

Dependent or Not: Detecting and Understanding Collections of Refactorings

Thiago do Nascimento Ferreira, James Ivers, Jeffrey J. Yackley, Marouane Kessentini and Ipek Ozkaya

Abstract—Refactoring is a program transformation to improve the internal structure of a program while preserving its external behavior. Developers frequently apply multiple refactorings that depend on each other to achieve goals such as improving code reusability. Although manually applying a sequence of dependent refactorings is a common practice, existing refactoring recommendation tools treat refactorings in isolation without revealing the dependencies among them to developers. One reason is that these relationships among refactorings are poorly understood. Current approaches treat refactoring recommendations as a strictly ordered sequence limiting developers' ability to understand, validate, and apply recommended refactorings. To address this gap, this paper describes a theory for reasoning about collections of refactorings through defining an ordering dependency relation among refactorings and organizing collection of refactorings as a set of refactoring graphs. We propose an algorithm for identifying refactoring dependencies and illustrate these concepts with a tool for visualizing such refactoring dependencies and refactoring graphs. Our validation results demonstrate that 43% of the 1,457,873 recommended refactorings from 9,595 projects that we studied are part of dependent refactoring graphs. Furthermore, refactorings are not only commonly involved in dependent relations, but also when applied, dependent refactoring graphs improve all of the quality attribute metrics in our experiments more than individual refactorings.

Index Terms—refactoring, search based software engineering, dependency

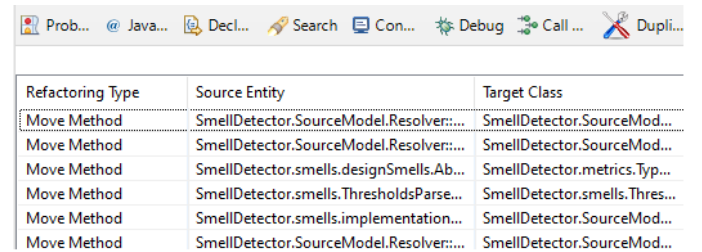
1 INTRODUCTION

EVEN for the most competent organizations, building and maintaining high performing software applications with high quality-code is a challenging and expensive endeavor [1]. Working in a fast-paced environment that demands frequent releases across several products and deployment environments, developers are often forced to compromise high quality standards in favor of meeting deadlines [2]. Lack of robust automated tools results in buggy and poor quality software that causes financial losses, high maintenance costs, increased fault-proneness, and delayed or canceled projects [3]. Software refactoring is widely recognized as an effective approach for maintaining high quality software by restructuring existing code without changing its external behavior [4].

Resolving code smells and broader code quality improvements across a project often requires applying multiple refactorings, which has lead to the creation of tools that recommend collections of refactorings to developers [5]. Manually applying a sequence of refactorings is common practice [6], [7], [5], however these tools treat each refactoring in the sequence in isolation. For instance, Cinnéide et al. [8] investigate the impact of only individual refactorings on quality attribute metrics, such as using Move Method to reduce the coupling of a class, without studying the impact of a sequence of refactorings. Bibiano et al. [6] evaluates the relationships between refactoring types (e.g., Move Class,

Extract Interface) and code smells. However, the study is based on the assumption that refactorings are only related if applied to the same code location, e.g., class. However, most refactoring types in fact modify multiple code fragments e.g., Move Method modifies two classes, the source and target of the move.

Current tools generate extensive lists of refactorings as recommendations to developers. However, these lists present refactorings as a sequence, with an implied strict ordering among refactorings. In practice, however, many of these orderings are not significant while others retain significant meaning. Developers lack a theory to tell the difference, which (among other things) makes refactoring recommendations harder to understand than is necessary. Figure 1 shows an output example of the refactoring recommendations of JDeodorant [9] where, similar to existing automated refactoring tools, the dependencies between the refactoring are not revealed, thus leaving the challenging task of interpreting the sequence of refactoring recommendations to the programmers.



Refactoring Type	Source Entity	Target Class
Move Method	SmellDetector.SourceModel.Resolver:...	SmellDetector.SourceMod...
Move Method	SmellDetector.SourceModel.Resolver:...	SmellDetector.SourceMod...
Move Method	SmellDetector.smells.designSmells.Ab...	SmellDetector.metrics.Typ...
Move Method	SmellDetector.smells.ThresholdsParse...	SmellDetector.smells.Thres...
Move Method	SmellDetector.smells.implementation...	SmellDetector.SourceMod...
Move Method	SmellDetector.SourceModel.Resolver:...	SmellDetector.SourceMod...

Fig. 1. Output Example of a refactoring recommendations tool: JDeodorant.

- Thiago do Nascimento Ferreira, Jeffrey J. Yackley and Marouane Kessentini are with the Department of Computer and Information Science at University of Michigan-Dearborn, Dearborn, USA. E-mail: thiagod@umich.edu, jyackley@umich.edu and marouane@umich.edu
- James Ivers and Ipek Ozkaya are with the Carnegie Mellon University Software Engineering Institute, Pittsburgh, USA. E-mail: jivers@sei.cmu.edu and ozkaya@sei.cmu.edu

Manuscript received ; revised

To close this gap, in this paper we describe a theory for

reasoning about collections of refactorings through a definition of ordering dependencies among refactorings and an algorithm for identifying these dependencies. We propose defining refactoring recommendations as sets of refactoring graphs rather than as refactoring sequences. We illustrate these concepts with a tool for visualizing refactoring dependencies and refactoring graphs.

Refactorings, when formalized, have clear pre-conditions defining the circumstances in which they can be applied and post-conditions defining the effects of applying them. For instance, one of the pre-conditions of a Move Method refactoring is that the method exists in the class from which it will be moved and one of its post-conditions is that the method must exist in the target class afterward. Therefore, a refactoring dependency exists when any post-condition of one refactoring matches any pre-condition of another refactoring e.g., the method exists in the relevant class. These linked refactorings then can be organized into groups based upon their dependencies. We represent these groups as directed acyclic graphs, where the nodes are the refactorings and the edges are the dependencies. This approach offers three main benefits: 1) developers can quickly and intuitively understand the dependencies among refactorings that constrain recommendations e.g., which refactorings must be done together; 2) developers can more easily compare recommendations by focusing on essential, rather than accidental, differences; and 3) tool builders can easily integrate new features to detect invalid refactorings and improve their recommendation algorithms.

We validated our proposed theory based on 1,457,873 refactorings recommended for 9,595 Java projects publicly available on GitHub. We considered 14 types of refactorings that are most commonly used in practice [10], [11]. We also developed a web tool, DPRef, that implements the proposed ordering dependency detection algorithm. It transforms a refactoring sequence recommended by existing refactoring tools [12], [9] into refactoring graphs based on the detected dependencies. We conducted experiments to evaluate the correctness of detected dependencies, discover what portion of refactorings in recommendations are actually dependent rather than independent, and estimate the potential impact of dependent refactorings on several quality attribute metrics. Finally, we conducted human validation with 27 developers to manually evaluate the correctness of the detected dependencies and their relevance.

Our implemented algorithm achieved 100% in correctly detecting all dependencies between refactorings and identifying invalid refactorings. Furthermore, our findings demonstrate that 43% of the 1,457,873 recommended refactorings are part of dependent refactoring graphs. This finding confirms that refactorings are commonly involved in dependent relations and cannot be applied truly independently. Furthermore, dependent refactorings improve all six QMOOD quality attribute metrics [13] in our experiments better than independent refactorings. The manual validation of the refactorings by 27 developers shows that all the identified dependencies are correct for a sample of 233 refactorings after applying them directly on the code of 61 open source projects based on the order proposed by DPRef. The post-study survey with the developers confirmed the relevance of detecting the dependencies to help them un-

derstand the sequence of recommended refactorings.

The authors also provide a **replication package**¹ that includes the refactoring dependency detection tool and necessary data for our large scale validation. The replication package will enable researchers and tool builders to integrate the refactoring dependency feature into existing refactoring recommendation and detection tools and further investigate the relationships among refactorings.

The rest of the paper is organized as follows: Section 2 discusses the related work and presents a motivating example; Section 3 provides our definitions of refactoring dependencies and an algorithm to detect them; Section 4 presents and discusses the obtained results; Section 5 highlights the threats to validity; Section 6 summarizes our research agenda; and Section 7 concludes.

2 RELATED WORK AND MOTIVATING EXAMPLE

Catalogs like Fowler's [4] have identified many types of refactorings e.g., Move Method, Extract Class, Pull Up Field, each of which is a semantics preserving code transformation that improves code structure. Developers routinely apply such refactorings in their day-to-day work, with modern development environments providing limited support for applying selected refactorings as directed by a developer. In this section, we survey two categories of refactoring research: recommendation tools and refactoring dependencies. We then summarize the challenges addressed in this paper through a motivating example.

2.1 Related Work

2.1.1 Refactoring Recommendations

There has been both industry and research interest in developing automated and semi-automated refactoring tools to support developers [14]. One representative example is JDeodorant, the tool proposed by Tsantalis and Chatzigeorgiou [9]. JDeodorant and similar recommendation tools [11], [15], [16], [17], [18], [19] generate recommendations as sequences of refactoring instances. The experiments described in this paper take this form of refactoring recommendation as input. Thus, our discussion in this section focuses on this category of studies. We point the interested reader to the survey by Bavota et al. [20] for an overview of approaches supporting code refactoring recommendations. In another refactoring recommendation tool, O'Keeffe and Cinnéide [21] formulate refactoring tasks as a multi-objective search problem to generate alternative designs by applying a sequence of refactoring operations. Such a search is guided by a quality evaluation function based on eleven object-oriented design metrics that reflect refactoring goals. Harman and Tratt [17] were the first to introduce the concept of Pareto optimality to search-based refactoring. They used it to combine two metrics, namely CBO (Coupling Between Objects) and SDMP (Deviation of Methods Per Class), into a fitness function and showed its superior performance as compared to a mono-objective technique [17]. The two aforementioned studies [21], [17] paved the way for several search-based approaches aimed at recommending refactorings [22], [23], [24], [15], [25], [12].

1. <https://sites.google.com/view/refactoringdependency>

A representative example of these search-based refactoring techniques is the work by Ouni et al. [12], who propose a multi-criteria code refactoring approach aimed at optimizing contrasting objectives: (i) minimizing the number of code smells; (ii) minimizing the refactoring cost (i.e., the number of recommended refactorings); (iii) preserving the design semantics (meaning considering textual information embedded in code identifiers and comments in the refactoring recommendation); and (iv) maximizing the consistency with code changes performed over the system's change history. In this study, we use the refactoring recommendations generated by this tool based on 1) its superior performance compared to the state of the art [12]; 2) the large number of supported refactoring types, and 3) its being publicly available. The contribution of this paper is not generating refactoring recommendations. Any refactoring recommendation approach can be used to generate the refactorings (input of our proposed approach) if they support some or all of the refactoring types summarized in Table 1.

2.1.2 Refactoring Dependencies

Chavez et al. [26] investigated how refactoring types affect five quality attributes based on the version history of twenty-three open source projects. They found that 94% of refactorings are applied to code with at least one low quality attribute value, with 65% of refactorings improving attributes and 35% of all refactorings being neutral on the system. Similarly, Cinnéide et al. [8] studied the impact of individual refactorings on quality attributes, such as using Move Method to reduce the coupling of a class. None of these studies considered the impact of a sequence of refactorings on the quality attributes.

Murphy-Hill et al. [7] investigated refactoring tool usage through both sampling developers' code and manually checking if their refactorings were performed with tool support and looked at 240,000 tool-assisted refactorings to find assumptions on how developers, in general, refactor code. They ultimately concluded that refactoring tools are rarely used by developers in practice with 90% of refactorings being performed manually, and that 40% of refactorings occur in batches.

Bibiano et al. [6] analyzed batch refactoring characteristics and their effects on code smells in open and closed source projects and concluded that 57% of batches/patterns are simple compositions of only two types of refactorings. They highlight lack of tool support to automatically detect refactoring dependencies as a barrier. However, this study is based on the assumption that refactorings are only related if applied to the same code location, which often is not the case for types of refactorings that modify multiple code fragments.

Mens et al. [27] analyzed dependencies at the model-level working with UML. Our work is at the code-level working directly with transformations on the code rather than on UML models where the type of refactorings are different and simplified when compared to the code-level ones. Furthermore, the authors were looking for mutually exclusive UML refactorings rather than detecting dependencies.

Overall, existing studies do not provide a rigorous definition of ordering dependencies among refactorings. They

mainly define what might be better considering similarity relations, such as a collection of refactorings that have similar effects (fixing a code smell) or similar context (applied by the same developer or to the same code location) [28], [29].

2.2 Motivating Example

The key to applying refactorings successfully is the decision of which refactorings to apply and where to apply them. In essence, this requires a developer to instantiate a refactoring by supplying the parameters that allow a type of refactoring to be unambiguously applied to code. For example, to instantiate a Move Method, a developer must supply parameters that indicate which method to move and where to move it. Throughout this paper, when we talk about refactorings and dependencies among refactorings, we are talking about refactoring instances.

While refactoring recommendations generated by tools to mimic this activity are typically represented as sequences, not all orderings in these sequences are significant. That is, the same code could be generated by two solutions that contain the same refactorings, but simply apply them in a different order. This is because while many refactorings are independent of one another, other refactorings are dependent on each other such that removing or reordering a refactoring from a solution could make other refactorings invalid. With the current growth of interactive tools to support refactoring [11], [18], developers are offered solutions that contain dozens to hundreds of refactorings and the option to selectively apply elements of a solution. Without a theory for reasoning about refactoring dependencies, developers can inadvertently make decisions (e.g., ignoring part of a solution) that result in failure (code cannot be successfully refactored) and enter a tedious trial and error loop. Making refactoring dependencies visible improves developers' understanding of how refactorings work together and allows them to make sound inferences regarding their application.

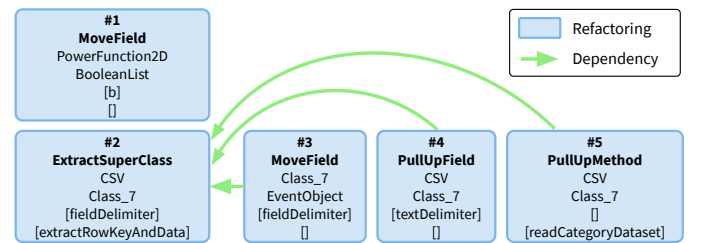


Fig. 2. A simplified solution of 6 refactorings for the JFreeChart project.

Figure 2 shows a simplified example of a solution composed of 5 refactorings to be applied to the JFreeChart project. Three of the refactorings (#3, #4, #5) depend on another refactoring (#2) because the Extract Super Class refactoring (#2) creates a new class (Class_7), on which refactorings #3, #4, and #5 operate. If the new class is not created first, then refactorings #3, #4, and #5 will fail. Thus, there exists an ordering dependency from each of #3, #4, #5 to #2. Refactoring #1, however, has distinct parameters, indicating that it operates on different code elements, thus it has no ordering dependencies on any others in this solution. Presenting these dependencies to a developer clarifies the

options that the developer has. For example, the developer could choose not to apply any refactoring except for refactoring #2 without consequences; if the developer chooses not to apply refactoring #2, then refactorings #3, #4, and #5 cannot be applied either. Detecting ordering dependency relationships among refactorings is essential to more effectively applying refactorings.

3 REFACTORING DEPENDENCY THEORY

The refactoring dependency theory for reasoning about collections of refactorings is built upon two concepts. The first is the definition of an ordering dependency relation among refactorings in a collection of refactorings. Pre- and post-conditions for refactoring types are used to detect refactoring dependencies. The second is the organization of a collection of refactorings as a set of refactoring graphs. Together, these concepts improve our ability to understand the meaning of collections of refactorings, allowable operations on them, and their composition in practice.

In this section we describe the elements of our proposed theory, the algorithm for detecting refactoring dependencies and an associated web tool that implements this detection algorithm.

3.1 Definitions

Our proposed dependency relation captures an ordering dependency between pairs of refactoring instances. Specifically, an *ordering dependency* ($rf_2 \mapsto rf_1$) between two refactoring instances (rf_1 and rf_2) exists when rf_2 can only be successfully applied after rf_1 has been applied. That is, rf_1 makes a change to code that is necessary in order to apply rf_2 . This condition can be evaluated based on the combination of pre- and post-conditions of the types of refactorings involved and the parameters of each refactoring instance. For example, to apply Move Method (a type of refactoring) to move method m_1 from class c_1 to class c_2 (m_1 , c_1 , and c_2 being the parameters of the refactoring instance), several preconditions must hold (e.g., m_1 , c_1 , and c_2 must all exist and m_1 must be defined on c_1). The pre- and post-conditions of each type of refactoring will be described in the next sub-section.

Building on this ordering dependency definition, we organize collections of refactorings as sets of refactoring graphs rather than as sequences of refactorings. A *refactoring graph* is a weakly connected directed acyclic graph composed of refactoring instance vertices and ordering dependency edges. Using the ordering dependencies as the basis for forming refactoring graphs (Algorithm 1) results in a set of graphs with the following traits:

- Each refactoring instance is an element of exactly one refactoring graph.
- Some graphs contain a single refactoring instance because that refactoring is truly independent of all others. We call these *trivial graphs* comprised of a single node of a refactoring instance.
- The remaining graphs contain multiple refactoring instances, each of which is part of one or more dependencies. We call these *non-trivial graphs*.
- Each refactoring graph is independent of every other graph in the solution.

3.1.1 Refactoring Pre- and Post-Conditions

As our approach for detecting ordering dependencies relies on the pre- and post-conditions of types of refactorings, we began with validated conditions in the current literature [12], [21], [4], [28] related to 14 types of refactorings. We selected the refactoring types summarized in Table 1 since they were those most frequently used in practice based on existing studies [10], [7], [26]. The pre- and post-conditions sets published in current literature were extensively validated for correctness and completeness [12], [21], [4], [28] thus they are not part of the contributions of this paper. It is out of the scope of this paper to generate automatically these conditions or re-validate them. The complete list of updated pre- and post-conditions for the 14 supported types of refactorings are organized in Table 1. In this table, the post-conditions that are presented are only those that represent a change. This is important for the dependency detection algorithm, allowing it to efficiently identify changes that enable pre-conditions of dependent refactorings. Also, the functions used for describing the pre- and post-conditions and their meanings are defined in [12].

3.2 Algorithm for Detecting Refactoring Dependencies

Algorithm 1 describes the process for detecting refactoring ordering dependencies. These dependencies are detected based on comparisons between pre- and post-conditions of refactoring instances. The proposed algorithm takes a list of refactoring instances as input and generates a set of refactoring graphs as output.

Lines 1 and 2 initialize the lists of refactoring instances (nodes of the graph, V) and refactoring dependencies (edges of the graph, E). Then, the post-conditions of each refactoring instance of the solution C (collection of refactorings) are evaluated for matching with the remaining refactoring instances in C (Lines 3–13). Specifically, the algorithm looks for any match between predicates of pre- and post-conditions from Table 1. That is, if any predicate of the post-condition of one refactoring (any element of P) matches any predicate of the pre-condition of another refactoring (any element of Q), then a dependency has been detected and an edge is added to the graph between those refactorings (Lines 5–10). We repeat this process until all the refactorings have been visited.

To illustrate Algorithm 1, consider the motivating example described in Figure 2. This example contains six refactoring instances:

```
#1 ExtractClass(OverwriteDataSet;Class_6;[x]; [addChangeListener])
#2 MoveField(PowerFunction2D;BooleanList;[b];[])
#3 ExtractSuperClass(CSV;Class_7;[fieldDelimiter]; [extractRowKeyAndData])
#4 MoveField(Class_7;EventObject;[fieldDelimiter];[])
#5 PullUpField(CSV;Class_7;[textDelimiter];[])
#6 PullUpMethod(CSV;Class_7;[];[readCategoryDataset])
```

Figure 3 illustrates the application of the algorithm to this example. In Step 1, the post-conditions of refactoring #1 are compared to the pre-conditions of all remaining refactorings. Since there is no match among pre- and post-conditions for any of them, no dependency is added. Next, the post-conditions of refactoring #2 are compared in Step 2 to the pre-conditions of the next refactorings in the sequence (#3 - #6). Again, no match is found. In Step 3, the post-conditions of refactoring #3 are compared to pre-conditions

TABLE 1
Refactoring types and their pre- and post-condition rules.

Refactoring	Cond.	Rules
Move Method (c1, c2, m)	Pre	exist(c1, c2, m) && NOT(inheritanceHierarchy(c1, c2)) && defines(c1, m) && NOT(defines(c2, sig(m)))
	Post	NOT(defines(c1, m)) && defines(c2, m)
Move Field (c1, c2, f)	Pre	exist(c1, c2, f) && NOT(inheritanceHierarchy(c1, c2)) && defines(c1, f) && NOT(defines(c2, f))
	Post	NOT(defines(c1, f)) && defines(c2, f)
Pull Up Field (c1, c2, f)	Pre	exist(c1, c2, f) && isSuperClassOf(c2, c1) && defines(c1, f) && NOT(defines(c2, f))
	Post	defines(c2, f) && NOT(defines(c1, f))
Pull Up Method (c1, c2, m)	Pre	exist(c1, c2, m) && isSuperClassOf(c2, c1) && defines(c1, m) && NOT(defines(c2, sig(m)))
	Post	defines(c2, m) && NOT(defines(c1, m))
Push Down Field (c1, c2, f)	Pre	exist(c1, c2, f) && isSuperClassOf(c1, c2) && defines(c1, f) && NOT(defines(c2, f))
	Post	defines(c2, f) && NOT(defines(c1, f))
Push Down Method (c1, c2, m)	Pre	exist(c1, c2, m) && isSuperClassOf(c1, c2) && defines(c1, m) && NOT(defines(c2, sig(m)))
	Post	defines(c2, m) && NOT(defines(c1, m))
Extract Class (c1, c2, {elements})	Pre	exist(c1) && NOT(exist(c2)) && methods(c1) > 2 && $\forall e \in \text{elements: defines}(c1, e)$
	Post	exist(c2) && $\forall e \in \text{elements: defines}(c2, e)$ && NOT(defines(c1, e))
Extract Super Class (c1, c2, {elements})	Pre	exist(c1) && NOT(exist(c2)) && methods(c1) > 2 && $\forall e \in \text{elements: defines}(c1, e)$
	Post	exist(c2) && isSuperClassOf(c2, c1) && $\forall e \in \text{elements: defines}(c2, e)$ && NOT(defines(c1, e))
Extract Sub Class (c1, c2, {elements})	Pre	exist(c1) && NOT(exist(c2)) && methods(c1) > 2 && $\forall e \in \text{elements: defines}(c1, e)$
	Post	exist(c2) && isSuperClassOf(c1, c2) && $\forall e \in \text{elements: defines}(c2, e)$ && NOT(defines(c1, e))
Encapsulate Field (c1, f)	Pre	exist(c1) && defines(c1, f) && NOT(defines(c1, set_f)) && NOT(defines(c1, get_f))
	Post	defines(c1, set_f) && defines(c1, get_f)
Decrease Field Security (c1, f, newSecLvl)	Pre	exist(c1) && defines(c1, f) && ((newSecLvl == "public" $\implies$ (NOT(isPublic(f)) && isProtected(f))) && (newSecLvl == "protected" $\implies$ (NOT(isProtected(f)) && NOT(isPublic(f)) && isPrivate(f))))
	Post	(newSecLvl == "public" $\implies$ isPublic(f)) (newSecLvl == "protected" $\implies$ isProtected(f))
Decrease Method Security (c1, m, newSecLvl)	Pre	exist(c1) && defines(c1, m) && ((newSecLvl == "public" $\implies$ (NOT(isPublic(m)) && isProtected(m))) && (newSecLvl == "protected" $\implies$ (NOT(isProtected(m)) && NOT(isPublic(m)) && isPrivate(m))))
	Post	(newSecLvl == "public" $\implies$ isPublic(m)) && (newSecLvl == "protected" $\implies$ isProtected(m))
Increase Field Security (c1, f, newSecLvl)	Pre	exist(c1) && defines(c1, f) && ((newSecLvl == "private" $\implies$ (NOT(isPrivate(f)) && isProtected(f))) && (newSecLvl == "protected" $\implies$ (NOT(isProtected(f)) && NOT(isPrivate(f)) && isPublic(f))))
	Post	(newSecLvl == "private" $\implies$ isPrivate(f)) && (newSecLvl == "protected" $\implies$ isProtected(f))
Increase Method Security (c1, m, newSecLvl)	Pre	exist(c1) && defines(c1, m) && ((newSecLvl == "private" $\implies$ (NOT(isPrivate(m)) && isProtected(m))) && (newSecLvl == "protected" $\implies$ (NOT(isProtected(m)) && NOT(isPrivate(m)) && isPublic(m))))
	Post	(newSecLvl == "private" $\implies$ isPrivate(m)) && (newSecLvl == "protected" $\implies$ isProtected(m))

Algorithm 1: Dependency Detection Algorithm.

Input: refactoring solution $C = \{r_1, r_2, r_3, \dots, r_n\}$
Output: forest of refactoring graphs
 $F = \{f_1, f_2, f_3, \dots, f_m\}$

```

1  $V \leftarrow \emptyset, E \leftarrow \emptyset;$ 
2 foreach  $r_i \in C$  do
3    $V \leftarrow V \cup r_i;$ 
4    $P \leftarrow \text{post\_conditions}(r_i);$ 
5   foreach  $r_j \in C \mid j > i$  do
6      $Q \leftarrow \text{pre\_conditions}(r_j);$ 
7      $M \leftarrow P \cap Q;$ 
8     if  $|M| \neq 0$  then
9        $E \leftarrow E \cup \{r_j, r_i\};$ 
10    end
11  end
12 end
13  $G \leftarrow (V, E);$ 
14  $F \leftarrow \text{partition}(G);$ 
15 return  $F;$ 

```

of refactorings #4, #5, and #6. For each, a match $\text{exist}(\text{Class_7})$ is found. Thus, three dependencies are added, from each of #4, #5, and #6 to #3. The algorithm continues, but no additional matches are found. Thus, Algorithm 1 transforms this sequence into three refactoring graphs (two trivial graphs and one non-trivial graph that includes four refactorings).

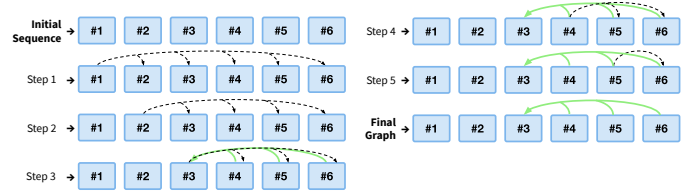


Fig. 3. Execution of Algorithm 1 on the example of Figure 2.

3.3 DPREf

To validate our definitions and algorithm and to make our refactoring dependency detection approach available to the community, we implemented DPREf², a free and open-source web platform that allows users to provide a sequence of refactorings to be applied as input and generates a set of refactoring graphs as output.

Figure 4 shows DPREf's output, which includes the generated set of refactoring graphs and associated information like the impact on quality attributes for each graph or individual refactoring. Developers are able to change the generated refactoring graphs and apply filters to show, for instance, just the graphs that significantly improve specific

2. <https://iselab-dpref.herokuapp.com/>

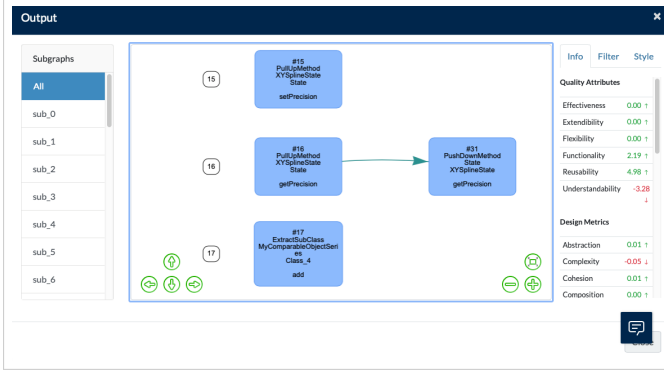


Fig. 4. DPREf, a tool for detecting refactoring dependencies.

quality attributes. DPREf also includes an Eclipse plug-in to execute any refactorings selected by the user on the actual source code. The tool is available as part of our replication package.

4 EMPIRICAL STUDY

In this section, we present our research questions, validation methodology, experimental setup, and discuss our findings.

4.1 Research Questions

The following research questions guide the evaluation of our refactoring dependency theory:

- RQ1.** (Precision) To what extent are the detected refactoring ordering dependencies correct?
- RQ2.** (Relation) To what extent are refactorings dependent?
- RQ3.** (Improvement) To what extent do non-trivial refactoring graphs improve quality attributes compared to trivial refactoring graphs (independent refactorings)?

We collected data from 9,595 open source repositories to evaluate the correctness of the detected ordering dependency relationships among refactorings. For each project, we executed existing refactoring recommendation tools [12], [9] to find relevant refactorings to be applied on the code of those projects. In this study, we use the refactoring recommendations generated by those tools based on their superior performance compared to the state of the art with over 90% of precision, recall and manual correctness based on large open source and industry projects; the large number of supported refactoring types; and its being publicly available. We describe the parameter settings used by such refactoring recommendation tools in the next section. Finally, we detected dependencies among refactorings and generated refactoring graphs based on Algorithm 1 for all generated refactoring solutions to understand if and how refactorings are applied together rather than in isolation. Finally, we studied and compared the impact of refactoring graphs on different well-defined quality attributes, based on the QMOOD model [13] detailed in Table 2.

To answer **RQ1**, we use two methods: automated and manual correctness. We used these complementary methods because the manual evaluation can be error-prone, time-consuming and not scalable while the automated evaluation may lack insights and feedback from developers.

TABLE 2
QMOOD quality metrics.

QMOOD Metrics	Definition / Computation
Reusability	$-0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * DesignSize$
Flexibility	$0.25 * Encapsulation - 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism$
Understandability	$-0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism - 0.33 * Complexity - 0.33 * DesignSize$
Functionality	$0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * DesignSize + 0.22 * Hierarchies$
Extendability	$0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism$
Effectiveness	$0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism$

For automated correctness, we removed refactorings from *valid* non-trivial graphs and determined whether the removal invalidated the graph. We define a valid graph as a refactoring graph for which all pre-conditions of all refactorings hold (Table 1). As defined earlier, if an ordering dependency exists between two refactorings (a head refactoring depends on a tail refactoring), then the head refactoring can only be successfully applied after the tail; removing any tail refactoring should then invalidate at least one pre-condition of a head refactoring. As such, our specific test was to remove one tail refactoring from each valid non-trivial graph with more than two refactorings and then count the number of valid and invalid non-trivial graphs. To answer **RQ1**, we calculated the Rate of Correctness (RC) after removing one tail refactoring from all non-trivial graphs as follows:

$$RC = \frac{\# \text{ of Invalid Non-trivial Graphs}}{\# \text{ of Non-trivial Graphs}} \quad (1)$$

Our assertion is that the refactoring dependency detection algorithm is correct if all non-trivial graphs become invalid if at least one tail refactoring is removed from each, as above.

For the manual evaluation, we asked 27 full-time developers to manually check the correctness of 50 valid non-trivial graphs totaling 233 refactorings for 5 open source projects³⁴⁵⁶⁷ that contained at least 5K LOC and involved significant refactorings in the last 2 years. The participants were asked to use our tool to identify refactoring dependencies, assess the correctness of those dependencies, and apply and compile the refactorings. When checking correctness, developers were asked to evaluate each refactoring independently to determine whether each identified dependency was necessary and whether there were any missing dependencies. We defined a valid refactoring in this exercise as a refactoring for which the set of dependencies was both

3. <https://github.com/phunware/maas-ads-android-sdk>
4. <https://github.com/solita/query-utils>
5. <https://github.com/forgo/roaster>
6. <https://github.com/goobi/goobi-ugh>
7. <https://github.com/kongchen/swagger-maven-example>

correct and complete and a manual correctness score (MC) as:

$$MC = \frac{\# \text{ of Valid Refactorings}}{\# \text{ of Evaluated Refactorings}} \quad (2)$$

To answer **RQ2**, we calculated the number of dependencies (edges) and graphs (trivial and non-trivial) for all projects. We also counted the number of refactorings in non-trivial graphs and the most frequently occurring refactoring types in them, as well as the Non-Trivial Rate (NTR) defined as follows:

$$NTR = \frac{\# \text{ of Refactorings in Non-trivial Graphs}}{\# \text{ of Refactorings}} \quad (3)$$

These evaluation metrics allow us understand the extent of refactoring dependencies. Furthermore, we can evaluate the refactoring types that are less commonly applied in isolation and also understand the complexity of the non-trivial graphs based on their sizes.

To answer **RQ3**, we compared the number of graphs that improve the quality attributes and design metrics from QMOOD [13] for both trivial and non-trivial graphs. We also considered the rates of improvement, in percentage, for all graphs taking into consideration the reusability, flexibility, understandability, functionality, extendibility, and effectiveness quality attributes captured by QMOOD metrics and available in Table 2, as well as, basic metrics such as coupling, cohesion, etc. We also calculated a Total Quality Index (TQI), aggregating all the metrics, after normalization, with equal weights into one metric.

These evaluation metrics are useful to understand the impact of collections versus individual refactorings on improving the quality and which quality attributes are more likely to be significantly improved using non-trivial graphs or independent refactorings.

4.2 Experimental Settings

We considered a total of 9,595 open-source Java projects to address the above research questions. The selection process limited consideration to projects with $\geq 5k$ LOC and at least 2 collaborators. We also eliminated any duplicate (cloned) projects from consideration. We applied these criteria on the list of one million GitHub projects provided by [30]. We performed this selection process in an attempt to eliminate small projects, such as student projects and small hobby/learner programs, that were not likely to be good candidates for refactorings.

Table 3 shows the min, average, and max for the number of the collaborators, code size (in LOC), # of classes, and # of recommended refactorings generated. The list of subject projects is also available in the replication package, along with all results (e.g., refactorings, quality metrics, refactoring graphs, etc.).

The total number of refactorings collected from recommendations for these 9,595 projects is almost 1.5 million (1,457,873 refactorings). We used the parameter settings recommended by the authors of the refactoring recommendations tool [12]: Single Point crossover with probability = 0.7, Bit Flip mutation with probability = 0.4, and stopping

TABLE 3
Statistics of the subject projects.

Metric	Min	Average	Max
Collaborators	2	2.6	65
Code Size	5.0k	25.9k	4,997.7k
# of Classes	1	931.3	23802
# of Refactorings	150	151.9	200

criterion was set to 100,000 evaluations. We also set the initial population size to 100 and utilized a tournament selection operator with $n=2$. The minimum and maximum number of refactorings per solution are limited to 150 and 200, respectively.

For the manual validation of the refactoring dependencies, we recruited 27 full-time developers from our networks, each of whom was unaware of our algorithm. These participants were first asked to fill out a pre-study questionnaire containing six questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, and their familiarity with software refactoring. The list of questions of all the questionnaires and the obtained results can be found in the online appendix. Although the vast majority of participants were already familiar with refactoring as part of their jobs and graduate studies, all the participants attended a two-hour lecture on refactoring by two the organizers of the experiments. The details of the selected participants can be found in Table 4, including their years of programming experience, familiarity with refactoring, etc.

TABLE 4
Selected Participants.

System	# of Subjects	Prog. Exp. (Years) [Min-Avg-Max]	Avg. Ref. Exp.
Dataset #1	5	[6.0 - 5.5 - 7.5]	High
Dataset #2	5	[5.5 - 7.0 - 8.0]	High
Dataset #3	7	[6.5 - 7.5 - 10.5]	High
Dataset #4	5	[6.0 - 6.5 - 8.5]	High
Dataset #5	5	[6.5 - 7.5 - 9.0]	High

All participants had a minimum of 6 years experience and work as active programmers with strong backgrounds in refactoring, Java, and software quality metrics. We divided the participants into 5 datasets where each dataset contains 10 samples of valid non-trivial graphs. We selected these samples based on the distribution of the refactoring types and number of refactorings in each graph as described in Figure 5.

Each participant was asked to assess the correctness of the refactoring dependencies using our DPREf tool. They were asked to execute the sequence of refactoring using our Eclipse plug-in and compile the code after applying the refactorings of each valid non-trivial graph. In addition to evaluating the refactorings, the participants were asked to configure, run, and interact with the tool on the different systems. We assigned tasks to the participants according to the datasets and developers' experience. Each participant was given a post-study survey. This second survey was more general as it collected the practitioners' opinions and

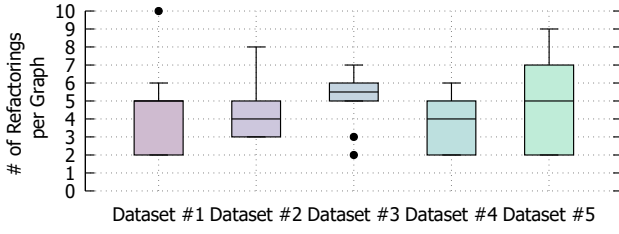


Fig. 5. Number of refactorings per non-trivial graph in each dataset.

their perception of the importance and relevance of detecting refactoring dependencies along with the usability of DPRef. The full details of our pre-study survey results can be found in the replication package.

4.3 Results and Discussions

4.3.1 Results for RQ1

Figure 6 summarizes the distribution of the results of removing one tail refactoring from all non-trivial graphs with more than two refactorings. One important outcome is that all non-trivial graphs become invalid after removing a tail refactoring (RC), which confirms that the proposed algorithm accurately identifies refactoring dependencies. The RC evaluation metric has a value of 1.0 (or 100%) across all the 9,595 projects, i.e., for all non-trivial graphs. The total number of non-trivial graphs is 257,725 with an average of 26.8 per project.

Another interesting result was that our algorithm detected many invalid refactorings among the solutions generated by the tool of Ouni et al. [12]. An average of 6.2 invalid non-trivial graphs were identified per project. The main reason is that the crossover operator used to exchange refactorings between solutions did not ensure that refactoring pre-conditions would remain satisfied after the exchange. As discussed in the future work section, the theory proposed in this paper can be integrated into existing refactoring recommendation tools to improve their correctness and contribute to the definition of intelligent change operators (including crossover) for search-based refactoring. Non-trivial graphs that were initially invalid were excluded from the calculation of RC and the removal of refactorings shown in Figure 6.

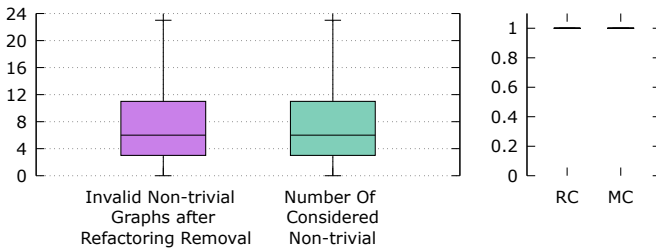


Fig. 6. Boxplots of refactoring dependency correctness for the 9,595 projects.

For the manual evaluation, all the non-trivial graphs were correctly executed by the participants on the open source projects and they agreed that the dependencies were

correctly and completely identified for each refactoring, thus the MC scores were 100% on all the selected datasets as shown in Figure 6. Thus, the manual correctness results confirm the automated method.

Maturation of automated tools to assist developers with complex tasks such as refactoring is an important gap to fill. The developers that we surveyed also agree with this. All the participants rated the importance of detecting refactoring dependencies as important or rather important, which confirms the need for tools to detect dependencies in a refactoring sequence. The vast majority of participants (24 out of 27) also rated the task of manually identifying these dependencies as difficult or very difficult based on their experiences in using existing refactoring tools. These developers described that the most important motivations of automatically detecting dependencies is to understand how they can fix code smells (typically fixed using a sequence of refactorings) and also to reduce the refactoring effort by applying refactorings incrementally.

We asked also the participants to rate their agreement on a Likert scale from 1 (complete disagreement) to 5 (complete agreement) with the following question on how important it is to detect refactoring dependencies automatically.

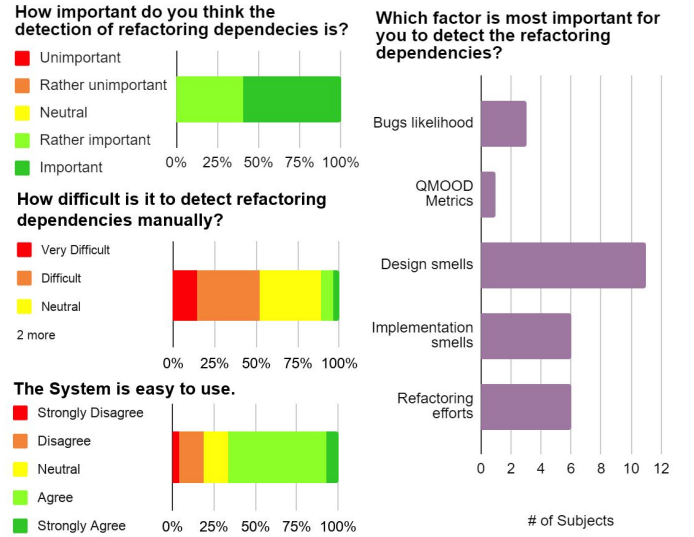


Fig. 7. Participant Survey.

As shown in Figure 7, all the participants rated this feature as important or rather important which confirms the need for tools to detect dependencies in a refactoring sequence. The majority of participants (14 out of 27) also rated the task of manually identifying these dependencies as difficult/very difficult based on their experiences in using existing refactoring tools. Figure 7 also describes that developers think that the most important motivations of automatically detecting dependencies is to understand how they can fix code smells (typically fixed using a sequence of refactorings) and also to reduce the refactoring effort by applying refactorings in an incremental way. The figure also shows that DPRef was easy to use by at least 18 out of the 27 participants and they were able to understand the dependencies without complications.

Key findings: To answer **RQ1**, the proposed algorithm for detecting refactoring dependencies achieved 100% correctness on all projects.

4.3.2 Results for RQ2

Figure 8 shows that while truly independent refactorings are more common, the mean NTR shows that more than 40% of all recommended refactorings are part of non-trivial graphs and for some projects, all refactorings are part of a single non-trivial graph (NTR=1.0). The portion of refactorings that are part of refactoring dependencies is significant.

Across all projects, non-trivial graphs have a mean number of dependencies of almost 40 (recall that this is based on solutions with a mean of roughly 150 refactorings), with nearly 100 dependencies observed for some projects. This indicates a range of connectedness in non-trivial graphs, with the more highly dependent/coupled likely being difficult for developers to understand and having a high risk of generating large numbers of invalid refactorings if they are not applied together. We noticed that the most connected refactoring graphs are more likely to be associated with small projects in which many refactorings are applied to common code locations.

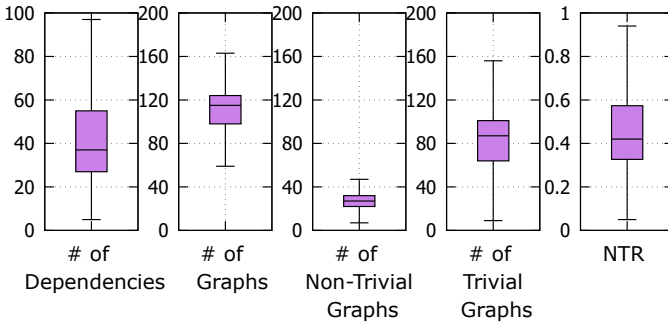


Fig. 8. Distribution of refactorings in trivial versus non-trivial graphs based on the 9,595 projects.

Comparing the range and distribution of the number of graphs, most projects have a mean of more than 100 graphs per solution, in which the number of trivial graphs is greater than non-trivial. However, as shown by NTR, the number of refactorings in non-trivial graphs is very similar or exceeds the number of refactorings in trivial graphs.

Figure 9 shows the distribution of the size of the non-trivial graphs; notice that there is a small number of non-trivial graphs including 31+ refactorings. However, the vast majority of non-trivial graphs include 2-5 refactorings. This may confirm that collections of dependent refactorings tend to be small, which can offer flexibility to developers if they want to modify these collections. Projects vary, however, with several including non-trivial graphs with 10-50 refactorings and with one including a graph of 138 refactorings.

Regarding refactoring types, Figure 10 shows that the most common refactoring types found in non-trivial graphs are *Extract Class*, *Extract Super Class* and *Extract Sub Class*. This finding confirms that modifying the hierarchy of code requires a combination of several refactorings and cannot be done with one isolated refactoring. Furthermore, the figure

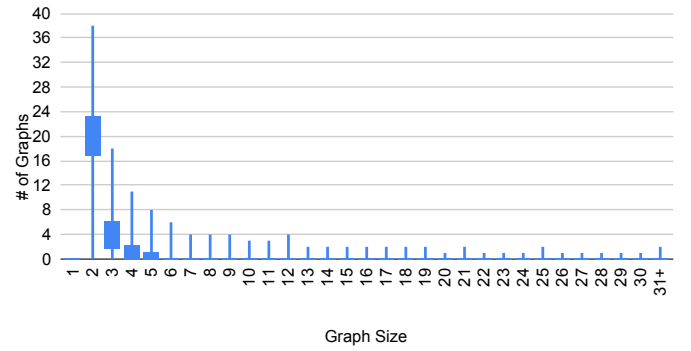


Fig. 9. Size of non-trivial refactoring graphs in the 9,595 projects.

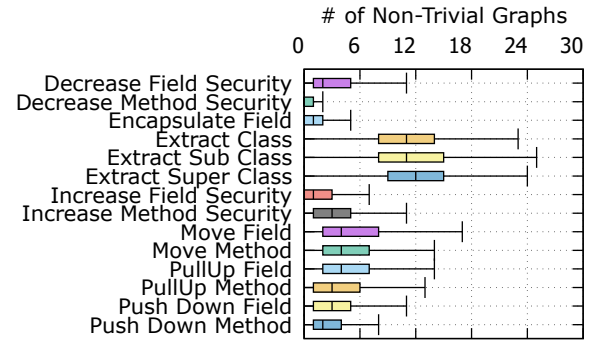


Fig. 10. Distribution of the refactoring types among non-trivial graphs for the 9,595 projects.

also shows that *Decrease Method Security*, *Encapsulate Field*, and *Increase Field Security* are the least common refactoring types in non-trivial graphs. It most likely indicates that these refactorings can be applied independently without requiring major restructuring effort.

Key findings: To answer **RQ2**, while more refactorings appear in trivial graphs than non-trivial graphs, the difference is not large. The mean value of NTR is 43%, indicating nearly half of all refactorings participate in refactoring dependencies and as such cannot be applied without consideration of other refactorings. Some refactoring types are more likely to be applied with other dependent refactorings than others.

4.3.3 Results for RQ3

Figure 11 shows how trivial and non-trivial refactoring graphs improve quality attribute and design metrics. The number of trivial graphs that improves each metric is greater than the number of non-trivial graphs that do so. However, the impact of each kind of graph on the improvement of each quality metric can vary considerably. We also analysed the improvements caused by the two kinds of refactoring graphs for a subset of the quality attribute metrics.

Figure 12 shows the rate of improvement in % for Effectiveness, Extendibility, Flexibility, Functionality, Reusability, and Understandability. This data shows that, for most projects, the improvement caused by the non-trivial graphs

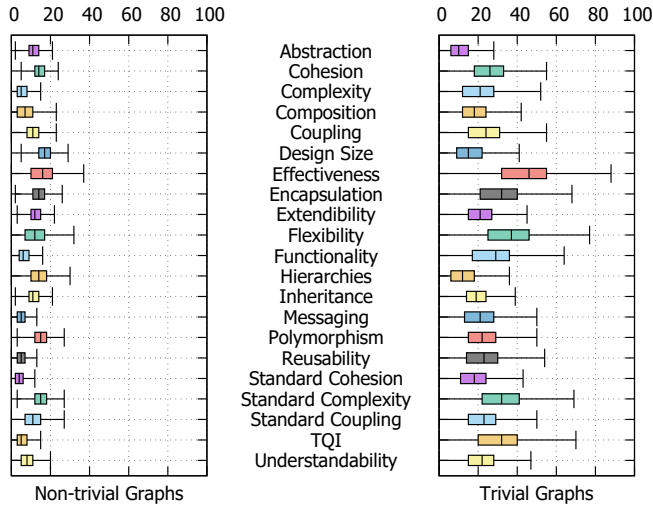


Fig. 11. The number of graphs that improved the quality metrics.

is greater than for trivial graphs. The implication is that non-trivial graphs may be more useful in practice for developers than marginal improvements obtained by individual refactorings. Some metrics, like Extendability (note the different scale) and Reusability, are more significantly improved using dependent refactorings. Even small changes are significant when considered as aggregate measures across an entire code base. This result is consistent with the fact that Extract [Super/Sub] Class refactorings are more likely to occur within a non-trivial graph and that these refactorings are natural choices for improving the Extendability and Reusability of software.

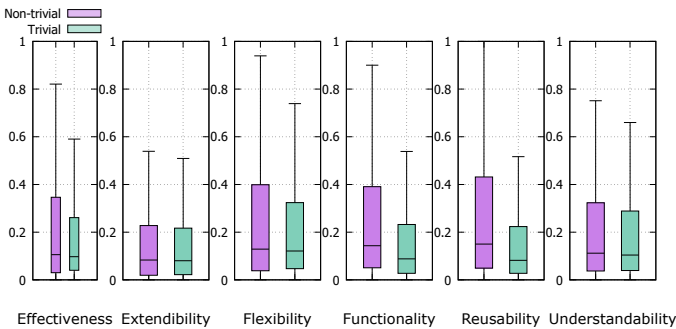


Fig. 12. Rate of quality improvement (%) for the refactoring graphs per metric.

Key findings: To answer RQ3, non-trivial refactoring graphs improve all six quality attribute metrics in our experiments better than independent refactorings. In particular, the improvement from the application of non-trivial graphs over trivial graphs is particularly significant for Functionality and Reusability.

5 THREATS TO VALIDITY

Conclusion validity. We used Design of Experiments (DoE) [31] to mitigate the internal threat related to parameter tuning used in our experiments. DoE is a methodology

for systematically applying statistics to experimentation and is one of the most efficient techniques for parameter settings of evolutionary algorithms for the used refactoring recommendation tool. Each parameter has been uniformly discretized in some intervals. Values from each interval have been tested for our application and we chose the best values.

Internal validity. We collected data from a very large number of repositories resulting in about 1.5 million refactorings for 9,595 projects. A possible internal threat is the diversity of these projects in terms of domains, size, etc. To mitigate this threat, we used several criteria (e.g., more than one contributor per project, over 5K lines of code, etc.) to select the projects and eliminate redundant ones or those with small size to avoid considering student projects on GitHub and so on. Also, different tools, with different recommendation strategies might be more or less dominated by non-trivial graphs. Historical refactorings from commit histories could likewise have a different composition. When we conclude that dependencies are common, the source of the source data matters. We make a replication package available including all the collected data that can be used and improved by the community.

Construct validity. The refactorings used in our experiments are generated using an existing refactoring tool [12]. Thus, it is possible that some of them are not relevant (e.g., small impact on quality). However, our goal is to evaluate the dependencies among the refactorings independently from their relevance. In addition, our approach can take as input any sequence of refactorings.

External validity. The types of refactorings considered in our experiments may threaten the generalizability of our results. Besides, our study was limited to the use of specific quality attributes to measure the impact of the application of refactoring graphs. Future replications of this study are necessary to further confirm our findings. Also, the number of participants can be extended in our future work to validate more refactoring dependencies.

6 IMPLICATIONS AND FUTURE WORK

Our proposed theory of organizing collections of refactoring instances as a set of refactoring graphs offers several advantages that address the challenges confirmed by developers:

- *explainability*: each refactoring graph is smaller and more coherent than a long sequence of refactorings. As each can be explained independently, the cognitive burden on a developer is much lower (e.g., contrast with determining which refactorings scattered across a sequence of dozens or hundreds of refactorings are related).
- *comparability*: search-based refactoring recommendation tools typically generate multiple recommendations on a Pareto front, leaving developers to choose one. Identifying common elements of different recommendations is simplified by comparing sets of graphs that do not contain the spurious orderings found in sequence representations.
- *search efficiency*: search-based refactoring recommendation tools that use genetic algorithms gain new options. Specifically, crossover operations can be more reliable (reducing failures) when using dependency analysis;

graphs may also be better genomes for crossover than individual refactorings.

Consequently, there are several directions for future work:

Refactoring pattern extraction. One important implication of the proposed refactoring dependency theory is the ability to extract common refactoring patterns by mining software repositories using tools such as RefMiner [16]. These patterns are the common non-trivial graphs that can be extracted on different commits/pull-requests of the same project or multiple projects. Such patterns of non-trivial graphs can be linked to refactoring opportunities such as resolving different types of code smells repeatedly.

Refactoring collaborations between developers. Studying the collaborations among multiple developers when refactoring code is a promising next step. Refactoring graphs extracted from commit histories can be linked to the authors of those commits. Then, a graph of collaborations among developers can be generated based on the dependencies among the applied refactorings. This can lead to new insights into why and when developers collaborate for refactoring.

Change operator in search-based refactoring. Random selection and application of crossover and mutation when evolving a population of solutions is a challenge in search-based refactoring. Refactoring dependency analysis can be used to avoid destroying good patterns in refactoring solutions and make change operators more intelligent, which can lead to better solutions and faster convergence.

Interactive refactoring tool support. Developers can more easily understand the implications of selecting which refactorings from a recommendation to apply, improving the interactive process and increasing their confidence in the recommendation tool. The only restriction in applying non-trivial refactoring graphs is that a refactoring can only be applied if every other refactoring that it depends on (transitively) is also applied. Thus, invalid refactorings can be detected and highlighted on the fly.

7 CONCLUSION

Although manually applying a collection of refactorings is common practice, existing empirical studies and refactoring recommendation/detection tools treat refactorings in isolation. In this paper, we proposed a definition for ordering dependencies among refactorings and an algorithm for detecting these dependencies. We also proposed defining refactoring recommendations as sets of refactoring graphs rather than as refactoring sequences and illustrated these concepts with a tool for visualizing refactoring dependencies and sets of refactoring graphs. We elaborated our research agenda for future work in Section 6.

We validated the proposed approach on 1,457,873 refactorings recommended for 9,595 projects. Our results show that the proposed approach achieved 100% in correctly detecting all dependencies among refactorings. Furthermore, we found that 43% of the 1,457,873 recommended refactorings are part of dependent refactoring graphs, which confirms that refactorings are commonly involved in dependent relations and cannot be applied truly independently. These concepts advance a theory for reasoning about refactorings

collectively, rather than individually, and offer clear benefits to developers applying refactoring recommendations (explainability and comparability) and authors of tools for recommending refactorings (search efficiency and improving correctness of recommendations).

ACKNOWLEDGMENT

Copyright 2021 IEEE. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. DM21-0546

REFERENCES

- [1] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1498–1516, 2013.
- [2] M. Kuuttila, M. Mäntylä, U. Farooq, and M. Claes, "Time pressure in software engineering: A systematic review," *Information and Software Technology*, vol. 121, p. 106257, 2020.
- [3] S. A. Slaughter, D. E. Harter, and M. S. Krishnan, "Evaluating the cost of software quality," *Communications of the ACM*, vol. 41, no. 8, pp. 67–73, 1998.
- [4] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
- [5] G. Bavota, A. D. Lucia, M. D. Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.
- [6] A. C. Bibiano, E. Fernandes, D. Oliveira, A. Garcia, M. Kalinowski, B. Fonseca, R. Oliveira, A. Oliveira, and D. Cedrim, "A quantitative study on characteristics and effect of batch refactoring on code smells," in *Proceedings of 13th the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '19)*. Porto de Galinhas, Brazil: IEEE, 2019, pp. 1–11.
- [7] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.
- [8] M. Ó. Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. H. Moghadam, "Experimental assessment of software metrics using automated refactoring," in *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '12)*. Lund, Sweden: ACM, 2012, pp. 49–58.
- [9] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.
- [10] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at Microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.
- [11] V. Alizadeh, M. Kessentini, W. Mkaouer, M. O. Cinnéide, A. Ouni, and Y. Cai, "An interactive and dynamic search-based approach to software refactoring recommendations," *IEEE Transactions on Software Engineering*, vol. 46, pp. 932–961, 2018.
- [12] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: an industrial case study," *ACM Transactions on Software Engineering and Methodology*, vol. 25, no. 3, p. 23, 2016.
- [13] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.
- [14] C. Abid, V. Alizadeh, M. Kessentini, T. d. N. F. Ferreira, and D. Dig, "30 years of software refactoring research: A systematic literature review," *arXiv preprint arXiv:2007.02194*, vol. NA, 2020.

- [15] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. O. Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. Vasteras, Sweden: ACM, 2014, pp. 331–336.
- [16] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Gothenburg, Sweden: ACM, 2018, pp. 483–494.
- [17] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation (GECCO '07)*. London, England: ACM, 2007, pp. 1106–1113.
- [18] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, and W. Zhao, "Interactive and guided architectural refactoring with search-based recommendation," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. Seattle, US: ACM, 2016, pp. 535–546.
- [19] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158 – 173, 2018.
- [20] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Recommending refactoring operations in large software systems," in *Recommendation Systems in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 387–419.
- [21] M. O'Keefe and M. O. Cinnéide, "A stochastic approach to automated design improvement," in *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java (PPPJ '03)*. Kilkenny City, Ireland: ACM, 2003, pp. 59–62.
- [22] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation (GECCO '06)*. Seattle, USA: ACM, 2006, pp. 1909–1916.
- [23] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in *Proceedings of the 19th International Conference on Program Comprehension (ICPC '11)*. Kingston, Canada: IEEE, 2011, pp. 81–90.
- [24] A. Ouni, M. Kessentini, and H. Sahraoui, "Search-based refactoring using recorded code changes," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR '13)*. Genova, Italy: IEEE, 2013, pp. 221–230.
- [25] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheue, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using nsga-iii," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 3, pp. 17:1–17:45, 2015.
- [26] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia, "How does refactoring affect internal quality attributes? a multi-project study," in *Proceedings of the 31st Brazilian Symposium on Software Engineering (SBES '17)*. Fortaleza, Brazil: ACM, 2017, pp. 74–83.
- [27] T. Mens, G. Taentzer, and O. Runge, "Detecting structural refactoring conflicts using critical pair analysis," *Electronic Notes in Theoretical Computer Science*, vol. 127, no. 3, pp. 113–128, 2005.
- [28] H. Melton and E. Tempero, "Identifying refactoring opportunities by identifying dependency cycles," in *Proceedings of the 29th Australasian Computer Science Conference (ACSC '06)*. Australia: ACM, 2006, pp. 35–41.
- [29] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "On refactoring support based on code clone dependency relation," in *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS '05)*. Como, Italy: IEEE, 2005, pp. 10–pp.
- [30] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.
- [31] J. R. Koehler and A. B. Owen, "Computer experiments. handbook of statistics," *Elsevier Science*, pp. 261–308, 1996.



Thiago do Nascimento Ferreira is a Post-doctoral Researcher at University of Michigan-Dearborn under the supervision of Dr. Marouane Kessentini in the ISELab. He received his PhD Degree in Computer Science from the Federal University of Parana in 2019. His research mainly focuses on the use of Preference and Search Based Software Engineering to address several software engineering problems such as Software Testing and Software Refactoring. Contact him at thiagod@umich.edu.



James Ivers is the lead of the Carnegie Mellon University Software Engineering Institute's software architecture group, which develops and matures tools and practices to support software architects. He is also co-author of the Documenting Software Architectures book. Contact him at jivers@sei.cmu.edu.



Jeffrey J. Yackley is a Ph.D. Candidate at the University of Michigan - Dearborn. He is co-advised by Dr. Marouane Kessentini in the ISE Lab and Dr. Bruce R. Maxim in the GAME Lab. His research focuses on search based software engineering and machine learning in order to address problems with software architecture, refactoring, and testing in addition to his research on computer science education where he focuses on applying active learning techniques in the classroom. Contact him at jyackley@umich.edu.



Marouane Kessentini is a recipient of the prestigious 2018 President of Tunisia distinguished research award, the University distinguished teaching award, the University distinguished digital education award, the College of Engineering and Computer Science distinguished research award, 4 best paper awards, and his AI-based software refactoring invention, licensed and deployed by industrial partners, is selected as one of the Top 8 inventions at the University of Michigan for 2018 (including the three campuses), among over 500 inventions, by the UM Technology Transfer Office. He is currently a tenured associate professor and leading a research group on Software Engineering Intelligence. Prior to joining UM in 2013, He received his Ph.D. from the University of Montreal in Canada in 2012. He received several grants from both industry and federal agencies and published over 110 papers in top journals and conferences. He has several collaborations with industry on the use of computational search, machine learning and evolutionary algorithms to address software engineering and services computing problems. Contact him at marouane@umich.edu.



Ipek Ozkaya is a technical director at the Carnegie Mellon University Software Engineering Institute, where she develops methods and practices for software architectures, agile development, and managing technical debt in complex systems. She coauthored a book on Managing Technical Debt: Reducing Friction in Software Development (2019). She received a PhD in Computational Design from CMU. Ozkaya is a senior member of IEEE and the 2019–2021 editor-in-chief of IEEE Software magazine. Con-

tact her at ozkaya@sei.cmu.edu.