Applied Artificial Optimization Algorithm in Design Flaws Detection

Sakorn Mekruksavanich
Department of Computer Engineering
School of Information and Communication Technology
University of Phayao, Phayao, Thailand
sakorn.me@up.ac.th

***Abstract*—The detection of design flaws is one of the most important aspects of software quality control, and the process should therefore be an integral part of the development and also the maintenance of software. It is possible to lower costs and extend the useful like cycle of the software simply by detecting design flaws at an early stage, and therefore attempts have been made to automate the procedures involved in detecting and fixing these flaws. One of the most common ways of detecting flaws is through the use of heuristic metrics which use predetermined standards as a means of analyzing the findings. While the approach can work successfully, the problem lies in the determination of those standards, or thresholds. This research study seeks to develop an enhanced method to improve threshold determination to be applied in flaw detection using metric-based designs. Accordingly, for each metric, an algorithm was employed for optimization of the contribution metrics to determine the threshold. The model produced threshold values which could then be adjusted to fit the requirements of the software data input. The findings from the experiments revealed that this approach could generate more appropriate thresholds for application in this context. In addition, the technique was relatively simple and could be used with different software programming languages, reducing the implementation time, and eliminating the need for the specialist expert support which would have traditionally been required in metric-based detection methods.**

***Index Terms*—Design Flaws, Detection, Refactoring, Artificial Optimization Algorithm**

I. Introduction

Software is critical for human life today, and the aim of software development is to create efficient and reliable software to ease the burdens in peoples lives, resulting in greater productivity. While software efficiency may not always seem as obvious as remarkable inventions such as airplanes, combustion engines, or the printing press, the impact can be just as significant, as it can affect every aspect of our lives today, encompassing, work, leisure, entertainment, health, and so forth. Although developers have been creating software now for around 75 years, there are still problems which arise during the development stage which lead to flaws. In particular, there are flaws which arise during the design of new software, which are known as design flaws [1].

Design flaws are those which incorporate problems within the rules or architecture of the software, and thus have a negative effect upon the overall structure quality of the design, leading to ongoing problems in the software development life cycle. Design flaws might not be described as wholly incorrect, and might not cause the failure of a program, but they can lead to slower operational use of a program and lead to a greater risk of errors arising in the future. Design flaws have long been seen as indicators of poor design and program selection, and in some cases they are the result of designers making decision which have sought to achieve simplicity but have involved sub-optimal choices [2].

One common technique used to detect and solve design flaws is the refactoring process [3]. It offers a means of checking the simple and meaningful design of the code without any need to change what the code actually does. The process involves making additions and changes to the source code through introducing reappearances, but the use of regular refactoring is not straightforward because factored codes are inclined towards decomposition. A number of forms can be produced through a factored/class, duplicate code, and other stages of mixed or discontinuous code. For all time periods, code modification without refactoring leads to degradation. This degradation of the source code is highly frustrating for the programmers since it takes up their time while lowering the overall useful lifetime of the software.

A code inspection approach performed manually can allow early detection of design flaws [4]. The source code must be meticulously examined, along with the documentation and design of the software, so that the examiner might uncover flaws based on his own long experience. Unfortunately, this method relies upon the skill of the examiner and takes a long time to complete. Furthermore, it cannot be scaled or repeated. One approach which could detect design flaws is declarative meta programming, since in such environments, it is possible to narrow the design flaw domain to allow easier detection with fewer metrics, or in the absence of metrics [5]. The approach is, however, more complex in terms of flaw identification. To simplify matters, heuristic metrics are suggested for the identification of software design flaws. The process of software metric detection applies a predetermined threshold or thresholds to support the interpretation of the process results [6], [7]. While metric-based flaw identification can effectively detect flaws, the approach relies upon the quality of the metrics and thresholds used, and therefore the detection of flaws based on metrics will be variable as it relies upon the threshold selection; there is always the potential for false negatives or false positives if the threshold is not finely

tuned to take into account these possibilities. The threshold must be set so as to obtain the right kind of information about the presence of any design flaws. The developer is therefore responsible for selecting the right approach to allow refactoring to remove any detected design flaws. The way in which software metrics are related to design flaws has been widely studied, given its importance.

This study proposes a new method to support the detection of design flaws using metrics. The concept involves finding software design flaws by combining an optimization algorithm with facts and rules metrics. The algorithm serves to improve the derived contribution facts and rules of the metrics and thus improve the thresholds for each of those metrics. The model can thus apply threshold values which have been selected specifically for requirements of the software data input. Furthermore, the algorithm is able to optimize flaws in order to verify the different software modules which comprise the software system. The technique does use refactoring in upgrading the software system performance in terms of the various programming languages used which may differ in their program structures and syntax. The performance of the model was evaluated using three different software datasets, with the experiments demonstrating that the new method could provide more appropriate thresholds for use in detecting design flaws through a metric-based technique. The simplicity and time-saving benefits of the new system were further advantages, along with the fact that no expertise was required for its implementation, representing an improvement over traditional approaches using metrics. This new concept would therefore be useful for software project managers and developers who would have access to reliable design flaw indicators for their software systems.

The structure of the paper takes the following form. Section 2 presents the concepts of related background. The new proposed methodology is discussed in detail in Section 3. Section 4 describes the experimental studies, presents the results, and compares the new method with other detection approaches. Finally, conclusions are drawn in Section 5.

II. Flaw Detection Background

A. Types of Design Flaws

When design flaws arise in the development of software they can cause significant problems in the subsequent design and maintenance of the software system. In such scenarios, Fowler provides a new concept, whereby different design flaw types are detected [8]. Subsequent approaches have found other types of design flaws in programs. The software development design flaws are very closely tied to the task of refactoring these flaws from the code in order to improve the operational reliability of that code. Where many developers are working to produce a single software system, they must identify any potential problems, which are referred to as bad smells, and determine whether these issues are likely to pose problems for the system with regard to functionality, maintenance, performance, or quality. This approach can enhance the search for laws and ease the refactoring process as flaws in the code are addressed. Within software systems, design flaws are often indicative of further problems. It is therefore necessary to perform refactoring and then re-check the system in order to improve its performance. One processing approach is provided by Kent Beck [3], and enhances software quality through design flaw refactoring. Typical design flaws and their detection in software systems using a metrics-based approach are described as follows:

- Large Class (LC): These classes possess large quantities of instance variables and many lines of code. Because they are large, they often exhibit software duplication issues. Related metrics to detect this flaw are described as follows:

$$LOC > 300 \; to \; 350, \; LM > 5$$
$$DIP > 3, \; coupling > 10$$

- Long Parameters (LP): Such parameters can be very challenging to detect within software systems, since the parameters exceed the limit. Related metrics to detect this flaw are described as follows:

$$NOP > 7, \; \sum nPOM = 148, \; \#P > AP$$
$$MinC = 88, \; AP = 3$$

- Dead Code (DC): In some cases, coding modules are designed but are not subsequently used within the software system. These unused modules are referred to as dead code. The problem is that such dead code increases the memory consumption and makes the system more expensive to operate. Related metrics to detect this flaw are described as follows:

$$UBoD = 24$$

- Lazy Class: These classes perform minimal work and the number of methods is null. Related metrics to detect this flaw are described as follows:

$$SoM = 0, \; LOC <= 300, \; \#M <= 2$$

- Duplicate Code: When code is duplicated, it is much more complicated to complete the updating process. It can also lead to longer detection times when the code appears more than once, and makes it harder to improve the system. Related metrics to detect this flaw are described as follows:

$$Total \; Number \; of \; Duplicate \; Code \; Block = 19$$

B. Biological Background

The behavior of animals in groups is often the result of their biological and sociological need to stay together, and hence they will herd, or swarm, or flock [9]. Each member of the group increases its chances of survival in this way because it is usually animals which are alone which will be picked off by predators. Accordingly, the flock, swarm, or herd can be characterized by its collective form of movement. In particular, social insects such as bees or ants will form

colonies which perform swarming behavior. Swarms are self-organizing, autonomous, and exhibit distributed functioning, governed by a communication system whereby each individual makes a contribution to the overall collective intelligence of the swarm. This is known as swarm intelligence [10].

Swarm intelligence is a subsection of artificial intelligence which examines he actions of individual components within a decentralized system. Decentralized, or multi agent systems, comprise physical or virtual components which must communicate and cooperate, sharing information in order to work together to achieve particular tasks within their particular environment. Models which use swarm intelligence in their design apply natural principles from the swarming phenomenon. The idea is not, however, to wholly replicate the natural system, but to take ideas from the phenomenon which might be applicable to the creation of the model [11].

C. Bee Colony Algorithm

In their natural environment, bees see food through exploration of their surroundings. They gather food which can be stored and used by their colleagues [12]. In the first step, a small number of bees scout the general area, then return and inform their colleagues about the location, amount, and type of food available. Whenever they find food in a location they have investigated they will dance in the hive to let the other bees know what they have food, and to encourage the others to follow them to the source. Any bee which wants to go to the food source will follow the scout bees to the suitable discovered flowers. The bee will then collect the nectar and bring it back to the hive for storage. There are three potential courses of action for the bee. The first is to give up on the food source and wait in the hive for further guidance. The second is to continue bringing food back from the food source alone. The third is to encourage other bees by dancing to inform them about the food source. At any given moment there may be several bees dancing having found sources, so it is not clear how the other bees decide which scout to follow. However, the extent to which bees can be recruited to go to the food source is always dependent upon the size and quality of the food source. This process is ongoing, as bees locate, gather, and store nectar.

The bee colony algorithm is designed to consist of three bee types: scouts, employed bees, and waiting bees [13]. The bees who leave their food sources will become scouts and search for food. The waiting bees wait for scouts to return and dance, then they will select a food source and become employed bees. All food sources are then processed by employed bees, and if they find new food sources, they will dance and encourage waiting bees to join them. When food sources are finished, they will be replaced by new sources discovered by scouts, thereby ensuring that the best food sources are discovered and used by the bees.

III. Proposed Methodology

In this study, the design flaw detection procedure relies upon mixing the facts and rules metrics with the Bee Colony Optimization algorithm so that design flaws can be identified. This optimization approach allows verification of the software modules so that any design flaws in the source code can be determined. The algorithm is able to improve both the metric thresholds and the derived contribution facts and rules of the metrics. The model threshold values are then selected to fit the requirements of the software data input. Application of the fact and rule metrics supports the processing of the source code and identification of the software modules in many different programming languages. The new method also makes use of refactoring approach to improve the software system performance. Different programming languages have different program structures and syntax, so the new approach works with these through the use of fact and rule processing, forming a link between the software module and the detection network in order to provide superior detection of design flaws. This technique is capable of working with C#, Java, and software systems involving refactoring in order to identify a range of design flaws.

A. Research Methodology

Figure 1 presents the workflow used in design flaw detection. This step-by-step process presents the inputs from the software system and is able to distinguish the modules from the larger systems. The input data makes use of the fact and rule programming which form a bridge connecting the software modules and the design flaw processing system using the metric. In the process described, step 4 allows the analysis of the source code flaws, which can be verified at the next stage. The verified data then serve as the output for the user seeking to detect the design flaws. Refactoring is then suggested to resolve design flaws and improve the software system performance. Limits are introduced in testing so as to evaluate the performance of the novel method in comparison to current approaches in order to encourage superior performance. The steps used are explained as follows:

- Step 1: Source code modules are input to detect the design flaws. First of all, the source code modules are selected (Java, C#, or other types of object-oriented software) to detect the flaws in the software system.
- Step 2: Source code modules are verified and divided into attributes, classes, and methods with the relationships classified on the basis of an abstract syntax tree of determined fact and rules. The required metrics are also calculated in the form of LOC (line-of-code), used variables.
- Step 3: In the stage the facts and rules of metrics are applied for the modules. The divided fraction of the source code is used in determining the facts and rules required to carry out the detection of design flaws.
- Step 4: The source code is measured and analyzed with regard to the specified software metrics. The metrics used in detecting the design flaws are formulated on the basis of the determined facts and rules. This stage is where comparisons are made between the detection metrics and the determined primary threshold values.

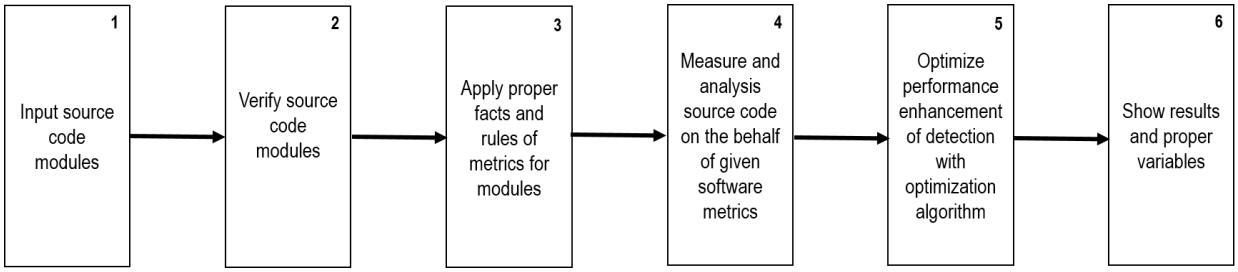

Fig. 1: Processes of the proposed research work

- Step 5: The optimization algorithm is used for performance enhancement. In this case, the artificial bee colony algorithm is employed. Details of this algorithm are shown in Algorithm 1. This stage generates the final threshold values.
- Step 6: The results and variables are presented at this stage after the detection stage is complete. The system performance can be analyzed and the results presented to the end user.

IV. Results and Discussion

Datasets of software design flaws were collected from previous research to conduct experiments. Online data entry software was also used to test the applicability of the proposed methodology. Table I lists the three datasets used.

The performance of detection approaches for two-class (i.e. fault-prone and nonfault-prone) problems was commonly evaluated using the pattern set in the confusion matrix, shown in Table II.

In Table II, f_{ij} expresses that the number of actual class = i is classified into class = j, where $i, j \in 0, 1$. Detection accuracy, a commonly used detection performance measure, was used to quantitatively evaluate the detection approaches. This measure was derived from the confusion matrix as follows:

$$\frac{f_{00} + f_{11}}{f_{00} + f_{01} + f_{10} + f_{11}} \tag{1}$$

The detection accuracy rates for the three systems were 90%, 92%, and 93.5% according to Figure 2, in order of the dataset numbers for the technique in which Large Class flaws were detected. In Figure 3, the results for accuracy of detection were 92.5%, 90%, and 90% order of the dataset numbers for the technique in which Lazy Class flaws were detected. In order to make comparisons with other common methods, the results of this experiments were assessed against findings obtained using the standard metric-based technique. The results are indicated in Figures 2 and 3.

V. Conclusion and Recommendations

This study put forward a novel approach to improve design flaw detection using metrics. The technique involved an artificial bee colony algorithm which was used to assist in selecting appropriate metrics to support the process of flaw detection. The study also proposed the evaluation function

Require: Initialization: Read problem data, parameter values (B and NC), and stopping criterion.
repeat
(1) Assign a(n) to empty solution to each bee.
(2)
for $i = 0$ to $i < NC$ **do**
do *forward pass*
(a)
for $b = 0$ to $b < B$ and $s = 0$ to $s < f(NC)$ **do**
(i) Evaluate possible moves
(ii) Choose one move using the roulette wheel
$b++$ and $s++$
end for
do *backward pass*
(b)
for $b = 0$ to $b < B$ **do**
Evaluate the (partial/complete) solution of bee b
$b++$
end for
(c)
for $b = 0$ to $b < B$ **do**
Loyalty decision for bee b
$b++$
end for
(d)
for $b = 0$ to $b < B$ **do**
if b is uncommitted **then**
choose a recruiter by the roulette wheel
end if
$b++$
end for
$i++$
end for
(3) Evaluate all solutions and find the best one.
Update x_{best} and f(x_{best})
until Stopping criterion is not satisfied
return (x_{best}, f(x_{best}))

Algorithm 1: Proposed artificial bee colony algorithm

TABLE I: Experimental datasets

Number	Datasets
1	CommonCLI v1.0
2	JUNIT v1.3.6
3	GANTTPROJECT v1.10.2

TABLE II: The confusion matrix

		Predicted class	
		Class = 0	Class = 1
Actual class	Class = 0	f_{00}	f_{01}
	Class = 1	f_{10}	f_{11}

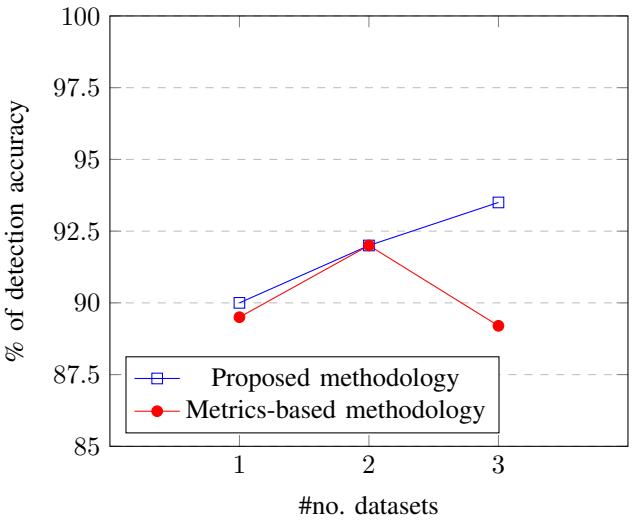

Fig. 2: The detection accuracy of the proposed approach and the metric-based approach on *Large Class* flaw detection

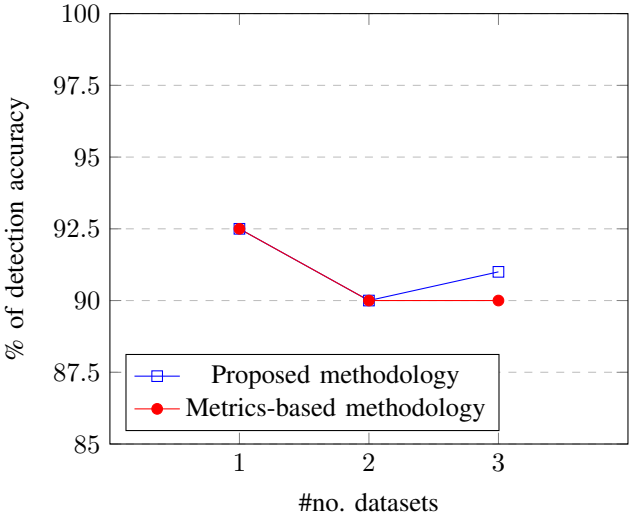

Fig. 3: The detection accuracy of the proposed approach and the metric-based approach on *Lazy Class* flaw detection

used to calculate the contribution threshold for each of the metrics in accordance with the requirements of the software data input. The experimental results showed that the proposed model established more suitable thresholds for metric-based design flaw detection.

It may be possible in future studies to examine the potential of other algorithms to be used for optimization to improve the detection accuracy. Genetic algorithms may be suitable for this purpose. Furthermore, it may also be possible to improve the algorithm performance through increasing the search for software design flaws since many other parameters can be applied in the process of design flaw detection.

ACKNOWLEDGMENT

This research was supported in part by the School of Information and Communication Technology, University of Phayao, Thailand.

REFERENCES

[1] T. Tourwe and T. Mens, "Identifying refactoring opportunities using logic meta programming," in *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, March 2003, pp. 91–100.

[2] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *2010 Seventh International Conference on the Quality of Information and Communications Technology*, Sept 2010, pp. 106–115.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999.

[4] D. A. Wheeler, B. Brykczynski, and R. N. Meeson, Jr., Eds., *Software Inspection: An Industry Best Practice for Defect Detection and Removal*, 1st ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996.

[5] S. Mekruksavanich and P. Muenchaisri, "Using declarative meta programming for design flaws detection in object-oriented software," in *2009 International Conference on Signal Processing Systems*, May 2009, pp. 502–507.

[6] S. Mekruksavanich, "Identifying behavioral design flaws in evolving object-oriented software using an ontology-based approach," in *2017 13th International Conference on Signal-Image Technology Internet-Based Systems (SITIS)*, Dec 2017, pp. 424–429.

[7] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Trans. Softw. Eng.*, vol. 36, pp. 20–36, Jan. 2010.

[8] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells," in *2007 IEEE International Conference on Software Maintenance*, Oct 2007, pp. 519–520.

[9] E. Bonabeau, M. Dorigo, and G. Theraulaz, *From Natural to Artificial Swarm Intelligence*. New York, NY, USA: Oxford University Press, Inc., 1999.

[10] J. Kennedy, R. C. Eberhart, and Y. Shi, "chapter one - models and concepts of life and intelligence," in *Swarm Intelligence*, ser. The Morgan Kaufmann Series in Artificial Intelligence, J. Kennedy, R. C. Eberhart, and Y. Shi, Eds. San Francisco: Morgan Kaufmann, 2001, pp. 3 – 34.

[11] G. Beni and J. Wang, "Swarm intelligence in cellular robotic systems," in *Robots and Biological Systems: Towards a New Bionics?*, P. Dario, G. Sandini, and P. Aebischer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 703–712.

[12] T. D. Seeley, S. Camazine, and J. Sneyd, "Collective decision-making in honey bees: how colonies choose among nectar sources," *Behavioral Ecology and Sociobiology*, vol. 28, no. 4, pp. 277–290, Apr 1991.

[13] D. Teodorović, *Bee Colony Optimization (BCO)*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 39–60.