DEPICTER: A Design-Principle Guided and Heuristic-Rule Constrained Software Refactoring Approach

Yangyang Zhao, Yibiao Yang, Yuming Zhou, and Zuohua Ding, *Associate Member, IEEE*

***Abstract*—Software refactoring is one of the most significant practices in software maintenance as the quality of software design tends to deteriorate during software evolution. But, refactoring software is a very challenging task as it requires a holistic view of the entire software system. To this end, recent studies introduced search-based algorithms to facilitate software refactoring. However, they still have the following major limitations: 1) the searched solutions may violate the design principles as their fitness functions do not directly reflect the degree of software's compliance with design principles; 2) most approaches start the searching process from a completely random initial population, which may lead to unoptimal solutions. In this article, we aim to develop effective search-based refactoring approach to recommend better refactoring activities for developers which can improve the degree of software's compliance with design principles as well as the software design quality. We propose `DEPICTER`, a design-principle guided and heuristic-rule constrained software refactoring recommendation approach. In particular, `DEPICTER` uses non-dominated sorting genetic algorithm (NSGA)-II genetic algorithm and employs design-principle metrics as fitness functions. Besides, `DEPICTER` leverages heuristic rules to improve the quality of initial population for subsequent generic evolution. Our evaluations, based on four widely used systems, show that `DEPICTER` is effective for guiding the development of better refactoring models in practice.**

***Index Terms*—Design principles, heuristic rule, search-based, software maintenance, software refactoring.**

I. Introduction

FOR object-oriented software development, especially large-scale system, the quality of software design has a great impact on software quality [1]. The software that follows the software design principles tends to have higher scalability, readability, and maintainability [1]. However, in practice, the quality of software design often decreases gradually with the evolution of software [2]. As software evolves, developers often need to make changes to software for fixing bugs and adding new features, which lead to architectural drift and erosion, hence increasing the deviation from the initial design and growing difficulty for system maintenance [3]. To this end, developers often turn to refactor software. Software refactoring aims to improve the internal structure of software, without changing software functions and external behavior, so as to optimize the software design and improve the maintainability of code. Since manual refactoring is time-consuming and error-prone [4], many approaches have been proposed to help developers refactor more efficiently and effectively.

The search-based refactoring method is one of the most promising approaches to solve the software refactoring problems. Different from other traditional refactoring methods, the search-based refactoring method converts the refactoring problem into an optimization problem and uses the search algorithm to find the optimal or nearly optimal refactoring solution in the huge search space. Search-based software refactoring makes use of the advantages of search algorithms, which can avoid some deviations that may be caused by human intuition, such that some unexpected feasible solutions can be constructed. Although many search-based software refactoring approaches have been proposed in previous studies and significant progress has been made, they still have the following limitations.

1) *Their fitness functions for refactoring do not directly measure the degree of software's compliance with design principles:* Generally speaking, the more the software follows the design principles, the higher the quality [1]. In other words, if the refactoring fitness functions in search algorithms do not directly evaluate the degree of software's compliance with design principles, the optimal solutions obtained through searching may not be able to significantly improve the software design quality and may even violate the design principles. However, to our best knowledge, no prior work comprehensively takes design principles into account when refactoring software.
2) *Most approaches start the searching process from a completely random initial population, which may lead to unoptimal solutions:* A completely random initial population may have some unreasonable refactoring operations, such as moving a class to a package without any association. Generally speaking, it is more likely to generate better offspring from high-quality ancestor [5]. In practice, these

Manuscript received December 14, 2021; revised February 15, 2022; accepted March 5, 2022. Date of publication April 22, 2022; date of current version June 2, 2022. This work was supported by the National Natural Science Foundation of China under Grants 62132014, 61772259 and 62072194. Associate Editor: H. Jiang. *(Corresponding author: Yangyang Zhao.)*

Yangyang Zhao and Zuohua Ding are with Zhejiang Sci-Tech University, Hangzhou 314423, China (e-mail: yangyangzhao@zstu.edu.cn; zouhuading@hotmail.com).

Yibiao Yang and Yuming Zhou are with Nanjing University, Nanjing 210023, China (e-mail: yangyibiao@nju.edu.cn; zhouyuming@nju.edu.cn).

This article has supplementary material provided by the authors and color versions of one or more figures available at https://doi.org/10.1109/TR.2022.3159851.

Digital Object Identifier 10.1109/TR.2022.3159851

approaches often set a maximum number of generations to terminate the searching process due to limited resources. Starting from a low-quality initial population, it is difficult to search a good solution in limited generations. To this end, the quality of the initial population is critical for quickly finding good solutions in search-based software refactoring.

To mitigate the above limitations, in this article, we, therefore, propose a multiobjective optimization software refactoring approach (called DEPICTER), which employs the non-dominated sorting genetic algorithm (NSGA)-II genetic algorithm, using design-principle metrics as fitness functions and leveraging heuristic rules as initialization constraints. To measure the degree of software's compliance with design principles, we select seven design-principle metrics [including module interaction index (MII), non-API method closedness index (NC), inheritance based inter-module coupling index (IC), state access violation index (SAVI), size uniformity index (CUL), common use (CU), and component cohesion (CC); see Section III-C for details] as the optimization fitness functions. In addition, to improve the effectiveness of software refactoring, we define a set of rules to guide the population initialization and ensure that the generated refactoring operations are feasible. The purpose of DEPICTER is to recommend good refactoring solutions for developers to improve the degree of software's compliance with design principles as well as the software design quality.

In order to evaluate DEPICTER, we first investigate whether DEPICTER can significantly improve the software design quality and the degree of compliance with design principles, and whether DEPICTER is useful from developers' perspectives. Then we build two baselines to examine whether DEPICTER can perform better than the existing remodularization approach proposed by Mkaouer *et al.* [6] and the common approach built using most widely used objectives. In addition, to evaluate the value of design-principle metrics and heuristic rules, we further study the influence of using design-principle metrics on refactoring performance compared with using traditional metrics and analyze the impact of the heuristic rules as initialization constraints on the refactoring effectiveness of DEPICTER.

Based on four widely used systems, the experimental results show that DEPICTER has positive effect on both the software design quality and the degree of compliance with design principles, with significant improvements on over six out of eight indicators and at least six out of seven fitness values. DEPICTER is considered useful from developers' perspectives, with a high usefulness degree of more than 0.75. In addition, our approach outperforms the existing remodularization approach and the common approach in terms of most evaluation indicators. And, as expected, the design-principle metrics are able to significantly improve the software design quality better than the traditional metrics, with more than five wins on the average %improvements. Besides, the defined heuristic rules have positive influence on the recommendation results of DEPICTER on more than half of indicators. These experimental results are critically important to help both researchers and practitioners understand whether DEPICTER is effective on software refactoring. We believe that they can guide the development of better refactoring models in practice.

Contribution: Following are the main contributions of this article.

1) *Novelty:* We comprehensively leverage design-principle metrics to guide searching solutions in search-based software refactoring. Besides, we adopt heuristic rule constraints to obtain high-quality initial population for subsequent generic evolution.
2) *Originality:* We propose a novel search-based software refactoring approach DEPICTER which uses design principle metrics as the fitness functions and adopts heuristic rule constraints to ensure that all the individuals in the initial population are more effective.
3) *Evaluation:* We conduct a detailed experiment to evaluate DEPICTER. Our results, based on four widely used Java systems, show that design-principle metrics are more effective for improving the design quality. Furthermore, the heuristic rules play a positive role in generating high-quality initial population and further obtaining more effective solutions.

Article outline:

The rest of this article is organized as follows. Section II introduces the background and motivation. Section III is dedicated to the proposed DEPICTER approach, including the overall framework of DEPICTER, the individual composition, the used fitness functions, the defined heuristic rules as constraints, and the evolution process. Section IV describes the experimental setup of our study, including the research questions, the experimental parameters, the evaluation indicators, the validation method, and the studied systems. Section V reports the evaluation results of our approach and presents experimental results in detail for each research question. Section VI analyzes the threats to the validity of our study, followed by the introduction of related works in Section VII. Finally, Section VIII concludes this article.

II. Background and Motivation

A. Background

In the process of software development or maintenance, there will always be some unscheduled code changes that need to be made, like incorporation of new functional requirements, bug fixes, and adaptation for environmental migration. During the code changing process, the inexperience of developers or the time/budget pressure will lead to the ignorance of design principles, which causes the defilement of the initial software design, and further decreases the software design quality. Poor software design quality may lead to a high software maintenance cost [7], [8].

One of the most widely used activities to reduce the maintenance cost is software refactoring. Software refactoring changes the internal structure of code without altering the software function and external behavior. It is widely used for improving the quality of the software design and further reducing the cost of software maintenance. Despite the importance of refactoring, some developers try to avoid refactoring for some reasons [9]. The main reason is that manual refactorings are time-consuming

and error-prone. Due to tight schedules, developers often do not have enough time for manual refactoring. And 77% of developers also find it hard to perform manual refactorings correctly [4].

In order to facilitate developers refactoring software, many approaches have been proposed. One of the most promising approaches to solve the software refactoring problems is using search-based algorithms. The search-based refactoring approaches transform traditional software refactoring problems into search-based optimization problems. They can find refactoring solutions in the huge search space and avoid some deviations that may be caused by human intuition. Genetic algorithm is the most widely used search algorithm [10], [11], which is inspired by the theory of natural evolution. It reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation. Genetic algorithm remains one of the most widely used optimization algorithms in modern nonlinear optimization, especially NSGA-II. NSGA-II [12] is one of most popular genetic algorithms, which introduces the elite strategy, the density value estimation strategy, and the fast nondominated sorting strategy to overcome the deficiencies of NSGA. What is more, NSGA-II is one of the most famous multiobjective optimization algorithms, which is widely used to solve refactoring problems. It has been shown in recent reviews that NSGA-II is the most used algorithm for search-based refactoring [10], [11].

For better understanding of our study, the main concepts of genetic algorithm are listed as follows:

1) *Individual:* A solution to the problem you want to solve (in this study, we use individual and solution interchangeably, as both of them indicate a refactoring scheme, i.e., a sequence of operations, that can lead to a remodularized system).
2) *Population:* A set of individuals. The number of individuals in the population is the population size.
3) *Fitness function:* The standard for evaluating the quality of each individual. It is used to determine how fit an individual is (the ability of an individual to compete with other individuals).
4) *Selection:* Taking fitness values as the criterion, select the fittest individuals from population and let them pass their genes to the next generation.
5) *Crossover:* With the crossover probability, two individuals in the population exchange certain genes in a certain way to produce two new individuals.
6) *Mutation:* With mutation probability, some genes of the individuals are changed to obtain the mutated individuals.

B. Motivation

In search-based software refactoring, an appropriate fitness function is one of the most basic and necessary conditions for the refactoring effectiveness. The fitness function is used to evaluate the individual (i.e., solution), which determines how fit an individual is (the ability of an individual to compete with other individuals). In the search space, the fitness function can guide the direction of the search and find the area that meets the conditions. A good refactoring solution should not improve only one aspect of the attribute but should achieve a good balance among many attributes, especially for the conflicting attributes. In other words, software refactoring problems are multiobjective optimization problems [13]. Therefore, multiobjective optimization methods can be more effectively applied to the actual refactoring scenarios. In previous studies, the wildly used fitness functions are mainly quality model for object oriented design (QMOOD) metrics, which evaluate the results of software refactoring from the aspects of software flexibility, reusability, comprehensibility, etc. However, the fitness functions of existing work cannot directly measure the degree of software's compliance with design principles. Intuitively, the more object-oriented software follows the design principles, the higher the quality of the software design. Nowadays, many object-oriented metrics have been proposed to evaluate the software modularization [14], which can quantify the degree of compliance with design principles. To this end, it is potential to use the design-principle metrics as fitness functions for refactoring.

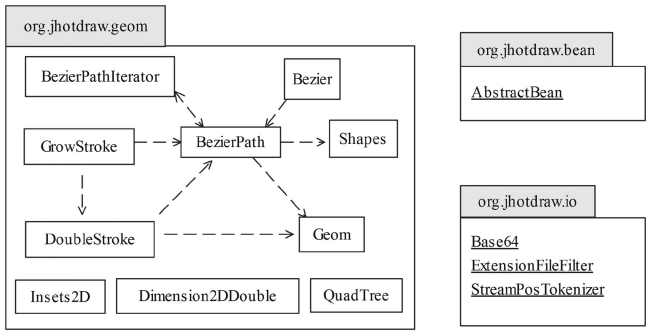

Fig. 1. Example from Jhotdraw v7.1.

TABLE I
EXAMPLE OF A RANDOMLY GENERATED SOLUTION

Operation	Operation type	Source	Entities	Target
Solution 1				
....	....	....	....	....
$oper_i$	Move class	*org.jhotdraw.geom*	*BezierPath*	*org.jhotdraw.bean*
$oper_{i+1}$	Move class	*org.jhotdraw.geom*	*BezierPath*	*org.jhotdraw.io*
....	....	....	....	....

Genetic algorithm is the most widely used algorithm in search-based software refactoring [10], [11], especially the NSGA-II algorithm. Genetic algorithms are commonly used to generate high-quality solutions to the search problems by relying on biologically inspired operators such as mutation, crossover, and selection, which reflects the process of evolution from the initial population [15]. An important part of genetic algorithm is the population initialization. In most existing studies, the initial population is randomly initialized. They randomly generate a refactoring operation sequence (i.e., a refactoring solution) in the large search space of source code entities, which may cause some operations to be nonfeasible. Fig. 1 shows an example with three packages from Jhotdraw v7.1, and Table I shows an example of a randomly generated solution. In this solution, $oper_i$ is to move class *BezierPath* in package

org.jhotdraw.geom to package *org.jhotdraw.bean*. Following oper_i, another randomly generated operation oper_{i+1} is to move *BezierPath* from *org.jhotdraw.geom* to *org.jhotdraw.io*. It is obvious that oper_{i+1} is nonexecutable since *BezierPath* is not a class in *org.jhotdraw.geom* anymore after the execution of oper_i.

Most approaches start the searching process from a completely random initial population. Since the search space is large, especially for large-scale systems, the completely random initialization process may lead to a low-quality initial population with some unreasonable refactoring operations. In the above example, Fig. 1 also illustrates the class dependencies of the three packages. As can be seen, *BezierPath* has a strong relationship with other classes in the package *org.jhotdraw.geom* but has no relationship with *org.jhotdraw.bean* or *org.jhotdraw.io*. In such a case, the operations in Table I to move *BezierPath* from *org.jhotdraw.geom* to *org.jhotdraw.bean* or *org.jhotdraw.io* are unreasonable according to the "high cohesion and low coupling" principle. Generally speaking, high-quality ancestors are more likely to generate better offspring [5]. The completely random initialization process may increase the difficulty of generating fitter offsprings. Besides, in practice, these approaches often set a maximum number of generations to terminate the searching process due to limited resources. Starting from a low-quality initial population, it is difficult to search a good solution in limited generations. To this end, the quality of initial population is critical for quickly finding good solutions in search-based software refactoring. Therefore, it is valuable to improve the quality of initial population for more effective evolution.

Therefore, in this study, we take the design principles in consideration for multiobjective software refactoring and explore how to improve the quality of the initial population by using heuristic rules.

III. DEPICTER: Design-Principle Guided and Heuristic-Rule Constrained Refactoring

A. Overall Framework

This study proposes a multiobjective optimization software refactoring approach DEPICTER (short for **DE**sign-**Pr**In**C**iple guided and heuris**T**ic-rul**E** constrained software **R**efactoring approach), which employs design-principle metrics as fitness functions and leverages heuristic rules as population initialization constraints. DEPICTER aims to recommend software refactoring solutions that can improve the degree to which the software follows the design principles as well as software design quality. The overall structural framework of DEPICTER approach is shown in Fig. 2. The following main parts are considered.

1) *Heuristic rules:* Define a set of heuristic rules to guide the population initialization.

2) *Initialization:* With the rule constraints, generate a set of individuals as the initial population and make each initial individual more reasonable.

3) *Fitness:* Use seven design-principle metrics (including MII, NC, IC, SAVI, CUL, CU, and CC) to measure the degree of software's compliance with the design principles for the newest system.

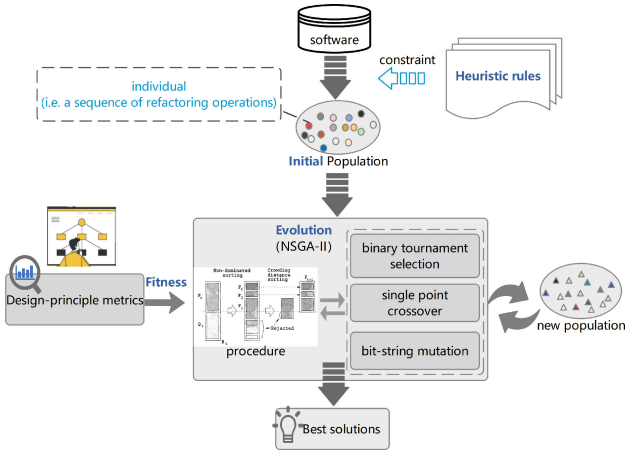

Fig. 2. Overall structural framework of DEPICTER approach.

DEPICTER algorithm based on NSGA-II

```
Input: software system, involved parameters, defined heuristic rules
Output: refactoring solution
P0 = initialize_population_with_rule_constraint() // use heuristic rules to constrain the initialization
Q0 = ∅ and t = 0
while not terminated() do:
    evaluate_fitness(Pt)              // evaluate each individuals in Pt using design-principle metrics
    Qt=make_new_population(Pt):       // generate new populations through selection, crossover
                                      // and mutation
          binary_tournament_selection
          single_point_crossover
          bit_string_mutation
    Rt=Pt ∪ Qt                        // Combine Pt and Qt to produce a combined population
    evaluate_fitness(Rt)              // evaluate each individuals in Rt using design-principle metrics
    F=fast-non-dominated-sort(Rt)     // Non-dominated sorting on Rt
    Pt+1=∅ and i=1                    // Define Pt+1 to store the next generation, i indicates the rank
    while | Pt+1 | + | Fi | ≤ N do:   // Until the number of individuals in Fi is greater than the
                                      // number of individuals required for Pt+1
        crowding-distance-assignment(Fi)   // Calculate crowding distance for individuals in Fi
        Pt+1=Pt+1 ∪ Fi                // Put individuals of Fi into Pt+1
        i=i+1
    end while
    sort(Fi, ≺n)                      // Sort individuals in Fi according to the crowding distance
    Pt+1=Pt+1 ∪ Fi[1:(N-| Pt+1 |)]    // Put the first (N-| Pt+1 |) individuals into Pt+1
    t++
end while
S=select_best_individuals(pt+1)
return S
```

Fig. 3. Algorithm of DEPICTER.

4) *Evolution:* Employ NSGA-II genetic algorithm to generate new populations.

Fig. 3 describes the algorithm of DEPICTER in detail. The main procedure is as follows:

1) Input the source code of the software, the parameters (described in Section IV-B), and the defined heuristic rules (described in Section III-D).
2) Leverage heuristic rules to restrict the initialization and generate the initial population P_0 with scale n.
3) Initialize t with a value of 0.
4) Evaluate the individuals in P_t using design-principle metrics.
5) The new population Q_t is generated through three basic operators on P_t, including selection (binary tournament selection), crossover (single point selection), and mutation (bit-string mutation).
6) Merge the parent population P_t with the child population Q_t to obtain a combined population R_t.
7) Evaluate the individuals in R_t using design-principle metrics.

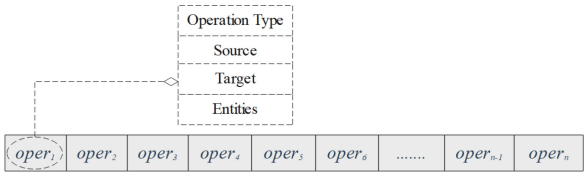

Fig. 4. Composition of individual.

8) Perform the nondominated sorting on R_t and calculate the crowding degree of individuals in each nondominated rank.
9) Select n suitable individuals to form a new parent population P_{t+1} according to the nondominated relationship and individual crowding distance.
10) Repeat the evolution process from step 4) to step 9), until the conditions for search termination are met.
11) Output the best individuals in the latest population.

More details about DEPICTER will be introduced in the following sections, including individual composition in Section III-B, fitness functions in Section III-C, heuristic rules as constraints in Section III-D, and the evolution process in Section III-E.

B. Individual Composition

In traditional genetic algorithms, a solution to the optimization problem (i.e., an individual) is often expressed as a sequence of variables, called a chromosome. The simplest genetic algorithm represents the chromosome as a string of digits, such as 0110001110. In order to apply the genetic algorithm to the software refactoring problem, we first need to modify the individual to adapt the refactoring problem. In this study, an individual is represented as a candidate refactoring solution, which is composed of a sequence of refactoring operations, namely [oper$_1$, oper$_2$, oper$_3$, ..., oper$_n$], as shown in Fig. 4. The information recorded by each operation includes operation type, source, target, and entities to refactor.

In our study, we attempt to remodularize the code from high level to improve software design quality and use design-principle metrics as fitness functions (see Section III-C for details), which are in module level. To this end, it may not be able to significantly improve the fitness values with a limited number of fine-grained operations. For this, we prefer the operations in coarse granularity. As shown in [11], the most widely used refactorings are the ones of the Fowler's catalog. Among the five mostly used refactorings in Fowler's catalog, we combine the top three (i.e., *pull up method*, *move method*, and *push down method*) into one category, i.e., *move method*, and also take *extract class* into consideration. In addition, we consider three other coarser grained operations, i.e., *merge package*, *extract package*, and *move class*, as Mkaouer *et al.* [6] did, which are also widely used in object-oriented remodularizations. Table II illustrates the five operations involved in our study, including *move class*, *merge packages*, *extract package*, *extract class*, and *move method*.

TABLE II
TYPES OF REFACTORING OPERATIONS

Operation type	Source	Target	Entities to refactor
Move class	Source package	Target package	A class in source package
Merge package	Source package	Target package	All entities in source package
Extract package	Source package	New package	A subset of classes in source package
Extract class	Source class	New class	A subset of methods in source class
Move method	Source class	Target class	A method in source class

TABLE III
INFORMATION OF THE SELECTED DESIGN-PRINCIPLE METRICS AS FITNESS FUNCTIONS

Fitness	Description	Design principle	source
CC	The percentage of directed relationships between intra-package classes	high cohesion	[16]
CU	The extent to which the classes in a package are used together by common clients	Single responsibility, high cohesion	[17]
MII	The extent to which all external calls made to a module are routed though the APIs of the module	Dependency inversion, low coupling	[14]
NC	The extent to which non-API public methods in a module are not called by other modules	Dependency inversion, low coupling	[14]
IC	The extent to which there are no inheritance-based inter-module dependencies between a module and other modules	composition/aggregation reuse,low coupling	[14]
SAVI	The extent to which the attributes defined in the classes in a module are not directly accessed by other classes	Reduce state access conflicts	[14]
CUL	The extent to which the classes in a module are different in terms of lines of code	Balanced scale	[14]

C. Fitness Functions

In the genetic algorithm, each individual will be evaluated based on the fitness functions. The fitness function is the only criterion for natural selection. The better the fitness, the higher the chance of being selected. The design of fitness functions should be specified according to the actual problem. This study aims to make the system follow the design principles more through refactoring and improve the design quality. To this end, the fitness function should be able to reflect the degree to which the system follows the design principles. Therefore, we select seven metrics as shown in Table III, which measure the degree of software's compliance with the design principles from different perspectives.

CC calculates the degree of interdependence between classes in a module, which is the ratio of the number of directed relations between classes in the package to the maximum possible number. CU measures the degree to which all classes in the module are dedicated to the same function. If a set of interfaces in a module is always used by other modules at the same time, it is considered that this set of interfaces has a common goal and can be regarded as a service. Therefore, CU describes the possibility that a module only provides one service to the outside. According to the definition of the single responsibility principle, a module is responsible for one responsibility, and CU indicates the degree to which the software complies with the single responsibility principle.

MII, NC, and IC describe the interaction between modules from different perspectives. Ideally, each module should use application programming interfaces (APIs) to provide good services to other modules and access others' resources through APIs. MII calculates the degree to which all external calls of a module are completed through the module's API. NC (non-API method closeness index) calculates the degree to which the non-API public methods of a module do not participate in the interaction between modules. Good design should avoid non-API methods in the module from being called by other modules. Dependency inversion principle is that the program should rely on the abstract interface rather than the concrete implementation. It can be seen that MII and NC are related to dependency inversion principle. Besides, IC (inheritance based intermodule coupling index) calculates the degree of noninherited coupling between modules. Good software design should carefully use the inheritance-based coupling relationship between modules. Principles of composition/aggregation reuse are to use composition/aggregation as much as possible and try not to use inheritance to achieve the purpose of reuse. Therefore, IC can reflect the extent to which the software follows the principle of composition/aggregation reuse.

In addition, SAVI measures the possibility of state access violation problems and calculates the degree to which attributes in a class are not directly accessed by classes in other modules. Because cross-module attribute access will reduce code quality, SAVI can reflect whether there is a status access conflict in the software. CUL (size uniformity index) calculates the degree of difference in the size (number of lines of code) of the classes in the module. If the scale of a certain class in the same module is much larger than other classes, it indicates that this class may not be sufficiently functional, and it is necessary to consider decomposing it into multiple smaller classes to optimize software design.

The above metrics measure the degree of software's compliance with design principles from different perspectives. DEPICTER uses these metrics as fitness functions to find the solutions that do the following:

1) maximize the percentage of directed relationships between intrapackage classes;
2) maximize the extent to which the classes in a package are used together by common clients;
3) maximize the extent to which all external calls made to a module are routed though the APIs of the module;
4) maximize the extent to which non-API public methods in a module are not called by other modules;
5) improve the extent to which there are no inheritance-based intermodule dependencies between a module and other modules;
6) improve the extent to which the attributes defined in the classes in a module are not directly accessed by other classes;
7) minimize the extent to which the classes in a module are different in terms of lines of code.

For each individual, after sequentially executing all the operations, these seven metrics will be re-calculated to evaluate how well the newly refactored system complies with the design principles.

D. Heuristic Rules as Constraints

Genetic algorithm first randomly generates a certain number of individuals to form the initial population, and the quality of the initial population has an important impact on the efficiency and effectiveness of subsequent evolution [5]. Generally speaking, appropriate manual intervention in this random initialization process is a way to improve the quality of the initial population. Therefore, in order to make each individual more reasonable, we define the following rules to restrict and guide the population initialization.

Move class: Move class c from package p_s to another package p_t. The selection of p_s, c, and p_t should satisfy the following conditions.

1) Randomly select a package as p_s that the number of classes in p_s is not less than two

$$\text{num}(C_{p_s}) \geq 2.$$

2) Randomly select a class as c from the classes in the selected p_s satisfying the following conditions. The number of noninheritance-related coupling relationships between c and p_s should not exceed half of the total number of classes in p_s, where c is a class of C_n. C_n is a collection of classes that have no inheritance or realization relationships with other classes in p_s. CBC measures the coupling between classes, indicating the number of noninheritance-related coupling relationships between this class c and other classes

$$c \in C_n, C_n \in C_{p_s}$$
$$\forall c_i \in C_n, \text{inheCoupling}(c_i, p_s) = 0$$
$$\text{CBC}(c, p_s) \leq \text{num}(C_{p_s})/2$$
$$\text{CBC}(c, p_s) = \sum_{c_i \in C(p_s), c_i \neq c} \text{isCoupled}(c, c_i).$$

3) With the selected c, then select a package p_t that meets the following condition: the number of noninheritance-related couplings between c and p_t is greater than the number of noninheritance-related couplings between c and p_s

$$\text{CBC}(c, p_s) < \text{CBC}(c, p_t).$$

In the above initialization rules of the move class operation, if one of them cannot be satisfied, it will fall back to the previous step and reselect the entities until all the selected entities meet the above rules.

Extract package: Extract a subset of classes C_m from package p_s to a new package p_{new}. The rules for selecting p_s and C_m are as follows: The number of classes in p_s is not less than the average number of classes in all packages; C_m is a collection of classes with a relatively high degree of relevance. In other word, for any c in C_m, there exits at least a class c' in C_m, which has

a direct or indirect dependency with c

$$\text{num}(C_{p_s}) \geq \text{avg}\left(\sum_{p_i \in P} \text{num}(C_{p_i})\right)$$

$$C_m \in C_{p_s}; \forall c \in C_m, \exists_{c' \in C_m, c' \neq c} \text{ withDependency } (c, c').$$

Merge package: Merge package p_1 and package p_2. The selection rules for p_1 and p_2 are as follows: The number of classes in p_1 is less than the average number of classes in all packages in the system, and the number of classes in p_2 is also less than the average number of classes in all packages in the system

$$\text{num}(C_{p_1}) < \text{avg}\left(\sum_{p_i \in P} \text{num}(C_{p_i})\right)$$

$$\text{num}(C_{p_2}) < \text{avg}\left(\sum_{p_i \in P} \text{num}(C_{p_i})\right).$$

Move method: Move method m in class c_1 to another class c_2. Particularly, we restrict that the move method operations can be only operated within packages. And, thus, the selection of c_1, m, and c_2 should satisfy the following rules.

1) The number of classes in package p should not be less than two; c_1 and c_2 are in the same package p

$$\text{num}(C_p) \geq 2, c_1 \in C_p, c_2 \in C_p.$$

2) The number of methods in class c_1 should be more than or equal to two

$$\text{num}(M_{c_1}) \geq 2.$$

3) Randomly select a method as m from the methods satisfying the following conditions in c_1: the NR between m and c_1 is less than the NR between m and c_2, where NR measures the relatedness among methods. NR(m, c_1) counts the number of the relatedness between m and the other methods in c_1, and NR(m, c_2) counts the number of the relatedness between m and the methods in c_2

$$\text{NR}(m, c_1) < \text{NR}(m, c_2)$$

$$\text{NR}(m, c) = \sum_{m_i \in M(c), m_i \neq m} (\text{calls}(m, m_i) \vee \text{callby}(m, m_i)).$$

Extract class: Extract the subset methods M_n in class c to a new class c_{new}. The selection rules are as follows: The number of methods in c is greater than the average number of methods for all classes; M_n is a collection of methods with a relatively high degree of relevance. That is, for any m in M_n, there exits at least a method m' in M_n, which is directly or indirectly calling m or called by m

$$\text{num}(M_c) \geq \text{avg}\left(\sum_{c_i \in C} M_{c_i}\right)$$

$$M_n \in M_c$$

$$\forall m \in M_{\text{n}}, \exists_{m' \in M_n, m' \neq m} (\text{calls}(m, m') \vee \text{callby}(m, m')).$$

In order to apply genetic algorithm to the problems of software refactoring, as described before, we represent the individual as a candidate refactoring solution, which is composed of a sequence of refactoring operations. In order to evaluate the solutions, it is necessary to execute the operations in sequence and then evaluate the changes in system quality after all operations are completed. Each operation should be executed in the new state of the system after all previous operations have been finished. If each operation in an individual is initialized independently in the original state of the system (i.e., the initial state before refactoring), some operations may not be executable. If the involved entity in a operation to be executed has been deleted or changed in the previous finished operations, the preconditions of this operation will not be met and cannot be executed. As an example, there are two operations oper_1 and oper_2 in the refactoring operation sequence. oper_1 is to merge package p_1 with package p_2, and then all entities in p_1 will be moved to p_2, and p_1 will be deleted; oper_2 is to move a class c in package p_1 to package p_3. The precondition for oper_2 to be executable is that p_1 and p_3 are two packages in the system and exist, and c is a class in p_1. It can be seen that after the execution of oper_1, there is no package p_1 in the system, which makes oper_2 not executable. Therefore, in order to ensure that all operations in each initial individual can be executed, we generate the next operation according to the new state of the system after the previous operation is completed. In the above example, with oper_1 generated, oper_1 will be executed in the following way: First, all classes in p_1 are moved to p_2, and then package p_1 is removed. After execution, the system status is updated. In the current new state of the system, oper_2 is initialized according to the rules defined above, so that the preconditions of oper_2 can be established, thus ensuring the executable of oper_2.

The reason why the above rules are loosely defined is to prevent using only the most suitable chromosomes to generate initial population. Moderate rules can enable the search pace as large as possible, avoid losing the useful information, and help generate more reasonable solutions. Through the definition of heuristic rules, there are certain constraints on population initialization, which can improve the quality of the population.

E. Evolution Process

In the genetic algorithm, the evolution of the population imitates the natural evolution model, generating progeny populations through genetic operators (selection, crossover, and mutation). For these three genetic operators, the processing methods in this study are as follows.

1) *Selection:* The purpose of selection is to allow good genes to be passed on to the next generation directly or after processing. The selection operator is based on the assessment of the fitness values of individuals in the population. Common selection strategies include roulette algorithm and tournament algorithm. We use the binary tournament algorithm. The main process of the tournament algorithm

is as follows: Randomly select k ($k < N$) individuals from N population and select the best fitness individual from these k individuals; repeat this process until N individuals are selected. After that, subsequent operations such as crossover and mutation will be performed among the N individuals to produce a new population.

2) *Crossover:* Crossover is the recombination of the genes of two parent chromosomes. In the process of generating the next generation of individuals, it is possible to combine the dominant genes of the two parent individuals through crossover to generate a new offspring with higher fitness. Commonly used techniques for crossover include single-point crossover, multipoint crossover, and uniform crossover. Among them, single-point crossover is currently the most used crossover algorithm. First, a crossover point is randomly selected on the chromosome, and then the two parents are exchanged and recombined according to the crossover point. In this study, we use single point crossover. In the process of multiobjective optimization, the generated offspring individuals are better to be closer to the parent individuals in order to achieve a more efficient search. Therefore, we use the trisection as the crossing boundary. The specific process is as follows: For two different individuals ind_1 and ind_2, suppose that the number of operations in ind_1 is less; so randomly select a position as the intersection point at the first 1/3 or the back 1/3 of ind_1, and then exchange the operations before the intersection point of ind_1 and ind_2 to generate two new operation sequences. More details are shown in the Appendix.

3) *Mutation:* Mutation simulates the genetic mutations caused by various accidental factors in the natural environment and randomly changes one or some genes in the chromosome with a small probability. Its purpose is to maintain and improve the diversity of gene types in the population and to avoid letting the chromosomes in the population get too close and fall into a local solution. In the binary coding system, the chromosome can be represented as a string of digits, such as 0110001110. For such a case, mutation means randomly changing a certain gene in the chromosome from 1 to 0 or from 0 to 1. For the software refactoring problem studied in this study, the individual refers to a sequence of refactoring operations, and each gene in the individual is represented as a refactoring operation. Therefore, mutation needs to be achieved by changing a certain refactoring operation. We use bit string mutation in this study. The specific method is as follows: First, determine whether to mutate according to the mutation probability; if to mutate, randomly select a mutation point; then execute all operations before the mutation point in sequence to update the state of the system; then, according to the current new system state, a new refactoring operation is randomly generated based on the preconditions described in Section III-D to ensure the operation executable; finally, the operation at the mutation point will be replaced with the newly generated operation. Please refer to the Appendix for more details.

As mentioned in Section III-D, during the individual initialization process, after the previous operation is completed, the next operation will be generated according to the new state of the system, thus ensuring that all operations of each individual can be executed. However, in the process of evolution, not all operations in the offspring individuals produced after selection, crossover, and mutation can be performed. For example, after crossover on two executable sequences of operations, the operations that can be performed in the original sequence may become inexecutable. Similarly, in terms of mutation, although the processing method in our study ensures that the new operation generated by the mutation is executable, it may still make some subsequent operations become inexecutable. Therefore, in order to verify whether an operation is executable, it is necessary to make conditional judgments before and after execution. As an example, the conditions for an operation $oper$ (moving c in p_1 to p_2) to be executable are as follows: 1) c is a class, and p_1 and p_2 are packages; 2) c exists in p_1; 3) c does not exist in p_2. Only when the conditions are met, $oper$ can be executed. After this operation is executed, the completion conditions of the operation need to be verified, which are as follows: 1) c is a class, and p_1 and p_2 are packages; 2) c does not exist in p_1; 3) c exists in p_2. In addition, if more than 10% of the operations in an individual cannot be performed, the individual is considered invalid and will be removed from the population.

IV. Experimental Setup

A. Research Questions

This study is an attempt to recommend refactoring solutions that can improve the software design quality, especially the degree of software's compliance with design principles. To this end, we propose a new software refactoring recommendation approach `DEPICTER`, which employs design-principle metrics as fitness functions and leverages heuristic rules as initialization constraints. In order to evaluate the effectiveness of our approach for refactoring recommendations, we define the following three research questions (including seven subquestions).

RQ1: The effectiveness and usefulness of `DEPICTER` .

1) *RQ1-a:* Can `DEPICTER` significantly improve the degree of software's compliance with design principles?
2) *RQ1-b:* Can `DEPICTER` significantly improve the software design quality?
3) *RQ1-c:* Is `DEPICTER` useful from developers' perspectives?

RQ2: The performance of `DEPICTER` *compared to related techniques.*

1) *RQ2-a:* How does `DEPICTER` perform compared to existing remodularization approach?
2) *RQ2-b:* How does `DEPICTER` perform compared to the common approach using most widely used objectives?

RQ3: The value of design-principle metrics and heuristic rules.

1) *RQ3-a:* What is the value of design-principle metrics as fitness functions compared to traditional metrics?
2) *RQ3-b:* What is the impact of the heuristic rules as initialization constraints on the effectiveness of `DEPICTER`?

The objective of *RQ1* is to determine whether DEPICTER is effective and useful. If the results from *RQ1-a* and *RQ1-b* show that DEPICTER can significantly improve the software design quality and the degree of compliance with design principles, we think DEPICTER is effective. Furthermore, if the results of *RQ1-c* show DEPICTER is useful, it is considered that DEPICTER is of practical value. Otherwise, there is no need to further study the following questions.

The objective of *RQ2* is to examine the performance of DEPICTER compared to other approaches. To evaluate this, we compare DEPICTER with a nearest existing remodularization approach (*RQ2-a*). Besides, we build a common approach as another baseline using the most widely used objectives (*RQ2-b*). If the comparisons of *RQ2* reveal that DEPICTER performs better, the value of DEPICTER is further validated.

RQ3 is derived from *RQ1* and *RQ2*. The objective is to perform additional empirical study on the value of design-principle metrics and heuristic rules, respectively. As the software that follows the software design principles tends to have higher quality, this leads us to naturally conjecture that using design-principle metrics as fitness functions will improve the software design quality better than other metrics. *RQ3-a* is to verify this. In addition, DEPICTER leverages heuristic rules to guide the population initialization process. *RQ3-b* attempts to investigate the influence of the heuristic rules as constraints on the refactoring effect of DEPICTER. If the results from *RQ3-b* show that heuristic rules lead to significantly better performance, it indicates that the defined rules can be employed in the search-based refactoring methods.

These three questions are of critical importance to help both researchers and practitioners understand whether DEPICTER is effective for improving software design quality and to provide a guideline for developing better refactoring models in practice.

B. Experimental Parameters

Genetic algorithm involves many settings of parameters (please refer to the Appendix for more details). Specifically, the parameter setting method in our study is based on the previous researches. The authors in [18]–[20] adjusted parameters according to the system size (i.e., the number of classes *n*). For the system with *n* bigger than 150, Candela *et al.* [20] set the population size as n, and the maximum evolutionary generation number as $20 \times n$. And for the system with small scale (i.e., $n \leq 150$), the population size and maximum number of generations are set to $2 \times n$ and $50 \times n$, respectively. As the size (number of classes) of the system used by Praditwong *et al.* [18] is relatively small, they set the population size to $10 \times n$. Similarly, we also set the population size and number of generations based on the system size. Since this study focuses more on the high-level design quality, the system size here is indicated by the number of packages. In terms of crossover probability and mutation probability, The authors in[18]–[20] set them to 0.8 and $0.004 \times \log_2(n)$, respectively. The authors in [21] and [22] also gave suggestions, recommending a crossover probability interval of $[0.45, 0.95]$ and a mutation probability interval of [0.06,0.1], respectively. In this study, the crossover probability is also set to 0.8. And since too small mutation probability will lead to the premature convergence of the algorithm to the local optimal solution, we set the mutation probability to 0.1 according to the recommended mutation probability interval. Table IV summarizes the parameter settings in the experiments, including the population size, maximum number of generations, the crossover probability, the mutation probability, and the interval of individual length. n is the number of packages in the system. The maximum number of generations is the termination condition of evolution. Individual length refers to the number of genes that make up each individual (a gene represents a refactoring operation). n_{opers} refers to the number of operation types. Since this study considers five kinds of refactoring operations, the value of n_{opers} is 5.

TABLE IV
EXPERIMENTAL PARAMETERS

Parameters	Systems with larger size ($N_p \geq 50$)	Systems with smaller size ($N_p < 50$)
size of population	n	$2 * n$
max number of generations	$5 * n$	$10 * n$
crossover probability	0.8	0.8
mutation probability	0.1	0.1
the interval of individual length	[n_{opers}, $20 * n_{opers}$]	[n_{opers}, $20 * n_{opers}$]

TABLE V
QUALITY INDICATORS FOR EVALUATION

Metric	Description	source
FCoh	the extent to which the classes in a package contribute toward a common purpose	[26]
PF	the ratio of the average number of interfaces used by client packages to the total number of interfaces in a package	[27]
IPSC	the average cohesiveness of package services from the similarity-of-purpose perspective	[27]
SCC	The ratio of the sum of weights of context and data relations to the number of all possible context and data relations	[28]
PCM	the average percentage of the internal classes directly or indirectly depended by each class in the package	[29]
IIPU	the index of inter-package usage, which is an indicator to the degree of collaboration among classes belonging to same packages	[27]
IIPE	the Index of Inter-Package Extending, which is an index about the extent to which class hierarchies are well organized into packages	[27]
IPCI	the index of package changing impact, indicating the extent to which a modularization is free for changes	[27]

C. Evaluation Indicators

To evaluate the design quality, we use eight indicators, which are described in Table V. Among them, index of inter-package extending (*IIPE*) and index of inter-package usage (*IIPU*) are indicators of interpackage interactions, which are used to evaluate the principle of information hiding. Index of package changing impact (*IPCI*) is a change impact indicator, in order to evaluate the variability, maintainability, and reusability of the system. Index of package goal focus (*PF*) and index of package services cohesion (*IPSC*) are the shared target index and the service cohesion index, respectively, to evaluate the common target principle. In addition, functional cohesion (*FCoh*) measures the extent to which the classes in a package contribute toward a common purpose, and *SCC* is the ratio of the sum of weights of context

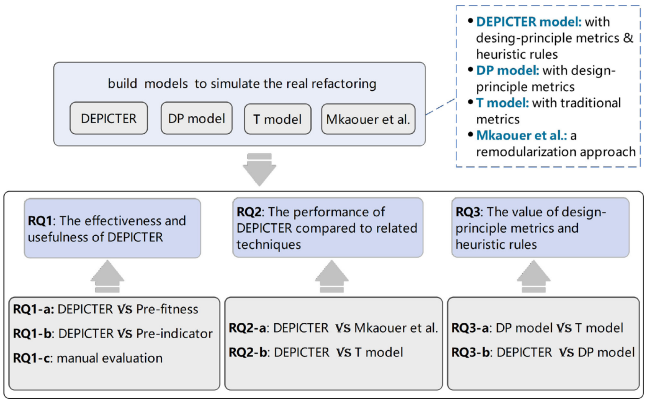

Fig. 5. Validation methods for each question.

and data relations to the number of all possible context and data relations. Besides, package cohesion metric (*PCM*) considers the internal relationships of packages. The reason for choosing the above metrics as evaluation indicators is that these metrics evaluate the quality of software modularization, the rationality of the structure, and the intelligibility from the perspective of high-level architecture (package level or system level) [23] and were widely used in many research work [24]–[26].

D. Validation Method

Fig. 5 illustrates the validation methods for each question. To answer *RQ1-a*, we first calculate the fitness values of the software before refactoring (i.e., "*prefitness*") and then analyze the fitness improvements after DEPICTER refactoring, so as to examine whether DEPICTER can significantly improve the degree of software's compliance with design principles. Similarly, for *RQ1-b*, we obtain the indicator values before refactoring (i.e., "*preindicator*") and investigate the indicator improvements to examine whether DEPICTER can significantly improve the design quality. We use Wilcoxon signed-rank test to statistically analyze the changes in the fitness values and the quality indicators of the system before and after DEPICTER refactoring. The Wilcoxon signed rank test was proposed by Wilcoxon [30]. It is a nonparametric statistical hypothesis test used either to test the location of a set of samples or to compare the locations of two populations using a set of matched samples. To answer *RQ1-c*, we perform a manual evaluation on the usefulness of DEPICTER. The evaluation involves one software engineer, two Ph.D. students, and four master's students. All of them have rich experience in java development (ranging from 2 to 12 years). They are divided into two groups, each group consisting of one Ph.D. student and two master's students. They will receive a set of solutions for each project and check the operations one by one. For each operation, a score ranging from 0 to 10 will be given. 10 means significantly meaningful, and 0 means meaningless. For each operation, we will get two scores from the two groups. If the scores are close, the mean value is taken as the final score. Otherwise, it turns to the software engineer for reevaluation. With discussion and double checking, consensus is reached and all operations are finally scored. UD indicates the usefulness degree of a solution, measured by the sum of scores of all operations in the solution divided by the maximum sum of scores

$$\mathrm{UD} = \frac{\sum_{\mathrm{op}\in\mathrm{Opers}} \mathrm{score}_{\mathrm{op}}}{\#\mathrm{Opers} * 10} \in [0, 1].$$

TABLE VI
METRICS USED IN T MODEL AS FITNESS

Metrics	description
Number of modifications	the number of modifications needed to apply refactorings
QMOOD	a hierarchical model for the assessment of quality attributes in object-oriented designs, such as reusability, flexibility, understandability, functionality, extensibility and effectiveness
Cohesion	refers to the degree of dependence between elements in the same module
Coupling	refers to the degree of interdependence between software modules
Semantic coherence	is based on the semantic similarity between the elements when refactorings are applied.

For *RQ2*, we study the performance of DEPICTER compared to other approaches from two aspects. First, to answer *RQ2-a*, we compare the performance of DEPICTER with a nearest remodularization approach proposed by Mkaouer *et al.* [6]. Their approach also attempts to refactor software from high level and uses the same five refactoring operations as we do. What is more, to answer *RQ2-b*, we build another baseline model (called "T model") based on the systematic overviews of the existing search-based refactoring approaches [10], [11]. T model is built using the most widely used search algorithm and fitness functions as well as the most commonly used parameter settings. It has common characteristics of the existing search-based refactoring approaches. We consider T model representative; so it makes sense to use it as another baseline for comparison. Table VI lists the metrics used as fitness functions in T model (we also call them "traditional metrics" in our study), which are most widely used in the previous related works [11]. In terms of *QMOOD*, we consider the reusability, flexibility, understandability, and effectiveness and exclude the functionality and extendibility factors as Ouni *et al.* [31] did.

To answer RQ3, we built a design-principle (DP) model using design-principle related metrics as fitness functions; for *RQ3-a*, we compare the performance of DP model against T model. Except for the fitness functions, all the other settings of DP model and T model are exactly the same. If the results show that DP model preforms better than T model, it means that using design-principle metrics as fitness functions can significantly improve the software design quality better than traditional metrics. For *RQ3-b*, we compare the refactoring effectiveness of DEPICTER against the DP model to gain insight into the influence of heuristic rules. The only difference between DEPICTER and DP model is that DEPICTER adopts heuristic rules to restrict the population initialization, while the DP model does not.

Since the genetic algorithm is of great randomness, in order to reduce the influence of randomness on the experimental results, we repeat 31 simulation experiments for each experimental system under constant settings. After each experiment, the top

TABLE VII
DESCRIPTIVE INFORMATION OF THE FOUR SYSTEMS UNDER STUDY

Subjects	Version	Number of packages	Number of classes	Number of methods	Line of code (KLOC)
GanttProject	v1.10.2	29	245	1550	41
JfreeChart	v1.0.9	74	521	10343	170
Jhotdraw	v7.1	46	471	5052	91
Xerces-J	v2.11	78	830	9395	161

3 optimal solutions are obtained. Therefore, for each system, each model will get 93 best solutions. And after executing each solutions, the corresponding refactored software quality indicators can be obtained. For *RQ2* and *RQ3*, we use the quality indicators in Table V to evaluate the effectiveness of different models.

E. Subject Systems

Our experiments are performed on four well-known open source java systems, i.e., Xerces-J, GanttProject, Jhotdraw, and JfreeChart. As shown in the review on search-based refactoring [11], these systems are the top 4 most used systems in the existing work. Among them, Xerces is Apache's collection of software libraries for parsing, validating, serializing, and manipulating XML. GanttProject is a tool for project scheduling and management using Gantt charts, which can be executed on different operating systems such as Windows, Linux, and Mac OS. Jhotdraw is a 2-D graphics framework for structured drawing editors. And JfreeChart is a flexible and powerful java library for generating charts, which can be used on the client-side or the server-side. Table VII describes the basic information of these systems, including the system version, the number of packages/classes/methods, and the number of lines of code (KLOC).

These four systems were chosen as the subject systems for the following reasons: First, these systems are open source systems and are widely used in the industry; second, as the main reason, these four systems are widely used in previous search-based refactoring studies [31], [32]. In the recent survey of software refactoring, statistics are made on the most commonly used experimental projects [11], [33], and the conclusion shows that these four subjects are the top 4, which means that they are highly recognized in the research field of software refactoring. Therefore, we choose these systems as the experimental subjects, which further ensures the validity and credibility of the experimental results.

V. EXPERIMENTAL RESULTS

A. RQ1: The Effectiveness and Usefulness of DEPICTER

To evaluate the effectiveness of DEPICTER on refactoring, we consider two aspects: 1) whether DEPICTER can significantly improve the degree of system's compliance with design principles (*RQ1-a*); 2) whether the system quality is significantly improved after applying the solutions recommended by DEPICTER (*RQ1-b*). In addition, to examining the usefulness of DEPICTER in practice, we perform a manual evaluation (*RQ1-c*).

TABLE VIII
PERFORMANCE OF DEPICTER IN TERMS OF FITNESS

	(a) Ganttproject			(b) Jfreechart		
Fitness	**Before**	**After**	**%improved**	**Before**	**After**	**%improved**
MII	0.204	0.299 ± 0.076 √	46.7	0.071	0.172 ± 0.014 √	142.4
NC	0.734	0.808 ± 0.196 √	10.1	0.828	0.847 ± 0.033 √	2.2
IC	0.918	0.947 ± 0.065 √	3.2	0.889	0.924 ± 0.035 √	4
SAVI	0.988	0.992 ± 0.011 √	0.4	0.992	0.992 ± 0.002 √	0.1
CUL	0.608	0.582 ± 0.095	4.3	0.666	0.635 ± 0.038 √	4.5
CU	0.207	0.329 ± 0.107 √	58.9	0.457	0.471 ± 0.070 √	3.2
CC	0.279	0.367 ± 0.137 √	31.4	0.281	0.426 ± 0.066 √	51.9
Win/tie/loss	-	6/1/0	7/0/0	-	7/0/0	7/0/0
	(c) Jhotdraw			**(d) Xerces**		
Fitness	**Before**	**After**	**%improved**	**Before**	**After**	**%improved**
MII	0.066	0.173 ± 0.029 √	162.7	0.034	0.084 ± 0.012 √	148.3
NC	0.833	0.853 ± 0.038 √	2.4	0.866	0.884 ± 0.032 √	2
IC	0.799	0.818 ± 0.057 √	2.4	0.952	0.962 ± 0.043 √	1
SAVI	0.979	0.981 ± 0.008 √	0.3	0.981	0.979 ± 0.004 ×	-0.2
CUL	0.706	0.655 ± 0.065 √	7.2	0.686	0.644 ± 0.039 √	6.1
CU	0.369	0.421 ± 0.133 √	14.2	0.282	0.472 ± 0.062 √	67.3
CC	0.405	0.528 ± 0.120 √	30.3	0.034	0.527 ± 0.069 √	31.8
Win/tie/loss	-	7/0/0	7/0/0	-	6/0/1	6/0/1

Results for RQ1-a: Table VIII illustrates the effectiveness of DEPICTER in terms of fitness. The first column of Table VIII lists the fitness metrics. In each subtable on the right, the first column reports the original fitness values before refactoring, the second column summarizes the average fitness and the corresponding standard deviation from 31 times executions of DEPICTER as well as how often fitness after refactoring are significantly better (denoted by ✓, significant at the 0.05 level) or worse (denoted by ×) than the fitness before refactoring by the Wilcoxon's signed-rank test. If the significance level is less than 0.05, it is considered significant, otherwise not significant. The third column reports the improvement in percentage in terms of the mean fitness value. In particular, the row "Win/tie/loss" further reports the following: 1) the number of fitness metrics for which DEPICTER model obtains a better, equal, and worse values than the original value before refactoring (under the first column in the subtable); 2) the number of fitness metrics for which the "% improved" is greater than, is equal to, and is less than zero (under the last column).

From Table VIII(a), we can see that DEPICTER significantly improves six out of seven fitness values and has a similar value in terms of the other one. About "%improved," DEPICTER has improvements of 46.7% and 58.9%, respectively, for MII and CU. Since CUL measures the extent to which the classes in a module are different in terms of lines of code, the smaller the value, the better. The mean CUL of DEPICTER is 0.582, smaller than the original one, with an improvement of 4.3%. Similar results can be observed from other subtables. Especially, in Tables VIII(b) and (c), all the seven fitness values after refactoring are significantly better than the fitness values before. In addition, the improvement is considerable in terms of MII, even more than 100% in the three subtables. In summary, the results from Table VIII clearly indicate that DEPICTER is indeed effective for improving the degree of system's compliance with design principles.

Results for RQ1-b: Similarly, Table IX summarizes the statistical information of software design quality indicators before and after refactoring. It is easy to observe that DEPICTER has a good effectiveness in terms of most evaluation indicators in all datasets, with considerable improvements. However, in terms of

TABLE IX
PERFORMANCE OF DEPICTER IN TERMS OF INDICATORS

	(a) Ganttproject			(b) Jfreechart		
Indicators	**Before**	**After**	**%improved**	**Before**	**After**	**%improve**
FCoh	0.277	0.432 ± 0.116 √	55.9	0.14	0.155 ± 0.049 √	11
PF	0.627	0.693 ± 0.099 √	10.5	0.579	0.725 ± 0.037 √	25.2
IPSC	0.851	0.908 ± 0.034 √	6.7	0.735	0.831 ± 0.026 √	13.2
SCC	0.406	0.566 ± 0.091 √	39.4	0.412	0.514 ± 0.060 √	24.5
PCM	0.332	0.528 ± 0.109 √	59.3	0.318	0.360 ± 0.064 √	13.4
IIPU	0.269	0.333 ± 0.026 √	60.8	0.214	0.301 ± 0.010 √	87.5
IIPE	0.756	0.769 ± 0.125	1.8	0.793	0.821 ± 0.051 √	3.6
IPCI	0.850	0.848 ± 0.077	-0.2	0.932	0.922 ± 0.017 ×	-1.1
Win/tie/loss	-	6/2/0	7/0/1	-	7/0/1	7/0/1
	(c) Jhotdraw			**(d) Xerces**		
Indicators	**Before**	**After**	**%improved**	**Before**	**After**	**%improve**
FCoh	0.446	0.485 ± 0.126 √	8.8	0.229	0.352 ± 0.042 √	54.1
PF	0.703	0.783 ± 0.093 √	11.3	0.466	0.645 ± 0.048 √	38.3
IPSC	0.840	0.908 ± 0.039 √	8.1	0.591	0.716 ± 0.053 √	21.2
SCC	0.479	0.534 ± 0.114 √	11.5	0.485	0.569 ± 0.061 √	17.4
PCM	0.443	0.563 ± 0.115 √	27.1	0.494	0.516 ± 0.071 √	4.5
IIPU	0.283	0.367 ± 0.034 √	64.7	0.272	0.355 ± 0.018 √	67.7
IIPE	0.717	0.722 ± 0.086	0.7	0.872	0.866 ± 0.083	-0.7
IPCI	0.921	0.908 ± 0.055	-1.4	0.953	0.945 ± 0.011 ×	-0.8
Win/tie/loss	-	6/2/0	7/0/1	-	6/1/1	6/0/2

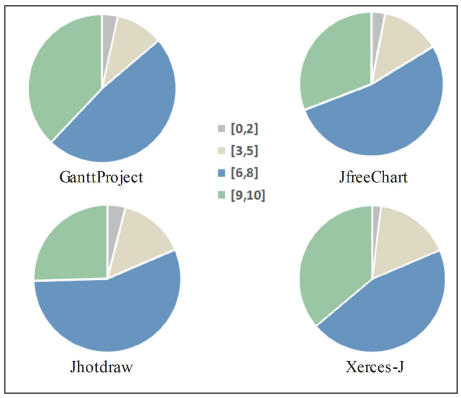

Fig. 6. Distribution of the operation scores.

IPCI, we can see that DEPICTER performs significantly worse on JfreeChart and Xerces and performs similarly on the other systems with slightly negative improvements. The reason may be that these two systems have good quality on the extent to which a modularization is free for changes, thus with a high value of IPCI larger than 0.85, which may be difficult to be further optimized by refactoring.

Results for RQ1-c: We evaluate the usefulness of DEPICTER manually. In the evaluation, the participants manually reviewed 60 recommended solutions (15 per project) with a total of 2287 operations. And about 18% of operations with divergences were rechecked by the software engineer. Fig. 6 shows the distribution of the operation scores. As can be seen, more than 80% of operations got a score above 5, of which more than 31% received high scores in [9] and [10], which indicates that most operations are considered meaningful from the participants' point of view. Fig. 7 reveals the average usefulness degree of recommended solutions in terms of UD. We can see that the mean values of UD are consistently above 0.75 for all projects. From the results of Figs. 6 and 7, we think that DEPICTER is useful in practice.

Overall, combining the results from *RQ1*, we believe that DEPICTER is of positive value on improving the software design

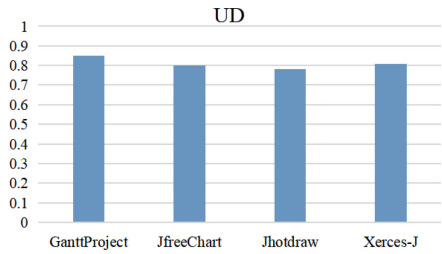

Fig. 7. Usefulness degree of the suggested solutions.

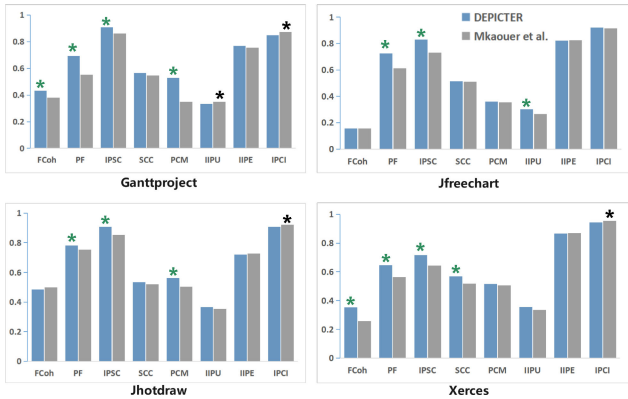

Fig. 8. Performance of DEPICTER and Mkaouer *et al.* [6].

quality and the degree of compliance with design principles, and from the developers' perspectives, DEPICTER is considered useful.

B. RQ2: The Performance of DEPICTER *Compared to Related Techniques*

For *RQ2*, we build two baselines to examine whether DEPICTER can perform better than the existing remodularization approach (*RQ2-a*) and the common approach with most widely used objectives (*RQ2-b*).

Results for RQ2-a: We compare the performance of DEPICTER with a nearest remodularization approach proposed by Mkaouer *et al.* [6]. Fig. 8 illustrates the performance of DEPICTER (in blue) and Mkaouer *et al.* (in gray). Each subfigure reveals the mean values of the evaluation indicators after refactoring. The bar with a star (*) symbol on the top means the corresponding value is significantly better than the other one. As can be seen from Fig. 8, DEPICTER has a larger mean value than Mkaouer *et al.* in terms of most indicators in all studied systems. More specifically, DEPICTER significantly outperforms Mkaouer *et al.* at more than three indicators. The results reveal that DEPICTER performs better than Mkaouer *et al.* in terms of most evaluation indicators.

Results for RQ2-b: We build a T model as another baseline model, using the most widely used traditional metrics as fitness functions. Table X shows the results from examining the effectiveness of DEPICTER when compared with T model. As can be seen, DEPICTER outperforms T model in all the datasets. On GanttProject, three quality indicators obtained by DEPICTER are significantly better than the indicators obtained by T model. And DEPICTER performs similarly with T model in terms

TABLE X
EFFECTIVENESS OF DEPICTER COMPARED WITH THE TRADITIONAL METHOD

	(a) Ganttproject			(b) Jfreechart		
Indicators	**T model**	**DEPICTER**	**%improve**	**T model**	**DEPICTER**	**%improved**
FCoh	0.377 ± 0.047	0.432 ± 0.116	14.6	0.161 ± 0.011	0.155 ± 0.049	-3.7
PF	0.711 ± 0.044	0.693 ± 0.099	-2.5	0.596 ± 0.021	0.725 ± 0.037 √	21.7
IPSC	0.873 ± 0.011	0.908 ± 0.034 √	3.9	0.740 ± 0.019	0.831 ± 0.026 √	12.4
SCC	0.495 ± 0.048	0.566 ± 0.091 √	14.4	0.428 ± 0.012	0.514 ± 0.060 √	20.1
PCM	0.433 ± 0.050	0.528 ± 0.109 √	22	0.337 ± 0.013	0.360 ± 0.064 √	6.8
IIPU	0.384 ± 0.103	0.333 ± 0.026	-13.4	0.243 ± 0.024	0.301 ± 0.010 √	24
IIPE	0.767 ± 0.015	0.769 ± 0.125	0.2	0.793 ± 0.001	0.821 ± 0.051 √	3.6
IPCI	0.883 ± 0.013	0.848 ± 0.077 ×	-3.9	0.938 ± 0.001	0.922 ± 0.017 ×	-1.7
Win/tie/loss	-	3/4/1	5/0/3	-	6/1/1	6/0/2
	(c) Jhotdraw			**(d) Xerces**		
Indicators	**T model**	**DEPICTER**	**%improve**	**T model**	**DEPICTER**	**%improved**
FCoh	0.497 ± 0.039	0.485 ± 0.126	-2.4	0.244 ± 0.013	0.352 ± 0.042 √	44.5
PF	0.737 ± 0.029	0.783 ± 0.093 √	6.3	0.482 ± 0.004	0.645 ± 0.048 √	33.7
IPSC	0.860 ± 0.023	0.908 ± 0.039 √	5.6	0.606 ± 0.001	0.716 ± 0.053 √	18.1
SCC	0.521 ± 0.025	0.534 ± 0.114	2.6	0.496 ± 0.008	0.569 ± 0.061 √	14.7
PCM	0.497 ± 0.036	0.563 ± 0.115 √	13.4	0.505 ± 0.008	0.516 ± 0.071	2.3
IIPU	0.354 ± 0.064	0.367 ± 0.034	3.6	0.272 ± 0.001	0.355 ± 0.018 √	30.8
IIPE	0.733 ± 0.019	0.722 ± 0.086	-1.4	0.873 ± 0.001	0.866 ± 0.083	-0.8
IPCI	0.931 ± 0.004	0.908 ± 0.055	-2.4	0.954 ± 0.002	0.945 ± 0.011 ×	-1
Win/tie/loss	-	3/5/0	5/0/3	-	5/2/1	6/0/2

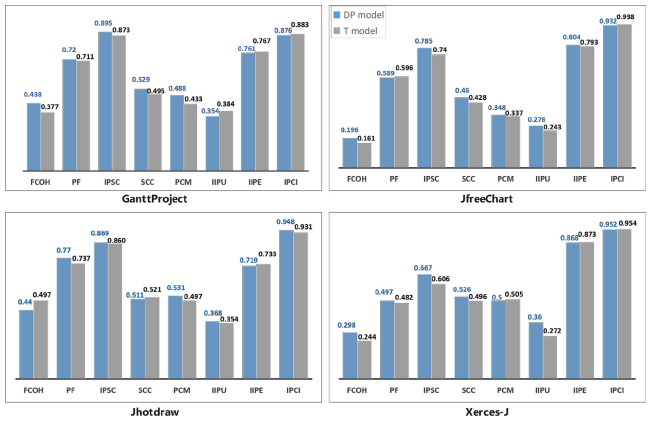

Fig. 9. Performance of DP model and T model.

of the other four indicators. About improvements, DEPICTER improves the mean values of five indicators. On JfreeChart, DEPICTER is significantly more effective than T model on 6 out of 8 indicators, with six wins, one tie, and one loss. Similar results can be seen in Tables X(c) and (d), and DEPICTER model consistently has a larger mean value than T model in terms of most evaluation indicators.

In summary, the core observation from *RQ2* is that DEPICTER has better performance than the existing remodularization approach and the common approach with most widely used objectives in terms of most evaluation indicators.

C. RQ3: The Value of Design-Principle Metrics and Heuristic Rules

For *RQ3*, we further study the influence of using design-principle metrics on refactoring performance compared with using traditional metrics (*RQ3-a*) and analyze the impact of the heuristic rules as initialization constraints on the refactoring effectiveness of DEPICTER (*RQ3-a*).

Results for RQ3-a: To investigate the value of design-principle metrics as fitness functions on refactoring, we build a DP model using design-principle metrics as fitness functions and a T model using traditional metrics as fitness functions. Except for the fitness functions, all the other settings of them are exactly the same.

Fig. 9 shows the refactoring effectiveness for DP model (denoted in blue) and T model (denoted in gray). Each subfigure corresponds to one distinct dataset and reveals the mean value in terms of the eight evaluation indicators. As can be seen from Fig. 9, DP model consistently has a larger mean value than T model in terms of at least five out of eight indicators in all datasets. And the improvements for some indicators are considerable, such as the FCoh and PCM in GanttProject, the FCoh and IIPU in JfreeChart, the PF and PCM in Jhotdraw, and the FCoh and IIPU in Xerces-J. In addition, Table XI shows the average improvement (in percentage) of DP model over T model and the corresponding significance obtained by Wilcoxon's signed-rank test. Each cell with a positive average %improvement with * indicates that the DP model outperforms T model at the significant level of 0.05 (denoted by ***), 0.10 (denoted by **), or 0.20 (denoted by *). Similarly, a negative value with * means that T model significantly outperforms DP model. In particular, the last column "Win/tie/loss" further reports the number of indicators for which the average %improvement is greater than, is equal to, and is less than zero. The results of Table XI reveal that, for each dataset, the average %improvement under most evaluation indicators are larger than 0, and at least half of the improvements are significant at the significant level of 0.20. More specifically, we also observe the following: 1) In terms of the IPSC indicator, the effectiveness of DP model is consistently better than T model at the significance of 0.05 in all datasets; 2) on JfreeChart, DP model is significantly more effective than T model in terms of six out of eight indicators, and similar performance on the other two indicators; 3) on the other datasets, DP model performs significantly better on at least half of the indicators, with at least five wins on the average %improvement.

Results for RQ3-b: We study the influence of the defined heuristic rules as initialization constraints on the refactoring effect of DEPICTER. We build a DP model. The only difference between DEPICTER and DP model is that DEPICTER adopts heuristic rules to restrict the population initialization, while the DP model does not.

Fig. 10 reveals the performance of DEPICTER and the DP model. Each subfigure in Fig. 10 corresponds to one distinct project and shows the distribution of the eight evaluation indicators. For each model, the boxplot shows the median (the horizontal line within the box), the 25th percentile, and the 75th percentile (the lower and upper sides of the box) of the indicators. The results from Fig. 10 show that DEPICTER has a larger median value than the DP model in terms of most indicators in all studied system. More specifically, for JfreeChart and Xerces-J, the median values of DEPICTER are larger on six indicators. For Jhotdraw, DEPICTER has a larger median value on seven out of eight indicators, and a slightly smaller median value on the other one. Furthermore, Table XII shows the average improvement. As can be seen, for JfreeChart, DEPICTER model has significantly better performance than the DP model in terms of six out of

TABLE XI
AVERAGE % IMPROVEMENT OF DP MODEL OVER T MODEL

Subjects	FCoh	PF	IPSC	SCC	PCM	IIPU	IIPE	IPCI	Win/tie/loss
GanttProject	**16.1** ***	1.3	**2.5** ***	**6.8** *	**12.8** ***	-7.9	-0.7 ***	-0.8	5/0/3
JfreeChart	**21.6** ***	-1.2	**6.1** ***	**7.6** ***	**3.1** ***	**13.5** ***	**1.4** ***	-0.6	6/0/2
Jhotdraw	-11.4 ***	**4.5** ***	**0.9** ***	-2	**6.8** ***	**3.8** **	-1.9	**1.9** ***	5/0/3
Xerces-J	**22** ***	**3.1** ***	**10.1** ***	**6** ***	-0.9 **	**32.7** ***	-0.6 ***	-0.3 **	5/0/3

TABLE XII
AVERAGE % IMPROVEMENT OF DEPICTER MODEL OVER THE DP MODEL

Subjects	FCoh	PF	IPSC	SCC	PCM	IIPU	IIPE	IPCI	Win/tie/loss
GanttProject	-1.3	-3.7 **	**1.4** ***	**7.1** ***	**8.2** ***	-6.1 ***	1	-3.1 ***	4/0/4
JfreeChart	-20.8 ***	**23.1** ***	**5.9** ***	**11.5** ***	**3.6** *	**9.2** ***	**2.2** ***	-1.1 ***	6/0/2
Jhotdraw	**10.2** ***	**1.7** *	**4.6** ***	**4.7** *	**6.2** ***	-0.2 ***	0.5	-4.2 ***	6/0/2
Xerces-J	**18.4** ***	**29.8** ***	**7.3** ***	**8.2** ***	3.3	-1.4 **	-0.2 **	-0.8 ***	5/0/3

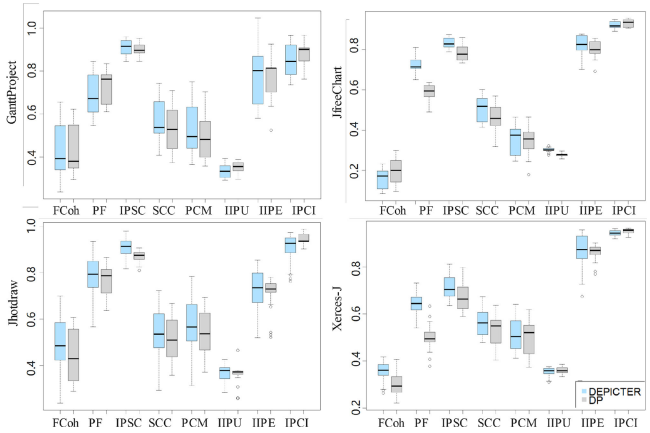

Fig. 10. Performance of DEPICTER model and the DP model.

eight indicators. For Jhotdraw, DEPICTER model outperforms DP model in terms of five indicators and obtains higher mean values on six indicators. Similar results can be seen for Xerces-J, and DEPICTER performs better than DP mode and gets five wins and three losses in terms of the %improvement.

In particular, for GanttProject, the results from Fig. 10 and Table XII reveal that DEPICTER has a larger median value than the DP model in terms of four indicators and a smaller median value in terms of the other four indicators. With regard to the %improvement, the performance of DEPICTER is not better or worse than DP model, with four wins and four losses. The results indicate that, on GanttProject project, the heuristic rules did not play a significantly positive role in the effectiveness of DEPICTER model. The reason for this phenomenon may be the relatively small scale of GanttProject with 29 packages. For such small-scale systems, once the conditional constraints are added, the number of entities that can be selected is greatly reduced. Although the rules are loosely defined in this study, they may still slightly affect the diversity of the population and cause the influence of the rules for GanttProject to be not significant.

In conclusion, the results from *RQ3* clearly reveal the following. 1) From the viewpoint of practical use, using design-principle metrics as fitness functions can considerably improve the effectiveness of software refactoring when compared with traditional metrics. 2) For most studied systems, the defined heuristic rules have positive influence on the recommendation results of DEPICTER model as expected, which can be considered in the future search-based refactoring studies.

VI. VALIDITY

In this section, we discuss the most important threats to the validity of our study, mainly including four aspects: the time cost, randomness, usefulness, and universality.

Time cost: Refactoring scenarios can often be divided into two types. One is refactoring during development. In such a case, developers often do the refactoring with other code changes such as fixing bugs and implementing new features. This scenario is more suitable for personal development level, with high efficiency requirements of the refactoring recommendation or automation. The other one is refactoring during maintenance, such as refactoring before version-level release or optimizing architecture of legacy software. For such complex refactoring at the architectural level, generally, more time will be reserved. DEPICTER is specifically for the second one, which is not so demanding for efficiency (that is why we do not discuss the time cost too much in this study). Even so, in terms of efficiency, we also conducted a preliminary verification manually. By comparing the time to initialize the population and the time of one generation, we find that DEPICTER is not worse than the baseline methods. As a result, we think the time consuming will not pose a big threat to the effectiveness of our method.

Randomness: Search-based algorithms are high of randomness. The main feature of such methods is that they can generate unexpected solutions which may be of great effectiveness. Another feature is that the results are not reproducible, specially when the random seed is not set. Even for the same project and the same search settings, the recommended results of two different executions may be different. In order to prevent the randomness from interfering with the validity of the experimental results, we execute the process 31 times for each project and select the top 3 solutions for each time. With these 93 best solutions, we use Wilcoxon signed-rank test to study the performance of our approach.

Usefulness: We study the usefulness of DEPICTER by manual evaluation. An inherent problem for all studies with manual processes is that different perceptions and preferences usually lead to different decisions. Therefore, the different participants involved in the experiments of *RQ3-c* may have divergent opinions about the recommended operations. To minimize the deviation, we group the participants into two groups and take the average value of the two groups as the score of the operation. The operations with divergences are rechecked by the software engineer. With discussion, consensus is reached and all operations are finally scored.

Universality: The threat to the universality of our study is that our conclusions may not be generalized to other systems. In our study, we use four open-source Java systems to investigate the four research questions. Therefore, our conclusion may not be generalized either to those systems developed by other programming languages or close-source systems. This is an inherent problem for all studies in software engineering, but not unique to us, as many factors cannot be characterized completely. To mitigate this threat, there is a need to replicate our study using a wide variety of systems in the future work.

VII. Related Works

A. Software Refactoring Techniques

Software refactoring is one of the most important activities of software engineering. The concept of software refactoring was first proposed by Opdyke in 1990 [34]. Subsequently, Fowler [35] published a book on refactoring, so that refactoring is well known and applied. Many refactoring-related methods have been proposed. For example, in terms of refactoring opportunity detection, Liu *et al.* [36] proposed a method for identifying opportunities for moving method and proposed a code smell detection method based on machine learning [37]. Terra *et al.* [38] implemented the Jmove tool to detect the refactoring opportunities of move method by analyzing the static dependency between the methods. In terms of assisting manual refactoring, Murphy-Hill *et al.* [39] implemented a tool to provide views and refactoring annotations to assist software developers in manual refactoring. Ge *et al.* [40] proposed the tool GhostFactor to assist developers in manually modifying the code and automatically checking the correctness of manual refactoring. Since the manual refactoring has high cost and is easy to introduce problems, more and more researches gradually focused on automated software refactoring methods. For example, many researchers studied clustering-based refactoring methods from different granularity and perspectives [41], [42]. Wang *et al.* [41] constructed a multirelational network diagram at the method level to analyze inheritance and noninheritance hierarchies and used weighted clustering methods to reorganize methods to achieve class merging and segmentation. Pourasghar *et al.* [42] proposed a clustering algorithm based on dependency graph, which used dependency depth to calculate the similarity between entities, combined with graph theory information to reconstruct the software. In addition, there are methods based on machine learning [43] and feature requests [44].

B. Search-Based Refactoring

Since some software quality attributes are mutually exclusive, that is, improving one indicator may reduce another at the same time, it is impossible to find a refactoring solution that improves all quality attributes for a software system. In fact, the software refactoring problem can be considered as an optimization problem, which can be achieved through search algorithm. To this end, search algorithms are widely used in software refactoring, which can achieve tradeoffs between different quality attributes. In addition, the automated search method can avoid some deviations that may be caused by human intuition, so that some unexpected feasible solutions can be constructed. The following introduces search-based software refactoring related work, mainly from two different dimensions of single-objective optimization and multiobjective optimization.

Single-objective optimization software refactoring method: Most existing works define search-based software refactoring as a single-objective optimization problem [11] to find the best refactoring solution according to one single fitness function. The main goal is to maximize or minimize one objective. Mancoridis *et al.* [45] first used search-based methods to solve software modularization problems in 1998, using hill-climbing algorithms and genetic algorithms to achieve single-objective optimization (i.e., high cohesion and low coupling). Mitchell and Mancoridis [46], [47] also used the same method and proposed Bunch (a tool that supports automatic system decomposition) to segment the relationship graph composed of entities in the source code. Seng *et al.* [48] presented an approach to improve the subsystem decomposition of object-oriented system, which treat subsystem decomposition tasks as single-objective optimization problems. Subsequently, they proposed a search-based method for refactoring recommendation, using evolutionary algorithms to simulate reconstruction [49]. O'Keeffe *et al.* [50] use a variety of search-based techniques, such as hill climbing algorithms and simulated annealing algorithms, to provide automated software refactoring support and use QMOOD to evaluate quality improvements. Kessentini *et al.* [51] proposed a single-objective optimization method to find the best sequence of refactoring operations and improve code quality by reducing the number of design defects in the source code as much as possible.

Multiobjective optimization software refactoring method: Since most software engineering problems are multiobjective optimization problems [13], single-objective optimization methods cannot be effectively applied to the actual refactoring scenarios. Therefore, recently, researchers have also proposed multiobjective optimization software refactoring methods, which has attracted great attention. Praditwong *et al.* [18] first regarded the software refactoring problem as a multiobjective optimization problem in 2010 and proposed a multiobjective evolutionary algorithm (two-archive algorithm). Ouni *et al.* [52] used genetic algorithm to automatically generate defect detection rules and then used the NSGA-II algorithm to search for repair solutions, so as to maximize the number of defects fixed while minimizing the workload of code refactoring. In addition, they also studied other multiobjective optimization effects, such as increasing the utilization of refactoring history with similar context, reducing

semantic errors, reducing the number of design defects, and reducing code changes [31]. Abdeen *et al.* [53] extended the previous work and regarded the refactoring task as a multiobjective optimization problem to improve the existing package structure while minimizing the modification of the original design. Mkaouer *et al.* [6] used the NSGA-III algorithm and took the average number of classes, the number of packages, cohesion, coupling, the number of code changes, the similarity with historical changes, and semantic similarity as the optimization objectives. Stefano *et al.* [54] designed a community-based refactoring recommendation method to optimize socio-technical congruence, while taking into account the source code dependencies among software components, so as to provide refactoring suggestions for extract class and extract package refactors. Kaur *et al.* [55] proposed a multiobjective optimization technique to generate refactoring solutions to improve software quality, maximize the use of smell severity, and increase the consistency with class importance. Mohan *et al.* [56] implemented a tool MultiRefactor, which includes four separate measures of software quality, code priority, refactoring coverage, and element recentness. Hou *et al.* [57] use the NSGA-II algorithm to optimize three fitness functions (reducing the number of bad smells, increasing the severity of the eliminated code smell, and maximizing the importance of the class where the code smell instance is eliminated). Maikantis *et al.* [58] employed a search-based method, relying on a set of metrics, to identify refactoring opportunities for move classes. Abid *et al.* [60] proposed X-SBR to provide explanations for refactoring solutions [59] and proposed intelligent change operators for multiobjective search, using QMOOD metrics as objectives. More literature about multiobjective refactoring are illustrated in the Appendix.

Mariani *et al.* [11] recently provided a systematic overview of the existing search-based refactoring approaches from multiple perspectives. In terms of objectives, the results showed that the mostly used metrics as fitness are the number of bad smells, the number of modification, the QMOOD metric, cohesion, coupling, and semantic coherence.

VIII. Conclusion

In this study, we proposed a multiobjective software refactoring approach DEPICTER, employing design-principle metrics as fitness functions and leveraging heuristic rules as initialization constraints. It aims to recommend good refactoring activities for developers to improve the software quality, especially the degree of software's compliance with design principles. To gain deep insights into the effectiveness of DEPICTER approach, we conducted detailed experiments based on four widely used systems. The experimental results showed the following. 1) DEPICTER is effective on improving the software design quality and the degree of compliance with design principles and is useful from the developers' perspectives. 2) DEPICTER performs better than the existing remodularization approach and the common approach with most widely used objectives. 3) Using design-principle metrics as fitness functions can considerably improve the effectiveness of refactoring when compared with traditional metrics; the defined heuristic rules have positive influence on the recommendation results of DEPICTER. We believe these results are valuable for guiding the development of better refactoring models in practice.

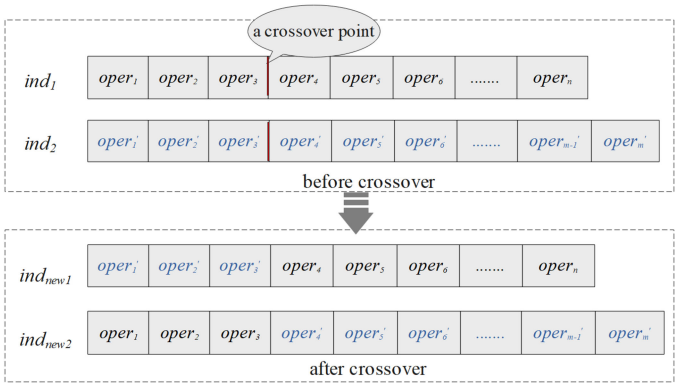

Fig. 11. Crossover process.

In the future, we plan to extend our study in the following two directions. The first one is to optimize the fitness setting of DEPICTER. Currently, DEPICTER uses seven design principle metrics as fitness functions to guide the searching process. We plan to explore more effective design-principle metrics in DEPICTER and then do redundancy analysis to select more valid metrics as fitness functions. We will perform the selection not only by comparing the fitness values but also by taking into account the relationship between fitness functions. The second one is to take developer's requirements or feedback into account. DEPICTER can improve the degree of compliance with software design principles. However, developers may have their specific requirements. In order to have more practically useful recommendations, it is necessary to adjust the recommendation strategy according to developers' requirements. Therefore, in the future, we plan to integrate the developers' requirements or feedback into the searching process.

Appendix A
Appendix

Fig. 11 shows the crossover process. As can be seen that, a crossover point is randomly selected on the chromosome, and then the two parents are exchanged and recombined according to the crossover point.

Fig. 12 shows the mutation process of changing a certain refactoring operation. Firstly, randomly select a mutation point; then execute all operations before the mutation point in sequence to update the state of the system; then, according to the current new system state, a new refactoring operation is randomly generated.

Fig. 13 shows the positions of the involved parameters in the framework of DEPICTER, including the interval of individual length, the size of population, the crossover probability, the mutation probability and the max number of generations.

Table XIII illustrates the overview of the selected literature about multi-objective refactoring, including the year, the used

TABLE XIII
AVERAGE % IMPROVEMENT OF DEPICTER MODEL OVER THE DP MODEL

Study	year	Technique	#Objectives	Fitness functions	Refactoring types	Granularity
[1]	2014	NSGA-III	15	WMC, RFC, LCOM, CC, NA, AH, MH, NLC, CBO, NAS,NC, DIT, PF, AIF and NOC	Add Parameter, Rename Method Encapsulate Collection/ Downcast/ Field, Collapse Hierarchy, Hide Method, Extract Class /Interface /Method/ Subclass/ Superclass, Inline Class/ Method, Move Field/ Method, Pull Up Field/ Method, Push Down Field/ Method and Remove Parameter/ Setting Method	class
[2]	2014	NSGA-II	3	improve software quality, reduce the number of refactorings, increase semantic coherence	Add Parameter, Rename Method Encapsulate collection/ Downcast/ Field, Collapse Hierarchy, Hide Method, Extract Class/Interface/ Method/Subclass/Superclass, Inline Class/Method, Move Field/Method, Pull Up Field/Method,Push Down Field/Method and Remove Parameter/Setting Method	class
[3]	2013	NSGA-II	5	Maximizing package cohesion, Minimizing package coupling, Minimizing package cycles, Avoiding Blob packages, Minimizing the modification of original modularizations	Move class	class
[4]	2017	NSGA-II	4	package cohesiveness index, package connectedness index, intra-package connection density, package size index	Move class	class
[5]	2016	NSGA-II	4	improves the quality by minimizing the number of design defects, minimizes code changes required to fix those defects, preserves design semantics, and maximizes the consistency wi th the previously code changes	move method, move field, pull up method, pull up field, push down method, push do wn field, inline class, move class, extract class, extract method, and extract interface	class
[6]	2014	NSGA-II	2	maximizing the external cohesion of the clusters in C, minimizing the number of clusters	Split a selected cluster, moves all methods from a cluster to another, moves methods between clusters	interface
[7]	2015	NSGA-III	7	number of classes per package (to minimize); number of packages in the system (to minimize); cohesion (to maximize), coupling (to minimize), Number of code changes, Similarity with history of code changes, Semantics	Move method, Extract class, Move class, Merge packages, and Extract/ Split package	package
[8]	2020	NSGA-II	3	reduce the number of code smells, maximize the severity of the removed code smell, maximize the importance of the class where the code smell instance is eliminated	Move method, move field, pull up field, pull up method, push down field, push down m ethod, inline class, extract class	class
[9]	2019	NSGA-II, NSGA-III	4	QMOOD, a priority objective allowing user defined preferences for classes to be refactored; a refactoring coverage objective where covering more areas of code is favoured; element recentness objective where newer elements are favoured for refactoring	9 types at field level, 9 types at method level, 8 types at class level	class
[10]	2021	MOSHO	3	software quality, use of smell severity, consistency with class importance	Move Class, move field, move method, extract class, extract interface, extract method, inline class, pull up field, pull up method, push down filed, push down method	class
[11]	2016	HCM	3	minimizing the inconsistencies between the target and source; improving OO design quality, such as coupling and cohesion; maximizing the lexical similarity between the target and source	Move method, pull up field/method, extract class	class
[12]	2018	NSGA-II	2	Minimize the number of code changes; Maximize software quality(QMOOD)	Move method, extract class, move method	class
[13]	201	NSGA-II	9	QMOOD (MReusability, Flexibility, Understandability, Functionality, Extendibility, Effec-tiveness, Complexity, Cohesion, Coupling)	Move Method, Move Field, Extract Class, Encapsulate Field, Pull Up Field, Pull Up Method, Push Down Field, Push Down Method, Extract SubClass, Extract SuperClass	class

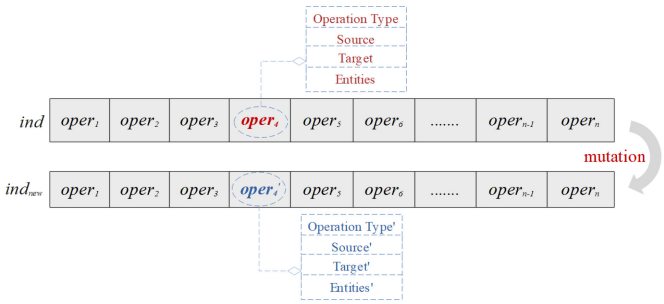

Fig. 12. Mutation process.

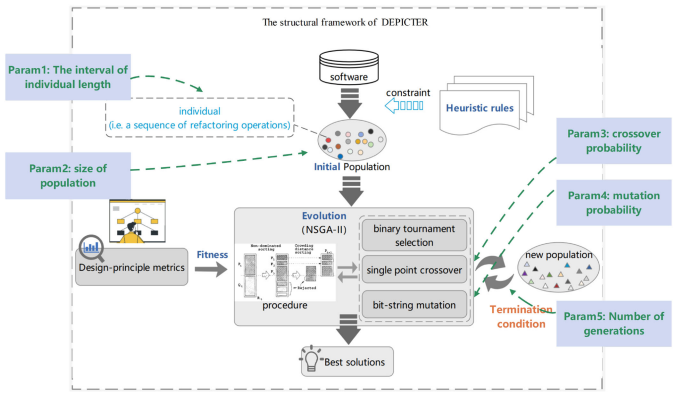

Fig. 13. The main parameters involved in the framework of DEPICTER.

search technique, the fitness functions, the refactoring types and the involved granularity.

REFERENCES

[1] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Connallen, and K. A. Houston, "Object-oriented analysis and design with applications," *ACM SIGSOFT Softw. Eng. Notes*, vol. 33, no. 5, pp. 29–29, 2008.
[2] C. C. Venters *et al.*, "Software sustainability: Research and practice from a software architecture viewpoint," *J. Syst. Softw.*, vol. 138, pp. 174–188, 2018.
[3] W. E. Wong and S. Gokhale, "Static and dynamic distance metrics for feature-based code analysis," *J. Syst. Softw.*, vol. 74, no. 3, pp. 283–295, 2005.
[4] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, pp. 1–11.
[5] A. B. Hassanat, V. Prasath, M. A. Abbadi, S. A. Abu-Qdari, and H. Faris, "An improved genetic algorithm with a new initialization mechanism based on regression techniques," *Information*, vol. 9, no. 7, 2018, Art. no. 167.
[6] W. Mkaouer *et al.*, "Many-objective software remodularization using NSGA-III," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 3, pp. 1–45, 2015.
[7] S. M. H. Dehaghani and N. Hajrahimi, "Which factors affect software projects maintenance cost more?," *Acta Informatica Medica*, vol. 21, no. 1, pp. 63–66, 2013.
[8] W. E. Wong, J. R. Horgan, M. Syring, W. Zage, and D. Zage, "Applying design metrics to a large-scale software system," in *Proc. 9th Int. Symp. Softw. Rel. Eng. (Cat. No. 98TB100257)*, 1998, pp. 273–282.
[9] E. Tempero, T. Gorschek, and L. Angelis, "Barriers to refactoring," *Commun. ACM*, vol. 60, no. 10, pp. 54–61, 2017.
[10] A. A. B. Baqais and M. Alshayeb, "Automatic software refactoring: A systematic literature review," *Softw. Qual. J.*, vol. 28, no. 2, pp. 459–502, 2020.
[11] T. Mariani and S. R. Vergilio, "A systematic review on search-based refactoring," *Inf. Softw. Technol.*, vol. 83, pp. 14–34, 2017.
[12] 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, Apr. 2002.
[13] M. Harman, "Software engineering: An ideal set of challenges for evolutionary computation," in *Proc. 15th Annu. Conf. Companion Genet. Evol. Comput.*, 2013, pp. 1759–1760.

[14] S. Sarkar, A. C. Kak, and G. M. Rama, "Metrics for measuring the quality of modularization of large-scale object-oriented software," *IEEE Trans. Softw. Eng.*, vol. 34, no. 5, pp. 700–720, Sep./Oct. 2008.

[15] M. Zhao, D. Li, L. Zhang, H. Liu, and S.-Y. Lee, "Optimization scheduling design of monitoring resources using a process-improved adaptive genetic algorithm," in *Proc. 7th Int. Symp. Syst. Softw. Rel.*, 2021, pp. 167–177.

[16] T. Vernazza, G. Granatella, G. Succi, L. Benedicenti, and M. Mintchev, "Defining metrics for software components," in *Proc. World Multiconference Systemics, Cybern. Inform.*, vol. 11, 2000, pp. 16–23.

[17] L. Ponisio and O. Nierstrasz, "Using contextual information to assess package cohesion," *Inst. Appl. Math. Comput. Sci.*, 2006.

[18] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Trans. Softw. Eng.*, vol. 37, no. 2, pp. 264–282, Mar./Apr. 2011.

[19] G. Bavota, F. Carnevale, A. De Lucia, M. Di Penta, and R. Oliveto, "Putting the developer in-the-loop: An interactive GA for software remodularization," in *Proc. Int. Symp. Search Based Softw. Eng.*, 2012, pp. 75–89.

[20] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using cohesion and coupling for software remodularization: Is it enough?," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 3, pp. 1–28, 2016.

[21] S.-J. Huang and N.-H. Chiu, "Optimization of analogy weights by genetic algorithm for software effort estimation," *Inf. Softw. Technol.*, vol. 48, no. 11, pp. 1034–1045, 2006.

[22] F. Ferrucci, C. Gravino, R. Oliveto, and F. Sarro, "Genetic programming for effort estimation: An analysis of the impact of different fitness functions," in *Proc. 2nd Int. Symp. Search Based Softw. Eng.*, 2010, pp. 89–98.

[23] S. Stevanetic and U. Zdun, "Software metrics for measuring the understandability of architectural structures: A systematic mapping study," in *Proc. 19th Int. Conf. Eval. Assessment Softw. Eng.*, 2015, pp. 1–14.

[24] Z. Li, P. Liang, P. Avgeriou, N. Guelfi, and A. Ampatzoglou, "An empirical investigation of modularity metrics for indicating architectural technical debt," in *Proc. 10th Int. ACM SIGSOFT Conf. Qual. Softw. Architectures*, 2014, pp. 119–128.

[25] Y. Zhao, Y. Yang, H. Lu, Y. Zhou, Q. Song, and B. Xu, "An empirical analysis of package-modularization metrics: Implications for software fault-proneness," *Inf. Softw. Technol.*, vol. 57, pp. 186–203, 2015.

[26] V. B. Misic, "Cohesion is structural, coherence is functional: Different views, different measures," in *Proc. 7th Int. Softw. Metrics Symp.*, 2001, pp. 135–144.

[27] H. Abdeen, S. Ducasse, and H. Sahraoui, "Modularization metrics: Assessing package organization in legacy large object-oriented software," in *Proc. 18th Work. Conf. Reverse Eng.*, 2011, pp. 394–398.

[28] T. Zhou, B. Xu, L. Shi, Y. Zhou, and L. Chen, "Measuring package cohesion based on context," in *Proc. IEEE Int. Workshop Semantic Comput. Syst.*, 2008, pp. 127–132.

[29] J. Gorman, "OO design principles & metrics," *Online Verfügbar Unter, Zuletzt Geprüft Amer.*, vol. 15, 2006, Art. no. 2009. Available: http://www.parlezuml.com/metrics/OO%20Design%20Principles%20&%20Metrics.pdf

[30] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in Statistics*. Berlin, Germany: Springer, 1992, pp. 196–202.

[31] 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.

[32] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. ÓCinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2014, pp. 331–336.

[33] J. A. Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Inf. Softw. Technol.*, vol. 58, pp. 231–249, 2015.

[34] W. F. Opdyke, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *Proc. Symp. Object-Oriented Program. Emphasizing Practical Appl.*, 1990.

[35] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*," Berkeley, CA, USA: Addison-Wesley, 1999.

[36] H. Liu, Y. Wu, W. Liu, Q. Liu, and C. Li, "Domino effect: Move more methods once a method is moved," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reengineering*, vol. 1, 2016, pp. 1–12.

[37] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, "Deep learning based code smell detection," *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 1811–1837, Sep. 2021.

[38] R. Terra, M. T. Valente, S. Miranda, and V. Sales, "Jmove: A novel heuristic and tool to detect move method refactoring opportunities," *J. Syst. Softw.*, vol. 138, pp. 19–36, 2018.

[39] E. Murphy-Hill and A. P. Black, "Programmer-friendly refactoring errors," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1417–1431, Nov./Dec. 2012.

[40] X. Ge and E. Murphy-Hill, "Manual refactoring changes with automated refactoring validation," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1095–1105.

[41] Y. Wang, H. Yu, Z. Zhu, W. Zhang, and Y. Zhao, "Automatic software refactoring via weighted clustering in method-level networks," *IEEE Trans. Softw. Eng.*, vol. 44, no. 3, pp. 202–236, Mar. 2018.

[42] B. Pourasghar, H. Izadkhah, A. Isazadeh, and S. Lotfi, "A graph-based clustering algorithm for software systems modularization," *Inf. Softw. Technol.*, vol. 133, 2021, Art. no. 106469.

[43] L. Jiang, H. Liu, and H. Jiang, "Machine learning based recommendation of method names: How far are we," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2019, pp. 602–614.

[44] A. S. Nyamawe, H. Liu, N. Niu, Q. Umer, and Z. Niu, "Feature requests-based recommendation of software refactorings," *Empirical Softw. Eng.*, vol. 25, no. 5, pp. 4315–4347, 2020.

[45] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *Proc. 6th Int. Workshop Prog. Comprehension (Cat. No 98TB100242)*, 1998, pp. 45–52.

[46] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 193–208, Mar. 2006.

[47] B. S. Mitchell and S. Mancoridis, "On the evaluation of the bunch search-based software modularization algorithm," *Soft Comput.*, vol. 12, no. 1, pp. 77–93, 2008.

[48] O. Seng, M. Bauer, M. Biehl, and G. Pache, "Search-based improvement of subsystem decompositions," in *Proc. 7th Annu. Conf. Genet. Evol. Comput.*, 2005, pp. 1045–1051.

[49] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Proc. 8th Annu. Conf. Genet. Evol. Comput.*, 2006, pp. 1909–1916.

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

[51] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *Proc. IEEE 19th Int. Conf. Prog. Comprehension*, 2011, pp. 81–90.

[52] A. Ouni, M. Kessentini, and H. Sahraoui, "Search-based refactoring using recorded code changes," in *Proc. 17th Eur. Conf. Softw. Maintenance Reengineering*, 2013, pp. 221–230.

[53] H. Abdeen, H. Sahraoui, O. Shata, N. Anquetil, and S. Ducasse, "Towards automatically improving package structure while respecting original design decisions," in *Proc. 20th Work. Conf. Reverse Eng.*, 2013, pp. 212–221.

[54] M. De Stefano, F. Pecorelli, D. A. Tamburri, F. Palomba, and A. De Lucia, "Refactoring recommendations based on the optimization of socio-technical congruence," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2020, pp. 794–796.

[55] S. Kaur, L. K. Awasthi, and A. Sangal, "A brief review on multi-objective software refactoring and a new method for its recommendation," *Arch. Comput. Methods Eng.*, vol. 28, no. 4, pp. 3087–3111, 2021.

[56] M. Mohan and D. Greer, "Using a many-objective approach to investigate automated refactoring," *Inf. Softw. Technol.*, vol. 112, pp. 83–101, 2019.

[57] D. Hou, Y. Yin, Q. Su, and L. Liu, "Software refactoring scheme based on NSGA-II algorithm," in *Proc. 7th Int. Conf. Dependable Syst. Appl.*, 2020, pp. 447–452.

[58] T. Maikantis *et al.*, "Software architecture reconstruction via a genetic algorithm: Applying the move class refactoring," in *Proc. 24th Pan-Hellenic Conf. Informat.*, 2020, pp. 135–139.

[59] C. Abid, D. Rzig, T. Ferreira, M. Kessentini, and T. Sharma, "X-SBR: On the use of the history of refactorings for explainable search-based refactoring and intelligent change operators," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2021.3105037.

[60] C. Abid, J. Ivers, T. D. N. Ferreira, M. Kessentini, F. E. Kahla, and I. Ozkaya, "Intelligent change operators for multi-objective refactoring," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2021, pp. 768–780.