JCaliper: Search-Based Technical Debt Management

Panagiotis Kouros, Theodore Chaikalis, Elvira-Maria Arvanitou, Alexander Chatzigeorgiou,
Apostolos Ampatzoglou, Theodoros Amanatidis

Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
pkouros@uom.edu.gr, chaikalis@uom.gr, earvanitoy@gmail.com, achat@uom.gr, apostolos.ampatzoglou@gmail.com, tamanatidis@uom.edu.gr

ABSTRACT

Technical Debt (TD) reflects problems in software maintainability along evolution. TD principal is defined as the effort required for refactoring an existing system to an ideal one (a.k.a. optimal) that suffers from no maintainability problems. One of the open problems in the TD community is that ideal versions of systems do not exist, and there are no methods in the literature for approaching them, even theoretically. To alleviate this problem, in this paper we propose an efficient TD management strategy, by applying Search-Based Software Engineering techniques. In particular, we focus on one specific aspect of TD, namely inefficient software modularity, by properly assigning behavior and state to classes through search space exploration. At the same time, in the context of TD, we: (a) investigate the use of local search algorithms to obtain a near-optimum solution and propose TD repayment actions (i.e., refactorings), and (b) calculate the distance of a design to the corresponding optimal (i.e., a proxy of TD principal). The approach has been implemented in the JCaliper Eclipse plugin enabling a case study, which validates the approach and contrasts it to existing measure of software evolution.

CCS CONCEPTS

Software and its engineering → Software creation and management → {Software development techniques → *Object-oriented development*, Software verification and validation → *Empirical software validation}*

KEYWORDS

Object-oriented Design, Refactoring, Software Quality

ACM Reference format:
P. Kouros, T. Chaikalis, E. M. Arvanitou, A. Chatzigeorgiou, A. Ampatzoglou, and T. Amanatidis, "JCaliper: Search-Based Technical Debt Management", In *Proceedings of ACM SAC Conference, Limassol, Cyprus, April 8-12, 2019 (SAC'19)*, 10 pages.

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297448

1. INTRODUCTION

Software engineering textbooks consider evolution and maintenance as a fundamental activity, caused by the need to consider changing customer and market requirements [15]. Software evolution and maintenance are hindered when the structural quality of the system is compromised in favor of business benefits or runtime qualities, collectively termed as Technical Debt (TD) [17]. The pillars of TD theory are two concepts borrowed from economics: principal and interest. On the one hand, principal corresponds to the effort that needs to be spent so as to refactor the existing system to an optimal one with respect to structural quality and maintainability [3]. On the other hand, interest corresponds to the additional costs that occur along maintenance, due to poor software quality [2]. One of the main problems for calculating TD principal is that identifying the optimal version of a system is a task that is far from trivial, as it never exists in practice.

To approach this problem, from a theoretical perspective, in this paper we model the quality properties that the development team wants to improve in the form of a fitness function. The approach then seeks to optimize the value of this function, as a typical search space exploration problem, by applying software refactorings[1]. To illustrate the approach we focus on one important type of technical debt, namely architectural technical debt (ATD) [30]. One of the qualities that are compromised and cause ATD is software modularity [20], [30]. According to van Vliet [32], modularity can be assessed by two basic properties, namely: coupling and cohesion. Coupling and cohesion are closely related to the proper allocation of behavior and state into system classes, in the sense that an improper allocation would: (a) violate the single responsibility principle—leading to low cohesion; and (b) increase method invocations to access fields of other classes—leading to high coupling. However, we need to note that the proposed method is quality attribute and property agnostic, since it could be easily applied to qualities other than modularity, and properties other than coupling and cohesion.

Our approach aims at: (a) assessing TD principal (i.e., the distance between the current and optimal design), and (b) proposing a TD repayment strategy (i.e., a sequence of refactorings) to reach it. The distance quantifies the difference in the selected fitness function and reflects the architectural quality of the examined system. The distance also translates to a number of refactorings required to convert the actual system to the corresponding optimum one. This strategy is employed to assess the evolution of entire projects by monitoring changes in the aforementioned distance for successive software versions. The proposed approach

[1] The corresponding research field is collectively referred as Search-Based Software Engineering (SBSE) [15].

has been implemented as an Eclipse plugin that is publicly available. The plugin allows the selection of multiple versions of a project and automatically applies the proposed analysis, for validation purposes.

The rest of the paper is organized as follows: In Section 2 we discuss related work, and in Section 3 we present a motivating example for justifying the need for proper allocation of behavior and state into system classes. Section 4 presents the proposed approach for extracting the optimum design, whereas Section 5 describes the accompanying tool. Section 6 outlines the case study design, whose results are presented in Section 7, and discussed in Section 8. In Section 9 we present the threats to validity, and we conclude the paper is Section 10.

2. RELATED WORK

In this section, we present works related to ours. In Section 2.1, we present techniques that have been developed for managing architectural technical debt, whereas in Section 2.2 we present studies that focus on TD principal quantification. Finally, in Section 2.3 we present the main contributions of this study.

2.1 Architectural TD Management

An approach for the identification and measurement of technical debt in object-oriented systems has been proposed by Marinescu [21]. The proposed method detects specific types of design flaws through object-oriented metrics in four steps: (1) selection of concerned design flaws, (2) definition of rules for detecting the design flaws, (3) measurement of the negative impact of each flaw instance, and, finally, (4) calculation of an overall score based on all detected flaws, to indicate the design quality of a system. The accuracy of the TD measurement in this approach depends on the design flaw detection effectiveness.

An architecture-focused metric that quantifies Technical Debt has been defined by Nord et al [24]. The value of this metric, calculated for each release, is the total cost of the implementation of new architectural elements introduced in this release, and the rework of pre-existing elements in previous releases.

Architectural rework is considered as the necessary adaptation effort for the addition of new architectural elements to an existing software system. The rework cost is calculated based on the analysis of the changing dependencies from existing adapted elements to the new introduced elements. This metric can be used in the calculation of the relative amount of ATD in different software evolution paths, i.e., release plans. Given two release plans RP1 and RP2, that implement the same features and therefore they generate the same amount of business value, the relative amount of ATD is the difference between the values of metric calculated on RP1 and RP2. The proposed metric can facilitate architecture decision-making. However, a main limitation of this approach is the accuracy of the estimation of implementing new features and rework, especially the latter. Each software evolution path involves several releases, which implies that the estimation of rework and new implementations of later releases are based on the estimation of earlier releases. This may pose a threat to the accuracy of architectural technical debt estimation.

2.2 Assessment of TD Principal

The principal of technical debt is related to the effort and accompanying cost to eliminate the debt from a given system or artifact according to Alves et al. [1]. Current software analysis tools offer estimates of TD principal based on detectable violations. According to Curtis et al. [9], three parameters are required for such estimates, (1) the number of violations that should be fixed, (2) the hours that each violation fix requires, and (3) the cost of labor. The SQALE method proposed by Letouzey and Coq [19], introduces the remediation index which is obtained from software quality requirements. For a requirement stating that all files should have at least 70% code coverage, the corresponding remediation action is to write additional tests. A remediation function maps effort to each action, for example, 20 minutes per uncovered line of code. Finally, for each artifact, the remediation index relating to all the characteristics of the Quality Model is obtained by adding all remediation indices linked to all quality requirements. The resulting SQALE index is considered to represent the principal of the TD for the assessed source code.

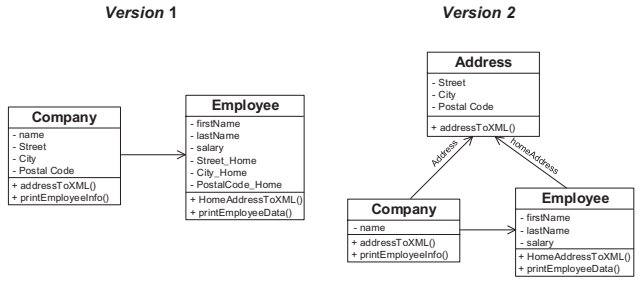
2.3 Contributions

The main contributions of this work compared to the technical debt state-of-research, are the following. First, this work is to the best of our knowledge the ***first that discusses SBSE techniques in the context of technical debt management***. In addition to this, the proposed search-based algorithms are ***performing substantially better*** compared to existing ones, providing the opportunity to apply them in larger systems (our approach is converging for systems of ~250 classes in ~20 minutes, whereas the best existing approaches were able to optimize systems of ~40 classes in 12 hours [25]). Second, this is the first study that ***collectively provides an estimation of TD principal***, based on structural characteristics, ***and the repayment actions*** that need to be completed before the optimal system is reached. Current techniques are mostly based upon rule violations, and the proposed refactorings are resolving these violations, not heavily relying on structural aspects that hinder maintainability, such as coupling, cohesion, and complexity [4].

3. MOTIVATING EXAMPLE

Software metrics have a long history in software engineering as a means of assessing different aspects of software quality. However, metrics are not reliable indicators of quality when comparing different products or even different versions of the same system. This can be better illustrated through an example, shown in Fig. 1, depicting a sample design evolving over two versions. In the initial design of *version-1*, the `Company` class contains a method accessing address related information, whereas employee address information is also contained in the `Employee` class. In the refactored design of *version-2* `address` information is placed on a separate class with the accompanying functionality, while delegate methods have been left in the initial classes to keep the public interface of the original classes intact.

The conventional application of software metrics would lead to the conclusion that the system suffers from software ageing as both examined metrics, namely Coupling (CBO) and Cohesion (LCOM) [8] exhibit worse values in the second version after aggregation at the system level. However, the system in the second version adheres to basic principles in object-oriented design as related data and functions have been grouped together.



* addressToXML() accesses Street, City and PostalCode

printEmployeeInfo() accesses all Employees' data

HomeAddressToXML() accesses Street_Home, City_Home and PostalCode_Home

printEmployeeData() accesses all Employee's data

Figure 1: Assessment of evolution by individual metrics

What is therefore required is a measure that assesses the quality without being subject to this kind of problems. Metrics for which quality cannot be directly deduced from their values but has to be determined on the basis of subjective threshold values are not ideal candidates. Quality can be objectively assessed by measures that evaluate a design against the optimum design that could be achieved for the particular context (i.e. functionality and data) or in other words, against a design that would attain the optimum value for selected metrics. In that way, one could decide whether a particular metric value is '*good*' or '*bad*' based on the best value that this metric can attain for that particular system.

4. PROPOSED METHOD

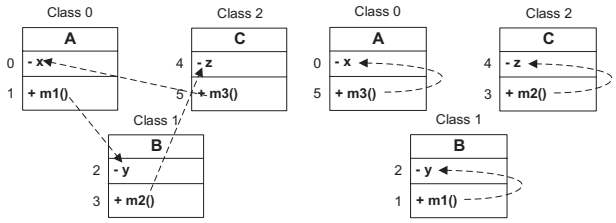
In this section we describe the method that we propose for calculating the distance of a current OO design from an ideal one; and the refactorings that need to be applied for reaching it. To describe the proposed approach in Section 4.1 we present the selected way for representing the system, in Section 4.2 we present the employed fitness function, whereas in Section 4.3 we present the search-based algorithms that we have tried for optimization.

4.1 System Representation

The modular structure of any software system can be represented as a set `S` composed of the constituent modules `M1,M2,..., Mn` [13]. In the case of OO systems, modules correspond to the classes of the design. A significant portion of the design effort is devoted to the identification of the semantics of modules, which is the distribution of responsibilities (state and behavior) among system classes [7]. This constitutes an essential, nontrivial and highly subjective part of the entire design process since it depends heavily on human expertise and experience. The allocation of methods and attributes in an object-oriented system determines a number of qualitative properties such as the comprehensibility, maintainability and reusability of the system and influences directly quantifiable properties such as coupling and cohesion. For example, inappropriate placement of methods and attributes might lead to the violation of several key principles such as Modularity, Separation of Concerns [7], Single Responsibility Principle (SRP) [22] and various associated design heuristics such as the minimization of collaborators and sent messages and the need to keep related data and behavior in one place [26].

The primary goal of the proposed approach can be illustrated by the following simplified example. Let us assume that a given system consists of three classes A, B and C and six entities (attributes and methods). We assume the current allocation of entities as shown in Fig. 2(a). Arrows indicate the attributes assessed by each method, while classes and entities are numbered for easier reference. As it can be observed, the current design suffers from extreme coupling and low cohesion since each method accesses an attribute located in another class and none from the class where it resides. Methods and attributes can be allocated to the existing system classes in various ways, which for the case of a system with x classes and n entities lead to an exploration space of x^n possible allocations. Each of these alternative allocations yields a different design having a different measure of quality. The problem is further complicated, since in order to achieve an optimal design (e.g. in terms of coupling and cohesion), new classes might be required and old classes might have to be removed.

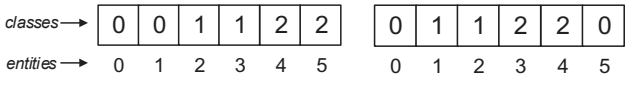
For this particular example, the design that minimizes coupling and maximizes cohesion is the one shown in Fig. 2(b) where each method is placed in the same class with the attribute that it accesses. In other words, the first goal is to address a search space exploration problem in order to identify a solution for the design that optimizes a selected fitness function. Once the optimal design is found, the difference in the value of the fitness function between the current and the optimal design is an indicator of its design quality or TD principal. Moreover, it becomes possible to determine the number of refactorings that have to be made to the current design in order to convert it to the optimal one.



(a) Current Design (b) Optimal Design

Figure 2: Illustrative Example

One intuitive approach that has been employed in the past [5] is to represent the allocation of entities to classes by employing a chromosome encoding. Such a chromosome has genes that correspond to each entity and the value that each gene can take (alleles) specifies the class to which the corresponding entity is placed. For example, the chromosomes corresponding to the current and optimal solutions of Fig. 2 are shown in Fig. 3. Although intuitive, the chromosome representation suffers from some limitations which reduce the performance and quality of solutions. The analysis of previous work reveals that the selection of the representation model is of major importance [10], [26].



(a) current design (b) optimal design

Figure 3: Chromosome Encodings

The proposed approach adopts a different encoding scheme in order to reduce the size of exploration space: the employed representation model for the solutions in the search space does not simply function as a symbolic notation for modeling the solutions during the search. It determines indirectly the most suitable data structures to be used during the implementation and as a consequence it affects drastically the efficiency as well as the quality of the final solutions. When entities are allocated to the system classes, a partition of the set of entities is performed. Thus, for the representation the symbolic notation for set partitions can be employed [16] according to which:

The partition P of a set S with cardinality m is a collection of n subsets S_i of S ($S_i \subseteq S, i = 1..n, 1 \le n \le m$) such as:

1. No subset is empty: $S_i \neq \varnothing, \forall i \in \{1,2,\ldots,n\}$
2. The intersection of any two subsets is empty: $S_i \cap S_j = \varnothing, \forall i \neq j, i, j \in \{1,2,\ldots,n\}$
3. The union of all subsets is the set S: $S_1 \cup S_2 \cup \ldots \cup S_n = S$

The use of multiple brackets in the notation can be avoided. For example, the partitions of set {1, 2, 3} can be written as:

$$1|2|3 \quad 12|3 \quad 13|2 \quad 1|23 \quad 123$$

If entities are represented in this manner, each class corresponds to a unique subset; thus, every design has a unique representation (partition). Moreover, classes are directly identifiable without the need to spend additional programming effort in mapping them to integer values. The latter is of major importance since it allows the reuse of already calculated values of the fitness function.

4.2 Fitness Function

As already mentioned, methods and attributes are the free variables which the designer has to allocate to the individual classes. Due to the countless alternatives, it is important to be able to reason about the quality of each solution. The software engineering community has established various design principles that should be followed [22] or design heuristics that should not be violated [26] when taking design decisions. Many of these principles and heuristics build upon the key concepts of coupling and cohesion signifying their importance in assessing the design quality of a system. Harman and Clark state that any software metric can be employed to guide the search for an optimal design ('*metrics are fitness functions too*' [15]). Indeed a number of previous works in Search Based Software Engineering employed widely used metrics as fitness functions [29]. Some of the proposed metrics attempt to derive 'wise' fitness functions which bring the optimum design closer to the design perceived by designers as best [29].

Among common metrics, coupling is the most widely used one that serves as a fitness function. However, coupling and cohesion are strongly interrelated and there is often a tradeoff among them. Thus, optimizing the one might deteriorate the other. If both concepts could be quantified with the same terms and notation, it would be possible to define a ratio of cohesion over coupling and form a single function expressing both properties, which the designer would seek to maximize. The rest of this section provides an overview of the Entity Placement metric proposed in [31] which serves perfectly the goal of single-objective optimization.

Measure of similarity among system entities. In an OO system, methods can access other entities (attributes and methods) that reside either in the class that they belong to (directly) or in other system classes (through references). Conversely, attributes can be accessed directly from methods of the class that they belong to and also from methods of other classes that have reference to that class. Each system entity e can be characterized by its entity set S_e which contains the attributes and methods that it accesses (in case of a method) or the methods that it is accessed from (in case of an attribute). For each class C it is possible to define its entity set containing all attributes and methods that belong to class C.

The intuitive interpretation of grouping behavior with related data implies that the similarity between an entity and a class should be high when the number of common entities in their entity sets is large. Thus, the similarity between any method or attribute and a class can be obtained employing the Jaccard similarity coefficient between the corresponding entity sets. For two sets A and B the Jaccard similarity coefficient is defined as the cardinality of their intersection divided by the cardinality of their union, while their distance is complementary to the similarity and is obtained by subtracting the Jaccard similarity from 1. In the context of our problem, let e be an entity of the system, C a class of the system and S_x the entity set of entity or class x. The distance between an entity e and a class C can be defined as follows:

Definition 1a. If the entity e does not belong to the class C, the distance is the Jaccard distance of their entity sets:

$$distance(e,C) = 1 - \frac{|S_e \cap S_C|}{|S_e \cup S_C|}, \text{ where } S_C = \bigcup_{e_i \in C} \{e_i\} \quad (1)$$

Definition 1b. If the entity e belongs to the class C, e is not included in the construction of S_C:

$$distance(e,C) = 1 - \frac{|S_e \cap S'_C|}{|S_e \cup S'_C|}, \text{ where } S'_C = \bigcup_{e_i \in C, e_i \neq e} \{e_i\} \quad (2)$$

Global measure of entity placement. In a system adhering to the principle of grouping behavior with related data, the distances of the entities belonging to a class (inner entities) from the class itself should be as small as possible (high cohesion). At the same time the distances of the entities not belonging to a class (outer entities) from that class should be as large as possible (low coupling). This aspect of quality can be quantified by considering for each class the ratio of average inner to average outer entity distances. For each class, the closer this ratio to zero is, the safer it can be concluded that inner entities have correctly been placed inside the class and outer entities to other classes. A formula that provides the above information for a class C is given by [31]:

$$EntityPlacement_C = \frac{Din_C}{Dout_C} = \frac{\dfrac{\sum_{e_i \in C} distance(e_i, C)}{|entities \in C|}}{\dfrac{\sum_{e_j \notin C} distance(e_j, C)}{|entities \notin C|}} \quad (3)$$

e denotes an entity of the system,

Din_C is the inner distance of a class C representing the average distance of the class from all its internal entities, and

$Dout_C$ is the outer distance of a class C representing the average distance of the class from all external entities.

The lower the value of this metric is, the better the placement of entities. In a way, the numerator expresses interclass coupling (which should be minimized) and the denominator expresses intra-class cohesion (which should be maximized). Therefore, Entity Placement can perfectly serve as a fitness function for behavior and state allocation. A global measure of how well entities have been placed in classes can be obtained as follows:

$$EntityPlacement_{System} = \sum_{C_i} \frac{|entities \in C_i|}{|all\ entities|} EntityPlacement_{C_i} \quad (4)$$

Entity Placement is an additively separable function since each term depends only on one class. This feature facilitates its recalculation on system refactoring and makes it most appropriate to be used as a fitness function.

4.3 Implemented Algorithms

Search-Based Software Engineering has exploited a large variety of search algorithms with a particular preference on Genetic Algorithms. Since we have used GAs in a previous work [5], we now opted for the local search algorithms: (a) Hill Climbing [28], (b) Simulated Annealing [28], and (c) Tabu Search [14], [28]. The reason for opting for these rather old but very-well studied algorithms is that they are sufficient for single-objective optimization, whereas for multi-objective optimization newer algorithms such as NSGA-II and MOEA/D should be investigated.

Any search algorithm has a number of parameters that govern its operation and have a large impact on its performance. As a result, systematic parameter tuning is required to maximize the quality of solutions [3]. Since it is not possible to test the performance of the selected algorithms for all possible parameter combinations, we employed the Response Surface Methodology (RSM) [23] to find the best tuning. Response Surface Methodology is a collection of statistical and mathematical techniques used in the creation of a functional relationship between a dependent variable y, and one or more independent (input or process) variables (x_1,x_2,..., x_i). For every configuration we executed each algorithm 30 times and noted the average of the optimum solution [3]. After the completion of data collection for all configurations we have built second order interaction models by using the RSM package of the R programming language. Table 2 presents the parameter values that tune each algorithm to the optimum configuration, as they result from the second order RSM model. We should notice that Hill Climbing does not involve parameters that need configuration and therefore has not been included in the RSM analysis.

Table 2: Parameter setup for the employed algorithms

	Parameters and selected configuration		
Algorithm	**Acc. Ratio**	**Cool. Rate**	**Term. Criterion**
Simulated Annealing	20%	40%	3
	Tabu Tenure		**Term. Criterion**
Tabu Search	5 * sqrt(size)		3,000
Tabu Dynamic	(1..5) * sqrt(size)		3,000

5. PROPOSED TOOL

The proposed approach has been implemented as an Eclipse plugin named JCaliper. The plugin is capable of analyzing an existing Java project, extracting the corresponding TD principal and suggesting a feasible TD repayment strategy (i.e. a list of refactoring steps). The current version of the tool does not proceed with the application of the proposed refactorings. The tool and the corresponding source code are publicly available[2].

All search algorithms have been implemented from scratch and each one is available in the form of an API. Currently, the tool offers the possibility to use Hill Climbing (Steepest-Ascent and First-Choice variants), Simulated Annealing and Tabu Search (with static and dynamic tenure). The tool employs Factory, Prototype, Singleton, Bridge, Composite, Flyweight and Strategy GoF design patterns [12] to increase the extensibility of the approach. JCaliper also examines the feasibility of entity movements among classes. In other words, before a move is considered, the tool investigates whether the corresponding prerequisites are satisfied. There are two types of restrictions: prohibited entity moves and prohibited destinations. Each entity contains an attribute 'movable' which dictates whether the entity can be moved at all. Prohibited destinations are represented as a set of 'Forbidden Classmates' for each entity. The restrictions which are considered in JCaliper are shown in Table 3.

Beyond these restrictions, JCaliper also treats some of the entities as a single unit. For example, accessor methods (getters and setters) are conceptually integrated with a corresponding attribute. Such methods should not be separated from the field that they access. In the implementation of JCaliper, attributes and the associated accessor methods are treated as a single entity which can be moved around the classes. The implementation is based on the Composite design pattern [12] which enables the handling of leaf entities as well as of composite entities in a uniform manner.

Table 3: Restrictions in Entity Movements

Compilation related	
Method to be moved contains super method invocations	immovable method
The target class for a entity move is an interface	interfaces prohibited
Method to be moved is a constructor	immovable method
Behavior related	
Method to be moved is synchronized	immovable method
The source class for a entity move is a superclass	immovable entity
Method to be moved is abstract	immovable method
Entity to be moved belongs to an abstract class	immovable entity
The target class for an entity move is abstract	classes prohibited
Quality related	
The source class does not 1-to-1 relate to target class	classes prohibited

Although so far only Move Method and Move Attribute refactorings have been discussed, JCaliper can suggest also other types of refactorings. For example, if methods and attributes are suggested to be moved to a new (non-existing) class, an Extract Class refactoring is proposed. Similarly, if the search algorithm extracts a solution where entities from several classes are placed in the same partition leaving the original classes empty, an Inline Class refactoring is suggested. With respect to Inheritance, the current implementation prohibits moves along a hierarchy. If this restriction is relaxed, Pull Up and Push Down (method/field) refactorings can be suggested as well.

[2] http://se.uom.gr/index.php/projects/jcaliper/

6. CASE STUDY DESIGN

In this section, we present a case study design with two goals: (a) evaluate the validity of the proposed method, by examining if its application can lead to meaningful refactoring (i.e., TD repayment) opportunities, and (b) investigate if the distance between a given system and its optimum one (i.e., TD principal) can be effectively applied to assess its evolution. We acknowledge the fact that TD is a far more multifaceted phenomenon than just coupling and cohesion; this study serves as an illustration of the method. *We note that the proposed approach can be performed with any fitness function that captures TD aspects, or can drive TD management in a more holistic way*.

Regarding ***goal-a***, we apply the proposed process in 6 OSS systems and calculate the benefit in terms of fitness function, using all algorithms that we have implemented. Regarding ***goal-b***, the change of this distance over successive versions provides a measure of how well the design adheres to established principles. The advantage of this approach lays in the fact that software ageing or improvement is not determined based on metric thresholds, which wouldn't be fair for the evaluation of different versions with different characteristics, but in an objective and reliable manner. In other words, for each version it is estimated where the design '*could have been*' in the best case and the effort to move the current design to the optimal. The study has been designed and reported according to the guidelines of Runeson et al. [27].

6.1 Research Objectives and Research Questions

Given the aforementioned goals, we have set two research questions. The first is related to the efficiency of the used algorithms, and the second with the presence of any trends in the evolution of quality from that particular perspective. As part of this investigation, we check whether these trends (if any) are related to the growth rate of the examined system, since maintaining a quality level might depend on the maintenance effort. The corresponding research questions are formulated as follows:

RQ_1: *Which of the algorithms available in the proposed approach is more efficient for TD repayment?*

RQ_2: *Is there any trend in the evolution of quality expressed by TD principal; and is this trend related to system growth?*

6.2 Case and Units of Analysis

As subjects of our study we have used 6 OSS projects. The selected projects have been chosen based on the following criteria: (a) source code should be publicly available; (b) source code should be written in Java since the developed Eclipse plugin analyzes currently Java code; (c) at least 10 versions of the projects should be available; and (d) projects should have been used in the context of software evolution analysis in other studies as well. Information on the selected projects is outlined in Table 4.

Table 4: Selected projects for Software Evolution Analysis

Project	Description	# Versions
JEdit	Programmer's text editor	21
JFlex	Lexical analyzer generator for Java	14
JFreeChart	Java chart library	31
JHotDraw	Java GUI framework for Graphics	16
JUnit	Framework for repeatable tests.	19
JDeodorant	Code smell identification plugin	10

6.3 Data Collection

The analysis of successive versions employs two measures: (a) the normalized Fitness Value Distance (D), and (b) the normalized Number of Refactorings (NoR). The Fitness Value Distance refers to the absolute difference between the values of fitness function of version `i` and the corresponding optimum system:

$$D_i = \left| f_i - f_i^{opt} \right| \quad (5)$$

where:

f_i is the fitness function value for the system in version i

f_i^{opt} is the fitness function value for the corresponding optimum system in version i.

Since the values of this measure are not normalized, direct comparison among different versions would be misleading. Normalization can be achieved as follows ensuring a range of [0..1]:

$$\hat{D}_i = \frac{D_i}{f_i^{opt}} \quad (6)$$

The *Number of Refactorings* (NoR_i) is the number of required Move Method and Move Field refactoring applications [11] to transform the system in version i to the corresponding optimum system. This measure, although abstract, since the application of refactoring needs to consider various conceptual parameters, is a relative indicator of the amount of effort that needs to be spent on perfecting each version. If optimization in terms of fitness function was the only maintenance goal, it can be considered as an estimate of the technical debt present at each version. Since the number of required refactorings is dependent on the system size, to provide a level of normalization we calculate the normalized *NoR* as follows:

$$\widehat{NoR_i} = \frac{NoR_i}{\#entities} \quad (7)$$

6.4 Data Analysis

To study the aforementioned research questions, we performed statistical analysis including Descriptive Statistics, Trend Test, Slope Estimation and Correlation Analysis, as shown in Table 5.

Table 5: Data Analysis per Research Question

RQs	Variables	Analysis
RQ_1	Distance	Descriptive Statistics
RQ_2	Growth Rate $\hat{D}_i$	Trend Test Slope estimation
	$\widehat{NoR}$	Correlation Analysis

For answering RQ_1, we are executing the method for each algorithm, and capture the distance that each algorithm achieves (i.e., a proxy of TD principal). Given the fact that all algorithms start from the same actual fitness function values, the distance is a measure of the improvement that the application of the algorithm has provided. For this analysis descriptive statistics are provided.

For answering RQ_2, the goal is to examine if there is a trend in the evolution of the two metrics that express quality and if so, to quantify this trend in comparable numbers. To determine if a trend is present in the evolution of a metric we employed linear

regression and the Mann – Kendall trend test. Linear regression is considered a robust modeling tool. However, to consider the results of a trend test based on linear regression as valid, a number of preconditions have to be satisfied, such as that no significant outliers exist, observations be independent, homoscedasticity and normal distribution of residuals. In case the assumptions do not hold, a nonparametric test which can provide reliable results is the Mann – Kendall trend test. When according to the Mann – Kendall test a trend is clearly evident, i.e. the null hypothesis can be rejected, the Theil – Sen Estimator was used to calculate the slope of the fitted trend-line. The slope obtained by the Theil – Sen Estimator is the median slope among all lines through all pairs of points in the dataset. To enable the comparison of slope steepness among projects, slopes should be scale independent. To this end, we performed the trend test analysis (either linear regression or Mann – Kendall trend test) on a normalized version of the original dataset. In particular, each value of an examined time series was divided by the maximum value in the time series yielding a normalized value in the range [0..1] exhibiting the same slope as the original dataset. Additionally, as part of answering this research question, we examine if there is a correlation between TD principal (as captured by the distance between the actual and the optimum design) and the growth rate. A high, positive and statistically significant coefficient implies that the evolution of the distance between the actual and the optimum design follows the trend of the growth rate. In other words, when the system grows in size the quality deteriorates.

7. RESULTS

Efficiency of algorithms (RQ$_1$). In RQ$_1$ we explored which of the offered algorithms is more efficient, both in terms of TD repayment (i.e., the achieved quality of solutions as measured by the decrease in the distance between the obtained solution and the corresponding optimum one) as well as in terms of the required execution time. The termination criteria for the algorithms are as follow: (a) *Hill Climbing*—reach a local optimum; (b) *Tabu Search*—no improvement last 3,000 iterations; and (c) *Simulated Annealing*—no improvement after 3 levels.

Table 6: Quality of solutions for the employed algorithms

Project	SA	TS_DYN	TS	HC_FC	HC_ST
JDeodorant 05	15.5%	16.2%	15.6%	11.8%	12.5%
JDeodorant 07	14.9%	14.1%	14.2%	11.6%	11.9%
JDeodorant 10	13.7%	12.7%	12.6%	9.5%	9.3%
JEdit 3.0	14.9%	14.9%	14.2%	10.4%	10.3%
JEdit 4.0	14.3%	15.4%	13.4%	9.8%	9.6%
JFlex 1.3	13.4%	12.6%	13.3%	9.6%	9.6%
JFlex 1.3.5	14.1%	13.0%	16.1%	9.6%	9.3%
JFlex 1.4.3	16.8%	15.5%	16.0%	12.6%	12.2%
JFreechart 0.8.0	8.5%	8.3%	6.7%	3.9%	3.9%
JFreechart 1.0.0	7.8%	7.9%	7.9%	4.7%	4.7%
JHotdraw 5.2	9.3%	8.6%	8.8%	6.0%	6.0%
JUnit 3.4	17.7%	17.0%	17.4%	10.2%	10.3%
AVG Improvement	*14%*	*13%*	*13%*	*8%*	*9%*

SA: Simulated Annealing, **TS**: Tabu Search, **TS_DYN**: Tabu Search - dynamic tenure, **HC_FC**: Hill Climbing - First-Choice, **HC_ST**: Hill Climbing - Steepest-Ascent

Regarding efficient TD repayment, we examine the optimum fitness function value for each of the algorithms. For the evaluation of efficiency we have compared the improvement that each algorithm offered to each problem. The results are shown in Table 6. The percentage of improvement is extracted from the values of the fitness function. The cell corresponding to the algorithm offering the maximum improvement for each project is shaded. Overall results are shown in the bottom row of Table 6. We note that since the results of this study are based on the optimization of only coupling and cohesion, they only map to one viewpoint of the total TD, i.e., the one related to modularity [30]. Nevertheless, according to Skiada et al. [30], modularity metrics are an accurate assessor of the total TD, as expressed by SonarQube.

In terms of quality of the solutions, it can be observed that Simulated Annealing followed by Dynamic Tabu Search offer the best results. Moreover, Hill Climbing variants are insufficient in terms of the obtained quality of solutions. To perform a systematic comparison we conducted a Wilcoxon signed-rank test. The mean difference of the obtained improvement over the actual system (initial state) is shown in Table 7 for each pair of algorithms. For example, the top-left cell indicates that Simulated Annealing offers on average 0.49% larger improvement than the Tabu Search, which according to Wilcoxon signed-rank test is statistically significant ($Z = -2.062$, $p<0.05$). Based on these results, Simulated Annealing consistently achieves a better improvement than its competitors. However, the benefit of using Simulated Annealing over Tabu Search is rather limited considering the significantly larger execution time as it will be shown next.

Table 7: Pairwise comparisons between algorithms

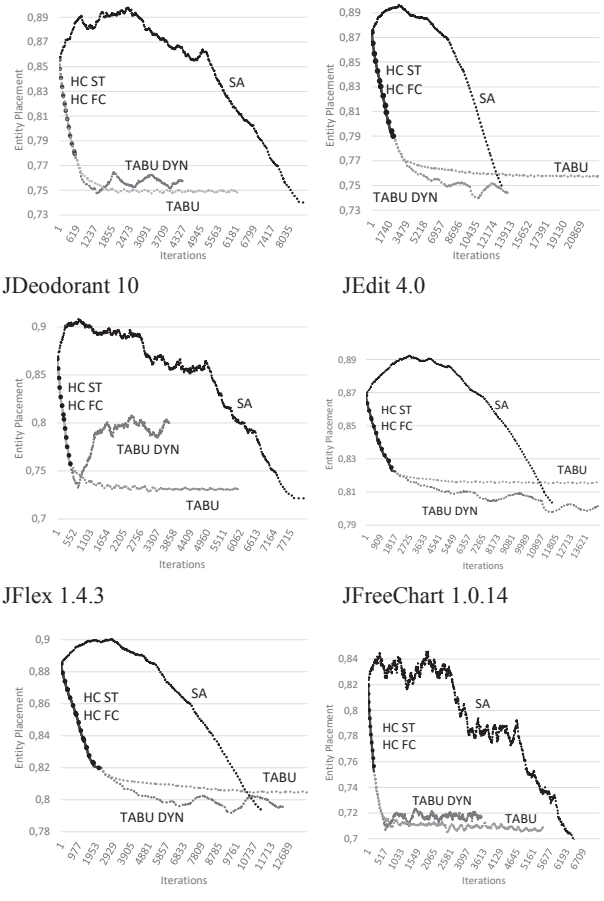
		TS	TS_DYN	HC	HC_ST
SA	**Mean Diff**	0.49%	0.50%	5.49%	4.51%
	Z	-2.062*	-2.132*	-3.18**	-3.18**
TS	**Mean Diff**		0.01%	4.99%	4.01%
	Z		-0.078	-3.18**	-3.18**
TS_DYN	**Mean Diff**			4.98%	4.00%
	Z			-3.18**	-3.18**
HC	**Mean Diff**				-0.98%
	Z				-.035

Z = Wilcoxon test Z-statistic, * $p < 0.05$ ** $p < 0.01$

To assess the performance of the proposed approach and enable a comparison among the algorithms one could measure the required execution time (in secs). However, given that the total execution time depends heavily on the number of iterations, which in many cases are performed without offering any further improvement, execution times might be misleading. For example, Tabu Search might reach an optimum solution at a particular time point t and then iterate until it satisfies the set termination criteria at time point `t+k`. The additional elapsed time k appears as time spent to find a solution, whereas it is redundant in the sense that with a different termination criterion the algorithm could have stopped earlier. To avoid such pitfalls in the interpretation of execution time we opted for the more appropriate convergence plots, which show the achieved fitness value (Entity Placement) over successive iterations of each algorithm. These plots apart from indicating which algorithm finds the best solution, also highlight how early algorithms reach their best solutions. The convergence plots for one version of each examined project are shown in Fig. 4.

As it can be observed, in all projects the dynamic version of Tabu Search achieves results that are almost as good as the results ob-

tained by Simulated Annealing (and in some cases even better), but converges much more rapidly. It should be noted that the termination criterion for Tabu Search is a number of iterations which do not yield any improvement in the quality of solutions. Therefore, they stop a specified number of iterations after the time point at which they reach an optimum value. This is the reason for which execution continues beyond a local optimum. On the other hand, Hill Climbing terminates whenever it reaches a local optimum or in other words, when it starts moving to states of lower quality. As it becomes evident it is almost always stuck in local optima. Simulated Annealing terminates when it does not observe any improvement in a number of consecutive temperature levels, leading to a very good exploration of the solutions landscape. The overshoot in the curves for Simulated Annealing (i.e. the moves towards worse solutions) is related to the fact that the initial state is not a random one, as it would be the case in other problems, but the actual, existing design of the software system.



JDeodorant 10 JEdit 4.0

JFlex 1.4.3 JFreeChart 1.0.14

JHotDraw 7.2 JUnit 4.10

Figure 4: Convergence of Examined Algorithms

The fact that the initial state for the search process is a design that reflects, despite its potential inefficiencies, the experience of the developers, is also the reason for which Simulated Annealing and Tabu Search achieve comparable results. If the search process had started from random allocations of entities to classes, Simulated Annealing would probably exhibit superior performance. The reason is that Simulated Annealing, in its early stages, operates as a "random walk" and therefore might benefit to a very small extent from a favorable starting point. Simulated Annealing would achieve similar results even if we had started from a different initial design. On the contrary, Tabu Search, despite its 'explorative' capabilities offered by a large tenure, is constrained to the exploration of the area around the starting point. Thus, the results achieved by Tabu Search depend heavily on the initial design.

It is worth mentioning that the dynamic version of Tabu, especially for small systems such as JFlex and JUnit, exhibits two patterns: a) a number of ripples which can be attributed to the existence of a list of prohibited moves which often means that the algorithm attempts to reach the same local optimum but from different trajectories and b) a departure from a local optimum towards worse solutions which can be attributed to the tenure. That is, entities which have to be further relocated, are not allowed to be moved and thus the algorithm moves well-placed entities, leading to designs of inferior quality.

Considering that the Dynamic Tabu search: (a) yields solutions which are nearly as good as the optimum solutions provided by Simulated Annealing and (b) converges more rapidly than Simulated Annealing; for the case study on software evolution analysis we have selected as a rational choice the Dynamic Tabu Search.

Evolution Trends (RQ$_2$). To investigate whether a trend exists in the evolution of the selected variables we performed the nonparametric trend test (Mann-Kendall). For slope values we report the Theil-Sen estimator and its significance in Table 8. Based on the fact that a single trend is not expected across the whole project evolution, it wouldn't make sense to attempt to extract a single trend for the entire evolution. Thus, we list slopes for each distinct period along project evolution.

Table 8: Trend Test Results

Project	Versions From To	*Slope*		
		Growth	$\hat{D}_i$	$\widehat{NoR}$
JEdit	2.3 – 4.2	0.07**	0.005**	0.005
	4.3 – 5.0	0.003**	0.005**	0.003**
JHotDraw	5.2 – 6.0	0.202	0.038	0.004
	7.0.6 – 7.6	0.067**	0.000	0.001
JUnit	3.4 – 3.8.1	0.033**	0.010	0.021
	4.0 – 4.10	0.053**	0.018	0.005
JFreeChart	0.5.6 – 0.9.20	0.07**	0.002	0.007**
	0.9.21 – 1.0.14	0.021**	0.03**	0.000
JFlex	1.3 – 1.4_pre3	0.15**	0.008	0.005**
	1.4_pre 4 – 1.4.3	0.007	0.11*	0.007
JDeodorant	1(001) – 5(232)	0.15**	0.055*	0.061*
	6(244) – 10(343)	0.07*	0.03	0.011*

* Correlation is significant at the 0.05 level
** Correlation is significant at the 0.01 level

From Table 8 it can be concluded that the growth rate exhibits always a statistically significant trend. This is reasonable since the growth rate reflects the constant evolution in alignment to the sixth law of Lehman which stipulates that "*the size of a system continuously grows over time*" [18]. On the other hand, for TD principal and the number of refactorings (i.e., discrete TD items repayment), clear trends can be observed in particular periods of evolution in some of the projects only. For 4 projects there is a statistically significant trend for at least one of the two variables and at least of the two periods. For the case where a trend is present, it adheres to the previously-made observations.

Table 9: Correlation Analysis

Project	Versions From To	Correlations	
		Growth vs D	Growth vs NOR
JEdit	2.3 – 4.2	0.849**	0.656*
	4.3 – 5.0	0.836**	0.695*
JHotDraw	5.2 – 6.0	0.975*	0.598
	7.0.6 – 7.6	0.199	0.467
JUnit	3.4 – 3.8.1	0.223	0.38
	4.0 – 4.10	0.895**	0.712**
JFreeChart	0.5.6 – 0.9.20	0.138	0.726**
	0.9.21 – 1.0.14	0.817**	0.239
JFlex	1.3 – 1.4_pre3	0.989**	0.981**
	1.4_pre1.4.3	0.641	0.932**
JDeodorant	001232	0.944**	0.991**
	244343	0.949**	0.855*

* Correlation is significant at the 0.05 level ** Correlation is significant at the 0.01 level

To study whether the evolution of quality is correlated to the evolution of the growth rate, we performed correlation analysis (see Table 9). Table 9 validates the aforementioned remarks. In general there is a statistically significant correlation between growth rate and TD aspects (i.e., principal and number of TD item repayment) in 67% of the cases. For projects JEdit, JFreeChart and JDeodorant the evolution of quality as expressed by both variables, deteriorates in the period of faster growth rate as expressed by positive correlation coefficients. On the other hand, quality improves in the period of weak or moderate growth rate. The same observation holds for JFlex with the exception that in the period of weak growth the distance measure does not have a negative correlation to the growth rate, but a lower one compared to the period of fast growth. For JHotDraw and JUnit this evolution pattern is not statistically verified, confirming the exceptional development practices for these projects.

Considering the aforementioned results, we claim that TD measures, i.e., TD principal and number of TD repayment actions, as obtained by JCaliper are correlated to the growth rate of the system. Therefore, they obey to the corresponding law of software evolution.

8. DISCUSSION

In this section we discuss the main findings of this paper. First, by answering RQ_1, we can claim that the proposed approach and tool can lead to an efficient TD repayment strategy, in the sense that they are able to propose a series of refactorings that reduced TD principal. Although for this study as a proxy of TD, only software modularity has been considered (this is an obvious threat to validity—see Section 9), we note that in many studies (e.g., [20] and [30]) it has been observed that TD is highly correlated with modularity metrics. Additionally, by answering RQ_2, we have validated that TD aspects (i.e., principal and number of refactorings) as quantified in this study are correlated to system growth and therefore obey in software evolution laws. By focusing on a project-by-project analysis, interesting discussions have arisen:

- For *JEdit*, *JFreeChart*, *JFlex*, and *JDeodorant* periods of rapid development exhibit increased rates of software ageing. For these projects, the period with the steepest increase in the number of entities or classes exhibits a deterioration of quality (reflected in an increasing TD principal, as well as an increasing number of required refactorings. Furthermore, the initial period is the one in which a decreasing quality can be observed: these systems appear to have a less mature and fast changing initial phase, while after years of development it becomes possible to achieve a stabilization or even improvement of quality. During one particular transition from one release to another one (which might last for several months or years) the value of the fitness function is improved (actual design moves closer to the optimum one). Although further research is required to investigate the cause of this improvement, the abrupt change in the TD principal could be attributed to refactoring application or architectural re-design.
- A noteworthy exception is project *JHotDraw* which does not exhibit ageing phenomena even when development is performed at a fast rate. Given that JHotDraw is known for the wide adoption of design patterns and application of design principles this might be an indication that proper software engineering can prevent software ageing.
- A second exception is project *JUnit* where the initial period of development exhibits a faster rate of design quality degradation compared to the second period, despite the fact that in the initial period the rate of system size increase is somehow lower. This could be attributed to the initial turbulence in the system architecture.

The findings presented in this study can be useful to both researchers and practitioners. On the one hand, ***practitioners*** are suggested to use the proposed tool: (a) as a refactoring-support tool if they are interested in optimizing software modularity, and (b) as a proxy of the TD introduced into their systems. We believe that the nature of the tool (i.e., an Eclipse plug-in) can substantially boost its applicability in practice. On the other hand, ***researchers*** are provided with some interesting implications and future work opportunities. First, they are provided with a tool for efficient technical debt management, which they need to further validate in an industrial setting. Second, since the study validates the appropriateness of SBSE in technical debt management, we suggest researchers to further explore this research direction. Finally, we suggest the adoption of the high-level rationale of the proposed approach with different fitness functions that cover TD management in a more holistic manner.

9. THREATS TO VALIDITY - LIMITATIONS

The proposed approach can be employed to assess the evolution of quality over successive software versions. However, it should be stressed that the notion of quality is restricted to the particular metric that is used as fitness function in the applied search algorithms. For example, the Entity Placement metric that has been used for the case study reflects only the decisions in the design related to coupling and cohesion and unavoidably overlooks other aspects of quality which might be of interest. Nevertheless, other metrics can also be investigated either in the context of separate analyses or by attempting a multi-objective optimization. In any case, it should be borne in mind that metrics-based assessment of design properties captures only specific aspects of quality and can never entirely substitute expert judgment and experience. This has been stressed by works that study the ability of automated refactoring suggestions (see Bavota et al. [6]).

With respect to the empirical application, the most obvious threat is the one to the external validity of the conclusions. Unavoidably, any observations which have been made regarding the relation of the growth rate and the evolution of quality reflect the tendencies in these particular projects. With respect to the application of search algorithms to derive the optimum design, an internal threat to validity stems from the fact that parameter settings for the configuration of each algorithm affect its performance [33]. However, as already pointed out, the main goal of the approach is not to compare local search algorithms in terms of their efficiency. Moreover, we have applied RSM to fine-tune algorithms.

Beyond these threats, restructuring an object-oriented system by means of optimization should consider side effects. For example, the proposed approach does not address changes that should be carried out in the accompanying documentation (e.g. traceability matrices) or unit tests. Obviously, changes in the public interfaces of system classes might render design documents, code comments and test cases invalid posing very interesting research challenges.

Finally, we should stress that such types of optimization techniques are inherently limited since only design artifacts are considered as parameters of the optimization. Developers are aware of the fact that software architectures reflect people's choices, styles and constraints and rearranging classes and methods might break such conceptual assumptions. Therefore, we should bear in mind that automated search-space optimization for software improvement can only yield suggestions to the development team.

10. CONCLUSIONS

The problem of optimizing an OO design can be efficiently treated as a search-space optimization task. In this paper we employed SBSE as a means of assessing TD principal (i.e., the distance between the actual design and an optimum as derived by search-space optimization) and proposing a set of refactorings (i.e., a TD repayment strategy) to reach it. To facilitate the analysis of large systems several optimizations have been applied on top of well-known search algorithms. The application on 6 OSS systems revealed that there is a correlation between the growth rate and the evolution of quality. In general, whenever the number of entities and classes increases at a fast pace, quality degradation can be observed. However, often design teams manage to add functionality at a fast pace without exhibiting signs of software ageing.

ACKNOWLEDGMENTS

Work reported in this paper has received funding from the European Union Horizon 2020 research and innovation programme under grant agreement No. 780572 (project: SDK4ED).

REFERENCES

[1] N. S. R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Inf.Softw.Technol.*, vol.70, pp.100–121, Feb. 2016.

[2] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, P. Abrahamsson, A. Martini, U. Zdun, and K. Systa, "The Perception of Technical Debt in the Embedded Systems Domain: An Industrial Case Study," in *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, Raleigh, NC, USA, 2016, pp. 9–16.

[3] A. Arcuri and G. Fraser, "On Parameter Tuning in Search Based Software Engineering," in *Search Based Software Engineering*, M. B. Cohen and M. Ó. Cinnéide, Eds. Springer Berlin Heidelberg, 2011, pp. 33–47.

[4] E. M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, M. Galster, and P. Avgeriou, "A mapping study on design-time quality attributes and metrics," *J. Syst. Softw.*, vol. 127, pp. 52–77, May 2017.

[5] M. Basdavanos and A. Chatzigeorgiou, "Placement of Entities in Object-Oriented Systems by Means of a Single-Objective Genetic Algorithm," in *2010 Fifth International Conference on Software Engineering Advances*, Nice, France, 2010, pp. 70–75.

[6] G. Bavota, F. Carnevale, A. D. Lucia, M. D. Penta, and R. Oliveto, "Putting the Developer in-the-Loop: An Interactive GA for Software Re-modularization," in *Search Based Software Engineering*, G. Fraser and J. T. de Souza, Eds. Springer Berlin Heidelberg, 2012, pp. 75–89.

[7] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen, and K. A. Houston, *Object-Oriented Analysis and Design with Applications*, 3 edition. Upper Saddle River, NJ: Addison-Wesley Professional, 2007.

[8] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[9] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the principal of an application's technical debt," *IEEE Softw.*, no. 6, pp. 34–42, 2012.

[10] E. Falkenauer, *Genetic Algorithms and Grouping Problems*, 1 edition. Chichester ; New York: Wiley, 1998.

[11] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, and E. Gamma, *Refactoring: Improving the Design of Existing Code*, 1 edition. Reading, MA: Addison-Wesley Professional, 1999.

[12] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 edition. Reading, Mass: Addison-Wesley Professional, 1994.

[13] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of software engineering*. Upper Saddle River, N.J.: Prentice Hall, 2003.

[14] F. Glover and M. Laguna, "Tabu Search," in *Handbook of Combinatorial Optimization: Volume1–3*, D.-Z. Du and P. M. Pardalos, Eds. Boston, MA: Springer US, 1999, pp. 2093–2229.

[15] M. Harman and J. Clark, "Metrics Are Fitness Functions Too," in *Proceedings of the Software Metrics, 10th International Symposium*, USA, 2004.

[16] D. E. Knuth, *The Art of Computer Programming*, 1 edition. Amsterdam: Addison-Wesley Professional, 2011.

[17] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: from metaphor to theory and practice," *Ieee Softw.*, no. 6, pp. 18–21, 2012.

[18] M. M. Lehman, "Laws of software evolution revisited," in *Software Process Technology*, 1996, pp. 108–124.

[19] J. Letouzey and T. Coq, "The SQALE Analysis Model: An Analysis Model Compliant with the Representation Condition for Assessing the Quality of Software Source Code," in *2010 Second International Conference on Advances in System Testing and Validation Lifecycle*, 2010, pp. 43–48.

[20] Z. Li, P. Liang, P. Avgeriou, N. Guelfi, and A. Ampatzoglou, "An empirical investigation of modularity metrics for indicating architectural technical debt," in *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures*, 2014, pp. 119–128.

[21] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM J. Res. Dev.*, vol. 56, no. 5, pp. 9–1, 2012.

[22] R. C. Martin, *Agile software development: principles, patterns, and practices*. Upper Saddle River, N.J.: Prentice Hall, 2003.

[23] R. H. Myers, D. C. Montgomery, and C. M. Anderson-Cook, *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. John Wiley & Sons, 2009.

[24] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, 2012, pp. 91–100.

[25] M. O'Keeffe and M. Ó. Cinnéide, "Search-based refactoring: an empirical study," *J. Softw.Maint.Evol.Res.Pract*, vol.20, no.5, pp.345–364, Sep. 2008.

[26] A. J. Riel, *Object-Oriented Design Heuristics*, 1 edition. Reading, Mass.: Addison-Wesley Professional, 1996.

[27] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*, 1 edition. Wiley, 2012.

[28] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3 edition. Upper Saddle River: Pearson, 2009.

[29] C. L. Simons and I. C. Parmee, "Elegant Object-Oriented Software Design via Interactive, Evolutionary Computation," *IEEE Trans. Syst. Man Cybern. Part C Appl. Rev.*, vol. 42, no. 6, pp. 1797–1805, Nov. 2012.

[30] P. Skiada, A. Ampatzoglou, E. M. Arvanitou, A. Chatzigeorgiou, and I. Stamelos, "Exploring the Relationship between Software Modularity and Technical Debt," in *4th Conference on Software Engineering and Advanced Applications (SEAA)* 2018.

[31] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," *IEEE Trans Softw Eng*, vol. 35, May 2009.

[32] H. van Vliet, *Software Engineering: Principles and Practice*, 3 edition. Chichester, England ; Hoboken, NJ: Wiley, 2008.

[33] S. Wang, S. Ali, and A. Gotlieb, "Random-Weighted Search-Based Multi-objective Optimization Revisited," in *Search-Based Software Engineering*, 2014, pp. 199–214.