2014 IEEE International Conference on Software Maintenance and Evolution

Refactoring Fat Interfaces Using a Genetic Algorithm

Daniele Romano
Software Engineering Research Group
Delft University of Technology
The Netherlands
Email: daniele.romano@tudelft.nl

Steven Raemaekers
Software Improvement Group
Amsterdam
The Netherlands
Email: s.raemaekers@sig.eu

Martin Pinzger
Software Engineering Research Group
University of Klagenfurt
Austria
Email: martin.pinzger@aau.at

***Abstract*—Recent studies have shown that the violation of the Interface Segregation Principle (ISP) is critical for maintaining and evolving software systems. Fat interfaces (*i.e.*, interfaces violating the ISP) change more frequently and degrade the quality of the components coupled to them. According to the ISP the interfaces' design should force no client to depend on methods it does not invoke. Fat interfaces should be split into smaller interfaces exposing only the methods invoked by groups of clients. However, applying the ISP is a challenging task when fat interfaces are invoked differently by many clients.**

In this paper, we formulate the problem of applying the ISP as a multi-objective clustering problem and we propose a genetic algorithm to solve it. We evaluate the capability of the proposed genetic algorithm with 42,318 public Java APIs whose clients' usage has been mined from the Maven repository. The results of this study show that the genetic algorithm outperforms other search based approaches (*i.e.*, random and simulated annealing approaches) in splitting the APIs according to the ISP.

***Index Terms*—Interface Segregation Principle; APIs; refactoring; search-based software engineering; genetic algorithms;**

I. Introduction

The main problem a developer can face in designing an interface of a software component is coping with *fat* interfaces [1]. Fat interfaces are interfaces whose clients invoke different subsets of their methods. Such interfaces should be split into smaller interfaces each one specific for a different client (or a group of clients). This principle has been formalized by Martin [1] in 2002 and is also known as the *Interface Segregation Principle* (*ISP*). The rationale behind this principle is that changes to an interface break its clients. As a consequence, clients should not be forced to depend upon interface methods that they do not actually invoke [1]. This guarantees that clients are affected by changes only if they involve the methods they invoke.

Recent studies have shown that violation of the ISP and, hence, fat interfaces can be problematic for the maintenance of software systems. First, in our previous work [2] we showed that such interfaces are more change-prone than *non*-fat interfaces. Next, Abdeen *et al.* [3] proved that violations of the ISP lead to degraded cohesion of the components coupled to fat interfaces. Finally, Yamashita *et al.* [4] showed that changes to fat interfaces result in a larger ripple effect. The results of these studies, together with Martin's insights [1], show the relevance of designing and implementing interfaces according to the ISP.

However, to the best of our knowledge, there are no studies that propose approaches to apply the ISP. This task is challenging when fat interfaces expose many methods and have many clients that invoke differently their methods, as shown in [5]. In this case trying to manually infer the interfaces into which a fat interface should be split is unpractical and expensive.

In this paper, we define the problem of splitting fat interfaces according to the ISP as a multi-objective clustering optimization problem [6]. We measure the compliance with the ISP of an interface through the *Interface Usage Cohesion* metric (*IUC*) as in [2], [3]. To apply the ISP we propose a multi-objective genetic algorithm that, based on the clients' usage of a fat interface, infers the interfaces into which it should be split to conform to the ISP and, hence, with higher IUC values. To validate the capability of the proposed genetic algorithm we mine the clients' usage of 42,318 public Java APIs from the Maven repository. For each API, we run the genetic algorithm to split the API into sub-APIs according to the ISP. We compare the capability of the genetic algorithm with the capability of other search-based approaches, namely a random algorithm and a multi-objective simulated annealing algorithm. The goal of this study is to answer the following research questions:

Is the genetic algorithm able to split APIs into sub-APIs with higher IUC values? Does it outperform the random and simulated annealing approaches?

The results show that the proposed genetic algorithm generates sub-APIs with higher IUC values and it outperforms the other search-based approaches. These results are relevant for software practitioners interested in applying the ISP. They can monitor how clients invoke their APIs (*i.e.*, which methods are invoked by each client) and they can use this information to run the genetic algorithm and split their APIs so that they comply with the ISP.

The remainder of this paper is organized as follows. Section II introduces fat APIs, the main problems they suffer from, and formulates the problem of applying the ISP as a multi-objective clustering problem. Section III presents the genetic algorithm to solve the multi-objective clustering problem. Section IV presents the random and local search (*i.e.*, simulated annealing) approaches implemented to evaluate the capability of the genetic algorithm. The study and its results are shown and discussed in Section V while threats to validity are discussed in Section VI. Related work is presented in Section VII. We draw our conclusions and outline directions for future work in Section VIII.

II. Problem Statement and Solution

In this section, first, we introduce *fat* APIs, their drawbacks, and the Interface Segregation Principle to refactor them. Then, we discuss the challenges of applying the Interface Segregation Principle for real world APIs. Finally, we present our solution to automatically apply the principle.

A. Fat APIs and Interface Segregation Principle

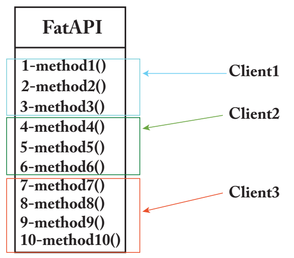
The Interface Segregation Principle (*ISP*) has been originally described by Martin in [1] and it copes with *fat* APIs. Fat APIs are APIs whose clients invoke different sets of their methods. As a consequence clients depend on interface methods that they do not invoke. These APIs are problematic and they should be refactored because their clients can be broken by changes to methods which they do not invoke. To refactor fat APIs Martin [1] introduced the *ISP*.

The ISP states that fat APIs need to be split into smaller APIs (referred to as sub-APIs throughout this paper) according to their clients' usage. Any client should only know about the set of methods that it invokes. Hence, each sub-API should reflect the usage of a specific client (or of a class of clients that invoke the same set of methods). To better understand the ISP consider the example shown in Figure 1. The API shown in Figure 1a is considered a fat API because the different clients (*i.e.*, *Client1*, *Client2*, and *Client3*) invoke different methods (*e.g.*, *Client1* invokes only *method1*, *method2*, and *method3* out of the 10 methods declared in the API). According to the ISP, this API should be split into three sub-APIs as shown in Figure 1b. These sub-APIs are specific to the different clients (*i.e.*, *Client1*, *Client2*, and *Client3*) and, as a consequence, clients do not depend anymore on interface methods they do not invoke.

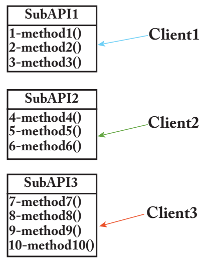
B. Fat APIs and Change-Proneness

Fat APIs are also problematic because they change frequently. In our previous work [2] we showed empirically that fat APIs are more change-prone compared to non-fat APIs. In this work we used *external* cohesion as a heuristic to detect fat APIs. The *external* cohesion was originally defined by Perepletchikov *et al.* [7], [8] for web APIs and it measures the extent to which the methods declared in an API are used by their clients. An API is considered externally cohesive if all clients invoke all methods of the API. It is not externally cohesive and considered a fat API if they invoke different subsets of its methods.

To measure the external cohesion we used the Interface Usage Cohesion metric (*IUC*) defined by Perepletchikov *et al.* [7], [8]. This metric is defined as:



(a) A fat API with different clients (*i.e.*, *Client1*, *Client2*, and *Client3*) invoking different sets of methods (denoted by rectangles). Clients depend on methods which they do not invoke.



(b) The fat API is split into sub-APIs each one specific for a client. Clients depend only on methods which they invoke.

Fig. 1: An example of applying the Interface Segregation Principle.

$$IUC(i) = \frac{\sum_{j=1}^{n} \frac{used\_methods(j,i)}{num\_methods(i)}}{n}$$

where *j* denotes a client of the API *i*; *used_methods (j,i)* is the function which computes the number of methods defined in *i* and used by the client *j*; *num _methods(i)* returns the total number of methods defined in *i*; and *n* denotes the number of clients of the API *i*. Note that the IUC values range between 0 and 1.

Consider the example shown in Figure 1. The *FatAPI* in Figure 1a shows a value of $IUC_{FatAPI} = (\frac{3}{10} + \frac{3}{10} + \frac{4}{10})/3 = 0.366$ indicating low external cohesion that is a symptom of a fat API. The sub-APIs generated after applying the ISP (shown in Figure 1b) show higher external cohesion. They have the following values for IUC: $IUC_{SubAPI1} = (\frac{3}{3})/1 = 1$, $IUC_{SubAPI2} = (\frac{3}{3})/1 = 1$, and $IUC_{SubAPI3} = (\frac{4}{4})/1 = 1$.

In [2] we investigated to which extent the IUC metric can be used to highlight change-prone Java interface classes. The results showed that the IUC metric exhibits the strongest correlation with the number of source code changes performed in Java interface classes compared to other software metrics (*e.g.*, C&K metrics [9]). The IUC metric also improved the performance of prediction models in predicting change-prone Java interface classes.

These results, together with Martin's insights [1] and results of previous studies [3], [4], motivated us to investigate and develop an approach to refactor fat APIs using the ISP.

C. Problem

The problem an engineer can face in splitting a fat API is coping with API usage diversity. In 2013, Mendez *et al.* [5] investigated how differently the APIs are invoked by their clients. They provided empirical evidence that there is a significant usage diversity. For instance, they showed that Java's `String` API is used in 2,460 different ways by their clients. Clients do not invoke disjoint sets of methods (as shown in Figure 1a) but the set of methods can overlap and can be significantly different. As a consequence, we argue that manually splitting fat APIs can be time consuming and error prone.

A first approach to find the sub-APIs consists in adopting brute-force search techniques. These techniques enumerate all possible sub-APIs and check whether they maximize the external cohesion and, hence, the value for the IUC metric. The problem with these approaches is that the number of possible sub-APIs can be prohibitively large causing a combinatorial explosion. Imagine for instance to adopt this approach for finding the sub-APIs for the *AmazonEC2* web API. This web API exposes 118 methods in version 23 [10]. The number of 20-combinations of the 118 methods in *AmazonEC2* are equal to:

$$\binom{118}{20} = \frac{118!}{20!98!} \approx 2 * 10^{21}$$

This means that for evaluating all the sub-APIs with 20 methods the search requires to analyze at least $2{*}10^{21}$ possible combinations, which can take several days on a standard PC.

As a consequence, brute-force search techniques are not an adequate solution for this problem.

D. Solution

To overcome the aforementioned problems we formulate the problem of finding sub-APIs (*i.e.*, applying the ISP) as a clustering optimization problem defined as follows. Given the set of n methods $X=\{X_1,X_2...,X_n\}$ declared in a fat API, find the set of non-overlapping clusters of methods $C=\{C_1,C_2...,C_k\}$ that maximize *IUC(C)* and minimize *clusters(C)*; where *IUC(C)* computes the lowest IUC value of the clusters in *C* and *clusters(C)* computes the number of clusters. In other words, we want to cluster the methods declared in a fat API into sub-APIs that show high external cohesion, measured through the IUC metric.

This problem is an optimization problem with two objective functions, also known as multi-objective optimization problem. The first objective consists in maximizing the external cohesion of the clusters in *C*. Each cluster in *C* (*i.e.*, a sub-API in our case) will have its own IUC value (like for the sub-APIs in Figure 1b). To maximize their IUC values we maximize the lowest IUC value measured through the objective function *IUC(C)*.

The second objective consists in minimizing the number of clusters (*i.e.*, sub-APIs). This objective is necessary to avoid solutions containing as many clusters as there are methods declared in the fat API. If we assign each method to a different sub-API, all the sub-APIs would have an IUC value of 1, showing the highest external cohesion. However, such sub-APIs do not group together the methods invoked by the different groups of clients. Hence, the clients would depend on many sub-APIs each one exposing a single method.

To solve this multi-objective clustering optimization problem we implemented a multi-objective genetic algorithm (presented in next section) that searches for the Pareto optimal solutions, namely solutions whose objective function values (*i.e.*, *IUC(C)* and *clusters(C)* in our case) cannot be improved without degrading the other objective function values.

Moreover, to compare the performance of the genetic algorithm with random and local search approaches we implemented a random approach and a multi-objective simulated annealing approach that are presented in Section IV.

III. Genetic Algorithm

To solve multi-objective optimization problems different algorithms have been proposed in literature (*e.g.*, [11], [12], [13]). In this paper, we use the multi-objective genetic algorithm NSGA-II proposed by Deb *et al.* [11] to solve the problem of finding sub-APIs for *fat* APIs according to the ISP, as described in the previous section. We chose this algorithm because 1) it has been proved to be fast, 2) to provide better convergence for most multi-objective optimization problems, and 3) it has been widely used in solving search based software engineering problems, such as presented in [11], [14], [15], [16]. In the following, we first introduce the genetic algorithms. Then, we show our implementation of the NSGA-II used to solve our problem. For further details about the NSGA-II we refer to the work by Deb *et al.* [11].

Genetic Algorithms (GAs) have been used in a wide range of applications where optimization is required. Among all the applications, GAs have been widely studied to solve clustering problems [17]. The key idea of GAs is to mimic the process of natural selection providing a search heuristic to find solutions to optimization problems. A generic GA is shown in Figure 2.

Different to other heuristics (*e.g.*, *Random Search*, *Brute-Force Search*, and *Local search*) that consider one solution at a time, a GA starts with a set of candidate solutions, also known as population (step 1 in Figure 2). These solutions are randomly generated and they are referred to as chromosomes. Since the search is based upon many starting points, the likelihood to explore a wider area of the search space is higher than other searches. This feature reduces the likelihood to get stuck in a local optimum. Each solution is evaluated through a fitness function (or objective function) that measures how good a candidate solution is relatively to other candidate solutions (step 2). Solutions from the population are used to form new populations, also known as generations. This is achieved using the evolutionary operators. Specifically, first a pair of solutions (parents) is selected from the population through a selection operator (step 4). From these parents two offspring solutions are generated through the crossover operator (step 5). The crossover operator is responsible to generate offspring solutions that combine features from the two parents. To preserve the diversity, the mutation operators (step 6) mutate the offspring. These mutated solutions are added to the population replacing solutions with the worst fitness function values. This process of evolving the population is repeated until some condition (*e.g.*, reaching the max number of iterations in step 3 or achieving the goal). Finally, the GA outputs the best solutions when the evolution process terminates (step 7).

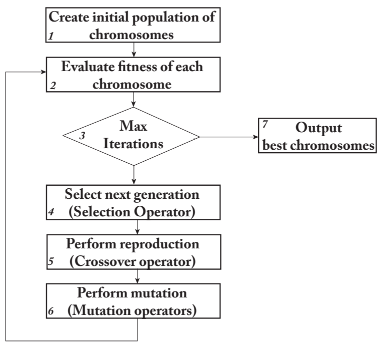

Create initial population of chromosomes
1
Evaluate fitness of each chromosome
2
3
Max Iterations
7
Output best chromosomes
Select next generation (Selection Operator)
4
Perform reproduction (Crossover operator)
5
Perform mutation (Mutation operators)
6

Fig. 2: Different steps of a genetic algorithm.

To implement the GA and adapt it to find the set of sub-APIs into which a *fat* API should be split we next define the fitness function, the chromosome (or solution) representation, and the evolutionary operators (*i.e.*, selection, crossover, and mutation).

A. Chromosome representation

To represent the chromosomes we use a *label-based integer encoding* widely adopted in literature [17] and shown in Figure 3. According to this encoding, a solution is an integer array of *n* positions, where *n* is the number of methods exposed in a *fat* API. Each position corresponds to a specific method (*e.g.*, position 1 corresponds to the method *method1()* in Figure 1a). The integer values in the array represent the clusters (*i.e.*, sub-APIs in our case) to which the methods belong. For instance in Figure 3, the methods 1,2, and 10 belong to the same cluster labeled with 1. Note that two chromosomes can be equivalent even though the clusters are labeled differently. For instance the chromosomes [1,1,1,1,2,2,2,2,3,3] and [2,2,2,2,3,3,3,3,1,1] are equivalent. To solve this problem we apply the *renumbering procedure* as shown in [18] that transforms different labelings of equivalent chromosomes into a unique labeling.

1	2	3	4	5	6	7	8	9	10
1	1	2	3	2	4	5	3	6	1

Fig. 3: Chromosome representation of our candidate solutions.

B. Fitness Functions

The fitness function is a function that measures how good a solution is. For our problem we have two fitness functions corresponding to the two objective functions discussed in Section II, namely *IUC(C)*) and *clusters(C)*. *IUC(C)* returns the lowest IUC value of the clusters in *C* and *clusters(C)* returns the number of clusters in C. Hence, the two fitness functions are f_1=*IUC(C)* and f_2=*clusters(C)*. While the value of f_1 should be maximized the value of f_2 should be minimized.

Since we have two fitness functions, we need a comparator operator that, given two chromosomes (*i.e.*, candidate solutions), returns the best one based on their fitness values. As comparator operator we use the dominance comparator as defined in NSGA-II. This comparator utilizes the idea of Pareto optimality and the concept of dominance for the comparison. Precisely, given two chromosomes A and B, the chromosome A dominates chromosome B (*i.e.*, A is better than B) if 1) every fitness function value for chromosome A is equal or better than the corresponding fitness function value of the chromosome B, and 2) chromosome A has at least one fitness function value that is better than the corresponding fitness function value of the chromosome B.

C. The Selection Operator

The selection operator selects two parents from a population according to their fitness function values. We use the Ranked Based Roulette Wheel (*RBRW*) that is a modified roulette wheel selection operator as proposed by Al Jadaan *et al.* [19]. *RBRW* ranks the chromosomes in the population by the fitness values: the highest rank is assigned to the chromosome with the best fitness values. Hence, the best chromosomes have the highest probabilities to be selected as parents.

D. The Crossover Operator

Once the GA has selected two parents (*ParentA* and *ParentB*) to generate the offspring, the crossover operator is applied to them with a probability P_c. As crossover operator we use the operator defined specifically for clustering problems by Hruschka *et al.* [17]. In order to illustrate how this operator works consider the example shown in Figure 4 from [17]. The operator first selects randomly *k* ($1 \leq k \leq n$) clusters from *ParentA*, where *n* is the number of clusters in *ParentA*. In our example assume that the clusters labeled 2 (consisting of methods 3, 5, and 9) and 3 (consisting of method 4) are selected from *ParentA* (marked bold in Figure 4). The first child (*ChildC*) originally is created as copy of the second parent *ParentB* (step 1). As second step, the selected clusters (*i.e.*, 2 and 3) are copied into *ChildC*. Copying these clusters changes the clusters 1, 2, and 3 in *ChildC*. These changed clusters are removed from *ChildC* (step 3) leaving the corresponding methods unallocated (labeled with 0). In the fourth step (not shown in Figure 4) the unallocated methods are allocated to an existing cluster that is randomly selected.

The same procedure is followed to generate the second child *ChildD*. However, instead of selecting randomly *k* clusters from *ParentB*, the changed clusters of *ChildC* (*i.e.*, 1,2, and 3) are copied into *ChildD* that is originally a copy of *ParentA*.

E. The Mutation Operators

After obtaining the offspring population through the crossover operator, the offspring is mutated through the mutation operator with a probability P_m. This step is necessary to ensure genetic diversity from one generation to the next ones. The mutation is performed by randomly selecting one of the following cluster-oriented mutation operators [18], [17]:

- **split**: a randomly selected cluster is split into two different clusters. The methods of the original cluster are randomly assigned to the generated clusters.
- **merge**: moves all methods of a randomly selected cluster to another randomly selected cluster.

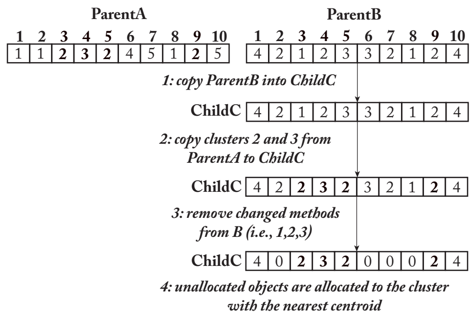

ParentA
ParentB
1 2 3 4 5 6 7 8 9 10
1 1 2 3 2 4 5 1 2 5
1 2 3 4 5 6 7 8 9 10
4 2 1 2 3 3 2 1 2 4
1: copy ParentB into ChildC
ChildC 4 2 1 2 3 3 2 1 2 4
2: copy clusters 2 and 3 from ParentA to ChildC
ChildC 4 2 2 3 2 3 2 1 2 4
3: remove changed methods from B (i.e., 1,2,3)
ChildC 4 0 2 3 2 0 0 0 2 4
4: unallocated objects are allocated to the cluster with the nearest centroid

Fig. 4: Example of crossover operator for clustering problems [17].

- **move**: moves methods between clusters. Both methods and clusters are randomly selected.

We implemented the proposed genetic algorithm on top of the *JMetal*[1] framework that is a Java framework that provides state-of-the-art algorithms for optimization problems, including the NSGA-II algorithm.

IV. Random and Local Search

To better evaluate the performance of our proposed genetic algorithm we implemented a random algorithm and a local search algorithm (*i.e.*, a multi-objective simulated annealing algorithm) that are presented in the following sub-sections.

A. Random Algorithm

The random algorithm tries to find an optimal solution by generating random solutions. To implement the random algorithm we use the same solution representation (*i.e.*, chromosome representation) used in the genetic algorithm described in Section III. The algorithm iteratively generates a random solution and evaluates it using the same fitness functions defined for the genetic algorithm. When the maximum number of iterations is reached the best solution is output. This algorithm explores the search space randomly relying on the likelihood to find a good solution after a certain number of iterations. We use a random search as baseline because this comparison is considered the first step to evaluate a genetic algorithm [20].

B. Multi-Objective Simulated Annealing

As second step to evaluate the performance of our proposed genetic algorithm we implemented a local search approach.

A local search algorithm (*e.g.*, *hill-climbing*) starts from a candidate solution and then iteratively tries to improve it. Starting from a random generated solution the solution is mutated obtaining the *neighbor* solution. If the *neighbor* solution is better than the current solution (*i.e.*, it has higher fitness function values) it is taken as current solution to generate a new *neighbor* solution. This process is repeated until the best solution is obtained or the maximum number of iterations is reached. The main problem of such local search approaches is that they can get stuck in a local optimum. In this case the local search approach cannot further improve the current solution.

[1]http://jmetal.sourceforge.net

To mitigate this problem advanced local search approaches have been proposed like *simulated annealing*. The simulated annealing algorithm was inspired from the process of annealing in metallurgy. This process consists in heating and cooling a metal. Heating the metal alters its internal structure and, hence, its physical properties. On the other hand, when the metal cools down its new internal structure becomes fixed.

The simulated annealing algorithm simulates this process. Initially the temperature is set high and then it is decreased slowly as the algorithm runs. While the temperature is high the algorithm is more likely to accept a *neighbor* solution that is worse than the current solution, reducing the likelihood to get stuck in a local optimum. At each iteration the temperature is slowly decreased by multiplying it by a *cooling factor* α where $0 < \alpha < 1$. When the temperature is reduced, worse *neighbor* solutions are accepted with a lower probability.

Hence, at each iteration a *neighbor* solution is generated mutating the current solution. If this solution has better fitness function values it is taken as current solution. Otherwise it is accepted with a certain probability called *acceptance probability*. This acceptance probability is computed by a function based on 1) the difference between the fitness function values of the current and neighbor solution and 2) the current temperature value.

To adapt this algorithm for solving our multi-objective optimization problem we implemented a Multi-Objective Simulated Annealing algorithm following the approach used by Shelburg *et al.* in [21]. To represent the solutions we use the same solution representation used in the genetic algorithm (*i.e.*, *label-based integer encoding*). We generate the neighbor solutions using the mutation operators used in our genetic algorithm. We compare two solutions using the same fitness functions and dominance comparator of our genetic algorithm. The *acceptance probability* is computed as in [21] with the following function:

$$AcceptProb(i, j, temp) = e^{\frac{-abs(c(i,j))}{temp}}$$

where *i* and *j* are the current and neighbor solutions; *temp* is the current temperature; and *c(i,j)* is a function that computes the difference between the fitness function values of the two solutions *i* and *j*. This difference is computed as the average of the differences of each fitness function values of the two solutions according to the following equation:

$$c(i, j) = \frac{\sum_{k=1}^{|D|}(c_k(j) - c_k(i))}{|D|}$$

where *D* is the set of fitness functions and $c_k(j)$ is the value of the fitness function *k* of the solution *j*. In our case the fitness functions are the *IUC(C)* and *clusters(c)* functions used in the genetic algorithm. Note that since this difference is computed as average it is relevant that the fitness function values are measured on the same scale. For this reason the values of the fitness function *clusters(C)* are normalized to the range between 0 and 1. For further details about the multi-objective simulated annealing we refer to the work in [22], [21].

V. Study

The *goal* of this empirical study is to evaluate the effectiveness of our proposed genetic algorithm in applying the ISP to Java APIs. The *quality focus* is the ability of the genetic algorithm to split APIs in sub-APIs with higher external cohesion that is measured through the *IUC* metric. The *perspective* is that of API providers interested in applying the ISP and in deploying APIs with high external cohesion. The *context* of this study consists of 42,318 public Java APIs mined from the Maven repository.

In this study we answer the following research questions:

Is the genetic algorithm able to split APIs into sub-APIs with higher IUC values? Does it outperform the random and simulated annealing approaches?

In the following, first, we show the process we used to extract the APIs and their clients' usage from the Maven repository. Then, we show the procedure we followed to calibrate the genetic algorithm and the simulated annealing algorithms. Finally, we present and discuss the results of our study.

A. Data Extraction

The public APIs under analysis and their clients' usage have been retrieved from the Maven repository.[2] The Maven repository is a publicly available data set containing 144,934 binary jar files of 22,205 different open-source Java libraries, which is described in more detail in [23]. Each binary jar file has been scanned to mine method calls using the ASM[3] Java bytecode manipulation and analysis framework. The dataset was processed using the DAS-3 Supercomputer[4] consisting of 100 computing nodes.

To extract method calls we scanned all `.class` files of all jar files. Class files contain fully qualified references to the methods they call, meaning that the complete package name, class name and method name of the called method is available in each `.class` file. For each binary file, we use an ASM bytecode visitor to extract the package, class and method name of the callee.

Once we extracted all calls from all `.class` files, we grouped together calls to the same API. As clients of an API we considered all classes declared in other jar files from the Maven repository that invoke public methods of that API. Note that different versions of the same class are considered different for both clients and APIs. Hence, if there are two classes with the same name belonging to two different versions of a jar file they are considered different. To infer which version of the jar file a method call belongs to we scanned the Maven build file (`pom.xml`) for dependency declarations.

[2]http://search.maven.org

[3]http://asm.ow2.org

[4]http://www.cs.vu.nl/das3/

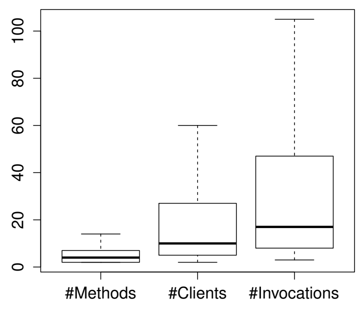


Fig. 5: Box plots of number of methods (#Methods), clients (#Clients), and invocations (#Invocations) for the public APIs under analysis. Outliers have been removed for the sake of simplicity.

In total we extracted the clients' usage for 110,195 public APIs stored in the Maven repository. We filtered out APIs not relevant for our analysis by applying the following filters:

- APIs should declare at least two methods.
- APIs should have more than one client.
- IUC value of the APIs should be less than one.

After filtering out non relevant APIs we ended up with a data set of 42,318 public APIs whose number of clients, methods, and invocations are shown by the box plots in Figure 5 where outliers have been removed for the sake of simplicity. The median number of methods exposed in the APIs under analysis is 4 while the biggest API exposes 370 methods. The median number of clients is 10 with a maximum number of 102,445 of the API *org.apache.commons.lang.builder.EqualsBuilder* (outlier not shown in Figure 5). The median number of invocations to the APIs is 17 with a maximum number of 270,569 for the API *org.apache.commons.lang.builder.EqualsBuilder* (outlier not shown in Figure 5).

B. GA and SA Calibration

To calibrate the GA and SA algorithms we followed a trial-and-error procedure with 10 toy examples. Each toy example consists of an API with 10 methods and 4 clients. For each of the 10 toy examples we changed the clients' usage. Then, we evaluated the IUC values output by the algorithms with different parameters. For each different parameter, we ran the algorithms ten times. We used the Mann-Whitney and Cliff's Delta tests to evaluate the difference between the IUC values output by each run. For the GA we evaluated the output with the following parameters:

- population size was incremented stepwise by 10 from 10 to 200 individuals.
- numbers of iterations was incremented stepwise by 1,000 from 1,000 to 10,000.
- crossover and mutation probability were increased stepwise by 0.1 from 0.0 to 1.0.

We noticed statistically different results only when the population size was less than 50, the number of iterations was less than 1,000, and the crossover and mutation probability was less than 0.7. Hence, we decided to use the default values specified in JMetal (*i.e.*, population of 100 individuals, 10,000 iterations, crossover and mutation probability of 0.9).

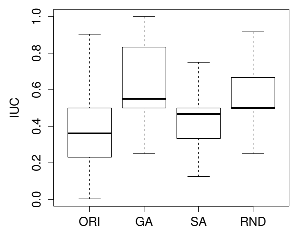
Similarly, the output of the SA algorithm was evaluated with different values for the cooling factor. The cooling factor was incremented stepwise by 0.1 from 0.1 to 1.0. We did not register any statistically significant difference and we chose a starting temperature of 0.0003 and a cooling factor of 0.99965 as proposed in [21]. The number of iterations for the SA and RND algorithms is 10,000 to have a fair comparison with the GA.

C. Results

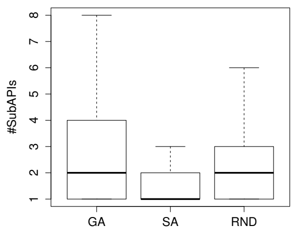
To answer our research questions, first, we compute the *IUC* value for each public API using the extracted invocations. We refer to this value as IUC_{before}. Then, we run the genetic algorithm (*GA*), the simulated annealing algorithm (*SA*), and random algorithm (*RND*) with the same number of iterations (*i.e.*, 10,000). For each API under analysis, these algorithms output the set of sub-APIs into which the API should be split. Each sub-API will show a different *IUC* value. Among these sub-APIs we take the sub-API with the *lowest* IUC value to which we refer as IUC_{after}. We chose the lowest IUC value because this gives us the lower boundary for the IUC values of the resulting sub-APIs.

Figure 6 shows the distributions of IUC_{after} values and number of sub-APIs output by the different algorithms. The box plots in Figure 6a show that all the search-based algorithms produced sub-APIs with higher IUC_{after} values compared to the original APIs (ORI). The genetic algorithm (GA) produced sub-APIs that have higher IUC_{after} values than the original APIs (ORI) and the sub-APIs generated by the simulated annealing algorithm (SA) and by the random algorithm (RND). The second best algorithm is the random algorithm that outperforms the simulated annealing.

The higher IUC_{after} values of the genetic algorithm are associated with a higher number of sub-APIs as shown in Figure 6b. These box plots show that the median number of sub-APIs are 2 for the genetic algorithm and the random algorithm. The simulated annealing generated a median number of 1 API, meaning that in 50% of the cases it kept the original API without being able to split it. We believe that the poor performance of the simulated annealing is due to its nature. Even though it is an advanced local search approach it is still a local search approach that can get stuck in a local optimum. To give a better view of the IUC values of the sub-APIs we show the distributions of IUC values measured on the sub-APIs generated by the genetic algorithm in Figure 7. *Min* represents the distribution of IUC values of sub-APIs with the lowest IUC (*i.e.*, IUC_{after}). *Max* represents the distribution of IUC values of sub-APIs with the highest IUC. *Q1*, *Q2*, and *Q3* represent respectively the first, second, and third quartiles of the ordered set of IUC values of the sub-APIs.



(a) Box plots of IUC values measured on the original APIs (*ORI*) and IUC_{after} measured on the sub-APIs output by the genetic algorithm (*GA*), by the simulated annealing algorithm (*SA*), and by the random algorithm (*RND*).



(b) Number of sub-APIs generated by the genetic algorithm (*GA*), the simulated annealing algorithm (*SA*), and the random algorithm (*RND*).

Fig. 6: IUC values and number of sub-APIs generated by the different search-based algorithms.

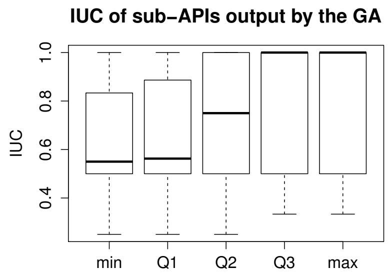


Fig. 7: Box plots of IUC values measured on the sub-APIs output by the genetic algorithm. Outliers have been removed for the sake of simplicity.

The box plots in Figure 6 already give insights into the capability of the different search-based algorithms of applying the ISP. To provide statistical evidence of their capability we compute the difference between the distributions of IUC_{before} and IUC_{after} generated by the different algorithms using the paired Mann-Whitney test [24] and the paired Cliff's Delta *d* effect size [25]. First, we use the Mann-Whitney test to analyze whether there is a significant difference between the distributions of IUC_{before} and IUC_{after}. Significant differences are indicated by Mann-Whitney *p-values* ≤ 0.01. Then, we use the Cliff's Delta effect size to measure the magnitude of the difference. Cliff's Delta estimates the probability that a value selected from one group is greater than a value selected from the other group. Cliff's Delta ranges between +1, if all selected values from one group are higher than the selected values in the other group, and -1, if the reverse is true. 0 expresses two overlapping distributions. The effect size is considered negligible for $d < 0.147$, small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.47$, and large for $d \geq 0.47$ [25]. We chose the Mann-Whitney test and Cliff's Delta effect size because the distributions of IUC values are not normally distributed as shown by the results of the Shapiro test. The Mann-Whitney test and Cliff's Delta effect size are suitable for non-normal distribution because they do not require assumptions about the variances and the types of the distributions (*i.e.*, they are non-parametric tests). The results of the Mann-Whitney test and Cliff's Delta effect size are shown in Table I.

The distribution of IUC_{after} values measured on the sub-APIs generated by the genetic algorithm is statistically different (M-W *p-value*<2.20E-16) from the original IUC values (GA vs ORI). The Cliff's Delta is 0.732 if we consider all the APIs (ALL) and 1 if we consider only APIs with more than 2 methods (#Methods>2). In both cases the Cliff's *delta* is greater than 0.47 and, hence, the effect size is considered statistically large. We obtained similar results comparing the distributions of IUC_{after} values of the sub-APIs generated by the genetic algorithm and the simulated annealing algorithm (GA vs SA). The Mann-Whitney *p-value* is <2.20E-16 and the Cliff's *delta* is large (*i.e.*, 0.705 for ALL and 0.962 for #Methods>2). The distributions of IUC_{after} values of the genetic algorithms and random algorithm (GA vs RND) are also statistically different (M-W *p-value*<2.20E-16). Its effect size is medium (*i.e.*, 0.339 for ALL and 0.463 for #Methods>2).

Moreover, from the results shown in Table I we notice that the Cliff's delta effect size is always greater when we consider only APIs with more than two methods. This result shows that the effectiveness of the genetic algorithm, random algorithm, and simulated annealing algorithm might depend on the number of methods declared in the APIs, number of clients, and number of invocations. To investigate whether these variables have any impact on the effectiveness of the algorithms, we analyze the Cliff's Delta for APIs with increasing numbers of methods, clients, and invocations. First, we partition the data set grouping together APIs with the same number of methods. Then, we compute the Cliff's Delta between the distributions of IUC_{before} and IUC_{after} for each different group. Finally, we use the paired Spearman correlation test to investigate the correlation between the Cliff's Delta measured on the different groups and their number of methods. We use the same method to analyze the correlation between the Cliff's Delta and the number of clients and invocations. The Spearman test compares the ordered ranks of the variables to measure a monotonic relationship. We chose the Spearman correlation because the distributions under analysis are non-normal (normality has been tested with the Shapiro test). The Spearman test is a non-parametric test and, hence, it does not make assumptions about the distribution, variances and the type of the relationship [26]. A Spearman *rho* value of +1 and -1 indicates high positive or high negative correlation, whereas 0 indicates that the variables under analysis do not correlate at all. Values greater than +0.3 and lower than -0.3 indicate a moderate correlation; values greater than +0.5 and lower than -0.5 are considered to be strong correlations [27].

The results of the Spearman correlation tests are shown in Table II. We notice that the Cliff's Delta between the distributions of IUC_{after} values of the genetic algorithm and the random algorithm (*i.e.*, GA vs RND) increases with larger APIs. The Cliff's Delta effect size are strongly correlated (*i.e.*, *rho*=0.617) with the number of methods (#Methods). This indicates that the more methods an API exposes the more the genetic algorithm outperforms the random algorithm generating APIs with higher IUC. Moreover, with increasing number of clients (*i.e.*, #Clients) and invocations (*i.e.*, #Invocations) the Cliff's Delta between the distributions of IUC_{after} values of the genetic algorithm and the other search algorithms increases as well. This is indicated by *rho* values that are greater than 0.3.

Based on these results we can answer our research questions stating that 1) the genetic algorithm is able to split APIs into sub-APIs with higher IUC values and 2) it outperforms the other search-based algorithms. The difference in performance between the genetic algorithm and random algorithm increases with an increasing number of methods declared in the APIs. The difference in performance between the genetic algorithm and the other search-based techniques increases with an increasing number of clients and invocations.

D. Discussions of the Results

The results of our study are relevant for API providers. Publishing stable APIs is one of their main concerns, especially if they publish APIs on the web. APIs are considered contracts between providers and clients and they should stay as stable as possible to not break clients' systems. In our previous study [2] we showed empirically that *fat* APIs (*i.e.*, APIs with low external cohesion) are more change-prone than non-*fat* APIs. To refactor such APIs Martin [1] proposed the Interface Segregation Principle (ISP). However, applying this principle is not trivial because of the large API usage diversity [5].

Our proposed genetic algorithm assists API providers in applying the ISP. To use our genetic algorithm providers should monitor how their clients invoke their API. For each client they should record the methods invoked in order to compute the IUC metric. This data is used by the genetic algorithm to evaluate the candidate solutions through fitness functions as described in Section III. The genetic algorithm is then capable to suggest the sub-APIs into which an API should be split in order to apply the ISP.

This approach is particularly useful to deploy stable web APIs. One of the key factors for deploying successful web APIs is assuring an adequate level of stability. Changes in a web API might break the consumers' systems forcing them to continuously adapt them to new versions of the web API. Using our approach providers can deploy web APIs that are more externally cohesive and, hence, less change-prone [2]. Moreover, since our approach is automated, it can be integrated into development and continuous integration environments to continuously monitor the conformance of APIs to the ISP. Providers regularly get informed when and how to refactor an API. However, note that the ISP takes into account only the clients' usage and, hence, the external cohesion. As a consequence, while our approach assures that APIs are external cohesive, it currently does not guarantee other quality attributes (*e.g.*, internal cohesion). As part of our

	GA vs ORI			GA vs SA			GA vs RND		
APIs	M-W *p-value*	Cliff's *delta*	Magnitude	M-W *p-value*	Cliff's *delta*	Magnitude	M-W *p-value*	Cliff's *delta*	Magnitude
ALL	<2.20E-16	0.732	large	<2.20E-16	0.705	large	<2.20E-16	0.339	medium
#Methods>2	<2.20E-16	1	large	<2.20E-16	0.962	large	<2.20E-16	0.463	medium

TABLE I: Mann-Whitney p-value (*M-W p-value*) and Cliff's delta between the distributions of IUC_{after} values measured on the sub-APIs generated by the genetic algorithm and measured on the original APIs (*i.e.*, GA vs ORI) and on the sub-APIs generated by the simulated annealing (*i.e.*, GA vs SA) and random algorithm (*i.e.*, GA vs RND). The table reports the results for all the APIs under analysis (*i.e.*, ALL) and for APIs with more than 2 methods (*i.e.*, #Methods>2).

	#Methods			#Clients			#Invocations		
	p-value	*rho*	*corr*	*p-value*	*rho*	*corr*	*p-value*	*rho*	*corr*
GA vs ORI	0.6243	0.070	none	**5.199E-13**	**0.446**	**moderate**	**<2.20E-16**	**0.541**	**strong**
GA vs SA	0.8458	-0.028	none	**8.872E-12**	**0.429**	**moderate**	**<2.20E-16**	**0.520**	**strong**
GA vs RND	**8.127E-06**	**0.617**	**strong**	**2.057e-08**	**0.424**	**moderate**	**9.447E-14**	**0.477**	**moderate**

TABLE II: *P-values* and *rho* values of the Spearman correlation test to investigate the correlation between the Cliff's Delta and number of methods, clients, and invocations. Values in bold indicate significant correlations. *Corr* indicates the magnitude of the correlations.

future work we plan to extend our approach in order to take into account other relevant quality attributes.

VI. Threats to Validity

This section discusses the threats to validity that can affect the empirical study presented in the previous section.

Threats to *construct validity* concern the relationship between theory and observation. In our study this threat can be due to the fact that we mined the APIs usage through a binary analysis. In our analysis we have used binary jar files to extract method calls. The method calls that are extracted from compiled `.class` files are, however, not necessarily identical to the method calls that can be found in the source code. This is due to compiler optimizations. For instance, when the compiler detects that a certain call is never executed, it can be excluded. However, we believe that the high number of analyzed APIs mitigates this threat.

With respect to *internal validity*, the main threat is the possibility that the tuning of the genetic algorithm and the simulated annealing algorithm can affect the results. We mitigated this threat by calibrating the algorithms with 10 toys examples and evaluating statistically their performance while changing their parameters.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. Wherever possible, we used proper statistical tests to support our conclusions. In particular we used non-parametric tests which do not make any assumption on the underlying data distribution that was tested against normality using the Shapiro test. Note that, although we performed multiple Mann-Whitney and Spearman tests, *p-value* adjustment (*e.g.*, Bonferroni) is not needed as we performed the tests on independent and disjoint data sets.

Threats to *external validity* concern the generalization of our findings. We mitigated this threat evaluating the proposed genetic algorithm on 42,318 public APIs coming from different Java systems. The invocations to the APIs have been mined from the Maven repository. These invocations are not a complete set of invocations to the APIs because they do not include invocations from software systems not stored in Maven. However, we are confident that the data set used in this paper is a representative sample set.

VII. Related Work

Interface Segregation Principle. After the introduction of the ISP by Martin [1] in 2002 several studies have investigated the impact of fat interfaces on the quality of software systems.

In 2013, Abdeen *et al.* [3] investigated empirically the impact of interfaces' quality on the quality of implementing classes. Their results show that violations of the ISP lead to degraded cohesion of the classes that implement fat interfaces.

In 2013, Yamashita *et al.* [4] investigated the impact of inter-smell relations on software maintainability. They analyzed the interactions of 12 code smells and their relationships with maintenance problems. Among other results, they show that classes violating the ISP manifest higher afferent coupling. As a consequence changes to these classes result in a larger ripple effect.

In our previous work [2], we showed that violations of the ISP can be used to predict change-prone interfaces. Among different source code metrics (*e.g.*, C&K metrics [9]) we demonstrated that fat interfaces (*i.e.*, interfaces showing a low external cohesion measured through the IUC metric) are more change-prone than *non*-fat interfaces. Moreover, our results proved that the IUC metric can improve the performance of prediction models in predicting change-prone interfaces.

The results of this related work show the relevance of applying the ISP and motivated us in defining the approach presented in this paper.

Search Based Software Engineering. Over the last years genetic algorithms, and in general search based algorithms, have become popular to perform refactorings of software systems. The approach closest to ours has been presented by Praditwong *et al.* [6] in 2011. The authors formulated the problem of designing software modules that adhere to quality attributes (*e.g.*, coupling and cohesion) as multi-objective clustering search problem. Similarly to our work, they defined a multi-objective genetic algorithm that clusters software components into modules. Moreover, they show that multi-objective approaches produce better solutions than existing single-objective approaches. This work influenced us in defining the problem as multi-objective problem instead of a single-objective problem. However, the problem we solve is different from theirs. Our approach splits fat API accordingly to the ISP and uses different fitness functions.

Prior to this work [6], many other studies proposed approaches to cluster software components into modules (*e.g.*, [28], [29], [30], [31], [32], [33]). These studies propose single-objective approaches that have been proven to produce worse solutions by Praditwong *et al.* [6].

To the best of our knowledge there are no studies that propose approaches to split fat APIs accordingly to the ISP as proposed in this paper.

VIII. Conclusions and Future Work

In this paper we proposed a genetic algorithm that automatically obtains the sub-APIs into which a fat API should be split according to the ISP. Mining the clients' usage of 42,318 Java APIs from the Maven repository we showed that the genetic algorithm is able to split APIs into sub-APIs. Comparing the resulting sub-APIs, based on the IUC values, we showed that the genetic algorithms outperforms the random and simulated annealing algorithms. The difference in performance between the genetic algorithm and the other search-based techniques increases with APIs with an increasing number of methods, clients, and invocations. Based on these results API providers can automatically obtain and refactor the set of sub-APIs based on how clients invoke the fat APIs.

While this approach is already actionable and useful for API providers, we plan to further improve it in our future work. First, we plan to evaluate qualitatively the sub-APIs generated by the genetic algorithm. The higher IUC values guarantee that sub-APIs are more external cohesive and, hence, they better conform to the ISP. However, we have not investigated yet what developers think about the sub-APIs. Hence, we plan to contact developers and perform interviews to investigate the quality of these sub-APIs. Next, we plan to extend our approach taking into account other quality attributes, such as internal cohesion. Finally, we plan to slightly modify the genetic algorithm to generate overlapping sub-APIs (*i.e.*, sub-APIs that share common methods).

References

[1] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*. Prentice-Hall, Inc, 2002.

[2] D. Romano and M. Pinzger, "Using source code metrics to predict change-prone java interfaces," in *ICSM*, 2011, pp. 303–312.

[3] H. Abdeen, H. A. Sahraoui, and O. Shata, "How we design interfaces, and how to assess it," in *ICSM*, 2013, pp. 80–89.

[4] A. F. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: an empirical study," in *ICSE*, 2013, pp. 682–691.

[5] D. Mendez, B. Baudry, and M. Monperrus, "Empirical evidence of large-scale diversity in api usage of object-oriented software," in *SCAM*, 2013, pp. 43–52.

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

[7] M. Perepletchikov, C. Ryan, and K. Frampton, "Cohesion metrics for predicting maintainability of service-oriented software," in *QSIC*, 2007, pp. 328–335.

[8] M. Perepletchikov, C. Ryan, and Z. Tari, "The impact of service cohesion on the analyzability of service-oriented software," *IEEE T. Services Computing*, vol. 3, no. 2, pp. 89–103, 2010.

[9] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, pp. 476–493, June 1994.

[10] D. Romano and M. Pinzger, "Analyzing the evolution of web services using fine-grained changes," in *ICWS*, 2012, pp. 392–399.

[11] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: Nsga-ii," in *PPSN*, vol. 1917, 2000, pp. 849–858.

[12] G. Rudolph, "Evolutionary search for minimal elements in partially ordered finite sets," in *Evolutionary Programming*, ser. Lecture Notes in Computer Science, vol. 1447, 1998, pp. 345–353.

[13] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach," *IEEE Trans. Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.

[14] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *ISSTA*, 2007, pp. 140–150.

[15] Y. Zhang, M. Harman, and S. L. Lim, "Empirical evaluation of search based requirements interaction management," *Information & Software Technology*, vol. 55, no. 1, pp. 126–152, 2013.

[16] Z. Li, Y. Bian, R. Zhao, and J. Cheng, "A fine-grained parallel multi-objective test case prioritization on gpu," in *SSBSE*, 2013, pp. 111–125.

[17] E. R. Hruschka, R. J. G. B. Campello, A. A. Freitas, and A. C. P. L. F. De Carvalho, "A survey of evolutionary algorithms for clustering," *Trans. Sys. Man Cyber Part C*, vol. 39, no. 2, pp. 133–155, Mar. 2009.

[18] E. Falkenauer, *Genetic Algorithms and Grouping Problems*. New York, NY, USA: John Wiley & Sons, Inc., 1998.

[19] C. R. O. Al Jadaan and L. Rajamani, "Improved selection operator for ga," *Journal of Theoretical and Applied Information Technology*, vol. 4, no. 4, 2008.

[20] S. N. Sivanandam and S. N. Deepa, *Introduction to Genetic Algorithms*, 1st ed. Springer Publishing Company, Incorporated, 2007.

[21] J. Shelburg, M. Kessentini, and D. R. Tauritz, "Regression testing for model transformations: A multi-objective approach," in *SSBSE*, ser. Lecture Notes in Computer Science, vol. 8084, 2013, pp. 209–223.

[22] D. Nam and C. H. Park, "Multiobjective simulated snnealing: a comparative study to evolutionary algorithms," *International Journal of Fuzzy Systems*, vol. 2, no. 2, pp. 87–97, 2000.

[23] S. Raemaekers, A. v. Deursen, and J. Visser, "The maven repository dataset of metrics, changes, and dependencies," in *MSR*, 2013, pp. 221–224.

[24] H. B. Mann and W. D. R., "On a test of whether one of two random variables is stochastically larger than the other," *Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947.

[25] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[26] S. D. S.Weardon and D. Chilko, *Statistics for Research. Probability ans Statistics*. John Wiley and Sons, 2004.

[27] W. Hopkins, *A new view of statistics*. Internet Society for Sport Science, 2000.

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

[29] S. Mancoridis, B. S. Mitchell, Y.-F. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *ICSM*, 1999, pp. 50–59.

[30] S. Mancoridis, B. S. Mitchell, C. Rorres, Y.-F. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *IWPC*, 1998, pp. 45–52.

[31] B. S. Mitchell and S. Mancoridis, "Using heuristic search techniques to extract design abstractions from source code," in *GECCO*, 2002, pp. 1375–1382.

[32] K. Mahdavi, M. Harman, and R. M. Hierons, "A multiple hill climbing approach to software module clustering," in *ICSM*, 2003, pp. 315–324.

[33] M. Harman, S. Swift, and K. Mahdavi, "An empirical study of the robustness of two module clustering fitness functions," in *GECCO*, 2005, pp. 1029–1036.