Putting the Developer in-the-loop: an Interactive GA for Software Re-Modularization

Gabriele Bavota[1], Filomena Carnevale[1], Andrea De Lucia[1]
Massimiliano Di Penta[2], Rocco Oliveto[3]

[1] University of Salerno, Via Ponte don Melillo, 84084 Fisciano (SA), Italy
[2] University of Sannio, Palazzo ex Poste, Via Traiano, 82100 Benevento, Italy
[3] University of Molise, Contrada Fonte Lappone, 86090 Pesche (IS), Italy
gbavota@unisa.it, flmn.carnevale@gmail.com, adelucia@unisa.it,
dipenta@unisannio.it, rocco.oliveto@unimol.it

Abstract. This paper proposes the use of Interactive Genetic Algorithms (IGAs) to integrate developer's knowledge in a re-modularization task. Specifically, the proposed algorithm uses a fitness composed of automatically-evaluated factors—accounting for the modularization quality achieved by the solution—and a human-evaluated factor, penalizing cases where the way re-modularization places components into modules is considered meaningless by the developer.
The proposed approach has been evaluated to re-modularize two software systems, SMOS and GESA. The obtained results indicate that IGA is able to produce solutions that, from a developer's perspective, are more meaningful than those generated using the full-automated GA. While keeping feedback into account, the approach does not sacrifice the modularization quality, and may work requiring a very limited set of feedback only, thus allowing its application also for large systems without requiring a substantial human effort.

1 Introduction

Software is naturally subject to change activities aiming at fixing bugs or introducing new features. Very often, such activities are conducted within a very limited time frame, and with a limited availability of software design documentation. Change activities tend to "erode" the original design of the system. Such a design erosion mirrors a reduction of the cohesiveness of a module, the increment of the coupling between various modules and, therefore, makes the system harder to be maintained or, possibly, more fault-prone [8]. For this reason, various automatic approaches, aimed at supporting source code re-modularization, have been proposed in literature (see e.g., [11, 14, 20]. The underlying idea of such approaches is to (i) group together in a module highly cohesive source code components, where the cohesiveness is measured in terms of intra-module links; and (ii) reduce the coupling between modules, where the coupling is measured in terms of inter-module dependencies. Such approaches use various techniques,

such as clustering [1, 11, 20], formal concept analysis [15] or search-based optimization techniques [13, 14] to find (near) optimal solutions for such objectives, e.g., cohesion and coupling.

While automatic re-modularization approaches proved to be very effective to increase cohesiveness and reduce coupling of software modules, they do not take into account developers' knowledge when deciding to group together (or not) certain components. For example, a developer may decide to place a function in a given module even if, in its current implementation, the function does not communicate a lot with other functions in the same module. This is because the developer is aware that, in future releases, such a function will strongly interact with the rest of the module. Similarly, a developer may decide that two functions must be placed in two different modules even if they communicate. This is because the two functions have different responsibilities and are used to manage semantically different parts of the system. In the past, some authors proposed approaches to account for developers' knowledge in software re-modularization [6]. However, such approaches assume the availability of a whole set of constraints before the re-modularization starts. This is often difficult to be achieved, especially for very large systems.

This paper proposes the use of Interactive Genetic Algorithms (IGAs) [17] to integrate, into a re-modularization approach, a mechanism allowing developers to feed-back automatically produced re-modularizations. IGAs are a variant of Genetic Algorithms (GAs) in which the fitness function is partially or entirely evaluated by a human while the GA evolves. Recently, IGAs have been applied to software engineering problems such as requirement prioritization [18] or upstream software design [16]. In our approach, part of the fitness (capturing aspects such as intra-module, extra-module dependencies, or modularization quality) is automatically evaluated, while the human adds penalties for artifacts that are not where they should be. Summarizing, the specific contributions of the paper are:

1. Different variants of IGAs, allowing the integration of feedback provided by developers upon solutions produced during the GA evolution. Specifically, the paper presents the integration of feedback in both a single-objective GA, using the Modularization Quality (MQ) measure [13], and a multi-objective GA proposed by Praditwong *et al.* [14].
2. The empirical evaluation of the proposed IGAs over two software systems. Although IGAs are conceived to allow a "live" feedback seeding, in this paper we simulated such a mechanism using constraints randomly identified from the actual system design. Results indicate that the IGAs are able to produce re-modularizations that better reflect the developer intents, without however sacrificing the modularization quality.

The paper is organized as follows. Section 2 describes the related work, while Section 3 describes the proposed IGA-based re-modularization. Section 4 reports the empirical study conducted to evaluate the proposed approach, while Section 5 concludes the paper and outlines directions for future work.

2 Background and Related Work

Several approaches have been proposed in the literature to support software re-modularization. Promising results have been achieved using clustering algorithms [1, 11, 20] and formal concept analysis [15]. In the following we focus only on search-based approaches.

Mancoridis *et al.* [10] introduce a search-based approach using hill-climbing based clustering to identify the modularization of a software system. This technique is implemented in Bunch [13], a tool supporting automatic system decomposition. To formulate software re-modularization as a search problem, Mancoridis *et al.* define (i) a representation of the problem to be solved (i.e., software module clustering) and (ii) a way to evaluate the modularizations generated by the hill-climbing algorithm. Specifically, the system is represented by the Module Dependency Graph (MDG), a language independent representation of the structure of the code components and relations [10]. The MDG can be seen as a graph where nodes represent the system entities to be clustered and edges represent the relationships among these entities. An MDG can be weighted (i.e., a weight on an edge measures the strength of the relationship between two entities) or unweighted (i.e., all the relationships have the same weight).

Starting from the MDG (weighted or unweighted), the output of a software module clustering algorithm is represented by a partition of this graph. A good partition of an MDG should be composed by clusters of nodes having (i) high dependencies among nodes belonging to the same cluster (i.e., high cohesion), and (ii) few dependencies among nodes belonging to different clusters (i.e., low coupling). To capture these two desirable properties of the system decompositions (and thus, to evaluate the modularizations generated by Bunch) Mancoridis *et al.* [10] define the Modularization Quality (MQ) metric as:

$$MQ = \begin{cases} (\frac{1}{k}\sum_{i=1}^{k} A_i) - (\frac{1}{\frac{k(k-1)}{2}}\sum_{i,j=1}^{k} E_{i,j}) & \text{if } k > 1 \\ A_1 & \text{if } k = 1 \end{cases}$$

where A_i is the Intra-Connectivity (i.e., cohesion) of the i^{th} cluster and $E_{i,j}$ is the Inter-Connectivity (i.e., coupling) between the i^{th} and the j^{th} clusters. The Intra-Connectivity is based on the number of intra-edges, that is the relationships (i.e., edges) existing between entities (i.e., nodes) belonging to the same cluster, while the Inter-Connectivity is captured by the number of inter-edges, i.e., relationships existing between entities belonging to different clusters.

Single-objective genetic algorithms have been used to improve the subsystem decomposition of a software system by Doval *et al.* [7]. The objective function is defined using a combination of quality metrics, e.g., coupling, cohesion, and complexity. However, hill-climbing have been demonstrated to ensure higher quality and more stable solutions than a single objective genetic algorithm [12]. Praditwong *et al.* [14] introduce two multi-objective formulations of the software re-modularization problem, in which several different objectives are represented separately. The two formulations slightly differ for the objectives embedded in the multi-objective function. The first formulation–named Maximizing Cluster

Approach (MCA)—has the following objectives: (i) maximizing the sum of intra-edges of all clusters, (ii) minimizing the sum of inter-edges of all clusters, (iii) maximizing MQ, (iv) maximizing the number of clusters, and (v) minimizing the number of isolated clusters (i.e., clusters composed by only one class). The second formulation—named Equal-Size Cluster Approach (ECA)—attempts at producing a modularization containing clusters of roughly equal size. Its objectives are exactly the same as MCA, except for the fifth one (i.e., minimizing the number of isolated clusters) that is replaced with (v) minimizing the difference between the maximum and minimum number of entities in a cluster. The authors compared their algorithms with Bunch. The conducted experimentation provides evidence that the multi-objective approach produces significantly better solutions than the existing single-objective approach though with a higher processing cost.

Based on the results achieved by Praditwong *et al.* [14], this paper defines an interactive version of the single-objective GA and of the multi-objective GA (based on the MCA algorithm). This allows to analyze the benefits provided by developers' feedback to solve a re-modularization problem.

3 The Proposed Interactive Genetic Algorithms

This section describes the IGA we use to integrate software engineers' feedback into the single-objective [10] and multi-objective [14] re-modularization process.

3.1 Solution Representation, Operators, and Fitness Function

The solution representation (chromosome) and GA operators are the same for both single- and multi-objective GAs. Given a software system composed of n software components (e.g., classes) the chromosome is represented as a n-sized integer array, where the value $0 < v \leq n$ of the i^{th} element indicates the cluster which the i^{th} component is assigned. A solution with the same value (whatever it is) for all elements means that all software components are placed in the same cluster, while a solution with all possible values (from 1 to n) means that each cluster is composed of one component only.

The crossover operator is a one-point crossover, while the mutation operator randomly identifies a gene (i.e., a position in the array), and modifies it by assigning to it a random value $0 < v \leq n$. This means moving a component to cluster v. The selection operator is the roulette-wheel selection.

The single-objective GA uses as fitness function (to be maximized) the MQ metric, while as said in Section 2 the multi-objective GA—implemented as Non-Dominating Sorting Genetic Algorithm (NSGA-II) [5]—considers five different objectives, related to maximizing MQ, intra-cluster connectivity and number of clusters, and minimizing the inter-cluster connectivity and the number of isolated clusters.

Algorithm 1 R-IGA: IGA for providing feedback about pairs of components.

```
1: for i = 1 ... nInteractions do
2:     Evolve GA for nGens generations
3:     Select the solution having the highest MQ
4:     for j = 1 ... nFeedback do
5:         Randomly select two components c_i and c_j
6:         Ask the developer whether c_i and c_j must go together or kept separate
7:     end for
8:     Repair the solution to meet the feedback
9:     Create a new GA population using the repaired solution as starting point
10: end for
11: Continue (non-interactive) GA evolution until it converges or it reaches maxGens
```

3.2 Single-Objective Interactive GAs

The basic idea of the IGA is to periodically add a constraint to the GA such that some specific components shall be put in a given cluster among those created so far. Thus, the IGA evolves exactly as the non-interactive GA. Then, every *nGens* generations, the best individual is selected and shown to the developer. Then, the developer analyzes the proposed solutions and provides feedback (which can be seen as constraints to the re-modularization problem), indicating that certain components shall be moved from a cluster to another. After enacting the developer's indications, a new GA population is created from such a best solution, and then the GA evolves for further *nGens* generations, keeping into account the provided constraints. One crucial point is choosing how to guide the developer to provide feedback. In principle, one could ask developers any possible kind of feedback. However, this would make the developer's task quite difficult. For this reason, we propose to guide developers in providing feedback, by means of two different kinds of IGAs. The first one—referred to as R-IGA and described by Algorithm 1—takes the best solution produced by the GA, randomly selects two components (from the same cluster or from different clusters), and then asks the developer whether, in the new solutions to be generated, such components must be placed in the same cluster (i.e., stay together) or whether they should be kept separated.

As the algorithm indicates, every *nGens* generations the developer is asked to provide feedback about a number *nFeedback* of component pairs from the best solution (in terms of MQ) contained in the current population. The feedback can either be (i) "c_i and c_j shall stay together" or (ii) "c_i and c_j shall be kept separate". After feedback is provided, the solution is repaired by enforcing the constraints, e.g., by randomly moving one of c_i and c_j away if the constraint tells that they shall be kept separate. After all *nFeedback* have been provided, a new population is created by randomly mutating such a repaired solution. Then, the GA starts again. When creating the new population and when evolving it, the GA shall ensure that the new produced solutions meet the feedback collected so far. Hence, we add a penalty factor to the fitness function (as proposed by Coello Coello [3]), aiming at penalizing solutions violating the constraints imposed by

the developers. Given $CS \equiv cs_1, \ldots cs_m$ the set of feedback collected by the users, the fitness $F(s)$ for a solution s is computed as follows:

$$F(s) = \frac{MQ(s)}{1 + k \cdot \sum_{i=1}^{m} vcs_{i,s}}$$

where $k > 0$ is an integer constant weighting the importance of the feedback penalty, and $vcs_{i,s}$ is equal to one if solution s violates cs_i, zero otherwise. After *nInteractions* have been performed, the GA continues its evolution in a non-interactive way until it reaches stability or the maximum number of generations. One consideration needs to be made about the selection of the pairs for which asking feedback. While in our experiments the selection is random (see Section 4), in a realistic scenario the developer could pick component pairs based on her knowledge, or else further heuristics could be used for such purposes.

The second IGA we propose—called IC-IGA and described by Algorithm 2—focuses on specific parts of the re-modularization produced by the GA. Among others, very small clusters should be subject to manual changes by the developer. In fact, automatic re-modularization approaches often tend to create a large number of many small clusters, that seldom reflect the actual or desired system decomposition. For this reason, the second variant of our IGA asks feedback on the *nClusters* smallest clusters in the best solution (in terms of MQ). Then, for each of these clusters, if it is an isolated cluster (i.e., composed of one component only), the developer is asked to specify a different cluster where the isolated component must be placed while for not isolated clusters the developer is asked to specify for each pair of components whether they must stay together or not. It is worth noting that the developer does not specify the cluster, she rather indicates whether, when moving such components to a different (randomly selected) cluster, they should be moved together or it must be made sure they are kept separate. Besides the nature of the collected feedback, it works similarly to R-IGA (the fitness function does not change).

Clearly, several other kinds of heuristics could be used to ask feedback to the developer (e.g., a combination of the two approaches presented in this paper, with feedback required on both random couples of elements and on elements belonging to small clusters). However, this is out of the scope of this paper.

3.3 Multi-Objective Interactive GAs

The multi-objective variants of our IGA are quite similar to the single-objective ones. Also in this case, we propose one— referred as R-IMGA—where feedback is provided on randomly selected pairs of components, and one—referred as IC-IMGA—where feedback is provided on components belonging to isolated (or smallest) clusters.

A crucial point in the multi-objective variant is the selection of the best individual for which the developer shall provide feedback and, after applying the feedback, to be used for generating the new GA population. The single-objective GA selects individual having the highest MQ, i.e., the highest fitness

Algorithm 2 IC-IGA: IGA for handling small and isolated clusters

```
1: for i = 1 ... nInteractions do
2:     Evolve the GA for nGens generations
3:     Select the solution having the highest MQ
4:     Find the nClusters smallest clusters and store them in C
5:     for all c_i ∈ C do
6:         if c_i is an isolated cluster then
7:             Specify the cluster where the component must be placed
8:         else
9:             Specify for each component pair whether the components must stay together or not
10:        end if
11:        Repair the solution to meet the feedback
12:        Create a new GA population using the repaired solution as starting point
13:    end for
14: end for
15: Continue (non-interactive) GA evolution until it converges or it reaches maxGens
```

value. As for the multi-objective GA, we again select solutions with the highest MQ, although they might not be the ones with the highest values for the other objectives. The motivation is similar to the one of Praditwong *et al.* [14], which used MQ to select the best solution in the NSGA-II Pareto fronts. While other objectives are useful to drive the population evolution, MQ is a measure that characterize the "overall" quality of a re-modularization solution, thus can be used to select—among a set of Pareto-optimal solutions—the one that would likely better pursue the re-modularization objectives. Finally, to ensure that the new produced solutions meet the feedback provided by the developer, also for the multi-objective GA we add a penalty factor to each fitness function following the same approach adopted for the single-objective GA.

4 Empirical Evaluation

This section reports the design and the results of the study we conducted to compare the different variants of IGAs with their non-interactive counterparts in the context of software re-modularization. The experimentation has been carried out on an industrial project, namely GESA, and on a software system, SMOS, developed by a team of Master students at the University of Salerno (Italy) during their industrial traineeship. GESA automates the most important activities in the management of University courses, like timetable creation and classroom allocation. It is operational since 2007 at the University of Molise (Italy)[4]. SMOS is a software developed for high schools, and offers a set of features aimed at simplifying the communications between the school and the student's parents. Table 1 reports the size, in terms of KLOC, number of classes, and number

[4] http://www.distat.unimol.it/gesa/

Table 1. Characteristics of the software systems used in the case study.

System	KLOC	#Classes	#Packages	Quality of modularization			
				Isolated clusters	Intra-edges	Inter-edges	MQ
GESA 2.0	46	297	22	1	330	4,472	2.78
SMOS 1.0	23	121	12	2	155	1,158	2.18

of packages, and the versions of the object systems. The table also reports the values of some metrics (e.g., MQ) to measure the modularization quality of the systems. Such metrics are also used as fitness functions in the implementation of the different variants of GAs (interactive and not).

4.1 How is Feedback Provided?

Since we are not performing a user study, we simulated the developer by automatically generating feedback extracted from the original design of the object systems (similarly to [18]). This means that every time the IGA asks whether two classes must go together or be kept separate (or it asks to specify the cluster where the class must be placed) a tool simulated the developer response by finding the answer in the original design. We believe this feedback would be representative of an expert's behavior, since the two object systems have a good modularization quality and have been previously used as gold standard to test other re-modularization approaches [2].

4.2 Study Planning and Analysis Method

Our study aims at answering the following research question:

> **RQ**: *How do IGAs perform—compared to non-interactive GAs—in terms of quality and meaningfulness of the produced re-modularizations?*

To answer this research question, we compare the modularizations achieved with the different variants of IGAs (R-IGA, IC-IGA, R-IMGA, and IC-IMGA) with those achieved applying canonical GA and MGA. For each algorithm, an initial population is randomly generated. All the algorithms have been executed 30 times on each object system to account the inherent randomness of GAs. For all the algorithms we used the same configuration. We calibrated the various parameters of the GA similarly to what done by Praditwong *et al.* [14], by properly adapting some of them to the characteristics of our object systems. Other calibration—mainly related to the number of times the GA interacts with the software engineer, the number of feedback provided, and the number of generations between one interaction and the subsequent one—were calibrated by trial-and-error for our object systems. Therefore, a proper calibration might be needed for larger systems. Specifically:

– We use a population size of n individuals for systems having $n > 150$ software components, a population of $2 \cdot n$ for smaller systems. Praditwong *et al.* [14] used a population size of $10 \cdot n$ instead, because the bigger system used in their study was considerably smaller than the one used in our study (i.e., GESA

with 297 classes). Similarly, we consider a maximum number of generations *maxGen* equal to $20 \cdot n$ for systems having $n > 150$ components, and equal to $50 \cdot n$ for smaller systems. Also for the maximum number of generations, Praditwong *et al.* used in their experimentation a higher number (i.e., $200 \cdot n$).

- The crossover probability is set to 0.8, while the mutation probability to $0.004 \cdot log_2(n)$ (as also done by Praditwong *et al.* [14]).
- The number of times (*nInteractions*) the GA stops the evolution and asks for an interaction is set to 5;
- The number of generations (*nGens*) between one interaction and the subsequent one is set to 10;
- The number of class pairs (*nFeedback*) for which Algorithm 1 asks the developer for a feedback every time is set to 3;
- The number of small/isolated clusters (*nClusters*) for which Algorithm 2 asks the developer for a feedback every time is set to 3;
- The weight k of the penalty factor in the fitness functions is set to 1.

As it can be noticed, the maximum number of feedback provided is equal to $5 \cdot 3 = 15$ class pairs for Algorithm 1 and $c \cdot 3$ (where c is the number of analyzed classes) for Algorithm 2. This might appears as a small amount of feedback, compared to the total number of classes of the object systems. However, in this context we are interested to evaluate how the IGA would have worked with a limited—i.e., cheap and feasible for the developer—interaction.

One parameter used by all the algorithms is the maximum number of clusters to extract. To the best of our knowledge this parameter has not been described in previous works. However, it is crucial for the setting of a GA. Defining the maximum number of clusters *a priori* is not a trivial task. Such a number highly depends by the system under analysis. In this paper we experimented two different heuristics. Specifically, we set the maximum number of clusters to n and $n/2$, respectively, where n is the number of classes in the system to be re-modularized.

When analyzing results, we compare the ability of GAs and IGAs to reach a fair trade-off between the optimization of some quality metrics (that is the main objective of GAs applied to software re-modularization) and the closeness of the proposed partitions to an authoritative one (and thus, their meaningfulness). Note that we use the original design of the object systems as authoritative partition. This choice is justified by the good modularization quality of the object systems, that have been previously used as gold standard to assess other re-modularization approaches [2]. On the one hand, to analyze the impact of provided feedback from the quality metrics point-of-view, we use four quality metrics previously adopted by Praditwong *et al.* [14], namely MQ, intra-edges, inter-edges, and number of isolated clusters. On the other hand, to measure the meaningfulness of the modularizations proposed by the experimented algorithms, we compute the MoJo eFfectiveness Measure (MoJoFM) [19] between the original modularization of the object systems (authoritative partitions) and that proposed by the algorithms. The MoJoFM is a normalized variant of the

MoJo distance and it is computed as follows:

$$MoJoFM(A,B) = 100 - (\frac{mno(A,B)}{max(mno(\forall A,B))} \times 100)$$

where $mno(A,B)$ is the minimum number of *Move* or *Join* operations one needs to perform in order to transform the partition A into B, and $max(mno(\forall\, A,B)$ is the maximum possible distance of any partition A from the gold standard partition B. Thus, $MoJoFM$ returns 0 if a clustering algorithm produces the farthest partition away from the gold standard; it returns 100 if a clustering algorithm produces exactly the gold standard.

We also statistically analyze whether the results achieved by different algorithms (interactive and not) significantly differ in terms of quality metrics or authoritativeness of the modularizations. In particular, the values of all the employed metrics (e.g., MQ) achieved in the 30 runs by two algorithms are statistically compared using the *Mann-Whitney* test [4]. In all our statistical tests we reject the null hypotheses for p-values < 0.05 (i.e., we accept a 5% chance of rejecting a null hypothesis when it is true [4]). We also estimate the magnitude of the difference between the employed metrics. We use the Cliff's Delta (or d), a non-parametric effect size measure [9] for ordinal data. The effect size is small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$ [9].

4.3 Analysis of Results

This section discusses the results achieved in our study aiming at responding to our research question. Working data sets are available for replication purposes[5].

Tables 2 and 3 report the descriptive statistics of the measured quality metrics. The analysis of Table 2 highlights that on GESA the interactive GAs, in most cases, achieve better results than their non-interactive counterparts. Note that this is true (i) for both single- and multi-objective algorithms, (ii) using both n or $n/2$ as maximum number of clusters, and (iii) using both the random (R-IGA and R-IMGA) or the isolated cluster (IC-IGA and IC-IMGA) heuristic to provide feedback to the interactive algorithm; only R-IGA and R-IMGA for GESA go slightly worse than their non-interactive counterparts. These better performances hold for all the exploited quality metrics. Moreover, the number of clusters generated by the interactive algorithms is much more close to the effective number of clusters of the original system decomposition (i.e., 22). The non-interactive GA and MGA always propose modularizations having a very high number of clusters, mostly composed of few classes. For example, GA[n] organizes the 297 classes of GESA into (on average) 149 clusters, 72 of which (on average) are isolated clusters, i.e., clusters containing only one class. The best configuration is MGA[n/2] that, however, still produces a quite fragmented system decomposition, clustering the 297 classes into an average of 108 clusters, 32 of which isolated. Even if this kind of systems decomposition could (near)

[5] http://www.distat.unimol.it/reports/IGA-remodularization/

Table 2. GESA: descriptive statistics of the measured quality metrics.

Algorithm	#Clusters			#Isolated Clusters			intra-edges			inter-edges			MQ		
	Mean	Med.	St. Dev.	Mean	Med.	St. Dev.	Mean	Med.	St. Dev.	Mean	Med.	St. Dev.	Mean	Med.	St. Dev.
GA[n]	149	159	5	72	72	7	62	63	5	5,009	5,007	10	3.94	3.95	0.22
IC-IGA[n]	79	72	26	22	12	19	421	439	177	4,290	4,254	353	4.86	4.97	0.36
R-IGA[n]	113	110	11	57	56	6	282	316	100	4,568	4,500	201	4.64	4.60	0.36
MGA[n]	155	156	7	83	84	9	36	34	9	5,060	5,064	17	2.04	2.00	0.34
IC-IMGA[n]	90	69	31	25	9	27	296	346	131	4,539	4,440	262	4.19	4.44	0.55
R-IMGA[n]	155	154	10	91	89	13	58	57	14	5,017	5,018	27	2.14	2.06	0.31
GA[n/2]	107	107	3	28	28	5	79	80	4	4,975	4,973	8	3.99	4.04	0.23
IC-IGA[n/2]	63	59	15	11	10	6	298	299	84	4,536	4,534	168	4.81	4.92	0.39
R-IGA[n/2]	88	89	8	29	29	5	359	315	193	4,415	4,502	386	4.65	4.71	0.29
MGA[n/2]	108	108	5	32	32	6	58	57	13	5,016	5,018	26	2.23	2.20	0.32
IC-IMGA[n/2]	59	51	26	10	6	10	324	329	183	4,485	4,475	366	3.97	4.09	0.61
R-IMGA[n/2]	111	112	4	41	40	6	111	104	35	4,911	4,924	71	2.12	2.08	0.22

Table 3. SMOS: descriptive statistics of the measured quality metrics.

Algorithm	#Clusters			#Isolated Clusters			intra-edges			inter-edges			MQ		
	Mean	Med.	St. Dev.	Mean	Med.	St. Dev.	Mean	Med.	St. Dev.	Mean	Med.	St. Dev.	Mean	Med.	St. Dev.
GA[n]	54	54	3	22	21	4	52	52	4	1,364	1,365	7	3.40	3.45	0.21
IC-IGA[n]	35	32	9	9	7	7	86	94	20	1,297	1,281	39	3.67	3.77	0.62
R-IGA[n]	42	41	4	15	15	4	76	77	10	1,316	1,315	19	3.78	3.77	0.22
MGA[n]	63	64	4	32	31	6	30	29	7	1,408	1,411	13	1.91	1.88	0.31
IC-IMGA[n]	37	35	7	10	8	6	82	86	20	1,305	1,296	41	2.66	2.68	0.39
R-IMGA[n]	63	64	4	37	37	5	45	45	10	1,379	1,378	20	1.97	1.95	0.30
GA[n/2]	42	42	3	11	11	3	62	63	4	1,344	1,342	8	3.68	3.68	0.21
IC-IGA[n/2]	20	18	7	1	1	1	114	118	29	1,240	1,233	58	3.00	2.84	0.36
R-IGA[n/2]	29	29	4	8	7	3	160	154	44	1,148	1,160	87	3.68	3.69	0.21
MGA[n/2]	46	47	4	15	15	3	50	47	14	1,368	1,375	28	2.10	2.04	0.31
IC-IMGA[n/2]	17	15	8	2	2	2	244	257	115	980	954	229	2.64	2.57	0.30
R-IMGA[n/2]	47	46	3	19	18	5	74	67	27	1,320	1,335	53	2.07	2.08	0.24

optimize some quality metrics, it represents a poor support for the developer during a software re-modularization task, requiring a substantial effort to manually refine the proposed modularization.

Results obtained for SMOS (Table 3), confirm what is seen for GESA. In addition, also on SMOS, the non-interactive GAs provide quite poor performances, especially in terms of number of clusters produced. Also in this case, the best configuration is obtained with MGA[n/2], that organizes the 121 classes of SMOS in 46 clusters (on average). Among them, 15 are isolated. Its interactive version (IC-IMGA[n/2]) is able to produce a more reasonable partition, composed of 17 clusters, of which only 2 are isolated.

All these considerations are also supported by statistical analyses (see Tables 4 and 5). Therefore, we can conclude that IGAs achieve better quality metrics value as compared to the non-interactive counterparts. This result is quite surprising, since the feedback provided to the IGAs in our experimentation are not targeted to improve some quality metrics (as the fitness function is), but only to integrate in the GA the developers' knowledge. It is worth noting that the feedback provided by the developers help the GA to sensibly reduce the number of produced clusters.

Concerning the authoritativeness of the experimented algorithms, the achieved values of MoJoFM on GESA and SMOS are reported in Table 6. Given the results obtained, it is not surprising that the modularizations proposed by the non-interactive GAs (both single- and multi- objective) are very far from the original design. The best results among the non-interactive GAs are achieved

Table 4. GESA: Results of the Mann-Whitney tests.

	MQ		# clusters		Isolated clusters		MoJoFM	
	p-value	d	p-value	d	p-value	d	p-value	d
GA[n] *vs.* IC-IGA[n]	0.00	0.92	0.00	-Inf	0.00	-0.96	0.00	0.91
GA[n] *vs.* R-IGA[n]	0.00	0.89	0.00	-0.97	0.00	-0.89	0.00	0.96
MGA[n] *vs.* IC-IMGA[n]	0.00	0.99	0.00	-0.99	0.00	-0.98	0.00	Inf
MGA[n] *vs.* R-IMGA[n]	0.34	0.21	0.88	-0.02	0.04	0.38	0.01	0.44
IC-IGA[n] *vs.* R-IGA[n]	0.02	-0.38	0.00	0.69	0.00	0.84	0.02	-0.39
IC-IMGA[n] *vs.* R-IMGA[n]	0.00	-0.99	0.00	0.99	0.00	0.99	0.00	-0.97
GA[n/2] *vs.* IC-IGA[n/2]	0.00	0.87	0.00	-Inf	0.00	-0.94	0.00	0.99
GA[n/2] *vs.* R-IGA[n/2]	0.00	0.92	0.00	-0.98	0.24	0.18	0.00	0.96
MGA[n/2] *vs.* IC-IMGA[n/2]	0.00	0.94	0.00	-0.93	0.00	-0.88	0.00	0.94
MGA[n/2] *vs.* R-IMGA[n/2]	0.14	-0.27	0.04	0.37	0.00	0.76	0.14	0.24
IC-IGA[n/2] *vs.* R-IGA[n/2]	0.04	-0.35	0.00	0.84	0.00	0.96	0.04	-0.33
IC-IMGA[n/2] *vs.* R-IMGA[n/2]	0.00	-0.96	0.00	0.97	0.00	0.98	0.00	-0.93

Table 5. SMOS: Results of the Mann-Whitney tests.

	MQ		# clusters		Isolated clusters		MoJoFM	
	p-value	d	p-value	d	p-value	d	p-value	d
GA[n] *vs.* IC-IGA[n]	0.00	0.43	0.00	-0.85	0.00	-0.83	0.00	0.94
GA[n] *vs.* R-IGA[n]	0.00	0.80	0.00	-0.96	0.00	-0.81	0.00	Inf
MGA[n] *vs.* IC-IMGA[n]	0.00	0.84	0.00	-1.00	0.00	-0.99	0.00	0.99
MGA[n] *vs.* R-IMGA[n]	1.00	0.10	1.00	-0.01	0.01	0.44	0.00	0.90
IC-IGA[n] *vs.* R-IGA[n]	0.90	-0.02	0.00	0.55	0.00	0.60	0.00	-0.70
IC-IMGA[n] *vs.* R-IMGA[n]	0.00	-0.83	0.00	1.00	0.00	1.00	0.00	-0.91
GA[n/2] *vs.* IC-IGA[n/2]	0.00	-0.84	0.00	-0.97	0.00	-1.00	0.00	0.97
GA[n/2] *vs.* R-IGA[n/2]	0.99	0.00	0.00	-0.99	0.00	-0.54	0.00	0.94
MGA[n/2] *vs.* IC-IMGA[n/2]	0.00	0.80	0.00	-0.99	0.00	-Inf	0.00	0.98
MGA[n/2] *vs.* R-IMGA[n/2]	1.00	0.00	0.69	0.14	0.00	0.61	0.01	0.45
IC-IGA[n/2] *vs.* R-IGA[n/2]	0.00	0.84	0.00	0.74	0.00	0.99	0.00	-0.72
IC-IMGA[n/2] *vs.* R-IMGA[n/2]	0.00	-0.87	0.00	1.00	0.00	Inf	0.00	-0.96

by GA[n/2] with MoJoFM equals to 21 on GESA and 37 on SMOS. Instead, thanks to the few feedback provided, the IGAs achieved much better results (as expected). The best performances are achieved using IC-MGA[n] with a maximum MoJoFM of 53 on GESA and 74 on SMOS. Also in this case, statistical analyses support our findings (see Tables 4 and 5).

To better understand what this difference between the performances of interactive and non-interactive GAs means from a practical point of view, Fig. 1 shows an example extracted from the re-modularization of the SMOS software system. The figure is organized in three parts. The first part (left side) shows how the subsystem *RegisterManagement* appears in the original package decomposition (i.e., which classes it contains) made by the SMOS's developers. This subsystem groups together all the classes in charge to manage information related to the scholar register (e.g., the students' delay, justifications for their absences and so on). The second part (middle) reports the decomposition of the classes contained in *RegisterManagement* proposed by the MGA. Note that some classes not belonging to the *RegisterManagement* were mixed to the original set

Table 6. Descriptive statistics of the MoJoFM achieved by the different algorithms.

Algorithm	MoJoFM				
	Mean	Med.	St. Dev.	Max	Min
GA[n]	14	14	2	19	11
IC-IGA[n]	29	33	8	38	13
R-IGA[n]	26	27	5	34	14
MGA[n]	10	11	2	13	7
IC-IMGA[n]	38	39	11	53	13
R-IMGA[n]	12	12	3	22	8
GA[n/2]	18	17	1	21	15
IC-IGA[n/2]	29	29	6	38	20
R-IGA[n/2]	25	25	4	32	18
MGA[n/2]	14	14	2	18	10
IC-IMGA[n/2]	38	39	11	53	13
R-IMGA[n/2]	15	15	2	20	11

(a) GESA

Algorithm	MoJoFM				
	Mean	Med.	St. Dev.	Max	Min
GA[n]	26	27	2.8	30	21
IC-IGA[n]	53	54	6.4	62	36
R-IGA[n]	39	39	4.7	48	31
MGA[n]	17	16	3.1	23	12
IC-IMGA[n]	59	62	12	74	22
R-IMGA[n]	26	27	4.6	38	15
GA[n/2]	31	31	2.6	37	25
IC-IGA[n/2]	52	53	8.2	62	30
R-IGA[n/2]	42	41	6.2	57	32
MGA[n/2]	24	23	3.5	32	20
IC-IMGA[n/2]	43	43	7.8	62	22
R-IMGA[n/2]	28	28	4.5	36	20

(b) SMOS

of classes. These classes are reported in light gray. Finally, the third part (right side) shows the decomposition of the classes contained in *RegisterManagement* proposed by the IC-IMGA. Also in this case, classes not belonging to the original *RegisterManagement* package are reported in light gray. As we can see, the original package decomposition groups 31 classes in the *RegisterManagement* package. When applying MGA, these 31 classes are spread into 27 packages, 13 of which are singleton packages. As for the remaining 14 they usually contain some classes of the *RegisterManagement* package mixed with other classes coming from different packages (light gray in Fig. 1). The solution provided by IC-IMGA is very different. In fact, IC-IMGA spreads the 31 classes in only 5 packages. Moreover, it groups together in one package 26 out of the 31 classes originally belonging to the *RegisterManagement* package. It is striking how much the partition proposed by IC-IMGA is closer to the original one resulting in a higher MoJoFM achieved by IC-IMGA with respect to MGA and thus, a more meaningful partitioning from a developer's point of view.

On summary, results of our study showed as the non-interactive GAs, in both their single- and multi- objective formulations, might produce modularizations that, albeit being good in terms of cohesion and coupling, strongly deviate from the developers' intent. This highlights the need for augmenting the GA with developers's knowledge trough IGAs. That is, IGAs would allow to obtain more meaningful solutions—that for our case studies are even better in terms of modularization quality—representing an acceptable starting point for a developer when performing a software re-modularization.

4.4 Threats to Validity

Threats to *construct validity* may essentially depend on the way we simulated the feedback. As said, we believe that simulating feedback from the original system design would be representative of a developer's behavior (similarly to [18]). Nevertheless, controlled experiments are needed to understand to what extent real developers are able to introduce their knowledge in the genetic algorithm through the feedback mechanism. Threats to *internal validity* can be related to the GAs parameter settings. For some of them we used parameters similar to previous studies [14], while for others we used a trial-and-error calibration procedure. Threats to *external validity* concerns the generalization of our results.

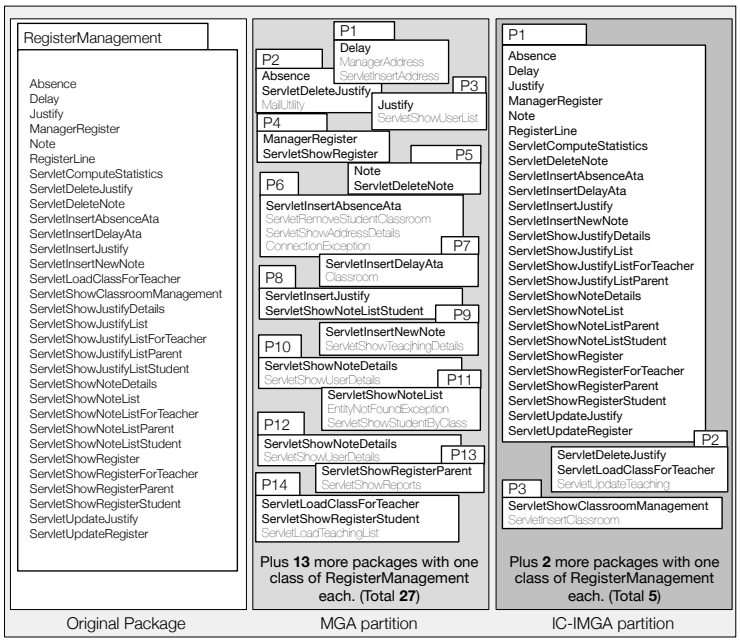

Fig. 1. MGA *vs* IC-IMGA in reconstructing the *RegisterManagement* package of SMOS.

Although we applied the approach on two realistic systems (one industrial system and one developed by students but actually used), further experimentation is needed, also for example on procedural (e.g., C) systems. Also, we evaluated the advantages of interactive feedback on GAs only, while it would be worthwhile to investigate it also for other heuristics such as hill-climbing. However, it is important to point out that in this paper we were interested to assess the *improvement* in the quality and meaningfulness of the produced solutions when using interactions (with respect to a similar algorithm without interaction), rather than in the absolute quality of the solutions.

5 Conclusion and Future Work

In this paper we proposed the use of IGAs to integrate developers' knowledge during software re-modularization activities. We implemented and experimented both single- and multi-objective IGAs comparing their performances with those achieved by their non-interactive counterparts. The achieved results show that the IGAs are able to propose re-modularizations (i) more meaningful from a developer's point-of-view, and (ii) not worse, and often even better in terms of modularization quality, with respect to those proposed by the non-interactive GAs. Also, IGAs is applicable with a limited number of feedback, and thus, require a relatively limited effort to the developer.

Future work will be devoted to replicate the empirical evaluation on further software systems and different heuristics such as hill-climbing. Also, we plan to evaluate the usefulness of IGAs during software re-modularization activities.

References

1. Anquetil, N., Lethbridge, T.: Experiments with clustering as a software remodularization method. In: WCRE. pp. 235–255 (1999)
2. Bavota, G., Lucia, A.D., Marcus, A., Oliveto, R.: Software re-modularization based on structural and semantic metrics. In: WCRE. pp. 195–204. (2010)
3. Coello Coello, C.A.: Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: A survey of the state of the art. Computer Methods in Applied Mechanics and Engineering 191(11-12) (2002)
4. Conover, W.J.: Practical Nonparametric Statistics. Wiley, 3rd edition. (1998)
5. Deb, K.: Multi-Objective Optimization Using Evolutionary Algorithms. Wiley (2001)
6. Di Penta, M., Neteler, M., Antoniol, G., Merlo, E.: A language-independent software renovation framework. JSS 77(3), pp. 225–240 (2005)
7. Doval, D., Mancoridis, S., Mitchell, B.S.: Automatic clustering of software systems using a genetic algorithm. In: STEP. pp. 73–82. (1999)
8. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional (1999)
9. Grissom, R.J., Kim, J.J.: Effect sizes for research: A broad practical approach. Lawrence Earlbaum Associates, 2nd edition. (2005)
10. Mancoridis, S., Mitchell, B.S., Rorres, C., Chen, Y.F., Gansner, E.R.: Using automatic clustering to produce high-level system organizations of source code. In: IWPC. pp. 45– (1998)
11. Maqbool, O., Babri, H.A.: Hierarchical clustering for software architecture recovery. IEEE TSE 33(11), 759–780 (2007)
12. Mitchell, B.S.: A Heuristic Search Approach to Solving the Software Clustering Problem. Ph.D. thesis, Drexel University, Philadelphia (2002)
13. Mitchell, B.S., Mancoridis, S.: On the automatic modularization of software systems using the bunch tool. IEEE TSE 32(3), 193–208 (2006)
14. Praditwong, K., Harman, M., Yao, X.: Software module clustering as a multi-objective search problem. IEEE TSE 37(2), 264–282 (2011)
15. Siff, M., Reps, T.W.: Identifying modules via concept analysis. IEEE TSE 25(6), 749–768 (1999)
16. Simons, C.L., Parmee, I.C., Gwynllyw, R.: Interactive, Evolutionary Search in Upstream Object-Oriented Class Design. IEEE TSE 36(6), 798–816 (2010)
17. Takagi, H.: Interactive evolutionary computation: Fusion of the capacities of EC optimization and human evaluation. Proceedings of the IEEE 89(9), 1275–1296 (2001)
18. Tonella, P., Susi, A., Palma, F.: Using interactive GA for requirements prioritization. In: SSBSE. pp. 57–66 (2010)
19. Wen, Z., Tzerpos, V.: An effectiveness measure for software clustering algorithms. In: IWPC. pp. 194–203. (2004)
20. Wiggerts, T.A.: Using clustering algorithms in legacy systems re-modularization. In: WCRE. p. 33. IEEE Computer Society (1997)