AUT Journal of Modeling and Simulation

AUT J. Model. Simul., 55(2) (2023) 227-242
DOI: 10.22060/miscj.2024.22381.5324

RMMOC: Refactoring Method based on Multi-Objective Algorithms and New Criteria

Mohammad Reza Keyvanpour*, Zahra Karimi Zandian, Zohreh Razani

Department of Computer Engineering, Faculty of Engineering, Alzahra University, Tehran, Iran

ABSTRACT: Some factors can change the software and affect the quality, such as the new users' requirements and the need for compatibility with modern techniques. These factors impose a high cost on technical software maintenance. One of the techniques for software quality improvement and maintenance cost reduction is refactoring. The advantage of this method is software behavior preservation. Because the cost of refactoring manually is high, a technique called the hybrid optimization problem has been proposed. The main challenge in refactoring is to propose a technique with high accuracy and less runtime. Hence, in the present work, a refactoring method based on the multi-objective algorithms called RMMOC is proposed to tradeoff between quality and runtime. This method uses a helpful search-based method called UMOCell to increase refactoring quality. This method inspires both population-based and local-based search algorithms. Another novelty in this paper is using new metrics for program quality assessment that help increase accuracy, decrease refactoring runtime, and find the best solutions. Because software metrics play a significant role in search-based refactoring approaches, this paper introduces two effective criteria called MPC and refactoring number reduction in addition to previously presented metrics. The experiments' results show that the proposed method's performance is remarkable and that using new metrics is effective.

Review History:

Received: Jun. 03, 2023
Revised: Jan. 09, 2024
Accepted: Mar. 02, 2024
Available Online: May, 30, 2024

Keywords:

Refactoring
Software Quality
Search-Based Refactoring
Multi-Objective Algorithm

1- Introduction

Software changes during the whole of its lifecycle [1]. These changes can be due to adding new features, software correction, improving the design and optimization of resource consumption, adapting the software to new needs and technologies, etc [2]. These developments reduce reliability, change the software's initial and expected behavior, and increase technical software maintenance costs [3]. Therefore, it is necessary to take appropriate actions to reduce these impacts and increase quality [4]. One of the approaches to improving software quality and design is refactoring [5][6][7]. Changing a software system that improves its internal structure without any alteration in the code's external behavior is called refactoring [8][9]. The idea is to organize variables, methods, and classes for subsequent development and adaptation [10]. This approach decreases software complexity [6] and increases developers' understanding. Memory and start-up time performance is recovered by refactoring [11], upgrading software comprehensibility, and modifying it at a lower cost.

One of the challenges in manual refactoring is the high cost of this process. Indeed, there is more than one correct solution. It means the order of the candidate sets of refactoring can be different, and, as a result, different designs are obtained [12].

On the other hand, choosing the best set is hard in complicated and extensive software. Researchers have proposed refactoring as a hybrid optimization problem and applied search-based methods to address this shortage [13]. The idea is to convert different problems to hybrid optimization or search solutions by meta-heuristic methods [14]. Search-based refactoring finds the best refactoring order automatically to improve software quality. The general structure of software search-based refactoring is shown in Figure 1.

As the main challenge in refactoring is to propose a technique with high accuracy and less runtime [15], a new search-based refactoring method called RMMOC is proposed, where an open-source tool called Recoder [16] and an automatic refactoring tool called Multi-refactor [17] are used. A new and valuable algorithm based on population-based and local-based algorithms is introduced to increase performance and quality. In addition, two effective criteria called Message Passing Coupling (MPC) and refactoring number reduction are suggested. These metrics help increase accuracy, decrease refactoring runtime, and find the best refactoring solutions. The results show the efficiency of the proposed method in refactoring.

The rest of the paper is organized as follows. In Section 2, the related work is discussed. In Section 3, the proposed

*Corresponding author's email: keyvanpour@alzahra.ac.ir

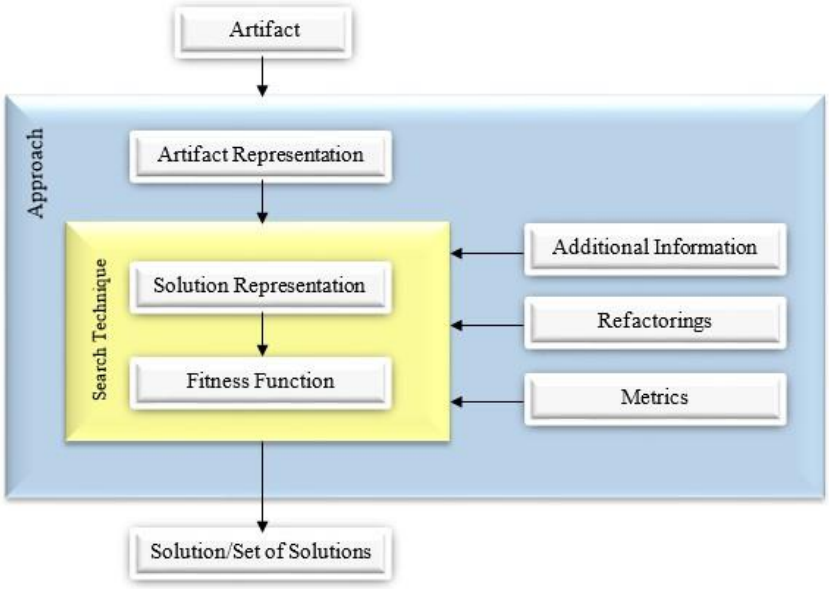

Fig. 1. The general structure of software search-based refactoring [13]

method is introduced; experiments and evaluation results are presented in Section 4, followed by the concluding remarks in Section 5.

2- Related Work

Each article should contain the following main parts: This section provides an overview of the related work in search-based refactoring. Before explaining the related work, we must define the software search-based refactoring problem for easy understanding.

Problem definition. $R_A : \Omega \rightarrow M_A, \Omega = ST \times M \times R \times I$ where R_A is artifact refactoring, M_A represents modified or refactored artifact obtained from the search-based algorithm, ST shows the chosen search-based algorithm, M is the metrics used or proposed for refactoring, R is candidate refactorings, and I is additional information.

So far, different methods have been proposed for problem optimization in software; one is a search-based algorithm. This approach has also been considered in the software refactoring field. As mentioned in some research, like [18][19], search-based refactoring has various challenges. A review of the proposed methods in search-based refactoring indicates that they can be classified based on five different views, as shown in Figure 2: refactored artifact, automated level, refactoring technique, search algorithm, and user feedback.

2- 1- Refactored Artifact

As shown in Figure 2, refactoring methods have used two artifact types: code-based and model-based. Approaches based on the model consider models as the first group of artifacts in the software life cycle [20]. In these methods, the source code quality created based on the models depends on the models' quality [21]. Recently, different methods have been proposed based on the model for class diagram [21][22][23], activity diagram [24], and sequence diagram [25] refactoring. The researchers in [26][27] have used model transformation for the refactoring method. In this approach, the model is obtained from the source code. The designer must decide about the applicable refactorings and metrics. Evaluating the effect of refactorings on the model takes much work.

Approaches based on the code consider source codes in the software life cycle [28] and try to boost code structures [29]. Source code refactoring is reported in various programming languages. Most search-based refactoring approaches have been designed for object-oriented open-source programs based on Java language [30]. In code-based methods, programs with various sizes and applications are used, and different metrics are presented for their evaluation. Besides, a wide range of search algorithms have been applied in this type of software refactoring. Some techniques decrease search spaces by removing the refactoring operations that overlap or are interdependent, such as [31][32]. Others do this by removing the refactoring sequence of the operations overlapped, like [26][27]. Some methods have been proposed for code refactoring at the packet level among code-based approaches, like [33][34]. In this approach, evaluating the effect of refactorings on the source code is easier than the model. On the other hand, due to the existing information in the code, investigating pre-conditions and applying the refactorings take much work.

2- 2- Automated Level

Figure 2 shows that the refactoring methods can be divided into two categories based on the automated level: manual and automated. In some methods like [33][34][35][36],

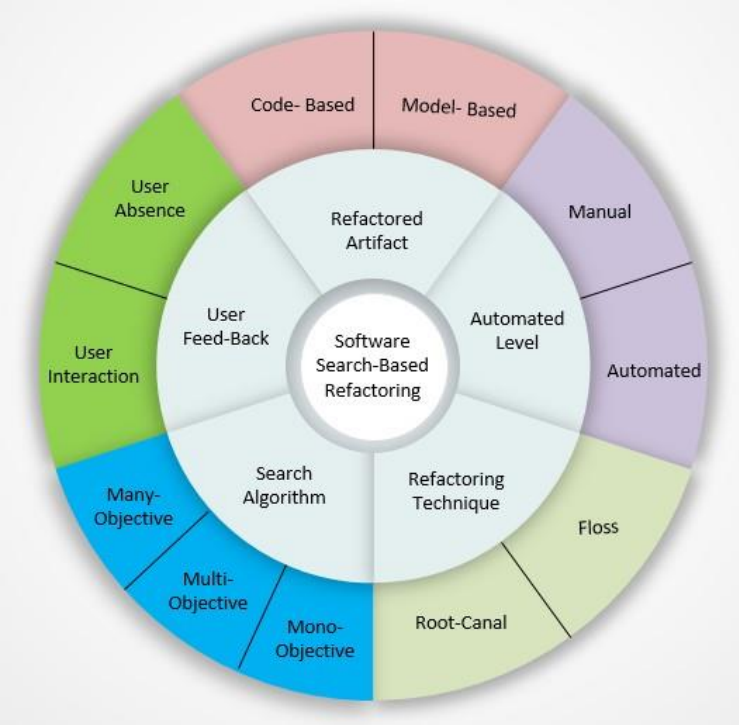

Fig. 2. Software Search-Based Refactoring Classification based on five views

refactoring is applied by the users manually. In this category, developers manually select and implement the refactoring when the search process is finished and the optimized refactorings are found. Indeed, the output of these methods is the proposed optimized order of refactorings. In this approach, the user's idea is applied to choose the refactorings, but this method is expensive and time-consuming. The application of refactoring is the user's task.

In contrast, automated approaches like those proposed in [17][37][38][39][40][41][42][43] are the methods where the refactorings are applied directly to the artifact. The output is the refactored artifact. As the refactoring process is automated in this category, the necessary time for refactoring is decreased compared to the manually refactoring-based category. It is also easier to ensure behavior preservation of the program to change unwantedly in the automated methods due to regular refactoring [13]. On the other hand, the programmers must be informed of the different changes in software designs [44], and the users have no role in the final decision.

2- 3- Refactoring Technique

The proposed methods in the area of refactoring use different refactoring techniques regarding how they are combined with other programming activities and repetition intervals of refactoring operations. By investigating these methods, the refactoring techniques can be divided into two groups [45] (Figure 2): floss and root-canal refactoring. In floss refactoring, the programmers perform refactoring simultaneously as the other kinds of program change. In other words, during the refactoring process, floss mixes other program changes with refactoring to keep the source code intact [45]. In [46], the researchers have proposed a floss-based method called ReCon.

In comparison, root-canal refactoring is used to modify degraded codes. Root-canal refactoring is suitable for search-based refactoring using the whole program and providing a set of refactorings as solutions, such as [47][48][49]. This approach improves the design quality more than the other one. On the other hand, the floss technique needs less time to refactor and is more common than root-canal refactoring [50].

2- 4- Search Algorithm

Search-based refactoring methods are divided into three categories (Figure 2) based on the algorithm used to search for the best solution: mono-objective, multi-objective, and many-objective algorithms [18]. The mono-objective search algorithm optimizes the problem based on one fitness function metric and returns just one final solution. Mono-objective methods can be divided into local and evolutionary ones like [44][46][51][52]. Despite local mono-objective methods, a sequence of valuable refactorings that improve the system's overall quality is evaluated in evolutionary mono-objective methods [52]. Eqs. (1) and (2) show the evaluation method

for local mono-objective and evolutionary algorithms, respectively.

$$Evaluate\ F(x); x = \{ref_1, ref_2, ..., ref_n\} \quad (1)$$

$$Evaluate\ F(X); X = \{x_1, x_2, ..., x_m\}; x_i = \{ref_1, ..., ref_n\} \quad (2)$$

Mono-objective methods are the easiest, although they do not allow different metrics and various solutions to be investigated simultaneously.

Multi-objective-based methods optimize more than one metric in fitness functions [33][48][49]. Eq. (3) expresses the evaluation method for multi-objective algorithms.

$$\begin{aligned} &Evaluate\ F_1(X), F_2(X), F_3(X); \\ &X = \{x_1, x_2, ..., x_m\}; \\ &x_i = \{ref_1, ref_2, ..., ref_n\} \end{aligned} \quad (3)$$

In some approaches, the metrics are calculated by their weighted sum [17][53]. Developers can choose a solution from this algorithm's solutions [47]. If the functions do not conflict, the second-category results are improved more than the first-category. Developers can investigate and choose suitable solutions among various ones, but runtime increases in this approach compared to mono-objective search algorithms. Having more than three fitness functions in a refactoring problem causes researchers to use a many-objective algorithm such as [54][55][56]. Eq. (4) shows the evaluation method for many-objective algorithms.

$$\begin{aligned} &Evaluate\ F_1(X), F_2(X), F_3(X), ..., F_k(X); \\ &X = \{x_1, x_2, ..., x_m\}; \\ &x_i = \{ref_1, ref_2, ..., ref_n\} \end{aligned} \quad (4)$$

Due to several objectives to optimize this method, monitoring and displaying the solutions are more complex [54][56]. As the purpose is to achieve the best solution among various solutions that is suitable for many objectives, the runtime is also the highest in this method [56], although the method improves the scalability of search-based approaches (due to investigating many objectives), which increases their efficiency in industrial and natural environments [57]. Also, developers can choose the best solutions in these methods.

2- 5- User Feedback

Based on the user participation rate in the search process, solution evaluation can be divided into user interaction and user absence (Figure 2). After obtaining solutions, user interactions can be utilized in solution fitness evaluation algorithms [58]. In this approach, the users contribute to choosing the target solutions, and their ideas affect the final result. Some approaches interact with the user in the refactoring sequence production step, like [22]. In addition, user interaction is considered in the UML designing step [42] or for source code refactoring [33][40][48]. In contrast, in some algorithms, users do not have any role in solution quality improvement, and the method chooses the best solution in the search process [59].

3- RMMOC: Refactoring Method based on Multi-Objective Algorithms and New Criteria

As mentioned before, the main challenge in refactoring is to propose a technique with high accuracy and less runtime. Therefore, we propose a new refactoring method based on the search-based algorithm. The proposed system receives the Java source code as the input, and the refactored source code in the particular file is presented as the output after the refactoring process. The general structure of RMMOC is shown in Figure 3. As indicated in this figure, the proposed software search-based refactoring includes three steps: converting source code to meta-model and vice versa using Recoder [16] framework, determining optimized refactorings, and applying optimized refactorings on the meta-model. As shown in the figure, we propose a new method to determine optimized refactoring.

3- 1- Converting Source Code to Meta-Model and vice versa based on Recoder

As shown in Figure 3, the input for converting the source code to the meta-model step is the Java source code, and the output is Abstract Syntax Trees (ASTs). Refactored Abstract Syntax Trees (RASTs) are the input of converting meta-model to source code step, and its output is refactored source code.

Recoder is a Java framework for source code meta-programming, whose purpose is to present an advanced infrastructure for analyzer, parser, and Java converter tools [16]. Recoder makes a meta-model, including source code entities and class files, to provide meta-programming. Indeed, this model is the exact syntax model of the program, including the comments. The syntax model is the attributed syntax tree where each entity connects to its parents and children. The Recoder uses this tree to transfer, analyze, parse, and convert source code to meta-model and vice versa.

3- 2- Determining Optimized Refactorings

This step receives ASTs, Desired Refactorings (DR), and Desired Metrics (DM) as inputs. The output of determining the optimized refactoring step is Optimized Refactoring Operations (ORO). Each embedded refactoring in the system and the preconditions determining their application are implemented in this step. Besides, each metric and its calculation method is introduced and investigated. DR used in the proposed method is mentioned in [8][17][37]. As shown in Figure 3, we propose novel metrics in addition to previous metrics like DM. To increase the accuracy of the refactoring and find the best refactoring solutions, we propose novel metrics in addition to previous metrics like DM. As shown in Figure 4, the first step in the search algorithm for solution space mining is to choose a refactoring operation

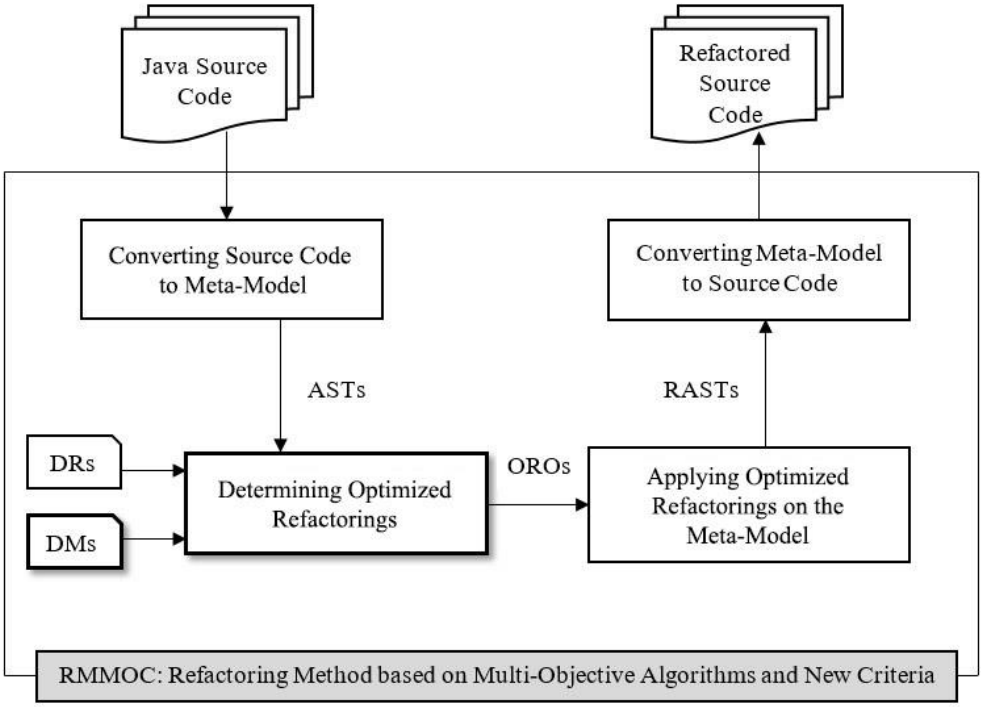

Fig. 3. The general structure of the RMMOC method

sequence. A refactoring operation includes declaring the refactoring operation type, method, field, or class undergoing the refactoring. After choosing the refactoring operation sequence, the chosen sequence applies to the meta-model in the second phase. According to the proposed method, the amount of the used metrics or meta-model fitness is calculated by RASTs obtained from the previous step in the next step. Then, in the last step, the meta-model is restored to its original state, or if the refactoring improves the meta-model, it is added to the refactoring order. Then, the stop condition and satisfying it are investigated. Optimized refactoring operations are returned from this step as the output.

In the proposed search algorithm, population-based search algorithms like NSGA-II [60], genetic algorithms, and search algorithms based on the local search, such as Hill Climbing and Simulated Annealing (SA), are used and investigated. When we use a population-based search algorithm, as shown in Figure 4, the meta-model is restored to its original state after calculating meta-model fitness. However, when we utilize a local-based algorithm, after calculating meta-model fitness, it is added to the refactoring order if the refactoring improves the meta-model.

In the population-based search algorithm, a refactoring sequence is chosen in each step of choosing refactoring until a sequence population is created. In contrast, in the search algorithm based on the local searches, one refactoring sequence is used throughout the search process. After applying the sequence on the meta-model and calculating the fitness function, if the program quality is improved, that sequence is added to previously optimized sequences. According to the proposed method in this paper, the updated search-based algorithm called UMOCell is used based on a multi-objective cellular genetic algorithm (MOCell) [61]. In UMOCell, the current sequence (individual) is replaced with the new one (obtained after combination and mutation) if it overcomes the current one. If both sequences are unsuccessful, the new sequence is compared with all the neighbors of the refactoring sequence in the population; if the new one has the worst congestion distance, that sequence is added to the optimal population (archive) and is not included in the current population. If the worst congestion gap does not belong to the new sequence, the sequence is added to the current population (auxiliary population) and the optimal population.

3- 2- 1- Choosing Refactoring Operation Sequence

As shown in Figure 4, this step receives DRs and ASTs as inputs and forwards OROs to the next step. In this step, the first one, refactoring, is chosen at random among the refactorings used in the system (DRs). The proposed method searches for suitable elements in ASTs for each chosen refactoring. If a suitable element is not found for the chosen refactoring, another refactoring is chosen from the list. Finally, the searching process is finished if the list becomes empty or all of the refactorings are matched with the elements. Upon completion of this process, refactorings used in this step and suitable for the elements (OROs) are forwarded to the next step.

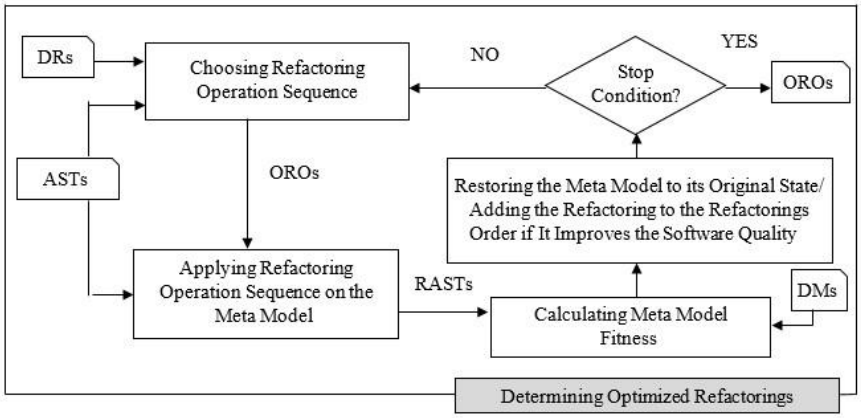

Fig. 4. The general structure of the determining optimized refactorings step

3- 2- 2- Calculating Meta-Model Fitness

After applying the refactoring operation sequence on the meta-model and obtaining RASTs, this output is sent to the calculating meta-model fitness as an input. In this step, in addition to RASTs, DMs are another input. According to RMMOC, this step aims to combine the valuable metrics existing in DMs and obtain a comprehensive meta-model fitness. The metrics are used for program quality assessment and measurement of the effect of the proposed refactoring on the system's performance. In this paper, we use object-oriented programming-based metrics like metrics sequence of QMOOD [62] (CDS, NOH, ANA, DAM, DCC, CAM, Agg, FA, NPM, CIS, NOM) and CK/MOOSE [63] (WMC, NOC). To create the fitness function from DMs, we need to combine them. For the metric combination, we use a normalization method, as Eq. (5) shows:

$$\sum_{m=1}^{N} D_m . W_m \left(\frac{C_m}{I_m} - 1\right) \tag{5}$$

Where D_m is metric direction, which means is it desirable to reduce the metric or increase it? (-1 or +1). W_m indicates the metric weight. C_m is the current amount of the metric. I_m is the initial amount of the metric. N implies the number of metrics.

As it is clear from Eq. (5), the method increment shows an improvement in the amounts of metrics. Therefore, according to the proposed normalization method, the proposed search-based algorithm aims to maximize the fitness function based on this normalization method.

The new metrics can promote the accuracy of refactoring and improve QMOOD and CK/MOOSE quality metrics, in addition to being helpful in program quality assessment. According to the RMMOC and previous metrics, two other metrics are proposed in this paper to evaluate the effect of refactoring on the program performance more accurately. One of the metrics proposed in QMOOD is direct class coupling (DCC). This metric is a count of different numbers of classes that a class is directly related to. DCC metric does not consider the methods called the class methods bodies, while these methods belong to other classes. Therefore, we use message-passing coupling (MPC) [36][64]. Eq. (6) is used to calculate the MPC metric.

$$MPC = \frac{\sum_{1}^{classes\_num} \frac{exmethods}{all\ methods}}{classes\_num} \tag{6}$$

Where $classes\_num$ is the number of all classes in the project. $Exmethods$ is the number of external methods. Decreasing MPC is desirable.

In the real world, applying extensive changes and refactoring is unpleasant. Many reforms cause the program to stay away from the original design. On the other hand, these changes make developers make more effort to apply or review them in the program. Therefore, researchers and developers prefer solutions that change the program less [33]. In contrast, decreasing the changes must maintain quality. Consequently, another metric proposed in this paper is refactoring number reduction. While x is a refactoring sequence, the purpose of the refactoring number reduction metric is to minimize the size of x .

4- Experiments

We used a system with an Intel Corei7 processor and 8 G RAM to perform the experiments. Eclipse software was used to run the programs. To evaluate the proposed method, it needs to present three main parts. First, the data set used in this paper is explained. In the second part, the evaluation

Table 1. The characteristics of the programs used

Program	The number of lines	The number of classes
Json 1.1	2196	12
Mango	3470	78
Beaver 0.9.11	6493	70
Apache xml-rpc 3.0	11616	79
JHotDraw 5.3	27824	241
GanttProject 1.11.1	31978	245
XOM 1.2.1	47691	217

Table 2. The refactorings used in the proposed system

Refactorings in field level	Refactorings in method-level	Refactorings at class level
Increase field visibility	Increase method visibility	Make class final
Decrease field visibility	Decrease method visibility	Make class non-final
Make field final	Make method final	Make class abstract
Make field non-final	Make method non-final	Make class concrete
Make field static	Make method static	Collapse hierarchy
Make field non-static	Make method non-static	Extract subclass
Move field down	Move method down	Remove class
Move field up	Move method up	Remove interface
Remove field	Remove method	-

criteria are introduced. In the last part, different test methods are introduced, and the results obtained are analyzed and compared with each other based on the evaluation criteria.

4- 1- Dataset

In this paper, the input of the proposed system (RMMOC) is the Java source code. This source code includes Java libraries and applications. The list of programs used in this paper to study and evaluate the RMMOC system is presented in Table 1. The number of lines and classes in each program is shown in this table. These programs have been used in some previous works in search-based refactoring, such as [17][40] [43][65][66].

Jason is a format for data storage and interchange. Mango is a Java library. Beaver is a parser generator. Apache xml-rpc is an implementation for xml-rpc. This program uses XML for the remote procedure call. JHotDraw is a two-dimensional graphical framework for structural drawing editors. GanttProject is a tool for project scheduling and management. XOM is an API for XML file processing.

4- 2- Evaluation Criteria

Before explaining the evaluation criteria, we must express the desired refactorings used and introduce the proposed metrics in this paper. As in other studies, we use some refactorings introduced in [8][17][37]. These refactorings are in field, method, or class levels. In previous sections, DR showed desired refactorings. Table 2 presents the refactorings used in the proposed system.

Researchers have also used various metrics to evaluate the program quality and the effect of the proposed and applied refactorings on the system. As mentioned before, the metrics used or proposed in this paper are based on object-oriented programming. We utilize some metrics applicable in some research, namely the metrics sequence of QMOOD [62] and CK/MOOSE [63].

As mentioned before, we proposed two other metrics in this paper and used what was proposed in other research. *MPC* and *refactoring number reduction* are proposed to accurately evaluate the effect of refactoring on the program's performance.

After introducing the desired refactoring and metrics, as well as the proposed metrics, evaluation criteria are expressed. In software search-based refactoring, different criteria are generally used to evaluate the performance of the proposed methods. In this paper, we use runtime [36][33][67] [68], and the fitness function amount [36][67][69] to evaluate RMMOC system.

We use the following equations to measure the improvement of our fitness function from the desired metrics and the proposed metrics in this paper.

$$MPC = D.W(\frac{C_{MPC}}{I_{MPC}} - 1) \quad (7)$$

$$DCC = D.W(\frac{C_{DCC}}{I_{DCC}} - 1) \quad (8)$$

$$Coupling = MPC + DCC \quad (9)$$

$$Size(x), x = refactoring\ sequence \quad (10)$$

$$Quality = \sum_{m=1}^{n} D_m.W_m(\frac{C_m}{I_m} - 1) \quad (11)$$

Where D_i is metric direction, which examines if it is desirable to reduce the metric or increase it (-1 or +1). W_i indicates the metric weight. C_i is the current amount of the metric. I_i is the initial amount of the metric. n implies the number of metrics.

To evaluate the proposed system based on the proposed metric MPC , investigate the effect of MPC and DCC on performance and use the benefits of both, Eqs. (7)-(9) are introduced. To assess the RMMOC system based on the *refactoring number reduction* metric, Eq. (10) is used. We finally utilize Eq. (11) to evaluate the proposed system based on the previous section and MPC 's desired metrics. As mentioned before, they need to be combined to create a fitness function from metrics. Therefore, we use this equation for this purpose. In this paper, we consider the effect of each metric on the fitness function to be the same and set the weight of each metric at 1. This decision is to prevent biasing the results to special metrics.

4- 3- Experiments Results

It must be tested in different respects to evaluate each new method in each field comprehensively. Therefore, in this paper, two tests have been designed and performed to evaluate the proposed system. In Test 1, the effect of 3 metrics, i.e., MPC, DCC, and Coupling, on the quality function is investigated first. With this test, the necessity of the proposed metric is specified. Then, mono-objective and multi-objective search-based algorithms are compared based on quality function, refactoring number reduction, and runtime. This part of Test 1 determines the superiority of the two search-based algorithm types. Test 2 compares the proposed search-based algorithm UMOCell and NSGA-II based on quality function and runtime for plenary evaluation.

4- 3- 1- Test 1: the effect of MPC, DCC, and Coupling on the quality function and the comparison between search-based algorithms based on quality function, refactoring number reduction, and runtime

The first purpose of this test is to investigate the effect of MPC as the proposed metric on the quality function. The second and main goal is to compare search-based algorithms based on quality function, refactoring number reduction, and runtime. To achieve this aim, we compare the quality function results based on MPC with the quality function results based on DCC and Coupling. Another motivation for doing this test is to compare mono-objective and multi-objective search-based algorithms based on quality function, refactoring number reduction, and runtime. This test shows the advantages and disadvantages of these algorithms. To investigate the effect of MPC on the quality of different refactoring search-based algorithms, we use Eqs. (7)-(9). The genetic algorithm is the primary search algorithm to run this test. The effect of MPC, DCC, and Coupling metrics on the quality function is studied and reported in Figure 5.

Discussion: As shown in Figure 5, the quality improvement gained from using both MPC and DCC as coupling metrics is the highest. Between MPC and DCC, using MPC increases the quality more than the other. This means using coupling leads to the use of the advantages of both, improving refactoring quality. According to these results, the quality based on the Coupling metric is used as the quality metric for other tests.

To compare mono-objective and multi-objective search-based algorithms based on quality function and refactoring number, we use the genetic algorithm as a basic mono-objective algorithm and the NSGA-II algorithm as a known multi-objective algorithm. The fitness functions utilized in this test are quality and refactoring number reduction. Six programs introduced in Table 2 are used as the input dataset. Json project is not used due to its small size. According to the test method in the search-based refactoring approaches [17] [47][48][54], each program runs for each algorithm 30 times. The average of these 30 times is reported for each program and algorithm. Table 3 reports the results of Test 1 based on the proposed metrics.

For greater transparency of the impact of using the proposed metrics, the mean of 5 values of quality and number of refactoring for each program is displayed in Figures 6 and 7.

Discussion: As shown in Figure 6, the mono-objective algorithm's mean quality is better than the multi-objective one for each input. It illustrates that considering the refactoring number reduction function, the quality function amount is decreasing. On the other hand, as shown in Table

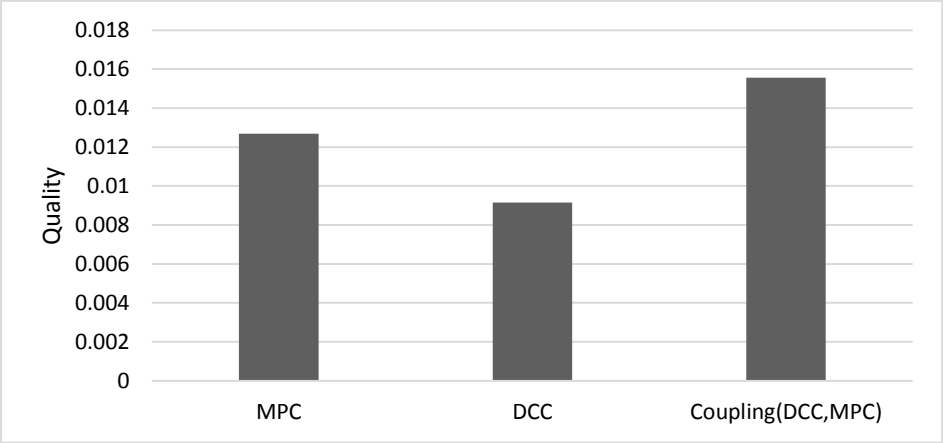

Fig. 5. The quality obtained from different metrics

Table 3. The results of Test 1 based on the proposed metrics

Run#	Quality: mono-objective	Quality: multi-objective	Number of refactoring: mono-objective	Number of refactoring: multi-objective
		Mango		
1	0.196568	0.190647	104	41
2	0.314965	1.017633	104	32
3	0.342409	0.250703	89	37
4	0.178561	0.26635	84	63
5	1.935016	0.176998	109	74
		beaver		
1	0.730704	0.365738	100	41
2	0.593734	0.391962	56	32
3	0.699963	0.315106	72	14
4	0.875781	0.469109	70	23
5	0.579344	0.376679	97	22
		xml-rpc		
1	0.322768	0.457224	82	96
2	0.313031	0.204879	172	53
3	0.31061	0.153211	65	37
4	0.295811	0.149758	116	27
5	0.377166	0.17347	90	30
		JHotDraw		
1	0.545903	0.413414	101	42
2	0.670168	0.310589	112	23
3	0.502241	0.431168	183	40
4	0.562147	0.369394	124	28
5	0.684667	0.402261	134	41
		Ganttproject		
1	0.273976	0.148812	130	44
2	0.278955	0.091526	142	21
3	0.224911	0.235551	93	101
4	0.253131	0.118144	91	32
5	0.246035	0.138006	189	21
		XOM		
1	0.838478	0.688116	145	42
2	0.649268	0.682168	53	27
3	0.775633	0.64568	148	40
4	0.893336	0.586945	120	28
5	0.786251	0.664308	86	61

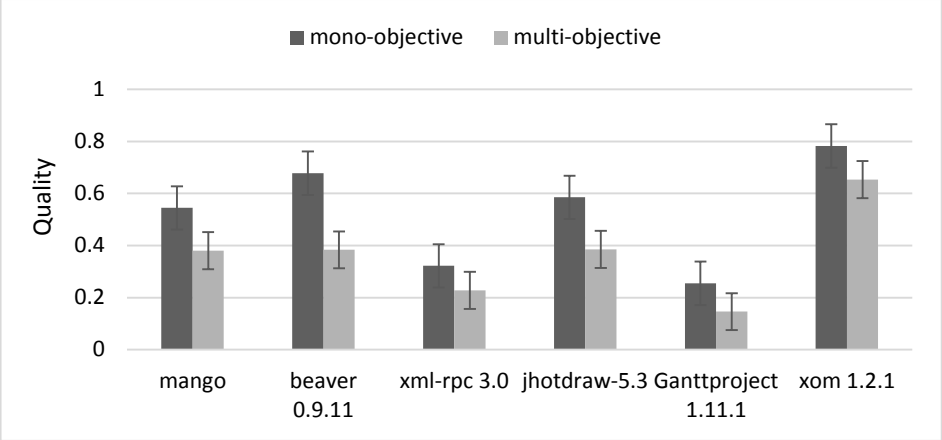

Fig. 6. The results of Test 1 based on the quality

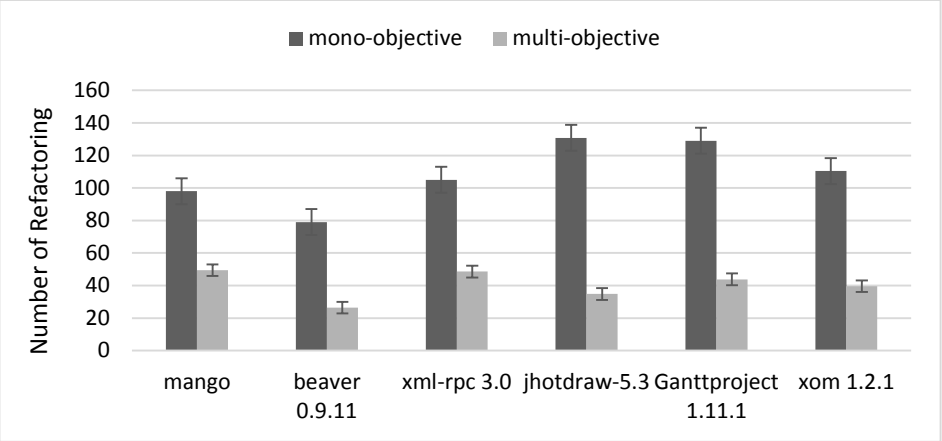

Fig. 7. The results of Test 1 based on the number of refactoring

4, the quality fitness function does not necessarily increase with the increasing refactoring number. Different refactoring sequences have different effects on the quality fitness function.

Discussion: According to Figures 6 and 7, for GanttProject input, more refactoring is suggested, and the lower quality fitness function is obtained. This can be attributed to the software structure under consideration or the type of applicable refactoring. Generally, these two figures show that the quality function fits more by increasing the number of refactoring.

Figure 7 indicates that the mean number of refactoring for multi-objective algorithms is lower than that for mono-objective algorithms.

Figure 8 presents the runtime of the mono and multi-objective algorithms. The unit of runtime in this figure is minute (m). Generally, runtime has increased due to the number of classes in the program. As shown in this figure, the runtime for the mono-objective algorithm is longer for all inputs than for the multi-objective algorithm. Since the number of applicable refactorings is significantly more significant in the mono-objective approach than in the multi-objective approach, it is natural to spend more time implementing the mono-objective algorithm.

4- 3- 2- Test 2: the comparison between UMOCell and NSGA-II based on runtime and quality function

It must be compared with other methods to evaluate the proposed method. Therefore, this section compares the refactoring method based on the UMOCell algorithm and NSGA-II based on quality function and runtime. Two fitness functions are measured to compare two multi-objective algorithms: positive and negative. The positive fitness function is the sum of the metrics with positive improvement, mainly CDS, NOH, ANA, DAM, CAM, Agg, FA, NPM, Abstractness, and Abstract ratio. The negative fitness function shows the sum of the metrics with negative improvement, mainly DCC, CIS, NOM, and WMC. The measurement of the sum of all metrics as an objective function regarding positive or negative improvement leads the improver to apply his preference on the improvement rate.

Consequently, we have proposed a multi-objective

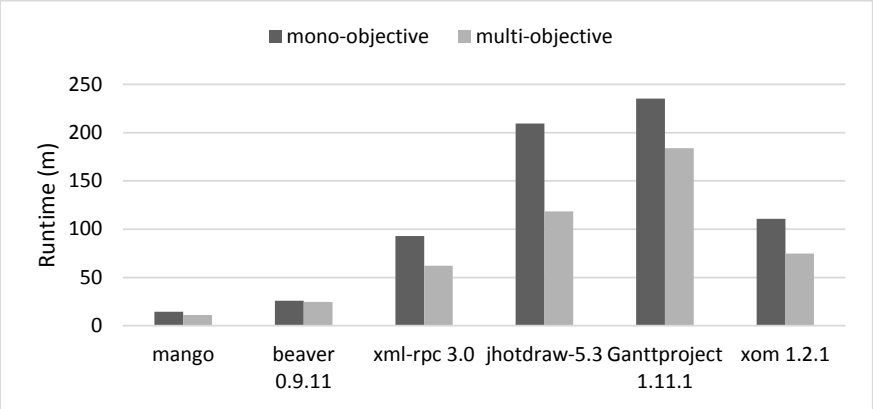

Fig. 8. The results of Test 1 based on the runtime (m)

Table 3. The results of Test 1 based on the proposed metrics

Run#	NSGA-II-positive	UMOCell-positive	NSGA-II-negative	UMOCell-negative
		json		
1	0.64693	1.182398	0.077675	0.385934
2	0.467005	1.226165	0.101982	0.58787
3	0.693031	1.182163	0.101476	0.040883
4	0.326917	1.343456	0.106152	0.700748
5	0.645893	1.598973	0.131038	0.178292
		beaver		
1	0.213142	0.964044	0.022562	0.16876
2	0.264097	0.863172	0.045591	0.200258
3	0.490644	0.921	0.06414	0.016637
4	0.217302	0.71617	0.038908	0.026182
5	0.442556	0.4644	0.018955	0.040277
		xml-rpc		
1	0.335702	1.033293	0.070596	0.127065
2	0.322416	0.69258	0.055543	0.182831
3	0.329173	0.882906	0.061149	0.140621
4	0.267743	0.994462	0.01342	0.050555
5	0.444467	0.807297	0.031711	0.121653
		JHotDraw		
1	0.471024	0.811985	0.014521	0.068899
2	0.572528	0.922504	0.028133	0.07562
3	0.327728	0.933846	0.018221	0.09732
4	0.350792	0.890966	0.015237	0.099403
5	0.484191	0.820966	0.017769	0.089403
		XOM		
1	0.653733	0.900388	0.0438	0.0763
2	0.679188	1.033102	0.038428	0.079897
3	0.60562	0.795776	0.037831	0.059233
4	0.688629	0.745144	0.041729	0.071314
5	0.595766	0.890966	0.020739	0.099403

algorithm. To measure positive and negative fitness functions and evaluate the UMOCell algorithm, we compare UMOCell and NSGA-II based on runtime and quality functions in this test. Like Test 1, each program runs for each algorithm 30 times. The average of these 30 times is reported for each program and algorithm. Table 4 reports the examples of running these algorithms for five programs and five times.

Figures 9 and 10 show the results of test 2 based on the quality function and the positive or negative one, respectively.

Discussion: Figures 9 and 10 show that the UMOCell algorithm performs better on average than the NSGA-II algorithm. This performance is due to the more significant

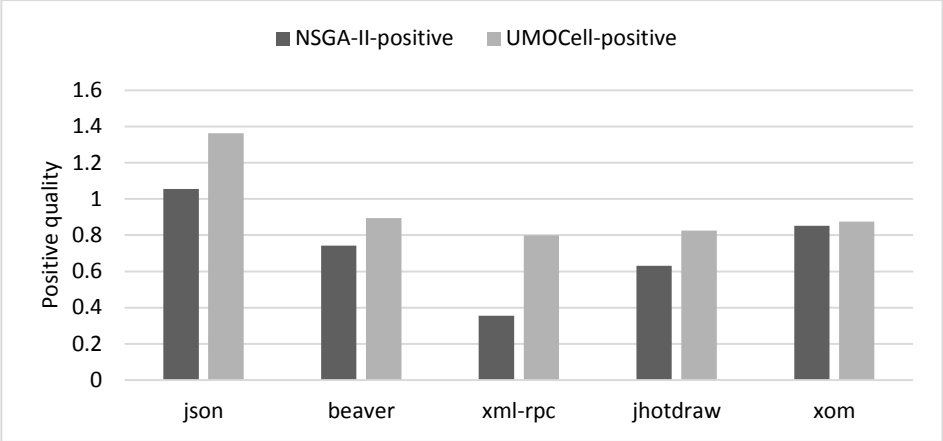

Fig. 9. The results of Test 2 based on the positive quality function

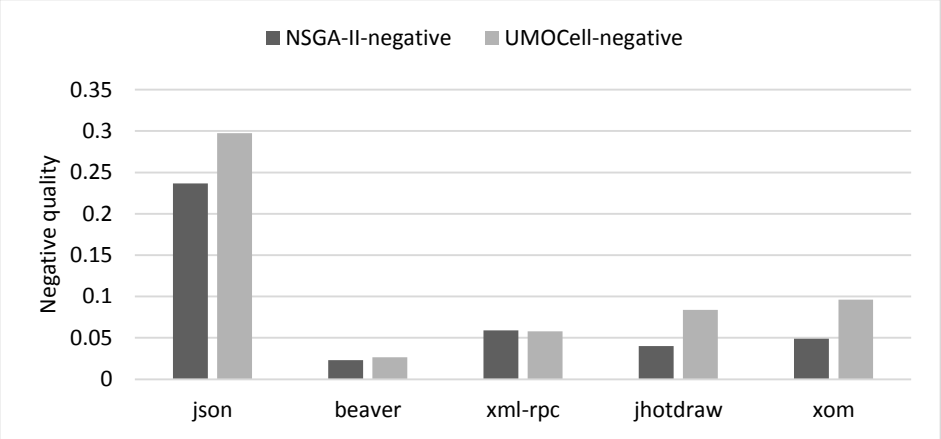

Fig. 10. The results of Test 2 based on the negative quality function

number of the proposed refactorings in the UMOCell algorithm than in NSGA-II.

Figure 11 illustrates the results of Test 2 based on runtime. The unit of runtime in this figure is minute (m).

Discussion: As shown in Figure 11, the runtime in UMOCell is more than that in NSGA-II. Contrary to NSGA-II, the UMOCell algorithm performs more slowly in selecting the solution and conducts dominance evaluation to add the solution to the optimal set at each step. It leads the proposed algorithm to have more runtime.

5- Conclusion

Refactoring is to improve the design and internal structure of the software while maintaining its external behavior and quality. Proposing a fast and accurate refactoring method is the main challenge in this field. Hence, researchers have formulated refactoring as an optimization problem and use search-based techniques. This research proposes a refactoring method based on multi-objective algorithms called RMMOC to develop search-based refactoring. In addition, a metric called MPC has been added to the system to measure the

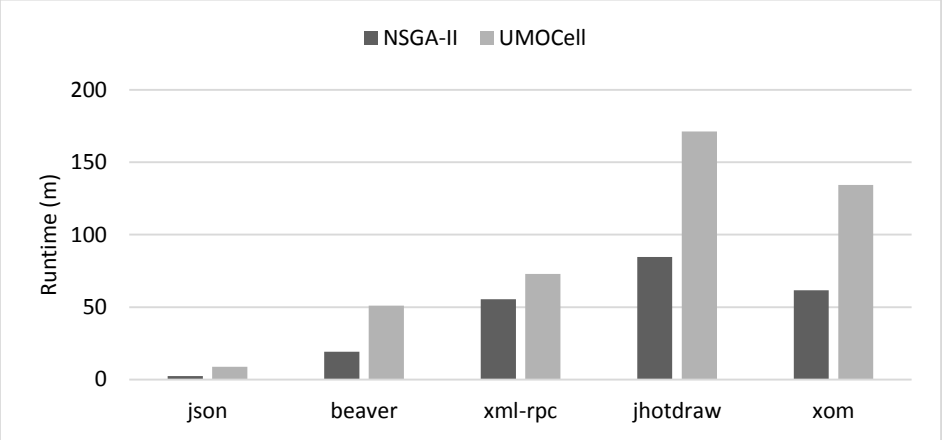

Fig. 11. The results of test 2 based on runtime (m)

coupling metrics used by the system more accurately. The number of refactorings metric has also been proposed to reduce software deviation from the original design as a secondary novelty. The experiments' results show that the proposed method's performance is remarkable and that using new metrics is effective.

References

[1] B. Bafandeh Mayvan, A. Rasoolzadegan, and A. Javan Jafari, "Bad smell detection using quality metrics and refactoring opportunities," J. Softw. Evol. Process, vol. 32, no. 8, p. e2255, 2020.

[2] M. Akour, M. Alenezi, and H. Alsghaier, "Software refactoring prediction using SVM and optimization algorithms," Processes, vol. 10, no. 8, p. 1611, 2022.

[3] P. Tripathy and K. Naik, Software evolution and maintenance: a practitioner's approach. John Wiley & Sons, 2014.

[4] R. C. Martin, Agile software development: principles, patterns, and practices. Prentice Hall, 2002.

[5] R. Alsarraj and others, "Refactoring for software maintenance: A Review of the literature," J. Educ. Sci., vol. 30, no. 1, pp. 89–102, 2021.

[6] S. M. Akhtar, M. Nazir, A. Ali, A. S. Khan, M. Atif, and M. Naseer, "A Systematic Literature Review on Software-refactoring Techniques, Challenges, and Practices," 2022.

[7] H. Ahmadi, M. Ashtiani, M. A. Azgomi, and R. Saheb-Nassagh, "A DQN-based agent for automatic software refactoring," Inf. Softw. Technol., vol. 147, p. 106893, 2022.

[8] M. Fowler, K. Beck, and W. R. Opdyke, "Refactoring: Improving the design of existing code," in 11th European Conference. Jyväskylä, Finland, 1997.

[9] H. Khosravi and A. Rasoolzadegan, "A Meta-Learning Approach for Software Refactoring," arXiv Prepr. arXiv2301.08061, 2023.

[10] W. F. Opdyke, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in Proc. SOOPPA'90: Symposium on Object-Oriented Programming Emphasizing Practical Applications, 1990.

[11] B. Du Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman, "Does god class decomposition affect comprehensibility?," in IASTED Conf. on Software Engineering, 2006, pp. 346–355.

[12] R. Morales, F. Chicano, F. Khomh, and G. Antoniol, "Exact search-space size for the refactoring scheduling problem," Autom. Softw. Eng., vol. 25, no. 2, pp. 195–200, 2018.

[13] T. Mariani and S. R. Vergilio, "A systematic review on search-based refactoring," Inf. Softw. Technol., vol. 83, pp. 14–34, 2017.

[14] M. O'Keeffe and M. O. Cinnéide, "A stochastic approach to automated design improvement," in ACM International Conference Proceeding Series, 2003, vol. 42, pp. 59–62.

[15] V. Cortellessa, D. Di Pompeo, V. Stoico, and M. Tucci, "Many-objective optimization of non-functional attributes based on refactoring of software models," Inf. Softw. Technol., vol. 157, p. 107159, 2023.

[16] D. Heuzeroth, U. Aßmann, M. Trifu, and V. Kuttruff, "The COMPOST, COMPASS, Inject/J and RECODER tool suite for invasive software composition: Invasive

composition with COMPASS aspect-oriented connectors," in International Summer School on Generative and Transformational Techniques in Software Engineering, 2005, pp. 357–377.

[17] M. Mohan, D. Greer, and P. McMullan, "Technical debt reduction using search based automated refactoring," J. Syst. Softw., vol. 120, pp. 183–194, 2016.

[18] Z. Razani and M. Keyvanpour, "SBSR Solution Evaluation: Methods and Challenges Classification," in 2019 5th Conference on Knowledge Based Engineering and Innovation (KBEI), 2019, pp. 181–188.

[19] N. Shafiei and M. R. Keyvanpour, "Challenges Classification in Search-Based Refactoring," in 2020 6th International Conference on Web Research (ICWR), 2020, pp. 106–112.

[20] D. C. Schmidt, "Model-driven engineering," Comput. Comput. Soc., vol. 39, no. 2, p. 25, 2006.

[21] A. Ghannem, G. El Boussaidi, and M. Kessentini, "Model refactoring using examples: a search-based approach," J. Softw. Evol. Process, vol. 26, no. 7, pp. 692–713, 2014.

[22] A. Ghannem, G. El Boussaidi, and M. Kessentini, "Model refactoring using interactive genetic algorithm," in International Symposium on Search Based Software Engineering, 2013, pp. 96–110.

[23] A. Ghannem, M. Kessentini, M. S. Hamdi, and G. El Boussaidi, "Model refactoring by example: A multi-objective search based software engineering approach," J. Softw. Evol. Process, vol. 30, no. 4, p. e1916, 2018.

[24] U. Mansoor, M. Kessentini, M. Wimmer, and K. Deb, "Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm," Softw. Qual. J., vol. 25, no. 2, pp. 473–501, 2017.

[25] A. A. B. Baqais and M. Alshayeb, "Sequence diagram refactoring using single and hybridized algorithms," PLoS One, vol. 13, no. 8, p. e0202629, 2018.

[26] B. Alkhazi, T. Ruas, M. Kessentini, M. Wimmer, and W. I. Grosky, "Automated refactoring of ATL model transformations: a search-based approach," in Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, 2016, pp. 295–304.

[27] M. Hentati, A. Trabelsi, L. Ben Ammar, and A. Mahfoudhi, "MoTUO: An Approach for Optimizing Usability Within Model Transformations," Arab. J. Sci. Eng., vol. 44, no. 4, pp. 3253–3269, 2019.

[28] M. Paixão et al., "Behind the intents: An in-depth empirical study on software refactoring in modern code review," in Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 125–136.

[29] E. Fernandes et al., "Refactoring effect on internal quality attributes: What haven't they told you yet?," Inf. Softw. Technol., vol. 126, p. 106347, 2020.

[30] M. Mohan and D. Greer, "A survey of search-based refactoring for software maintenance," J. Softw. Eng. Res. Dev., vol. 6, no. 1, pp. 1–52, 2018.

[31] R. Morales, F. Chicano, F. Khomh, and G. Antoniol, "Efficient refactoring scheduling based on partial order reduction," J. Syst. Softw., vol. 2, no. 5, pp. 25–51, 2018.

[32] H. Liu, G. Li, Z. Y. Ma, and W. Z. Shao, "Conflict-aware schedule of software refactorings," IET Softw., vol. 2, no. 5, pp. 446–460, 2008.

[33] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai, "An Interactive and Dynamic Search-Based Approach to Software Refactoring Recommendations," IEEE Trans. Softw. Eng., 2018.

[34] A. Ouni, M. Kessentini, M. Ó Cinnéide, H. Sahraoui, K. Deb, and K. Inoue, "MORE: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells," J. Softw. Evol. Process, vol. 29, no. 5, p. e1843, 2017.

[35] M. Kessentini, T. J. Dea, and A. Ouni, "A context-based refactoring recommendation approach using simulated annealing: two industrial case studies," in Proceedings of the Genetic and Evolutionary Computation Conference, 2017, pp. 1303–1310.

[36] A.-R. Han and S. Cha, "Two-Phase Assessment Approach to Improve the Efficiency of Refactoring Identification," IEEE Trans. Softw. Eng., vol. 44, no. 10, pp. 1001–1023, 2017.

[37] M. Mohan and D. Greer, "MultiRefactor: automated refactoring to improve software quality," in International Conference on Product-Focused Software Process Improvement, 2017, pp. 556–572.

[38] I. Griffith, S. Wahl, and C. Izurieta, "TrueRefactor: An automated refactoring tool to improve legacy system and application comprehensibility," in 24th International Conference on Computer Applications in Industry and Engineering, ISCA 2011, 2011.

[39] I. H. Moghadam, "Multi-level automated refactoring using design exploration," in International Symposium on Search Based Software Engineering, 2011, pp. 70–75.

[40] V. Alizadeh and M. Kessentini, "Reducing interactive refactoring effort via clustering-based multi-objective search," in 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2018, pp. 464–474.

[41] I. H. Moghadam and M. Ó Cinnéide, "Code-Imp: A tool for automated search-based refactoring," in Proceedings of the 4th Workshop on Refactoring Tools, 2011, pp. 41–44.

[42] I. H. Moghadam and M. O. Cinnéide, "Resolving conflict and dependency in refactoring to a desired design," e-Informatica Softw. Eng. J., vol. 9, no. 1, 2015.

[43] M. O'Keeffe and M. Ó. Cinnéide, "Search-based software maintenance," in Conference on software maintenance and reengineering (CSMR'06), 2006, pp.

10--pp.

[44] M. O'Keeffe and M. O. Cinnéide, "Search-based refactoring for software maintenance," J. Syst. Softw., vol. 81, no. 4, pp. 502–516, 2008.

[45] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," IEEE Softw., vol. 25, no. 5, pp. 38–44, 2008.

[46] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, "On the use of developers' context for automatic refactoring of software anti-patterns," J. Syst. Softw., vol. 128, pp. 236–251, 2017.

[47] M. W. Mkaouer, M. Kessentini, M. Ó. Cinnéide, S. Hayashi, and K. Deb, "A robust multi-objective approach to balance severity and importance of refactoring opportunities," Empir. Softw. Eng., vol. 22, no. 2, pp. 894–927, 2017.

[48] H. Wang, M. Kessentini, and A. Ouni, "Interactive refactoring of web service interfaces using computational search," IEEE Trans. Serv. Comput., 2017.

[49] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study," ACM Trans. Softw. Eng. Methodol., vol. 25, no. 3, pp. 1–53, 2016.

[50] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," IEEE Trans. Softw. Eng., vol. 38, no. 1, pp. 5–18, 2011.

[51] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in Proceedings of the 9th annual conference on Genetic and evolutionary computation, 2007, pp. 1106–1113.

[52] 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, 2017.

[53] O. Seng, J. Stammel, and 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, 2006, pp. 1909–1916.

[54] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó. Cinnéide, and K. Deb, "On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach," Empir. Softw. Eng., vol. 21, no. 6, pp. 2503–2545, 2016.

[55] A. L. Jaimes, C. A. C. Coello, and J. E. U. Barrientos, "Online objective reduction to deal with many-objective problems," in International Conference on Evolutionary Multi-Criterion Optimization, 2009, pp. 423–437.

[56] H. Ishibuchi, N. Tsukamoto, and Y. Nojima, "Evolutionary many-objective optimization: A short review," in 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence), 2008, pp. 2419–2426.

[57] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "High dimensional search-based software engineering: finding tradeoffs among 15 objectives for automating software refactoring using NSGA-III," in Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, 2014, pp. 1263–1270.

[58] D. Meignan, S. Knust, J.-M. Frayret, G. Pesant, and N. Gaud, "A review and taxonomy of interactive optimization methods in operations research," ACM Trans. Interact. Intell. Syst., vol. 5, no. 3, pp. 1–43, 2015.

[59] A. Ramirez, J. R. Romero, and C. L. Simons, "A systematic review of interaction in search-based software engineering," IEEE Trans. Softw. Eng., vol. 45, no. 8, pp. 760–781, 2018.

[60] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," IEEE Trans. Evol. Comput., vol. 6, no. 2, pp. 182–197, 2002.

[61] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba, "Mocell: A cellular genetic algorithm for multiobjective optimization," Int. J. Intell. Syst., vol. 24, no. 7, pp. 726–746, 2009.

[62] P. K. Goyal and G. Joshi, "QMOOD metric sets to assess quality of Java program," in 2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT), 2014, pp. 520–533.

[63] D. Boshnakoska and A. Mišev, "Correlation between object-oriented metrics and refactoring," in International Conference on ICT Innovations, 2010, pp. 226–235.

[64] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," J. Syst. Softw., vol. 23, no. 2, pp. 111–122, 1993.

[65] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), 2015, pp. 1–12.

[66] G. Antoniol, M. Di Penta, and M. Harman, "Search-based techniques applied to optimization of project planning for a massive maintenance project," in 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 240–249.

[67] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," Autom. Softw. Eng., vol. 20, no. 1, pp. 47–79, 2013.

[68] A. Ouni, M. Kessentini, S. Bechikh, and H. Sahraoui, "Prioritizing code-smells correction tasks using chemical reaction optimization," Softw. Qual. J., vol. 23, no. 2, pp. 323–361, 2015.

[69] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "Earmo: an energy-aware refactoring approach for mobile apps," IEEE Trans. Softw. Eng., vol. 44, no. 12, pp. 1176–1206, 2017.

HOW TO CITE THIS ARTICLE
M. R. Keyvanpour, Z. Karimi Zandian, Z. Razani, RMMOC: Refactoring Method based on Multi-Objective Algorithms and New Criteria, AUT J. Model. Simul., 55(2) (2023) 227-242.

DOI: 10.22060/miscj.2024.22381.5324