R., 2004. Competitive coevolution through evolutionary complexification. *Journal of artificial intelligence research*, 21, pp.63-100.
8. Schumacher, J., Zazworka, N., Shull, F., Seaman, C. and Shaw, M., 2010, September. Building empirical support for automated code smell detection. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (p. 8). ACM.

9. Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M. and Ouni, A., 2011, June. Design defects detection and correction by example. In *2011 19th IEEE International Conference on Program Comprehension* (pp. 81-90). IEEE.
10. Fontana, F.A., Mariani, E., Mornioli, A., Sormani, R. and Tonello, A., 2011, March. An experience report on using code smells detection tools. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on* (pp. 450-457). IEEE.
11. Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K. and Hamdi, M.S., 2015. Improving multi-objective code-smells correction using development history. *Journal of Systems and Software*, 105, pp.18-39.
12. Ouni, A., Kessentini, M., Bechikh, S. and Sahraoui, H., 2015. Prioritizing code-smells correction tasks using chemical reaction optimization. *Software Quality Journal*, 23(2), pp.323-361.