Alexandria Engineering Journal (2018) 57, 1631–1642

HOSTED BY

Alexandria University

Alexandria Engineering Journal

www.elsevier.com/locate/aej
www.sciencedirect.com

REVIEW

Model level code smell detection using EGAPSO based on similarity measures

G. Saranya [a], H. Khanna Nehemiah [a,*], A. Kannan [b], V. Nithya [c]

[a] Ramanujan Computing Centre, Anna University, Chennai 600025, Tamil Nadu, India
[b] Department of Information Science & Technology, Anna University, Chennai 600025, Tamil Nadu, India
[c] Department of Computer Science & Engineering, Anna University, Chennai 600025, Tamil Nadu, India

Received 27 September 2016; revised 6 June 2017; accepted 10 July 2017
Available online 7 August 2017

KEYWORDS

Software maintenance;
Code smell;
Software metrics;
Search based software engineering;
Euclidean distance;
Open source software

Abstract Software maintenance is an essential part of any software that finds its use in the day-to-day activities of any organization. During the maintenance phase bugs detected must be corrected and the software must evolve with respect to changing requirements without ripple effects. Software maintenance is a cumbersome process if code smells exist in the software. The impact of poor design is code smells. In code smells detection, majority of the existing approaches are rule based, where a rule represents the combination of metrics and threshold. In rule based approach, defining the rules that detect the code smells are time consuming because identifying the correct threshold value is a tedious task, which can be fixed only through trial and error method. To address this issue, in this work Euclidean distance based Genetic Algorithm and Particle Swarm Optimization (EGAPSO) is used. Instead of relying on threshold value, this approach detects all code smells based on similarity between the system under study and the set of defect examples, where the former is the initial model and the latter is the base example. The approach is tested on the open source projects, namely Gantt Project and Log4j for identifying the five code smells namely Blob, Functional Decomposition, Spaghetti Code, Data Class and Feature Envy. Finally, the approach is compared with code smell detection using Genetic Algorithm (GA), DEtection and CORrection (DECOR), Parallel Evolutionary Algorithm (PEA) and Multi-Objective Genetic Programming (MOGP). The result of EGAPSO proves to be effective when compared to other code smell detection approaches.

* Corresponding author.
E-mail address: nehemiah@annauniv.edu (H. Khanna Nehemiah).
Peer review under responsibility of Faculty of Engineering, Alexandria University.

http://dx.doi.org/10.1016/j.aej.2017.07.006

Contents

1. Introduction

Software evolution and maintenance involve high costs of the development process, particularly as systems become larger and more complex. A usual concern that makes software maintenance and evolution difficult is the existence of structural design problems. These design problems are otherwise known as code smells. The majority of the existing approaches of code smell detection is in the code level [1–5].

The quality of the software can be improved if code smells are identified in the model level of the software. Detecting the code smells in the model level is very difficult as all the metrics are not supported in model level [6]. A model maintenance activity includes adding new functionalities, incorporating modifications if needed, detecting poor design fragments and correcting them [4].

In model level maintenance, majority of the existing approaches is rule based [6,7]. All code smells cannot be detected in rule based approach because finding the right threshold value of the metrics is a tedious task. For example, a rule that detects large classes involves metrics related to the class size. Even though the metrics of large classes are easily measurable, finding the right threshold value is significant. To be precise, a class considered as a large class in a program can be an average class in any other program. The objective of the work is to identify the code smells in the model level. This paper focuses on detecting the code smell in the UML class diagram, which does not rely on the definition of rules. Thus the approach does not require a specification of the metrics to use their related threshold values for code smell detection.

In this work, code smell detection is based on, estimating the similarities between the initial model and the base example. To this end, EGAPSO [8] is used for finding the code smells in the model level. Three open source softwares are used in this work. Two of them are considered as the initial models and the third one is considered as the base example. Metrics are computed from both initial model and base example. Based on the metrics, the similarity between the class of the initial model (CIM) and the class of the base example (CBE) is determined. Finally, using EGAPSO, all the code smells are detected. The EGAPSO is compared with the code smell detection approaches namely genetic algorithm (GA), DEtection and CORrection (DÉCOR), Parallel Evolutionary Algorithm (PEA) and Multi-Objective Genetic Programming (MOGP).

The remainder of this paper is organized as follows: Section 2 describes a literature review of code smell detection. Section 3 presents the relevant background material for the presented work. Section 4 describes the approach for code smell detection using EGAPSO; empirical study definition and design are discussed in Section 5. In Section 6 experimental results and its analysis are discussed; Section 7 presents the threats that could affect the validity of the code smell detection using EGAPSO is discussed. Finally, conclusion and scope for future work are discussed in Section 8.

2. Literature review

Smell detection has been addressed in the literature from different perspectives. The approach for code smell detection can be classified into three broad categories. The first category is search based approach, the second category is non-search based approach and the third category is the model based approach.

2.1. Search based techniques

Seng et al. presented an approach that uses genetic algorithm for suggesting the list of refactoring that do not change the external behaviour of the software [9]. The researchers used a genetic algorithm for finding the set of refactoring that has to be applied in the software. They used a fitness function that depends on an existing set of object oriented metrics which is evaluated on the open source software JHotDraw. Using this approach, more than 90% of refactoring operations were found.

Harman et al. used Pareto optimality to improve the search-based refactoring, which combines metrics and provides numerous sequences of optimal refactoring [10]. Several runs of this search-based refactoring system produced Pareto front, whose values represent Pareto optimal sequences of refactoring. The drawback is, the Pareto optimal search explicitly focused only on suggesting the refactoring, not on detecting the defects in the code.

Jensen et al. proposed an approach REMODEL based on genetic programming and a set of software metrics to make out the most appropriate set of refactoring that introduce design patterns to the software [11]. The researchers used Quality Model for Object Oriented Design (QMOOD) [12] a suite of OO metrics to improve the quality of the software design. The approach REMODEL was validated using the Repository for Model Driven Development (REMODD) a web based software system. The limitation of the approach REMODEL is that the design defects are not detected explicitly as the main focus is on suggesting the suitable refactoring to improve the quality of the software.

Kessentini et al. used genetic programming for an automatic approach to define rules for code smell detection. The rules are the combination of metrics and threshold values [5]. The researchers compared their approach to Harmony Search (HS), Particle Swarm Optimization (PSO) and Simulated Annealing (SA) to find a near optimal set of detection rules. When compared to other approaches, the researchers found a better result of an average of 75% of known code smells.

Ouni et al. had developed an approach for code smell detection and correction [13]. The approach developed by Ouni et al. has two objectives, one is code smell detection and the other is the correction of code smell. The correction of code smells focuses on suggesting the suitable refactoring operations. Detection is based on the technique of Genetic Programming whereas Non-dominated Sorting Genetic Algorithm (NSGA-II) is used for the correction of code smells. This approach had been evaluated on six open source projects, namely Gantt project, Quick UML, Azureus, Log4J, ArgoUML and Xerces. The approach attained 94% precision and 97% recall for Ganttproject, 91% precision and 91% recall for Xerces, 87% precision and 93% recall for ArgoUML, 86% precision and 93% recall for QuickUML, 73% precision and 87% recall for Azureus and 79% precision and 83% recall for Log4J.

Ghannem et al. proposed an approach to detect code smell based on Genetic Algorithm [14]. The approach detects code smells in the design stage of the project with the help of well-known design defects. The approach was evaluated on four open source program, namely GanttProject, ArgoUML, Log4J and Xerces which targets to detect the three design defects namely blob, functional decomposition and data class. This approach attained 95% precision and 75% recall.

Shizhe Fu et al. proposed a novel approach to detect code smells through Evolutionary data mining [29]. They used association rules mined from the change history of software systems along with the heuristics algorithms to detect code smells. Experimentation was performed on 5 open source projects namely Eclipse, Junit, Guava, Closure Compiler and Maven to detect 3 code smells such as duplicate code, shotgun surgery and divergent change. On the basis of their experiments and results they concluded that their approach achieves precision between 50% and 100%, recalls between 60% and 100% and F-measure between 58% and 100%. The researchers have compared their approach with SonarQube which detects more duplicate code but achieves low F-measure in the detection of duplicate code. They have also compared their approach with DECOR, JDeodorant and DCPP to show that their approach outperforms other approaches.

Kessentini et al. (2014) proposed parallel evolutionary algorithm (PEA) for the detection of code smell namely blob, functional decomposition, spaghetti code, feature envy, data class, long parameter list, lazy class and shotgun surgery [33]. In PEA the researchers used genetic programming and genetic algorithm in parallel to generate the code smell detection rules and the detectors from well-designed code examples respectively. The approach has been evaluated on nine open source projects namely JFreechart, Ganttproject, ApacheAnt V1.5.2, ApacheAnt V1.7.0, Nutch, Log4J, Lucene, Xerces-J and Rhino. The PEA used for code smell detection approach is compared with random search (RS) algorithm and two single population based approaches and two code smell detection techniques that are not based on meta-heuristics search. In the nine open source software, the approach PEA attained an average precision and recall of 85% on eight different types of code smell detection.

Mansoor et al. introduced a multi-objective approach to generate rules for the detection of code smell [34]. The researchers used Multi-Objective Genetic Programming (MOGP) to find the best combination of metrics that maximize the detection of code smell examples and minimize the detection of well-designed code examples. They achieved 86% precision and 91% recall.

The major limitation of using these search based techniques is that, most of them can be applied only for refactoring the software system. The results produced by search based techniques are complementary when compared with other code analysis techniques. Some of the search based techniques require generation of rules for identifying the code smells. In this paper, a search based technique EGAPSO is used, which overcomes all the limitations of the existing approaches. It does not require the generation of rules for defect detection; instead it uses defect examples to identify the code smells.

2.2. Non-search based techniques

Marinescu proposed a metric-based approach to detect code smells with detection strategies [4]. The metric based approach defines a set of rules for detecting the defects in the object oriented design at method, class or subsystem level. The strategies capture deviations from good design principles and heuristics. The detection strategies have been evaluated on multiple industrial case studies of size ranging from 700 KLOC to 2000 KLOC. For all the strategies, the accuracy rate is over 50%, and the average accuracy is over 67%. However, there are some limitations in their detection strategies, such as they have no justification for choosing the metrics, thresholds and the combination of the metrics and threshold.

Moha et al. introduced an approach called DÉCOR that includes all the necessary steps for the specification and detection of code and design smells [15]. This approach describes the symptoms of the defects in abstract rule language. These descriptions are then mapped to detection algorithms. To overcome the threshold, this method uses some heuristics to approximate some notions. The approach DÉCOR had detected the code smells namely Blob, Swiss army knife, Functional decomposition, Spaghetti code and their underlying fifteen code smells. The approach is tested on the open source project Xerces-J and obtained an overall precision of 60.5% and recall of 100%. However, they have not compared approach DÉCOR with the other existing approaches related to smell detection.

Budi et al. proposed an approach to detect the defects in layered oriented architecture based on stereotypes [16]. In this layered approach, the classes are automatically separated into three groups boundary, entity and control to detect the design flaws associated with each stereotype. The limitation of this approach is that it can detect only the defects associated with the stereotype and not the real design defects.

Palomba et al. have proposed an approach Historical Information for Smell deTection (HIST) to detect the code smells in the source code [17]. The approach HIST detected five different code smells, namely Shotgun Surgery, Parallel Inheritance, Divergent Change, Feature Envy and Blob by exploiting change history information mined from versioning systems. For example, classes that change frequently are considered as a blob. The researchers evaluated the approach HIST to eight projects such as Apache Ant, Apache Tomcat, jEdit and five Android applications and obtained an overall precision of 71% and overall recall of 81%. Vimala et al. proposed a refactoring framework based on game theory to restructure the PL/SQL code [18]. The similarity measure has been used to compute the payoff matrix and refactoring is based on game theory.

Fontana et al. proposed an approach based on Machine learning techniques [30]. The focus of the approach was based on the detection of code smells namely God class, Data class, Long method, Feature Envy. They considered 76 systems for the analysis and experimented on 6 different machine learning algorithms. Experimentation was performed on J48, Random forest, Naive Bayes, Jrip, SMO, LibSVM classifiers through k-fold cross validation in different configurations to find the best one. They have also produced a learning curve to determine which algorithm learns more quickly. On the basis of the experiment and results, they concluded that J48, Random Forest, Jrip and SMO have accuracy values greater than 90% for all data sets and an average they have best performances. Naive Bayes has slightly lower performances on Data class and Feature Envy than on the other two smells. LibSVM performances are lower than the others. The limitation of this approach is that they have also used default parameters instead of optimized parameters. Better results can be obtained with optimized setting of parameters.

Hamid et al. formed a comparative study on two tools namely JDeodorant and incode to detect two bad smells such as God class and Feature Envy [31]. The incode uses metrics based approach and JDeodorant uses the agglomerative clustering technique to detect God class smell. Feature envy smells detected by incode tool are static methods whereas JDeodorant detects non-static methods. They have also extracted some suggestions on the bases of variations found in the results of both detection tools. They concluded that besides development of new tools current tools also need to be revised. They have differentiated static and non-static methods with the use of two tools and are unable to find the reason why inCode does not detect non-static methods detected by Jdeodorant.

Rasool et al. discussed analysis of code smell mining techniques. They illustrated generic code smell detection process steps to diagnose symptoms in the design [32]. They presented a comparison of seven approaches namely manual approach, symptoms based, metric based, probabilistic approach, search-based approach and cooperative approach based on their key features. They highlighted some of the issues in the code smell detection tools such as the integration of tool with other tools is not considered by most authors. They also pointed out the missing feature of tools in the detection of design and architectural smells. Through the review of 46 selected primary papers, the authors have not found any language-independent code smell detection technique that requires the attention of the researchers. They presented a summary of techniques/tools, and stated that not a single technique/tool is capable of detecting all 22 code smells of Fowler. Experiments reveal the differences in results computed by different tools. The majority reasons in the disparity of results are due to the different threshold values on the number of parameters used by these tools. Though an up-to-date review on the state of the art techniques and tools used for mining code smells are presented, they are limited by covering the area of code smells on detection rather than covering all aspects of code smells.

To conclude, the major drawback of existing non-search based techniques is mainly its difficulty in defining the threshold values for metrics which are used in identifying the defects.

2.3. Model based approaches

Model maintenance is defined as the different modifications made to the model in-order to improve its quality, to add new functionalities to the model, to easily detect the badly designed fragments in the model, to correct the defects in the model and to modify the model. When arbitrary changes are made to a model, it is quite likely that its behaviour consistency is not preserved. To preserve the behaviour, the maintainers apply the refactoring operations. The goal of the refactoring operation was to improve the structure of the model and preserve the behavioural properties. Van Der Straeten et al.

explored the relationship between behaviour consistencies of model refinements and behaviour preservation of model refactoring [6]. Moreover, the researchers developed a plug-in which is an extension of the Poseidon plug-in to show the practical use of relation model refinement and refactoring. This plug-in detects automatically, whether model refinements and model refactoring are behaviourally consistent. It has been tested in ATM software developed by Prof. Russell Bjork from Gordon University. The main drawback of this approach is that the experiment has been carried out only on small projects.

Van Kempen et al. described the refactoring for the UML model [19]. The researchers used the class diagram of Software Architecture Analysis Tool (SAAT) as a case study for refactoring. From the class diagram of SAAT, the researchers calculated the metrics. Using the metrics, they identified the design defects and the corresponding refactoring operation is applied. The researchers also used Communicating Sequential Process (CSP) to prove the behaviour equivalence of the SAAT after refactoring. The drawback is that this process becomes difficult when it is operated over more complex designs.

Zhang et al. proposed a rule based approach that describes how the model refactoring is applied [20]. The researchers developed a model transformation engine named as C-SAW, which provides the general refactoring tool for manipulating models, and then it has been integrated with the refactoring browser to enable the automation of refactoring methods. Ivan Porres defined model refactoring as a rule-based model transformation which is similar to Zhang et al. But, the refactoring tool of Porres does not support automated refactoring and domain-specific model refactoring [21].

Most of the model based approaches are based either on the rules or on graph transformation and focus on refactoring operations. However, complete code smell detection was not considered in model level.

The existing search based techniques in general can be applied only to refactoring the software system. Also they have limitation in defining the rules and ensuring the correctness of the rules in identifying the code smells. The existing non-search based technique presents complexity in finding the correct threshold values for metrics which are used to detect the defects. The existing model based approaches do not focus in code smell detection. Hence, to overcome these issues, an EGAPSO based approach is proposed to detect code smell in the model level. The proposed EGAPSO based approach aims in overcoming these limitations of rule based approach using a similarity search based optimization search technique rather than relying on a simple threshold value. Hence the proposed approach yields better results than the other approaches.

Table 1 Metrics description.

Metrics	Description of the metrics
NA	The total number of attributes per class
NPvA	The total number of private attributes per class
NpbA	The total number of public attributes per class
NprotA	The total number of protected attributes per class
NM	The total number of methods per class
NPvM	The total number of Private methods per class
NPbM	The total number of Public methods per class
NprotM	The total number of Protected methods per class
NAss	The total number of Associations
Ngen	The total number of Generalization relationships

3. Background and problem statement

This section provides the necessary background material used for the detection of code smells. The definition of code smells, the metrics used for code smell detection and EGAPSO are discussed below:

3.1. Code smell

Code smells are first defined by Fowler and Beck [22]. They defined code smells as the symptoms of code and design problems [22]. In this paper, code smells detection that occurs in the model level, especially in the class diagram is focused, thereby improving the model quality. The following five code smells were considered and evaluated in this paper.

- **Blob:** Blob is found in designs where one large class monopolizes the behaviour of a system. It is a large class that has many fields and methods with low cohesion and almost the class does not have parent class and children. It is a class which implements different responsibilities and has large size.
- **Functional Decomposition (FD):** FD occurs when a class is designed with the intent of performing a single function. FD is found in a class, where inheritance and polymorphism are poorly used. This is found in class diagrams produced by non experienced object-oriented developers.
- **Data Class (DC):** It is a class that has only data. It does not have any processing on the data. The only methods that are defined by this class are the getters and the setters.
- **Feature Envy:** Feature envy is a classic smell that occurs when a method get fields of another method in some other class than the one it is actually implemented in. It is often characterized by a large number of dependencies with the envied class, so it reduces the cohesion and increases the coupling of the class.
- **Spaghetti Code:** Spaghetti Code is a classic code smell that occurs when the code does not use appropriate structuring mechanisms. Spaghetti Code prevents the use of object oriented mechanisms, namely polymorphism and inheritance. Classes affected by this smell, are characterized by complex methods, with no parameters and interaction between them using instance variables.

3.2. Software metrics

Software metrics provide some useful information that helps in assessing the quality of the software [23]. It can also be used for detecting the similarities between the software systems [24]. In this work, ten metrics are considered, where, all these metrics are related to the class entity in the class diagram. Among the ten metrics, NA, NPvA, NpbA, NprotA, NM, NPvM, NPbM, NprotM metrics give information about the number of methods and attributes in the class diagram, while NAss and Ngen give the relationship between classes. The

metrics that are considered in this work with expressiveness and usefulness are listed in Table 1 [25].

3.3. Euclidean distance based genetic algorithm and particle swarm optimization (EGAPSO)

The EGAPSO [8] is a population based heuristic search optimization algorithm developed by Kim and Park in 2006. The EGAPSO is a hybrid approach of Particle Swarm Optimization (PSO) [26] and Genetic Algorithm (GA) [27] which is based on Euclidean distance. This algorithm was used to tune the proportional integral derivative (PID) controller in a steam temperature control system of the thermal power plant, industrial system of chemical process and also in biomedical process [8]. EGAPSO works as follows: In EGAPSO, each particle is considered as individuals, whereas a group of particles is called as a swarm. Each particle in the swarm has its own position and its velocity. Initially the particles are placed at random positions in the search space, and the velocity of the particle is represented as zero. For each generation the position of the particle and its velocity in EGAPSO can be updated using the formula given in the Eqs. (1) and (2).

$$v_i(t+1) \leftarrow \omega v_i(t) + c_1 r_1(pbest_i(t) - x_i(t)) + c_2 r_2(gbest_i(t) - x_i(t)) \quad (1)$$

$$x_i(t+1) \leftarrow x_i(t) + v_i \quad (2)$$

In the Eqs. (1) and (2), $x_i(t)$ and $x_i(t+1)$ represent the position of the particle at time (t) and time (t + 1), respectively. $v_i(t)$ is the velocity of the particle at time (t), $pbest_i(t)$ is the best position of the particle found so far, $gbest_i(t)$ is the global best position of the particles, c_1 and c_2 are the acceleration coefficients that influence the best position of the particles, r_1 and r_2 are the random variables and ω represents the inertia weight of the particles.

During the search process, the position of a particle is guided by two factors, namely local best (*pbest*) and global best (*gbest*). The best visited position for the particle by itself is local best (*pbest*) and it arrives at the best position obtained so far by any particle in the neighbourhood is global best (*gbest*). During iterations particle converges to its local point and it moves according to the following iterative mathematical models:

Gantt Dialog Info
Abstract DOM Parser
Blob

Fig. 1 Individual representation.

$$pbest_i(t+1) = \begin{cases} pbest_i(t) & if\ f(pbest_i(t)) \leqslant f(x_i(t+1)) \\ x_i(t+1) & if\ f(pbest_i(t)) > f(x_i(t+1)) \end{cases} \quad (3)$$

$$gbest(t+1) = \min\{f(y), f(gbest_i(t))\} \quad (4)$$

where $y \in \{pbest_0(t), pbest_1(t) \ldots\ldots\ldots\ldots pbest_s(t)\}$.

After computing the local best and global best of the particles, Euclidean distance is computed between each particles *pbest* and the *gbest* value using Eq. (5).

$$Euclideandistance_{(j,i)} = \alpha.f(p_j) - f(p_i)/||p_j - p_i|| \quad (5)$$

$$\alpha = ||s||/f(p_g) - f(p_w) \quad (6)$$

$$S = \sqrt{\sum_{i=1}^{p}(x_{i1} - x_{i2})^2} \quad (7)$$

In the above Eqs. (5)–(7) $f(p_j)$ represents the *pbest* value of the j^{th} particle, $f(p_i)$ represents the *pbest* value of the i^{th} particle, 'α' represents the scaling factor which can be computed from the formula given in the Eq. (6). In Eq. (6) $f(p_g)$ represents the global best of the particle and $f(p_w)$ represents the global worst of the particle. 'S' represents the size of the search space which can be computed using the formula given in Eq. (7). In Eq. (7) p represents the size of the swarm, x_{i1} and x_{i2} represent the position of the particle. Based on the value of the Euclidean distance, the particles at a minimum distance with the *gbest* are chosen for single pair crossover and random mutation operation. The obtained offsprings from the random mutation operation are added to the existing particles for the next generations. The process is repeated until the optimal solution is returned.

4. A search based approach for detecting code smells

In this paper, EGAPSO is used for identifying the code smells present in the given model specifically from class diagrams. In the EGAPSO, the knowledge from the class of the base example (CBE) and software metrics are used to detect the code smell in the class of the initial model (CIM). In the following subsections, the paper describes how the code smell detection is encoded using EGAPSO algorithm.

4.1. Individual representation

An individual is represented as a block. A block is a triplet which consists of three parts. It includes CIM, CBE and the code smell detected from the base example. An example of an individual is represented below in Fig. 1. The set of individuals is combined to form the initial population.

4.2. Fitness function

The fitness function denotes the quality of the individuals. It indicates the similarity between the model under analysis

Table 2 Classes from the initial model, base example and their metric values.

Classes	NA	NPvA	NProtA	NPbA	NM	NPvM	NProtM	NPbM	NAss	Ngen
AbstractChart Implementation	10	10	0	0	45	2	8	35	0	1
AbstractDOM Parser	51	6	45	0	45	0	7	8	0	0

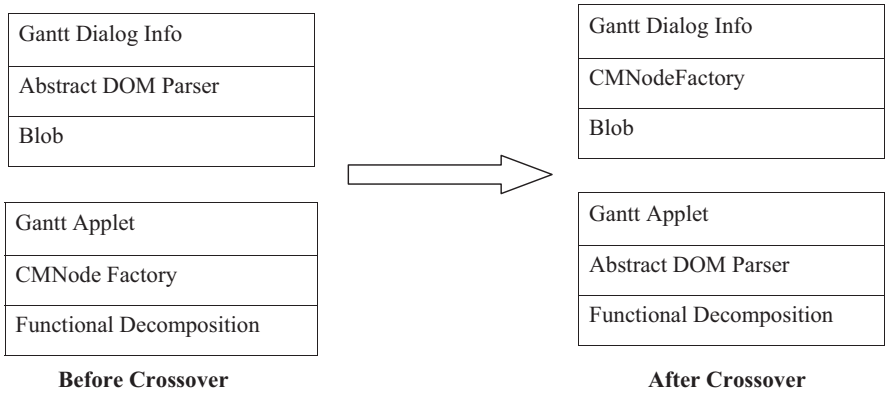

Fig. 2 Crossover operation on individuals.

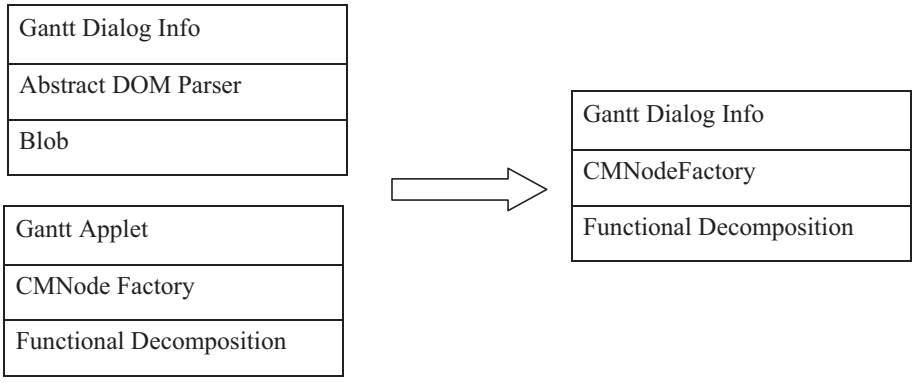

Fig. 3 Mutation of two individuals.

(initial model) and the existing model (base example) to infer the code smell that must be corrected. In this work, the similarity between *CIM* and *CBE* is calculated. If the similarity value is high, then the code smell in the base example exists in the initial model. The formula for computing the similarity between *CIM* and *CBE* is given in Eqs. (8) and (9).

$$Similarity(CIM, CBE) = \frac{1}{m}\sum_{i=1}^{m} Sim(CIM_i, CBE_i) \tag{8}$$

$$Sim(CIM_i, CBE_i) = \begin{cases} 1 & if\ CIM_i = CBE_i \\ 0 & if\ (CIM_i = 0 and CBE \neq 0) \\ & or\ (CIM_i \neq 0\ and\ CBE_i = 0) \\ CIM_i/CBE_i & if\ CIM_i < CBE_i \\ CBE_i/CIM_i & if\ CBE_I < CIM_i \end{cases} \tag{9}$$

where m is the number of metrics and $Sim(CIM_i, CBE_i)$ is given in Eq. (9). CIM_i denotes the metric value of i^{th} class in the initial model and CBE_i denotes the metric value of i^{th} class in the base example. Then the *fitnessfunction* is calculated using Eq. (10).

$$fitnessfunction = f_1 + f_2/2 \tag{10}$$

The value of the *fitnessfunction* is normalized in the range [0,1]. Using similarity between the classes, the function f_1 is calculated using Eq. (11) and the function f_2 ensures the completeness of an individual using Eq. (12).

$$f_1 = \frac{1}{n}\sum_{j=1}^{n} Similarity(CIM_{Bj}, CBE_{Bj}) \tag{11}$$

where n is the number of blocks and CIM_{Bj} and CBE_{Bj} are the classes composing the first two parts of the j^{th} block.

$$f_2 = IndividualSize/MaximumIndividualSize \tag{12}$$

4.2.1. Illustration of fitness function

To illustrate the computation of fitness function of two individuals, two classes are taken, one from initial model (Abstract Chart Implementation) and the other class from base example (Abstract DOM Parser). First the metric values are calculated from the two classes which are represented in Table 2.

$f_1 = 1/1[1/10[(10/51+6/10+0+1+1+0+7/8+8/35+1+0)]]$

$f_2 = 1$

$finalfitness = 0.74$

After computing the fitness function, the individuals are sorted based on their fitness values.

4.3. Crossover of the particles

In this work, single point crossover is applied. Two parent individuals for crossover are selected based on the minimum Euclidean distance of the particles. From the selected individuals the CBE block is interchanged to produce two new off-

springs. The crossover operation on individuals is shown in Fig. 2.

4.4. Mutation of the particles

The individuals for mutation are selected based on minimum Euclidean distance. From the selected individuals, the CBE and Defects from base example of one individual are replaced with the CBE and Defects from base example of other individual. In this paper, the random mutation operation is applied. Fig. 3 shows the effect of the mutation operation that the class Abstract DOM parser (base example) of the design defect Blob is replaced with the class CMNode Factory (base example) of the design defect Functional decomposition.

4.5. Adaptation of EGAPSO for code smell detection

```
Input: CIM, Set of Quality metrics, Set of code smells from CBE
Output: Best solution: Code smells in CIM
1: I:= set of (CIM, CBE, code smells in CBE) /*I represents an individual*/
2: P:= Set of (I)                /*P represents the size of the swarm*/
3: initial population (P, Max_size)
4: Repeat
5:    For all I in P do    /*Compute the fitness for every particle*/
6:      Compute fitness f1(I) using Eq. (11)
7:      Compute fitness f2(I) using Eq. (12)
8:      fitnessfunction(I):= {f1(I)+f2(I)}/2
9:    End for
10:   For all I in P do    /*finds the best-solution*/
11:     best_fitnessfunction (I):= min(fitnessfunction(I))
12:     Best_Solution:= best_fitnessfunction (I)
13:   End for
14:   While the stop criteria is not satisfied do
15:     Update velocity of the individuals using Eq. (1)
16:     Move the individuals using Eq. (2)
17:     Calculate the Euclidean distance using Eq. (5)
18:     Perform single point crossover
19:     Perform random mutation operation
20:   End
21:     P := generate_new_ population (P)   /*new population generation*/
22:     it = it + 1;
23: Until it = max_it or fitnessfunction (best_solution);
24:   Return best_solution
25: End
```

The pseudo code for the EGAPSO algorithm for code smell detection in model level is given above. EGAPSO takes input as, CIM, a set of software metrics and set of code smells from CBE as controlling parameters, it returns the set of code smells detected as output. Lines 1–3 construct an initial population, which is a set of individuals that stand for possible solutions, The function *set of (CIM, CBE, code smells in CBE)* returns an individual *I* which is represented as a block. The individual representation is discussed in Section 4.1. The function *Set of (I)* returns the size of the swarm. In lines 4–9 the search space is explored. During the iteration, the quality of each individual is evaluated using the *fitnessfunction* which is defined as an average of two functions f_1 and f_2, where f_1 computes the similarities between the CIM and CBE of each block composing the individual and *f2* computes the ratio of the individual size by the maximum individual size. In line 10–13 the individual having the *best_fitnessfunction (I)* is saved. In line 21, a new population (P + 1) of individuals is generated from the current population. In line 14–20, the position and velocity of the individuals are updated from the new population and the Euclidean distance is calculated. Based on the Euclidean distance, the individuals with minimum distance are selected. From the selected individuals, crossover and mutation operation are performed. This produces the population for the next generation. The algorithm terminates after the given maximum number of generations or the best *fitnessfunction* value (line 23) and returns the best_solution (line 24).

5. Empirical study definition and design

The ***goal of the study*** is to examine the EGAPSO approach, in detecting the code smells, namely blob, data class, spaghetti code, functional decomposition and feature envy in software systems. The ***quality focus*** is the detection accuracy on code smells when compared to the GA approach, while the ***perspective*** is of other researchers, who want to evaluate the effectiveness of the approach in identifying code smells to build better recommenders for developers. The ***context*** of the study consists of three open source projects, namely Gantt-Project, Log4j and Xerces-J. Gantt-Project is a cross-platform tool for project scheduling. Log4j is a Java-based logging utility. Finally, Xerces-J is a family of software packages for parsing XML. Gantt-Project and Log4j are used as initial model and Xerces-J is used as base example.

5.1. Research question, data analysis and metrics

This work aims at addressing the following two research questions:

RQ1: What type of code smells does it detect correctly?
RQ2: How does the proposed approach perform compared to the existing code smell detection approaches?

To answer RQ1, the authors investigated the types of code smell that are detected by this EGAPSO approach. To answer RQ2, this paper compares the EGAPSO approach to the existing approaches using evaluation parameters, namely precision, recall, average number of defects detected and $F_{measure}$ [28].

Table 3 Code smell present in GanttProject.

Code smells	Gantt project
Number of classes	245
Number of blobs	10
Number of data class	10
Number of functional decomposition	17
Number of feature envy	11
Number of spaghetti code	16

Table 4 Code smell present in Log4j.

Code smells	Log4j
Number of classes	227
Number of blobs	3
Number of data class	5
Number of functional decomposition	11
Number of feature envy	2
Number of spaghetti code	8

Table 5 Parameter setting.

Algorithms	Parameters	Values
EGAPSO	Population size	100
	Number of generations	500
GA	Population size	100
	Number of generations	1000

5.2. Precision

Precision denotes the fraction of correctly detected code smells over the detected code smells. From the value of the precision, one can infer the probability that the detected code is correct.

$$Precision = \{(\text{Re}leventCodeSmells) \cap (DetectedCodeSmells)\}/(DetectedCodeSmells) \quad (13)$$

5.3. Recall

Recall denotes the fraction of correctly detected code smells among the set of manually detected code smells to find out how many code smells have not been missed. From the value of the recall, one can infer the probability that an expected code smell is detected.

$$\text{Re}call = \{(\text{Re}leventCodeSmells) \cap (DetectedCodeSmells)\}/(\text{Re}leventCodeSmells) \quad (14)$$

5.4. Average number of defects detected

Average Number of Defects Detected (ANDD) is the fraction of the defects detected by the approach over the number of defects that are actually present.

$$Average\ Number\ of\ Defects\ Detected = Number\ of\ defects\ \text{detected}/Number\ of\ defects\ actually\ present \quad (15)$$

5.5. $F_{Measure}$

$F_{measure}$ is defined as the harmonic mean of precision and recall.

Table 6 Precision and recall values for GA and EGAPSO.

Open source software	Defects	GA						EGAPSO					
		Precision (%)	Recall (%)	ANDD (%)	$F_{measure}$ (%)	Specificity (%)	AUC (%)	Precision (%)	Recall (%)	ANDD (%)	$F_{measure}$ (%)	Specificity (%)	AUC (%)
Gantt project	Blob	100	90	90	94	100	95	100	95	100	100	100	98
	Functional decomposition	100	47	47	64	100	73	100	76	76	86	100	88
	Feature envy	100	81	82	90	100	90	100	82	82	100	100	91
	Data class	90	88	100	90	99	93	100	98	100	90	100	99
	Spaghetti code	90	62	68	74	99	81	100	81	81	94	100	91
Log4j	Blob	100	97	100	100	100	99	100	97	100	100	100	99
	Functional decomposition	100	63	64	77	100	82	100	81	81	89	100	91
	Feature envy	50	47	100	50	99	73	100	97	100	100	100	99
	Data class	75	60	80	66	99	80	80	77	100	80	99	88
	Spaghetti code	80	50	62	61	99	75	83	62	75	71	99	81

Table 7 Comparison of precision values for the proposed code smell detection approach using EGAPSO with the other state of art approaches.

Open source software	Defects	Precision of EGAPSO (%)	Precision of GA Ghannem et al. [14] (%)	Precision of DECOR Moha et al. [15] (%)	Precision of PEA Kessentini et al. [33] (%)	Precision of MOGP Mansoor et al. [34] (%)
Gantt Project	Blob	100	100	90	93	83
	Functional decomposition	100	88	26.7	88	77
Log4j	Blob	100	100	100	82	–
	Functional decomposition	100	100	54.5	93	–

$$F_{measure} = 2 * |\text{Precision} * recall / \text{Precision} + recall|\% \quad (16)$$

5.6. Experimental setup

For the code smell detection approach, three open source softwares are used namely GanttProject, Log4j and Xerces-J. In this approach GanttProject, Log4j are taken as initial model and Xerces-J is taken as the base example. First metrics are computed from the initial model using UML generator plug-in in Netbeans. Then the metrics and code smells are detected from the base example. Finally, the code smells in the initial model are detected using EGAPSO using the metrics detected from both the initial model and base example and code smells detected from the base example, and compared with the code smells detected using GA. Tables 3 and 4 provide the information about these open source software including number of classes and the number of code smell detected. The parameter setting of EGAPSO and GA is given in Table 5.

6. Evaluation of results

A preliminary evaluation of EGAPSO approach was performed on a well-designed open-source system, namely GanttProject, Log4j and Xerces-J. Here GanttProject, Log4j are taken as initial model and Xerces-J as the base example. For identifying the code smells, first the metrics are computed from the initial model (Gantt Project and Log4j) and base example (Xerces-J), then the code smells are detected from the base example using infusion tool. Then using EGAPSO the code smells are identified from the open source project. The evaluation is aimed at investigating the detection accuracy of the code smells using EGAPSO approach. This approach can identify the code smell (Blob, Functional decomposition, Feature envy, Data class and Spaghetti Code) more accurately when compared with that of genetic algorithm.

To test the accuracy of EGAPSO approach, the measures, namely precision, recall, average number of defects detected (ANDD), $F_{measure}$ and area under the ROC curve (AUC) have been calculated. The computed values of precision, recall, ANDD, $F_{measure}$ and AUC for genetic algorithm and EGAPSO are tabulated in Table 6.

6.1. Result of RQ1

To answer this question, this work uses the precision values to prove that all the code smell detected by the proposed approach were true. To further analyse the effectiveness of the proposed approach, the precision of the proposed EGAPSO approach is compared to the other state of the art approaches such as GA (Ghannem et al., 2016), DECOR [15], PEA [33] and MOGP [34] using precision values mentioned in Table 7. The precision values for the state of the art approaches in Table 7 are obtained directly without implementation from these works.

6.2. Result of RQ2

The comparison of the approach EGAPSO over the existing GA in terms of precision, recall, average number of defects detected and $F_{measure}$ values reveals the efficiency and accuracy of the EGAPSO over the GA.

Using the code smell from base example and set of metrics computed from both initial model and base example, the EGAPSO approach identifies not only the code smell, but also improves the cohesion of the system under study.

7. Threats to validity

External validity refers to the generalizability of the findings. In this work, the experiments are performed on the open source softwares which are described in Tables 3 and 4. However, it cannot be asserted that the results can be generalized to industrial applications, other programming languages and to other practitioners. Future work of this study is necessary to confirm the generalizability of the findings.

Construct validity is concerned with the relationship between theory and what is observed. Most of what is measured in this experiment are standard metrics such as precision, recall, average number of defects detected and $F_{measure}$ that are widely accepted for quality of code-smell detection solutions. In future, this approach will be compared using different meta-heuristics search algorithms. Another threat to construct validity arises because, although five types of code-smells are considered, we did not evaluate the detection of other types of code-smells. In future, it is planned to evaluate the performance of this approach to detect other types of code smells.

8. Conclusion and future work

Using the EGAPSO approach, five code smells namely blob, functional decomposition, feature envy, data class and spaghetti code have been detected from the open source software

namely Gantt Project and Log4j. The fitness function used in EGAPSO aims in maximizing the similarity between the model under analysis and the model in the base example which there by maximizing the number of detected defects. The main advantage of the approach is that this method can identify the code smells present in the model level of the open sources; whereas most of the existing approaches cannot be used for identifying code smells present in the model level of the software. The algorithms EGAPSO and GA are evaluated and the results revealed the completeness and correctness of EGAPSO over the existing GA. The approach has also increased the average precision, recall, average number of defects detected and $F_{measure}$ over the existing approach. Moreover, the precision of EGAPSO is compared with the other state of the art approaches namely GA, DECOR, PEA and MOGP. In future, this approach can be applied on other open source projects. Correction of code smell can also be considered in future.

References

[1] H. AliKacem, H. Sahraoui, Détection D'anomalies Utilisant Un Langage De Description De Règle De Qualité, in: Actes Du 12e Colloque LMO, 2006, pp. 185–200.

[2] K. Erni, C. Lewerentz, Applying Design-Metrics to Object-Oriented Frameworks, in: *Proceedings of the 3rd International Software Metrics Symposium*, IEEE, 1996, pp. 64–74.

[3] G. El Boussaidi, H. Mili, Understanding design patterns—what is the problem?., Software: Practice Experience 42 (12) (2012) 1495–1529.

[4] R. Marinescu, Detection strategies: metrics-based rules for detecting design flaws, in: Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM), 2004, pp. 350–359.

[5] M. Kessentini, H. Sahraoui, M. Boukadoum, M. Wimmer, Search-based design defects detection by example, in: Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the Joint European Conferences on Theory and Practice of Software, Springer, (SaarbrüCken, Germany), 2011, pp. 401–415.

[6] R. Van Der Straeten, V. Jonckers, T. Mens, A formal approach to model refactoring and model refinement, Software Syst. Modeling 6 (2007) 139–162.

[7] B.D. Bois, S. Demeyer, J. Verelst, Refactoring Improving Coupling and Cohesion of Existing Code, in: Proceedings of the 11th Working Conference on Reverse Engineering, IEEE Computer Society, 2004, pp. 144–151.

[8] H. Kim, J. Park, Improvement of genetic algorithm using PSO and Euclidean data distance, Int. J. Inf. Technol. 12 (3) (2006) 142–148.

[9] O. Seng, J. Stammel, D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, in: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, ACM, 2006, pp. 1909–1916.

[10] M. Harman, L. Tratt, Pareto optimal search based refactoring at the design level, in: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, ACM, London, England, 2007, pp. 1106–1113.

[11] A.C. Jensen, B.H.C Cheng, On the use of genetic programming for automated refactoring and the introduction of design patterns, in: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation. (Portland, Oregon, USA), 2010, pp. 1341–1348.

[12] J. Bansiya, C.G. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Trans. Software Eng. 28 (1) (2002).

[13] A. Ouni, M. Kessentini, H. Sahraoui, M. Boukadoum, Maintainability defects detection and correction: a multi-objective approach, Automated Software Engineering (ASE) 20 (2013) 47–79.

[14] A. Ghannem, G. El Boussaidi, M. Kessentini, On the use of design defect examples to detect model refactoring opportunities, Software Quality J. 24 (4) (2016) 947–965.

[15] N. Moha, Y.G. Gueheneuc, L. Duchien, A.F. Le Meur, DECOR: a method for the specification and detection of code and design smells, Software Eng., IEEE Trans. 36 (1) (2010) 20–36.

[16] A. Budi, D. Lucia Lo, L. Jiang, S. Wang, Automated detection of likely design flaws in N-tier architectures, Software Eng. Knowledge Eng. (SEKE) (2011) 613–618.

[17] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, Detecting bad smells in source code using change history information, ASE (2014) 268–278.

[18] S. Vimala, H. Khanna Nehemiah, R.S. Bhuvaneswaran, G. Saranya, A. Kannan, Applying game theory to restructure PL/SQL code, Int. J. Soft Comput. 7 (2012) 264–270.

[19] M. Van, Kempen, M. Chaudron, D. Kourie, B. Andrew, Towards proving preservation of behaviour of refactoring of UML models, in: Proceedings of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries (SAICSIT), 2005, pp. 252–259.

[20] J. Zhang, Y. Lin, J. Gray, Generic and domain-specific model refactoring using a model transformation engine, In Model-driven Software Development—Research and Practice Software Engineering (2005) 199–217.

[21] Ivan Porres, Model refactorings as rule-based update transformations, in: Proceedings of UML Conference, Volume 2863 of Lecture Notes in Computer Science, San Francisco, Springer-Verlag, 2003, pp. 159–167.

[22] M. Fowler, K. Beck, J. Brant, Refactoring: improving the design of existing code, in: Proceeding of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods, Springer, 1999, p. 256.

[23] N.E. Fenton, S.L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, PWS Publishing Co., Boston, MA, 1998.

[24] A. Ben Fadhel, M. Kessentini, P. Langer, M. Wimmer, Search based detection of high level model changes, ICSM (2012) 212–221.

[25] M. Genero, M. Piattini, C. Calero, Empirical validation of class diagram metrics, Proc. Int. Symp. Empirical Software Eng. (2002) 195–203.

[26] J. Kennedy, R. Eberhart, Particle swarm optimization, Proc. IEEE Int. Conf. Neural Networks 4 (1995) 1942–1948.

[27] D.E. Goldberg, Genetic Algorithms in Search, Addison-Wesley, Optimization and Machine Learning, 1989.

[28] R. Baeza-Yates, B. Ribeiro-Neto, in: Modern Information Retrieval, ACM Press, New York, 1999, p. 463.

[29] Shizhe Fu, B. Shen, Code bad smell detection through evolutionary data mining, in: Proceedings of ACM/IEEE the International Symposium on Empirical Software Engineering and Measurement (ESEM), 2015, pp. 1–9.

[30] F.A. Fontana, M. Zanoni, A. Marino, M.V. Mantyla, Code smell detection: towards a machine learning-based approach, in: Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM), 2013, pp. 396–399.

[31] A. Hamid, M. Ilyas, M. Hummayun, A. Nawaz, A comparative study on code smell detection tools, Int. J. Adv. Sci. Technol. 60 (2013) 25–32.

[32] G. Rasool, Z. Arshad, A review of code smell mining techniques, J. Software: Evolution Process 27 (11) (2015) 867–895.

[33] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, A. Ouni, A cooperative parallel search-based software engineering approach for code-smells detection, IEEE Trans. Software Eng. 40 (9) (2014) 841–861.

[34] U. Mansoor, M. Kessentini, B.R. Maxim, K. Deb, Multi-objective code-smells detection using good and bad design examples', Software Quality J. (2016) 1–24.