University of Brasília

Exact Sciences Institute
Computer Science Department

Applying Fine-grained Co-change Dependencies to Identify Refactoring Opportunities

Marcos César de Oliveira

Thesis submitted in partial fullfilment of
the requirements to Doctoral Degree in Informatics

Advisor
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2021

University of Brasília

Exact Sciences Institute
Computer Science Department

Applying Fine-grained Co-change Dependencies to Identify Refactoring Opportunities

Marcos César de Oliveira

Thesis submitted in partial fullfilment of
the requirements to Doctoral Degree in Informatics

Prof. Dr. Rodrigo Bonifácio de Almeida (Advisor)
CIC/UnB

Prof. Dr. Marco Túlio Valente
Federal University of Minas Gerais

Prof. Dr. Tiago Lima Massoni
Federal University of Campina Grande

Prof. Dr. Vander Ramos Alves
CIC/UnB

Prof. Dr. Genaína Nunes Rodrigues
CIC/UnB

Prof. Dr. Genaína Nunes Rodrigues
Coordinator of Graduate Program in Informatics

Brasília, Apr 28, 2021

Dedicated to

Donald E. Knuth

for showing the level of excellence that computer programming and technical writing can achieve, albeit it is unattainable for the rest of us.

Acknowledgements

To my wife, Paula, and children, Beatriz, Bruno, Gabriela e Vítor, not only for the patience and understanding on the uncountable hours of dedication expended on this work, but also for their motivation and support.

To my advisor, Prof. Dr. Rodrigo Bonifácio, for his great energy, intelligence and intuition, that were crucial for the development and conclusion of this work. To Prof. Gustavo Pinto and Prof. David Lo, for their continuous and solid partnership during all this journey.

To all professors and colleagues that I have the honor to met during this program, for their invaluable insights and collaboration. This work would be impossible without their generosity.

To the Ministry of Economy, for the support and trust during my dedication to this work. I hope to give back to the Institution and to Brazil by becoming a better professional and spreading what I learned.

Resumo

Uma dependência de co-mudança de granularidade fina surge quando duas entidades de código fonte de granularidade fina, por exemplo, um método, mudam frequentemente juntas. Esse tipo de dependência é relevante ao considerar esforços de remodularização (por exemplo, para manter métodos que mudam frequentemente em uma mesma classe). Trabalhos de pesquisa existentes sugerem que dependências de co-mudança estão correlacionadas com problemas de *design*. Contudo, as atuais abordagens de recomendação de refatoramento que alteram a decomposição do *software* (tal como um *move method*) não exploram o uso de dependências de co-mudança de granularidade fina. Nessa tese apresentamos uma nova abordagem (chamada Draco) que recomenda refatoramentos de *move method* e *move field*, que remove dependências de co-mudança e *evolutionary smells*, um tipo particular de dependência que surge quando entidade de granularidade fina que pertencem a classes diferentes são alteradas juntas com frequência. Primeiramento avaliamos nossa abordagem usando 47 projetos Java *open-source*. Draco revelou 8,405 *evolutionary smells* e recomendou 4,844 refatoramentos que removem dependências de co-change—sem introduzir outros tipos de dependências. Uma avaliação quantitativa revelou que Draco supera outras abordagens existentes (por exemplo, REsolution e JDeodorant) ao recomendar refatoramentos quando se lida com dependências de co-mudança. Também avaliamos nossa abordagem submetendo *pull-requests* com as recomendações produzidas por nossa técnica, além das recomendações de outras ferramentas (REsolution, JDeodorant e JMove), no contexto de um sistema Java grande e dois de tamanho médio. Uma avaliação qualitativa mostrou que nossa abordagem é efetiva, não somente para recomendar refatoramentos mas também para revelar oportunidades de melhorias de *design*. Outro resultado dessa tese é que os resultados de ambas avaliações (quantitativa e qualitativa) sugerem que Draco pode complementar outras abordagens, já que suas recomendações não se sobrepõem.

Palavras-chave: Refactoração, dependências de co-mudança, remodularização, clusterização, qualidade de arquitetura

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). Existing research suggests that co-change dependencies are correlated with design problems. 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 thesis we present a novel approach (named Draco) 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. We first evaluated our approach using 47 open-source Java projects. Draco revealed 8,405 evolutionary smells and recommended 4,844 refactorings that remove co-change dependencies—without introducing other types of dependencies. A quantitative assessment reveals that Draco outperforms existing approaches (e.g., REsolution and JDeodorant) for recommending refactorings when dealing with co-change dependencies. We also evaluate our approach by submitting pull-requests with the recommendations of our technique, in addition to the recommendations from other tools (REsolution, JDeodorant, and JMove), in the context of one large and two medium size proprietary Java systems. A qualitative evaluation shows that our approach is effective, not only for recommending refactorings but also to reveal opportunities of design improvements. Another outcome of this thesis is that the results of both assessments (quantitative and qualitative) suggest that Draco can complement other approaches, since their refactoring recommendations do not overlap.

Keywords: Refactoring, co-change dependencies, remodularization, clustering, architecture quality

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Almost all non-trivial modern software is decomposed into smaller pieces, or modules. On one hand, decomposing a software system into modules might bring benefits w.r.t. comprehensibility and flexibility [1]. On the other, as the number of modules increases, it is also possible to introduce new dependencies that might, in fact, hinder developers to evolve the systems. Several kinds of dependencies are possible, such as, for example, the *static dependency*, where a module calls a method/function or access data from another module.

Software design is the discipline of decomposing a software into a set of modules accordingly to some criteria, in order to satisfy a software quality attribute. The information hiding principle as a decomposition criteria was first proposed by Parnas [1]. This principle argues that the information exposed by a module must reveal as little as possible about its inner workings [1].

Accordingly, we can assess this quality perspective of a software design by measuring how much a module "knows" about other modules, and how much the elements inside the modules are interconnected. Usually, we use the *coupling* and *cohesion* metrics as a proxy for this quality perspective of a software design [2]. We consider a software design as "good" if it presents a low total coupling between modules and a high average modules cohesion.

Computing how often the changes on a given module propagate to other modules also helps to estimate a software design quality [3]. The less *change propagation* had happened, the better the design is. To measure past change propagation, we can use the software's change history, which is usually recorded by a Version Control System (VCS). We can predict change propagations by detecting if two modules changed together in the past [4].

In order to reduce future change propagations between modules, and therefore to improve the software design, we can reorganize the source-code in such a way that internal elements from a module are moved to another module that frequently changed with them.

However, there is a gap in the state-of-the-art tools (e.g., [5, 6]) regarding the consideration of using the aforementioned technique.

In this thesis we use the concept of *co-change dependency* to predict future change propagations. A co-change dependency arises when two source-code elements are *frequently* changed together. We explore the definition of "frequently" later in this thesis.

The main goal of this thesis is to provide method and tools to enable practitioners to improve software design by performing source-code refactorings in order to reduce the amount of co-change dependencies between modules.

1.1 Research Contributions

In this thesis we explore how to leverage co-change dependencies to improve software design. We also investigated if reducing co-change dependencies could help to reduce bugs. This leads to our first research question.

RQ1 To what extent co-change dependencies metrics correlate to defect density?

One of the motivations for investigating this question is that results from previous research suggest a correlation between co-change dependencies metrics and defect density [7–9]. Therefore, answering this question helps us understand whether or not modules with high degrees of co-change dependencies are also more error-prone. In this way, breaking co-change dependencies could also reduce the number of bugs in a system.

Results.

We investigated the correlation between co-change dependencies with defect density. We extracted 22,532 bug-fixing commits from 29 Java Apache projects. Contrasting to previous research [7], we found a small to negligible correlation between co-change dependency metrics and defect density. A possible reason for this discrepancy is that previous research works ground their conclusions using a smaller set of systems.

RQ2 Could one improve the quality of the design of a system by reorganizing the source-code in order to keep elements that frequently change together?

In order to answer this question we proposed a method[1] to discover opportunities of redesigning a system and thus to improve its quality attributes. The method outline is as follows:

1. we mine the change history from the VCS to discover the co-change dependencies between the modules' source-code elements;

[1] specifically, the proposed method has three variations

2. we compute clusters of source-code elements based on the principle of low coupling and high cohesion considering the co-change dependencies between them;

3. we detect *evolutionary smells* that occurs when we have two source-code elements from different modules belonging to the same cluster but that do not have any kind of dependency upon another element from the same module;

4. we propose recommendations of moving source-code elements between modules in order to reduce co-change dependencies between them, i.e., we aim to remove evolutionary smells.

Results.

We evaluated our approach using 47 open-source systems and found 8,405 evolutionary smells on these systems, and 4,844 refactoring recommendations. After applying the recommended refactorings, we found that our approach improves the design of the system (considering software design quality metrics) and outperforms state-of-the-art refactoring recommendation tools.

RQ3 How do practitioners perceive the benefits of applying our method in comparison with other refactoring recommendation tools?

While the RQ2 has a quantitative nature, we also investigated qualitatively what are the perceptions of the developers about the the recommended refactorings.

We applied four refactoring recommendation tools (JDeodorant, REsolution, JMove, and ours) to the source-code of three proprietary software systems. These tools provided over 500 refactorings recommendations. We then curated a list of 145 recommendations, and asked the software developers of the systems to assess these recommendations.

Results. We observed that recommendations from JDeodorant and Draco were more positively evaluated than the recommendations from REsolution. For instance, the software developers perceived improvements on flexibility in only 14% of the REsolution recommendations (for our approach and JDeodorant the results were 37% and 43% respectively). Moreover, JDeodorant was also the one with the highest acceptance ratio (62% of the refactorings recommended by JDeodorant were accepted and integrated into the systems, the acceptance ratio for our approach and REsolution were 38% and 24% respectively). JMove was able to recommend refactorings for one system only. We also find that our approach is suitable to complement other refactoring recommendation tools, because its recommendations are not produced by any other studied tool. Finally, the accepted recommendations from our approach demonstrate its feasibility, and we also found that some of the rejected recommendations started discussions about design flaws and its alternative solutions.

1.2 Publications

The work associated with this thesis resulted in the following peer-reviewed publications.

1. Amaral, L., Oliveira, M. C., Luz, W., Fortes, J., Bonifácio, R., Alencar, D., Monteiro, E., Pinto, G., and Lo, D. (2020, September). How (Not) to Find Bugs: The Interplay Between Merge Conflicts, Co-Changes, and Bugs. In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 441-452). IEEE.

   This publication corresponds to the Chapter 2 and addresses the first research question of this thesis. While the paper also reports an investigation on the correlation between merge conflicts and bugs, we edited Chapter 2 to only contains material related to co-change and bugs.

2. Oliveira, M. C., Freitas, D., Bonifácio, R., Pinto, G., and Lo, D. (2019, September). Finding needles in a haystack: Leveraging co-change dependencies to recommend refactorings. The Journal of Systems and Software (JSS). Elsevier.

   This publication corresponds to Chapters 3 and 4, which presents our approach (Chapter 3) and answers the second and third research questions (Chapter 4). In this thesis, we improved several sections of the original publication, including new algorithms and the qualitative study.

3. Tarchetti, A. P. M., Amaral, L., Oliveira, M. C., Bonifácio, R., Pinto, G., and Lo, D. (2020, September). DCT: An Scalable Multi-Objective Module Clustering Tool. In 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM) (pp. 171-176). IEEE.

   This publication corresponds to Supplement I and describes our multi-objective software clustering tool that I have implemented and used to compute the co-change clusters in our experiments. While a few previous multi-objective software clustering tools exist, they do not scale for large graphs built from fine-grained co-change dependencies.

1.3 Thesis Outline

This thesis is organized as follows:

- Chapter 2 reports an investigation on the correlation between co-change metrics and bug density.

- Chapter 3 describes our method, and its variations, of recommending refactorings to remove *evolutionary smells*, which arises when two methods/fields from different classes frequently change together.
- Chapter 4 presents the evaluation of our method, which comprises of a quantitative study, with 47 open-source systems, and a qualitative study with one large and two medium size systems.
- Chapter 5 concludes this thesis.
- Supplement I presents our multi-objective Software Module Clustering tool, which we use the NSGA-II algorithm to find an adequate system decomposition. Our implementation aims to optimize memory and CPU when compared with existing tools, specially for large graphs.

Chapter 2

How (Not) to Find Bugs: The Interplay Between Co-Changes and Bugs

Abstract

Context: In a seminal work, Ball et al. [10] investigate if the information available in version control systems could be used to predict defect density, arguing that practitioners and researchers could better understand errors "*if [our] version control system could talk*". In the meanwhile, several research works diverge about the correlation between co-change dependencies and defect density. *Problem:* The correlation between co-change dependencies and bug density has been only investigated using a small number of case studies—which can compromise the generalization of the results. *Goal:* To address this gap in the literature, this chapter presents the results of a comprehensive study whose goal is to understand whether or not co-change dependencies are good predictors for bug density. *Method:* We first build a curated dataset comprising the source code history of 29 popular Java Apache projects and leverage the SZZ algorithm to collect the sets of bug-fixing commits. We then combine the SZZ results with the set of co-change dependencies of the projects. Finally, we use exploratory data analysis and machine learning models to understand the strength of the correlation between co-change dependencies with defect density. *Findings:* there is a negligible to a small correlation between co-change dependencies and defect density—contradicting previous studies in the literature.

Keywords: Refactoring, co-change dependencies, remodularization, clustering, architecture quality

2.1 Introduction

Software teams spend a significant amount of time trying to locate defects and fixing bugs [11]. Actually, fixing a bug involves isolating the part of the code that causes an unexpected behavior of the program and changing it to correct the error [12, 13]. This is a challenging task, and developers often spend more time fixing bugs and making the code more maintainable than developing new features [14–16].

To mitigate the time spent fixing bugs, it is crucial to better understand the development practices and the properties of the systems that are more likely to introduce bugs.

Existing research works have investigated the correlation between structural properties of the systems (such as object-oriented metrics) and defect density [17–19]. Researchers have also investigated whether the complexity of code changes could be used to estimate the incidence of bugs in software assets [20, 21]; while others have leveraged information available in version control systems (VCSs) either to (a) characterize the properties of changes that may introduce bugs [22] or to (b) investigate if co-change metrics are good predictors for defect density [7, 10, 23, 24].

Although some studies investigated the characteristics of bug-introducing changes (e.g., [22, 25]), there are many other categories of these changes that have not been explored before. Therefore, exploring specific categories of bug-introducing changes is essential to aid developers in avoiding bugs. In this chapter we explore a potential category of bug-introducing changes: co-changes dependencies. While previous research works have investigated the relation between co-change dependencies metrics and defect density, the conclusions have been drawn from a small number of samples and are inconclusive—some works claim that co-change dependencies might be used to predict defects [7], while others claim the contrary [10]. The lack of a general understanding of this aspect brings the general research question we address in this chapter:

RQ1 *To what extent co-change dependencies metrics correlate to defect density?* Answering this research question is important because it could reveal a negative side of a system decomposition that leads to co-change dependencies, either confirming or refuting results of previous studies [7, 8, 26].

To investigate this research question, we first mine the source code history of a curated dataset comprising 29 popular Java Apache projects hosted on GitHub. We then leverage the SZZ algorithm to identify the bug-fixing changes (BFCs). We relate the outcomes of the SZZ algorithm with the information about co-change dependencies. Finally, we use statistical methods to answer the research question.

Contrasting to previous research [7], we did not find evidence that co-change dependency metrics are good predictors for defect density.

2.2 Related Work

In this chapter, we leverage the SZZ algorithm to investigate to what extent co-change dependency metrics relate to defect density. Because our work investigates whether co-changes relate to bugs, we surveyed the related research regarding co-changes.

2.2.1 Research on Co-change Dependencies

Ball et al. [10] present one of the first research works that explore the use of co-change dependencies (a.k.a, change coupling or logic coupling) to analyze the structure of systems. In fact, the research on co-change dependencies have focused on getting new insights about the structure of systems [24, 27–29] and finding opportunities to rethink architectural decisions [30, 31]. For instance, Beyer and Noack [27] use information from version control systems to build a graph from assets that frequently change together. The goal is to find clusters in this graph that correspond to *subsystem candidates*. Other research works focus on the interplay between structural dependencies and co-change dependencies [24, 29], highlighting that there is no linear correlation between these types of dependencies—classes that are statically dependent do not necessarily change together. Other research works find opportunities to change the decomposition of the systems using co-change dependencies [30, 31].

Besides reasoning about the structure of the systems, other research works investigate the relationship between co-change dependencies and defect density. However, we could not find a consensus about this topic. Some findings reported in the literature [26, 32] claim that there is no correlation between highly co-change coupled assets (such as files or classes) and the bug incidence in these assets (i.e., frequency that these assets change due to bug fixes). For instance, Knab et al. [26] use *decision trees* to find rules that can be used to predict defect density. Using data extracted from the Mozilla Web Browser source code history, the authors conclude that "*change couplings are of little value for the prediction of defect density*" [26].

Contrasting, other research works [7–9] suggest that there is a correlation between co-change dependency metrics (such as the number of co-dependent classes of a given class) and bug density. For instance, D'Ambros et al. [7] present the results of an empirical study using three open source systems (ArgoUML, JDT Core, and Mylyn). The authors investigate the correlation between five *change coupling metrics* and the number of bugs of the components (Java classes)—reporting a moderate to a high correlation between change coupling metrics and defects. Other research works explored the same question, though using a small number of systems [8, 9], and concluded that co-change dependencies could be used to predict defect density.

Our research question investigates whether or not co-change dependency metrics correlate to defect density. Nonetheless, differently from previous studies [7–9] that draw conclusions from one or two systems, here we consider a curated dataset with the source code history from a set of 29 Apache open source systems, increasing the generalization of the results.

2.2.2 The SZZ algorithm

The SZZ algorithm was introduced by Śliwerski et al. [22], to identify the potential change-sets (commits) responsible for fixing defects. It is a well-known algorithm, being widely used in the Just-in-Time Defect Prediction research agenda. Rodríguez-Pérez et al. [33] present the results of a literature review, assessing 187 papers that made use of the SZZ algorithm to evaluate the reproducibility and credibility of these publications in Empirical Software Engineering.

Several limitations of the SZZ algorithm have been reported, including technical (e.g., mislabeled changes) and methodological ones (e.g., difficulty to reproduce the studies). For instance, the first SZZ [22] variant has several problems. In particular, it considers cosmetic changes (as indentation, blank lines, and comments) as possible bug-introducing commits. Nonetheless, cosmetic changes do not modify the software behavior.

To deal with the technical limitations of the original SZZ design, researchers developed new variants of the SZZ algorithm [25, 34, 35], in order to reduce noise. When considering the first phase of the algorithm (finding bug-fixing commits), the limitation relies on how bug reports are linked to commits, i.e., if the bug fix is not identified, the bug commit cannot be determined, causing a false negative. False-positive happens when a bug report does not describe a real bug, but a fixing commit is linked to it. As reported by early studies 33.8% [36] to 40% [37] of the bugs in issue tracking system are miss-classified.

The second part of the algorithm, which is concerned with identifying the bug-introducing commits, can also produce false positives and negatives. Addressing these limitations requires a manual and tedious validation process [33], and Costa et al. [25] proposed a framework to evaluate and compare different implementations of SZZ.

Neto et al. [35] showed that discarding cosmetic changes and refactoring contributions improve the precision of the second phase of the original SZZ, from 37% using their `RA-SZZ` implementation to 97% using the `RA-SZZ`* variant. Moreover, `RA-SZZ`* outperforms another recent SZZ implementation (MA-SZZ [25]). After experimenting with other implementations, and reading these results in the literature, we decided to use `RA-SZZ`* in our research.

2.3 Study Settings

In this section, we present the settings of our study, whose main goal is to investigate whether commits that lead to co-change dependencies relate to bugs. As such, we answer the research question we introduce in Section 2.1.

2.3.1 Project Selection

Our procedures for project selection consider the existence of tools for mining bug fixing commits and tools that we could use to identify non-cosmetic changes (e.g., changes that go beyond adding a comment of a piece of code), and compute co-change dependencies. To mine bug-fixing commits, we leverage in our research the RA-SZZ* [35] tool—a refactoring aware implementation of the SZZ algorithm. RA-SZZ* collects project information from a git source-code repository and from a JIRA database with the project issues. RA-SZZ* then populates a relational database with all necessary information to find bug-fixing commits, taking into account refactoring and cosmetic changes. The decision of using RA-SZZ* led us to consider the Apache community as an initial project population, since a set of Apache projects use JIRA as an issue management system, and developers of Apache projects often link code contributions to the JIRA issues—a requirement for improving the performance of RA-SZZ*. By mining from Apache we are controlling for the quality of our dataset as we are much less likely to perform our study on unrepresentative projects.

We then focused on Apache Java projects, due to the availability of tools to co-change dependencies [31]. Furthermore, following existing recommendations for mining GitHub repositories [38], we include the number of stars as a measurement of popularity. As a result, we selected Apache Java projects hosted on GitHub having more than 200 stars. Applying this filter on the Apache GitHub organization revealed 101 repositories, which we considered as our initial dataset. This initial dataset includes projects with different characteristics, from medium size libraries and web frameworks (e.g., Struts and Wicket) to full-fledged textual search engines and database systems (e.g., Lucene and Cassandra).

2.3.2 Finding Bug-fixing commits

We mined software repositories to detect bug-fixing commits (BFCs) from the source-code history of the selected projects. To this end, we leveraged the RA-SZZ* [35] tool to identify BFCs. The main reason that support our choice of using RA-SZZ* is the previous results in the literature, that show that RA-SZZ* outperforms other implementations [25, 35].

In this section, we use the Apache Nifi project as a running example to describe our methodology. Apache Nifi is hosted on GitHub and uses JIRA as the issue tracking system (as all instances of our initial project population). We follow the steps below to mine the bug fixing commits:

(S1) **Fetch bug issues:** The first step is to collect bug issues from JIRA, using its REST API, and filtering the issues using the issue type = **bug**, the status = (**resolved** or **closed**), and the resolution = **fixed**. As an output, we collected 1988 issues from Apache Nifi.

(S2) **Clone the project:** The second step is to clone the project repository locally to get its source-code history.

(S3) **Find Bug Fixes:** The third step is to use the resulting files from previous steps to link bug-fixing commits (BFCs) to *issues*. In this case, it is necessary to specify how a bug fix should mention the issue in a commit message, and then `RA-SZZ`* finds some patterns to decide whether or not a commit is a bug-fix. As a result, we obtain a file containing all BFCs. For the running example, we found 1847 bug-fixing commits, mapping 92% of the issues from JIRA to BFCs on git-log.

In summary, considering our running example, SZZ identified 1025 unique BFCs. We created additional scripts to replicate the pilot study, through running `RA-SZZ`* for the remaining 100 project repositories. After assessing the results in these repositories, we filter out several outlier projects from our analysis, as we discuss in Section 2.4.1.

2.3.3 Computing Co-change Dependencies

To answer our research question, we first have to compute the co-change dependencies of the systems. A co-change dependency arises when two source-code entities, such as classes, interfaces, methods, or fields, frequently change together. We compute co-change dependencies using the source-code history of the systems. Popular VCSs such as `git` and Subversion maintain the evolution of source-code artifacts (typically files), and the history of changes can be described as a sequence of commits $H = (c_1, c_2, \ldots, c_n)$, where each commit refers to a subset of artifacts in the form $c_i \subseteq A$. From this sequence of commits, we can build a (di)graph whose vertexes correspond to the source-code entities of a system and whose edges correspond to the co-change dependencies. Although it is possible to compute co-change dependencies for finer-grained entities [31], in this study we focus on coarse-grained entities (i.e., the vertexes of our graph correspond to Java classes). The reason for using coarse-grained entities in this study is to make possible to compare our results with other studies, that use co-change metrics based on classes. Nevertheless, we can consider that when two fine-grained elements are co-change dependent the classes that contains these elements are (usually) co-change dependent too, the results from this study could be a good proxy for fine-grained entities.

Like other studies [39–42], 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 might differ from the confidence between e_b and e_a. We consider that e_a and e_b change frequently together if their support count and confidence are above the threshold for support 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., [28, 30]).

We also compute two additional metrics [7] from the co-change dependencies: Number of Coupled Classes (NOCC) and Sum of Class Coupling (SOCC). The first computes the *number of classes n-coupled with a given class*—where n specifies a dependency threshold corresponding to the minimum number of changes between two components. The second is the *sum of the shared transactions* (commits) *between a given class c and all the classes n-coupled with c*. Accordingly, SOCC considers the strength of the coupling between the two components. Finally, we use statistical methods (hypothesis testing and regression analysis) to estimate the strength of the relationship between these metrics and metrics that estimate how a given component is prone to bugs.

2.4 Results

In this section we present the results of our empirical study. We first report the outcomes of an exploratory data analysis, and then we answer our research question using statistical methods (either hypothesis testing or regressions models).

2.4.1 Data Description

To achieve the general goal of the original paper—that also investigated the correlation between merge scenarios and bugs—we conducted an exploratory data analysis to get a general understanding about the frequency of merge scenarios and conflicting merge scenarios, as well as to refine and build curated datasets we use to answer our research questions. To curate our dataset, we removed projects that neither have merge scenarios nor conflicting merge scenarios. Interesting, in nine projects, we did not find any merge commit (e.g., Commons-IO). Although we do not investigate this issue in details, we conjecture that some projects employ alternative procedures to integrate software changes (e.g., rebase). Furthermore, we eliminated projects that do not have at least 26 (first quartile) merge scenarios and filtered out projects in which it was not possible to collect at least 200 (first quartile) closed bug-issues, to guarantee that we would have linked a representative number of issues to bug-introducing commits.

Altogether, our curated dataset, which is the intersection of the outcomes generated by our two procedures (see Sections 2.3.2, and 2.3.3) and also the procedure described above, contains information about 29 Java Apache projects. The average number of issues

and bug-fixing commits per project is 2445 and 1911, respectively. For instance, we have mined 15 465 closed bug issues and linked 14 333 bug-fixing commits in Apache Ambari; while we got only 158 bug-fixing commits for 203 closed bug issues collected from JIRA in Apache Fineract. Figure 2.1 shows a histogram that considers the rate of bug-fixing commits over the number of issues per project. Overall, the first phase of RA-SZZ* linked 78.16% of the issues to bug-fixing commits. In seven projects, RA-SZZ* linked more 90% of the issues to BFCs (e.g., Zeppelin and Lucene-sorl). Nonetheless, in the Apache Cordova-Android project, RA-SZZ* linked only 508 bug-fixing commits to a total of 4709 issues (which represents 10.79%). This situation occurs because Apache Cordova-Android is a submodule of Apache Cordova, which shares the same JIRA repository with other modules. Nonetheless, in our analysis we only considered Apache Cordova-Android.

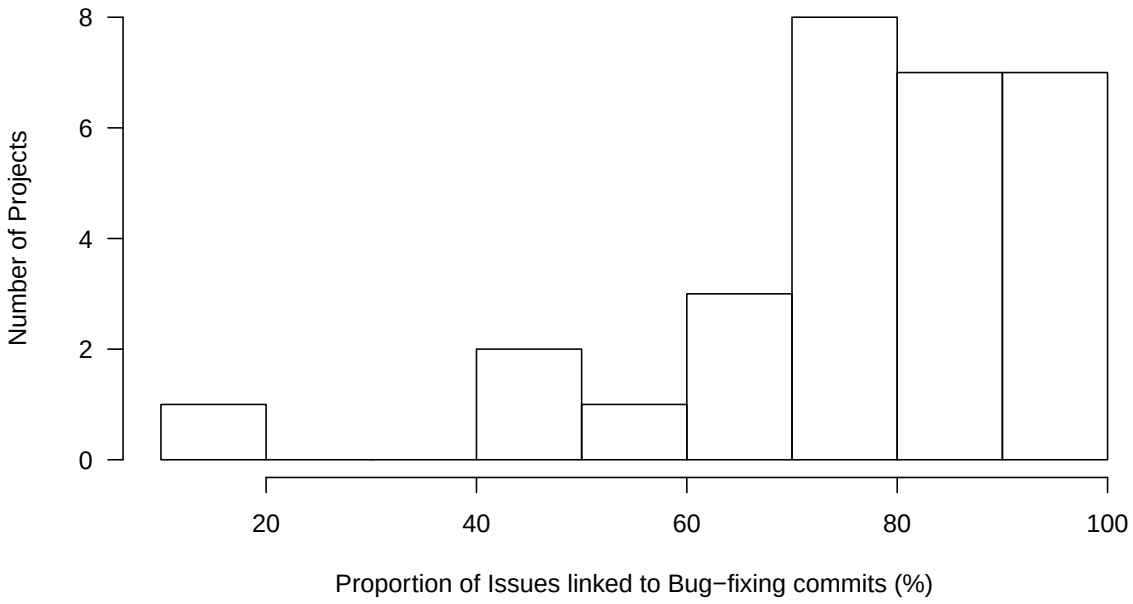


Figure 2.1: Proportion of bug issues linked to bug-fixing commits over the projects

2.4.2 To what extent co-change dependencies metrics correlate to defect density?

The goal of this research question is to investigate the relationship between bug incidence and co-change dependencies. This question has been investigated before by D'Ambros et al. [7], though using only three systems, while we collected data from 29 projects.

According to their findings, bug predictions models can be improved when considering co-change dependencies (change-coupling in the previous work).

To answer this research question, we first use the change history of the systems to compute the co-change dependencies between software components (see Section 2.3.3)—at the coarse-grained level only (i.e., files and classes). From the co-change dependencies, we compute two additional metrics, similarly to the work by D'Ambros et al. [7]: Number of Coupled Classes (NOCC) and Sum of Class Coupling (SOCC), using $n = 2$ as threshold—which showed the best performance in the previous work [7]. We use three datasets in this analysis. The first dataset contains the co-change data, consisting of observations with the name and the metrics NOCC and SOCC of the components. The second dataset contains the change history of all components—each row indicating that a commit changed a given component. The third dataset contains all bug-fixing commits of the systems, which we compute using the first phase of `RA-SZZ`*. We then merge these datasets and compute the number of non bug-fixing (NBC) and bug-fixing commits (BC) of every component. After that, we estimate the buggy ratio (Br) of a component c using Eq. (1).

$$Br(c) = \frac{BC(c)}{NBC(c) + BC(c)} \tag{2.1}$$

We use the Spearman correlation and simple linear regression analysis to estimate the strength of the relationships between NOCC and SOCC with the buggy ratio and the total number of bug-fixing commits of a component. Simple linear regression allows us to (a) investigate if there is a relationship between NOCC and SOCC with the defect density of the components (buggy ratio and number of bug-fixing commits) and also (b) explain how strong the relationship between these features and defect density are [43].

Table 2.1 shows some descriptive statistics from the co-change metrics observations. Interestingly, considering our final dataset, most of the observations rely on the interval from four to 25 co-change dependencies (first and third quartiles, respectively)—although we found a specific component with 1022 co-change dependencies. Since these unusual cases increase the mean value of NOCC and SOCC, we decided to remove the components having either NOCC > 25 or SOCC > 75 from our dataset.

Table 2.1: Descriptive statistics for the NOCC and SOCC

Metric	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
NOCC	1	4	11	19.40	25	1022
SOCC	2	10	30	68.04	75	5984

Figure 2.2 shows a matrix of correlation for the metrics NOCC, SOCC, BFCs (number of bug-fixing commits), and Ratio (Buggy Ratio). In our research, different from the work by D'Ambros et al. [7], we found a small correlation between the number of bug-fixing commits and the metrics NOCC and SOCC. This might contradict their findings and suggest that co-change dependencies are not effective predictors for defect density. In addition, a correlation between NOCC and SOCC with the total number of bug-fixing commits might actually suggest that NOCC and SOCC correlate to the total number of commits of a component—something that is expected. That is, considering only the total of bug-fixing commits might mislead the conclusions, since there is a difference in the error proneness of a given component A with three bug-fixing commits and 10 non-bug-fixing commits (a buggy ratio of 23%) and another component B with the same number of bug-fixing commits and 20 non-bug-fixing commits (a buggy ratio of 13%). Previous work only consider the absolute value of number of bug-fixing commits. Accordingly, in Figure 2.2, we find a negligible correlation between the metrics NOCC and SOCC with the Buggy Ratio of a component, which might better characterize defect density.

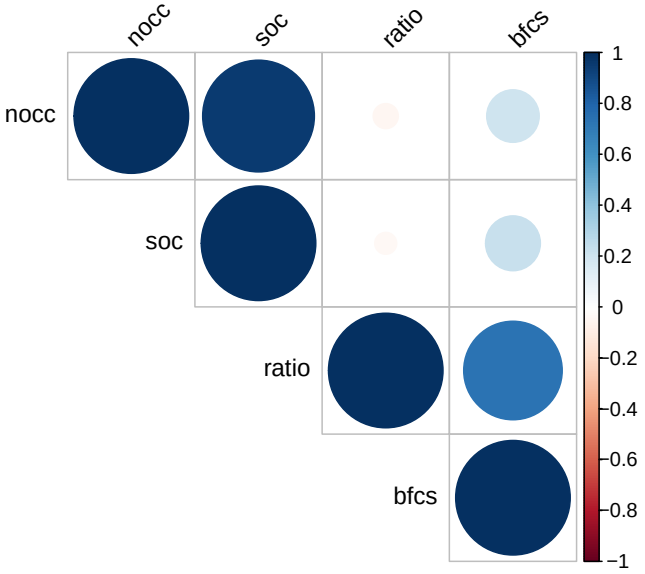


Figure 2.2: Correlation matrix between the metrics NOCC, SOCC, BFCs (number of bug-fixing commits), and Ratio (Buggy Ratio)

We use linear regression models (m1.: $Ratio \approx \beta_1 \times NOCC + \beta_0$ and m2.: $Ratio \approx \beta_1 \times SOCC + \beta_0$) to investigate if we could predict *buggy ratio* using the metrics NOCC and SOCC. Although the p-values for both models suggest that exist associations between these predictors and the *buggy ratio*, the *adjusted* R^2 is also close to zero (for both models),

supporting our findings that one cannot truly explain the buggy ratio variation in terms of NOCC and SOCC.

> Altogether, we argue that components with high co-change dependencies are not more subject to defect density than other components. This result contradicts previous studies [7–9] that claim that co-change dependencies metrics are good predictors to bug introducing changes.

We also replicate the correlation analysis to all individual projects, considering the total number of bug-fixing commits of a component and the metrics NOCC and SOCC. In more than 80% of the projects, we found either a small (< 0.5) or a negligible correlation (< 0.3) for both metrics.

2.5 Discussion and Threats to Validity

2.5.1 Discussion

Altogether, we show evidence that co-change dependency metrics do not correlate with defect density, contrasting with the findings of previous studies [7–9]. A possible reason for this discrepancy is that previous research works ground their conclusions using a smaller set of systems. Another reason, can be that the set of systems selected for this study have a superior design quality than the systems used on previous studies—i.e., they are less error-prone than the average systems given their superior quality. Due to these conflicting results, we argue that further research should be conducted, either to confirm or refute our findings that co-change dependencies might not be efficient for predicting bugs.

2.5.2 Threats to Validity

We leverage the SZZ framework to identify bug-fixing commits as well as to trace the source of those bugs. Although, SZZ has known limitations, as we present in Section 2.2, some of our study procedures mitigate part of its usage threats.

We highlight that considering our curated dataset, 75% of the issues reported on JIRA were linked to bug-fixing commits in the first phase of SZZ. This number actually supports the use of SZZ in our research, mitigating part of the critics about the use of SZZ. However, the set of bugs fixed in a source code repository over time might go beyond those handled by SZZ. For instance, a bug can be introduced and fixed before it is even reported in a bug tracker tool. In addition, some contributions might have not been correctly linked by developers with the bug issues associated (using the commit message). In spite of that, we also performed manual validations to mitigate reliability issues in our results.

Trying to improve the quality of our datasets, we removed several projects that we initially collected data from the repositories. For instance, we established a minimum number of 200 issues (1st quartile) on the JIRA issue database for each project. As a final criteria, we excluded projects in which `RA-SZZ`* linked a small number of issues to bug-fixing commits. Therefore, we reduced the number of systems in our corpus, which might compromise the external validity of our study. However, we believe that this decision would not change our findings, because we still collected a reasonable number of issues, and more than 75% of them were linked to bug-fixing commits.

To investigate the research question, we had to estimate the number of bugs related to each component (i.e., Java classes). As such, we identified the commits that (a) affect a given class and (b) that also relate to bug fixes in the commit message. To this end, we leverage the first phase of the SZZ algorithm only. Although previous research works did not use the SZZ algorithm [7–9], we believe that our methodologies are quite similar (since previous works also associate commits to the issues databases). Therefore, the divergence of our findings is not fully explained by our decision to use the first phase of SZZ. Instead, this divergence is more likely to occur due to our larger dataset of projects.

Nonetheless, despite using a curated dataset, we still believe that we cannot generalize our results to scenarios that do not explore the development practices of open source projects and use languages different than Java.

2.6 Final Remarks

This chapter has investigated the correlation between co-change dependencies with defect density. We extracted 22 532 bug-fixing commits from 29 Java Apache projects.

The research presented in this chapter allowed us to investigate our first research question (RQ1: *To what extent co-change dependencies metrics correlate to defect density?*), whose answer we summarize as follow:

> Answer to **RQ1**: Contrasting to previous research [7], we found a small to negligible correlation between co-change dependency metrics and defect density.

Chapter 3

The Draco Approach: Leveraging Co-change Dependencies to Recommend Refactorings

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 chapter 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.

Keywords: Refactoring, co-change dependencies, remodularization, clustering, architecture quality

3.1 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 [44]. In practice, software design tends to decay over time—independently of how elaborate the design of the software is [45]. 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 [44]; (2) tight schedules that lead developers to take bad decisions, introducing technical debt and hindering the redesign of a software [46]; or (3) unanticipated requirements that do not fit in the original decomposition [45]. In the end, the lack of maintainability often leads to the problem of *software erosion* [47], 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 [44, 47].

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 [30, 48] or to suggest sequences of refactorings to improve the design of a software [49]. Existing approaches rely on *source-code dependencies* to recommend alternate decompositions [30, 48], considering a set of specific goals (such as minimizing coupling or maximizing cohesion) [6, 50]. 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 [28].

Several studies [7, 23, 51, 52] correlate co-change coupling with software quality problems. For instance, Zhou et al. [51] claim that co-change dependency analysis has the potential to provide early warnings of a potential design or architectural flaw. Therefore, presenting refactoring recommendations, which aim to reduce co-change dependencies, might improve the overall quality of a system. Accordingly, recent works [28, 53, 54] 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 dependency between them is hidden—and we can only reveal this dependency using a co-change dependency analysis.

In this chapter we present a novel refactoring recommendation approach (named Draco) that removes co-change dependencies between classes. The *motivation* of this approach 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,

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.

In the remaining of the chapter, we first discuss the background and related research in Section 3.2 and then present the Draco approach (and its variants) in Section 3.3. Next chapter presents comprehensive evaluation of Draco, in terms of both quantitative and qualitative studies.

3.2 Background and Related work

3.2.1 Software Decomposition and Remodularization

The concept of a software decomposition we use in this work is based on the definition of Mitchell and Mancoridis [48], 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. [10] 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. [55] introduced the use of fine-grained co-change dependencies on MDGs. Building on these previous works, in this chapter, we work with MDGs whose vertices are fine-grained entities (similar to the Zimmermann et al. approach [55]) 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 [27].

The use of software clustering as basis for software remodularization has been discussed in the literature for almost 20 years [56]. The work of Anquetil and Lethbridge [57], for instance, compares different strategies for software clustering to this specific goal. More

recently, Maqbool and Babri [58] 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. Contrasting, Silva et al. [30] 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. [59] 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 [60].

Also regarding co-change dependencies, Oliveira et al.[28] 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 should not 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.

3.2.2 Code Smells and Source Code Refactoring

Code smell (or bad smell) is a symptom of bad decisions about the system design [61]. Research studies discuss that code smells could hinder maintainability and increase fault-proneness [62], increasing the motivation to develop methods to detect and remove bad smells using program refactorings. Fowler [61] describes 22 code smells and the respective refactoring operations to remove them. Several approaches were proposed to detect bad smells in source code [63–66]. Like our work, Ratiu et al. [67] and Palomba et al. [68] also recommend the use of the source code history to detect code smells, however, their approaches aim 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 [61]. 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 studies, and for this work we are also particularly interested in the research on automated refactoring recommendation approaches.

Ouni et al. [54] proposed an approach to recommend sequences of refactoring using the multi-objective genetic algorithm [69] NSGA-II [70]. 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 the 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. [53] 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 [71] 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. [6] 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 well-known refactoring recommendation tool suit [5, 72, 73], which supports several refactorings, such as **move method** and **extract class**. JDeodorant addresses different bad smells, including *feature envy* and *god class*. The *feature envy* bad smell occurs whenever one method depends on several methods of a different class. Considering the particular goal of removing *feature envy* using the **move method** refactoring, JDeodorant uses an algorithm that computes the distance between the methods and fields of a class and the class itself. Accordingly, the designers of JDeodorant introduce a metric called *Entity Placement* that builds a ranking of the refactoring recommendations—according to their effect on the design. JDeodorant has been used as a state-of-the-art tool for assessing the performance of refactoring recommendation tools, though, to the best of our knowledge, most of these efforts have concentrated their analysis either using quantitative methods that compare the tools using metrics for estimating an internal quality of the design (such as coupling or cohesion) [74, 75] or using the opinion of subjects that were not the actual developers of the systems [76]. Tsantalis et al. present an extensive review of the body of knowledge related to JDeodorant [73].

Sales et al. propose and evaluate *JMove* [77, 78]. JMove is an Eclipse based plugin for recommending refactorings, which deals with *Feature Envy* and *Long Method* code smells—using the **move method** refactoring for Java projects. The authors of JMove argue that it is more efficient than JDeodorant, because JMove considers not only structural properties of the source code (e.g., size of methods and static dependencies), but also semantic dependencies based on the source code vocabulary. Terra et al. evaluate JMove using 10 open-source systems, in order to compare JMove with JDeodorant and inCode [78]. Their evaluation focus on precision, recall, and performance, using as ground truth methods that were randomly selected and manually moved from the original classes. The authors report that JMove requires more time to recommend refactoring, though it improves both precision and recall (when compared to both JDeodorant and inCode) [78].

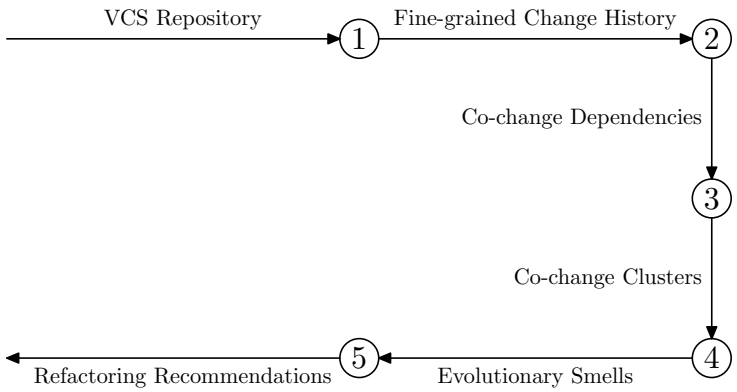
Methodbook is an approach to recommend *move method* refactorings that aims to remove *feature envy* bad smells [79]. It uses relational topic models to discover the "friends" of the methods in a system, and the class that contains the highest number of friends of the method under analysis is suggested 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.

3.3 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). The approach is composed by five steps, where the input is a VCS repository and the output is a set of refactoring recommendations. The first step produces a fine-grained change history, while the next three steps work together to detect a set of evolutionary smells. And finally the last step produces a set of refactoring recommendations that remove the evolutionary smells. Figure 3.1 shows an overview of the approach, and the following subsections detail its steps.

Figure 3.1: Overview of the Draco approach for recommending refactorings (the numbered circles represent the steps).



3.3.1 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 refers to a subset of artifacts in the form $c_i \subseteq A$. Since in this chapter 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 refers to 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

[1] `https://git-scm.com/`

[2] `https://subversion.apache.org/`

been modified. We take advantage of Kenja[3], a software utility that produces fine-grained change history from Git repositories.

3.3.2 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., [28, 30, 40, 60, 80]). 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 4.3.8.

3.3.3 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. (3.1)), proposed by Mitchell and Mancoridis [48].

$$MQ = \begin{cases} \left(\frac{1}{k}\sum_{i=1}^{k} A_i\right) - \left(\frac{1}{\frac{k(k-1)}{2}}\sum_{i,j=1}^{k} E_{i,j}\right) & \text{if } k > 1 \\ A_1 & \text{if } k = 1, \end{cases} \tag{3.1}$$

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) [69] to compute an optimal partition. The goal of a GA is to find acceptable solutions for

[3] https://github.com/niyaton/kenja

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:

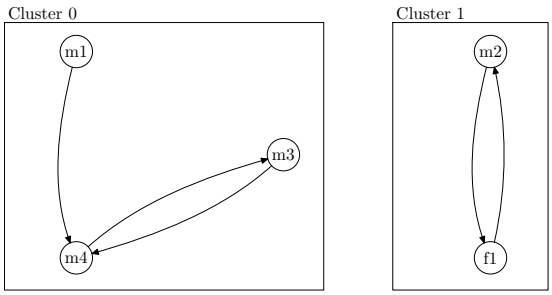
1. It first generates an initial population (i.e., a set of individuals) randomly;
2. It repeatedly produces a new population, by (a) selecting individuals from the previous population using the fitness values and (b) combining them using the genetic operators *crossover* and *mutation*;
3. It proceeds 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 [69]. 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 [69]. Accordingly, the result is a set of solutions not dominated by any other solution. This set is named the Pareto Set, while the fitness values of these solutions constitute the Pareto Front.

Similarly to a previous work [81], we used a multi-objective genetic algorithm to compute the co-change clusters (in our case the Non-dominating Sorting Genetic Algorithm—NSGA-II [70]), 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 3.2-(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 3.2-(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 [69]. 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

Figure 3.2: Individual representation



(a) A simple example of co-change clusters

0	1	2	3	4
0	1	0	0	1

(b) Corresponding *array* representation.

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 $s = b_1 b_2 \ldots b_n$, 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 [81], 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. [59]. 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:

- $PS = \begin{cases} 2n & \text{if } n \leq 300 \\ n & \text{if } 300 < n \leq 3000 \\ n/2 & \text{if } 3000 < n \leq 10000 \\ n/4 & \text{if } n > 10000 \end{cases}$

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

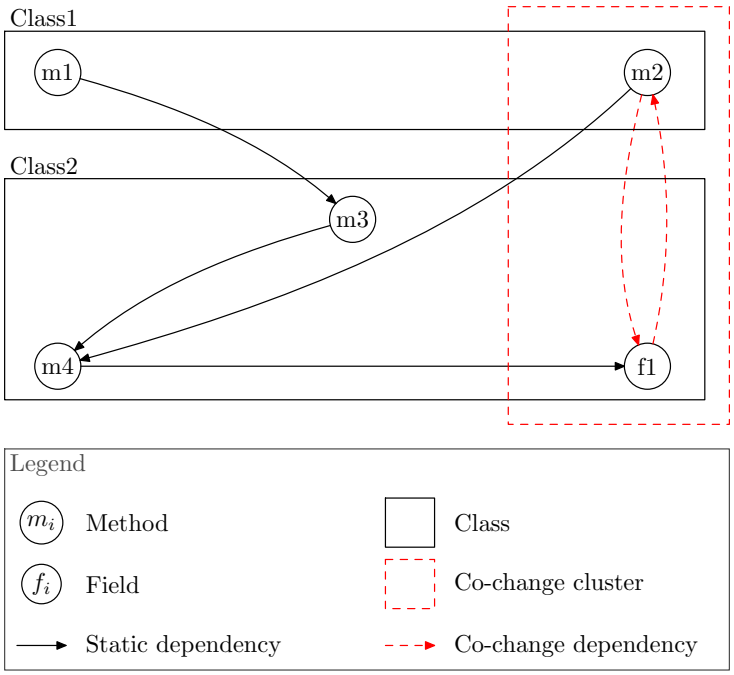
3.3.4 Discovering Evolutionary Smells

Building on Martin's Common Closure Principle [82], 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. [68]), we are not using co-change dependencies to detect well-known bad smells [61]. 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 using one of the following options:

- using the Pareto Set: when a co-change cluster from any partition belonging to the Pareto Set 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 3.3 illustrates an instance of this smell;
- using only the partition from the Pareto Set with the best *Modularization Quality*: same as previous item, however explore the clusters belonging to the best MQ partition only;
- using only co-change dependencies: Specifically, we identify these smells by looking for co-change dependencies of the form $f \rightarrow C$, where f is a fine-grained element (method or field) of a class C_f, and f is co-change dependent of some element from class C, where $C \neq C_f$; and no dependencies (static or co-change) of the form $f \rightarrow f'$ exists, where $f \neq f'$ and $f' \in C_f$. Figure 3.4 illustrates an instance of this situation.

Intuitively, when we find a situation similar to the aforementioned, we can suppose that the fine-grained source-code entity (e.g., `m2` in Figures 3.3 and 3.4) 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,

Figure 3.3: 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).



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" [61] 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.

3.3.5 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-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:

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

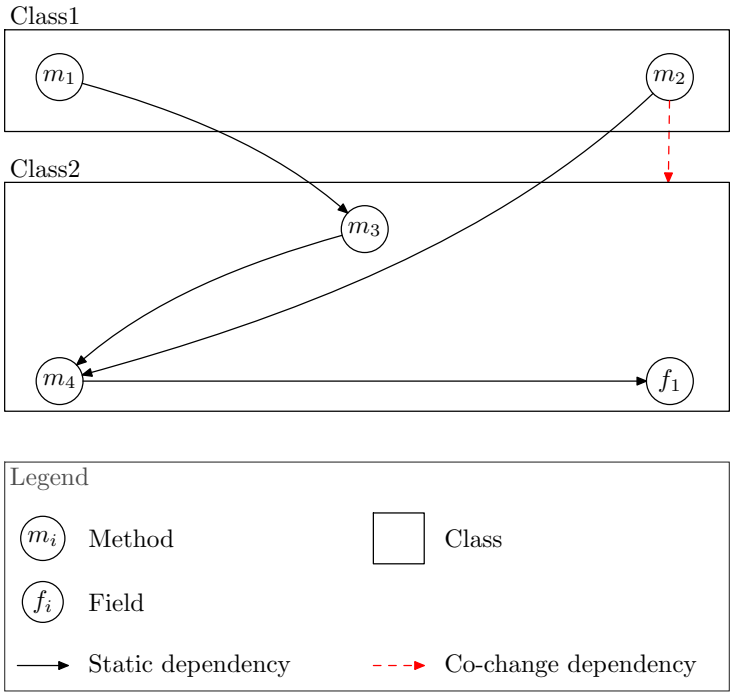


Figure 3.4: Example of an evolutionary smell involving co-change dependencies that might characterize a bad design decision. Method m_2 is co-change dependent of Class2, and does not call any other method or access any field from its own class (Class1).

- (**Constraint #1**) the total number of co-change dependencies (edges) of the MDG representing the software *after* applying the refactoring must be smaller than the number of co-change dependencies of the MDG representing the software *before* applying the refactoring, and no new static dependency is introduced unless the source and destination classes of the method/field to be moved already statically depend on each other. Bellow we present some situations where this constraint is **not** satisfied:
  - There is at least one co-change dependency 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 be co-change dependent 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 co-change 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.

Kim et al. [83] describe the precondition checks that move method refactoring engines use. While the Draco approach is not a refactoring engine, its refactoring recommendations enforce the majority of the preconditions, or offer a workaround. We further discuss this subject in Sections 4.3.9, 4.3.10, 4.4.3, and 5.3.

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.

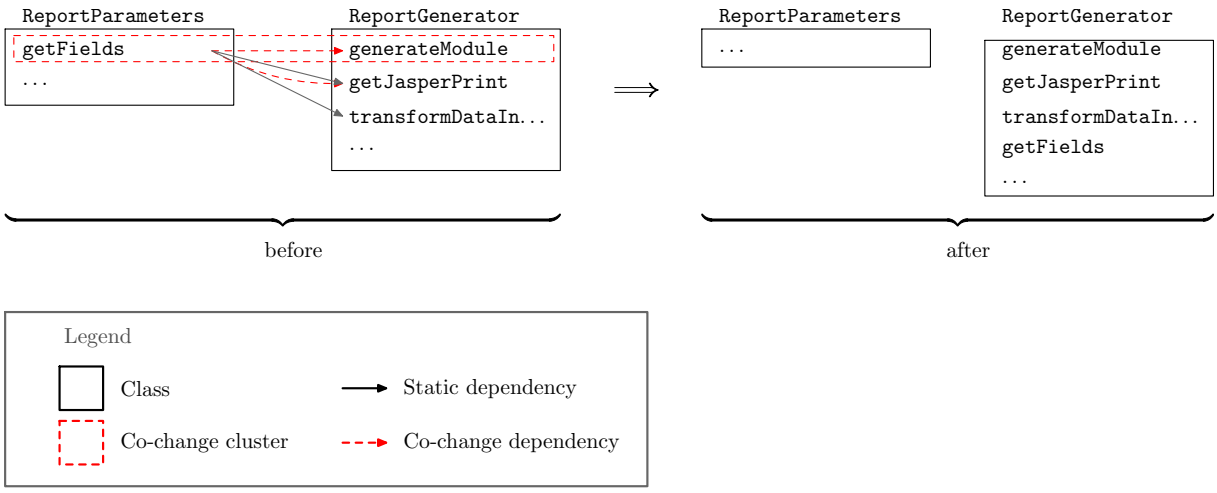
According to our decisions, if the element to be moved e has multiple destination classes available, we choose as the destination class of the refactoring the class that have the highest number of dependencies (static or co-change) with the original class of e that will be removed after applying the 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.

Figure 3.5 shows a more concrete example of a recommended refactoring computed using our approach. In this example (from an enterprise Java system called SIOP), 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.

Figure 3.5: 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

3.4 Implementation

We implemented the Draco approach as a command line interface (CLI) tool—that reads two MDG files: an MDG file with co-change dependencies and a second MDG file with static dependencies. Optionally, our tooling could also receive as input a co-change cluster DOT file[6] or a directory containing the co-change clusters DOT files representing the Pareto Set. Using these inputs, our implementation outputs a list of recommendations, where each of them contains the full path of the method/field and the destination class.

[6] `https://graphviz.org/doc/info/lang.html`

We implemented Draco using the Go programming language, and made it publicly available[7]. Some sample invocations of the tool look like this:

```
$ recommender <static mdg file> <co-change mdg file>
$ recommender --dot-file <co-change dot file> <static mdg> <co-change mdg>
$ recommender --dot-dir <dir of co-change dot files> <static mdg> <co-change mdg>
```

We produce the input files using another publicly available tools we have implemented, specifically:

- Co-change MDG files: we use the co-change miner tool[8], that reads a fine-grained Git repository produced by the Kenja tool as input;
- Static MDG files: we use the static dependencies collector tool[9], that reads a source-code root folder as input;
- DOT files: we use the Draco Clustering Tool (DCT), which is detailed in the Supplement I, that reads a co-change MDG file as input.

We chose the Go as programming language because it produces programs compiled ahead of time to native machine code, therefore the compiled programs can execute efficiently, considering both memory and CPU usage. Furthermore, this property makes the use of CLI tools more convenient, since they would not require a virtual machine to run. While a Graphical User Interface potentially could be more intuitive, it makes experiments automation more difficult, when compared with CLI tools.

3.5 Conclusion

In this chapter 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 [59]. 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.

[7] https://github.com/project-draco/tools
[8] https://github.com/project-draco/tools/tree/master/mining/co-change
[9] https://github.com/project-draco/static-dependencies-collector

Chapter 4

The Draco Approach Evaluation

Abstract

In the previous chapter, we presented our approach (Draco) that uses co-change information to recommend refactorings that remove the so called evolutionary smells. The goal of this chapter is to report the results of two empirical assessments. In the first assessment, we quantitatively evaluate our approach using 47 open-source Java projects, finding 8,405 evolutionary smells. Our approach automatically computes 4,844 refactoring recommendations that *break* co-change dependencies. The results of the first assessment show that our approach outperforms existing approaches for recommending refactorings when dealing with co-change dependencies. In the second assessment we investigate the practitioners' perceptions about how useful refactoring recommendations from co-change dependencies are, and how our approach compares to other alternatives. To this end, we conduct an empirical investigation on the practical usage of four automated tools that recommend refactoring (Draco, JDeodorant, JMove, and REsolution). We first executed the refactoring recommendation tools in the latest version of one large and two medium size proprietary Java Enterprise Systems and then asked the developers of the systems to review the recommendations. The results of the second assessment reveal that the participants would be intended to integrate 49% of the recommendations (including the recommendations from Draco) into the systems, which indicates that refactoring recommendation tools are effective in identifying opportunities for moving methods in industrial settings. In particular, Draco presents a performance comparable with the other tools. This qualitative study also reveal that JDeodorant, REsolution, JMove, and Draco do not consider the design constraints of the systems—while recommending a transformation, which is the main reason that led the participants of our study to reject 51% of the transformations.

Keywords: Refactoring, co-change dependencies, remodularization, clustering, architecture quality

4.1 Introduction

The principle that software decays over time is not something new. Parnas coined the term "software aging" in the 90s [44], but literature from the 70s [84] suggest that this problem was already present in the daily life of software developers long before. Unfortunately, the problem of software aging is still alive and well. There are many causes that have the potential of eroding a software system, including the number of software contributors with different skills [85], the turnover of software developers [86], and the presence of technical debt [87] (such as delayed refactoring [88]). To make the matter worse, software engineers, who often operate under tight schedules, have little chance and incentives to put effort on improving the internal quality of a software system.

In this grim scenario, tools that could help developers to rejuvenate the internal codebase of a software system are highly demanded [89]. As a consequence of decades of software system being thrown away, it comes as no surprise that researchers have been dedicating several efforts to introduce methods, techniques, and tools to identify refactoring opportunities, to mitigate aging, or to rejuvenate software. In particular, refactoring research has been flourishing in the last decade [90], resulting in tools that target different design concerns. Just to exemplify a few tools, JDeodorant (and JMove) recommend refactorings that aim to remove the *Feature Envy* bad smell [5, 73], REsolution recommend refactorings that aim to minimize coupling and maximize cohesion [6], and Draco recommend refactorings that aim to remove dependencies that arise between entities (e.g., classes or methods) that frequently change together [31]. The Draco approach seems to have particular applicability because existing research suggests that those so-called co-change dependencies lead to design flaws [7, 23, 28, 51, 52].

In this chapter, we report the results of two empirical assessments. The first is a quantitative assessment using 47 open-source Java projects. We found a total of 8,405 evolutionary smells in all projects. We automatically computed 4,844 recommendations of move method/field refactorings. All the recommendations lead to design improvements (according to well-known metrics), without introducing any new static dependency.

In the second assessment we explored how practitioners evaluate refactoring recommendations that aim to remove co-change dependencies, as well as how Draco compares to other approaches. In summary, the main goal of this second assessment is to present a qualitative assessment of Draco, that investigates the practical implications of using four refactoring recommendation tools (JDeodorant, REsolution, JMove, Draco) in industrial settings. Using the refactoring recommendations from these tools, we explore whether developers would consider their outcomes relevant to improve the design of a large scale and two medium size Java Enterprise systems, and whether the developers would intend to integrate the refactoring recommendations into the systems.

These tree systems came from two different organizations. The large system (named SIOP), belongs to the Brazilian Ministry of Economy, while the two medium size systems belong to Brazilian Army. Accordingly, we divided the qualitative assessment in two, one for each organization.

For the qualitative assessment 1, we executed the four aforementioned tools in one of SIOP specific revisions, generating a curated list of recommendations. After that, we started a process of code review, asking the current software developers of SIOP to evaluate the recommendations in terms of their relevance for improving the design of SIOP and their chance of acceptance. To assess the developers perception, we conducted a survey and a focus group discussion. The goal of the survey was to understand the benefits/limitations of the refactoring recommendations, while the aim of the focus group was broader: to understand the reasons behind the recommendation's acceptance/rejection, as well as possible points for improving the tools.

For the qualitative assessment 2, we conducted a qualitative assessment of Draco, together with the other three aforementioned refactoring recommendation tools. In this qualitative 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.

Our research produced a set of findings. For instance, the REsolution approach led to several recommendations of moving methods involving (the source and target) classes with different responsibilities. We wonder whether it would be possible to improve these tools' effectiveness by considering the architectural constraints or a set of previous manual refactorings of the systems. Our findings also give evidence that developers could benefit from using distinct approaches for refactoring recommendations, since it seems that each approach complements each other: we found just a few identical recommendations from JDeodorant and REsolution.

Altogether, the main contributions of this chapter are:

- An extensive quantitative evaluation on over 47 non-trivial open-source projects showing the benefits of the proposed approach. We also compared our our approach with two state of the art approaches for refactoring recommendation [6].
- A qualitative assessment of the application of four different refactoring recommendation tools (Draco, JDeodorant, JMove, and REsolution) in an industrial settings.
- A list of lessons learned that can aid researchers to further develop refactoring recommendation tools and practitioners to consider integrate the kind of tool in their daily activities.

- 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.

Differently from previous works, we evaluate different tools with different goals when recommending refactorings, allowing us to compare the usefulness of different approaches, instead of investigating the performance of different tools that share the same goal (e.g., to remove *feature envy* code smell). During our research, our goal was to use all the aforementioned tools, though focusing on the **move method** and **move field** refactorings, because this is a kind of refactoring that the developers frequently use [91, 92], and that solve many kind of design problems, as seen before. Regarding the qualitative evaluation, our work differs from the literature in at least three dimensions: First, we experimented with four different refactoring recommendation tools in a large and two medium sized industrial system. Although other works have qualitatively evaluated these refactoring tools in open source projects (e.g., [78]), we believe that our focus on an industrial system enrich the understanding of the usefulness of these tools (e.g., industrial systems do not need to attract new contributors, and then may be less interested in accepting automatic generated patches). Second, the developers that participate in our study are well experienced professional Java developers. Although other works heard the opinions from the inside developers (e.g., [79]), they often rely on students, which may not be representative of the global software industry. Third, we evaluate Draco [31], which was the first refactoring tool based on co-change dependencies.

4.2 Research Questions

In this chapter we aim to answer the second and the third research questions stated in the introduction (Chapter 1). To answer the second research question stated in the introduction, we conducted an quantitative study 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.

Another goal of this research is to better understand how useful refactoring recommendations that aim to remove co-change dependencies are. Accordingly, to answer the third

research question stated in the introduction, we qualitatively assess Draco, and then investigate the practical implications of using four refactoring recommendation tools (Draco, JDeodorant, JMove, and REsolution) in industrial settings.

Based on the first goal of our empirical study (and the second question stated in the introduction), we organized this investigation with the aim of answering the following research questions, that refine the **RQ2** from introduction:

(RQ2.1) How does the Draco approach behave when improving the design quality of a system?

(RQ2.2) How does the Draco approach compare to state of the art approaches for refactoring recommendation?

(RQ2.3) What is the impact of the different thresholds when extracting co-change dependencies on the results?

To answer **RQ2.1** 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 **RQ2.2** we executed two state of the art approaches [6, 73] for recommending refactorings, and used them as a baseline when comparing with our approach, using the same metrics they used in the original published work of the first approach [6]. To answer **RQ2.3**, 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 achieve the second goal (and to answer the third question stated in the introduction), our research relies on a field study approach [93]—considering the opinion of software developers that work on an industrial software project. We investigate the following general research question:

(RQ3) What are the practitioners' perceptions about the refactoring recommendations from Draco, JDeodorant, JMove and REsolution?

This research question is qualitative in nature. Our intention is to understand the practitioners' perception regarding the automatic-generated refactoring patches. For instance, since these tools lack context, some of these patches could introduce violations in the architectural design. Then, although the patches could improve quality attributes in the source code, such as readability, they may not be accepted if practitioners believe, for

instance, that they break a rule that exists in their application domain (but the refactoring tool is not aware).

Besides, we also explored a set of metrics—such as number of recommendations each tool proposed, the number of false positives (recommendations that do not make sense, according to the practitioners), and the acceptance rate of the recommendations—while investigating. This perspective is also relevant because we want to counterbalance the idea that a refactoring recommendation tool is "good" because the tool proposes a high number of potential recommendations. We want to extend the notion of "good" with "usefulness" attributes.

To find answers to these overlooked but important research question, we conducted two qualitative assessments. We applied the first qualitative assessment in the large scale Java Enterprise system (SIOP). The first qualitative assessment comprised two studies. In the study 1, we collected refactoring recommendations reported by the JDeodorant, REsolution, and Draco tools; then asked the developers of this system to assess the recommendations. For study 2, we ran an online focus group with the developers to better understand the context in which one recommendation is useful or not.

In the second qualitative assessment, we asked software developers and architects to qualitatively analyze refactoring recommendations, considering two proprietary enterprise Java systems from the Brazilian Army (SISDOT and SISBOL). For these 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).

Convenience was the main reason for using these systems to answer **RQ3**. 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 **RQ3**, since in this kind of project, pull-requests might have to wait an excessive time to be reviewed [94, 95], or the recommendations might be rejected without an insightful explanation.

4.3 Quantitative Assessment

The goal of the quantitative assessment is to answer the three research questions (**RQ2.1**, **RQ2.2**, and **RQ2.3**), introduced in the previous section. We made an analysis based on

metrics and compared the results after applying the recommendations from Draco, from REsolution and from JDeodorant.

4.3.1 Studied Systems

We considered a number of representative open-source Java systems to investigate questions **RQ2.1**, **RQ2.2**, and **RQ2.3**. 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 [96]. This is also a common approach for selecting open source projects to investigate [97, 98]. 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 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 47 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 4.1 presents additional information about the systems we considered in this assessment.

4.3.2 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[1]. 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 [60], 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 3.3.3. Due to the

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

Table 4.1: Studied systems. Abbreviations means: BMQ — Best MQ variation; PS — Pareto Set variation; CCD – Co-change Dependencies variation; ES – number of Evolutionary Smells; RR — number of Refactoring Recommendations.

				BMQ		PS		CCD	
Index	System	KLOC	Commits	ES	RR	ES	RR	ES	RR
1	Actor Messaging platform	157	8,772	4	2	10		128	84
2	The ownCloud Android App	36	5,329	1		2	1	142	80
3	Atmosphere Event Driven Framework	41	5,748	2		3		60	50
4	Bazel build system	375	7,258	11	4	18	6	255	146
5	BigBlueButton web conferencing system	82	13,420	2	1	8	1	114	63
6	Broadleaf Commerce – Enterprise eCommerce	168	9,784	8	2	10	3	87	54
7	Buck build system	412	7,726	3	1	3	1	220	131
8	CAS - Enterprise Single Sign On	87	6,268					15	10
9	Cassandra partitioned row store	385	21,710	17	2	30	2	537	353
10	c:geo Android geocaching app	75	10,183	20	6	30	7	115	68
11	Closure Compiler	303	8,293	2		7	1	270	121
12	CoreNLP suite of core NLP tools	552	11,963	43	9	68	9	866	437
13	Deeplearning4j deep learning & linear algebra	121	5,645					56	30
14	Drools rule engine	16	10,395	8		15		137	72
15	Druid analytics data store	297	7,452	10	6	10	7	96	64
16	Elasticsearch Engine	611	24,491	10	3	16	3	333	182
17	Fabric8 microservices platform	45	13,130	3	1	4		32	22
18	FBReaderJ e-book reader	68	9,012	4	1	10	2	81	49
19	Flink stream processing framework	419	9,565	3		7		109	65
20	Gradle build tool	283	38,756	22	4	40	6	310	188
21	Grails Web Application Framework	71	17,315	6	2	9	3	125	77
22	Groovy core language	156	12,379	3		11		144	64
23	Groovy language	161	13,465	7		12		137	66
24	H2O-2 Machine Learning Platform	95	16,172			2		128	68
25	H2O-3 Machine Learning Platform	143	19,336	5	2	7	1	331	237
26	Hibernate Object-Relational Mapping	628	7,302	1		3	1	94	59
27	Hive data warehouse facilities	1,025	9,201	27	9	40	15	560	274
28	Jitsi communicator	326	12,420	19	4	27	2	69	38
29	jMonkeyEngine game development suite	183	5,966			1		72	39
30	jOOQ SQL generator	133	5,022	1		2	1	32	16
31	Kill Bill Billing & Payment Platform	139	5,361	2	2	3	1	51	36
32	LanguageTool Style and Grammar Checker	75	19,121	2	1	4	2	39	28
33	libGDX game development framework	257	12,562	6	2	8	1	150	89
34	Liquibase database source control	77	5,360	14	9	17	4	84	43
35	Minecraft Forge	72	5,498	7	5	13	5	101	64
36	Openfire XMPP server	196	7,436	3	1	4	2	147	90
37	openHAB home automation platform add-ons	331	8,868	2	2	4		167	92
38	OpenTripPlanner multi-modal trip planner	90	8,698	5	1	9	1	111	69
39	OrientDB Multi-Model DBMS	390	14,118	13		19		300	183
40	OsmAnd navigation application	230	34,278	21	11	29	9	439	273
41	Pinpoint Application Performance Management	245	8,565	7	2	10	3	92	73
42	Presto distributed SQL query engine for big data	400	8,597	9	5	16	4	131	79
43	Processing Core and Development Environment	97	12,171	3		5	1	93	57
44	React Native framework for building native apps	48	7,842	3		7	1	19	6
45	Spring Framework	548	13,312	13	3	26	6	147	75
46	Storm distributed realtime computation system	213	7,451			4		43	23
47	VoltDB in-memory SQL RDBMS	573	23,131	15	4	24	2	636	357
Total				367	107	607	114	8,405	4,844

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 4.2 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 4.2: Time and space requirements for a representative sample of studied systems.

System	KLOC	Commits	Draco		REsolution		Observations
			Time	Space	Time	Space	
Hive	1,025	9,201	5h	4GB	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

4.3.3 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) [99], Coupling Between Objects (CBO) [100], and the set of QMOOD (Quality Model for Object Oriented Design) metrics [101]. We chose these metrics because they have been used in a number of studies, including a recent research work that evaluates a state of the art approach for recommending refactorings [6]. 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 chapter.

- **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 chapter 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 [101] 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 [101] 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 [101] as:

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

4.3.4 Applying the Refactorings

In this first assessment we followed the approach of Tsantalis and Chatzigeorgiou [50] 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 $w : V \times V \to \mathbb{N}$ 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.

4.3.5 Results

After running Draco according to the previous sections on the 47 selected systems, we were able to identify:

- using best MQ variation, 367 evolutionary smells on 42 systems, leading to 107 recommendations of move method/field refactorings that resolve evolutionary smells from 30 systems;
- using Pareto Set variation, 607 evolutionary smells on 45 systems, leading to 114 recommendations of move method/field refactorings that resolve evolutionary smells from 33 systems (this set of recommendations includes the set of the previous variation);
- using co-change dependencies variation, 8,504 evolutionary smells on all systems, leading to 4,844 recommendations of move method/field refactorings that resolve evolutionary smells from all systems (this set of recommendations includes the set of the previous two variations).

All these refactorings satisfy the constraints discussed in Section 3.3. Table 4.1 summarizes these results.

We also manually executed the *REsolution* provided by Wang et al. [6], and the *JDeodorant* tools[2], and collected the move method/field refactoring recommendations for the 47 systems. However, *JDeodorant* recommended refactorings for 32 systems only, while *REsolution* recommended refactorings for 36 systems.

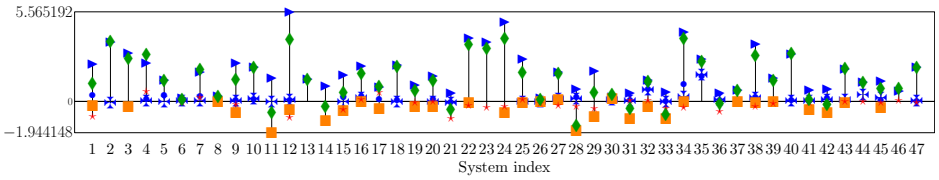
We then virtually applied the recommended refactorings in four different ways:

- first, we applied the refactoring recommendations to all systems that the three variations of Draco found recommendations, one time for each variation;

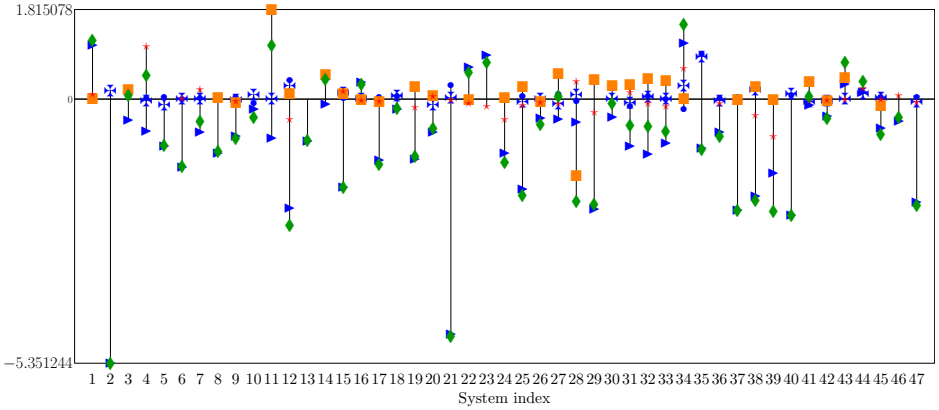
[2]We could not execute the JMove tool because of the amount of processing time needed, as it would exceed our available timeframe for this experiment.

- second, we applied the refactoring recommendations to the 32 systems that JDeodorant found a recommendation;
- third, we applied the refactoring recommendations to the 36 systems that REsolution found a recommendation;
- fourth, we applied the refactoring recommendations from all approaches to all systems.

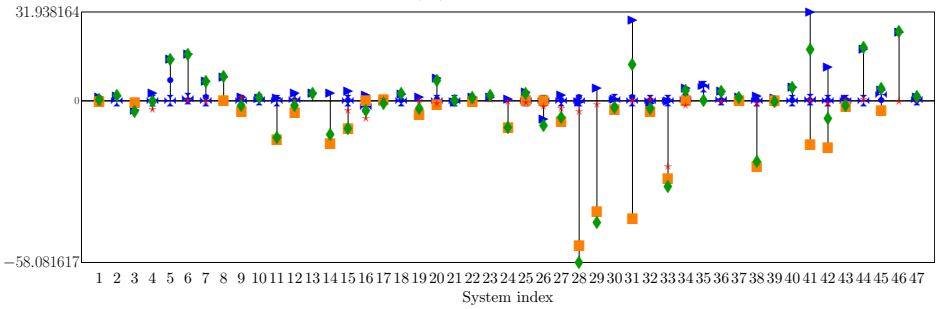
Figure 4.1: Improvement on coupling metrics after applying recommended refactorings. Symbols mean: ●=Best MQ Draco variation, ✠=Pareto Set Draco variation, ▶=Co-change Dependencies Draco variation, ■=JDeodorant, ⋆=REsolution, ♦=All approaches combined. The x axis represents the system index according to the Table 4.1.



(a) CBO



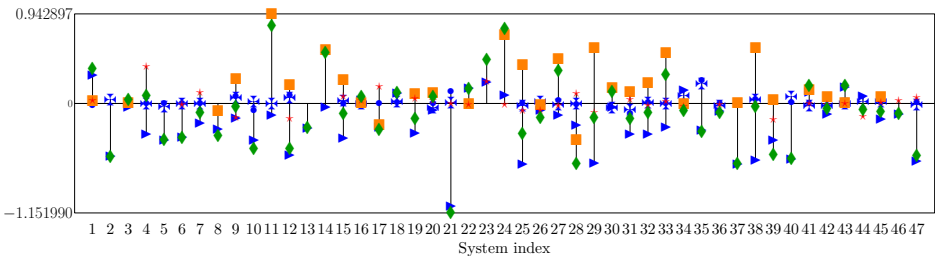
(b) MPC



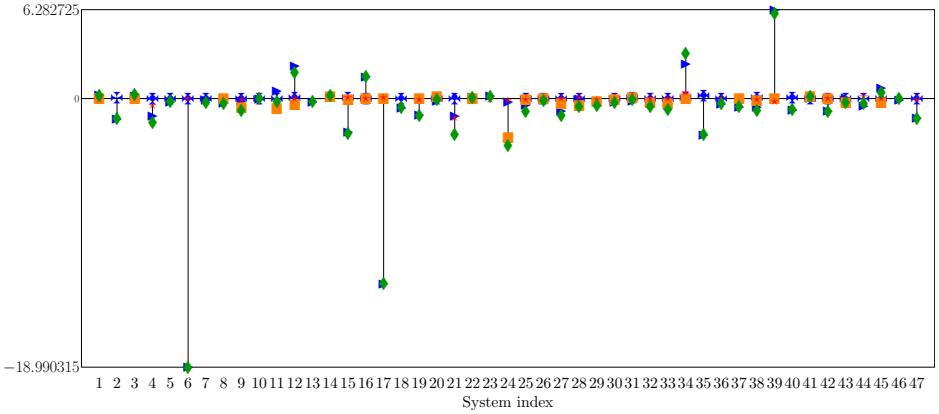
(c) PC

Figures 4.1 and 4.2 shows the impact on the metrics *CBO*, *MPC*, *PC* (that measure coupling), *Reusability*, *Flexibility*, and *Understandability* (that measure quality attributes), for the 47 systems. The values represent the impact on the metrics after applying the refactorings. We normalized the metrics in all figures, and thus the *better values*

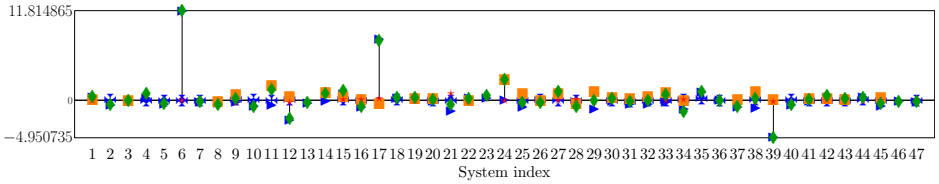
Figure 4.2: Improvement on design attributes after applying recommended refactorings. Symbols mean: ●=Best MQ Draco variation, ✠=Pareto Set Draco variation, ▶=Co-change Dependencies Draco variation, ■=JDeodorant, ⋆=REsolution, ♦=All approaches combined. The x axis represents the system index according to the Table 4.1.



(a) Reusability



(b) Flexibility



(c) Understandability

correspond to the greater values. The values express the percentage of improvement. The results for each approach are denoted by different symbols, as follows. A "●" symbol represents the Best MQ variation of Draco approach (ours). A "✠" symbol represents the Pareto Set variation of Draco approach (ours). A "▶" symbol represents the Co-change dependencies variation of Draco approach (ours). A "■" symbol represents JDeodorant (Tsantalis et al.) approach. A "⋆" symbol represents REsolution (Wang et al.) approach. A "♦" symbol represents the combination of all approaches. Based on these results, it is possible to realize that all variations of Draco outperforms both REsolution and JDeodorant approaches, in the majority of the cases.

Table 4.3: Mann-Whitney U test p-values of Best MQ Draco variation when compared with Wang et al., Tsantalis et al., and the original system metrics (with Benjamini-Yekutieli correction). Note that this Draco variation improves about 45% of the metrics with a statistical significance at least 95% (*p-value*< 0.05), and about 56% of the metrics with a statistical significance of 90% (*p-value*< 0.1).

Metric	Wang et al.	Tsantalis et al.	Original System
CBO	0.0000014	0.0002040	0.0002286
MPC	1.0000000	0.2199714	0.0806226
PC	0.0000014	0.0185294	0.0026699
Reusability	1.0000000	1.0000000	0.3092649
Flexibility	0.0002833	0.0500178	0.0049782
Understandability	1.0000000	1.0000000	0.6325093

Table 4.4: Mann-Whitney U test p-values of Pareto Set Draco variation when compared with Wang et al., Tsantalis et al., and the original system metrics (with Benjamini-Yekutieli correction). Note that this Draco variation improves about 45% of the metrics with a statistical significance at least 95% (*p-value*< 0.05), and half of the metrics with a statistical significance of 90% (*p-value*< 0.1).

Metric	Wang et al.	Tsantalis et al.	Original System
CBO	0.0000026	0.0006662	0.0006952
MPC	1.0000000	0.1863700	0.1646635
PC	0.0000026	0.0084822	0.0065073
Reusability	1.0000000	1.0000000	0.2322727
Flexibility	0.0007789	0.0669334	0.0065073
Understandability	1.0000000	1.0000000	0.3675567

4.3.6 (RQ2.1) 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. The Mann-Whitney U test is a non-parametric test, that does not assume anything about the undelying distribution. It however assumes that the two samples comes from the same population and that the two distributions are similar in shape. Both assumptions are true for our case, because both distributions derive from the same population, which is the set of metrics of the studied systems. Specifically, we tested if Draco performs significantly better than (a) Wang et al. approach, (b) Tsantalis et al. approach, and (c) if the improvement is significant upon the original system metrics. Tables 4.3, 4.4, and 4.5 show the results of the significance tests. Considering *CBO*, *PC*, and *Flexibility*, the Best MQ and Pareto Set Draco variations lead to a significant improvement when compared both with the original system decomposition and the resulting decomposition computed using the Wang et al. and Tsantalis et al. approaches

Table 4.5: Mann-Whitney U test p-values of Co-change Dependencies Draco variation when compared with Wang et al., Tsantalis et al., and the original system metrics (with Benjamini-Yekutieli correction). Note that this Draco variation improves about 38% of the metrics with a statistical significance at least 95% (*p-value*< 0.05).

Metric	Wang et al.	Tsantalis et al.	Original System
CBO	0.0000000	0.0000000	0.0000000
MPC	1.0000000	1.0000000	1.0000000
PC	0.0000001	0.0000013	0.0000010
Reusability	1.0000000	1.0000000	1.0000000
Flexibility	1.0000000	1.0000000	1.0000000
Understandability	1.0000000	1.0000000	1.0000000

Table 4.6: Cohen's d effect size statistics of Best MQ Draco variation when compared with Wang et al., Tsantalis et al., and the original system metrics. Note that the Draco approach leads to a positive and non-negligible improvement on all metrics when compared with original decomposition (except for *Understandability* metric), and on the majority of the metrics when compared with the other approaches.

Metric	Wang et al.	Tsantalis et al.	Original System
CBO	1.1119140 (large)	0.8867138 (large)	0.6802267 (medium)
MPC	−0.2150356 (small)	0.2144177 (small)	0.3345363 (small)
PC	0.8009070 (large)	0.4537811 (small)	0.4035344 (small)
Reusability	−0.6493640 (medium)	−0.0276635 (negligible)	0.2157479 (small)
Flexibility	0.5148191 (medium)	0.3309473 (small)	0.4886344 (small)
Understandability	−0.6794275 (medium)	−0.1774650 (negligible)	0.08594048(negligible)

(at a 0.05 significance level). Considering *MPC*, *Reusability* and *Understandability*, the improvement was not statistically significant.

Regarding the results of the Cohen's d effect size test (Tables 4.6, 4.7, and 4.8), it is possible to realize that Best MQ and Pareto Set Draco variations leads to a non-negligible improvement for all metrics when compared with the original decomposition (except for *Understandability* metric on the Best MQ approach), while the Co-change Dependency variation lead to a non-negligible and positive effect only for the *CBO* and *PC* metrics, when compared with the original decomposition. When compared with the Wang et al. approach, the Best MQ and Pareto Set variations leads to a positive and non-negligible improvement for *CBO*, *PC*, and *Flexibility* metrics, while the Co-change dependencies leads to a positive and non-negligible improvement for *CBO* and *PC* metrics. When compared with the Tsantalis et al. approach, the Best MQ and Pareto Set variations leads to a positive and non-negligible improvement for all metrics except *Reusability* and *Understandability*, while the Co-change Dependencies variation lead to a positive and non-negligible improvement for *CBO* and *PC* metrics.

We also measured how the systems' attributes relates to the improvement on the

Table 4.7: Cohen's d effect size statistics of Pareto Set Draco variation when compared with Wang et al., Tsantalis et al., and the original system metrics. Note that the Draco approach leads to a positive and non-negligible improvement on all metrics when compared with the original decomposition, and on the majority of the metrics when compared with the other approaches.

Metric	Wang et al.	Tsantalis et al.	Original System
CBO	1.0463810 (large)	0.8040940 (large)	0.5615590 (medium)
MPC	−0.2060146 (small)	0.2352446 (small)	0.3957810 (small)
PC	0.7551641 (medium)	0.3957061 (small)	0.3119489 (small)
Reusability	−0.6626160 (medium)	0.0037149 (negligible)	0.3312043 (small)
Flexibility	0.5028700 (medium)	0.3031919 (small)	0.5209327 (medium)
Understandability	−0.7063708 (medium)	−0.1569276 (negligible)	0.2149195 (small)

Table 4.8: Cohen's d effect size statistics of Co-change Dependencies Draco variation when compared with Wang et al., Tsantalis et al., and the original system metrics. Note that the Draco approach leads to a positive and non-negligible improvement on *CBO* and *PC* metrics when compared with the other approaches and the original system.

Metric	Wang et al.	Tsantalis et al.	Original System
CBO	2.1627940 (large)	2.2237500 (large)	2.0275050 (large)
MPC	−1.1305820 (large)	−0.9453627 (large)	−1.0684640 (large)
PC	1.1213190 (large)	0.9387947 (large)	0.8882317 (large)
Reusability	−1.3057570 (large)	−1.0768640 (large)	−1.1114080 (large)
Flexibility	−0.2633624 (small)	−0.3033493 (small)	−0.3168353 (small)
Understandability	−0.1405114 (negligible)	0.01765848(negligible)	0.03762118(negligible)

quality metrics presented in Section 4.2. 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. Tables 4.9, 4.10, and 4.11 show the results, revealing that the most effective attributes on quality metric improvement is *refactoring recommendations count* and *co-change clusters density*, since they have a positive statistical significance of 99.9% (p-*value*< 0.001).

Actually, the *static graph density* also affect the *CBO* metric with 95% of statistical significance (p-*value*< 0.05) when using the Co-change Dependencies variation. The *Co-change graph density* attribute also affects the *Understandability* metric (in this case with 99% of significance). Finally, the *co-change graph density* negatively affects the *Flexibility* metric (with 99% of significance) when using the Co-change Dependencies variation. A dense co-change graph might suggest that the entities are heavily co-change coupled, and therefore the Co-change Dependencies variation could be led to be too aggressive on recommending refactorings.

Table 4.9: Effect of attributes on metrics improvement using the Best MQ variation. Note that we have only two attributes influencing a metric with 99.9% of significance (*p-value*< 0.001, denoted by a *** suffix).

	CBO	*MPC*	*PC*	*Reusab.*	*Flexib.*	*Understandab.*
Intercept	0.0015	−0.0004	0.0094	−0.0002	0.0001	−0.0004
	(0.0017)	(0.0008)	(0.0073)	(0.0003)	(0.0002)	(0.0004)
Refactoring recommendations count	**0.0004****	−0.0000	0.0005	−0.0000	**0.0001*****	**−0.0001***
	(0.0002)	(0.0001)	(0.0007)	(0.0000)	(0.0000)	(0.0000)
Co-change clusters density	0.0004	**0.0010*****	−0.0010	**0.0003*****	**0.0001***	**0.0004*****
	(0.0004)	(0.0002)	(0.0016)	(0.0001)	(0.0001)	(0.0001)
Entities count	−0.0000	−0.0000	−0.0000	0.0000	**−0.0000***	0.0000
	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)
Static graph density	8.0278	1.7523	−11.5864	0.2327	0.7725	0.2979
	(9.9560)	(4.4916)	(42.9487)	(1.6373)	(1.3751)	(2.4411)
Co-change graph density	−22.0918	−0.8685	−33.0254	0.9607	−2.7501	1.9936
	(14.6642)	(6.6158)	(63.2595)	(2.4116)	(2.0253)	(3.5956)
R^2	0.2959	0.4856	0.0454	0.3182	0.5314	0.3503
Adj. R^2	0.2101	0.4228	−0.0710	0.2351	0.4743	0.2710
Num. obs.	47	47	47	47	47	47

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

Table 4.10: Effect of attributes on metrics improvement using the Pareto Set variation. Note that we have only one attribute influencing a metric with 99.9% of significance (*p-value*< 0.001, denoted by a *** suffix).

	CBO	*MPC*	*PC*	*Reusab.*	*Flexib.*	*Understandab.*
Intercept	0.0006	−0.0000	0.0056	−0.0001	0.0001	−0.0001
	(0.0016)	(0.0007)	(0.0053)	(0.0002)	(0.0002)	(0.0003)
Refactoring recommendations count	0.0002	0.0001	0.0005	0.0000	0.0000	0.0000
	(0.0001)	(0.0001)	(0.0005)	(0.0000)	(0.0000)	(0.0000)
Co-change clusters density	**0.0008***	**0.0007*****	**0.0029***	**0.0002****	**0.0002*****	**0.0002****
	(0.0003)	(0.0001)	(0.0011)	(0.0000)	(0.0000)	(0.0001)
Entities count	−0.0000	−0.0000	−0.0000	−0.0000	**−0.0000***	−0.0000
	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)
Static graph density	11.1409	6.4369	−9.1852	1.7795	0.9174	2.4334
	(9.2231)	(4.1507)	(30.9032)	(1.3449)	(1.0164)	(1.9612)
Co-change graph density	−26.0487	−10.0773	−35.5090	−2.0252	−2.3331	−3.3061
	(13.5580)	(6.1015)	(45.4276)	(1.9770)	(1.4941)	(2.8829)
R^2	0.2826	0.5092	0.2536	0.3674	0.5117	0.3204
Adj. R^2	0.1951	0.4493	0.1626	0.2903	0.4521	0.2375
Num. obs.	47	47	47	47	47	47

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

Table 4.11: Effect of attributes on metrics improvement using the Co-change Dependencies variation. Note that we have only two attributes positively influencing a metric with 99% of significance (*p-value*$<$ 0.01, denoted by a ** suffix). While we have one attribute influencing three metrics with 95% significance (*p-value*$<$ 0.05, denoted by a * suffix).

	CBO	*MPC*	*PC*	*Reusab.*	*Flexib.*	*Understandab.*
Intercept	0.0082	0.0061	**0.1385***	−0.0010	−0.0162	0.0090
	(0.0068)	(0.0074)	(0.0515)	(0.0018)	(0.0212)	(0.0138)
Refactoring recommendations count	**0.0001****	−0.0000	−0.0001	−0.0000	0.0001	**−0.0001***
	(0.0000)	(0.0000)	(0.0002)	(0.0000)	(0.0001)	(0.0000)
Co-change clusters density	0.0024	−0.0004	0.0014	−0.0001	−0.0010	0.0003
	(0.0015)	(0.0016)	(0.0114)	(0.0004)	(0.0047)	(0.0031)
Entities count	−0.0000	−0.0000	−0.0000	0.0000	−0.0000	0.0000
	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)	(0.0000)
Static graph density	**85.2395***	−82.6447	−284.2324	−1.9720	207.8758	−148.9762
	(39.2626)	(42.8241)	(297.0605)	(10.6098)	(122.5053)	(79.7970)
Co-change graph density	−82.2944	−44.2657	−465.3395	−11.2475	**−669.0292*****	**419.4931****
	(58.2410)	(63.5239)	(440.6506)	(15.7383)	(181.7207)	(118.3686)
R^2	0.4314	0.2585	0.1202	0.1329	0.2771	0.2831
Adj. R^2	0.3620	0.1681	0.0129	0.0272	0.1890	0.1957
Num. obs.	47	47	47	47	47	47

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

Answer to **RQ2.1**: The Draco approach reduces coupling when measured by *CBO* and *PC* metrics, and the Best MQ and Pareto Set variations also improve *flexibility*—considering both co-change and static dependencies. The improvement is proportional to the number of identified refactoring opportunities.

4.3.7 (RQ2.2) How does the Draco approach compare to state of the art approaches for refactoring recommendation?

The results discussed in the last section leads to several findings. First, at least one variation of Draco improved the *CBO* and *PC* metrics for almost all systems. However, the Wang et al. approach improved *CBO* for only 7 out of 36 systems, *MPC* for only 12 systems, and *PC* for only 13 systems. The Tsantalis et al. approach improved *CBO* for only 2 out of 32 systems, *MPC* for only 20 systems, and *PC* for only 1 system. For these three metrics (*CBO*, *MPC*, and *PC*), in only 30 cases, out of 141 (3 metrics $\times$ 47 systems), either Wang et al. or Tsantalis et al. approaches outperform any variation of 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 Draco approach and the two

other approaches are complementary. This occurs because no refactoring recommended by Draco was also recommended by the Wang et al. or Tsantalis et al. approaches.

Given the measures of quality attributes investigated here (*Reusability*, *Flexibility*, and *Understandability*), the best MQ Draco variation improved 56 of 90 measures (62.22%, for 3 metrics × 30 systems), the Pareto Set Draco variation improved 69 of 99 measures (69.69%, for 3 metrics × 33 systems), and the Co-change dependencies Draco variation improved 36 of 141 measures (25.53%, for 3 metrics × 47 systems), while the Wang et al. approach improved 48 of 108 measures (44.44%, for 3 metrics × 36 systems), and the Tsantalis et al. approach improved 56 of 96 measures (58.33%, for 3 metrics ×32 systems). Besides that, Draco outperforms both Wang et al. and Tsantalis et al. approaches on 68 measures, while the opposite occurs on 73 measures. Likewise the other quality metrics, the improvement of the two approaches can be summed together when we apply them together.

Overall, Draco outperformed the other approaches on 63% of the measurements.

Answer to **RQ2.2**: these results suggest that the Draco approach outperforms state of the art techniques for recommending move method/field refactorings, when we consider both co-change and static dependencies.

4.3.8 (RQ2.3) 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

Table 4.12: 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 $C_{min} = \beta$

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.

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 4.3.2), in this analysis we used only the 10 smaller systems from the original set of 47 studied systems. Furthermore, we experimented only with the Best MQ variation.

Table 4.12 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 4.3.2. 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 **RQ2.3**: 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.

4.3.9 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 [61] (in addition to the evolutionary smell, of course). Accordingly, we manually applied the *Push Down* refactoring [61], 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 [102, 103]. In summary, after a few adjustments, we successfully applied all Draco refactoring recommendations we selected for manual verification.

4.3.10 Threats to Validity to the Quantitative 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 4.3.8.

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 possi-

ble 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 4.4.3 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 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, REsolution and JDeodorant) 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".

4.4 Qualitative Assessment 1: SIOP

This section presents our first qualitative assessment, where we executed the Draco approach in the context of an large enterprise system: SIOP.

4.4.1 Studied System

SIOP is a Java Enterprise system from the financial domain and have approximately 660k LOC and 3,200 files. The SIOP development history spans for more than 10 years, and comprises more than 30,000 commits. Currently, there are 15 full-time developers maintaining the SIOP codebase, although more than 40 developers worked on it at some point in its history. As expected, this developer turnover increased the system susceptibility to quality problems. The SIOP design contains 26 subsystems, where a dedicated team of developers maintain each subsystem. Each subsystem has a "owner", i.e., a single developer is responsible for the source code of a subsystem. Furthermore, a subsystem have another collaborators, though the final decision about changes in its source code are made by its owner. There is no automated code review process, thus the discussions about source code changes happens on a ad hoc basis. On average, 300 commits are pushed monthly, spanning all subsystems.

4.4.2 Methodology (Survey + Focus Group)

In this first qualitative assessment, we employed two studies: a survey and a focus group.

Survey

In the survey, we investigated the outcomes of four refactoring recommendation tools: Draco, JDeodorant, JMove, and REsolution. Although JDeodorant and REsolution identify other kinds of refactorings, for the scope of this research, we restricted our interest to move method/field refactorings. We focused on the **move method** refactoring, because it is very common [91, 92], and helps to solve design problems.

We used the refactoring recommendation tools to reveal possible improvements on the SIOP design. From a population of 15 developers of the system, eight of them participated in this research. The participants have, on average, 15 years on software development practice and seven years working on the system. Five of them use popular refactoring tools on a daily basis (those tools included in IDEs such as Eclipse and IntelliJ), and all of them know the purposes and benefits of the move method/field refactoring. Two of them are familiar with existing research on refactoring tools.

Using the survey instrument, we requested the software developers of each module to evaluate a set of refactoring recommendations, and then answer the following (survey) questions:

(SQ1) Do you agree that this refactoring recommendation improves any quality attribute of SIOP?

(SQ2) Would you intend to integrate this refactoring recommendation into the SIOP source code?

(SQ3) In your opinion, what are the weaknesses of this refactoring recommendation?

Questions (SQ1) and (SQ2) are closed-questions. Question SQ1 aims to collect the opinion of the participants using a Likert scale with five options, from *strongly disagree* to *strongly agree*, with a neutral point. We presented to the participants three quality attributes that often motivate a refactoring: flexibility, reusability, and comprehensibility. SQ2 is a "yes" or "no" question, mimicking the procedure of reviewing a code—either to accept or reject a pull-request, for instance. Question (SQ3) is an open-ended question, allowing the respondents to justify the decision for accepting or rejecting a transformation.

Data collection procedure. We ran the three refactoring recommendation tools in SIOP, on a particular version of the master branch. We obtained a total of 530 recommendations: 354 recommendations from JDeodorant, 148 recommendations from REsolution, no recommendations from JMove, and 24 recommendations from Draco. We used

the *out-of-the-box* configurations for JDeodorant, JMove and REsolution. Draco is a highly configurable approach, and then we used the values of 4 and 0.5 to its support cont and confidence parameters. These values produce a good trade-off between statistical significance of the co-change dependency and the number of recommendations, as well as being often used in the literature [28, 30, 40, 60, 80]. Furthermore, we used all three variations of Draco to compute their recommendations (the available options was explained in Section 3.3). Nonetheless, we decided to remove JMove from our analysis, because it did not reveal any refactoring opportunity for SIOP.

Since the participants of this study are experienced developers and architects who work full time, we had to limit the number of recommendations according to the developers' availability. Since Draco produced fewer recommendations than the other tools (Draco: 24, JDeodorant: 354, REsolution: 148), we selected all of the recommendations from Draco. However, we have to limit the recommendations from the other tools to not increase the burden on the participants. We randomly chose 20% of the recommendations from JDeodorant and REsolution, (70 and 30 recommendations, respectively). From this initial set, we identified two similar recommendations coming from both JDeodorant and REsolution. To avoid asking developers to evaluate similar recommendations, we removed these transformations from our dataset. The result was a final corpus of 122 refactorings recommendations. As far as we know, this is the most extensive corpus of refactoring recommendations that practitioners have empirically evaluated in a research study. We assigned a subset of the refactoring recommendations for each participant of our study, using code ownership as distribution criteria. It must be noted that we did not chose a random developer. As we have seen previously, each subsystem of SIOP is owned by a single developer. Accordingly, we sent to this developer the list of refactoring recommendations that involved the source code of their subsystem.

The participants knew that the refactoring recommendations were from automated tools, but they did not know which tools were used to produce them. No refactoring recommendation was evaluated more than once, and the software developers evaluated between 3 and 35 patch recommendations. For each software developer, we curated a document describing the intention of this research, a description of the design problem that the refactorings mitigate, and the recommended refactorings themselves. The developers had ten days to review the recommendations. After that period, all of them responded; the majority analyzed all recommendations they received, while some participants left a few recommendation analysis missing.

Focus group discussion

We conducted our focus group discussion electronically, with the eight software developers that analyzed the refactorings—where we question about automated tests and refactoring, evaluation criteria, recommendation quality, and reasons for rejecting a recommendation. Focus group is a research approach that emerged in the 1950s, mostly the social sciences [104]. This approach relies on carefully planned discussions, designed to obtain the perceptions of the group members on a well-defined area of interest. The outcomes of a focus group produce qualitative information about the objects of a study [104]—in our case, the perceptions of the refactoring recommendations.

We conducted the focus group discussions using an online group chat and lasted approximately two hours. The mediator presented four open-ended questions and one multiple choice question to the participants, one at a time, and then they proceeded to comment each question. The questions are as follows:

- *What is the relevance of automated tests to accept refactoring recommendations from tools?*
- *Which criteria do developers employ for evaluating the refactoring recommendations?*
- *What are the problems that may lead the developers to stop using a refactoring recommendation tool?*
- *What is your general perception about the quality of the refactoring recommendations?*

We used the feature Group Chat of Telegram for conducting this investigation, a feature that had already been used in the technical discussions related to SIOP. In this way, we could promote an engaging discussion with the participants, without the need to group all participants together in the same physical place.

4.4.3 Results of the Survey

Table 4.13 summarizes the outcomes of a quantitative analysis on the survey answers. Developers evaluated 110 out of the 122 transformations in our final dataset. In terms of accepted patches, JDeodorant presents the best performance: developers accepted 62% of its recommendations. Draco-CCD and REsolution present a lower acceptance rate (38% and 24%, respectively). The set of recommendations from Draco-CCD includes the recommendations from Draco-BMQ and Draco-PS. The recommendations from Draco-BMQ

and Draco-PS are the same (50%). These results support the practical application of refactoring recommendation tools, since the SIOP developers would accept 54 code transformations that are likely to fix design flaws of the system.

Table 4.13: Summary of the analysis for the recommendations from three different tools. BMQ means Best Modularization Quality variation, PS means Pareto Set variation, and CCD means Co-change Dependencies variation.

Tool	Draco			JDeodorant	REsolution
	BMQ	PS	CCD	JDeodorant	REsolution
Evaluated	4	4	24 (100%)	65 (93%)	21 (75%)
Accepted	2 (50%)	2 (50%)	9 (38%)	40 (62%)	5 (24%)

Figure 4.3 summarizes the developers' perceptions about possible improvements of the refactoring recommendations on *flexibility*, *reusability*, and *comprehensibility*. It is possible to realize that only JDeodorant presents a positive leaning in all quality attributes considered in the research—even though the positive feeling is not that strong and at least 32% of the answers lie in a negative opinion (either strongly disagree or disagree) that JDeodorant recommendations improve *flexibility*, *reusability*, and *comprehensibility*. There is no consensus that JDeodorant improves these quality attributes, though it is not clear what other characteristic it excels. Since both Draco and REsolution present a lower acceptance level (bellow 50%), it is more clear that they would not show a positive leaning towards improving these quality attributes—even though developers still argue that Draco improves these quality attributes in at least 30% of the assessments; while REsolution presents the lowest percentage of positive answers (bellow 15%). In what follows, we present more details about the assessment of each individual tool.

Draco analysis. Draco recommended 24 refactorings and 9 were accepted. Four accepted recommendations had the same source/destination classes. Furthermore, for the accepted recommendations, the involved methods have low coupling with the source class, i.e., they do not call any method or access any field from the source class. According to the analysis of the developers, these refactorings would lead to improvements on the flexibility, reusability, and comprehensibility of the source code. Figure 4.4 shows an example of an accepted recommendation.

Regarding the rejected recommendations, we found that 8 out of the 15 rejected recommendations involve methods that override a method declared in a Java interface, and therefore could not be moved. One could argue that the destination class could be updated to also implement the interface, then allowing moving the method. However, this would introduce additional changes other than just moving the method, some of which could require a more careful analysis from the experts.

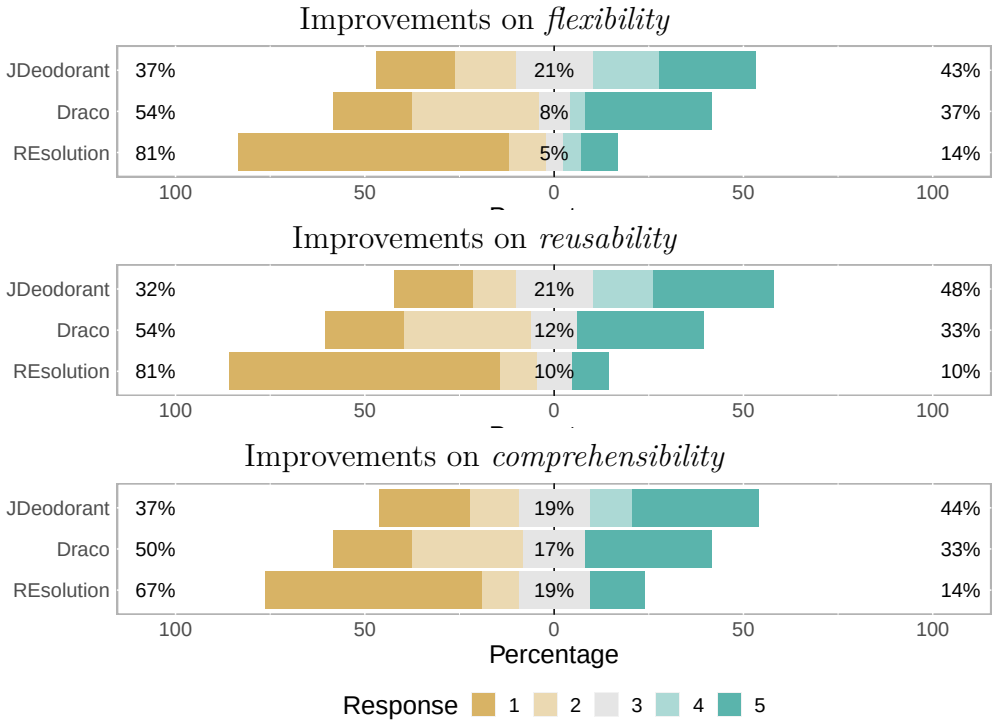
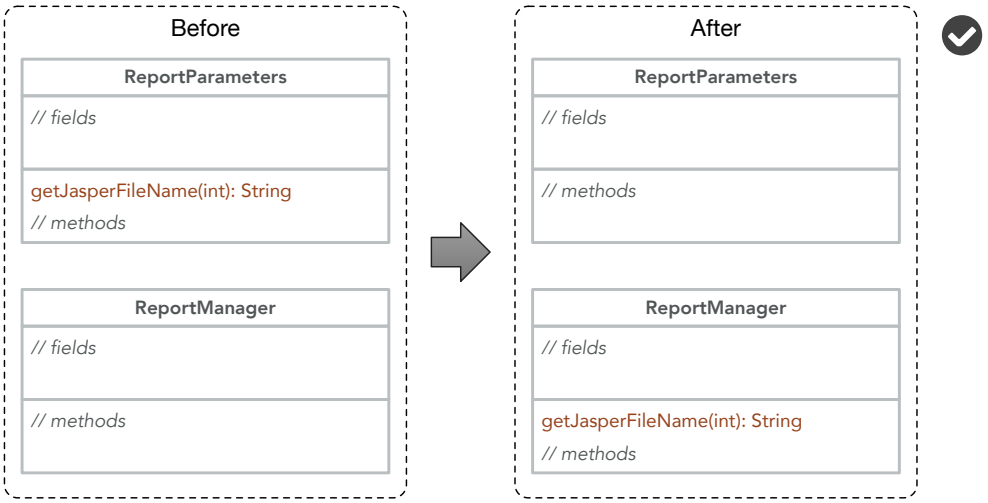


Figure 4.3: From top to bottom: perceived improvement on *flexibility*, *reusability*, and *comprehensibility* of the refactoring recommendations. From "Strongly Disagree"=1 to "Strongly Agree"=5. Left percentages represent negative perception, and right percentages represent positive perception.

Developers rejected one of the Draco recommendations because it suggests moving a method from a *business service* class to a *user interface controller* class. However, these two classes have different architectural roles. These classes implement the so-called "Core J2EE Patterns," a set of well-known recommendations for developing enterprise systems in Java. More concretely, while the former implements business logic, the latter implements user interface logic. For this reason, although this recommendation removes co-change dependencies, it leads to a violation of an architectural constraint. Figure 4.5 shows an example of a recommendation from Draco that has not been accepted.

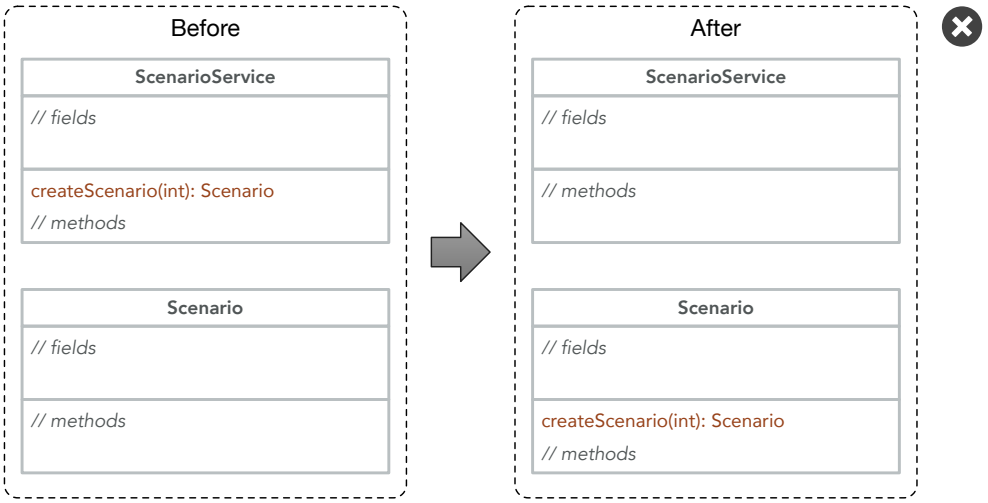
One particular rejected recommendation from Draco is worth discussion. The recommendation was to move a method (`groupRevenuesBySource`) that belongs to an interface implemented by the source class (`ScenarioValue`) to the class that uses the interface (`SourceProcessing`), as shown in Figure 4.6. The 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




```
public static String getJasperFileName(int rpt) {
    String fName = null;
    switch (rpt) {
    case IReportManager.PROGRAM_REPORT:
        fName = "program.jrxml";
        break;
        // long method...
    }
    return fName;
}
```

Figure 4.4: Accepted recommendation from Draco. It recommended to move the method `getJasperFileName` from class `ReportParameters` to class `ReportManager`. The moved method has a long sequence of code that uses the interface `IReportManager` which is implemented by the destination class `ReportManager`.

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 exclusively handled by class `SourceProcessing`. Therefore, a better design would be the use of the *visitor pattern*, as shown in Figure 4.7. 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




```
public Scenario createScenario(Integer period) {
    Scenario s = new Scenario();
    s.setPeriod(period);
    s.setProjectStartDate(...);
    s.setConsolidationTime(...);
    // ...
    return s;
}
```

Figure 4.5: Rejected recommendation from Draco. It recommended to move the method `create` from class `ScenarioService` to class `Scenario`. The recommended destination class has a different architectural role than the source class (*Entity* and *Business Service*, respectively), therefore the rejection, even though the two classes have a high coupling, both statically and co-change wise.

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.

JDeodorant analysis. JDeodorant recommended 354 move method refactorings, and, from this initial set, we selected 70 random recommendations for further analysis. From these selected recommendations, we received 65 answers from the practitioners. The practitioners would be intended to accept 40 of these recommendations (62% of the recommendations we explored in the analysis phase). In general, we observed that the accepted recommendations involve moving methods presenting a low cohesion with the source class, though with a high coupling with the destination class.

Figure 4.6: 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)

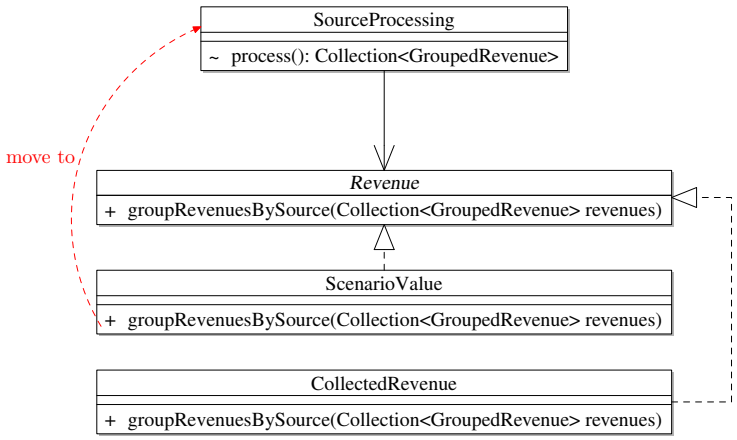
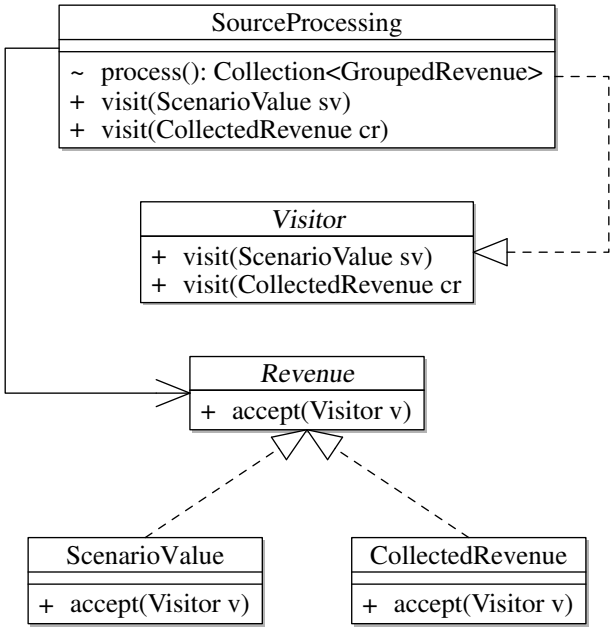


Figure 4.7: A better design for the classes and interfaces involved in the rejected refactoring recommendation for SIOP as illustrated by Figure 4.6. The adoption of the *visitor pattern* allowed to decouple the logic of grouping revenues by source from the `Revenue` implementors, keeping the logic of `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)



Again, regarding the JDeodorant recommendations that has not been accepted, most cases violate architectural constraints, such as the source and destination classes having different responsabilities—e.g., when the source class is a *business service* class and the destination class is a *value object*. We also realized that the participants do not accept recommendations when the refactoring would actually increase the coupling between the source and destination class, i.e., in situations where the number of method calls and field access would increase.

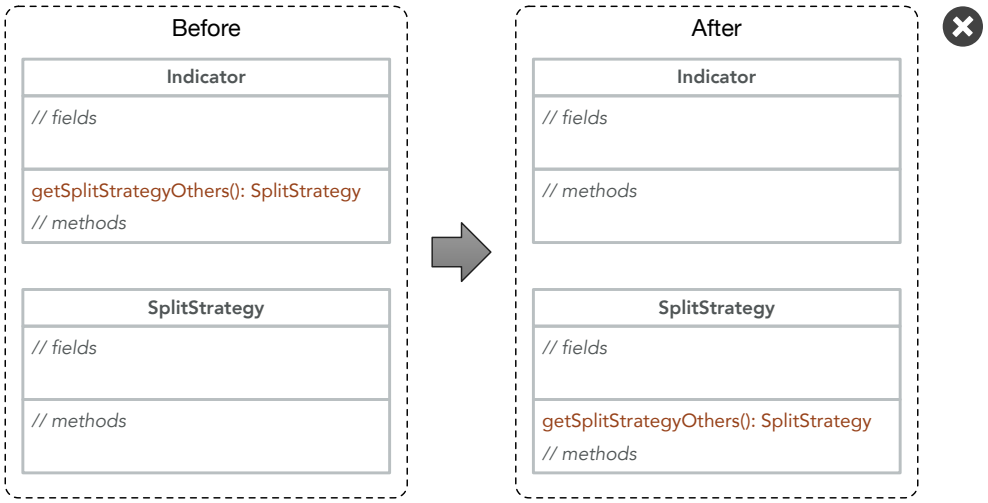
REsolution analysis. The refactoring recommendations from REsolution tool had the lowest acceptance rate. From the initial set of 28 recommendations, we received 21 answers from the developers. From these, developers accepted only five recommendations (24% of the recommendations we considered in the analysis phase). When analyzing the accepted recommendations, we found that these recommendations lead to a lower coupling between the involved classes.

Regarding the rejected recommendations, we found that the main reason for developers not accepting a REsolution recommendation occurs when the destination class is an unnatural place for the method. For example, when the source class is an *Entity Object*, i.e., a mapping for a real-world object, and the method to be moved represents an attribute of the entity (in the form of a getter or setter). From the domain point of view, moving such methods does not make sense, because the attribute belongs intrinsically to the entity. Listing 4.8 shows an example of this kind of recommendation. We also found that the REsolution tool recommended moving fields that represent value of an *enumeration*, which also does not make sense, because the values of an enumeration represent a highly cohesive set of values, and it is natural that other classes have a high coupling with them. In some cases the developers claimed as a reason for rejecting a recommendation the situations where moving the method will lead to an increase in the coupling between the source and destination classes.

Table 4.14 summarizes the alleged reasons for rejecting a refactoring recommendation that we extracted from reading the answers from the open question "Which are the weakness of this refactoring recommendation?". We can observe that the main reason is the incorrect responsibility of the recommended destination class, i.e., the method to be moved implements a concept or rule that does not belong to the destination class.

4.4.4 Results of the Focus Group Discussion

To cross-validate our impressions about the results of the analysis of the refactoring recommendations, we conducted a focus group discussion with the eight experts that analyzed



(a)

```
public SplitStrategy getSplitStrategyOthers() {
  for(SplitIndicator s: this.getIndicatorList()) {
   if(s.getStrategy()!= null && s.getStrategy().isOther())
      return s;
  }
  return null;
}
```

Figure 4.8: Rejected recommendation from REsolution. It recommended to move the method `getSplitStrategyOthers` from class `Indicator` to class `SplitStrategy`. The attribute that the *getter* method represents (`SplitStrategyOthers`), belongs to the source class, and the recommended destination class does not have to known its concept.

the refactoring recommendations for understanding their perception about refactoring recommendation tools.

The first open-ended question is about the *relevance of automated tests to accept refactoring recommendations from tools*. The studied system has a low test code coverage (less than 10%), and all participants agreed that if the coverage was higher, the chances of accepting a refactoring recommendation would be higher too. They argued that there is a risk in accepting the refactoring recommendations, and this risk would be significantly lower in the presence of automated tests. Still, the participants reported that the size of the change has no influence on its acceptance, and that tests would also be needed to properly assess small changes too. Yet, they argued it does not matter whether the refactoring recommendation was proposed by a tool or by a human.

The second open-ended question is about *which criteria the developers employ for evaluating the refactoring recommendations*.

The participants mentioned the following criteria:

Table 4.14: Alleged reasons for rejecting a refactoring recommendation.

Tool	Draco	JDeod.	REsol.
Incorrect responsibility	2	8	12
Increases coupling	0	7	2
Architectural constraints	5	8	0
Method belongs to an interface	8	0	0
Field belongs to an enumeration	0	0	2
Harms readability	0	2	0

- familiarity with the code to be moved;
- comprehensibility of the code;
- if after the refactoring the coupling of the two involved classes will not be raised;
- if the refactoring would not cause a cyclic dependency between the source and destination classes;
- if it is clear that the responsibility of the destination class were compatible with the moved method;
- manual effort to perform the refactoring, is most relevant when the IDE of choice is not capable to fully automate the refactoring and therefore the developer must perform it manually.

The third open-ended question relates to *the kinds of problems that may lead the developers to stop using a refactoring recommendation tool*. They said the tools must observe if the code involved in the recommendations is in active use. Some of the rejections were influenced by this particular problem, i.e., the methods recommended to be moved were not used anymore. The participants suggested that it would be nice if the recommendation were given at pre-commit time, and involving only the code to be committed, because at this moment the source-code is fresh in the developers memory and thus it would be easier to evaluate the recommendation.

The fourth and last open-ended question consider *the general perception about the quality of analyzed refactoring recommendations*. All participants agreed that they were of good quality and pertinent. (This was an unexpected feedback from the participants, as they rejected half of the recommendations.)

The last question in the focus group presented to the participants is *which is the main reason to reject a refactoring recommendation?*, with the following possible choices:

(R1) violation of some architectural constraint (Figure 4.5 shows an example of this kind of problem)

(R2) incompatibility of the moved method with the responsibilities of the destination class (Figure 4.8 shows an example of this kind of problem)

(R3) incompatibility with the constraints imposed by some framework (like the necessity of implementing some specific interface)

(R4) raising of coupling between the source, destination or some other class

Half of the participants chose item 1 and half chose item 2. No other item had any vote. Altogether, we can summarize an answer to our research question (*What are the practitioners' perceptions about the refactoring recommendations from Draco, JDeodorant, and REsolution?*) as follow:

> Overall, practitioners are positive about the recommendations from the three tools, since they agree that 49% of the recommendations could be integrated into the SIOP code base. However the acceptance rate varies according to the refactoring recommendation tools, where JDeodorant has the highest acceptance rate, REsolution the lowest, and Draco relies between the two. The finds suggest: (a) the relevance of using refactoring recommendation tools, (b) the complementarity of the evaluated approaches (due to the small intersection between recommendations from distinct tools), and (c) the use of co-change dependencies to recommend refactoring can be useful, encouraging further research on the topic.

4.4.5 Discussion

In this section we summarize the lessons we learned throughout the conduction of this study. In the next section we present the threats that could limit the generalization of our results.

Some design constraints only became visible when applying refactoring recommendation tools in the wild. Moreover, we found some limitations for the Draco approach. In particular, it recommended moving methods from classes that implements a Java interface to classes that do not implement that interface. If developers accept this recommendation as is, it could lead to compilation errors in the system. Although it is straight-forward to fix this issue in Draco, the importance of this limitation only became clear after running this study.

Automatic generated refactoring recommendations can indeed be useful. First, when we started this investigation, we did not have the expectation of having many

refactoring recommendations accepted. In particular, the developers that evaluated the submitted patches are really critical: only contributions that clearly improve quality attributes of the system would be considered for integration into the codebase. Therefore, in this scenario, even a low acceptance rate of refactoring recommendations coming from automated tools could be considered a promising result, since it could help to explain the benefits and eventual limitations of existing tools. More concretely, we found an acceptance rate of 62% for the evaluated JDeodorant refactoring recommendations. This corresponds to 40 changes that improved the source code and that have been integrated into SIOP. The Draco approach also appears promising, even if we consider its smaller number of recommendations. Overall, we report an acceptance rate for Draco around 38%. This result evidenced that co-change dependencies could complement existing approaches for refactoring recommendation. To the best of our knowledge, this is the first time that a co-change dependency approach for recommending refactorings has been so extensively evaluated in a practical setting. (Mkaouer et al. [53] used change history only to choose recommendations that are similar to past refactorings).

Generating many recommendations is not the same of generating useful recommendations. Interesting, although REsolution is a state-of-the-art approach for recommending move method refactorings, by finding an optimal balance between coupling and cohesion, its acceptance rate was lower than any other tool (24%); only 5 out of the 21 evaluated recommendations were integrated into the system. This low acceptance rate might indicate that coupling and cohesion may be overrated features for refactoring recommendation, but more research is needed to confirm or refute this hypothesis. Actually, the participants of this study considered that most of the refactoring recommendations from REsolution decrease the design's flexibility, reusability opportunities, and comprehensibility.

Refactoring tools should take context into consideration. We observed that some recommendations do not make much sense because they somehow break the architectural decisions of the system. These issues became clear to us when one of the developers stated that:

> "*The target class is just a value object, and the method in question involves business logic.*"

These refactoring tools could be augmented to avoid these scenarios. For instance, considering classes that have different responsibilities (e.g., one implements a controller and another that implements a view in the MVC pattern). It makes little sense to move methods between these classes. Refactoring tools could then use, for instance, topic model techniques to avoid these issues. Another approach would be to establish simple rules

stating that the refactoring tool should not recommend move methods from classes in a certain Java packages to classes in another Java package, according to the architecture of a specific system.

Refactoring tools should avoid simple methods. Java software projects are often full of simple methods such as getters and setters, hashcode, equals, and toString, among others. Other rules that avoid moving these methods are straightforward to implement and may also decrease the number of false positives.

Refactoring tools should take better advantage of version control systems. Moreover, we envision a scenario in which refactoring tools take better advantage of the version control systems. For instance, we could state a simple rule to avoid moving methods in components that do not have been changed for a certain period of time. This avoids a scenario where the tool suggests moving a method that makes sense, but the origin or destination class has been rarely changed.

Refactoring tools should consider refactoring composition. Another interesting aspect that we learned is the relevance of the support for refactoring composition. That is, after applying a number of refactorings, a tool should be able to identify *new* refactoring opportunities. For instance, after recommending a move method from an origin to a destination class, a tool should be able to find that merging both classes would actually bring more significant benefits (according to some criteria). This became clear when a participant state:

> "*All refactoring recommendations regarding these two classes, would only make sense if we merge them both. Some of the recommendations would require calls to the original class. I particularly prefer the decomposition as it is, smaller classes, but I would have no objections for merging them.*"

Automatic generated test cases would make refactoring less risky. Finally, based on the results of the Focus Group session, when some participants mentioned that the low testing coverage made the acceptance of any refactoring recommendation *risky*, we envision that it would be interesting to integrate the evaluated tools with tools that could automatically generate test cases (e.g., [105, 106]) before and after the application of the refactoring. In this way, we could have given a bit more evidence that the recommend refactorings would not change the behavior of the system.

4.4.6 Threats to Validity

This study is empirical in nature, and as any empirical study, it has many limitations and threats to validity. As we studied only one software system, which is mostly written in

one programming language, we cannot generalize the results to other proprietary projects written in Java or in other programming languages. Nevertheless, since the system is composed of 26 subsystems, has a large codebase (more than 600k LOC), and an extensive change history (30K commits), we believe it is representative, and therefore the results of this study are useful and complement other studies that use either smaller proprietary systems or open-source software as target. Moreover, there are a plenty of refactoring tools available out there. However, in this work we focused on three tools that implement the move method/field refactoring. Our study does not also generalize to other tools that automate other refactorings. Regarding the Draco approach, we found that the number of its recommendations was lower than the recommendations from the other tools. While we can increase even more the number of recommendations adjusting some parameters (such as *support count* and *confidence*) to lower values, we believe that the values we used are a good trade-off between recommendation quality and quantity. However, we must note that the recommendations from Draco complements the recommendations from other tools. In fact, no recommendation from Draco (or Draco) was also suggested by the other tools. In that sense, we can say that Draco does not compete with another refactoring recommendation tool and therefore can be used to complement them.

Furthermore, we ran this study within the government agency that builds SIOP. Indeed, the author of this thesis works as a software developer in the agency, and also works with SIOP regularly. Therefore, the software developers that participated in the study could have been more positive towards the recommendations than what they actually are. Nevertheless, when it comes to accepting the refactoring recommendations, we perceived that the software developers were harsh in their decisions. Yet, we opted not to disclose to them the use of the other refactoring tools in order to avoid bias toward them (developers not always rely on static analysis tools [107]).

4.5 Qualitative Assessment 2: Brazilian Army Systems

This section presents our second qualitative assessment, where we executed the Draco approach in the context of two medium size enterprise systems: SISDOT and SISBOL.

4.5.1 Studied Systems

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 [108, 109].

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 [110]. Its current implementation comprises almost 20 contributors, 1130 commits, 20 KLOC of Java code and 10 KLOC of JavaScript code using the AngularJS framework.

4.5.2 Methodology

In this third qualitative 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.

4.5.3 Results

Table 4.15 summarizes the results of this second qualitative assessment. It is possible to realize that JMove presents the higher acceptance rate. The other tools recommended refactorings with an acceptance between 20% and 30%. Since the number of the recommendations of all tools were small, we were able to evaluate all kind of refactoring recommendations besides move method/field. However, the JDeodorant tool is the only one that recommends such refactorings.

Draco Analysis

Draco recommended five SISDOT refactorings using the Co-change dependencies variation, that includes the recommendations from the other two variations. One of them was accepted. The accepted recommendation involve a method that does no uses the "this" parameter, i.e. it uses only the declared parameters of the method. Furthermore, it calls two methods from another two classes, with one of them the destination of the move method refactoring recommendation. This situation is similar to the *feature envy*

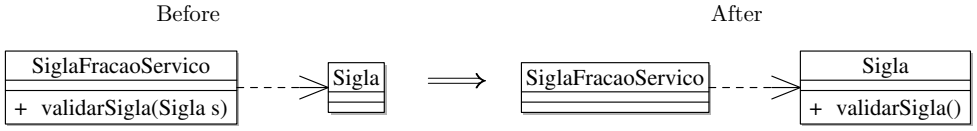
Table 4.15: Summary of the analysis for the refactoring recommended from four different tools. BMQ means Best Modularization Quality variation, PS means Pareto Set variation, and CCD means Co-change Dependencies variation.

Tool	Draco			REsolution	JDeodorant	JMove
	BMQ	PS	CCD			
SISDOT Recommendations	3	3	5	3	10	2
SISDOT Accepted	1	1	1	1	3	2
SISBOL Recommendations	2	2	2	0	1	0
SISBOL Accepted	0	0	0	0	0	0
Total Recommended	5	5	7	3	11	2
Total Accepted	1	1	1	1	3	2
Acceptance rate	20%	20%	14%	33%	27%	100%

bad smell. The refactoring removes a co-change dependency and does not create new static dependencies between the two classes. The first rejected recommendation involve an overrided method. The second rejected recommendation have a mixed responsibility, the piece of code related to one of this responsibilities could be moved to the target class that was recommended, however, the remaining source-code of the method, could not be moved, because it refers to a business context that does not corresponds to the target class. Accordingly, just a move method would not be enough, the developer must split the method in two before moving the method. Nevertheless, the refactoring recommendation exposed a design flaw that could be subject do discussion and a further refactoring. The third rejected recommendation involve a method that is cohesive and therefore is not a good candidate for moving. The fourth rejected recommendation was a move method between a data access class and a business service class. Therefore, they are in different tiers of the architecture and have different responsibilities. That was the reason for the rejection.

Draco recommended two SISBOL refactorings. All variations produced the same recommendations. 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 typical 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

Figure 4.9: A move method recommendation from REsolution. This recommendation led to an accepted pull-request



classes.

REsolution Analysis

Table 4.15 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 4.9). This recommendation improves cohesion and helps to avoid *anemic domain objects* [111]. 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.

JMove Analysis

As one can see from Table 4.15, 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.

JDeodorant Analysis

According to Table 4.15, 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 4.1. 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 4.1). 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 4.1: 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 4.15 also shows that the remaining recommendations from JDeodorant were all rejected. For instance, Listing 4.2 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 4.2: 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 4.3 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 4.3: 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`.

4.5.4 Discussion

Although we cannot generalize our findings (as discussed in the next section), the results of our qualitative study reveal that, for 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.

Besides the issue with recommendations that violate design constraints—reducing the number of accepted recommendations, 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 [112] to answer this question and to identify the number of refactorings performed in both systems during their development process. Table 4.16 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.

Table 4.16: 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

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 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 **RQ3** regarding the third study: the results of our third 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 between 20% and 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.

4.5.5 Threats to Validity

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 20%—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 [59]. 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 4.5, 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 [103]. 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 two 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 [94, 95])

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 two studied systems of this third 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.

4.6 Conclusion and Future Work

We quantitatively evaluated our approach using 47 open-source systems and found 8,405 evolutionary smells on all systems, and 4,844 refactoring recommendations on all systems. 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 state of the art refactoring recommendation tools (REsolution and JDeodorant) [6, 73].

In the qualitative study we applied four refactoring recommendation tools (JDeodorant, REsolution, JMove, and Draco) to the source code of three proprietary software system, named SIOP, SISDOT, and SISBOL. These tools provided over 500 refactorings recommendations. JDeodorant produced 364 recommendations, REsolution produced 151 recommendations, and Draco produced 31 recommendations. For SIOP, we then curated a list of 122 recommendations, and asked the software developers to assess these recommendations. And for SISDOT and SISBOL we submitted a set off pull-requests containing the refactoring recommendations.

Among the findings, in the SIOP case, we observed that recommendations from JDeodorant and Draco were more positively evaluated than the recommendations from REsolution. For instance, the software developers perceived improvements on flexibility in only 14% of the REsolution recommendations (for Draco and JDeodorant the results 37% and 43% respectively). Moreover, JDeodorant was also the one with the highest acceptance ratio (62% of the refactorings recommended by JDeodorant were accepted and integrated in SIOP, the acceptance ratio for Draco and REsolution are 38% and 24% respectively). We also find that Draco is suitable to complement other refactoring recommendation tools, because its recommendations are not produced by any other studied tool.

Sill in SIOP case, we learned that for Draco the main reason for rejection was recommendations involving methods that implement interfaces. This suggests that the Draco tool must check for such situation before recommending refactorings. Furthermore, our results suggest that although some refactoring tools provide a great number of refactoring recommendations, not all of these recommendations were perceived as useful. Moreover, by qualitatively investigating the reasons for not accepting these recommendations, we

ended up with a list of lessons learned that may be helpful not only for future improvements of Draco, but also for researchers and industry practitioners who want to build or extend their refactoring tools to be more aligned with software practice.

In the analysis of SISDOT and SISBOL 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.

Chapter 5

Conclusion

5.1 Summary

The "software aging" problem is well-know at least since the 70s [85]. Accordingly to Parnas [44], this problem leads to an increasing amount of effort to support older systems. The causes of this problem includes the need to accommodate changing requirements and the the (poor) quality of past modifications on the codebase. It is possible to ameliorate this problem by refactoring the source-code to improve its design quality, in order to smooth the introduction of future changes. Co-change dependencies are particularly relevant to this discussion because there are evidence that they lead to design problems [7, 23, 28, 51, 52]. Accordingly, we proposed and evaluated a new method to identify and remove "bad smells" related to co-change dependencies.

We described our method (and its variations) in Chapter 3. In short, our method comprises the following steps:

1. it produces a fine-grained change history of the source-code, where each change set refers to methods or fields instead of files;

2. it computes co-change dependencies, which results in a graph where nodes represent methods or fields and edges represent a co-change dependency between them (it uses the *support count* and *confidence* metrics to determine a co-change dependency);

3. optionally, it computes co-change clusters, using a multi-objective genetic algorithm;

4. it detects *evolutionary smells*, that are identified using one of the following criteria:

    - co-change clusters: if a co-change cluster contains methods or fields from different classes, and at least one of them does not have any dependency (static or co-change) upon another element from the same class where it is declared;

- co-change dependencies: if exists a co-change dependency between methods or fields from different classes, and at least one of them does not have any dependency upon another element from the same class;

5. it recommends **move method** or **move field** refactorings that remove such smells.

Furthermore, in the Draco tool, it is possible to choose between three approaches: (a) based on co-change clusters, using specifically the Best MQ (Modularization Quality) partition, or (b) based on co-change clusters, using all Pareto Set, or (c) based solely on Co-change Dependencies.

As seen in Chapter 4, our method is feasible and indeed could improve the software design quality, while complements other refactoring tools, since they do not have recommendations in common. We conducted a quantitative assessment using 47 open-source Java projects. We found a total of 8,405 evolutionary smells in all projects. We automatically computed 4,844 recommendations of move method/field refactorings. All the recommendations lead to design improvements (according to well-known metrics), without introducing any new static dependency.

We also conducted a qualitative study to explore how practitioners evaluate refactoring recommendations that aim to remove co-change dependencies, as well as how Draco compares to other approaches. The results reveal that the participants are willing to integrate 49% of the recommendations into the systems, which indicates that refactoring recommendation tools are effective in identifying opportunities for moving methods in industrial settings. In particular, Draco presents a performance comparable with the other tools. Our results also reveal that JDeodorant, REsolution, JMove, and Draco do not consider the design constraints of the systems—while recommending a transformation, which is the main reason that led the participants of our study to reject 51% of the transformations.

5.2 Contributions

Altogether, the contributions of this thesis are:

- A method (with three variations) 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 quantitative evaluation on over 47 non-trivial open-source projects showing the benefits of the proposed approach. We also compared our our approach with two state of the art methods for refactoring recommendation [6].

- A qualitative assessment of the application of four different refactoring recommendation tools (Draco, JDeodorant, JMove, and REsolution) in an industrial settings.
- A list of lessons learned that can aid researchers to further develop refactoring recommendation tools and practitioners to consider integrate the kind of tool in their daily activities.
- A publicly available tool and dataset that allows the reproduction of this study and that might be useful for researchers and practitioners alike.
- The Draco Clustering Tool (DCT), a public tool that performs automated Software Module Clustering using multi-objective genetic algorithms, as seen in Supplement I.

5.3 Limitations

The main limitations of Draco approach were explored in previous chapters. The summary of these limitations are below.

- The recommendation algorithm ignores design constraints. This limitation leads to recommendations that are likely to be rejected. It is important to note that this limitation also occurs in the other tools that we used in this thesis.
- Draco produces only one kind of refactoring recommendation (move method/field). In some cases, a merge/split class can be more meaningful than a set of move method refactorings.
- Draco only works in Java programs.
- Draco produces less recommendations than the compared tools. Since the number of recommendations are correlated with the improvement of the software design quality metrics, as seen in Chapter 4, more refactorings could lead to a better design.

Besides the aforementioned limitations, the Draco approach also have a limitation regarding the enforcement of refactoring engines preconditions. Kim et al.[83] describe the 10 preconditions that a move method refactoring engine should consider. Table 5.1 summarizes these preconditions and reports whether or not Draco takes them into account. Note that Draco enforces the majority of the preconditions or there is a strategy to allow the application of the refactoring. In Chapter 4 we shown some examples of how to apply a strategy to overcome preconditions unsatisfied by refactoring recommendations. In particular, the preconditions not enforced by Draco were not found in none of the manually evaluated refactoring recommendations in our empirical studies.

Table 5.1: Move Method Refactoring Preconditions Checks

Precondition	Enforced by Draco
Target has method	No, the workaround is to rename the method first
Is abstract	Yes, Draco ignores classes with subclasses
Is native	Yes, Draco only analyzes methods with source-code
Is constructor	Yes, Draco ignores constructors
Is inside interface	Yes, Draco ignores interfaces
The target is an interface	Yes, Draco ignores interfaces
Is polymorphic	No, the workaround is to create a delegate method first
The method references a type parameter of a generic class	No
The method is called with a null home value	No
The method references "super"	No

Nevertheless, as we have discussed, even when some preconditions are unsatisfied, the refactoring recommendation is still useful, since it can inspire reasoning about the quality of the design. We reported some examples of such situations in Chapter 4. Furthermore, Mongiovi et al. [103] report that refactoring engines might have overly strong preconditions preventing developers from applying useful transformations. As we have shown, this is the case of the refactoring recommendations from Draco.

5.4 Further Work

We envision the future work on recommending refactorings from co-change dependencies as stated bellow.

5.4.1 Improve precision

- To develop a precondition verification plugin that will allow to filter out the refactoring recommendations that does not satisfy some precondition. In fact, this plugin could be used by any refactoring recommendation tool besides the Draco tool.
- An investigation and further development of a tool to allow to specify design constraints and to verify if these constraints were satisfied by refactoring recommendation produced by tools—since we found that the majority of the rejection of recommendations were caused by a lack of knowledge of design constraints.

5.4.2 Improve expressiveness

- To explore different levels of abstractions, e.g., classes, packages, files.
- To explore other kinds of refactoring, such as split class and merge classes.
- An experiment using the Draco approach with systems written in different programming languages.

- Further investigate if weakening some refactoring constraints would increase the number of refactoring recommendations—and the number of accepted refactoring recommendations too.

5.4.3 Improve empirical soundness

- Conduct further research either to confirm or refute our findings that co-change dependencies might not be efficient for predicting bugs.

- Explore the correlation between co-change dependencies and more specific design constraints of a system (including the use of design patterns). We conjecture that some frameworks induce the existence of co-change dependencies between the entities needed to implement some feature supported by the framework. For example, if a framework defines two interfaces that are strongly related, and if it recommends to create two different classes that implements each interface, it is possible that these two classes would be co-change dependent. In the same sense, we envision the study of the correlation between the usage of design patterns and the occurrence of co-change dependencies between the involved entities.

References

[1] Parnas, David Lorge: *On the criteria to be used in decomposing systems into modules*. Communications of the ACM, 15(12):1053–1058, 1972. 1, 20

[2] Stevens, Wayne P., Glenford J. Myers, and Larry L. Constantine: *Structured design*. IBM Systems Journal, 13(2):115–139, 1974. 1

[3] Parnas, David Lorge: *Software aging*. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press, ISBN 0-8186-5855-X. `http://dl.acm.org/citation.cfm?id=257734.257788`. 1

[4] Hassan, Ahmed E and Richard C Holt: *Predicting change propagation in software systems*. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 284–293. IEEE, 2004. 1

[5] Fokaefs, M., N. Tsantalis, and A. Chatzigeorgiou: *Jdeodorant: Identification and removal of feature envy bad smells*. In *2007 IEEE International Conference on Software Maintenance*, pages 519–520, Oct 2007. 2, 25, 38

[6] WANG, Y., H. Yu, Z. l. Zhu, W. ZHANG, and Y. l. ZHAO: *Automatic software refactoring via weighted clustering in method-level networks*. IEEE Transactions on Software Engineering, PP(99):1–1, 2017, ISSN 0098-5589. 2, 21, 24, 38, 39, 41, 45, 47, 83, 86

[7] D'Ambros, M., M. Lanza, and R. Robbes: *On the relationship between change coupling and software defects*. In *2009 16th Working Conference on Reverse Engineering*, pages 135–144, Oct 2009. 2, 8, 9, 13, 14, 15, 16, 17, 18, 21, 38, 85

[8] Kirbas, Serkan, Alper Sen, Bora Caglayan, Ayse Bener, and Rasim Mahmutogullari: *The effect of evolutionary coupling on software defects: An industrial case study on a legacy system*. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM'14, New York, NY, USA, 2014. Association for Computing Machinery, ISBN 9781450327749. 2, 8, 9, 17, 18

[9] Steff, M. and B. Russo: *Co-evolution of logical couplings and commits for defect estimation*. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 213–216, 2012. 2, 9, 17, 18

[10] Ball, Thomas, Jung min Kim, Adam Porter, and Harvey Siy: *If your version control system could talk ...* In *ICSE '97 Workshop on Process Modeling and Empirical Studies of Software Engineering*, October 1997. 7, 8, 9, 22

[11] Zeller, Andreas: *Why programs fail: a guide to systematic debugging*. Elsevier, 2009. 7

[12] Beller, Moritz, Niels Spruit, Diomidis Spinellis, and Andy Zaidman: *On the dichotomy of debugging behavior among programmers*. In *Proceedings of the 40th International Conference on Software Engineering*, pages 572–583, 2018. 7

[13] Beller, Moritz, Georgios Gousios, and Andy Zaidman: *How (much) do developers test?* In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 559–562. IEEE, 2015. 7

[14] Pan, Kai, Sunghun Kim, and E James Whitehead: *Toward an understanding of bug fix patterns*. Empirical Software Engineering, 14(3):286–315, 2009. 7

[15] Murphy-Hill, Emerson, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan: *The design space of bug fixes and how developers navigate it*. IEEE Transactions on Software Engineering, 41(1):65–81, 2014. 7

[16] LaToza, Thomas D, Gina Venolia, and Robert DeLine: *Maintaining mental models: a study of developer work habits*. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM, 2006. 7

[17] Emam, Khaled El, Walcelio Melo, and Javam C. Machado: *The prediction of faulty classes using object-oriented design metrics*. J. Syst. Softw., 56(1):63–75, February 2001, ISSN 0164-1212. 8

[18] D'Ambros, Marco, Alberto Bacchelli, and Michele Lanza: *On the impact of design flaws on software defects*. In Wang, Ji, W. K. Chan, and Fei-Ching Kuo (editors): *Proceedings of the 10th International Conference on Quality Software, QSIC 2010, Zhangjiajie, China, 14-15 July 2010*, pages 23–31. IEEE Computer Society, 2010. https://doi.org/10.1109/QSIC.2010.58. 8

[19] Bacchelli, Alberto, Marco D'Ambros, and Michele Lanza: *Are popular classes more defect prone?* In Rosenblum, David S. and Gabriele Taentzer (editors): *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010. Paphos, Cyprus*, volume 6013 of *Lecture Notes in Computer Science*, pages 59–73. Springer, 2010. https://doi.org/10.1007/978-3-642-12029-9_5. 8

[20] Hassan, A. E.: *Predicting faults using the complexity of code changes*. In *2009 IEEE 31st International Conference on Software Engineering*, pages 78–88, 2009. 8

[21] Moser, R., W. Pedrycz, and G. Succi: *A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction*. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 181–190, 2008. 8

[22] Śliwerski, Jacek, Thomas Zimmermann, and Andreas Zeller: *When do changes induce fixes?* In *ACM sigsoft software engineering notes*, volume 30, pages 1–5. ACM, 2005. 8, 10

[23] Kouroshfar, Ehsan: *Studying the effect of co-change dispersion on software quality.* In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1450–1452. IEEE Press, 2013. 8, 21, 38, 85

[24] Oliva, G. A. and M. A. Gerosa: *Experience report: How do structural dependencies influence change propagation? an empirical study.* In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 250–260, 2015. 8, 9

[25] Costa, Daniel Alencar da, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan: *A framework for evaluating the results of the szz approach for identifying bug-introducing changes.* IEEE Transactions on Software Engineering, 43(7):641–657, 2017. 8, 10, 11

[26] Knab, Patrick, Martin Pinzger, and Abraham Bernstein: *Predicting defect densities in source code files with decision tree learners.* In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR'06, page 119–125, New York, NY, USA, 2006. Association for Computing Machinery, ISBN 1595933972. https://doi.org/10.1145/1137983.1138012. 8, 9

[27] Beyer, D. and A. Noack: *Clustering software artifacts based on frequent common changes.* In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 259–268, 2005. 9, 22

[28] Oliveira, Marcos César de, Rodrigo Bonifácio, Guilherme N. Ramos, and Márcio Ribeiro: *Unveiling and reasoning about co-change dependencies.* In *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, pages 25–36, New York, NY, USA, 2016. ACM, ISBN 978-1-4503-3995-7. 9, 13, 21, 23, 27, 38, 60, 85, 104

[29] Ajienka, Nemitari and Andrea Capiluppi: *Understanding the interplay between the logical and structural coupling of software classes.* Journal of Systems and Software, 134:120 – 137, 2017, ISSN 0164-1212. 9

[30] Silva, Luciana Lourdes, Marco Tulio Valente, and Marcelo de A. Maia: *Co-change Clusters: Extraction and Application on Assessing Software Modularity*, pages 96–131. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, ISBN 978-3-662-46734-3. https://doi.org/10.1007/978-3-662-46734-3_3. 9, 13, 21, 23, 27, 60

[31] Oliveira, Marcos César de, Davi Freitas, Rodrigo Bonifácio, Gustavo Pinto, and David Lo: *Finding needles in a haystack: Leveraging co-change dependencies to recommend refactorings.* Journal of Systems and Software, 158:110420, 2019, ISSN 0164-1212. http://www.sciencedirect.com/science/article/pii/S0164121219301943. 9, 11, 12, 38, 40, 104, 105, 110

[32] Graves, T. L., A. F. Karr, J. S. Marron, and H. Siy: *Predicting fault incidence using software change history.* IEEE Transactions on Software Engineering, 26(7):653–661, 2000. 9

[33] Rodríguez-Pérez, Gema, Gregorio Robles, and Jesús M González-Barahona: *Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm.* Information and Software Technology, 99:164–176, 2018. 10

[34] Williams, Chadd and Jaime Spacco: *Szz revisited: verifying when changes induce fixes.* In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36, 2008. 10

[35] Neto, Edmilson Campos, Daniel Alencar da Costa, and Uirá Kulesza: *Revisiting and improving szz implementations.* In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12. IEEE, 2019. 10, 11

[36] Herzig, Kim, Sascha Just, and Andreas Zeller: *It's not a bug, it's a feature: how misclassification impacts bug prediction.* In *Proceedings of the 2013 international conference on software engineering*, pages 392–401. IEEE Press, 2013. 10

[37] Rodríguez-Pérez, Gema, Jesús M Gonzalez-Barahona, Gregorio Robles, Dorealda Dalipaj, and Nelson Sekitoleko: *Bugtracking: A tool to assist in the identification of bug reports.* In *IFIP International Conference on Open Source Systems*, pages 192–198. Springer, 2016. 10

[38] Kalliamvakou, Eirini, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian: *The promises and perils of mining github.* In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 92–101, New York, NY, USA, 2014. Association for Computing Machinery, ISBN 9781450328630. https://doi.org/10.1145/2597073.2597074. 11

[39] Oliveira, Marcos César de, Rodrigo Bonifácio, Guilherme N. Ramos, and Márcio Ribeiro: *Unveiling and reasoning about co-change dependencies.* In Fuentes, Lidia, Don S. Batory, and Krzysztof Czarnecki (editors): *Proceedings of the 15th International Conference on Modularity, MODULARITY 2016, Málaga, Spain, March 14 - 18, 2016*, pages 25–36. ACM, 2016. https://doi.org/10.1145/2889443.2889450. 12

[40] Silva, L. L., M. T. Valente, M. de A. Maia, and N. Anquetil: *Developers' perception of co-change patterns: An empirical study.* In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 21–30, Sept 2015. 12, 27, 60

[41] Zimmermann, Thomas, Andreas Zeller, Peter Weissgerber, and Stephan Diehl: *Mining version histories to guide software changes.* IEEE Transactions on Software Engineering, 31(6):429–445, 2005. 12

[42] Silva, Luciana Lourdes, Marco Tulio Valente, and Marcelo de Almeida Maia: *Co-change clusters: Extraction and application on assessing software modularity.* LNCS Trans. Aspect Oriented Softw. Dev., 12:96–131, 2015. 12

[43] James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani: *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014, ISBN 1461471370. 15

[44] Parnas, David Lorge: *Software aging*. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press, ISBN 0-8186-5855-X. http://dl.acm.org/citation.cfm?id=257734.257788. 20, 38, 85

[45] Gurp, Jilles van and Jan Bosch: *Design erosion: problems and causes*. Journal of Systems and Software, 61(2):105 – 119, 2002, ISSN 0164-1212. http://www.sciencedirect.com/science/article/pii/S0164121201001522. 20

[46] Ahmed, I., U. A. Mannan, R. Gopinath, and 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)*, pages 1–10, Oct 2015. 20

[47] Silva, Lakshitha de and Dharini Balasubramaniam: *Controlling software architecture erosion: A survey*. Journal of Systems and Software, 85(1):132 – 151, 2012, ISSN 0164-1212. http://www.sciencedirect.com/science/article/pii/S0164121211002044, Dynamic Analysis and Testing of Embedded Software. 20

[48] Mitchell, Brian S. and Spiros Mancoridis: *On the automatic modularization of software systems using the bunch tool*. IEEE Trans. Softw. Eng., 32(3):193–208, March 2006, ISSN 0098-5589. 21, 22, 27, 103, 105, 107, 108, 109, 110

[49] Lin, Yun, Xin Peng, Yuanfang Cai, Danny Dig, Diwen Zheng, and Wenyun 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, pages 535–546, New York, NY, USA, 2016. ACM, ISBN 978-1-4503-4218-6. http://doi.acm.org/10.1145/2950290.2950317. 21

[50] Tsantalis, N. and A. Chatzigeorgiou: *Identification of move method refactoring opportunities*. IEEE Transactions on Software Engineering, 35(3):347–367, May 2009, ISSN 0098-5589. 21, 46

[51] Zhou, Daihong, Yijian Wu, Lu Xiao, Yuanfang Cai, Xin Peng, Jinrong Fan, Lu Huang, and Heng Chen: *Understanding evolutionary coupling by fine-grained co-change relationship analysis*. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, pages 271–282, Piscataway, NJ, USA, 2019. IEEE Press. https://doi-org.ez54.periodicos.capes.gov.br/10.1109/ICPC.2019.00046. 21, 38, 85

[52] Wiese, Igor Scaliante, Rodrigo Takashi Kuroda, Reginaldo Re, Gustavo Ansaldi Oliva, and Marco Aurélio 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*, pages 3–12. Springer, 2015. 21, 38, 85

[53] Mkaouer, Wiem, Marouane Kessentini, Adnan Shaout, Patrice Koligheu, Slim Bechikh, Kalyanmoy Deb, and Ali Ouni: *Many-objective software remodularization using nsga-iii.* ACM Trans. Softw. Eng. Methodol., 24(3):17:1–17:45, May 2015, ISSN 1049-331X. http://doi.acm.org/10.1145/2729974. 21, 24, 71

[54] Ouni, Ali, Marouane Kessentini, Houari Sahraoui, and Mohamed Salah 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*, pages 1461–1468. ACM, 2013. 21, 24

[55] Zimmermann, T., A. Zeller, P. Weissgerber, and S. Diehl: *Mining version histories to guide software changes.* IEEE Transactions on Software Engineering, 31(6):429–445, June 2005, ISSN 0098-5589. 22

[56] Wiggerts, T. A.: *Using clustering algorithms in legacy systems remodularization.* In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, WCRE '97, pages 33–, Washington, DC, USA, 1997. IEEE Computer Society, ISBN 0-8186-8162-4. 22

[57] Anquetil, Nicolas, Cédric Fourrier, and Timothy C. Lethbridge: *Experiments with clustering as a software remodularization method.* In *Proceedings of the Sixth Working Conference on Reverse Engineering*, WCRE '99, pages 235–, Washington, DC, USA, 1999. IEEE Computer Society, ISBN 0-7695-0303-9. 22, 105

[58] Maqbool, Onaiza and Haroon Babri: *Hierarchical clustering for software architecture recovery.* IEEE Trans. Softw. Eng., 33(11):759–780, November 2007, ISSN 0098-5589. 23, 105

[59] Candela, Ivan, Gabriele Bavota, Barbara Russo, and Rocco Oliveto: *Using cohesion and coupling for software remodularization: Is it enough?* ACM Trans. Softw. Eng. Methodol., 25(3):24:1–24:28, June 2016, ISSN 1049-331X. http://doi.acm.org/10.1145/2928268. 23, 29, 35, 82, 104, 105, 109

[60] Beck, Fabian and Stephan Diehl: *On the impact of software evolution on software clustering.* Empirical Software Engineering, 18(5):970–1004, 2013. 23, 27, 43, 60

[61] Fowler, Martin, Kent Beck, John Brant, William Opdyke, and Don Roberts: *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999. 23, 24, 30, 31, 57

[62] Khomh, Foutse, Massimiliano Di Penta, Yann Gaël Guéhéneuc, and Giuliano Antoniol: *An exploratory study of the impact of antipatterns on class change-and fault-proneness.* Empirical Software Engineering, 17(3):243–275, 2012. 23

[63] Marinescu, Radu: *Detection strategies: Metrics-based rules for detecting design flaws.* In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 350–359. IEEE, 2004. 23

[64] Van Emden, Eva and Leon Moonen: *Java quality assurance by detecting code smells.* In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 97–106. IEEE, 2002. 23

[65] Munro, Matthew James: *Product metrics for automatic identification of" bad smell" design problems in java source-code.* In *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 15–15. IEEE, 2005. 23

[66] Khomh, Foutse, Stéphane Vaucher, Yann Gaël Guéhéneuc, and Houari Sahraoui: *A bayesian approach for the detection of code and design smells.* In *2009 Ninth International Conference on Quality Software*, pages 305–314. IEEE, 2009. 23

[67] Rapu, D, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu: *Using history information to improve design flaws detection.* In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 223–232. IEEE, 2004. 23

[68] Palomba, F., G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia: *Mining version histories for detecting code smells.* IEEE Transactions on Software Engineering, 41(5):462–489, May 2015, ISSN 0098-5589. 23, 30

[69] Goldberg, David E: *E. 1989. genetic algorithms in search, optimization, and machine learning.* Reading: Addison-Wesley, 1990. 24, 27, 28, 106, 107

[70] Deb, K., A. Pratap, S. Agarwal, and T. Meyarivan: *A fast and elitist multiobjective genetic algorithm: Nsga-ii.* IEEE Transactions on Evolutionary Computation, 6(2):182–197, Apr 2002, ISSN 1089-778X. 24, 28, 107

[71] Deb, Kalyanmoy and Himanshu 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(4):577–601, 2013. 24

[72] Mazinanian, D., N. Tsantalis, R. Stein, and Z. Valenta: *Jdeodorant: Clone refactoring.* In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 613–616, May 2016. 25

[73] Tsantalis, N., T. Chaikalis, and A. Chatzigeorgiou: *Ten years of jdeodorant: Lessons learned from the hunt for smells.* In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 4–14, March 2018. 25, 38, 41, 83

[74] Charalampidou, Sofia, Apostolos Ampatzoglou, and Paris Avgeriou: *Size and cohesion metrics as indicators of the long method bad smell: An empirical study.* In *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE '15, pages 8:1–8:10, New York, NY, USA, 2015. ACM, ISBN 978-1-4503-3715-1. `http://doi.acm.org/10.1145/2810146.2810155`. 25

[75] Palomba, F., A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia: *The scent of a smell: An extensive comparison between textual and structural smells.* IEEE Transactions on Software Engineering, 44(10):977–1000, Oct 2018, ISSN 0098-5589. 25

[76] Fokaefs, M., N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou: *Jdeodorant: identification and application of extract class refactorings.* In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1037–1039, May 2011. 25

[77] Sales, V., R. Terra, L. F. Miranda, and M. T. Valente: *Recommending move method refactorings using dependency sets.* In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 232–241, Oct 2013. 25

[78] Terra, Ricardo, Marco Tulio Valente, Sergio Miranda, and Vitor Sales: *Jmove: A novel heuristic and tool to detect move method refactoring opportunities.* Journal of Systems and Software, 138:19 – 36, 2018, ISSN 0164-1212. `http://www.sciencedirect.com/science/article/pii/S0164121217302960`. 25, 40

[79] Bavota, Gabriele, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia: *Methodbook: Recommending move method refactorings via relational topic models.* IEEE Transactions on Software Engineering, 40(7):671–694, 2014. 25, 40

[80] Oliveira, Marcos César de, Rodrigo Bonifácio, Guilherme N. Ramos, and Márcio Ribeiro: *On the conceptual cohesion of co-change clusters.* In *Proceedings of the 2015 29th Brazilian Symposium on Software Engineering*, SBES '15, pages 120–129, Washington, DC, USA, 2015. IEEE Computer Society, ISBN 978-1-4673-9272-3. 27, 60

[81] Praditwong, Kata, Mark Harman, and Xin Yao: *Software module clustering as a multi-objective search problem.* IEEE Trans. Softw. Eng., 37(2):264–282, March 2011, ISSN 0098-5589. 28, 29, 104, 105, 106, 108

[82] Martin, Robert C.: *Design principles and design patterns*, 2000. `www.objectmentor.com`, online. 30

[83] Kim, Jongwook, Don Batory, Danny Dig, and Maider Azanza: *Improving refactoring speed by 10x.* In *Proceedings of the 38th International Conference on Software Engineering*, pages 1145–1156, 2016. 33, 87

[84] Brooks, Jr., Frederick P.: *The Mythical Man-Month: Essays on Softw.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1978, ISBN 0201006502. 38

[85] Van Gurp, Jilles, Jan Bosch, and Sjaak Brinkkemper: *Design erosion in evolving software products.* In *ELISA workshop*, volume 134, 2003. 38, 85

[86] Foucault, Matthieu, Marc Palyart, Xavier Blanc, Gail C Murphy, and Jean Rémy Falleri: *Impact of developer turnover on quality in open-source software.* In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 829–841. ACM, 2015. 38

[87] Kruchten, P., R. L. Nord, and I. Ozkaya: *Technical debt: From metaphor to theory and practice.* IEEE Software, 29(6):18–21, Nov 2012, ISSN 0740-7459. 38

[88] Samarthyam, Ganesh, Mahesh Muralidharan, and Raghu Kalyan Anna: *Understanding Test Debt*, pages 1–17. Springer Singapore, Singapore, 2017, ISBN 978-981-10-1415-4. https://doi.org/10.1007/978-981-10-1415-4_1. 38

[89] Pinto, Gustavo H. L. and Fernando Kamei: *What programmers say about refactoring tools?: an empirical investigation of stack overflow.* In *Proceedings of the 2013 ACM Workshop on Refactoring Tools, WRT@SPLASH 2013, Indianapolis, IN, USA, October 27, 2013*, pages 33–36, 2013. 38

[90] Dig, Danny: *The changing landscape of refactoring research in the last decade.* In *2nd IEEE/ACM International Workshop on API Usage and Evolution, WAPI@ICSE 2018, Gothenburg, Sweden, June 2, 2018*, page 1, 2018. https://doi.org/10.1145/3194793.3194800. 38

[91] Silva, Danilo, Nikolaos Tsantalis, and Marco Tulio Valente: *Why we refactor? confessions of github contributors.* In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 858–870. ACM, 2016. 40, 59

[92] Murphy-Hill, Emerson, Chris Parnin, and Andrew P Black: *How we refactor, and how we know it.* IEEE Transactions on Software Engineering, 38(1):5–18, 2011. 40, 59

[93] Kim, Miryung, Thomas Zimmermann, and Nachiappan Nagappan: *A field study of refactoring challenges and benefits.* In Tracz, Will, Martin P. Robillard, and Tevfik Bultan (editors): *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 50. ACM, 2012. https://doi.org/10.1145/2393596.2393655. 41

[94] Dias, Luiz Felipe, Igor Steinmacher, Gustavo Pinto, Daniel Alencar da Costa, and Marco Aurélio 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*, pages 473–477, 2016. 42, 82

[95] Dantas, R., A. Carvalho, D. Marcilio, L. Fantin, U. Silva, W. Lucas, and 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)*, pages 497–501. IEEE Computer Society, mar 2018. 42, 82

[96] Borges, Hudson, André C. Hora, and Marco Tulio 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*, pages 334–344, 2016. 43

[97] Pinto, Gustavo, Igor Steinmacher, and Marco Aurélio 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*, pages 112–123, 2016. 43

[98] Ray, Baishakhi, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov: *A large-scale study of programming languages and code quality in github*. Commun. ACM, 60(10):91–100, September 2017, ISSN 0001-0782. 43

[99] MacCormack, Alan, John Rusnak, and Carliss Y. Baldwin: *Exploring the structure of complex software designs: An empirical study of open source and proprietary code*. Manage. Sci., 52(7):1015–1030, July 2006, ISSN 0025-1909. 45

[100] Chidamber, S. R. and C. F. Kemerer: *A metrics suite for object oriented design*. IEEE Trans. Softw. Eng., 20(6):476–493, June 1994, ISSN 0098-5589. 45

[101] Bansiya, Jagdish and Carl G. Davis: *A hierarchical model for object-oriented design quality assessment*. IEEE Trans. Softw. Eng., 28(1):4–17, January 2002, ISSN 0098-5589. http://dx.doi.org/10.1109/32.979986. 45, 46

[102] Vakilian, Mohsen and Ralph E. Johnson: *Alternate refactoring paths reveal usability problems*. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1106–1116, 2014, ISBN 978-1-4503-2756-5. http://doi.acm.org/10.1145/2568225.2568282. 57

[103] Mongiovi, M., R. Gheyi, G. Soares, M. Ribeiro, P. Borba, and L. Teixeira: *Detecting overly strong preconditions in refactoring engines*. IEEE Transactions on Software Engineering, 44(5):429–452, May 2018, ISSN 0098-5589. 57, 82, 88

[104] Kontio, Jyrki, Laura Lehtola, and Johanna Bragge: *Using the focus group method in software engineering: obtaining practitioner and user experiences*. In *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE'04.*, pages 271–280. IEEE, 2004. 61

[105] Pacheco, Carlos and Michael D. Ernst: *Randoop: Feedback-directed random testing for java*. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM, ISBN 978-1-59593-865-7. http://doi.acm.org/10.1145/1297846.1297902. 72

[106] Fraser, Gordon and Andrea Arcuri: *Evosuite: Automatic test suite generation for object-oriented software*. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM, ISBN 978-1-4503-0443-6. http://doi.acm.org/10.1145/2025113.2025179. 72

[107] Johnson, Brittany, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge: *Why don't software architects use static analysis tools to find bugs?* In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press, ISBN 978-1-4673-3076-3. http://dl.acm.org/citation.cfm?id=2486788.2486877. 73

[108] Costa, Pedro Henrique Teixeira, Edna Dias Canedo, and Rodrigo 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, pages 102–111, New York, NY, USA, 2018. ACM, ISBN 978-1-4503-6554-3. http://doi.acm.org/10.1145/3267183.3267194. 73

[109] Oliveira, Caio Matheus Campos de, Edna Dias Canedo, Henrique Faria, Luis Henrique Vieira Amaral, and Rodrigo Bonifácio: *Improving student's learning and cooperation skills using coding dojos (in the wild!)*. In *IEEE Frontiers in Education*, pages 1–9. IEEE Computer Society, 2018. 73

[110] Erl, Thomas: *Soa: principles of service design*, volume 1. Prentice Hall Upper Saddle River, 2008. 74

[111] Fowler, Martin, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford: *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002. 76

[112] Tsantalis, Nikolaos, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig: *Accurate and efficient refactoring detection in commit history*. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 483–494, New York, NY, USA, 2018. ACM, ISBN 978-1-4503-5638-1. 79

[113] Dahiya, S. S., J. K. Chhabra, and S. Kumar: *Use of genetic algorithm for software maintainability metrics' conditioning*. In *15th International Conference on Advanced Computing and Communications (ADCOM 2007)*, pages 87–92, 2007. 103

[114] Lethbridge, T. C., J. Singer, and A. Forward: *How software engineers use documentation: the state of the practice*. IEEE Software, 20(6):35–39, 2003. 104

[115] Garlan, David: *Software architecture: a roadmap*. In *Proceedings of the Conference on the Future of Software Engineering*, pages 91–101, 2000. 104

[116] Huang, Jinhuang, Jing Liu, and Xin Yao: *A multi-agent evolutionary algorithm for software module clustering problems*. Soft Computing, 21(12):3415–3428, 2017. 104

[117] Chhabra, Jitender Kumar *et al.*: *Many-objective artificial bee colony algorithm for large-scale software module clustering problem*. Soft Computing, 22(19):6341–6361, 2018. 104

[118] Prajapati, Amarjeet and Jitender Kumar Chhabra: *Madhs: Many-objective discrete harmony search to improve existing package design*. Computational Intelligence, 35(1):98–123, 2019. 104

[119] Sun, Jiaze and Beilei Ling: *Software module clustering algorithm using probability selection*. Wuhan University Journal of Natural Sciences, 23(2):93–102, 2018. 104

[120] Bishnoi, M. and P. Singh: *Modularizing software systems using pso optimized hierarchical clustering*. In *2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*, pages 659–664, 2016. 104

[121] Singh, V.: *Software module clustering using metaheuristic search techniques: A survey.* In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 2764–2767, 2016. 104

[122] Oliveira Barros, Márcio de: *Evaluating modularization quality as an extra objective in multiobjective software module clustering.* In *International Symposium on Search Based Software Engineering*, pages 267–267. Springer, 2011. 105

[123] Monçores, Marlon C, Adriana CF Alvim, and Márcio O Barros: *Large neighborhood search applied to the software module clustering problem.* Computers & Operations Research, 91:92–111, 2018. 105, 107, 109

[124] Barros, Marcio de Oliveira: *An analysis of the effects of composite objectives in multiobjective software module clustering.* In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 1205–1212, 2012. 107, 109

[125] Marsaglia, George: *Xorshift rngs.* Journal of Statistical Software, Articles, 8(14):1–6, 2003, ISSN 1548-7660. `https://www.jstatsoft.org/v008/i14`. 108

[126] Mancoridis, Spiros, Brian S. Mitchell, Yih-Farn Chen, and Emden R. Gansner: *Bunch: A clustering tool for the recovery and maintenance of software system structures.* In *1999 International Conference on Software Maintenance, ICSM 1999, Oxford, England, UK, August 30 - September 3, 1999*, page 50. IEEE Computer Society, 1999. `https://doi.org/10.1109/ICSM.1999.792498`. 110

Supplement I

DCT: An Scalable Multi-Objective Module Clustering Tool

Abstract

Maintaining complex software systems is a time-consuming and challenging task. Practitioners must have a general understanding of the system's decomposition and how the system's developers have implemented the software features (probably cutting across different modules). Re-engineering practices are imperative to tackle these challenges. Previous research has shown the benefits of using software module clustering (SMC) to aid developers during re-engineering tasks (e.g., revealing the architecture of the systems, identifying how the concerns are spread among the modules of the systems, recommending refactorings, and so on). Nonetheless, although the literature on software module clustering has substantially evolved in the last 20 years, there are just a few tools publicly available. Still, these available tools do not scale to large scenarios, in particular, when optimizing multi-objectives. In this supplement we present the *Draco Clustering Tool* (DCT), a new software module clustering tool. DCT design decisions make multi-objective software clusterization feasible, even for software systems comprising up to 1,000 modules. We report an empirical study that compares DCT with other available multi-objective tool (HD-NSGA-II), and both DCT and HD-NSGA-II with mono-objective tools (Bunch and HD-LNS). We evidence that DCT solves the scalability issue when clustering medium size projects in a multi-objective mode. In a more extreme case, DCT was able to cluster Druid (an analytics data store) 221 times faster than HD-NSGA-II.

Keywords: Refactoring, co-change dependencies, remodularization, clustering, architecture quality

I.1 Introduction

With increasing complexity of modern software, there is an increased demand for automated tools to support the maintainability and scalability of those systems (Dahiya et al. [113]). Fundamental contributions to this subject include, for instance, the introduction of the automated Software Module Clustering (SMC) tool by Mitchell and Mancoridis [48]. This appliance began with the purpose to offer techniques to reveal the structure

of a software system by grouping its modules into clusters. They based their algorithm on the principle of "low coupling and high cohesion". The input of the algorithm is a set of modules and dependencies between them. Typically, these modules correspond to files (or classes, in object-oriented programming languages), and the dependencies correspond to function/method calls or variables/fields access. While this kind of modules and dependencies are common, other representations are useful too, such as methods/fields as modules and co-change dependencies [28].

Revealing the software structure by using SMC tools can help to overcome complications related to misleading or insufficient documentation. The problem with documentations is accurately comprehended in Lethbridge et al. [114], this study is consisted of interviews with software engineers, and the general answers about documentation was the following: documentation is frequently out of date, often poorly written, challenging in terms of finding useful content and has a considerable untrustworthy fraction. In this context, it becomes very meaningful the chase for computational mechanisms such as SMC so that the documentation gap could be filled, hence, making it possible to support the six main aspects of software development pointed out by Garlan [115]: understanding, reuse, construction, evolution, analysis, and management.

Besides software structure recovering, SMC techniques can also be used to: (a) recommend or reveal alternative decompositions [28], (b) recommend refactorings in order to conform to some alternative decomposition [31], and (c) detect anomalies in the software design [28, 31].

Past researches have proposed many alternative SMC approaches [81, 116–121], however, they failed to provide publicly available tools that use multi-objective genetic algorithms in their designs. One of the primary benefits of multi-objective algorithms is that they output a set of best solutions in contrast with mono-objective where there is only one "best" solution. The problem with pursuing only one solution is that we have to chose between conflicting objectives. For example, it is hard to chose between a solution with better cohesion or other with better coupling; i.e. for a SMC tool to find the best solution among several candidate solutions they have to decide about questions like "which is better: coupling or cohesion?" [59, 81].

In order to overcome this problem, in this supplement we present the Draco Clustering Tool (DCT), a public tool that performs automated SMC using multi-objective genetic algorithms.

I.2 Background and Related Work

The re-engineering process in large scale software projects requires appropriate and scalable techniques. With the focus on software module clustering (SMC) techniques, the work of Anquetil and Lethbridge [57], for instance, compares different strategies for using SMC as a software remodularization recommender. More recently, Maqbool and Babri [58] investigate the use of hierarchical clustering algorithms for architecture recovering.

Given this context, it is noticeable that the majority of SMC approaches use mono-objective algorithms. Praditwong et al. [81] proposed to represent the SMC problem as a multi-objective search problem. They formulated the problem representing separately several different objectives (including cohesion and coupling). The rationale of this proposal is that it is not always possible to capture the relative importance of some desirable properties (for example, it is hard to decide if cohesion is more important than coupling or vice-versa).

Candela et al. [59], investigated which properties developers consider relevant for a high-quality software remodularization. To be able to compare different properties, they had to use a multi-objective genetic algorithm to compute the software module clusters. Accordingly, they presented to the developers several recommendations of remodularization, and investigated which property (e.g. cohesion or coupling) the developers regard most. This kind of study was only possible by using a multi-objective SMC tool.

Other works are also worth mention here, because they provide different SMC implementations. First, M. Barros discusses the effects of using the MQ metric as an extra objective on a multi-objective SMC tools [122]. Second, Monçores et al. present a large study addressing a heuristic based on the mono-objective *Large Neighborhood Search* algorithm, applied to SMC problems [123]. Both works publish tools that we explored in this supplement. Finally, in a recent work, work [31] we leveraged a multi-objective software module clustering tool to produce a set of alternative decompositions of a software. Our needs to use a multi-objective approach to find these alternative decompositions, and the lack of scalable multi-objective SMC tools, motivated us to implement DCT.

I.3 Draco Clustering Tool

Draco Clustering Tool (DCT) is a command line interface (CLI) tool, that reads a *Module Dependency Graph* (MDG) [48] from the standard input and writes a clustered graph represented as a DOT[1] file in the standard output. It was implemented in Go[2] programming

[1] `https://graphviz.org/doc/info/lang.html`
[2] `https://golang.org`

language, and is publicly available.[3] A typical invocation of the tool looks like this:

```
$ clustering < software.mdg > software.dot
```

The main use case of the tool is to run experiments involving multi-objective SMC computation. Accordingly, the following principles guided the design of DCT:

- **An easy to use interface.** While a Graphical User Interface potentially could be more intuitive, it makes experiments automation more difficult;
- **Minimal memory usage.** DCT users might want to run the tool in parallel, so its memory consumption must be minimal;
- **Runtime efficiency.** Similarly, the time spent running a experiment must be minimal;
- **Extensible.** To experiment with multiple scenarios, it must be possible to replace portions of the clustering algorithm or to tune its parameters values;
- **Standard formats.** To make comparisons of DCT with other tools easier, DCT must adopt well-known file formats, both for input (MDG) and output (DOT);

In order to address these principles, we chose Go as programming language. Go programs are compiled ahead of time to native machine code, therefore compiled programs can execute efficiently. Furthermore, this property makes the use of CLI tools more convenient, since they would not require a virtual machine to run. In addition, we address the extensibility principle using Go interfaces. For instance, we have a Go interface to abstract the random number generator (see more details bellow).

In DCT we used the definition of the SMC problem as a multi-objective optimization problem, using the same set of objects recommended by Praditwong et al. [81]. The input is a MDG represented by a graph $G = (V, E)$ from a set of modules V and a set of dependencies $E \subseteq V \times V$; and the output is a set of solutions. A solution is a partition of a MDG that corresponds to a set of clusters. Although the original design of DCT uses a multi-objective genetic algorithm (GA) [69] to compute optimal partitions, it is also possible to extend DCT to use mono-objective algorithms.

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;

[3] `https://github.com/project-draco/tools/tree/master/clustering`

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*;

3. It proceeds until a stop condition is met.

In DCT, each GA component (e.g., the fitness function or the crossover operator) is defined as Go interfaces, which enables the replacement for other implementations. The default implementations of these interfaces are specified next.

The default DCT implementation relies on the multi-objective genetic algorithm NSGA-II [70], responsible to implement the selection operator of the GA.

When using multi-objective GAs, each individual has a vector of fitness values [69]. To compare two individuals, we use 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 [69] (this applies to optimizations where the goal is to maximize the objective values, if the goal is the opposite, we must invert the comparisons).

As such, we represent the individuals as a mapping from a module to the cluster it belongs to (typically a module represents a file or class). Technically, an individual is an array where each position corresponds to a module, and each value corresponds to a cluster. Two different modules belong to the same cluster when they refer to the same value. Figure I.1-(a) illustrates this representation, showing four modules (m1, m2, m3, m4, f1). All modules belong to the cluster C_0, except for m2 that belongs to the cluster C_1 (together with module f1).

Differently from previous works [48, 123, 124], DCT saves computer's main memory since the array is codified as a binary string (i.e., as a sequence of bits), as we can see in Figure I.1-(b). The maximum number of clusters is set to $\frac{|V|}{2}$, and each element of the array occupies $\left\lceil \log_2 \frac{|V|-1}{2} \right\rceil$ bits of the binary string—where V is the set of vertices of the MDG. Previous works represent the individual as an array of "integers" [48, 123, 124], which could place a toll on today processors that take 64 bits. For example, if we have a MDG with 10,000 vertices, one element of the array will occupy 13 bits, while the state of the art would occupy 64 bits.

The genetic operators transform the population through successive generations, maintaining the *diversity* and *adaptation* properties from previous generations. In this work, 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 bi-

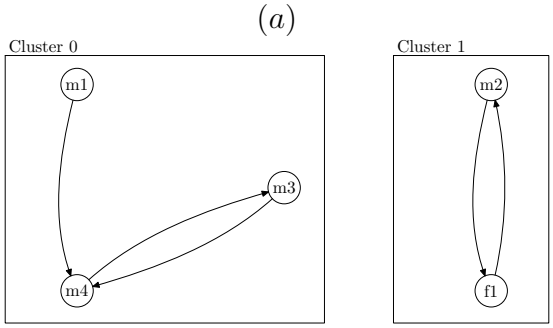


Figure I.1: Individual representation.

nary 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* $s' = 10111$. In DCT we used the Xorshift algorithm in order to generate random numbers; which is a known fast algorithm [125]. To the best of our knowledge, no other SMC tool uses this algorithm.

As mentioned before, we setup the GA to optimize five objectives [81]:

- maximize *Modularization Quality* (MQ);
- maximize intra-edge dependencies;
- minimize inter-edge dependencies;
- maximize number of clusters;
- minimize the difference between the maximum and minimum number of source-code entities in a cluster.

MQ was defined by Mitchell and Mancoridis [48] as follows:

$$MQ = \sum_{i=1}^{k} CF_i$$

$$CF_i = \begin{cases} \dfrac{\mu_i}{\mu_i + \frac{1}{2} \sum\limits_{\substack{j=1 \\ j \neq i}}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & \mu_i > 0 \\ 0 & \mu_i = 0. \end{cases}$$

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

With relation to the parameters, we chose their values similarly to Candela et al [59]. As such, given a software module 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:

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

In summary, DCT is a *full-fledged* multi-objective SMC tool written in the Go programming language, which (a) uses NSGA-II as default implementation, (b) employs a simple CLI to ease the execution of experiments, and (c) explores two optimization techniques: binary strings to represent individuals and the Xorshift random number generator algorithm.

I.4 Study Settings

This empirical assessment aims to evaluate the performance of DCT for clustering software systems of different sizes and complexities. We conducted two experiments. The first compares the performance of DCT against one software clustering tool that runs in a multi-objective mode (Heuristic Design NSGA-II [124]). The second compares the performance of DCT and HD-NSGA-II against two software clustering tools that use a mono-objective strategy (Bunch [48] and Heuristic Design LNS [123]). Although many research studies on software clustering are available in the literature, most of these publications do not provide tools we can use.

We investigate the following questions in our study:

(a) How does the complexity of the systems affect DCT performance?

(b) How does the DCT performance compare to the performance of multi-objective tools (HD-NSGA-II)?

(c) How does the performance of multi-objective tools (DCT and HD-NSGA-II) compare to the performance of mono-objective tools (Bunch and HD LNS)?

The multi-objective algorithm of DCT must explore a solution space of exponential complexity. As such, answering the first research question allows us to understand if DCT could be used to cluster software systems in real settings. Answering the second research question, allows us to understand the performance of DCT in comparison with another NSGA-II implementation. Finally, regarding the last research question, it is still unclear to what extent the use of multi-objective algorithms compromise the performance of publicly available SMC tools. Answering the last research question allows us to better estimate the effect of using a multi-objective algorithm to cluster software systems.

We leveraged three metrics to answer these research questions: **TS** is the elapsed time in seconds to cluster each studied system; **MMC** is the Maximum Memory Consumption (in KB) necessary to cluster each studied system; and **MQ** is a metric for estimating the Modularization Quality of the clusters [48, 126].

We ran Bunch and HD LNS tools with their default settings. On the other hand, HD-NSGA-II was not concluding the process even on small systems. To reduce the number of evaluations, we set the parameters *population size* and *maximum number of generations* to $2n$ and $4n$, respectively, where n is the number of vertices on the MDG. The default values of these parameters are $10p$ and $200p$, where p is the *package count*. The definition of *package* used in HD-NSGA-II corresponds to a `package` in the Java programming language. Furthermore, we had to write a tool to convert MDGs to the proprietary file format used by HD-NSGA-II. Finally, we ported the HD-NSGA-II and HD-LNS implementations to Java libraries and implemented a command line tool to execute both of them.[4]. We hope that this decision could help other researchers to experiment with these tools.

We used the `time` Linux tool to compute the first two metrics. To calculate the MQ metric we considered the outcomes of the clustering tools (Bunch, Heuristic Design, and DCT). We used a dataset of 17 MDGs in our study. These MDGs come from a convenient sample population of open source systems we used in a previous research work [31]. These systems are from different domains and range from small to medium size systems (in terms of lines of code). Moreover, we set 48h as the maximum execution time. Table I.1 presents some characteristics of these systems.

[4] `https://github.com/project-draco/cms_runner`

We executed our experiments using an Intel(R) Xeon(R) E-2124 CPU @ 3.30GHz with 32 GB of RAM, running a Linux Ubuntu distribution (18.04.4 LTS).

Table I.1: Projects used in the empirical assessments

System	Modules	Deps.	KLOC	Commits
React Native Framework	190	1006	48	7842
Storm distributed realtime system	388	3249	213	7451
Bigbluebutton web conf. system	497	3661	82	13420
Minecraft Forge	501	3403	72	5498
CAS - Enterprise Single Sign On	513	1718	87	6268
Atmosphere Event Driven Framework	658	3523	41	5748
Druid analytics data store	668	2648	297	7452
Liquibase database source control	716	3981	77	5360
Kill Bill Billing & Payment Platform	767	5422	139	5361
Actor Messaging Platform	768	7452	157	8772
The ownCloud Android App	833	3389	36	5329
Hibernate Object-Relational Mapping	836	2935	628	7302
jOOQ SQL generator	851	4118	133	5022
LanguageTool Style/Grammar Checker	871	1931	75	19121
Bazel build system	965	3813	375	7258
H2O-3 - Machine Learning Platform	1586	27725	143	19336
Jitsi communicator	2557	6742	326	12420

I.5 Results

In this section we highlight the main findings of our empirical study and provide answers to the research questions we introduced in Section I.4.

I.5.1 How does the complexity of the systems affect the DCT performance?

To answer this research question, we first considered the complexity of the MDGs (in terms of number of modules) as a model of the log of the elapsed time (TS) to compute the clusters. That is, we expressed this model as $log(TS) \approx Modules$. Considering the adjusted R^2, this model indicates that we can explain 88.87% of the TS variance as an exponential function on the number of modules. This exponential model better explains this variance, in comparison to a quadratic model ($R^2 = 0.73$) and a linear model ($R^2 = 0.38$).

In practice, DCT finds a cluster solution to a small system with 190 modules and 48 KLOC in `00:01:57` (React Native Framework), to a medium size system with 767 modules and 139 KLOC in `00:23:49` (Kill Bill Billing & Payment Platform), and to a large system with 2557 modules and 326 KLOC in `08:30:07` (Jitsi communicator). That is, although we confirmed the exponential cost necessary for DCT to compute the clusters (as a function on the number of modules), we argue that it can still be used in practice, particularly for small and medium size systems. For larger systems, DCT might

find a solution in an interval from hours to a few days (for extra large systems). So, regarding our first question, we found that:

> Our empirical assessment suggests that we can predict the time necessary for DCT compute a cluster using an exponential formula on the system's number of modules.

> In the longest scenario in our experiment, DCT found a cluster in 08:30:07 for a system with more than 2500 modules. We argue that this is still a reasonable time for running a SMC reengineering task on a large system using a multi-objective approach.

I.5.2 How does the DCT performance compare to the performance of multi-objective tools (HD-NSGA-II)?

Our goal to answer this question is to understand how DCT compares to another multi-objective SMC tool. Nonetheless, HD-NSGA-II only concluded the execution for seven (out of the 17 projects we consider in our study) within our maximum time threshold (48 hours). Considering only these seven projects, we realized a substantial benefit on the DCT speed-up, ranging from 2.13x to 221x (see Table I.2).

Table I.2: Comparison of the elapsed time to generate the clusters (considering the multi-objective tools DCT and HD-NSGA-II).

System	DCT (TS)	HD-NSGA-II (TS)	Speed-up
React Native	117	249	2.13x
Storm	228	12448	54.60x
Big Blue Button	442	36264	82.05x
Minecraft Forge	579	54691	94.46x
CAS Single Sign On	335	39963	119.29x
Atmosphere	970	90954	93.77x
Druid	741	164428	221.90x

Regarding the other metrics (MMC and MQ), DCT improved memory consumption up to 2x (minimum gain of 1.8x — see Table I.3) and slightly decreased the MQ metrics in six out of the seven cases (see Table I.4). Specifically, DCT presents a significant reduction on the time necessary to compute the clusters, in comparison to the HD-NSGA-II tool; however, we observed a slight reduction on the quality of the clusters. In the worst case, (Atmosphere project), DCT found a cluster with MQ = 69.64; while HD-NSGA-II found a cluster with MQ = 95.70. Altogether, we answer our second research question as follows:

Our assessment reveals that DCT scales better than HD-NSGA-II, finishing the clusterization process of the Druid tool in 741 seconds (while HD-NSGA-II needed 164 428 seconds). Considering larger projects, HD-NSGA-II did not finish the analysis within our maximum time threshold.

We observed that HD-NSGA-II clusters are slightly better than the clusters produced by DCT

Table I.3: Comparison of the memory usage generating the clusters (considering the multi-objective tools DCT and HD-NSGA-II).

System	DCT (MB)	HD-NSGA-II (MB)	Improv.
React Native	91	188	2.07
Storm	218	463	2.12
Bigbluebutton	282	511	1.81
Minecraft Forge	320	595	1.86
CAS - Enterprise Single Sign On	300	528	1.76
Atmosphere	416	741	1.78
Druid	396	713	1.80

Table I.4: Comparison of the clusters' MQ (considering the multi-objective tools DCT and HD-NSGA-II).

System	DCT (MQ)	HD-NSGA-II (MQ)	Improv.
React-native	39.27	33.57	1.17
Storm	60.40	66.60	0.91
Big Blue Button	71.15	79.31	0.90
Minecraft Forge	87.94	92.76	0.95
CAS - Enterprise Single Sign On	92.77	99.07	0.94
Atmosphere	69.64	95.70	0.73
Druid	122.65	128.00	0.96
Average			0.94

I.5.3 How does the performance of multi-objective tools (DCT and HD-NSGA-II) compares to the performance of mono-objective tools (Bunch and HD-LNS)?

The boxplots in Figure I.2 show the performance of the tools (DCT, HD-NSGA-II, Bunch, and HD-LNS), considering execution time (TS), memory consumption (MMC), and modularization quality (MQ). One could observe that multi-objective SMC implementations requires much more time to compute the clusters. In the worst scenario, DCT requires `00:40:48` while Bunch required `00:00:04`, and HD-LNS requires `00:02:57` on the same comparison.

Regarding memory consumption, the Bunch tool achieved the best performance, with an average memory consumption of ∼126MB; while HD-LNS achieved an average

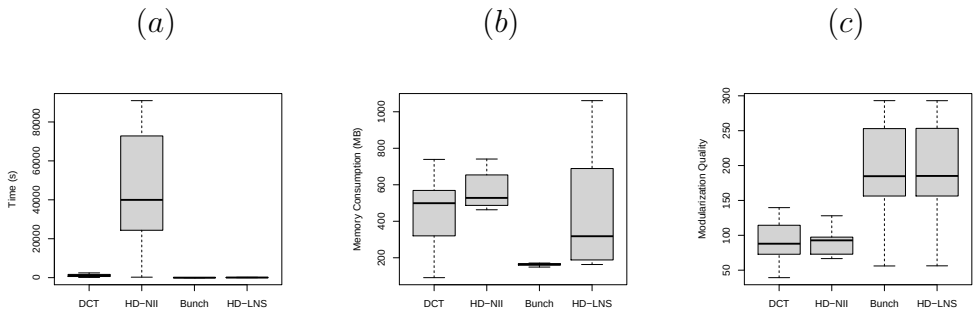


Figure I.2: Performance comparison of SMC tools. (a) Compares TS, (b) compares MMC, and (c) compares MQ. We removed the outliers in the boxplots.

consumption of ∼546MB. Considering the impact on the MQ metric, Figure I.2 shows a (median) decreasing of 44% on the modularization quality of the clusters from multi-objective SMC tools. Differently, the mono-objective tools preserve the average quality of the clusters Altogether, we answer our second research question as follows.

> The use of multi-objective SMC implementations brings a negative impact on both performance and modularization quality, in comparison with the multi-objective tools we used in our research. That is, on average, we found a central tendency of (a) increasing in 400x the time necessary to compute the cluster and (b) decreasing in 44% the modularization quality.

> Comparing to HD-LNS, Bunch brings significant improvements in two metrics (on average): time necessary to compute the clusters (up to 20x) and maximum memory consumption (up to 2x).

I.6 Final Remarks

In this supplement we presented a new Software Module Clustering tool that address scalability issues. This property is particularly important for running experiments that uses SMC tools as part of its process. In this use case, normally is required to repeatedly run several instances of the experiment. Accordingly, the tools' runtime efficiency is critical. We reported an comparison with other multi-objective SMC tool where we shown that our tool speeds up the elapsed time from 2 to 220 times, while using 2 times less memory and with a slightly decrease in MQ (6%). In the future, we will explore additional genetic algorithms, such as, NSGA-III, and pursue further optimizations.