12-2019

Finding needles in a haystack: Leveraging co-change dependencies to recommend refactorings

Marcos César DE OLIVEIRA
University of Brasilia

Davi FREITAS
University of Brasilia

Rodrigo BONIFACIO
University of Brasilia

Gustavo PINTO
Federal University of Parana, Brazil

David LO
Singapore Management University, davidlo@smu.edu.sg

Published in Journal of Systems and Software, 2019, Volume 158, article no. 110420.
https://doi.org/10.1016/j.jss.2019.110420

Finding Needles in a Haystack: Leveraging Co-change Dependencies to Recommend Refactorings

Marcos César de Oliveira[a], Davi Freitas[a], Rodrigo Bonifácio[a,b], Gustavo Pinto[c], David Lo[d]

[a] *Computer Science Department, University of Brasília, Brasília, Brazil*
[b] *Paderborn University, Germany*
[c] *Faculty of Computing, Federal University of Pará, Belém, Brazil*
[d] *School of Information Systems, Singapore Management University, Singapore*

Abstract

A fine-grained co-change dependency arises when two fine-grained source-code entities, e.g., a method, change frequently together. This kind of dependency is relevant when considering remodularization efforts (e.g., to keep methods that change together in the same class). However, existing approaches for recommending refactorings that change software decomposition (such as a move method) do not explore the use of fine-grained co-change dependencies. In this paper we present a novel approach for recommending move method and move field refactorings, which removes co-change dependencies and *evolutionary smells*, a particular type of dependency that arise when fine-grained entities that belong to different classes frequently change together. First we evaluate our approach using 49 open-source Java projects, finding 610 evolutionary smells. Our approach automatically computes 56 refactoring recommendations that remove these evolutionary smells, without introducing new static dependencies. We also evaluate our approach by submitting pull-requests with the recommendations of our technique, in the context of one large and two medium size proprietary Java systems. Quantitative results show that our approach outperforms existing approaches for recommending refactorings when dealing with co-change dependencies. Qualitative results show that our approach is promising, not only for recommending refactorings but also to reveal opportunities of design improvements.

Keywords: Refactoring, co-change dependencies, remodularization, software clustering, design quality

1. Introduction

A modular software design should support the incremental development of a system, and thus enabling seamless changes that often occur during a software life cycle [1]. However, it is a non-trivial effort to maintain the characteristics

of a design throughout its evolution [2]. In practice, software design tends to decay over time—independently of how elaborate the design of the software is [3]. This challenge occurs due to different reasons, including (1) the lack of knowledge of the current development team about the original design decisions of the software [2]; (2) tight schedules that lead developers to take bad decisions, introducing technical debt and hindering the redesign of a software [4]; or (3) unanticipated requirements that do not fit in the original decomposition [3]. In the end, the lack of maintainability often leads to the problem of *software erosion* [5], which occurs when the current design of a software does not reflect the idealized design anymore. As a result, developers might have a harder time to either introduce new features or to fix existing bugs [2, 5].

In order to ameliorate this problem, software engineers and architects might improve the current modular decomposition of the systems by means of *sequences of refactorings*, such as move method/field and split/merge classes. For this reason, several approaches have been proposed to either identify a software decomposition that best fit the needs of a remodularization effort [6, 7] or to suggest sequences of refactorings to improve the design of a software [8]. Existing approaches rely on *source-code dependencies* to recommend alternate decompositions [6, 7], considering a set of specific goals (such as minimizing coupling or maximizing cohesion) [9, 10]. The challenge here is that each time we change a system, e.g., to fix a bug or to introduce a new feature, we can change a set of source-code entities (classes, interfaces, methods, fields) that are not *statically dependent* (that is, they do not call methods or access fields from each other). This situation leads to a different notion of coupling based on *co-change dependencies*, that are not explicit at the source code [11].

Several studies [12, 13, 14, 15] correlate co-change coupling with software quality problems. For instance, Zhou et al. [12] claim that co-change dependency analysis has the potential to provide early warnings of a potential design or architectural flaw. Other authors report that co-change dependencies induce defects [13, 14, 15]. Therefore, presenting refactoring recommendations, which aim to reduce co-change dependencies, might improve the overall quality of a system. Accordingly, recent works [11, 16, 17] explore the use of source-code history to enrich the analysis of existing software modular decompositions, recommending alternative decompositions that better fit the software evolution history. The rationale is that, if a set of source-code entities frequently change together, some opportunities to move source-code entities arise, in order to keep co-changing entities together in the same class (if the entities are methods or fields), or in the same package (if the entities are classes or interfaces).

Despite recent efforts, little is known about the benefits of using **fine-grained** co-change dependencies when suggesting move method/field refactorings, which aim to improve the design of a software when considering properties such as coupling and cohesion. Fine-grained co-change dependency analysis helps to find the sets of fine-grained entities (such as methods or fields) that change together. That is, a co-change dependency between two source code entities means that they had frequently changed together. In addition, if the entities do not have any static dependency upon each other, the existing depen-

dency between them is hidden—and we can only reveal this dependency using a co-change dependency analysis.

In this paper we present and evaluate a novel refactoring recommendation approach (named Draco) that removes co-change dependencies between classes. The *motivation* of this tool is to help to improve the software design, reducing the co-change coupling of its entities, in the sense that this kind of coupling might be correlated with design problems and defects, as seen before. One of its relevant properties is that it does not recommend refactorings that introduce new static dependencies between classes. Our interest is to detect and remove *evolutionary smells*, which arise when methods or fields from different classes are co-change dependent from each other, although they do not depend upon methods or fields from the same class where they are declared. Draco recommends move method/field refactorings that remove evolutionary smells, by *breaking* co-change dependencies (and possibly static dependencies too) between classes. In addition, differently from related work, Draco only recommends refactorings that present some guarantees that lead to an improvement on the design quality. In this way, our approach is quite conservative: it only applies a refactoring when the transformation does not introduce new dependencies into the software.

We used a multi-method approach for evaluating Draco. We first conducted a quantitative assessment using 49 well-known, well-used, and non-trivial open-source Java projects. We found a total of 610 evolutionary smells in all projects but one. We automatically computed 56 recommendations of move method/field refactorings. All the recommendations lead to design improvements (according to well-known metrics), without introducing any new static dependency. After that, we conducted a qualitative assessment of Draco considering three Java enterprise systems, two from the Brazilian Army and one from the Brazilian Ministry of Economy. We also assessed other three refactoring recommendation tools using the two Brazilian Army Systems. In this second assessment, we go beyond the typical evaluation of these tools (i.e., using metrics based on source code dependencies), trying to identify the relevance of refactoring recommendations tools by means of pull-requests submitted to the software projects. Altogether, the main contributions of this paper are the following:

- A method for recommending move method/field refactorings that removes "evolutionary smells" and improves design quality by reducing coupling in terms of co-change dependencies.
- An extensive evaluation on over 49 non-trivial open-source projects showing the benefits of the proposed approach. We also compared our our approach with a state of the art method for refactoring recommendation [10].
- A qualitative assessment of the application of Draco tool considering two proprietary software systems from the Brazilian Army and one proprietary system from the Brazilian Ministry of Economy. In this qualitative assessment we also investigate the outcomes of the application of three different refactoring recommendations tools (REsolution [10], JDeodorant [9], and JMove [18]), considering the two Brazilian Army software systems. Based

on this assessment, we concluded that our approach is promising and reveals design problems of the systems.

- A publicly available tool and dataset that allows the reproduction of this study and that might be useful for researchers and practitioners alike.

We argue that some design decisions of a system (including architectural constraints) should also be considered by refactoring recommendation approaches—besides the typical information used for recommending refactorings (such as static dependencies, semantic dependencies, or even co-change dependencies). Not considering these decisions might actually hinder the acceptance of recommendations from existing approaches.

2. Background and Related work

2.1. Software Decomposition and Remodularization

The concept of a software decomposition we use in this work is based on the definition of Mitchell and Mancoridis [6], in which a software is represented as a graph—typically named a *Module Dependency Graph* (MDG). The vertices of an MDG represent source-code entities and the edges represent some kind of dependency between these entities, such as method calls, field access, or class inheritance. Thus, a software decomposition can be understood as a graph partitioning problem, where a partition is a set of clusters of source-code entities. The work from Ball et al. [19] was one of the first to propose the representation of co-change dependencies as edges on an MDG, though only considering coarse-grained entities (e.g., classes or files) as vertices. Later, Zimmermann et al. [20] introduced the use of fine-grained co-change dependencies on MDGs. Building on these previous works, in this paper, we work with MDGs whose vertices are fine-grained entities (similar to the Zimmermann et al. approach [20]) and whose edges represent both static and co-change dependencies of the software. We also leverage the use of co-change clusters as a process for partitioning a co-change graph—as Beyer and Noack suggest [21].

The use of software clustering as basis for software remodularization has been discussed in the literature for almost 20 years [22]. The work of Anquetil and Lethbridge [23], for instance, compares different strategies for software clustering to this specific goal. More recently, Maqbool and Babri [24] investigate the use of hierarchical clustering algorithms for architecture recovering. Differently from our work, the previous mentioned works only consider source-code static dependencies as input for building software clusters. Silva et al. [7] estimates software modularity using co-change clusters, and compares the resulting decomposition with the actual Java software package organization. According to their work, mismatches between the co-change clusters and the package decomposition suggest new directions for restructuring the package hierarchy.

In a recent work, Candela et al. [25] investigated which properties developers consider relevant for a high-quality software remodularization. Their goal was to provide insights on the design of techniques and tools to recommend

new software decompositions. After collecting responses from a survey with 29 developers, they reported that 52% of them consider the clear separation between application layers important, 38% consider package cohesion important, 28% consider low coupling important, and 21% consider grouping entities that change together important. This result suggests the relevance of considering co-changing when supporting software remodularization—as in a previous work of Beck and Diehl [26].

Also regarding co-change dependencies, Oliveira et al.[11] discuss that adding co-change dependencies to a coarse-grained MDG, based on static dependencies, reveals several dependencies that were hidden by the assessments considering only static dependencies. This result suggests that co-change dependencies cannot be neglected when reasoning about the decomposition of a system. The authors also report about the benefits they achieved after using *coarse-grained* co-change dependencies as input to suggest a software decomposition improvement, which tries to preserve almost the same number of modules (packages) of the original decomposition.

2.2. Code Smells and Source Code Refactoring

Code smell (or bad smell) is a symptom of bad decisions about the system design [27]. Research works discuss that code smells could hinder maintainability and increase fault-proneness [28], increasing the motivation to develop methods to detect and remove bad smells using program refactorings. Fowler [27] describes 22 code smells and the respective refactoring operations to remove them. Several approaches were proposed to detect bad smells in source code [29, 30, 31, 32]. Like our work, Ratiu et al. [33] and Palomba et al. [34] also recommend the use of the source code history to detect code smells, however, the aim of their approaches is to detect well known code smells, while our approach defines a new kind of code smell based on co-change dependencies (*evolutionary smell*), and specifies how to detect them.

A refactoring is a program transformation that improves the internal quality of a software design while preserving its external behavior [27]. Several *Integrated Development Environments* have tools that perform the mechanical aspects of popular kinds of refactoring, such as *extract method*, *rename method*, *move method*, etc. Refactoring has been a topic explored by many research works, and for this study we are particularly interested in the research on automated refactoring recommendation approaches.

Ouni et al. [17] proposed an approach to recommend sequences of refactoring using the multi-objective genetic algorithm [35] NSGA-II [36]. Their approach aims to find the best sequence of refactoring that (a) minimizes the number of bad smells, (b) maximizes the use of development history, and (c) maximizes the semantic coherence. To compute the use of development history, they use three metrics: (1) similarity with previous refactorings applied to similar code fragments, (2) number of changes applied in the past to the same code elements to modify, and (3) a score that characterizes the co-change of elements that will be refactored. The third metric uses as input the co-change dependencies

between the coarse-grained entities that contains the entities to be refactored—i.e., if the recommended refactoring is a move method, then the score that characterizes the co-change of this refactoring is the number of times the source and destination class of the method was changed together in the past. While Ouni et al. approach uses coarse-grained co-change dependencies as a source of information to their refactoring recommendation algorithm, our approach uses fine-grained co-change dependencies. In addition, they use co-change information to complement other metrics, while our approach aims to *remove* co-change dependencies.

Mkaouer et al. [16] also proposed an approach to recommend sequences of refactorings using a multi-objective genetic algorithm. Differently from Ouni et al., they use the newer NSGA-III [37] algorithm. They also use the source-code change history as an input to their algorithm, but only to compute the similarity of a candidate refactoring with past refactorings. In their algorithm, a good refactoring recommendation must present a high similarity with past refactorings. The work of Wang et al. [10] explores clustering algorithms on MDGs containing *fine-grained* source-code entities as a basis for identifying refactoring opportunities. Their approach is a system-level multiple refactoring algorithm, which is able to automatically identify move method, move field, and extract class refactoring opportunities, according to the "high cohesion and low coupling" principle. Their algorithm works by merging and splitting related classes to obtain the optimal functionality distribution from the system-level. In their work, they present the *REsolution* as an publicly available automatic refactoring tool that implements their approach. Although their work brings empirical evidence about the potential of using fine-grained source-code entities to improve a software decomposition, the authors do not take into account co-change dependencies.

JDeodorant is a refactoring tool plugin for Eclipse that detects code smells in Java software, and recommends the appropriate refactorings to correct them [9]. JDeodorant address five types of code smells, including Feature Envy, Long Method, and God Class. It also supports several refactorings, including *move method* and *extract class*. JDeodorant uses a metric that their authors call *Entity Placement*, which is used to sort the refactoring recommendations according to their effect on design. JMove is also an Eclipse based plugin for recommending refactoring, though only deals with *Feature Envy* and *Long Method* code smells—using the *move method* refactoring for Java projects [18]. The authors of JMove argue that it is more efficient than JDeodorant, because it considers not only the structural properties of the source code (e.g., size of methods and static dependencies), but also semantic dependencies based on the source code vocabulary.

Methodbook is an approach to recommend *move method* refactorings that aims to remove *feature envy* bad smells [38]. It uses relational topic models to discover the "friends" of the methods in a system, and the class that contains the highest number of friends of the method under analysis is suggested as the target class of the move method refactoring.

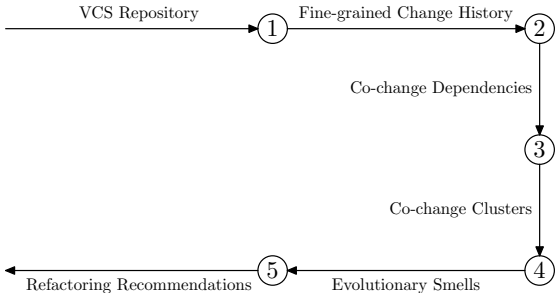
Differently, our approach does not aim to remove well-known code smells.

Instead, we leverage the knowledge about co-change dependencies to discover evolutionary smells and to recommend refactorings that removes these smells and reduce the total number of co-change dependencies between classes in a system.

3. Draco Approach

In this section we present the major design decisions related to the Draco approach for recommending move method and move field refactorings, which relies on historical data available in Version Control Systems (VCS). Figure 1 shows an overview of the approach, and the following subsections detail its steps.

Figure 1: Overview of the Draco approach for recommending refactorings.



3.1. Producing Fine-grained Change History

Popular VCSs such as Git[1] and Subversion[2] help to maintain the evolution source-code artifacts in a reliable way. An user of a VCS submit change sets (involving one or more artifacts) in the form of a *commit*. Accordingly, the history of changes submitted to a VCS can be described as a sequence of commits $H = (c_1, c_2, \ldots, c_n)$, where each commit contains a subset of artifacts in the form $c_i \subseteq A$. Since in this paper we are actually interested in the change history of fine-grained source-code entities (e.g., methods or fields), instead of coarse-grained entities (e.g., files or classes), here we first have to preprocess the original change history to produce a more detailed one (which we call *fine-grained change history*). This detailed change history can be described as a sequence $H' = (c'_1, c'_2, \ldots, c'_n)$, where each commit is a subset of fine-grained source-code entities $c'_i \subseteq F$ that changed together. To transform a change history (H) into a fine-grained change history (H'), we analyze each source-code artifact of a commit to discover which fine-grained entities have been modified. We take advantage of Kenja[3], a software utility that produces fine-grained change history from Git repositories.

[1] `https://git-scm.com/`
[2] `https://subversion.apache.org/`
[3] `https://github.com/niyaton/kenja`

3.2. Computing Co-change Dependencies

As discussed before, two source-code entities are co-change dependent upon each other when they *frequently* change together. Certainly, the precise definition of *frequently* depends upon how often these two entities changed together, and we compute this information considering the fine-grained change history. More specifically, we use two metrics to determine if two entities e_a and e_b change frequently together: *support count* and *confidence*. The first counts the number of commits in which both e_a and e_b appear together; while the second corresponds to the ratio of the *support count* between e_a and e_b and the number of commits containing e_a. Note that, while the support count is commutative, i.e., the support count between e_a and e_b is the same of the support count between e_b and e_a, the confidence is not, i.e., the confidence between e_a and e_b can be different from the confidence between e_b and e_a. We consider that e_a and e_b change frequently if their support count and confidence are above the threshold for supporting count S_{min} and confidence C_{min} at least in one direction. Several studies on co-change dependencies use the values $S_{min} = 2$ and $0.4 \leq C_{min} \leq 0.5$ (e.g., [26, 7, 39, 11, 40]). Although we relied on the literature and employed these thresholds for our metrics, we present a discussion about how these parameters influence the Draco approach in Section 5.8.

3.3. Computing the Co-change Clusters

We create a *co-change graph* $G = (V, E)$ from a set of fine-grained source-code entities V and a set of co-change dependencies $E \subseteq V \times V$. A partition of a co-change graph corresponds to a set of (co-change) clusters, whose quality (high cohesion and low coupling) depends on the number of dependencies that are internal or external to the cluster.

To measure the quality of a partition, in this study we use the Modularization Quality (MQ) metric (see Eq. (1)), proposed by Mitchell and Mancoridis [6].

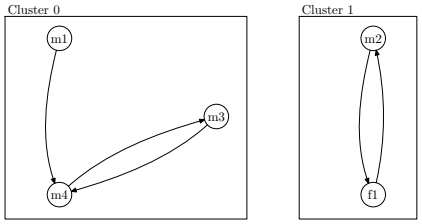
$$MQ = \begin{cases} \left(\frac{1}{k}\sum_{i=1}^{k} A_i\right) - \left(\frac{1}{\frac{k(k-1)}{2}}\sum_{i,j=1}^{k} E_{i,j}\right) & \text{if } k > 1 \\ A_1 & \text{if } k = 1, \end{cases} \tag{1}$$

In this equation, k is the number of clusters, A_i is the number of edges within the i^{th} cluster, and $E_{i,j}$ is the number of edges between the i^{th} and the j^{th} clusters.

Due to the number of possible solutions ($O(2^{|V|})$), we use a genetic algorithm (GA) [35] to compute an optimal partition. The goal of a GA is to find acceptable solutions for optimization problems. In general, to use a genetic algorithm, it is necessary to precisely define the concept of *individuals* and *fitness functions* for the problem domain. A typical GA executes as follows:

1. It first generates an initial population (i.e., a set of individuals) randomly;
2. It repeatedly produces a new population, by (a) selecting individuals from the previous population using the fitness values and (b) combining them using the genetic operators *crossover* and *mutation*;

Figure 2: Individual representation



(a) A simple example of co-change clusters

0	1	2	3	4
0	1	0	0	1

(b) Corresponding *array* representation.

3. It proceeds until a stop condition is met.

An extension for the traditional GAs is necessary when the problem has several objectives to be optimized. In this case, each individual has not only one fitness value, but instead a vector of values [35]. Accordingly, to compare two individuals, we used the concept of *Pareto Dominance*: a vector v dominates another vector u if no value v_i is smaller than the value u_i, and at least one v_j is greater than u_j [35].

Similarly to a previous work [41], we used a multi-objective genetic algorithm to compute the co-change clusters (in our case the Non-dominating Sorting Genetic Algorithm—NSGA-II [36]), representing the individuals as a mapping from a fine-grained source-code entity to the cluster it belongs to. Technically, an individual is an array where each position corresponds to a source-code entity, and each value corresponds to a co-change cluster. Two entities belong to the same cluster when they appear at different positions and refer to the same value. Figure 2-(a) illustrates this representation, showing four methods (m1, m2, m3, and m4) and one field (f1). All methods belong to the cluster C_0, except for m2 that belongs to the cluster C_1 (together with field f1). In addition, as we can see in Figure 2-(b), the array is codified as a binary string (i.e., as a sequence of bits) and the maximum number of clusters is set to $\frac{|V|}{2}$. In this way, we set each element of the array to occupy $\left\lceil \log_2 \frac{|V|-1}{2} \right\rceil$ bits of the binary string—where V is the set of vertices of the co-change graph.

There are several choices of selection operators, such as *roulette wheel*, whose probability of selecting an individual is proportional to its fitness value, and *tournament selection*, which selects the best individual according to a fitness value [35]. Here we use the tournament selection operator. The genetic operators transform the population through successive generations, maintaining the *diversity* and *adaptation* properties from previous generations. In more details, we use the one-point crossover operator, which takes two binary strings (parents) and a random index as input; and produces two new binary strings (offspring) by swapping the parents' bits after that index. For example, if we have the parent binary strings $p_1 = 101010$ and $p_2 = 001111$, and an index $i = 1$, the offspring will be $c_1 = 101111$ and $c_2 = 001010$. We also used a mutation operator that can flip any bit of the individual's binary string at a specified probability.

That is, given a mutation probability p and a binary string $s = b_1 b_2 \dots b_n$, we produce a random number $0 \leq r_i < 1$ for each bit b_i, flipping b_i in the cases where $r_i < p$. For example, if we have a binary string $s = 10011$, a mutation probability $p = 0.1$, and a sequence of random numbers $r = (0.9, 0.3, 0, 0.6, 0.5)$, the algorithm will produce a *mutant binary string* $s' = 10111$.

Also relying on the Praditwong et al. work [41], we setup our GA to optimize five objectives:

- maximize MQ;
- maximize intra-edges;
- minimize inter-edges;
- maximize number of clusters;
- minimize the the difference between the maximum and minimum number of source-code entities in a cluster.

We chose the parameters similarly to Candela et al [25]. As such, given a co-change graph $G = (V, E)$, and $n = |V|$, we defined the parameters population size (PS), maximum number of generations (MG), crossover probability (CP), and mutation probability (MP) as follows:

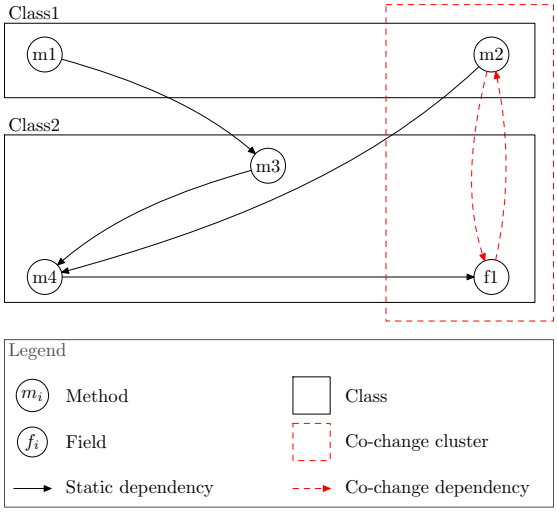
- $PS = \begin{cases} 2n & \text{if } n \leq 300 \\ n & \text{if } 300 < n \leq 3000 \\ n/2 & \text{if } 3000 < n \leq 10000 \\ n/4 & \text{if } n > 10000 \end{cases}$
- $MG = \begin{cases} 50n & \text{if } n \leq 300 \\ 20n & \text{if } 300 < n \leq 3000 \\ 5n & \text{if } 3000 < n \leq 10000 \\ n & \text{if } n > 10000 \end{cases}$
- $CP = \begin{cases} 0.8 & \text{if } n \leq 100 \\ 0.8 + 0.2(n - 100)/899 & \text{if } 100 < n < 1000 \\ 1 & \text{if } n \geq 1000 \end{cases}$
- $MP = \frac{16}{100\sqrt{n}}$

3.4. Discovering Evolutionary Smells

Building on Martin's Common Closure Principle [42], an evolutionary smell appears when fine-grained entities that frequently change together are not declared within the same class. Note that, differently from other works (e.g., Palomba et al. [34]), we are not using co-change dependencies to detect well-known bad smells [27]. Instead, we are describing a suspicious situation involving co-change dependencies between seemingly unrelated pieces of source-code, which could lead to a reorganization of the code. We identify evolutionary

smells when a co-change cluster contains fine-grained entities from more than one class, and at least one of the entities does not have any dependency (static or co-change) upon another entity from the same class. Figure 3 illustrates an instance of this smell.

Figure 3: Example of evolutionary smell. Method m2 and field f1 are from different classes but belong to the same co-change cluster and method m2 does not have any dependency on any other method or field from its own class (Class1).



Intuitively, when we find a situation similar to the aforementioned, we can suppose that the fine-grained source-code entity (e.g., `m2` in Figure 3) might have been declared at the wrong place. Nonetheless, we only characterize an evolutionary smell when the fine-grained entity has at least one static dependency with another class. Therefore, besides computing co-change dependencies, to find an evolutionary smell we also have to calculate the static dependencies of a project. For this reason, our tooling suite includes a static dependency finder that we implemented using two existing libraries: JavaParser[4] and JavaSymbolSolver.[5]

While this definition of evolutionary smell bear a resemblance to the "shotgun surgery" [27] bad smell, it is a stricter definition that brings focus on detecting methods or fields that could have more affinity with another class than with the class where it is declared.

3.5. Recommending Refactorings to Remove Evolutionary Smells

A naive solution to remove an evolutionary smell is to move the corresponding fine-grained source-code entity to one of the classes that belong to the co-

[4] `https://github.com/javaparser/javaparser`
[5] `https://github.com/javaparser/javasymbolsolver`

change cluster. Unfortunately, this is not always possible because when we move the source-code entity we also move the dependencies (static or co-change) from the source class to the destination class, and this might actually introduce new dependencies as a side effect. Our decision was to design a quite conservative approach. Accordingly, given the entity source-code e from class C_1, which belongs to a co-change cluster that contains entities from a class C_2, we only recommend to move the entity e to class C_2 when:

- (**Constraint #1**) the total number of dependencies (edges) of the MDG representing the software *after* applying the refactoring must be smaller than the number of dependencies of the MDG representing the software *before* applying the refactoring. Bellow we present some situations where this constraint is **not** satisfied:
  - There is at least one dependency (static or co-change) between C_1 and C_2 not involving the entity e. In this case, it is useless to move entity e to C_2 because, after that, C_1 will still depend on C_2 .
  - There is at least two dependencies between another class, say C_3, and C_1, one of them involving the entity e and the another not involving e. Moreover, there is no dependency between C_2 and C_3. In this case, if we move entity e to C_2, we will increase the number of dependencies—since C_3 will depend on both C_1 and C_2 after moving the entity e from C_1 to C_2.
- (**Constraint #2**) if C_1 has subclasses, we cannot move entities from C_1, since this could change the behavior of the system unpredictably. This is a general constraint for the move method/field refactoring that we also have to consider to implement a behavior-preserving transformation.

If all these constraints are satisfied, we can recommend to move a fine-grained source-code entity e to another class belonging to the same cluster while **reducing** the number of dependencies of the system. This is possible because the moved method does not use the implicit parameter (`this`); otherwise the method would have a static dependency with the source class, and therefore the definition of "evolutionary smell" would not be fulfilled.

Furthermore, there is nothing particularly special about circular dependencies. For example, consider that we have four classes with each one having on static dependency on another, such that $C_1.m_1 \rightarrow C_2.m_2 \rightarrow C_3.m_3 \rightarrow C_4.m4 \rightarrow C_1.m1$, where $C_x.m_y$ means "the method m_y from class C_x", and $x \rightarrow y$ means that x depends on y. If we move the method m_1 from class C_1 to class C_2, then we will remove the dependency between the two classes, and the dependency $C_4.m_4 \rightarrow C1.m_1$ will become $C_4.m_4 \rightarrow C_2.m1$. Therefore, the constraints will be fulfilled and the refactoring will be recommended. Differently, if we also have a dependency $C_4.m_4 \rightarrow C_1.m_5$, it will remain even after the move method, and therefore the dependencies will be moved, but the number of dependencies will be the same. In this case, Draco does not recommend a refactoring.

According to our decisions, if the element to be moved e has dependencies with two or more classes in a co-change cluster, we choose as the destination class of the refactoring the class that have the highest number of dependencies (static or co-change) with the original class of e that will be removed after applying the refactoring. If there are two or more target classes that will result in the same number of reduced dependencies, the Draco tool presents these classes as alternative recommendations.

4. Evaluation

We conducted two empirical studies to analyze the outcomes of applying a set of refactorings based on the recommendations of our approach. The general goal of this assessment is to understand whether or not the use of co-change dependencies to recommend refactorings leads to an improvement on software design.

4.1. Research Questions

Based on the general goal of our empirical studies, we organized this investigation with the aim of answering the following research questions:

(RQ1) How does the Draco approach behave when improving the design quality of a system?

(RQ2) How does the Draco approach compare to a state of the art approach for refactoring recommendation?

(RQ3) What is the impact of the different thresholds when extracting co-change dependencies on the results?

(RQ4) How effective are Draco and other refactoring recommendation tools?

To answer **RQ1** we first collected a set of design quality metrics of open-source systems in their original form, executed our approach on them, and then collected the same metrics again, though considering the effect of the recommended refactorings. We also carried out statistical tests to better understand the result of applying our proposed approach. To answer **RQ2** we executed a state of the art approach [10] for recommending refactorings, and used it as a baseline when comparing it with our approach, using the same metrics they used in their original published work [10]. To answer **RQ3**, we executed our approach with several combinations of the *support count* and *confidence* parameters, and compared the resulting number of evolutionary smells and refactoring recommendations that Draco found.

To answer **RQ4**, we asked software developers and architects to qualitatively analyze refactoring recommendations from Draco, considering three proprietary enterprise Java systems, two from the Brazilian Army (SISDOT and SISBOL) and one from the Brazilian Ministry of Economy (SIOP). For the two Brazilian Army Systems, we sent pull-requests and requested the contributors of these

systems to analyze a set of refactoring recommendations—which came not only from Draco, but also from other three different tools (REsolution, JDeodorant, and JMove). Considering the SIOP case, we conducted a survey with a list of recommendations, to collect the opinion of the developers that are responsible for maintaining the code involved in refactorings. Convenience was the main reason for using these systems to answer **RQ4**. First, the source code repositories of these systems were available to our research. Second, we had access to the architects and developers that were developing these software systems. Accordingly, we could discuss with the original architects of these systems the reasons for accepting or rejecting a contribution, and alternative solutions for the problems that were spotted by the recommendations tools. This is the main reason we did not use pull-request to open-source projects to answer **RQ4**, since in this kind of project, pull-requests might have to wait an excessive time to be reviewed [43, 44], or the recommendations might be rejected without an insightful explanation.

Figure 4 shows a more concrete example of a recommended refactoring computed using our approach. In this example (from the SIOP project), our approach detected an evolutionary smell involving the `getFields` method of the `ReportParameters` class. This method has co-change dependencies with the `generateModule` and `getJasperPrint` methods and static dependencies with the `generateModule` and `transformDataIntoDataSource` methods, all from the `ReportGenerator` class. The methods `getFields` and `generateModule` belong to the same co-change cluster. Our approach then recommended to move the method `getFields` to the `ReportGenerator` class, and thus it removes four dependencies between `ReportParameters` and `ReportGenerator` classes.

5. First Study: Quantitative Assessment

The goal of the quantitative assessment is to answer the first three research questions (**RQ1**, **RQ2**, and **RQ3**), introduced in the previous section. We made an analysis based on metrics and compared the results after applying the recommendations from Draco and from REsolution.

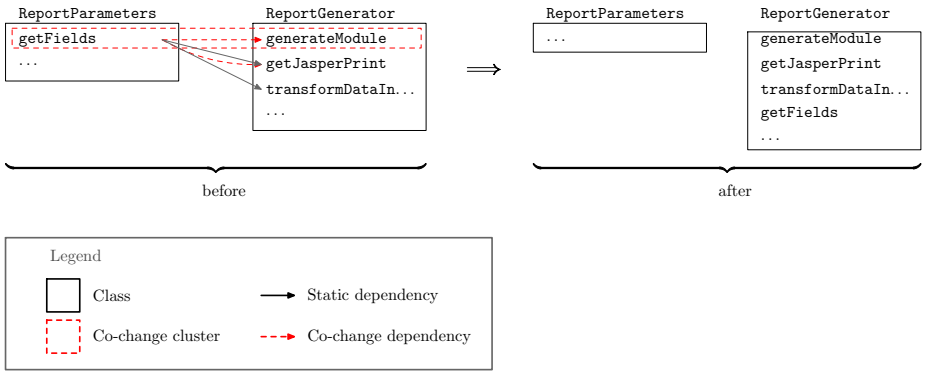
5.1. Studied Systems

We considered a number of representative open-source Java systems to investigate questions **RQ1**, **RQ2**, and **RQ3**. To this end, we first used GitHub to search for popular candidate projects, according to their number of stars. Star is known as a proxy for project popularity, as it reflects the project's activity level and developer interest [45]. This is also a common approach for selecting open source projects to investigate [46, 47]. In order to filter out small (to avoid toy projects) or very large projects (to avoid spending an excessive processing time and to keep the experiment in a reasonable time frame), we only considered projects whose *change history size* was in the interval between 5,000 and 50,000 commits, and a minimum code size of 10MB. To get the list of projects we used a query from the GitHub GraphQL API. The number of stars and the

Figure 4: Real example of a successful refactoring using our approach.

```java
package br.gov.siop.service.report;
public class ReportParameters {
    public static Map<String, String> getFields(int reportType) {
        Map<String, String> map = new LinkedHashMap<String, String>();
        switch (reportType) {
        case IReportGenerator.REPORT_GENERATOR_PROGRAM:
            map.put("program_name", "Program");
            map.put("program_agency_name", "Agency");
            // long method...
        }
        return map;
    }
}
```

(a) Excerpt from SIOP source-code before refactoring



(b) Graphical representation of the dependencies before and after refactoring

number of commits appear in the results of a query, allowing us to select only the projects that satisfy our criteria. The minimum code size was passed as an argument to the query. After applying all these filters, we selected the first 49 Java software systems, sorted according with their number of stars (we do not used a threshold for the stars, we select this number of projects according to the available time we had for the experiment). The set of selected systems include popular projects, such as `Cassandra`, `Gradle`, and `React Native`. Table 2 presents additional information about the systems we considered in this assessment.

5.2. Software Mining Procedures

Regarding the first assessment, we converted each project repository under study from GitHub to a fine-grained repository. We ignored both *automatic*

generated and *testing* code from our analysis (for example, we ignored all source-code within the `src/test` folders in Maven and Gradle projects). The resulting repositories are publicly available at the companion websites[6]. After that, we extracted the co-change dependencies from each fine-grained repository using the thresholds 2 for minimum support count and 0.5 for minimum confidence. Still, to reduce noise, as suggested by Beck and Diehl [26], we also discarded commits that affect more than 50 fine-grained entities. Whenever we found a commit on the fine-grained change history that removes a previously included (and maybe updated) entity, we do not include that entity in the co-change graph (and the corresponding edges). After that, we computed the co-change clusters for all projects using a genetic algorithm (NSGA-II), configured as detailed in Section 3.3. Due to the intrinsic randomness nature of the NSGA-II algorithm, we repeated the clustering process 30 times for each project, and consider the highest MQ value on all executions to select the best partition for the individuals.

The clustering process is particularly resource-intensive. Table 1 shows the time and space taken by a single execution of the clustering algorithm — and for the sake of comparison, a execution of the REsolution tool — for a sample which we consider in our research. We selected this sample according to their properties (both largest and smallest codebase, and largest and smallest number of commits.) As we have to run the clustering algorithm 30 times for each system, we often allocate multiple CPU cores such that we can run multiple clustering processes in parallel, one for each core. However, the memory consumption increases linearly in relation to the number of cores used, i.e., if we use eight cores in parallel, we will consume eight times more memory than using only one core. To collect these measures, we ran the tools in a machine with an eight-core i7 Intel CPU with 3.4 GHz and 16GB of memory.

Table 1: Time and space requirements for a representative sample of studied systems.

System	KLOC	Commits	Draco		REsolution		Observations
			Time	Space	Time	Space	
Hadoop	1,211	14,528	9h	5GB	14h	4GB	Largest codebase
OsmAnd	230	34,278	6h	4GB	15min	2GB	Largest number of commits
Drools	16	10,395	2h	2GB	1.5h	2GB	Smallest codebase
jOOQ	133	5,022	19min	0.5GB	8min	1.8GB	Smallest number of commits

5.3. Metrics

In order to evaluate the effect of applying the recommended refactorings on the design quality, we used several metrics such as Propagation Cost (PC) [48], Coupling Between Objects (CBO) [49], and the set of QMOOD (Quality Model for Object Oriented Design) metrics [50]. We chose these metrics because they have been used in a number of studies, including a recent research work that

[6] `https://github.com/project-draco` and `https://github.com/project-draco-hr`

evaluates a state of the art approach for recommending refactorings [10]. In this way, we actually evaluate three quality attributes (*Reusability*, *Flexibility*, and *Understandability*) that are defined in terms of these design metrics. In what follows, we present the set of metrics and quality attributes considered in this paper.

- **Coupling Between Objects (*CBO*)** Indicates if there is a dependency between two classes. That is, CBO is zero when there is no dependency; and one if there is least one dependency (such as a method call or a field access).
- **Message Passing Coupling (*MPC*)** Total of method calls and field access between classes. In this paper we also sum up the number of the co-change dependencies between classes.
- **Propagation Cost (*PC*)** Number of direct and indirect dependencies between classes. If the classes and dependencies between them are represented by a graph, the *PC* metric is the number of edges of the transitive closure of that graph.
- **Cohesion Among Methods of Class (*CAM*)** Average length of the intersection of parameters types of a method with all parameters types in a class.
- **Class Interface Size (*CIS*)** Number of public methods in a class.
- **Design Size in Classes (*DSC*)** Total number of classes in a system.
- **Data Access Metric (*DAM*)** Ratio between the number of non-public fields and the total number of field in a class.
- **Measure of Aggregation (*MOA*)** Number of fields of user defined types.
- **Number of Polymorphic Methods (*NOP*)** Number of overridden methods.
- **Average Number of Ancestors (*ANA*)** Average number of classes from which a class inherits.
- **Number of Methods (*NOM*)** Number of methods defined in a class.
- **Reusability** Ability of a design to be reapplied to a new problem without significant effort. It is defined by Bansiya and Davis [50] as:

  $$Reusability = -0.25 \times MPC + 0.25 \times CAM + 0.5 \times CIS + 0.5 \times DSC$$

- **Flexibility** Ability of a design to incorporate changes. It is by Bansiya and Davis [50] defined as:

  $$Flexibility = 0.25 \times DAM - 0.25 \times MPC + 0.5 \times MOA + 0.5 \times NOP$$

- **Understandability** Property of a design that enables it to be easily to learn and comprehend. It is defined by Bansiya and Davis [50] as:

$$\begin{aligned} Understandability &= -0.33 \times ANA + 0.33 \times DAM \\ &\quad -0.33 \times MPC + 0.33 \times CAM \\ &\quad -0.33 \times NOP - 0.33 \times NOM - 0.33 \times DSC \end{aligned}$$

5.4. Performing the Refactorings

In this first assessment we followed the approach of Tsantalis and Chatzigeorgiou [9] to simulate and evaluate the application of recommended refactorings. That is, instead of applying the refactorings on the original source-code, we first build a graph $G = (V, E)$, where V is the set of classes of a system and $E \subseteq V \times V$ is the set o dependencies between them. After that, we *virtually* apply the refactorings in this graph, changing the edges of the graph G according to the move method/field recommendations.

In more details, one class C_1 depends upon another class C_2 if there is either a static or co-change dependency from any fine-grained source-code entity of C_1 to any entity of C_2. After (virtually) applying the recommended refactorings on G, we obtain a new graph $G' = (V, E')$, where $E' \subseteq V \times V$ is possibly different from E. There is also a *weight* function $w : V \times V \rightarrow \mathbb{N}$ that represents the number of dependencies (static or co-change) from the entities in the source class to the entities in destination class of the edge. If a method m_1 makes n calls to a method m_2, the result of applying the weight function is n. We simulate a move method/field in three steps. First, we move a fine-grained entity from the source to the destination class. After that, we recompute all edges involving the source and destination classes. Finally, we recompute all weights of the affected edges.

5.5. Results

After running Draco according to the previous sections on the 49 selected systems, we were able to identify 610 evolutionary smells on 48 systems, leading to 56 recommendations of move method/field refactorings that resolve evolutionary smells from 18 systems. All these refactorings satisfy the constraints discussed in Section 3.

We compared Draco with the existing Wang et al. work [10], that also recommends move method/field refactorings. This related work is particularly relevant because it outperforms several earlier research techniques for refactoring recommendation (e.g., [51, 52, 53, 54]).

We manually executed the *REsolution* tool, provided by Wang et al. [10], and collected the move method/field refactoring recommendations for the same 18 systems. However, *REsolution* recommended refactorings for 14 systems only. Table 2 summarizes these results.

We then virtually applied the recommended refactorings in three different ways:

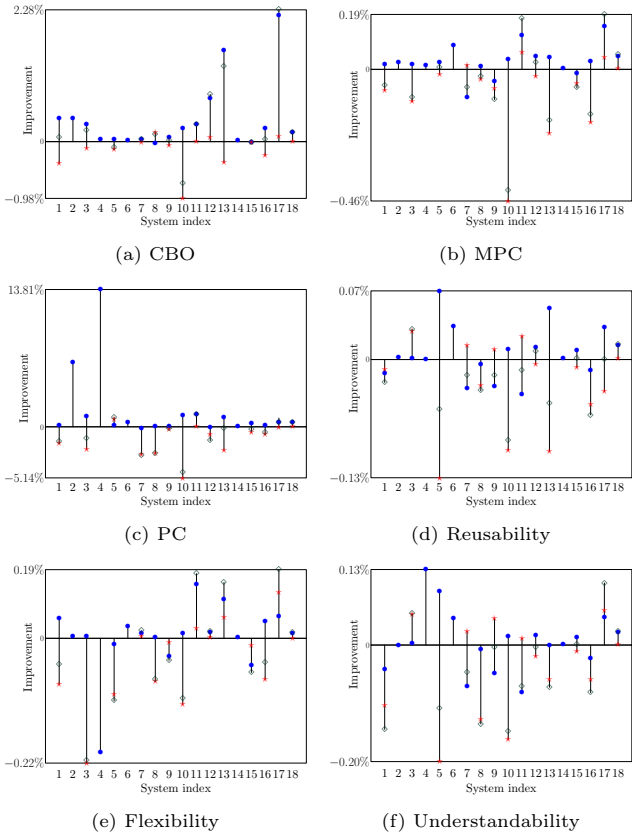
Table 2: Studied systems.

System	KLOC	Commits	# Evol. Smells	# Ref. Recomm.
Actor Messaging platform	157	8,772	5	2
The ownCloud Android App	36	5,329	3	
Atmosphere Event Driven Framework	41	5,748	2	
Bazel build system	375	7,258	21	
BigBlueButton web conferencing system	82	13,420	2	1
Broadleaf Commerce – Enterprise eCommerce	168	9,784	22	
Buck build system	412	7,726	5	1
CAS - Enterprise Single Sign On	87	6,268	3	1
Cassandra partitioned row store	385	21,710	19	1
c:geo Android geocaching app	75	10,183	21	
Closure Compiler	303	8,293	5	
CoreNLP suite of core NLP tools	552	11,963	51	
Deeplearning4j deep learning & linear algebra for Java	121	5,645	1	
Drools rule engine	16	10,395	12	1
Druid analytics data store	297	7,452	12	
Elasticsearch Engine	611	24,491	11	
Fabric8 microservices platform	45	13,130	3	
FBReaderJ e-book reader	68	9,012	11	
Flink stream processing framework	419	9,565	5	
Gradle build tool	283	38,756	41	4
Grails Web Application Framework	71	17,315	7	
Groovy core language	156	12,379	10	
Groovy language	161	13,465	15	
H2O-2 Machine Learning Platform	95	16,172	4	
H2O-3 Machine Learning Platform	143	19,336	13	2
Hadoop distributed computing	1,211	14,528	18	
Hazelcast In-Memory Data Grid	531	21,136	32	4
Hibernate Object-Relational Mapping	628	7,302	2	
Hive data warehouse facilities	1,025	9,201	40	
Jitsi communicator	326	12,420	20	3
jMonkeyEngine game development suite	183	5,966	2	
jOOQ SQL generator	133	5,022	2	
Kill Bill Billing & Payment Platform	139	5,361	4	2
LanguageTool Style and Grammar Checker	75	19,121	8	1
libGDX game development framework	257	12,562	7	
Liquibase database source control	77	5,360	23	17
Minecraft Forge	72	5,498	7	
Openfire XMPP server	196	7,436	6	
openHAB home automation platform add-ons	331	8,868	2	1
OpenTripPlanner multi-modal trip planner	90	8,698	6	
OrientDB Multi-Model DBMS	390	14,118	16	
OsmAnd navigation application	230	34,278	30	
Pinpoint Application Performance Management	245	8,565	8	2
Presto distributed SQL query engine for big data	400	8,597	13	6
Processing Core and Development Environment	97	12,171	3	
React Native framework for building native apps	48	7,842	3	2
Spring Framework	548	13,312	23	5
Storm distributed realtime computation system	213	7,451	0	
VoltDB in-memory SQL RDBMS	573	23,131	31	
Total			610	56

- first, we applied the refactoring recommendations to all 18 systems that Draco found a recommendation;
- second, we applied the refactoring recommendations to the 14 systems that REsolution found a recommendation;
- third, we applied the refactoring recommendations from both approaches to the 14 systems that both *REsolution* and Draco at least one recommendation.

Figure 5 shows the impact on the metrics *CBO*, *MPC*, *PC* (that measure coupling), *Reusability*, *Flexibility*, and *Understandability* (that measure quality attributes), for the 18 systems. The values represent the impact on the metrics

Figure 5: Improvement on design metrics after applying recommended refactorings from both tools. Symbols mean: ●=Draco, ⋆=REsolution,◇=Draco and REsolution combined. The correspondences between index and system are: 1-Actor, 2-BigBlueButton, 3-Buck, 4-CAS, 5-Cassandra, 6-Drools, 7-Gradle, 8-H2O-3, 9-Hazelcast, 10-Jitsi, 11-Kill Bill, 12-LanguageTool, 13-Liquibase, 14-openHAB, 15-Pinpoint, 16-Presto DB, 17-React Native, 18-Spring Framework.



(a) CBO (b) MPC

(c) PC (d) Reusability

(e) Flexibility (f) Understandability

after applying the refactorings. We normalized the metrics in all figures, and thus the *better values* correspond to the greater values. The results for each approach are denoted by different symbols, as follows. A "●" symbol represents the Draco approach (ours). A "⋆" symbol represents REsolution (Wang et al.) approach. A "◇" symbol represents the combination of Draco + REsolution. Based on these results, it is possible to realize that Draco outperforms the REsolution approach, in the majority of the cases. Also, the combination of the two approaches frequently is beneficial w.r.t. the improvement of the quality metrics measured.

Table 3: Mann-Whitney U test p-values of Draco when compared with Wang et al. and the original system metrics (with Benjamini-Yekutieli correction). Note that the Draco approach improves the majority of the metrics with a statistical significance at least 95% ($p\text{-}value < 0.05$).

Metric	Wang et al.	Original System
CBO	0.0031403	0.0020175
MPC	0.0008967	0.0020175
PC	0.0031402	0.0059976
Reusability	0.2537220	0.4706940
Flexibility	0.0031402	0.0329610
Understandability	0.4380600	0.7791000

Table 4: Cohen's d effect size statistics of Draco when compared with Wang et al. and the original system metrics. Note that the Draco approach leads to a non-negligible improvement on all metrics when compared with Wang et al. approach, and on the majority of the metrics when compared with the original system.

Metric	Wang et al.	Original System
CBO	0.8733506 (large)	0.6618745 (medium)
MPC	0.9090697 (large)	0.9057448 (large)
PC	0.9323233 (large)	0.4244010 (small)
Reusability	0.4381047 (small)	0.2465364 (small)
Flexibility	0.9200228 (large)	0.1503500 (negligible)
Understandability	0.3255604 (small)	0.1307608 (negligible)

5.6. (RQ1) How does the Draco approach behave when improving the design quality of a system?

We executed the Mann-Whitney U statistical significance test and the Cohen's d effect size test for these six metrics. Specifically, we tested if Draco performs significantly better than the Wang et al. approach and if the improvement is significant upon the original system metrics. Table 3 shows the results of the significance tests. Considering *CBO*, *MPC*, *PC*, and *Flexibility*, Draco leads to a significant improvement when compared with the original system decomposition and the resulting decomposition computed using the Wang et al. approach (at a 0.05 significance level). Considering *Reusability* and *Understandability*, the improvement was not statistically significant. However, considering the results of the Cohen's d effect size test (Table 4), it is possible to realize that Draco leads to a non-negligible improvement for all metrics when compared with the Wang et al. approach, and also a non-negligible improvement for all metrics when compared with the original system decomposition, except for *Flexibility* and *Understandability*.

We also measured how the systems' attributes relates to the improvement on the quality metrics presented in Section 4. We considered the following attributes: (1) refactoring recommendations count; (2) co-change clusters mean density; (3) fine-grained source-code entities count; (4) static graph density; and (5) co-change graph density. We employed a multiple regression analysis

model to determine if these attributes have a statistically significant effect on the quality metrics improvement. Table 5 shows the results, revealing that the most effective attribute on quality metric improvement is *refactoring recommendations count*, since it has a statistical significance of 99% (*p-value*< 0.01).

Table 5: Effect of attributes on metrics improvement. Note that we have only two attributes influencing a metric with 99% of significance (*p-value*< 0.01, denoted by a ** suffix). While we have three attributes influencing three metrics with 95% significance (*p-value*< 0.05, denoted by a * suffix).

	CBO	*MPC*	*PC*	*Reusab.*	*Flexib.*	*Understandab.*
Intercept	−0.0097	0.0001	0.1085	0.0001	−0.0010	0.0008
	(0.0055)	(0.0007)	(0.0556)	(0.0005)	(0.0010)	(0.0010)
Refactoring recommendations count	**0.0007****	**0.0001***	−0.0018	0.0000	0.0001	0.0000
	(0.0002)	0.0000	(0.0022)	0.0000	0.0000	0.0000
Co-change clusters density	0.0010	−0.0005	0.0465	**−0.0008***	−0.0008	−0.0005
	(0.0040)	(0.0005)	(0.0403)	(0.0003)	(0.0007)	(0.0007)
Entities count	0.0000	0.0000	**0.0001***	0.0000	0.0000	0.0000
	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Static graph density	**127.3095****	5.9265	−377.8864	1.1333	6.0888	−1.2924
	(32.0436)	(3.8240)	(321.3327)	(2.7005)	(5.8045)	(5.6753)
Co-change graph density	−137.8211	−0.6133	−1320.5310	11.6540	14.1054	2.2192
	(83.5465)	(9.9701)	(837.8044)	(7.0409)	(15.1339)	(14.7971)
R^2	0.801154	0.559688	0.403463	0.390744	0.328650	0.147522
Adj. R^2	0.718302	0.376225	0.154906	0.136887	0.048920	−0.207677
Num. obs.	18	18	18	18	18	18
RMSE	0.003176	0.000379	0.031844	0.000268	0.000575	0.000562

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Actually, the *static graph density* also affect the *CBO* metric with 99% of statistical significance (*p-value*< 0.01). The *refactoring recommendations count* attribute also affects the *MPC* metric (in this case with 95% of significance). The *entities count* negatively affects the *PC* metric (with 95% of significance), suggesting that for smaller systems Draco tends to produce greater improvements in the *PC* metric. Finally, the *co-change clusters density* negatively affects the *Reusability* metric (with 95% of significance). A dense co-change cluster might suggest that the entities belonging to a cluster tend to change together. Based on the results of this analysis, we did not find any attribute that significantly affect the remaining two metrics (*Flexibility* and *Understandability*).

Answer to **RQ1**: The Draco approach reduces coupling when measured by *CBO*, *MPC*, and *PC* metrics, considering both co-change and static dependencies. The improvement is proportional to the number of identified refactoring opportunities.

5.7. (RQ2) How does the Draco approach compare to a state of the art approach for refactoring recommendation?

The results discussed in the last section leads to several findings. First, Draco improved the *CBO*, *MPC*, or *PC* metrics for all 18 systems. However, the Wang et al. approach improved *CBO* for only 5 out of 14 systems, *MPC* for only 4 systems, and *PC* for only 2 systems. For these three metrics (*CBO*, *MPC*, and *PC*), in only two cases the Wang et al. approach outperforms Draco.

We also found that in the situations where both approaches improve a given metric, the use of them in combination improves even further. That is, combining both approaches tends to lead to an improvement that is equivalent to the sum of the improvements of the approaches taken individually, which suggest that the two approaches are complementary. This occurs because no refactoring recommended by Draco was also recommended by the Wang et al. approach.

Given the measures of quality attributes investigated here (*Reusability*, *Flexibility*, and *Understandability*) for the 18 studied systems (54 measures), Draco improved 36 (66.66%) measures, while the Wang et al. approach improved 17 of 42 measures (40%, for 14 systems). Besides that, Draco outperforms the Wang et al. approach on 29 measures, while the opposite occurs on 13 measures. Likewise the other quality metrics, the improvement of the two approaches can be summed together when we apply them together.

Answer to **RQ2**: these results suggest that the Draco approach outperforms a state of the art techniques for recommending move method/field refactorings, when we consider both co-change and static dependencies.

5.8. (RQ3) What is the impact of the different thresholds when extracting co-change dependencies on the results?

To choose the parameters used to compute co-change dependencies, namely *minimum support count* (S_{min}) and *minimum confidence* (C_{min}), we had to consider several trade-offs. The values of S_{min} or C_{min} are inversely proportional to the number of co-change dependencies found, i.e., low values produce a higher number of dependencies, and high values produce a lower number of dependencies. Furthermore, low values leads to "weaker" co-change dependencies, and since they produce a higher number of dependencies the computation of co-change clusters takes more time. Also, since high threshold values produce fewer dependencies, the likeability of find refactoring opportunities is lower.

Table 6: Effect of parameters combinations on results. The best combination in terms of *refactoring recommendations* is in **bold**.

Parameter	Smells detected	Refactoring recommendations	Edges count
S_2 $C_{0.4}$	57	25	32,663
$\boldsymbol{S_2}$ $\boldsymbol{C_{0.5}}$	57	32	28,509
S_2 $C_{0.6}$	58	25	26,850
S_2 $C_{0.7}$	48	26	22,545
S_2 $C_{0.8}$	49	26	21,609
S_3 $C_{0.5}$	22	18	4,888
S_4 $C_{0.5}$	11	9	1,575
S_5 $C_{0.5}$	7	7	714

S_α means $S_{min} = \alpha$, and C_β means $C_{min} = \beta$

To analyze how different parameters values affect the result of our experiment, we computed evolutionary smells and refactoring recommendations for 8 different parameters combinations. We experimented the values $3, 4, 5$ for the parameter S_{min} while setting the parameter $C_{min} = 0.5$, and the values $0.4, 0.6, 0.7, 0.8$ for the parameter C_{min} while setting the parameter $S_{min=2}$. As the computation of co-change clusters is a time-consuming task (see Section 5.2), in this analysis we used only the 10 smaller systems from the original set of 49 studied systems.

Table 6 shows the results of this analysis. Accordingly, the parameter combination that produces more *refactoring recommendations* is $S_{min} = 2$ and $C_{min} = 0.5$, which is the combination we used in Section 5.2. We can see that except for $C_{min} = 0.5$ all other combinations with $S_{min} = 2$ are equivalent in terms of refactoring recommendations, mainly because the number of the edges of the co-change graphs reduces just a few when we increase the confidence value. Also, we can see that lower confidence values does not necessarily increase the number of refactoring recommendations, according to the results for $S_2\ C_{0.4}$ and $S_2\ C_{0.5}$. This result suggests that weaker co-change dependencies could not be good enough to produce refactoring recommendations. On the other hand, increasing the support count parameter value significantly reduces both the number of smells detected and the number of refactoring recommendations, mainly because the number of edges of the co-change graphs is severely reduced when the support count increases.

Answer to **RQ3**: confidence parameters ranging from 0.4 to 0.8 have little impact on the number of evolutionary smells and refactoring recommendations found. However, the use of support count values greater than 2 significantly reduces both the number of evolutionary smells and refactoring recommendations.

5.9. Manual Verification of the Refactorings

To mitigate a possible threat related to the applicability of the Draco refactoring recommendations, we manually applied a sample of the Draco recommendations. To this end, we randomly selected 10 refactoring recommendations from 9 systems. Before applying the refactorings, we built the systems and ran their unit test cases. Nonetheless, we were not able to successfully build or test three projects, and thus we discarded three refactoring recommendations. From the six remaining projects we were able to apply five refactorings without any modification.

From the two remaining move method refactorings, we had to rename a method before applying one of the recommendations, because the target class already had a method with the same name. Although we could have introduced a new constraint to avoid moving a method to a class that already declares a method with the same name, we make the decision to recommend the refactoring, delegating the renaming of the method before moving it to the software developers. The last recommendation involved the overriding of a method from its superclass. We observed that the method had an empty implementation in

the superclass and only one of the subclasses in fact overrides that method. Therefore, this is an instance of the *Refused Bequest* bad smell [27] (in addition to the evolutionary smell, of course). Accordingly, we manually applied the *Push Down* refactoring [27], which is the appropriate refactoring to remove the Refused Bequest smell. Again, we could have introduced a new constraint to avoid moving a method that overrides a method from an interface or superclass, but we prefer to recommend the refactoring, delegating the necessary adjustments before the refactoring to a software developer.

These two *weak preconditions* align to the argument that developers often prefer tools that do not discard refactoring opportunities, even when some fixes are necessary to perform the refactoring [55, 56]. In summary, after a few adjustments, we successfully applied all Draco refactoring recommendations we selected for manual verification.

5.10. Threats to Validity Related to the First Study

Although we applied the same approach to all studied systems, we cannot ensure that a given combination of thresholds favor or disfavor a particular system. To mitigate this effect, we chose the co-change dependencies thresholds according to the procedure detailed at Section 5.8. For computing co-change clusters, we did not compare the performance of the Genetic Algorithm with other alternatives, like Hill Climbing or Random Search. While previous work discusses that multi-objective GA's usually outperforms other approaches for software clustering [41], the results can be different when using a different clustering algorithm.

We only found refactorings for 18 out of 48 systems. This may suggest that part of our constraints are overly strong. Although we believe that weakening these constraints would increase the number of refactorings recommendations, a more detailed investigation is left for future work. To do that, we might accept to move a method/field to another class even in the cases where some (static or co-change) dependencies would remain. Nevertheless, the baseline technique we used to compare with our approach found refactoring opportunities in a smaller number of systems, even considering that our approach is rather conservative.

We selected a set of open-source Java systems for this study. This can potentially limit the generalization of our results. However, we choose a wide range of applications domains, that had a large code base with a long history of maintenance tasks. Therefore we expect that our findings would be reproducible in some other projects too. In the future, we plan to reduce these threat by experimenting with systems written in different programming languages.

Finally, during the manual verification of the refactorings, we found that some recommendations require a few adjustments before they could be applied. In particular, the recommendations can involve methods that (a) collide with methods in the target class that have the same name of the moving method, or (b) override methods from superclasses or interfaces. We chose to keep these recommendations in the study because it is possible to overcome this limitation, for example, by (a) renaming the moving method before the refactoring, or (b) letting the target class implement the interface that declares the method, or

(c) performing a Push Down refactoring as we have discussed in the previous section.

Furthermore, these recommendations can spot design problems that might lead to a redesign of the classes involved in the refactoring (see Section 6.2.1 for a concrete example). Nevertheless, this might be also an option in the Draco tool, which could allow the user to choose if the refactoring recommendations can refer to methods that have names of methods already present in the target class, and methods that override some interface or superclass method.

Since the goal of this first study was to quantitatively assess the effect of the refactoring recommendations on design quality metrics, we did not analyze if the recommendations (both from Draco and REsolution) introduce new architectural issues. In particular we did not verify if the sequence of refactorings lead to the creation of "god classes", or even violate some architectural constraints or design quality attributes (such as separation of concerns). Our idea is to investigate the feasibility of augmenting the Draco tool with additional options, in order to avoid recommending refactorings that might either violate architectural constraints or that could eventually produce "god classes".

6. Second Study: Qualitative Assessment

The goal of this second study is to answer the fourth research question, and therefore to understand how useful are the recommendations made by Draco, as well as three other existing refactoring tools: REsolution, JDeodorant, and JMove.

In this section, we use the term "useful" to indicate how suitable the recommendations from these tools are, considering the opinion of architects and developers from three existing enterprise systems (SIOP, SISDOT, and SISBOL). SIOP (Integrated Planning and Budget System) was developed by the Brazilian Ministry of Economy, and is one of the most important Brazilian systems in the public finance domain. It supports the Federal Public Planning and Budget process, where all agencies from all federal powers participate. It is a large Java Enterprise Edition System with more than 600 KLOC (thousand lines of code) of Java code, 30,000 commits, and is currently maintained by a team with more than 15 developers. SISDOT and SISBOL systems have been developed in a research cooperation project between the Brazilian Army and the University of Brasília, and have been used in previous studies to explore development approaches and technical design decisions [57, 58]. The first system (*Material Endowment System*, SISDOT) deals with the distribution of materials and equipment to all organizational units of the Brazilian Army (considering well-defined rules of distribution). This is a Java Enterprise Edition system with more than 15 contributors, 1800 commits, and 40 KLOC of Java code. The second system (*Bulletin System*, SISBOL) manages the internal official communication of events within the Brazilian Army. It is a configurable system, which supports specific communication workflows for the individual organizational unities of the Brazilian Army. SISBOL is an enterprise system

based on a service-oriented architecture [59]. Its current implementation comprises almost 20 contributors, 1130 commits, 20 KLOC of Java code and 10 KLOC of JavaScript code using the AngularJS framework.

6.1. Study Settings

In this second assessment, we gathered the outcomes of Draco and three other tools for recommending refactorings, all configured using their default settings, to find refactoring opportunities in both SISDOT and SISBOL. We then concretely applied the recommended refactorings. First, we got a copy of the `develop` branch and then created a new branch for each recommended refactoring (from the `develop` branch). After applying the recommended changes in the source code, we sent a pull-request for each recommendation. In this way, we could collect the perceptions of the architects and developers to each individual recommendation. We also assessed the refactoring recommendations from Draco for SIOP. However, instead of applying the refactorings, in this case we actually sent lists of refactoring recommendations to the developers that are responsible for the maintenance of the code involved in the recommendation. After that, this developers performed the refactorings in their personal branches and sent back to us their feedback about the recommendations.

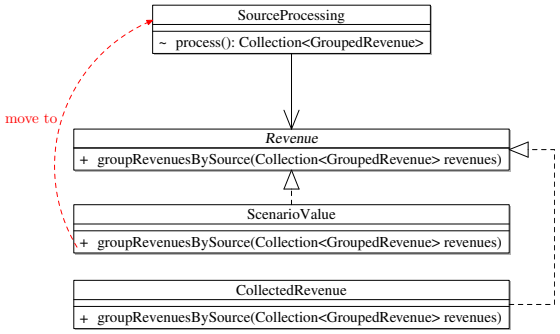
6.2. Results

Table 7 summarizes the results of this second study. It is possible to realize that JMove presents the higher acceptance rate, although it recommended a smaller number of refactorings. The other tools recommended refactorings with an acceptance rate around 30%. Curiously, none of the refactoring recommendations for SISBOL have been accepted. In addition, the results suggest that the tools complement each other, in particular because there is almost no intersection between the refactoring recommendations from the different tools. In the following sections we present more details about the results of each refactoring recommendation tool.

Table 7: Summary of the analysis for the refactoring recommended from four different tools

Tool	Draco	REsolution	JDeodorant	JMove
SISDOT Recommendations	0	3	10	2
SISDOT Accepted	0	1	3	2
SISBOL Recommendations	2	0	1	0
SISBOL Accepted	0	0	0	0
SIOP Recommendations	4	–	–	–
SIOP Accepted	2	–	–	–
Total Recommended	6	3	11	2
Total Accepted	2	1	3	2
Acceptance rate	33%	33%	27%	100%

Figure 6: A rejected move method recommendation from Draco for SIOP. The recommendation was to move the method `groupRevenuesBySource` from class `ScenarioValue` to class `SourceProcessing`. It was rejected because it would cause a compilation error, since the implementation of the method is required by interface `Revenue`. (The classes and interfaces in this figure has additional methods and attributes that were omitted because they are unrelated to the refactoring recommendation)

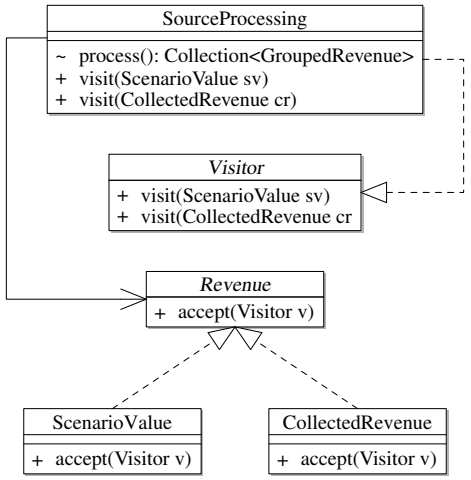


6.2.1. Draco Analysis

Draco recommend four SIOP refactorings and two were accepted. The first accepted recommendation was to move a method (`getFields`) that returns a list of report fields given a report type, from class `ReportParameters` to the class `ReportGenerator` (see Figure 4 presented in Section 4). Both are service classes. The recommendation spotted a misplaced responsibility allocation at class `ReportParameters`, as the method `getFields` is unrelated with report parameters. The recommended refactoring reduced the coupling between the two classes and augmented the cohesion of the destination class. The second accepted recommendation from Draco was to move a method (`create`) from a service class (`ScenarioService`) to a presentation class (`ScenarioPageBean`). The method `create` just initializes a `Scenario` object with default values for use with an registration form. No class is interested in this method except `ScenarioPageBean`. After the refactoring, the coupling between the two classes decreased and the cohesion of the destination class `ScenarioPageBean` increased.

Draco also recommended two SIOP refactoring that were rejected. The first one was to move a method (`groupRevenuesBySource`) that belongs to an interface implemented by the source class (`ScenarioValue`) to the class that uses the interface (`SourceProcessing`), as shown in Figure 6. The recommendation was rejected, because if we apply this refactoring, the source-code would not compile. However, after analyzing the recommendation with careful consideration, we saw that the implementations of the method `groupRevenuesBySource` have low cohesion with the classes where they are declared. The rationale for the creation of this interface is that the logic of grouping revenues by source depends on the kind of revenue. As each kind of revenue is implemented by a class, the architects let these classes implement the logic of grouping. Thus, the caller would have to know only the interface. However, in practice the implementation classes do not have to know how to group revenues by source, as this logic is ex-

Figure 7: A better design for the classes and interfaces involved in the rejected refactoring recommendation for SIOP as illustrated by Figure 6. The adoption of the *visitor pattern* allowed to decouple the logic of grouping revenues by source from the `Revenue` implementors, keeping the logic of `process` method generic. (The classes and interfaces in this figure has additional methods and attributes that were omitted because they are unrelated to the refactoring recommendation)



clusively handled by class `SourceProcessing`. Therefore, a better design would be the use of the *visitor pattern*, as shown in Figure 7. This new design will preserve the generic nature of the caller, while moving the "grouping by source logic" to the class of the caller, and therefore increasing their cohesion. As this alternative design must be manually implemented, the architects decided to keep the original design for now. The second rejected refactoring recommendation from Draco for SIOP was to move the method `doFilter` from class `GZIPFilter` to class `GZIPResponseStream`, both are presentation classes. This method is declared in the interface `javax.servlet.Filter` (which is part of the *Java Enterprise Edition* specification), which is implemented by class `GZIPFilter`. If we simply move the method from the current class, the resulting source code would not compile. This is the reason for not accepting this recommendation. However, the recommendation also revealed a design problem, that is the high logical coupling between the classes `GZIPFilter` and `GZIPResponseStream`. The architects agree that a better design would be merging the two classes and let the resulting class implement the `javax.servlet.Filter` interface (in addition to other interfaces that the two original classes implement). Again, since this design change has to be manually implemented, the current design will be kept for now.

Draco recommended two SISBOL refactorings. None of these recommendations were accepted. The recommendations made by Draco to SISBOL were rejected due to a similar reason. `PR-SISBOL-39` recommends to move a method from a service class to a resource class. These types of classes follow a typi-

Figure 8: A move method recommendation from REsolution. This recommendation led to an accepted pull-request

cal JavaEE service-oriented decomposition, and both should deal with different responsibilities. For this reason, although this recommendation removes co-change dependencies, moving methods between these types of classes violates one architectural constraint of the system. `PR-SISBOL-40` recommends moving a static attribute from a class (that declares several `strings` for mapping keys into user messages) to a service class. Again, although this recommendation reduces co-change dependencies, it violates an architectural constraint that states that all message keys should be kept in specific classes.

6.2.2. REsolution Analysis

Table 7 shows that REsolution found 3 refactoring opportunities for SISDOT (and none for SISBOL). One of these pull-requests (`PR-SISDOT-28`) was accepted by the architects and developers. `PR-SISDOT-28` recommends to move a method from a class that implements a business service (`SiglaFracaoServico`) to a class that implements a domain model (`Sigla`) (see Figure 8). This recommendation improves cohesion and helps to avoid *anemic domain objects* [60]. Also considering REsolution, two other pull-requests were rejected. `PR-SISDOT-30` suggests to move a static method from an utility class (`QuadroDeCargosUtil`) to a value object. This recommendation might improve some structural property, though decreases the cohesion of both class and does not comply with the SISDOT architectural constraint of keeping utility methods in utility classes. Surprisingly, `PR-SISDOT-29` recommends moving another static method from/to the same classes. Considering that SISDOT has more than 400 Java classes, we were not expecting a small number of move methods recommendations involving the same classes. Moreover, we also discarded a REsolution refactoring. In this case, this refactoring suggested to move the `values()` method from an enumeration to another class. Since this method is generated by the compiler, the refactoring is not possible. This qualitative assessment suggests that the REsolution tool can be the subject of further studies.

6.2.3. JMove Analysis

As one can see from Table 7, the two recommendations proposed by JMove were accepted by the software development team. As the name of the refactoring tool suggests, both recommendations are related to the move method refactoring. In the first pull-request (`PR-SISDOT-7`), the intention was to move a method from a business class to a model class. The rationale here is that the method `generateCode` does not pertain to the business class because it does not use any attribute from this class. In fact, this method does string

concatenation using solely the attributes from the model class (characterizing a feature envy smell). After moving this method to the appropriated class, the client code changed from `entity.setCodot(generateCode(entity));` to `entity.setCodot(entity.gerarCodot());`. The other pull-request (`PR-SISDOT-8`) goes along the same lines.

6.2.4. JDeodorant Analysis

According to Table 7, JDeodorant found 10 refactoring opportunities when considering the SISDOT project. Three of these recommendations were accepted—all based on the particular pattern we show in Listing 1. For this particular case (`PR-SISDOT-17`), we replace an assignment in the form `<var> = null;` followed by some particular logic for correctly initializing `<var>` (`classes` in the code of Listing 1). The original assignment and related *correct initialization* are factored out using the factory method design pattern. All accepted recommendations from JDeodorant are based on this refactoring template.

Listing 1: DIFF of the pull request `PR-SISDOT-17`

```
-   HashMap<Integer, ClasseMaterial> classes = null;
-
-   if(!consolidacao.equals(NivelDetalhamento.DETALHES)) {
-     classes = vo.consolidarMateriais();
-   }
-   else {
-     classes = vo.getClasses();
-   }
-
+   HashMap<Integer, ClasseMaterial> classes = classes(vo, consolidacao);
// ...
+ private HashMap<Integer, ClasseMaterial> classes(FracaoQDMRelatorioVO vo,
+   NivelDetalhamento consolidacao) {
+   HashMap<Integer, ClasseMaterial> classes = null;
+   if(!consolidacao.equals(NivelDetalhamento.DETALHES)) {
+     classes = vo.consolidarMateriais();
+   } else {
+     classes = vo.getClasses();
+   }
+   return classes;
+ }
```

Table 7 also shows that the remaining recommendations from JDeodorant were all rejected. For instance, Listing 2 shows a refactoring recommended (and rejected) by JDeodorant. In this particular case (`PR-SISDOT-22`), the recommendation tries to solve a long method bad smell (almost 50 lines of code) by removing an assignment to a call to a new method. The result is that it does not significantly reduce the number of lines of the original (long) method and introduces a new method and a method call. In this case, JDeodorant correctly

identified the long method, but the proposed refactoring does not lead to a code improvement (based on the opinion of the SISDOT development team).

Listing 2: DIFF of the pull request `PR-SISDOT-22`

```
//long method here....
  boolean found = true;
  while (found) {
-  found = false;
+  found = found(paragraph, searchText, found);
   int pos = paragraph.getText().indexOf(searchText);
   if (pos >= 0) {
-    found = true;
//... end of the long method.
}
// new method recommended by JDeodorant
+ private boolean found(XWPFParagraph paragraph, String searchText, boolean found) {
+   found = false;
+   int pos = paragraph.getText().indexOf(searchText);
+   if(pos >= 0) {
+    found = true;
+   }
+   return found;
+ }
```

Other refactoring recommendation from JDeodorant (`PR-SISDOT-21`) creates a new method that is a clone of an existing method from the superclass. Similarly to `PR-SISDOT-22`, `PR-SISDOT-20` corresponds to an extract method recommendation that reduces four lines of code of a long method by introducing a new method and a method call.

Listing 3 shows the resulting diff of applying another recommended refactoring from JDeodorant (`PR-SISDOT-19`). In this case, a `for` loop was moved to a new method (`map`), though without leading to a perceptive improvement in the source code (in particular because the original method has only five lines of code). `PR-SISDOT-18` and `PR-SISDOT-13` presents a similar structure, and for this reason they were rejected, while `PR-SISDOT-15` was considered hard to understand and bringing small benefit to the design.

Listing 3: DIFF of the pull request `PR-SISDOT-18`

```
  void replace(XWPFDocument document, Map<String, V> map) {
    List<XWPFParagraph> paragraphs = document.getParagraphs();
+   map(map, paragraphs);
+ }
+
+ void map(Map<String, V> map, List<XWPFParagraph> paragraphs) {
    for (XWPFParagraph paragraph : paragraphs) {
      replace(paragraph, map);
```

```
}
```

Regarding SISBOL, JDeodorant found one opportunity for applying the *extract class* refactoring. However, although the development team agree that the existing class should be refactored, extracting part of its responsibilities to a new class would not be the right decision—because this would lead to a design that does not fit the architectural decomposition of SISBOL. For this reason, the SISBOL development team decided to reject pull-request `PR-SISBOL-37`.

6.3. Discussion

Although we cannot generalize our findings (as discussed in the next section), the results of our qualitative study reveal that, for one large and two typical small to medium size Java Enterprise Systems, existing tools for recommending refactorings identify a few opportunities to improve the design of a software, and several recommendations do not bring concrete benefits to the design. In particular, here we give evidence that refactoring recommendation tools should be augmented with the architectural decisions of the projects, reducing the number of recommendations that do not fit the design of the systems. This might suggest future research development.

For SISDOT and SISBOL, the small number of recommendations might have been motivated due to an "above of the average" quality of the systems. For instance, their development leveraged agile practices like code inspection, pair-programming and coding dojo, which in the end could mitigate design problems. Perhaps even more importantly, the development teams of these two systems are a mix of both experienced and novice developers. This leads to another question: "How often the developers of SISDOT and SISBOL refactor the design of the systems?" We used Refactoring Miner [61] to answer this question and to identify the number of refactorings performed in both systems during their development process. Table 8 shows the results.

In the case of SISDOT, Refactoring Miner identified 1,877 refactorings, including 277 move methods, 176 extract methods, and 23 extract and move methods. These are the types of refactorings we are most interested in and that might have been recommended by the tools we analyzed here. Therefore, state-of-the-art refactoring tools may be missing potential refactoring opportunities. Unfortunately, when performing with SISBOL, Refactoring Miner did not successfully complete the analyses, and several exceptions of type `CheckoutConflictException` were logged during its execution. For this reason, Refactoring Miner identified only 88 refactorings performed at SISBOL.

Altogether, this study brings some evidence and other open questions. First, state-of-the-art refactoring recommendation tools identify a small fraction of the refactoring opportunities that are manually identified by developers during their development activities. In this way, we believe that it is necessary to further investigate how to improve these tools to make them more effective—at least for the domain of Java enterprise systems. Second, it is necessary to complement refactoring recommendation tools to consider design constraints of the systems, in order to avoid false-positive recommendations. Third, it might be

Table 8: Results of mining refactorings in both systems using Refactoring Miner

Refactoring	SISDOT	SISBOL
Change Package	4	0
Extract And Move Method	23	2
Extract Interface	13	9
Extract Method	176	4
Extract Superclass	20	4
Extract Variable	28	5
Inline Method	10	3
Inline Variable	5	0
Move And Rename Class	12	1
Move Attribute	94	12
Move Class	180	1
Move Method	277	13
Parameterize Variable	3	1
Pull Up Attribute	52	0
Pull Up Method	179	0
Push Down Method	4	0
Rename Attribute	45	3
Rename Class	111	3
Rename Method	423	9
Rename Parameter	85	6
Rename Variable	111	12
Replace Variable With Attribute	10	0
Total	1,877	88

worth to develop a product line of refactoring recommendation tools, so that we could run these tools in different configurations of heuristics to identify refactoring opportunities. In particular, the evaluated tools recommended different refactorings for both systems, and thus they might actually complement each other. Fourth, existing studies that only use metrics to compare refactoring recommendations tools are insufficient to explain the real consequences of using a particular approach.

Answer to **RQ4**: The results of our second study suggests that state-of-the-art tools for recommending refactoring are rather ineffective, because they recommend a small fraction of the refactorings carried out by the developers during the development of the software and because they recommend refactorings that do not consider the architectural decisions of the systems. Nonetheless, although JMove recommended only 2 refactorings, both have been accepted and integrated into SISDOT. The other tools recommend refactorings with an acceptance rate around 30%.

Although most of the recommendations have been rejected, it is important

to notice that the additional analysis of the recommendations by the architects and the comments to the pull requests suggest that the evaluated tools were able to correctly identify classes and methods with design flaws. In some situations, this information was relevant to at least start a discussion about future manual refactoring efforts.

6.4. Threats to Validity Related to the Second Study

One threat to validity of this study is that the refactoring tools do not recommend exactly the same kind of refactorings. For instance, JDeodorant have an extensive catalog of refactoring recommendations, expanding our notion of move method refactoring, covering other refactorings such as extract method and extract class. Therefore, since JDeodorant is broader in essence, it would be more likely to have more recommendations than, say, Draco. However, in our qualitative study, the focus was not on the number of recommendations found per se. Instead, we focused on whether the recommendations (wrapped within pull-requests) made any sense.

Similarly, we cannot guarantee that all recommendations can be fully automated. For example, we can possibly produce refactorings involving methods that override interface methods. On the other hand, other tools also produce recommendations that are hardly possible to apply. As an example, REsolution recommended a refactoring to move a method that is compiler-generated. We discarded these cases.

Moreover, a reader might consider that Draco is an ineffective approach, presenting an acceptance rate around 33%—although this rate is similar to other tools. Specifically, we observed that the majority of accepted refactorings did not affect the the design of the systems. However, since the Draco tool suggests refactorings that aim to improve the design, it might face some resistance. In fact, the requirement that refactoring tools must obey architectural constraints was studied before [25]. Still, the relatively low number of recommendations found by Draco could also be seen as a threat to validity. However, the recommendations still can be used to provoke discussion about the suitability of the design w.r.t. keeping co-changed source-code methods in the same class. Nevertheless, in a future work we will explore if relaxing some constraints of Draco approach—specifically allowing the introduction of new static dependencies after a refactoring recommendation—would increase the number of recommendations and acceptance rate, and consequently the effectiveness of the tool.

It is important to note that Draco does not blindly apply any refactoring, but instead, it recommends transformations that the developers of a system must ultimately review. As we present in Section 6.2, some of the Draco recommendations have not been integrated into the systems. The same is true for other refactoring recommendation approaches. Draco, and the other tools as well, could even recommend a refactoring that breaks either architectural constraints or the building process of a system. We can mitigate this problem by enforcing additional constraints, though we decide to weaken some of them, according to the recommendations of a previous work [56]. This motivates an additional question: *what are the implications of using Draco in a* well designed

system?. Trying to investigate this question, we used Draco to recommend refactorings for JHotDraw (a system recognized by its architecture and design decomposition). Even in this particular case, Draco recommended seven refactorings, which might somehow compromise the original design of the system. An interesting research question, which we aim to explore as future work, is the correlation between co-change dependencies and more specific design constraints of a system (including the use of design patterns).

Another limitation is that we studied only three software projects. However, we believe that they are representative once they are written using the same programming language that the studied refactoring tools work on. More importantly, since we could have access to the development teams, we could have better discussions regarding their rationale behind accepting or not the recommendations, which is not always the case when dealing with open-source projects (e.g., some pull-requests have to wait many months to be reviewed, others are not reviewed at all [43, 44])

Finally, we must note that, for different systems, the architecture, and therefore the architectural constraints, can vary. Therefore, we may have constraints in the studied systems that lead to rejection of refactoring recommendation that otherwise would be accepted by another architect of another system. The acceptance decision also depends on personal judgment of the architects, and this might be a confounding factor. Nevertheless, the three studied systems of this second study are based on the standard Java Enterprise Edition specification, and thus its architectural constraints are common w.r.t another enterprise systems that adopt the same architectural style.

7. Conclusion

In this paper we presented a novel approach that addresses the lack of refactoring recommendation tools that consider co-change dependencies. Developers regard this kind of dependency as important, when reasoning about software remodularization [25]. Accordingly, our approach remove co-change dependencies (and eventually static dependencies) between classes, reducing the coupling of the system and therefore improving its design. Our approach detect and remove evolutionary smells, which manifest when methods or fields from different classes are co-change dependent but do not have any dependency on methods or fields from the same class where they are declared. We evaluated our approach using 49 open-source systems and found 610 evolutionary smells on 48 systems, and 56 refactoring recommendations on 18 systems—even considering that our approach is overly conservative. After applying the recommended refactorings, we found that our approach improves the design of the system (considering coupling metrics such as *CBO*, *MPC*, and *PC*) and outperforms a state of the art refactoring recommendation tool (*REsolution*) [10]. Still, we conducted a qualitative assessment to understand the effectiveness of the recommendations (i.e., if developers would be willing to accept the automated contributions). To this end, in addition to the two refactoring tools (Draco and REsolution), we also

explored the refactorings recommended by other two state-to-the-art refactoring tools (JMove and JDeodorant). In this analysis we perceived that, although the overall number of recommendations was small, some of them were, indeed, accepted by the software development team. Finally, the accepted recommendations from Draco demonstrate its feasibility, and we also found that some of the the rejected recommendations started discussions about design flaws and its alternative solutions.

References

[1] D. L. Parnas, On the criteria to be used in decomposing systems into modules, Commun. ACM 15 (1972) 1053–1058. URL: http://doi.acm.org/10.1145/361598.361623. doi:10.1145/361598.361623.

[2] D. L. Parnas, Software aging, in: Proceedings of the 16th International Conference on Software Engineering, ICSE '94, IEEE Computer Society Press, Los Alamitos, CA, USA, 1994, pp. 279–287. URL: http://dl.acm.org/citation.cfm?id=257734.257788.

[3] J. van Gurp, J. Bosch, Design erosion: problems and causes, Journal of Systems and Software 61 (2002) 105 – 119. URL: http://www.sciencedirect.com/science/article/pii/S0164121201001522. doi:https://doi.org/10.1016/S0164-1212(01)00152-2.

[4] I. Ahmed, U. A. Mannan, R. Gopinath, C. Jensen, An empirical study of design degradation: How software projects get worse over time, in: 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2015, pp. 1–10. doi:10.1109/ESEM.2015.7321186.

[5] L. de Silva, D. Balasubramaniam, Controlling software architecture erosion: A survey, Journal of Systems and Software 85 (2012) 132 – 151. URL: http://www.sciencedirect.com/science/article/pii/S0164121211002044. doi:https://doi.org/10.1016/j.jss.2011.07.036, dynamic Analysis and Testing of Embedded Software.

[6] B. S. Mitchell, S. Mancoridis, On the automatic modularization of software systems using the bunch tool, IEEE Trans. Softw. Eng. 32 (2006) 193–208.

[7] L. L. Silva, M. T. Valente, M. de A. Maia, Co-change Clusters: Extraction and Application on Assessing Software Modularity, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 96–131. URL: https://doi.org/10.1007/978-3-662-46734-3_3. doi:10.1007/978-3-662-46734-3_3.

[8] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, W. Zhao, Interactive and guided architectural refactoring with search-based recommendation, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, ACM, New York, NY, USA,

2016, pp. 535–546. URL: http://doi.acm.org/10.1145/2950290.2950317. doi:10.1145/2950290.2950317.

[9] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Transactions on Software Engineering 35 (2009) 347–367. doi:10.1109/TSE.2009.1.

[10] Y. WANG, H. Yu, Z. l. Zhu, W. ZHANG, Y. l. ZHAO, Automatic software refactoring via weighted clustering in method-level networks, IEEE Transactions on Software Engineering PP (2017) 1–1. doi:10.1109/TSE.2017.2679752.

[11] M. C. de Oliveira, R. Bonifácio, G. N. Ramos, M. Ribeiro, Unveiling and reasoning about co-change dependencies, in: Proceedings of the 15th International Conference on Modularity, MODULARITY 2016, ACM, New York, NY, USA, 2016, pp. 25–36.

[12] D. Zhou, Y. Wu, L. Xiao, Y. Cai, X. Peng, J. Fan, L. Huang, H. Chen, Understanding evolutionary coupling by fine-grained co-change relationship analysis, in: Proceedings of the 27th International Conference on Program Comprehension, ICPC '19, IEEE Press, Piscataway, NJ, USA, 2019, pp. 271–282. URL: https://doi-org.ez54.periodicos.capes.gov.br/10.1109/ICPC.2019.00046. doi:10.1109/ICPC.2019.00046.

[13] M. D'Ambros, M. Lanza, R. Robbes, On the relationship between change coupling and software defects, in: 2009 16th Working Conference on Reverse Engineering, IEEE, 2009, pp. 135–144.

[14] I. S. Wiese, R. T. Kuroda, R. Re, G. A. Oliva, M. A. Gerosa, An empirical study of the relation between strong change coupling and defects using history and social metrics in the apache aries project, in: IFIP International Conference on Open Source Systems, Springer, 2015, pp. 3–12.

[15] E. Kouroshfar, Studying the effect of co-change dispersion on software quality, in: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, 2013, pp. 1450–1452.

[16] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, A. Ouni, Many-objective software remodularization using nsga-iii, ACM Trans. Softw. Eng. Methodol. 24 (2015) 17:1–17:45. URL: http://doi.acm.org/10.1145/2729974. doi:10.1145/2729974.

[17] A. Ouni, M. Kessentini, H. Sahraoui, M. S. Hamdi, The use of development history in software refactoring using a multi-objective evolutionary algorithm, in: Proceedings of the 15th annual conference on Genetic and evolutionary computation, ACM, 2013, pp. 1461–1468.

[18] R. Terra, M. T. Valente, S. Miranda, V. Sales, Jmove: A novel heuristic and tool to detect move method refactoring opportunities, Journal of Systems and Software 138 (2018) 19–36.

[19] T. Ball, J.-m. Kim, A. Porter, H. Siy, If your version control system could talk ..., in: ICSE '97 Workshop on Process Modeling and Empirical Studies of Software Engineering, 1997.

[20] T. Zimmermann, A. Zeller, P. Weissgerber, S. Diehl, Mining version histories to guide software changes, IEEE Transactions on Software Engineering 31 (2005) 429–445. doi:10.1109/TSE.2005.72.

[21] D. Beyer, A. Noack, Clustering software artifacts based on frequent common changes, in: 13th International Workshop on Program Comprehension (IWPC'05), 2005, pp. 259–268. doi:10.1109/WPC.2005.12.

[22] T. A. Wiggerts, Using clustering algorithms in legacy systems remodularization, in: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97), WCRE '97, IEEE Computer Society, Washington, DC, USA, 1997, pp. 33–.

[23] N. Anquetil, C. Fourrier, T. C. Lethbridge, Experiments with clustering as a software remodularization method, in: Proceedings of the Sixth Working Conference on Reverse Engineering, WCRE '99, IEEE Computer Society, Washington, DC, USA, 1999, pp. 235–.

[24] O. Maqbool, H. Babri, Hierarchical clustering for software architecture recovery, IEEE Trans. Softw. Eng. 33 (2007) 759–780.

[25] I. Candela, G. Bavota, B. Russo, R. Oliveto, Using cohesion and coupling for software remodularization: Is it enough?, ACM Trans. Softw. Eng. Methodol. 25 (2016) 24:1–24:28. URL: http://doi.acm.org/10.1145/2928268. doi:10.1145/2928268.

[26] F. Beck, S. Diehl, On the impact of software evolution on software clustering, Empirical Software Engineering 18 (2013) 970–1004. URL: https://doi.org/10.1007/s10664-012-9225-9. doi:10.1007/s10664-012-9225-9.

[27] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: improving the design of existing code, Addison-Wesley Professional, 1999.

[28] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change-and fault-proneness, Empirical Software Engineering 17 (2012) 243–275.

[29] R. Marinescu, Detection strategies: Metrics-based rules for detecting design flaws, in: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings., IEEE, 2004, pp. 350–359.

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

[31] M. J. Munro, Product metrics for automatic identification of" bad smell" design problems in java source-code, in: 11th IEEE International Software Metrics Symposium (METRICS'05), IEEE, 2005, pp. 15–15.

[32] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, H. Sahraoui, A bayesian approach for the detection of code and design smells, in: 2009 Ninth International Conference on Quality Software, IEEE, 2009, pp. 305–314.

[33] D. Rapu, S. Ducasse, T. Gîrba, R. Marinescu, Using history information to improve design flaws detection, in: Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings., IEEE, 2004, pp. 223–232.

[34] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, A. D. Lucia, Mining version histories for detecting code smells, IEEE Transactions on Software Engineering 41 (2015) 462–489. doi:10.1109/TSE.2014.2372760.

[35] D. E. Goldberg, E. 1989. genetic algorithms in search, optimization, and machine learning, Reading: Addison-Wesley (1990).

[36] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: Nsga-ii, IEEE Transactions on Evolutionary Computation 6 (2002) 182–197. doi:10.1109/4235.996017.

[37] K. Deb, H. Jain, An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints, IEEE transactions on evolutionary computation 18 (2013) 577–601.

[38] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, A. De Lucia, Methodbook: Recommending move method refactorings via relational topic models, IEEE Transactions on Software Engineering 40 (2013) 671–694.

[39] M. C. d. Oliveira, R. Bonifácio, G. N. Ramos, M. Ribeiro, On the conceptual cohesion of co-change clusters, in: Proceedings of the 2015 29th Brazilian Symposium on Software Engineering, SBES '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 120–129.

[40] L. L. Silva, M. T. Valente, M. de A. Maia, N. Anquetil, Developers' perception of co-change patterns: An empirical study, in: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, pp. 21–30. doi:10.1109/ICSM.2015.7332448.

[41] K. Praditwong, M. Harman, X. Yao, Software module clustering as a multiobjective search problem, IEEE Trans. Softw. Eng. 37 (2011) 264–282.

[42] R. C. Martin, Design principles and design patterns, 2000. URL: www.objectmentor.com, online.

[43] L. F. Dias, I. Steinmacher, G. Pinto, D. A. da Costa, M. A. Gerosa, How does the shift to github impact project collaboration?, in: 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016, 2016, pp. 473–477.

[44] R. Dantas, A. Carvalho, D. Marcilio, L. Fantin, U. Silva, W. Lucas, R. Bonifacio, Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate java programs, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE Computer Society, 2018, pp. 497–501. doi:10.1109/SANER.2018.8330247.

[45] H. Borges, A. C. Hora, M. T. Valente, Understanding the factors that impact the popularity of github repositories, in: 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016, 2016, pp. 334–344.

[46] G. Pinto, I. Steinmacher, M. A. Gerosa, More common than you think: An in-depth study of casual contributors, in: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1, 2016, pp. 112–123.

[47] B. Ray, D. Posnett, P. Devanbu, V. Filkov, A large-scale study of programming languages and code quality in github, Commun. ACM 60 (2017) 91–100.

[48] A. MacCormack, J. Rusnak, C. Y. Baldwin, Exploring the structure of complex software designs: An empirical study of open source and proprietary code, Manage. Sci. 52 (2006) 1015–1030.

[49] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (1994) 476–493.

[50] J. Bansiya, C. G. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Trans. Softw. Eng. 28 (2002) 4–17. URL: http://dx.doi.org/10.1109/32.979986. doi:10.1109/32.979986.

[51] G. Bavota, A. Lucia, A. Marcus, R. Oliveto, Automating extract class refactoring: An improved method and its evaluation, Empirical Softw. Engg. 19 (2014) 1617–1664. URL: http://dx.doi.org/10.1007/s10664-013-9256-x. doi:10.1007/s10664-013-9256-x.

[52] M. Fokaefs, N. Tsantalis, E. Stroulia, A. Chatzigeorgiou, Identification and application of extract class refactorings in object-oriented systems, J. Syst. Softw. 85 (2012) 2241–2260. URL: http://dx.doi.org/10.1016/j.jss.2012.04.013. doi:10.1016/j.jss.2012.04.013.

[53] I. Moore, Automatic inheritance hierarchy restructuring and method refactoring, in: Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '96, ACM, New York, NY, USA, 1996, pp. 235–250. URL: http://doi.acm.org/10.1145/236337.236361. doi:10.1145/236337.236361.

[54] M. Streckenbach, G. Snelting, Refactoring class hierarchies with kaba, in: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '04, ACM, New York, NY, USA, 2004, pp. 315–330. URL: http://doi.acm.org/10.1145/1028976.1029003. doi:10.1145/1028976.1029003.

[55] M. Vakilian, R. E. Johnson, Alternate refactoring paths reveal usability problems, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, 2014, pp. 1106–1116. URL: http://doi.acm.org/10.1145/2568225.2568282. doi:10.1145/2568225.2568282.

[56] M. Mongiovi, R. Gheyi, G. Soares, M. Ribeiro, P. Borba, L. Teixeira, Detecting overly strong preconditions in refactoring engines, IEEE Transactions on Software Engineering 44 (2018) 429–452. doi:10.1109/TSE.2017.2693982.

[57] P. H. T. Costa, E. D. Canedo, R. Bonifácio, On the use of metaprogramming and domain specific languages: An experience report in the logistics domain, in: Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse, SBCARS '18, ACM, New York, NY, USA, 2018, pp. 102–111. URL: http://doi.acm.org/10.1145/3267183.3267194. doi:10.1145/3267183.3267194.

[58] C. M. C. de Oliveira, E. D. Canedo, H. Faria, L. H. V. Amaral, R. Bonifácio, Improving student's learning and cooperation skills using coding dojos (in the wild!), in: IEEE Frontiers in Education, IEEE Computer Society, 2018, pp. 1–9.

[59] T. Erl, Soa: principles of service design, volume 1, Prentice Hall Upper Saddle River, 2008.

[60] M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, R. Stafford, Patterns of Enterprise Application Architecture, Addison-Wesley Professional, 2002.

[61] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, D. Dig, Accurate and efficient refactoring detection in commit history, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, ACM, New York, NY, USA, 2018, pp. 483–494. doi:10.1145/3180155.3180206.