A Genetic Algorithm-based Approach for Design-level Class Decomposition

Bayu Priyambadha[1], Nobuya Takahashi[2], Tetsuro Katayama[3]
Faculty of Computer Science, Universitas Brawijaya, Malang, Jawa Timur, Indonesia[1]
Faculty of Engineering, University of Miyazaki, Miyazaki, Japan[2,3]

***Abstract*—Software is always changed to accommodate environmental changes to preserve its existence. While changes happen to the software, the internal structure tends to decline in quality. The refactoring process is worth running to preserve the internal structure of the software. The decomposition process is a suitable refactoring process for Blob smell in class. It tried to split up the class based on the context in order to arrange it based on each responsibility. The previous approach has been implemented but still leaves problems. The optimum arrangement of class cannot be achieved using the previous approach. The genetic algorithm provides the search mechanism to find the optimum state based on the criterion stated at the beginning of the process. This paper presents the use of genetic algorithms to solve the design-level class decomposition problem. The paper explained several points, including the conversion from class to the chromosome construct, the fitness function calculation, selection, crossover, and mutation. The results show that the use of a genetic algorithm was able to solve the previous problems. The genetic algorithm can solve the local optimum problem from the previous approach. The increment of the fitness function of the study case proves it.**

Keywords—Genetic algorithm; refactoring; class decomposition; blob smell; software internal quality

I. Introduction

Software will always be changed due to the changes in its environment. This statement is also stated in Lehman's law about software evolution [1], [2]. During the operation period, the environment somehow changes. This environment encompasses various components, including hardware, operating systems, libraries, frameworks, databases, and external services. These changes can significantly impact how software functions and interacts with its surroundings. Software environment changes are inevitable, and developers need to proactively manage and adapt their applications to ensure continued functionality, security, and compatibility as qualified software in evolving environments.

It is essential to develop software that is flexible and adaptable to changes to mitigate environmental changes. The easiness of adaptation or changes in software, as feedback of environment changes, is called software maintainability. Good software maintainability can be achieved by maintaining the software's internal structure quality. Adapting to environmental changes without concern for the software's internal structure quality will lead to difficulties in future changes. Compared to poorly structured software, software with well-designed structures will make it easier to adapt to changes.

The refactoring process alters the software's internal structure without changing the external behavior [3]. Implementing this process is worthwhile to prevent software from becoming obsolete. In Refactoring, the alteration of software structure is done based on the existing problem or declining area in terms of quality. Then, those areas are called "smell."

In the previous research, we proposed a refactoring process to solve the Blob smell in the class diagram [4]. Blob smell is one anomaly condition that is expressed in class that showed in class that monopolizes a lot of processes. The main problem with this smell is that a lot of responsibility is allocated to a single class. Based on the clean architecture theory [5], one class must only have one responsibility (Single Responsibility Principle). That is why blob smell can be solved by using class decomposition to split the responsibility and allocate it to several classes.

Knowing the blob smell and decomposing it at the class diagram level has been proposed in previous publications [6], [7]. The threshold-based hierarchical agglomerative clustering was implemented to perform class decomposition to solve blob smell in class at the level of the class diagram. This approach looked promising due to the result showing the significance of the impact of the decomposition process on software maintainability [8].

The class decomposition mostly uses the clustering process. To evaluate the result of decomposition mostly based on the cluster quality produced by proposed approaches. In the previous study, two variables were used to measure cluster quality: silhouette coefficient and class usability. Class usability is important because, in the case of class decomposition, the usability of clustering results must be considered. Based on the previous result, problems remain, especially related to class usability. In some cases, the cluster result is considered unable to be implemented as a class because there is no class interface, or all elements are not accessible except the class itself. It is making the class instantiate selfish objects.

This study used one clustering method to decompose class, as in the previous experiment. The method is threshold-based hierarchical agglomerative clustering by considering the semantic and static similarity between class elements [6], [7]. The research results show that the approach increased the quality measurement metrics called the Maintainability Index (MI). The experiment compared the original and the class after decomposing using the approach. On average, all data are increased by the MI. Overall, the results show that the approach sounds promising in the future to prevent software's internal structure quality [8]. But, there is a problem that needs attention to be solved. Several classes still contain the problems after the decomposition process. The issues that still exist in the last result experiment are:

- The unusable (class with no method) class can still be produced event by the evaluation process. This condition makes the MI value low.
- It is a fact that the decompose candidate class only consists of one public method, which makes it challenging to find the optimum class usability after the decomposition process.

Besides those problems, getting a higher evaluation value for the problematics class is possible than the result of the previous approach. This condition immerges the assumption that the previous approach tends to trap the local optimum result.

Based on the result, this research will study the problems in the previous research. The optimum cluster composition is the main purpose of the class decomposition research, which considers several factors, including cluster compactness and class usability.

Genetic algorithms (GA) are optimization algorithms inspired by the process of natural selection and evolution. This method is commonly used for solving optimization and search problems by mimicking the principles of biological evolution. In general, the GA consists of several processes: initialization, selection, crossover (recombination), mutation, evaluation, and termination. GA is well-suited for global optimization problems, where the goal is to find the best solution from a large and complex solution space. The ability of GA to explore diverse solutions makes them effective in finding global optima [9]. Therefore, this research will use GA to carry out the class decomposition process to resolve the remaining problems. GA will perform the clustering process and arrange the optimum decomposed class to produce the best composition.

The rest of this paper will be arranged as follows. Section two will describe the implementation of the genetic algorithm to do the clustering process and explain the proposed genetic algorithm in the class decomposition process. Section three explains the experiment scenario, dataset, and environment. Section four describes the result of the experiment and discussion as an interpretation of the result. Section five is about the conclusion of this research experiment.

II. Genetic Algorithm For Class Decomposition

A. Genetic Algorithm Research

GA can also be employed for clustering processes in this research experiment. Genetic algorithms are versatile optimization techniques that can be adapted for various problem domains, and clustering is one such domain. The application of genetic algorithms to clustering problems involves representing clusters as solutions and optimizing the clustering configuration based on certain criteria [10], [11].

In software engineering, genetic algorithms are used to optimize the software engineering process. GA can be applied to automate the generation of test cases, particularly for complex software systems.

By evolving sets of test cases, GA can effectively explore the behavior of the system under various conditions, helping to uncover bugs and vulnerabilities [12], [13], [14].

GA can also be utilized to automate the process of refactoring software components by representing different refactoring transformations as genes within chromosomes. Each chromosome represents a potential refactoring solution, consisting of a sequence of refactoring to be applied to the target codebase. The fitness of each solution is evaluated based on predefined criteria such as improved code readability, reduced complexity, enhanced modularity, and adherence to design principles [11].

The other usage of GA in the refactoring process is conducted in this research. GA will be used to do one of the refactoring processes called class decomposition to solve the blob smell. The following sections will describe the use of GA in this research experiment.

B. Initialization and Chromosome Construction

The population in the genetic algorithm is the collection of genes (solution) that will be randomly generated as an initial process. The genes or solutions in the population are based on the case that will be solved in this experiment. The gene or chromosome representation in the case of class diagram decomposition is described as follows. One gene or solution represents all elements in the class and the cluster where each element is assigned. The cluster number (second row of genes) will automatically be generated at the beginning of the initialization process. The number of genes or chromosomes in one population depends on the initial definition. Fig. 1 show the illustration of chromosomes in this case.

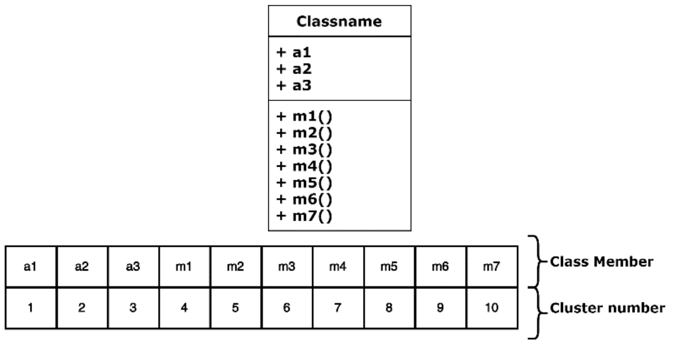


Fig. 1. Chromosome construction.

C. Fitness Function and Selection Process

In this approach, the parent candidates are selected by using the linear ranking selection process. Linear ranking selection is used in genetic algorithms to select an individual for reproduction based on their relative fitness ranks rather than their actual fitness values. This method aims to strike a balance between favoring high-fitness individuals and maintaining diversity in the population.

All individuals in the population are ranked based on their fitness values. The ranking is done in descending order depending on the goal of this experiment. This experiment aims to find the best class cluster construction based on the cluster compactness and class usability, which are calculated as eval value [6]. Higher eval values show better cluster construction for the class. The eval value is calculated as follows.

$$Eval = a.s(i) + b.CUsability \quad (1)$$

where $s(i)$ is the silhouette coefficient value, and $CUsability$ is the class usability value. a and b are the weights for each considered variable. $s(i)$ measures the similarity of one class element to the other element in the same cluster compared to the other cluster's elements. The silhouette coefficient is computed as follows [10]:

$$s(i) = \frac{\sum_{i=1}^{n} \frac{b(i)-a(i)}{\max\{a(i);b(i)\}}}{n} \quad (2)$$

where $a(i)$ is the average dissimilarity of the current element i to all elements in the same cluster, and $b(i)$ is the minimum of the average dissimilarity of the current element i, to all elements of the other clusters.

$CUsability$ shows how a cluster (will be a class) usability by looking at the number of public methods. One cluster with one public method is considered useful because it has an interface method to collaborate with the other class or object [6]. $CUsability$ is calculated using following formula:

$$CUsability = \begin{cases} 0 & , mpub = 0 \\ 1 & , mpub \geq 1 \end{cases} \quad (3)$$

where $mpub$ is the number of public methods in the class candidate (in the cluster).

Once individuals are ranked, the process continues to calculate selection probabilities based on the individual's ranks. Linear ranking typically uses a linear function to assign these probabilities. The probability P_i of selecting the individual with rank i is calculated using the following formula [15]:

$$P_i = \frac{maxProb - minProb}{N-1} \times (s - i) + minProb \quad (4)$$

where:

- N is the population size,
- s is a selection pressure parameter,
- $maxProb$ and $minProb$ are the maximum and minimum selection probabilities, respectively. These values are set such that $maxProb + minProb = 1$.

The selection of individuals is done by defining the threshold r (between 0 and 1). The individual is in descending order of rank until cumulative probability (P_i) surpasses r. The individual corresponding to the point where this threshold is crossed is selected. The process repeated until two individuals were selected for crossover and mutation.

D. Crossover Process

The two parents that are taken from the selection process are used in the crossover process. The crossover process used the single-point crossover, which is commonly used in the genetics algorithm.

This process aims to combine the genetic information from two parent chromosomes to create offspring or children's chromosomes.

In single-point crossover, a single crossover point is selected randomly along the length of the parent chromosomes. Genetic material beyond this point is exchanged between the parent chromosomes to create offspring chromosomes. This process divides each parent chromosome into two segments: one segment up to the crossover point and another segment beyond the crossover point. Offspring chromosomes are created by combining the segments from both parents at the crossover point.

Let's denote two parents as P_1 and P_2 with numeric representation as follows:

$P_1 = p_{11}p_{12} \dots p_{1n}$
$P_2 = p_{21}p_{22} \dots p_{2n}$
where:

- n is the length of the chromosomes,
- p_{1i} and p_{2i} represent the numeric alleles at position i in P_1 and P_2 respectively.

The crossover point will be selected based on the crossover rate such that $1 \leq c \leq n-1$. The offspring or children's chromosomes O_1 and O_2 are created as follows:

$O_1 = p_{11}p_{12} \dots p_{1c}p_{2c+1}p_{2c+2} \dots p_{2n}$
$O_2 = p_{21}p_{22} \dots p_{2c}p_{1c+1}p_{1c+2} \dots p_{1n}$
The position of c in the genes is based on the crossover rate of gene length. As a clearer explanation, Fig. 2 depicts the crossover process according to the formulation that has been explained.

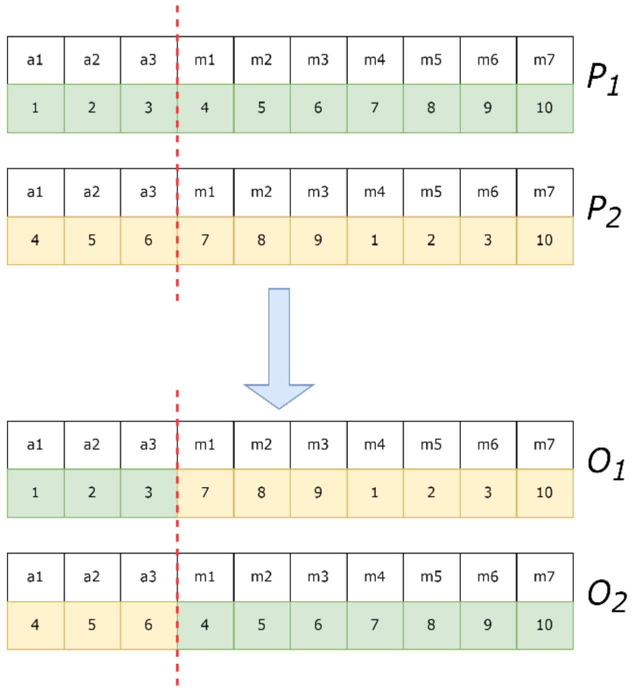


Fig. 2. Crossover process.

E. Mutation Process

Swap mutation is a mutation operator commonly used in the genetic algorithm to introduce population diversity by randomly swapping gene positions within chromosomes. This mutation helps explore new regions of the search space and can prevent premature convergence by maintaining genetic diversity. Let's consider a chromosome C with the numeric representation:

$$C = c_1 c_2 \dots c_i \dots c_j \dots c_n$$

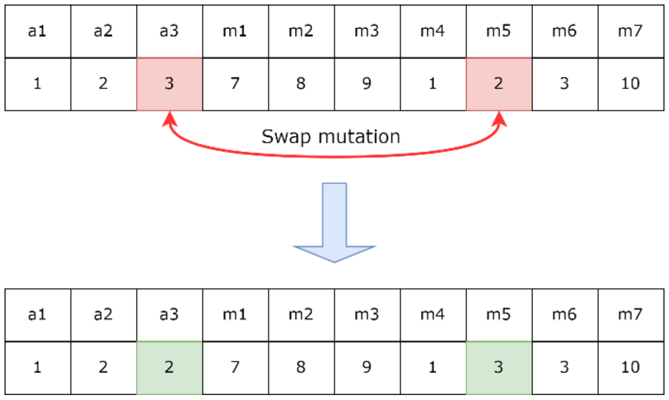
where:

- n is the length of the chromosomes,
- c_i and c_j represent the numeric alleles at position i and j, respectively.

The positions i and j will selected randomly within the chromosome such that $1 \leq i, j \leq n$, and $i \neq j$. The swap mutation is performed by swapping the alleles at position i and j, resulting in a mutated chromosome C'.

$$C' = c_1 c_2 \dots c_j \dots c_i \dots c_n$$

After mutation, the mutated chromosome C' can replace the original chromosome in the population. Fig. 3 shows in detail the implementation of swap mutation in this research. Not all individuals will be mutated. The mutation process only runs when the mutation probability is under the mutation rate. The mutation rate is denoted as p_m represents the probability that a mutation will occur in an individual's genes. The mutation rate notated as $0 \leq p_m \leq 1$.



a1	a2	a3	m1	m2	m3	m4	m5	m6	m7
1	2	3	7	8	9	1	2	3	10

a1	a2	a3	m1	m2	m3	m4	m5	m6	m7
1	2	2	7	8	9	1	3	3	10

Fig. 3. Mutation process.

F. Termination Condition

Termination conditions are typically based on criteria that indicate when the genetic algorithm has reached a satisfactory solution or when further iterations are unlikely to yield significant improvements. There are many options to define the termination conditions, including reaching the maximum number of generations, achieving a desired fitness level, reaching a stagnation point, or exhausting computational resources. In this experiment, the stagnation termination is chosen to terminate the regeneration process in the algorithm.

Stagnation termination is one of the termination conditions options that checks if the algorithm has reached a point where there is no significant improvement in the population's fitness over several generations. In this experiment, S_{max} represents the maximum number of generations without improvement.

The termination condition can be expressed as:

$$generation - last\_improvement \geq S_{max}$$

where $last\_improvement$ is the generation index when the last significant improvement occurred.

III. Experiment Scenarios

The implementation of GA to do class decomposition on the design level using a class diagram was conducted based on the problems that were found in the previous research. There is a class that still has problems related to optimal decomposition. The class is PerspectiveConfigurator from ArgoUML applications. The PerspectiveConfigurator class has been decomposed using the previous approach, but we still have not found the optimal composition due to only one public method in the class and the possibility of being trapped in a local optimum.

The scenario of this research is as follows. The first, the genetic algorithm concept that is described in section two, will be implemented in the prototype application to make the experiment run efficiently. The next step is class profile extraction. This process aims to collect all class information as a basic knowledge to do decomposition [7], [16]. After the class profile was collected, the process continued to decomposition. The decomposition will be done in two processes: the first decomposition using the previous approach [3] and the second using the proposed approach. For the final process, the result of the decomposition will be analyzed and compared to get the comparison results. Fig. 4 shows how this experiment will be held.

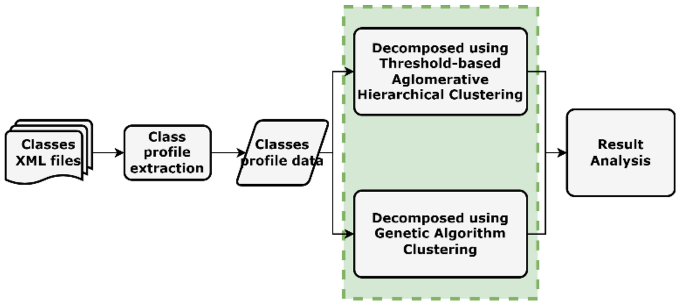

Classes XML files
Class profile extraction
Classes profile data
Decomposed using Threshold-based Aglomerative Hierarchical Clustering
Decomposed using Genetic Algorithm Clustering
Result Analysis

Fig. 4. Experiment scenario.

The use of genetic algorithms is justified by its advantages in overcoming local optimum problems. This research assumes that the decomposition of class at the level of design using genetic algorithms will produce better results in class composition. Before running the scenarios that are explained in Fig. 4, the preliminary experiment will be run to find the best configuration in the case of this research. The genetic algorithm might generate a different solution based on the references in each run [17]. Therefore, every experiment attempt will run ten times to find the best solution.

IV. Experiment Result and Discussion

Several factors can influence the performance of GA, but one of the most crucial factors is the selection of appropriate parameters. The factors are as follows [18].

1) Population Size: The size of the population impacts the diversity of solutions explored by the algorithm.

2) Crossover Rate: The probability of crossover determines the extent to which genetic material is exchanged between individuals in the population. A higher crossover rate encourages exploration by promoting the recombination of genetic material, while a lower rate may lead to slower convergence.

3) Mutation Rate: The mutation rate controls the probability of introducing random changes in individuals' genomes. A higher mutation rate can help maintain genetic diversity and prevent premature convergence, while a lower rate may lead to stagnation in the search process.

Furthermore, the performance analysis is done by running several scenarios based on the three factors. This analysis aims to know how the best configuration of GA is to be implemented in the class decomposition process (using PerspectiveConfigurator class as an object of study).

B. Population Size

The first scenario was done with the population size. The GA will run five times with different population sizes, starting from 10 increments by ten until 50. The result of the experiment is shown in Table I. Table I shows the data in every run based on the population size. The data collected are average fitness, average time, number of generations, and standard deviation. Based on the comparison of collected data in Table I, the population size 10 is the best solution among the other populations. The average fitness value is 0.449, and the average time is 1575.7 milliseconds, which is the smallest generation number. However, the standard deviation is the highest compared to the others. The high standard deviation indicates that each individual's fitness value is spread and is not close to the average fitness. Sometimes, the GA finds very low fitness and sometimes very high. However, even so, in a population of 10, it can find the best solution more frequently. The standard deviation shows the performance of GA in the case of exploration and exploitation. The standard deviation value is assumed to correlate with the ability to perform randomization to produce diversity. The high value assumes that the randomization leads to a fast convergence result. Sometimes, it leads to the right way, but sometimes, it will get lost in the wider search space. It is worrying that with a high standard deviation value, there are areas that are not explored in the search space. That is why this approach is more effective in small populations.

TABLE I. Decomposition of Each Population Size

No.	Population	Average Fitness	Average Time (ms)	Generation	Standard Deviation
1	10	0.449	1575.7	714	0.425
2	20	-0.485	6853.9	1494	0.050
3	30	-0.530	35180.1	4858.6	0.023
4	40	-0.542	212647.6	21621.4	0.0227
5	50	-0.557	1912107.1	135802.5	0.0400

Fig. 5 shows the comparison of average fitness in different population sizes. With this result, running GA in the small population size in this research proves that ten individuals are an effective number to find the solution. The solution exploration is limited to a small area near the most optimum solution in a small population.

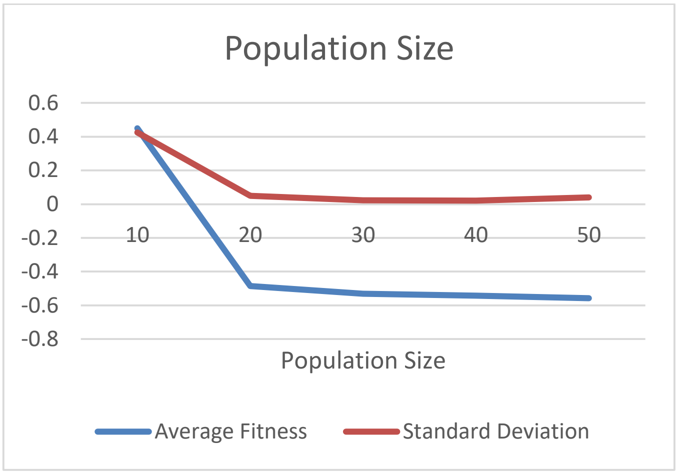


Fig. 5. Average fitness on different population size.

The higher population size produces more diverse solutions in the solution space and lowers the probability of finding the best solution. This is confirmed by the population size 20 to 50 data, which shows that the average fitness is in the range of -0.4 and below and has a low standard deviation. The next experiment scenario will use ten as the population size.

In Fig. 5, the population from ten to 20 seems to decrease significantly. For more detail, it runs more decomposition processes using population size detail between ten to 20 with the increment of two. Fig. 6 shows the result of the decomposition process. The trend of average fitness between populations ten to 20 is decreasing slightly with every increment of population size. The standard deviation sometimes increases due to the discovery of the best solution among decomposition experiments using GA.

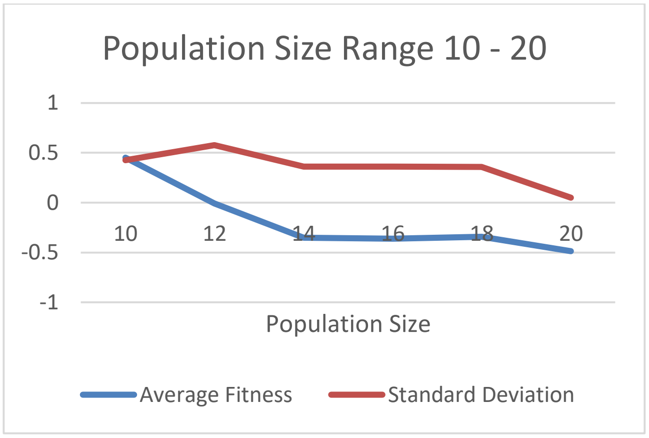


Fig. 6. Population size between 10 and 20.

C. Crossover Rate

GA is a random-based solution finder. The random process seems to be the core of finding the best solution. The crossover rate is the percentage of crossover that will be done in every regeneration process. This rate will determine how many genes will be exchanged between two parents to produce offspring or children. This experiment will be run at several crossover rates starting from 0.1 with increments of 0.1 until 0.9. Every crossover rate runs ten times. Table II shows the result of the experiment using different crossover rates.

TABLE II. CROSSOVER RATE (POPULATION SIZE = 10)

No.	Crossover Rate	Average Fitness	Average Time (ms)	Generation	Standard Deviation
1	0.1	-0.324	3670.5	1632.4	0.348
2	0.2	0.009	3017.6	1331.6	0.563
3	0.3	-0.118	2065.8	883.1	0.539
4	0.4	0.010	1860	797.5	0.562
5	0.5	0.228	2167	938.7	0.560
6	0.6	0.115	1917.1	838.2	0.577
7	0.7	0.223	2324.8	1001.6	0.566
8	0.8	0.222	2879.8	1078	0.568
9	0.9	-0.218	7240.9	2660.4	0.464

Based on the result in Table II, the crossover rate of 0.5 is the best solution, with an average fitness of 0.228. The standard deviation is relatively high, with the same pattern as the crossover rate of 0.2 to 0.8. A high standard deviation value indicates that at the specific crossover rate, there is a possibility of finding a solution with high fitness. The crossover rates of 0.1 and 0.9 have lower standard deviations, which means a lower possibility of finding the best solution. The high average fitness is on the crossover rate between 0.5 to 0.8. Based on this result, genes' best exchange and shuffling portion start from 50% to 80%. Fig. 7 shows the average fitness on different crossover rates, with the best rate of 0.5. The next scenario will use 0.5 to do the next scenario.

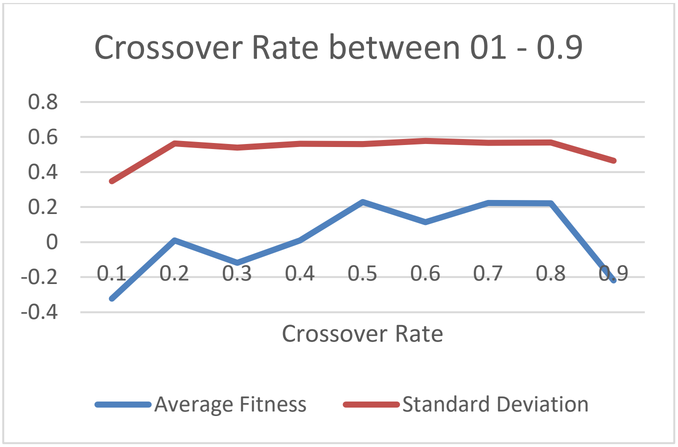


Fig. 7. Average fitness on different crossover rates.

D. Mutation Rate

The next scenario is to run GA for class decomposition in different mutation rates. The mutation is the gene shuffle in the individual. The mutation rate is the possibility that the mutation process will be implemented in the individual during every regeneration process. The experiment will run using several mutation rates starting from 0.1 until 1 with an increment of 0.1. The result of the experiment is shown in Table III.

TABLE III. MUTATION RATE (POPULATION SIZE = 10)

No.	Mutation Rate	Average Fitness	Average Time (ms)	Generation	Standard Deviation
1	0.1	-0.577	84.6	22.1	0.026
2	0.2	-0.575	120.8	41.4	0.026
3	0.3	-0.588	424	172.4	0.024
4	0.4	-0.565	844.1	353.2	0.048
5	0.5	-0.491	2001.8	834.9	0.039
6	0.6	-0.339	2406.7	977.2	0.353
7	0.7	-0.236	2649.4	1061.8	0.474
8	0.8	-0.110	2422.2	952.7	0.534
9	0.9	-0.095	1953.7	819.5	0.523
10	1.0	0.327	1706.2	730.3	0.539

Based on the result shown in Table III, the best result is mutation rate 1. This means that the best solution can be found when mutated genes are in every regeneration more frequently than the other's mutation rate. The mutation rate of 1.0 produces an average fitness of 0.327, but the standard deviation is relatively high. It has the same pattern as the other experiment scenarios. Fig. 8 shows the average fitness in every mutation rate. The average fitness is climbing, starting from a mutation rate of 0.1 to 1. The standard deviation starts to climb higher on the mutation rate of 0.6 simultaneously with the increase of standard deviation. This means that starting from 0.6, the possibility of finding the best solution increases until the mutation rate is 1.

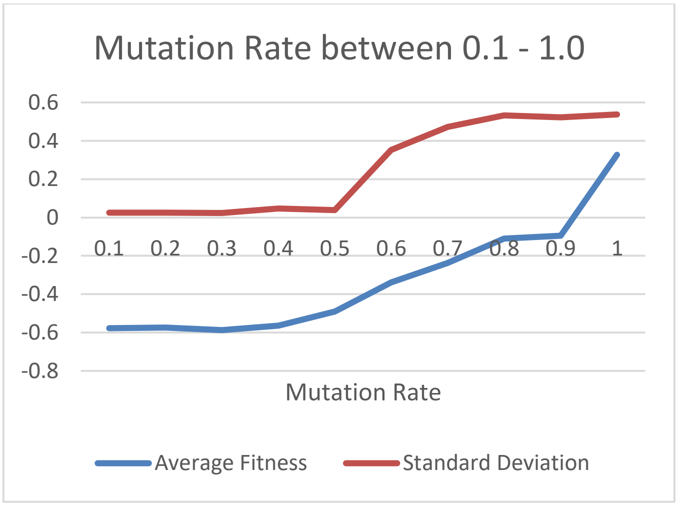


Fig. 8. Average fitness on different mutation rates.

E. Comparing to the Previous Experiment Result

Based on the previous experiment, the PerspectiveConfigurator class is one of the problematics classes [8]. There is only one public method in this class, which is also indicated as Blob class. The clustering process using the previous method (Agglomerative Hierarchical Clustering/AHC + Evaluation) indicates that there was still a problem with the result.

In this research, one attempt of the experiment was re-run using AHC, and the evaluation process used the weight of Silhouette and CUsability, which are 0.5 and 0.5, respectively.

The results show that (Table IV) this approach produces two clusters, and one of those clusters cannot be implemented due to the nonexistence of a public method. The cluster without a public method will be implemented as a class without a public method. For the instantiation, it will produce the selfish object that cannot collaborate with the other objects. Table IV shows the output of the clustering process by prototype application.

TABLE IV. CLUSTERS RESULT OF AHC APPROACH

Cluster 1	Cluster 2
perspectiveConfigurator perspectiveRulesList sortJListModelMethod doRemoveRuleMethod doAddRuleMethod updateRuleLabelMethod updatePersLabelMethod updateLibLabelMethod renameTextField splitPane perspectiveListModel perspectiveRulesListModel ruleLibraryListModel configPanelNorth configPanelSouth INSET_PX LOG loadLibraryMethod loadPerspectivesMethod ruleLibLabel makeListsMethod rulesLabel persLabel makeButtonsMethod makeLayoutMethod makeListenersMethod moveUpButton newPerspectiveButton ruleLibraryList perspectiveList resetToDefaultButton moveDownButton removeRuleButton	duplicatePerspectiveButton addRuleButton removePerspectiveButton

For comparison, the clustering process is done using the genetic algorithm with the same specification (weight 0.5 for each Silhouette and CUsability). The results show that the genetic algorithm produces only one cluster. In other words, the most optimum cluster composition for PerspectiveConfigurator is not to be decomposed because of the constraint of class usability.

Decomposition results that cannot be used are something to be avoided because they are useless. With only one cluster produced by the GA, it will be instantiated to be one object that still has the ability to collaborate with other objects.

The AHC and GA approaches produce the highest Eval values of 0.42 and 0.661, respectively. The following figure shows the comparison. Fig. 9 shows the iteration log of the decomposition process using AHC, which shows the growth of the Eval value. Fig. 10 shows the generation log using GA, which shows the average fitness value (Eval value). It shows the growth of fitness value from generation one until generation 499. At the end of the regeneration process, to find the best solution, there are significant fitness fluctuations. The random process from GA (crossover and mutation) seems to produce significant movement to find the best solution. The results shown in Fig. 9 and Fig. 10 prove that GA, with its random mechanism, is able to find a higher fitness value (Eval value) than AHC, which is assumed to pass the local optimum.

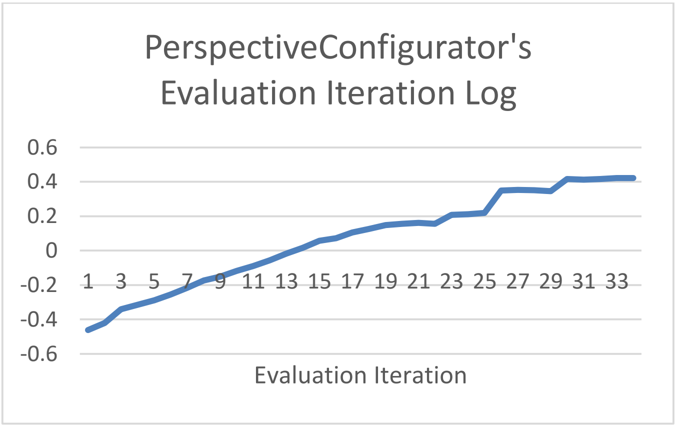


Fig. 9. Perspective configurator using previous approach's log.

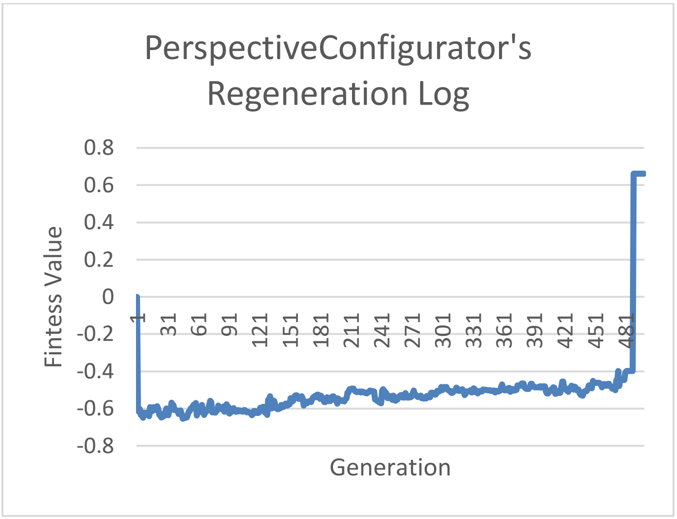


Fig. 10. Perspective configurator using GA's log.

V. CONCLUSION

Class decomposition is one of the interesting fields in refactoring research, especially when refactoring is done at the design level. This paper has proposed an approach to class decomposition utilizing GA and demonstrated its superiority over the traditional agglomeration hierarchical clustering method (previous approach). Through rigorous experimentation and analysis, it has been evidenced that the GA-based approach outperforms the hierarchical clustering method in terms of result quality. Based on the problems that are a focus of this research, the PerspectiveConfigurator class's problem can be solved by utilizing GA. The fitness value comparison (Eval value in the previous approach) shows that there is an increment in the utilization of GA in this research. The fitness values of AHC and GA are 0.42 and 0.661, respectively. On the fitness value 0.661, the clustering result produced by GA results in only one cluster, but it matches the quality criterion (considering the silhouette and CUsability).

The rationalization behind the effectiveness of the GA lies in its ability to explore through the search space and efficiently navigate through various combinations of class decompositions. Unlike hierarchical clustering, which tends to produce suboptimal solutions due to its greedy nature and dependence on initial conditions, the genetic algorithm employs a population-based evolutionary strategy to converge towards globally optimal solutions while avoiding local optima. An increase in the fitness value of GA proves this.

The standard deviation typically plays a role in guiding the exploration and exploitation phases of the optimization process. Specifically, the standard deviation is often associated with randomization operators within the GA. Finding a high standard deviation value raises the desire to conduct a deeper exploration regarding this matter in the future. For the future plan, adjusting the standard deviation by adjusting the randomization operators (mutation and crossover mechanism) is assumed to fine-tune the balance between exploration and exploitation, thereby influencing the GA's ability to efficiently search for more optimal solutions.

REFERENCES

[1] Sommerville, Software Engineering, 9th ed. Harlow, England: Addison-Wesley Professional, 2010.

[2] R. Pressman, Software Engineering : A Practitioner's Approach, 7th ed. USA: McGraw-Hill, Inc., 2009.

[3] M. Fowler et al., Refactoring Improving the Design of Existing Code Second Edition, Second Ed. United State of America: Pearson Education - Wesley, 2019.

[4] B. Priyambadha, T. Katayama, Y. Kita, H. Yamaba, K. Aburada, and N. Okazaki, "The Seven Information Features of Class for Blob and Feature Envy Smell Detection in a Class Diagram," The 2021 International Conference on Artificial Life and Robotics (ICAROB2021), pp. 348–351, 2021.

[5] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design. in Robert C. Martin Series. Boston, MA: Prentice Hall, 2017.

[6] B. Priyambadha and T. Katayama, "Enhancement of Design Level Class Decomposition using Evaluation Process," International Journal of Advanced Computer Science and Applications, vol. 13, no. 8, pp. 130–139, 2022, doi: 10.14569/IJACSA.2022.0130816.

[7] B. Priyambadha and T. Katayama, "Design Level Class Decomposition using the Threshold-based Hierarchical Agglomerative Clustering," International Journal of Advanced Computer Science and Applications, vol. 13, no. 3, pp. 57–64, 2022, doi: 10.14569/IJACSA.2022.0130310.

[8] B. Priyambadha and T. Katayama, "The Impact of Design-level Class Decomposition on the Software Maintainability," International Journal of Advanced Computer Science and Applications, vol. 14, no. 4, pp. 405–413, 2023, doi: 10.14569/IJACSA.2023.0140445.

[9] X.S. Yang, Nature-inspired metaheuristic algorithms. Luniver Press, 2010.

[10] H. Nguyen, S. J. Louis, and T. Nguyen, "MGKA: A genetic algorithm-based clustering technique for genomic data," 2019 IEEE Congress on Evolutionary Computation, CEC 2019 - Proceedings, pp. 103–110, Jun. 2019, doi: 10.1109/CEC.2019.8790225.

[11] S. Kebir, I. Borne, and D. Meslati, "A genetic algorithm-based approach for automated refactoring of component-based software," Inf Softw Technol, vol. 88, pp. 17–36, Aug. 2017, doi: 10.1016/j.infsof.2017.03.009.

[12] L. Gang, "Genetic Algorithm and Its Application in Software Test Data Generation," in 2023 International Conference on Applied Intelligence and Sustainable Computing (ICAISC), 2023, pp. 1–6. doi: 10.1109/ICAISC58445.2023.10200303.

[13] Y. Dong and J. Peng, "Automatic generation of software test cases based on improved genetic algorithm," in 2011 International Conference on Multimedia Technology, 2011, pp. 227–230. doi: 10.1109/ICMT.2011.6002999.

[14] S. I. Ayon, "Neural Network based Software Defect Prediction using Genetic Algorithm and Particle Swarm Optimization," in 2019 1st International Conference on Advances in Science, Engineering and Robotics Technology (ICASERT), IEEE, May 2019, pp. 1–4. doi: 10.1109/ICASERT.2019.8934642.

[15] A. E. Eiben and J. E. Smith, Natural Computing Series Introduction to Evolutionary Computing, Second Edition. Springer Publishing Company, Incorporated, 2015.

[16] B. Priyambadha and T. Katayama, "Tree-based keyword search algorithm over the visual paradigm's class diagram xml to abstracting class information," 2020 IEEE 9th Global Conference on Consumer Electronics, GCCE 2020, pp. 280–284, 2020, doi: 10.1109/GCCE50665.2020.9291865.

[17] W. F. Mahmudy, R. M. Marian, and L. H. S. Luong, "Real Coded Genetic Algorithms for Solving Flexible Job-Shop Scheduling Problem - Part II: Optimization," in Key Engineering Materials III, in Advanced Materials Research, vol. 701. Trans Tech Publications Ltd, Mar. 2013, pp. 364–369. doi: 10.4028/www.scientific.net/AMR.701.364.

[18] A. E. Eiben and J. E. Smith, Introduction to Evolutionary Computing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. doi: 10.1007/978-3-662-44874-8.