Finding the Best Compromise Between Design Quality and Testing Effort During Refactoring

Rodrigo Morales*, Aminata Sabané*, Pooya Musavi*, Foutse Khomh*, Francisco Chicano†, Giuliano Antoniol*
*Polytechnique Montréal, Canada; {rodrigo.morales, pooya.musavi, aminata.sabane, foutse.khomh, giuliano.antoniol}@polymtl.ca,
†Dept. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Andalucía Tech, Spain; chicano@lcc.uma.es

***Abstract*—Anti-patterns are poor design choices that hinder code evolution, and understandability. Practitioners perform refactoring, that are semantic-preserving-code transformations, to correct anti-patterns and to improve design quality. However, manual refactoring is a consuming task and a heavy burden for developers who have to struggle to complete their coding tasks and maintain the design quality of the system at the same time. For that reason, researchers and practitioners have proposed several approaches to bring automated support to developers, with solutions that ranges from single anti-patterns correction, to multiobjective solutions. The latter approaches attempted to reduce refactoring effort, or to improve semantic similarity between classes and methods in addition to removing anti-patterns. To the best of our knowledge, none of the previous approaches have considered the impact of refactoring on another important aspect of software development, which is the testing effort. In this paper, we propose a novel search-based multiobjective approach for removing five well-known anti-patterns and minimizing testing effort. To assess the effectiveness of our proposed approach, we implement three different multiobjective metaheuristics (NSGA-II, SPEA2, MOCell) and apply them to a benchmark comprised of four open-source systems. Results show that MOCell is the metaheuristic that provides the best performance.**

I. Introduction

Software system design is critical. Previous studies have provided evidence that defects related to system design are considerably more expensive to fix than those introduced during its implementation [1–4]. Therefore, design quality is an attribute that should be controlled, in order to ease the introduction of new features, code enhancements and other coding tasks. To assess the quality of a software system, several quality models and metric suites haven been proposed in the literature [5, 6]. Though these approaches allow to identify problematic components in several aspects like cohesion, coupling, or inheritance, they provide little or no information about how to improve the quality in those components. Another way to evaluate design quality is by the identification of poor design choices known as *anti-patterns* [7].

Anti-patterns hinder the maintenance and evolution of a system. An example of anti-pattern is the *Spaghetti Code*, which is a class without structure that declares long methods without parameters [8]. This anti-pattern depicts an abuse of procedural programming in object-oriented systems, that prevents code reuse. In a previous study, Bavota et al. [9] found that industrial developers assign a very high severity level to this anti-pattern and the *Blob*, which is a large controller class "that implements too many responsibilities" and is associated with passive data classes. Blob classes are considered to be bad programming practices [7, 8] because they hinder code maintainability and testing. In another study, Khomh et al. [10] found that there is a strong correlation between the occurrence of anti-patterns and the change-proneness of source files. Moreover, Taba et al. [11] and D'Ambros et al. [12] found that source files that contain anti-patterns tend to be more fault-prone than other source files. To remove anti-patterns developers apply a series of behavior-preserving code transformations known as refactoring [13–15].

Several studies have assessed the benefits of refactoring in Academia and Industrial contexts. Rompaey et al. [16] found that refactoring can reduce over 50% of memory usage, and it can increase the startup time of an application by 33%. In an empirical study with several revisions of an open-source system, Soetens and Demeyer [17] found that refactorings could reduce the Cyclomatic complexity [18], especially when they target duplicate code. Du Bois et al. [19] performed an experiment with students and observed that the decomposition of Blob classes improves the comprehensibility of the source code. In another industrial setting at Microsoft, Kim et al. [20] found that modules that undergo refactoring have less inter-module dependencies and less post-release faults. During the past years, practitioners and the research community have formulated the problem of refactoring anti-patterns as a combinatorial optimization problem [21–24]. These previous works have interspersed the correction of anti-patterns with other important objectives, like the reduction of coding effort, the preservation of domain semantics between classes and methods, etc. However, they have ignored another important aspect of the software development process, which is the testing effort.

Testing is an activity that aims to ensure that a system behaves according to its design specifications on a finite, but representative, set of test cases, taken from the infinite execution domain [25]. Researchers have investigated different ways to reduce testing effort and increase its effectiveness at different levels, *e.g.*, unit testing [26], integration testing [27], etc.; as well as for different software artifacts like documentation [28], and source code [26]. Refactoring operations are among the factors that can impact the testing effort of a system. In fact, *move method* or *extract class* refactorings applied to redistribute the responsibilities of a large class, either to its collaborators or to new entities, can allow units of code to be tested separately; reducing the number of scenarios to test. Moreover, writing

test cases for the refactored class is simplified as its related components can be easily replaced with mock objects during testing.

However, to the best of our knowledge, despite its importance, testing has been mostly overlooked so far, during automated refactoring. We hypothesize that if we consider the reduction of testing effort (as an additional objective) during automated refactoring, we can obtain refactoring solutions that not only improve the design quality of the system, in terms of anti-patterns correction, but also reduce the testing effort at the same time.

To test our hypothesis, we introduce *TARF* (Testing-Aware Refactoring Framework), a novel multiobjective approach (MO) for the problem of refactoring that minimizes the testing effort while improving the design quality. We perform a case study to assess the effectiveness of TARF using four different metaheuristics (one single and three multiobjective) and a benchmark of four open-source systems. Results show that TARF can remove up to 63% of anti-patterns, while reducing the testing effort by 21%.

Paper organization. The remainder of this paper is organized as follows. Section II provides background details about refactoring and testing. Section III describes our proposed approach while Section IV describes our experimental methodology. Section V presents and discusses the results of our experiments. Section VI discloses the threats to the validity of our study. Section VII overviews the related literature. Finally, Section VIII concludes our work and lays out some directions for future work.

II. Background

Refactoring. It is a reengineering technique that transforms the structure of a code, without altering its behavior [25]. It aims to improve the maintainability of systems at the class level by reorganizing methods and attributes; at system level by adding, modifying or removing entities (classes, interfaces, etc.) as well as their respective relationships. As a result, the current design of the system is transformed into an *improved version* according to the goals defined by the software maintainer. Refactorings can be used to remove *anti-patterns* [7] in a system. Although anti-patterns do not cause direct failures in a system, classes that contain anti-patterns have been identified to change more frequently and to have a higher probability to experience faults in the future, than classes without anti-patterns [10]. In addition, classes with anti-patterns have been found to require more testing effort [29] than other classes. Hence, correcting anti-patterns is important to improve the maintainability of software systems.

The process of refactoring starts by the detection of anti-patterns. Once different anti-patterns have been detected, they need to be corrected in an optimal way. This step is cumbersome, as the number of candidate refactorings is typically extensive, and the order in which they have to be applied is uncertain. More formally, if k is the number of available refactorings, then, the number of possible solutions (NS) is given by $NS = (k!)^k$ [30], which results in a space of possible solutions that is too large to be explored exhaustively. Therefore, researchers have reformulated the problem of automated refactoring as a combinatorial optimization problem and proposed different techniques to solve it. The techniques range from single-objective approaches using local-search metaheuristics, *e.g.*, hill climbing, and simulated annealing [31, 32], to evolutionary techniques like single genetic algorithm, and multiobjective approaches, *e.g.*, NSGAII and MOGA [30, 33–35]. In this research work, we propose a MO approach that aims to optimize not only the number of anti-patterns corrected, but also the testing effort. In the following subsections, we explain how we measure and include testing effort in our proposed approach.

Testing effort measurement. We refer to testing effort as the number of test cases required for each class in a system, according to the MaDUM testing strategy [36]. MaDUM as well as other object-oriented (OO) testing strategies, *e.g.*, state-based condition [37], and the pre-and-post conditions testing [38] have been proposed to overcome the limitations of traditional techniques, *e.g.*, white-box and black-box testing, when testing OO systems. Indeed, as pointed out by many authors [36, 37], the traditional testing strategies used in the context of procedural programming are insufficient to test OO programs because they are conceived to test functions as stand-alone code units, raising the possibility of missing state-based errors occurring during intra-method interactions.

Among the testing strategies that consider the OO paradigm, we choose MaDUM because it does not require any kind of software artifact apart from the source code. Hence, a simple static analysis of the source code is enough to estimate the number of test cases required to find code deviations. Then, that estimation can then be leveraged by an automated approach to guide the refactoring process towards a design that minimizes the unit testing effort. Because testing all possible interactions between methods and attributes within a class is expensive, if not impossible, OO testing strategies seek to reduce the number of sequences of methods to test.

MaDUM testing uses a divide to conquer strategy to perform unit testing: the class is divided in data slices and its correctness is evaluated in terms of the correctness of all its slices tested separately. A *data slice* is the set of methods that access to a particular attribute (field) in a class. The identification of the data slices is based on the *enhanced call-graph (ECG)* and the *minimal data members usage matrix (MaDUM)*. The ECG represents the type of usages among the members of a class and it is defined as: $ECG(C) = (M(C), F(C), Emf, Emm)$, where $M(C)$ is the set of methods of C, $F(C)$ is the set of fields of C. $Emf = (m_i, f_j)$ indicates that method i accesses field j, and $Emm(m_i, m_j)$ that method i invokes method j. MaDUM is an $nf \times nm$ matrix where nf and nm are the numbers of fields and methods in the class. It is built using the ECG of the class. MaDUM defines four categories to classify the methods, that are: class constructors (c), transformers (t), *i.e.*, methods that modify the state of a field, reporters (r), *i.e.*, methods that return the value of an attribute, and others (o), *i.e.*, methods that do not fall in the previous categories.

Once the MaDUM of a class has been built, the order for testing that class is the following: first reporters are tested to ensure that they do not alter the state of the attribute they are reporting on. Constructors are then tested to ensure that attributes are correctly initialized, and in the right order. The testing of transformers is performed by generating for each slice all permutations of transformers in that slice for each constructor context. For example, let c be the set of constructors and t the set of transformers in a given slice, it is necessary to produce $|c| \times |t|!$ test cases, where the function $|x|$ denotes the cardinality of the set x. Others (o) are tested using traditional black or white-box testing. Note that a method m_j can access a field f_i directly or indirectly through another method m_k invoked by m_j. Although, in the last scenario, if m_j accesses f_i only through m_k, and m_k has been already tested in the f_i slice, there is no need to retest m_j in the slice f_i. The total number of test cases required to test a given class is computed as follows:

$$te(C_i) = |c| + |r| + |o| + \sum_{i=1}^{n} |c_i| * |t_i|! \quad (1)$$

Where $|x|$ is the number of methods of type x in the class, n the number of slices in the class, and $|x_i|$ the number of methods of type x in the slice i. When a class in a system presents a high number of transformers in a slice, *i.e.*, methods that modify the state of an attribute, the probability of having points of failures increases, and consequently a higher number of test cases is required in order to thoroughly test the class. Hence to reduce testing effort and the risk of failures, due to state inconsistencies, the number of transformers within a slice should be kept as low as possible. Considering testing effort in automated-refactoring can alleviate the problem by prioritizing refactorings that reduce the number of transformers, and therefore the number of test cases required. For example, since for each given slice we need a number of test cases equivalent to the number of permutation of slice transformers $\times$ each constructor context, if we apply *move method* refactoring to move one or more of these transformers from a large class to any other class in the system that has a low number of transformers, the number of test cases required for the large class will decrease in a significant proportion, while the number of test cases for the small class will only increase slightly, making the sum of test cases for both, the large and small classes, less than they were before the refactoring. Nevertheless, certain refactorings like the introduction of parameter object class increase the number of test cases (one for each parameter extracted from the source class to the new class object, plus one for the new constructor).

Multiobjective optimization (MO). It is the problem of finding vectors of decisions variables which satisfies constraints and optimize a vector function whose elements represent the objective functions. In the following we describe the MO search-based techniques used in this paper.

- The Non-dominated sorting genetic algorithm (NSGA-II) [39] proceeds by evolving a new population from a starting solution by applying variation operators like crossover and mutation. Then it merges the candidate solutions from both populations, and sort them according to their rank, extracting the best candidates to create the next generation. When there is a conflict of selecting individuals with the same ranking, the difference is solved using a measure of density in the neighborhood, *a.k.a.,* crowding distance.
- The Strength Pareto Evolutionary Algorithm 2 (SPEA2) [40] uses variation operators to evolve a population, like NSGAII, but with the addition of an *external archive*. The archive is a set of non-dominated solutions, and it is updated during the iteration process to maintain the characteristics of the non-dominated front. In SPEA2, each solution is assigned a fitness value that is the sum of its strength fitness plus a density estimation.
- The Multiobjective Cellular Genetic Algorithm (MOCell) is a cellular algorithm [41], that includes an external archive like SPEA2 to store the non-dominated solutions found during the search process. It uses the crowding distance of NSGA-II to maintain the diversity in the Pareto front. Note that the version used in this paper is an *asynchronous* version of MOCell called aMOCell4 [42]. The selection consists in taking individuals from the neighborhood of the current solution (cells) and selecting another one randomly from the archive. After applying the variation operators, the new offspring is compared with the current solution and replaces the current solution if both are non-dominated, otherwise the worst individual in the neighborhood will be replaced by the offspring.

We choose these MO algorithms because they are evolutionary techniques that have been successfully applied to solve combinatorial discrete problems in several contexts.

III. Testing-Aware Automated Refactoring

This section presents the foundations of our proposed approach *TARF* that aims to improve the design quality of OO systems, while minimizing the effort required to test the system. We use the incidence rates of anti-patterns as a proxy for software design quality and measure the testing effort as the number of unit test cases required to find code deviations in the classes of a system, according to the MaDUM technique. Algorithm 1 summarizes the main steps of *TARF*. We describe each step in more details in the following paragraphs.

Generation of Abstract Model. This step consists in generating a light-weight representation of the system under maintenance that contains information about the entities (classes, methods, and attributes) and how they interact between each other. This model is used to detect anti-patterns, compute testing effort, and apply refactoring sequences in the serach of non-dominated solutions.

Computation of MaDUM. At this step, we compute for each individual class, the corresponding ECG, and MaDUM. Once the MaDUM is built, we can compute the number of test cases required per class, and then we sum this value for all classes in the system to obtain the overall testing effort.

Detection of anti-patterns. This step consists in detecting

Algorithm 1: TARF Approach

```
   Input  : System to refactor (SW)
   Output : Optimal refactoring sequence(s)
1  Pseudocode TARF (SW)
2      AM=Generation of Abstract Model
       /* From the source code generate a light-weight
          representation of the code                          */
3      MAD=Computation of the MaDUM matrix
       /* Generate the ECG and the MaDUMs for each class in
          the system                                          */
4      AP=Detection of Anti-patterns
       /* Detect anti-patterns in the system and generate a
          map of classes that contain anti-patterns           */
5      RS=Generation of refactoring opportunities
       /* Generate a list of refactoring operation
          candidates based on the previous step               */
6      Search-based refactoring(AM, MAD, RS)/* This is a generic
          template of a Metaheuristic used to find the
          optimal refactoring solution                        */
7      return
8  Procedure Search-based refactoring(AM, MAD, RS)
9      AM' = AM
10     MAD' = MAD
11     S = RS
12     P_0 = GenerateInitialPopulation(S)
13     A = ∅
14     for all S_i ∈ P_0 do
15         apply_refactorings(AM', S_i)
16         compute_Quality(AM')
17         compute_TestingEffort(AM', MAD)
18     end for
19     Evaluate(P_0)
20     A = Update(A, P_0)
21     t = 0
22     while not StoppingCriterion do
23         t = t + 1
24         P_t = Variation_Operators(P_{t-1})
25         for all S_i ∈ P_t do
26             apply_refactorings(AM', S_i)
27             compute_Quality(AM')
28             compute_TestingEffort(AM', MAD')
29         end for
30         Evaluate(P_t)
31         A = Update(A, P_t)
32     end while
33     best_solution = A
34     return best_solution
```

anti-patterns in the system. We use the count of anti-patterns to assess the design quality of the system.

Generation of refactoring opportunities. Fowler [43] describes a set of refactoring operations that can be applied to remove anti-patterns in a system. This step consists in selecting refactoring operations that can remove anti-patterns detected in the previous step.

Search-based refactoring. In this step, we apply search-based techniques to find the best sequence of refactorings that achieves a maximum reduction of the number of anti-patterns, while keeping the testing effort as minimum as possible. Hence, a candidate solution of our approach is represented as a sequence of refactorings. A generic template for the search-based techniques employed by TARF is presented from line 8 to line 34, and is described below:

The algorithm takes as input the abstract model (*AM*), the set of MaDUMs for each class (*MAD*), and the list of candidate refactorings (RS). With this information, it randomly generates sequences of refactorings that constitutes the initial population P_0. In the next step, each sequence is applied to *AM'* and the number of anti-patterns and test cases are computed. Then, the sequences are evaluated and sorted according to their objective values, (lines 8-20), and the non-dominated solutions are retrieved (A). After generating the initial population, the main search loop starts (line 22). For each iteration t, a new set of solutions P_t are evolved from P_{t-1} using the variation operators defined for each search technique. Then, each candidate solution in P_t is applied on the system using a copy of the abstract model. The testing effort and the number of anti-patterns are then measured (lines 24-29). The non-dominated solutions are retrieved (lines 30-31). The process ends when the algorithm reaches the stop condition. For example a predetermined execution time, or a maximum number of evaluations.

IV. Case Study Design

The *goal* of this case study is to assess the effectiveness of TARF in correcting anti-patterns in OO systems, while reducing the effort required to test the system.

The *quality focus* is the improvement of the design quality of OO systems and the reduction of testing effort through search-based refactoring. The *perspective* is that of researchers interested in developing automated refactoring tools and practitioners interested in improving the design quality of their software system while controlling for testing effort.

The *context* consists of four open-source software systems (ArgoUML, Gantt Project, JHotDraw, and Mylyn) and four evolutionary metaheuristics one single objective algorithm (*i.e.*, Genetic Algorithm) and three MO algorithms (MOCell, NSGA-II, and SPEA2). Table I presents relevant information about the systems under study. We select these systems because (1) they are open-source projects, with different purposes and sizes; and (2) they have been used in previous studies on anti-patterns and refactorings[23, 44, 45].

Table I: Descriptive statistics of the studied systems.

Name	Num. of classes	Num. of anti-patterns	Num. of initial test cases
ArgoUML 0.34	1754	456	587220340
GantProject 1.10.2	188	38	2510
JhotDraw 5.4	450	89	10943
Mylyn 3.4	2365	183	7303813

We instantiate our generic approach TARF using one single-objective approach (genetic algorithm) and compared to three MO metaheuristics, described in Section II. We choose single-objective Genetic Algorithm (GA) as it has been used in previous studies [32, 46] on refactoring. More details about these metaheuristics are available in [47].

The search of optimal solutions is guided by the following two fitness functions.

- $Quality = 1 - \frac{NDC}{NC \times NAT}$, where NDC is the number of classes that contain anti-patterns, NC is the number of classes, and NAT is the number of different types of anti-patterns. This objective function was first formulated by Ouni et al. [30]. We choose to follow his formulation because it is easy to implement and computationally inexpensive. The value of $Quality$ increases when the number of anti-patterns in the system is reduced after applying a refactoring sequence. The output value of $Quality$ is normalized between 0 and 1. A value of 1

represents the complete removal of anti-patterns, hence we aim to maximize the value of $Quality$.

- $STF = \sum_{i=1}^{n} te(C_i)$, where STF is the test effort of the system, Te is calculated from Equation (1), and n is the total number of classes. We aim to minimize the value of STF.

In our GA implementation, a single objective function is obtained by multiplying the values of NDC and STF, while the other metaheuristics do not combine the values of the fitness functions, but use them as a tuple.

Solution representation.. We use a vector representation where each element is a refactoring operation (RO) that includes: an *Id* field (unique identifier) to know which refactorings have been applied so far. The anti-pattern's source class, and the type of refactoring. The type of refactoring is used to determine if a conflict with a previous RO in the sequence will arise. In addition to this, we can have more fields providing extra information, *e.g.*, target class and method name for move method, or long method names for spaghetti code class.

Variation operators. We define the variation operators as follows. Binary tournament for individual selection, 'Cut and splice" as crossover operator, which consists in randomly setting a *cut point* for each parent, and recombining with the elements of the other parent's cut point and vice-versa, resulting in two individuals with different lengths. For mutation we propose a new operator that consists in choosing a random point in the sequence and removing the refactoring operations from that point to the end. Then, we complete the sequence by adding new random refactorings until we reach the original size of the sequence. By doing this, we reduce the overhead of validating the correctness of refactorings added in the middle of a valid sequence, and bring more diversity.

A. Dependent and Independent Variables

To assess whether TARF can improve design quality while reducing testing effort, we consider the following dependent and independent variables:

The *independent variables* are our four selected metaheuristics, *i.e.*, GA, MOCell, NSGA-II, and SPEA2.

The *dependent variables* are the following two metrics used to evaluate the effectiveness of TARF at improving the design quality of systems while reducing the testing effort.

- Difference of number of anti-patterns between the original and the refactored version (DAP). DAP is an indication of the improvement of the design quality of the system. A DAP value close to the number of of anti-patterns in the original system denotes good design quality as most of the anti-patterns are removed, while a negative value indicates a degradation of the quality after refactoring, *i.e.*, a rise in the number of anti-patterns.
- Difference of required test cases between the original and the refactored version (DTC). Like DAP, a positive DTC value indicates a reduction of the testing effort, while a negative value means the contrary effect, and 0 means not improvement at all.

The anti-patterns considered in this case study are: Long parameter list, Blob, Lazy class, Speculative Generality, and Spaghetti Code. We select these anti-patterns because (1) they are well defined in the literature, with the recommended steps to remove them [7, 43], (2) they are recognized easily by developers [9], (3) they have been studied in previous works [10, 44, 45, 48] and have been identified to be problematic to the evolution of systems [9].

To detect anti-patterns in a system, we use the tool *DECOR* [44]. We select this tool because it reports the highest recall in the literature, and has been applied in several studies on anti-patterns and code smells[10, 23, 49, 50].

We now briefly describe the characteristics of each anti-pattern and the refactoring strategies followed to correct them.

Blob (BL): two main characteristics of classes containing BL anti-pattern is that they have a large size with low cohesion. Hence the strategy for refactoring such classes is to decompose them by moving functionality to related classes, through move method. In this way we do not only reduce the size of the BL class, but distribute the responsibilities among other classes in the system. To detect candidate classes to receive functionality from BL classes, we search for classes that contain methods or attributes that are extensively used by the BL class. When there is no suitable class to move any existing method, we try to extract functionality to a new class by selecting the methods that do not call or are called by other methods in the class, *i.e.*, with low cohesion. After moving the functionality from the Blob to an existing or newly created class, we update the existence references in all the classes in the system.

Lazy class (LC) is a small class with low complexity that does not justify its existence in the system, hence the proposed refactoring is to *inline* that class, removing the LC after moving its features to another class in the system. This refactoring is comprised of a series of *low level* refactorings that have to be applied in a specific order, *e.g.*, move method(s), move field(s), update call sites, and delete class. Also, we have to satisfy certain preconditions and postconditions in order to preserve the semantic of the classes. For example, one precondition is that we do not inline parent classes, as inlining those classes will introduce regression, but instead we could *collapse the hierarchy* of a LC. An example of postcondition is that the target class implements the features of the LC class after refactoring. In order to improve the quality of the design, in terms of cohesion, the destiny class has to be related to the LC class to some extent. To select such a class, we iterate over all the classes in the system, searching for methods and attributes that access the features of the LC class directly, or by public accessors (getters or setters). From those classes, we choose the one with the larger number of access to the LC class.

Long parameter list classes (LP) are classes that contain one or more methods with an excessive number of parameters, in comparison with the rest of the entities (to detect LP classes, DECOR defines a threshold based on the computation of boxplot statistics involving all the classes in the system). Hence, the refactoring strategy consists in (1) extracting a new class for each long parameter list method, that will encapsulate a

group of parameters that are often passed together, and that can be used by more than one method or class (improving the readability of the code); and (2) updating the signature of each method to remove the migrated parameters, and updating the callers and method body in the LP class to instantiate and replace the parameters with the new parameter object.

Spaghetti Code (SC) are characterized by their lack of structure and the presence of long methods without parameters. The refactoring strategy proposed for SC includes the extraction of one or more long methods as new objects. This requires creating a new class for each long method, where the local variables become fields, and a constructor that takes as parameter a reference to the SC class; the body of the original method is copied to a new method *compute*, and any invocation of the method in the original class will be referenced through the parameter (stored as final field) to the SC class. Finally, the *original* long method is replaced in the SC class by the creation of the new object, and a call to *compute* method.

Finally, for classes affected by Speculative Generality (SG) (*i.e.*, an abstract class that is not actually needed, as it is not specialized by any other class), we first pull up the methods and attributes from the child class to the parent class. Next, we update the constructor and then we remove the children class from the system, the *abstract modifier* from the parent class, and update the call sites and types to point to the parent class. There is one case where we omit the application of this strategy, and it is when the child class is defined as inner class inside a different class. Inner classes are an integral part of the event-handling mechanism in user interfaces events [51], which differs from the definition of the SG anti-pattern, *ergo* moving the features of inner classes to other entities may introduce a regression in the system

B. Research Questions

We formulate the research questions of our case study as follows :

(RQ1) To what extent can the proposed approach correct anti-patterns and reduce testing effort?
This research question aims to assess the effectiveness of TARF at improving design quality, while reducing testing effort.

(RQ2) To what extent is design quality improved after refactoring when considering testing effort?
While the number of anti-patterns in a system serves as a good estimation of design quality, there are other quality attributes such as those defined by the QMOOD quality model [5] that are also relevant for developers. This research question aims to assess the impact that the application of TARF has on these aforementioned attributes.

C. Analysis Method

To answer RQ1, we compared the results of applying the three MO metaheuristics (MOCell, NSGA-II, and SPEA2), to the ones of the mono-objective approach to ensure that the refactoring solutions found by the first ones provide better compromise solutions between quality and testing effort, than those found by the former single objective formulation. Otherwise, there is no need to define a MO formulation. We implement all the metaheuristics described before using the jMetal Framework [52], which is a wide-use library for solving optimization problems. Given that we are comparing techniques with different sources of information (population, archive, etc.), we opt for number of evaluations as the stop criteria, and set it to 2500, which is an accepted value for optimization problems in general.

Parameters of the metaheuristics. We are using four evolutionary metaheuristics in our experiments. As we mentioned before, they make use of variation operators (selection, mutation and crossover) to move through the decision space in the search for an optimal solution. To determine the best parameters for our metaheuristics, we run each algorithm with different configurations 30 times, in a *factorial design* in the following way: we test 16 combinations of mutation probability $p_m = (1, 0.8, 0.5, 0.2)$, and crossover probability $p_c = (1, 0.8, 0.5, 0.2)$, and obtained the best results with the pair (0.8, 0.8). This is not a surprise as in [22] they found high mutation and crossover values to be the best trade for algorithm performance. For the specific problem of automated refactoring, setting the initial size of the refactoring sequence is crucial to find the best sequence in a reasonable time, especially when we have a huge number of candidate refactorings, because setting a low value will lead to find poor solutions in terms of anti-patterns correction. On the contrary, if the initial size is very large, we may obtain the reverse effect because applying many refactorings not necessarily implies better quality, as refactorings can improve one aspect of quality while worsen others. Hence, we experiment running the algorithms with three relative thresholds: 25%, 50%, 75% and 100%, of the total number of refactoring opportunities, and found that 50% give us the best results in terms of removal of anti-patterns and reduction of testing effort. The population size is set to 100 individuals as default value.

In order to measure the performance of the MO metaheuristics used here, we need to consider the quality of their resulting non-dominated set of solutions [53]. We use two quality indicators for that purpose; the Hypervolume (HV) and Spread (Δ). HV provides a measure that considers the convergence and diversity of the resulting approximation set. Higher values of the HV metric are desirable. Spread measures the distribution of solutions into a given front. Lower values close to zero are desirable, as they indicate that the solutions are uniformly distributed. For further details we refer the reader to the source references [39, 54].

To answer RQ2, we use the Quality Model for Object-Oriented Design (QMOOD) [5] to evaluate the impact of the proposed refactoring sequences on several quality attributes. This model is suitable for automated-refactoring experimentation because the definition of quality attributes is obtained from a metric-quotient weighted formula to determine how close does the refactored design conforms to QMOOD model.

The rational for selecting QMOOD is that previous studies have used it before to assess the effect of refactoring [21, 22, 32]; it defines six desirable quality attributes (reusability, flexibility, understandability, functionality, effectiveness and extendibility) based on 11 OO metrics. From the six quality attributes proposed by QMOOD, we omit *functionality* because refactoring *per se* is behavior-preserving, hence we do not expect an impact on this attribute. The formulas to compute the aforementioned quality attributes are presented in Table II. We present a brief descriptions of the attributes considered in this study below:

- Reusability: the degree to which a software module or other work product can be used in more than one computer program or software system.
- Flexibility: the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.
- Understandability: the properties of designs that enable it to be easily learned and comprehended. This directly relates to the complexity of design structure.
- Effectiveness: the design's ability to achieve desired functionality and behavior by using OO concepts.
- Extendibility: The degree to which a system can be modified to increase its storage or functional capacity

Table II: QMOOD evaluation functions.

Quality Factors	Quality Index Calculation
Reusability	-0.25 * DCC + 0.25 * CAM + 0.5 * CIS + 0.5 * DSC
Flexibility	0.25 * DAM - 0.25 * DCC + 0.5 * MOA +0.5 * NOP
Understandability	-0.33 * ANA + 0.33 * DAM - 0.33 * DCC + 0.33 * CAM -0.33 * NOP - 0.33 * NOM - 0.33 * DSC
Effectiveness	0.2 * ANA + 0.2 * DAM + 0.2 * MOA + 0.2 * MFA + 0.2 * NOP
Extendibility	0.5 * ANA -0.5 * DCC + 0.5 * MFA + 0.5 * NOP

where DSC is design size, NOM is number of methods, DCC is coupling, NOP is polymorphism, NOH is number of hierarchies, CAM is cohesion among methods, ANA is avg. num. of ancestors, DAM is data access metric, MOA is measure of aggregation, MFA is measure of functional abstraction, and CIS is class interface size.

We obtain the quality gain of the refactored design (D') by dividing each quality attribute value by the corresponding value for the original design (D).

V. Case Study Results

In this section we present the results of our case study with respect to our two research questions.

(RQ1) To what extent can the proposed approach correct anti-patterns and reduce testing effort?

One main feature of MO metaheuristics is that they do not produce a single solution as mono-objective techniques, *e.g.*, GA, but a set of solutions. From this set of solutions we are interested in those that are non-dominated, *i.e.*, the solutions whose objective values cannot be improved without worsening others. The set of non-dominated solutions for an instance of a problem is known as the Pareto set. The image of the Pareto optimal set by the vector objective functions is the Pareto front. Pareto reference Front (RF) is an approximation of the true Pareto Front, and similar to other combinatorial optimization studies [21], we assume that the production of the true Pareto front is not feasible, hence we use the reference front, created from the optimal values after 30 independent executions. In Figure 1 we present the Pareto reference front for JHotDraw, extracted from the three MO metaheuristics analyzed. The points in Figure 1 represent a compromise between quality and testing effort. The $x - axis$ represents the normalized values of quality, measured in terms of number of anti-patterns corrected; $y - axis$ represents the number of required test cases for that solution. The best solutions are found in the right-bottom corner of the plot. Hence, the software maintainer is able to choose a solution according to its preferences, for example if someone is interested mostly in reducing the number of test cases, she can opt for a solution located in the left-bottom of the plot. However, that solution would not remove as many anti-patterns as the solutions located in the middle or in the extreme right position in the plot. The advantage of considering testing as another objective function is that maintainers obtain the possibility to choose among trade multiple solutions.

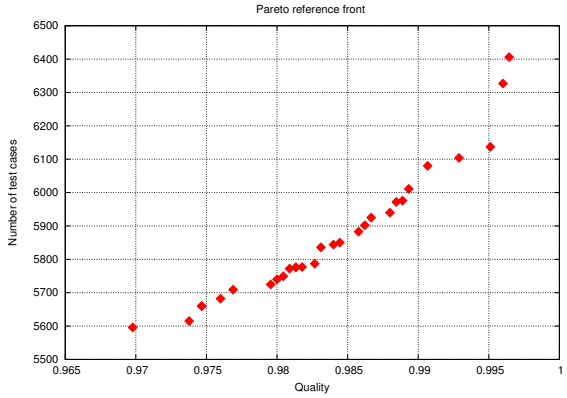

Fig. 1: The Pareto reference front of JHotDraw.

Note that to compare the three MO with GA, we need to select one solution (**bs**) in the Pareto front. One technique often used to determine the best solution in the Pareto Front of a problem, is selecting the solution that has the minimal distance with a hypothetical *ideal solution* using the Euclidean distance. The *ideal solution* for our approach is the one where we end with zero anti-patterns and the number of test cases is close to zero. The complete equation is provided in Equation (2).

$$bs = \min_{i=1}^{n} \left(\sqrt{(1 - Quality[i]^2) + (-1 * te[i]^2)} \right), \quad (2)$$

where n is the number of solutions in the Pareto front returned by the MO metaheuristic.

Once we extract the best solution from the Pareto reference front, we are able to compare it with the one of the single objective metaheuristic (GA). In Table III, we present the results obtained after applying the four metaheuristics to each of the programs studied. These results are median values of the 30 independent runs. Columns 2 to 6 contain the difference of the anti-patterns count per type, before and after refactoring, while DAP (column 7) is the sum of these columns. Similarly, DTC is the difference of number of required test cases between the original and the refactored version, and ROs (column 9) is

the number of refactoring operations. A negative number in a column indicates an increase from the original value, and "–" (columns 2-6) indicates zero anti-patterns detected. The best results of columns DAP and DTC are highlighted with dark grey. We observe that in general the three MO metaheuristics reduce more the number of test cases than the single approach. Moreover, in two systems (ArgoUML, JHotdraw) the mono-objective technique did not reduce, but considerably increase the testing effort. On the contrary, concerning anti-pattern correction, GA overcomes the MO metaheuristics (in the same systems), being ArgoUML the one with highest correction of anti-patterns and with the highest increase of test cases at the same time. Gantt project is championed by MOCell, and in Mylyn the average of the two metrics favor MOCell as well. If we consider the number of refactorings applied (column ROs), we observe that while longer refactoring sequences seem to reduce more anti-patterns, at least for the three first studied systems, a pattern to characterize the behavior of testing effort is less evident. While in ArgoUML and JhotDraw, the shortest sequences report the less difference in number of test cases, in Mylyn and Gantt the trend seems to be inverted.

Table III: Median count of anti-patterns removed, and number of test cases after refactoring.

Metaheuristic	BL	LC	LP	SC	SG.	DAP	DTC	ROs
ArgoUML								
GA	-1	34	335	1	1	370	-1336	1847
MOCell	0	1	8	0	0	9	140	64
NSGA-II	0	1	9	0	0	10	177	63
SPEA2	0	2	14	0	0	16	218	92
Gantt Project								
GA	0	1	12	–	2	15	-55	93
MOCell	4	2	14	–	4	24	535	381
NSGA-II	3	2	9	–	4	18	519	317
SPEA2	3	2	13	–	4	22	499	346
JHotDraw								
GA	–	1	59	–	–	60	-507	454
MOCell	–	0	21	–	–	21	5321	216
NSGA-II	–	1	28	–	–	29	5283	350
SPEA2	–	0	30	–	–	30	5301	389
Mylyn								
GA	1	19	101	–	–	121	556	1959
MOCell	1	18	77	–	–	96	7266802	2159
NSGA-II	1	17	81	–	–	99	7266748	1974
SPEA2	1	19	83	–	–	103	7266765	2047

We conclude that considering testing effort as an objective to minimize when applying automatic refactoring, one can significantly reduce the number of test cases, while keeping reasonable correction results.

Performance of the three multiobjective metaheuristics. In Table IV we present the mean and the standard deviation of the quality indicators (HV, Spread) values of the metaheuristics for each system on 30 independent runs. A special notation appears in the table: a gray colored background denotes the best (dark gray) and second-best (lighter gray) performing technique. The HV indicates that MOCell has been able to approximate the Pareto fronts with the highest indicator values, while in the second-best is disputed between NSGA-II and SPEA2, except for JHotDraw, where the former one overcomes the first one. Concerning the spread indicator, the best spread is divided between SPEA2 and MOCell, and NSGA-II appears to be the less effective metaheuristic. To determine the significance of the obtained results, we compute the Wilcoxon rank-sum test between two metaheuristics at time. The results are summarized in Table V. In each cell, a ▲ or a ▽ symbol implies a p-value < 0.05, indicating that the null hypothesis (the two distribution have the same median) is rejected; otherwise, a – is used. The ▲ denotes that the metaheuristic in the row obtained a better value than the one in the column; the ▽ indicates the opposite. Hence, the only conclusion we can draw from these results is that 1) MOCell overcomes SPEA2 and NSGA-II in more than a half of the systems analyzed in terms of HV, while the performance between NSGA-II and SPEA2 remains unclear. Concerning the spread indicator, we omit the results of the Wilcoxon test because the results were not statistically significant, thus we cannot draw any conclusion about the performance of the metaheuristics using this indicator. Note that the aim of this paper is not to propose a new multiobjective algorithm to perform automated refactoring, but *reformulate* the problem of refactoring to include testing effort as a goal regardless of the metaheuristic employed.

Although the obtained results point out that MOCell is the most effective technique for the formulation of automatic refactoring considering quality and testing effort, and among the metaheuristics studied, further studies with more systems, and more quality indicators are required to validate this result.

Table IV: Quality indicators Mean and standard deviation

	SPEA2	MOCell	NSGAII
	Hypervolume		
ArgoUML	$4.80e-01_{5.7e-03}$	$5.04e-01_{4.8e-03}$	$4.89e-01_{1.1e-02}$
Gantt	$0.00e+00_{0.0e+00}$	$2.30e-01_{1.6e-01}$	$1.00e-02_{2.0e-02}$
JHotDraw	$5.37e-01_{2.3e-02}$	$5.93e-01_{2.5e-02}$	$5.36e-01_{3.0e-02}$
Mylyn	$1.49e-01_{4.6e-02}$	$2.40e-01_{5.7e-02}$	$2.01e-01_{3.5e-02}$
	SPREAD		
ArgoUML	$5.60e-01_{1.0e-01}$	$5.94e-01_{6.3e-02}$	$6.12e-01_{2.2e-02}$
Gantt	$8.48e-01_{8.1e-02}$	$7.89e-01_{1.7e-01}$	$8.97e-01_{8.3e-02}$
JHotDraw	$6.95e-01_{4.3e-02}$	$6.45e-01_{5.0e-02}$	$7.23e-01_{3.7e-02}$
Mylyn	$9.63e-01_{1.3e-01}$	$9.77e-01_{1.6e-01}$	$1.09e+00_{2.3e-01}$

Table V: Wilcoxon rank-sum test for HV indicator.

	MOCell				NSGA-II			
	ARG	GAN	JHD	MYL	ARG	GAN	JHD	MYL
SPEA2	▽	▽	▽	–	–	–	–	
MOCell					–	▲	▲	–

(RQ2) To what extent is design quality improved after refactoring when considering testing effort?

Although we have shown that automated refactoring can improve the quality of the system and reduce the testing effort, some software maintainers may wonder whether the refactorings applied will produce a new code that is still readable, or if it will be easy to come back later and modify it or extend it. Since concepts such as reusability, or understandability in design quality are quite vague and hard to define, we consider the QMOOD evaluation functions

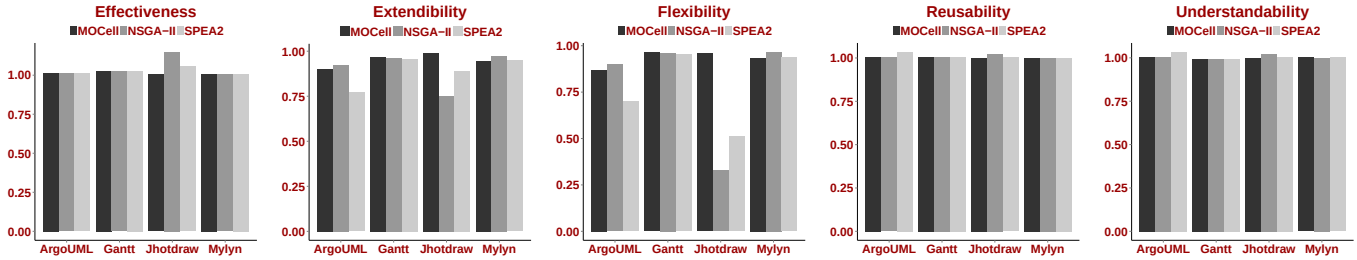

Fig. 2: The quality gain of the best refactoring solutions on QMOOD quality attributes.

as *examples* of how to correctly characterize good design properties. In Figure 2 we present the obtained quality gain values (change quotient) that we computed for each QMOOD quality attribute (QQA) before and after refactoring for each studied system.

We can observe that the system quality increases across the five QQA in an even manner, that ranges from 1.15 in effectiveness (Jhotdraw, MOCell) to 0.33 in flexibility (JHotDraw, NSGA-II).

We suggest that the low value in flexibility compared to effectiveness is in part the result of the weight that each of these two quality function assigns to MOA metric (0.2, 0.5 respectively). According to QMOOD, MOA is the number of user-defined classes, and we observe in Table III that MOCell removed less long parameter list anti-patterns than GA. We remind that the suggested refactoring for LP is *introduce parameter object*, which creates new classes to store the long parameter list, hence the increment in the number of user-defined classes is less in MOCell compared to GA. On the other hand, effectiveness assigns a lower weight to this metric (0.2), but integrates other desirable metrics related to OO design like abstraction, encapsulation and inheritance. Finally, understandability, reusability, and extendibility factors are benefited from the extensive application of move method refactorings, and reported an increment similar to effectiveness, because Move method is known to impact metrics like coupling (DCC), cohesion (CAM) and design size (DSC) that serves to calculate these quality attributes.

We conclude that our approach was successful in improving design quality not only in terms of anti-patterns correction, but also in terms of quality attributes such as understandability, reusability, flexibility, effectiveness and extendibility.

VI. Threats to validity

This section discusses the threats to validity of our study following common guidelines for empirical studies [55].

Construct validity threats concern the relation between theory and observation. This is mainly due to possible mistakes in the detection of anti-patterns, in the refactorings applied. We based the APs detection on DECOR [44], and despite the high recall and precision of DECOR, there is no warranty that we detect all the possible APs, or that those detected are indeed true APs. Concerning the application of refactorings, we manually validate the outcome of refactorings performed in source code and the ones applied to the abstract model to ensure that the output values of the objective functions correspond to the changes performed. However, we rely on the correct representation of the code by the abstract model. In this study we use PADL [56], which has been used in several studies concerning anti-patterns, design patterns, and software evolution with more than ten years of active development.

A second threat is the use of MaDUM as proxy to estimate the testing effort, because other techniques could bring different results. Moreover, MaDUM estimation does not include the effort of writing and running each test case. Instead, it gives an estimate of the number of test cases required to test the class and highlights classes with multiple transformers as difficult classes to be tested. Finally MaDUM only works at the level of unitary testing, without considering class interactions. Therefore, we can only claim that, no matter the testing strategy, automated-refactoring approaches should consider the impact of refactorings not only in terms of design quality but in testing effort.

Threats to internal validity concern our selection of anti-patterns, tools, and analysis method. In this study we used a particular yet representative subset of anti-patterns as proxy for design quality

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the constructed statistical models. In particular, we used a non-parametric test, Wilcoxon rank sum, that does not require any assumption on the underlying data distribution.

Threats to external validity concern the possibility to generalize our results. Our study focuses on four open source software systems having different sizes and belonging to different domains. Nevertheless, further validation on a larger set of software systems is desirable, considering systems from different domains, as well as several systems from the same domain. Future replications of this study are necessary to confirm our findings.

VII. Related work

In this section, we present works related to automated refactoring of anti-patterns, testing strategies, and discuss their differences with our work.

Automated refactoring. Harman and Tratt [21] introduce for the first time a multiobjective (MO) approach for the problem of refactoring, where objective functions were defined based on two conflicting quality metrics, and showed that this approach can find a good sequence of move methods refactorings solutions. The disadvantage of this approach is that they restricted the refactoring operations to one type and did not consider other aspects like the testability of the code. Ouni et al. [24] propose a MO approach based on NSGA-II, with two conflicting objectives: removing anti-patterns, while preserving semantic coherence. For the first objective, they generate a set of rules to characterize anti-patterns from a set of bad design examples. The second objective is achieved by implementing two techniques to measure similarity among classes, after refactoring. The first technique evaluates the cosine similarity of the name of the constituents, *e.g.*, methods, fields, types; and the second technique considers dependencies between classes. Mkaouer et al. [57] propose an extension of this work, by allowing user's interaction with refactoring solutions. Their approach consists in the following steps: (1) a NSGA-II algorithm proposes a set of refactoring sequences; (2) an algorithm ranks the solutions, and present them to the user to be judged; (3) a local-search algorithm updates the set of solutions after several user iterations, or when many refactorings have been applied. However, they did not consider the impact of the refactorings proposed on testing effort. Moghadam and Cinnéide [58] propose a single-objective automated approach to conform a desired design, described as an UML diagram. The approach takes as input the original source code and the UML diagram of the desired model. Then it generates an UML diagram from the source code and maps the differences between the two design models to a sequence of code-level refactorings to be applied in the source code to reach the desired model. The disadvantage of this approach is that they assume that this model can represent all the design quality features that a software maintainer wants to improve, which is not always the case.

Our proposed approach differs from the aforementioned works in the following points: it introduces for the first time testing effort as a new objective to satisfy; 3 out of 4 of these approaches require additional input from the user, *e.g.*, bad design examples, or desired design model. However, in practice it is not always feasible to have that information at hand. Moreover, it is also discouraging for an automated approach to put extra work on the user like creating and managing a database of bad design examples.

Testing Strategies. Testing is an essential but expensive activity to ensure software quality and reliability. Beizer [59] estimates the cost of software testing at 50% of the overall cost of software, hence researchers investigate various directions to reduce testing effort and increase its effectiveness. Studies related to factors that impact testing effort can be found in [26, 29]; while approaches to automatically generate test data are found in [60, 61]. Finally other studies define strategies that can efficiently target specific type of system or specific kind of faults. In this category, we can cite the different OO strategies that have been proposed to overcome traditional testing strategies limitations regarding the test of OO systems: state based testing [36], pre-and-post conditions testing [38], and MaDUM testing [62]. Another direction to reduce testing effort is to refactor a system specifically for testing. Belonging to this last category are the refactoring as testability transformation works [63, 64]. Refactoring as testability transformation is different from refactoring of anti-patterns in the sense that the system is not changed to improve the design quality of a system, but another version is created just to facilitate the generation of test data that will be used to test the original system. To the best of our knowledge, there is no work that automatically apply refactoring of anti-patterns to reduce testing effort. Sabane et al. [29] present some refactoring actions to reduce testing effort based on the MaDUM strategy. They manually apply some extract method refactorings to reduce the number of transformers of a class under test. These refactorings were performed manually, attempting to reduce the testing effort, not to remove APs or improve design quality. In fact, some of them were found to decrease the understandability of the class. However our approach aims to provide a mean to remove the anti-patterns automatically, considering the number of test cases to propose solutions that lead to a design with good quality and with less effort for the testers.

VIII. Conclusion and future work

This paper presents a novel search-based refactoring approach that includes testing effort, for the first time, as a mean to reduce testing effort of refactoring solutions that improve the design quality of a system. We found that the solutions proposed by our approach maintain a compromise between anti-patterns corrected, and number of test cases required to test an object-oriented system, according to MaDUM strategy. We validate our approach with four different metaheuristics, one single and three multiobjective, and found that we get better results with the latest approach in a benchmarking comprised of four open-source systems. The results suggest that MOCell, a cellular genetic algorithm, can provide the best performance among the other two multiobjective metaheuristics. But this finding has to be corroborated with more extensive studies.

We also assessed the design quality of the solutions proposed using five quality attributes defined in the hierarchical QMOOD model, and found that we can increase the quality in terms of reusability, flexibility, understandability, effectiveness, and extendibility.

As future work, we plan to extend our approach with additional refactorings to remove anti-patterns from other domains such as web and mobile applications, and also consider the impact of refactoring in other aspects of testing like derived classes and integration testing.

REFERENCES

[1] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[2] W. S. Humphrey, T. R. Snyder, and R. R. Willis, "Software Process Improvement at Hughes Aircraft," *IEEE Software*, vol. 8, no. 4, pp. 11–23, 1999.

[3] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What we have learned about fighting defects," in *Proc. of the 8th IEEE Symposium on Software Metrics*, 2002, pp. 249–258.

[4] S. McConnell, *Code Complete*. Microsoft, 2004.

[5] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *Software Engineering, IEEE Transactions on*, vol. 28, no. 1, pp. 4–17, 2002.

[6] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, Jun 1994.

[7] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1^{st} ed. John Wiley and Sons, March 1998.

[8] A. J. Riel, *Object-oriented design heuristics*. Addison-Wesley Reading, 1996, vol. 335.

[9] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do they really smell bad? a study on developers' perception of bad code smells," in *Software Maintenance and Evolution (ICSME), 2014 IEEE Int'l Conference on*. IEEE, 2014, pp. 101–110.

[10] F. Khomh, M. D. Penta, Y.-G. Gueheneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Softw. Engg.*, vol. 17, no. 3, pp. 243–275, Jun. 2012.

[11] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting bugs using antipatterns," in *Proc. of the 29th Int'l Conference on Software Maintenance*, 2013, pp. 270–279.

[12] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *Quality Software (QSIC), 2010 10th Int'l Conf. on*. IEEE, 2010, pp. 23–31.

[13] M. Fowler, *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[14] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[15] W. G. Griswold, "Program restructuring as an aid to software maintenance," Ph.D. dissertation, University of Washington, Seattle, WA, USA, 1992.

[16] B. van Rompaey, B. Du Bois, S. Demeyer, J. Pleunis, R. Putman, K. Meijfroidt, J. C. Dueas, and B. Garcia, "Serious: Software evolution, refactoring, improvement of operational and usable systems," in *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conf. On*, 2009, Conference Proceedings, pp. 277–280.

[17] Q. D. Soetens and S. Demeyer, "Studying the effect of refactorings: A complexity metrics perspective," in *Quality of Information and Communications Technology (QUATIC), 2010 7th Int'l Conf. On the*, 2010, Conference Proceedings, pp. 313–318.

[18] T. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, dec 1976.

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

[20] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proc. of the ACM SIGSOFT 20th Int'l Symposium on the Foundations of Softw. Eng.*, ser. FSE '12. ACM, 2012, pp. 50:1–50:11.

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

[22] M. O'Keeffe and M. O. Cinnéide, "Getting the most from search-based refactoring," *Gecco 2007: Genetic and Evolutionary Computation Conference, Vol 1 and 2*, pp. 1114–1120, 2007.

[23] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*. IEEE, 2011, pp. 81–90.

[24] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "Search-based refactoring: Towards semantics preservation," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, Conference Proceedings, pp. 347–356.

[25] P. Bourque, R. E. Fairley *et al.*, *Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.

[26] M. Bruntink and A. van Deursen, "An empirical study into class testability," *J. Syst. Softw.*, vol. 79, no. 9, pp. 1219–1232, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2006.02.036

[27] Y. Le Traon, T. Jéron, J.-M. Jézéquel, and P. Morel, "Efficient object-oriented integration and regression testing," *Reliability, IEEE Transactions on*, vol. 49, no. 1, pp. 12–25, 2000.

[28] B. Baudry and Y. Le Traon, "Measuring design testability of a uml class diagram," *Information and software technology*, vol. 47, no. 13, pp. 859–879, 2005.

[29] A. Sabane, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc, "A study on the relation between antipatterns and the cost of class unit testing," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 2013, pp. 167–176.

[30] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 2013.

[31] M. O'Keeffe and M. Ó. Cinnéide, "Search-based software maintenance," in *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10^{th} European Conference on*, 2006, Conference Proceedings, pp. 10 pp.–260.

[32] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," *GECCO 2006: Genetic and Evolutionary Computation Conference, Vol 1 and 2*, pp. 1909–1916, 2006.

[33] C. L. Simons, I. C. Parmee, and R. Gwynllyw, "Interactive, evolutionary search in upstream object-oriented class design," *Software Engineering, IEEE Transactions on*, vol. 36, no. 6, pp. 798–816, 2010.

[34] R. Mahouachi, M. Kessentini, and M. Ó. Cinnéide, *Search-Based Refactoring Detection Using Software Metrics Variation*. Springer, 2013, pp. 126–140.

[35] M. W. Mkaouer, M. Kessentini, S. Bechikh, and M. Ó. Cinnéide, *A Robust Multi-objective Approach for Software Refactoring under Uncertainty*. Springer, 2014, pp. 168–183.

[36] I. Bashir and A. L. Goel, *Testing Object-Oriented Software: Life-Cycle Solutions*, 1st ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2000.

[37] R. Binder, *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[38] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on java predicates," in *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4. ACM, 2002, pp. 123–133.

[39] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *Evolutionary Computation, IEEE Transactions on*, vol. 6, no. 2, pp. 182–197, 2002.

[40] E. Zitzler, M. Laumanns, L. Thiele, E. Zitzler, E. Zitzler, L. Thiele, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm," 2001.

[41] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba, "Mocell: A cellular genetic algorithm for multiobjective optimization," *International Journal of Intelligent Systems*, pp. 25–36, 2007.

[42] ——, "Design issues in a multiobjective cellular genetic algorithm," in *Proceeding of the Conference on Evolutionary Multi-Criterion Optimization, volume 4403 of LNCS*. Springer, 2007, pp. 126–140.

[43] M. Fowler, *Refactoring – Improving the Design of Existing Code*, 1^{st} ed. Addison-Wesley, June 1999.

[44] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A. Le Meur, "Decor: A method for the specification and detection of code and design smells,"

Software Engineering, IEEE Transactions on, vol. 36, no. 1, pp. 20–36, 2010.

[45] F. Khomh, S. Vaucher, Y.-G. Gueheneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Quality Software, 2009. QSIC'09. 9th International Conference on*. IEEE, 2009, pp. 305–314.

[46] M. O'Keeffe and M. Ó. Cinnéide, "Search-based refactoring: an empirical study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 5, pp. 345–364, 2008.

[47] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comput. Surv.*, vol. 35, no. 3, pp. 268–308, Sep. 2003.

[48] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*. IEEE, 2002, pp. 97–106.

[49] S. M. Olbrich, D. S. Cruze, and D. I. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.

[50] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 268–278.

[51] "Inner class example," https://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html, accessed: 2015-06-03.

[52] J. J. Durillo and A. J. Nebro, "jmetal: A java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, pp. 760–771, 2011.

[53] A. Jaszkiewicz, "A comparative study of multiple-objective metaheuristics on the bi-objective set covering problem and the pareto memetic algorithm," *Annals of Operations Research*, vol. 131, no. 1-4, pp. 135–158, 2004.

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

[55] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.

[56] Y.-G. Guéhéneuc and G. Antoniol, "Demima: A multi-layered framework for design pattern identification," *Software Engineering, IEEE Transactions on*, vol. 34, no. 35, pp. 667–684, Sep 2008.

[57] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 331–336.

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

[59] B. Beizer, *Software Testing Techniques 2nd edition*. International Thomson Computer Press, 1990.

[60] P. McMinn and M. Holcombe, "Evolutionary testing of state-based programs," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '05. New York, NY, USA: ACM, 2005, pp. 1013–1020. [Online]. Available: http://doi.acm.org/10.1145/1068009.1068182

[61] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Software Eng.*, vol. 39, no. 2, pp. 276–291, 2013.

[62] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE transactions on software engineering*, no. 3, pp. 178–187, 1978.

[63] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Trans. Softw. Eng.*, vol. 30, no. 1, pp. 3–16, Jan. 2004. [Online]. Available: http://dx.doi.org/10.1109/TSE.2004.1265732

[64] M. Harman, "Refactoring as testability transformation," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 414–421. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2011.38