Identifying Extract Method Refactoring Opportunities based on Functional Relevance

Sofia Charalampidou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou,
Antonios Gkortzis, Paris Avgeriou, Senior Member IEEE

Abstract—'Extract Method' is considered one of the most frequently applied and beneficial refactorings, since the corresponding Long Method smell is among the most common and persistent ones. Although Long Method is conceptually related to the implementation of diverse functionalities within a method, until now, this relationship has not been utilized while identifying refactoring opportunities. In this paper we introduce an approach (accompanied by a tool) that aims at identifying source code chunks that collaborate to provide a specific functionality, and propose their extraction as separate methods. The accuracy of the proposed approach has been empirically validated both in an industrial and an open-source setting. In the former case, the approach was capable of identifying functionally related statements within two industrial long methods (approx. 500 LoC each), with a recall rate of 93%. In the latter case, based on a comparative study on open-source data, our approach ranks better compared to two well-known techniques of the literature. To assist software engineers in the prioritization of the suggested refactoring opportunities the approach ranks them based on an estimate of their fitness for extraction. The provided ranking has been validated in both settings and proved to be strongly correlated with experts' opinion.

Index Terms— D.2.2 Design Tools and Techniques, D.2.3.a Object-oriented programming, D.2.8 Metrics/Measurement

1 Introduction

The term *code smell*[1] has been introduced by Kent Beck [10] in late 1990s to refer to parts of the source code that suffer from specific problems, usually related to a quality attribute. The term was widely popularized through the influential book of Fowler et al. [17] in 1999. According to Fowler et al. [17], code smells can be resolved through the application of refactorings, i.e., transformations that improve certain quality attributes but do not affect the external behavior of the software.

In their seminal book on refactorings, Fowler et al. [17] describe 22 possible code smells and the associated refactorings. In order to investigate the application frequency of refactorings in practice, Murphy-Hill et al. [33] performed a case study with 99 Java developers that used the Eclipse IDE refactoring tools. Based on their results the most commonly applied refactorings (among those proposed by Fowler et al.) are the *Rename Method* and the *Extract Method*. Similarly, based on the usage statistics[2] of JDeodorant (i.e., an Eclipse plugin for providing refactoring suggestions), the Extract Method refactoring stands for approximately 45% of the total refactoring actions performed by the tool.

In a similar context, but by investigating the occurrence frequency of code smells in real projects, Chatzigeorgiou and Manakos [15] conducted a case study using past versions of two open source software (OSS) systems. Specifically, they investigated the presence and evolution of four types of code smells, i.e., Long Method, Feature Envy, State Checking, and God Class. Their results indicated that *Long Method* was considerably more common than the other smells. In addition, according to Gregg et al. [22] in real-world applications 35%-55% of the methods consist of more than 90 statements. Considering that methods larger than 30 lines of code [27] are more error prone, one can understand the need for refactoring such large methods (longer than 90 statements). Given the high frequency of both the Long Method smell and its refactoring (the Extract Method[3]), this paper focuses on the suggestion of Extract Method opportunities that are able to resolve the Long method smell.

The *Long Method* smell concerns methods of large size that serve multiple purposes or functionalities. To *Extract Methods* out of longer ones we propose the use of the Single Responsibility Principle (SRP) [29]. SRP is an object-oriented principle that has been introduced at the class or package level and we tailor it so as to apply at the method level. SRP states that every module (package or class) should have exactly one responsibility, i.e., be related to only one functional requirement, and therefore

Sofia Charalampidou is with the Institute of Mathematics and Computer Science, University of Groningen, Netherlands, s.charalampidou@rug.nl

Apostolos Ampatzoglou is with the Institute of Mathematics and Computer Science, University of Groningen, Netherlands, a.ampatzoglou@rug.nl

Alexander Chatzigeorgiou is with the Department of Applied Informatics, University of Macedonia, achat@uom.gr

Antonios Gkortzis is with the Institute of Mathematics and Computer Science, University of Groningen, Netherlands, antonis.gkortzis@gmail.com

Paris Avgeriou is with the Institute of Mathematics and Computer Science, University of Groningen, Netherlands, paris@cs. rug.nl

[1] B*ad smells*, despite its original definition at the implementation level, is mostly used for higher levels of abstraction, like design [29] and architecture [21]. In this paper, we focus on code smells.

[2] https://users.encs.concordia.ca/~nikolaos/stats.html

[3] According to Fowler et al. Extract Method is the most appropriate solution for eliminating Long Method smells. Extract Method suggests to group functionally related statements into a method, whose name explains it purpose [20].

have only one reason to change. The term single responsibility has been inspired by the *functional module decomposition*, as introduced by De Marco [17] in 1979. In order to assess if a class conforms to the SRP, one needs to assess its cohesion [23, 29], which is related to the number of diverse functionalities that a class is responsible for [17]. Despite the fact that Long Methods tend to violate the SRP in their implementations (by serving more than one unrelated functionalities), to the best of our knowledge there are no approaches in the literature that aim at identifying Extract Method opportunities by checking their conformance to the Single Responsibility Principle. Although the application of the SRP is not the only way for extracting methods out of longer ones, we argue that it can identify large and functionally meaningful parts of a method, in contrast to existing approaches. As the research state-of-the-art stands, current approaches extract rather small methods, mostly involving one variable, and are not retrieved based on functionality, but based on other techniques (e.g., abstract syntax tree parsing, slicing, etc.). A detailed comparison to related work can be found in Section 2.4.

In this study we propose an approach called SRP-based Extract Method Identification (SEMI). In particular, the approach recognizes fragments of code that collaborate for providing functionality by calculating the cohesion between pairs of statements. The extraction of such code fragments can reduce the size of the initial method, and subsequently increase the cohesion of the resulting methods (i.e., after extraction); therefore, it can produce more SRP-compliant methods, since the number of diverse functionalities is decreased. To validate the ability of the proposed approach to extract parts of a Long Method that concern a specific functionality, we conducted:

- ***an industrial case study*** in a large company producing printers in Netherlands. Specifically, we applied the proposed approach to two Long Methods (approximately 1,000 lines in total) and validated the appropriateness of the proposed refactoring opportunities with three software engineers. The study's outcome suggests that the proposed approach is able to perform method extraction based on functionality with a high recall rate.
- ***a comparative case study*** on open source software. In particular, we applied SEMI on five benchmark software systems (obtained from the literature) and compared the accuracy (in terms of precision and recall) of our approach to two state-of-the-art tools (namely JDeodorant [38] and JExtract [37]). The outcome of this study suggested that our approach achieves the best combination of recall and precision (i.e., F-measure) among the examined tools. Additionaly, it scales better in terms of accuracy compared to other approaches/tools (i.e., its accuracy is almost uniform for medium- and large-sized methods).

The organization of the rest of the paper is as follows: In Section 2 we present related work, whereas in Section 3 we present in detail the rationale of the proposed approach. In Section 4, we discuss the industrial case study design and present its results, and in Section 5 we present the design and the results of our comparative case study. Next, in Section 6 we discuss the main findings, and in Section 7 the threats to validity. Finally, in Section 8 we conclude the paper.

2 Related Work

In the literature there are two different types of studies dealing with refactoring opportunities. The first type of studies concerns the introduction of new approaches aiming to identify refactoring opportunities for a single bad smell, while the second type, uses existing approaches (usually identifying different types of refactoring opportunities) aiming at investigating the issues of ranking or prioritizing the identified opportunities (e.g., [39], [31], [35]).

In this section we will focus only on the first type of studies, and specifically on studies that propose approaches for identifying Extract Method opportunities (see Section 2.1) or Extract Class opportunities (see Section 2.2). Both are considered related to our study, in the sense that they both focus on extracting parts of the code on a new artifact at a different level of granularity (i.e., method and class). Additionally, we will present studies that are indirectly related work, in the sense that they aim at feature or functionality identification (see Section 2.3). These studies are considered related to ours, as the proposed approach aims to identify code fragments that provide a specific functionality. Finally, in Section 2.4, we will compare related work to our study.

2.1 Extract Method Identification

Tsantalis and Chatzigeorgiou [38], suggest an approach that uses complete computational slices (i.e., the code fragments that are cooperating in order to compute the value of a variable) for identifying Extract Method opportunities. The evaluation of the approach consists of qualitative and quantitative assessment for an open-source project. Specifically, the authors have investigated: (a) the soundness and usefulness of the extracted slices, (b) their impact on slice-based cohesion metrics, and (c) their impact on the external behavior of the program. Additionally, as part of the evaluation process precision and recall metrics have been calculated, against the findings of independent evaluators on two research projects. The precision and recall has been calculated for 28 methods and ranged from 50-52% and from 63-75% respectively.

Yang et al. [40] suggest that the code of the Long Method should be decomposed either based on control structures (i.e. for-statements, if-statements, etc.) or code styling (i.e., blank lines in the code). The approach suggests that the composition of Extract Method opportunities should basically consider the size of the created

method, by setting appropriate thresholds. Later the calculation of coupling metrics is used in order to rank the Extract Method opportunities. The evaluation of the study aims at investigating three aspects: (a) the accuracy of the proposed approach, (b) its impact on refactoring cost, and (c) its impact on software quality. To achieve these evaluation goals, the authors conducted a case study using an open source software system of about 20,000 lines of code, spread into 269 classes. The results of the case study showed that the proposed approach achieves an accuracy of 92.82% (i.e. recommended fragments that were accepted without any adjustments) and achieves up to 40% cost reduction, in the sense of less working hours due to the automation of the process. The impact on software quality is calculated through 10 metrics and the results show improvement after the Extract Method refactoring is applied. We note that the accuracy, as calculated by Yang et al. is not comparable to precision and recall, since the independent evaluator assesses the results obtained by the provided tool and has not built a golden standard to carry out the assessment before obtaining the results of the method.

Meananeatra et al. [30] propose the decomposition of source code using the abstract syntax tree (i.e., data flow and control flow graphs) and the proposition of Extract Method opportunities based on the calculation of complexity and/or cohesion metrics. Specifically, Meananeatra et al., proposed an approach aiming at resolving the Long Method smell by applying several refactorings (not only the Extract Method one). Their approach consists of four steps. Initially they calculate a set of metrics with regard to the maintainability of the software. In the second step they calculate another set of metrics to find candidate refactorings. Candidate refactorings are also found using a set of predefined filtering conditions. During the third step they apply the refactorings and recompute the maintainability metrics, in order to compare them with the initial measurements. In the final step, the refactoring that can achieve the better maintainability improvement is proposed. The effectiveness of this approach has been evaluated through a toy example provided by Fowler's book on refactorings [17]. Through this illustration no recall and precision measures could be obtained.

Finally, Silva at al. [37] proposes the use of the abstract syntax tree and the creation of all possible combinations of lines within the blocks as candidates for extraction. These candidates are subsequently filtered based on syntactical and behavioral preconditions, and finally ranked by using their structural dependencies to the rest of the method. The precision and recall of the algorithm is evaluated through two case studies: (a) one with a system that has been developed from the authors for this reason (where Long Methods have been deliberately created), and (b) on two OSS projects (JUnit and JHotDraw). Concerning precision and recall, in the author-developed system the approach achieved a precision of 50% and a recall of 85%, whereas for the two OSS projects the precision varied from under 20% to 48%, and recall from 38% to 48%.

2.2 Extract Class Identification

Bavota et al. [8] created an extract class refactoring approach based on graph theory that exploits structural and semantic relationships between methods. Specifically, the proposed method uses a weighted graph to represent a class to be refactored, where each node represents a method of the class. The weight of an edge that connects two nodes (methods) is a measure of the structural and semantic relationship between two methods that contribute to class cohesion. A MaxFlow-MinCut algorithm is used to split the built graph in two sub-graphs, cutting a minimum number of edges with a low weight. These two sub-graphs can be used to build two new classes having higher cohesion than the original class. The attributes of the original class are also distributed among the extracted classes according to how they are used by the methods in the new classes. The method was empirically evaluated through two case studies. The first case study was performed on three open source projects (ArgoUML, Eclipse, and JHotDraw) and aimed at analyzing the impact of the configuration parameters on the performance of the proposed approach as well as verifying whether or not the combination of structural and semantic measures is valuable for the identification of refactoring opportunities. The second case study was based on a real usage scenario and focused on the user's opinion while refactoring classes with low cohesion. The results of the empirical evaluation highlighted the benefits provided by the combination of semantic and structural measures and the potential usefulness of the proposed method as a feature for software development environments. The approach has been evaluated using F-measure, which has been calculated as approximately 0.75 for all examined applications.

Fokaefs et al. [19] implemented an Eclipse plugin that identifies extract class refactoring opportunities, ranks them based on the improvement each one is expected to bring to the system design, and applies the refactoring chosen by the developer, in a fully automated way. The first step of the approach relies on an agglomerative clustering algorithm, which identifies cohesive sets of class members within the system classes, while the second step relies on the Entity Placement metric as a measure of design quality. The approach was evaluated on various systems in terms of precision and recall, while it was also assessed by an expert and through the use of metrics. The evaluation showed that the method can produce meaningful and conceptually correct suggestions and extract classes that developers would recognize as meaningful concepts that improve the design quality of the underlying system. The accuracy of the proposed approach has been evaluated on six open source classes, leading to a precision of 77% and a recall rate of 87%.

Bavota et al. [9] proposed an approach recommending extract class refactoring opportunities, based on

game theory. Given a class to be refactored, the approach models a non-cooperative game with the aim of improving the cohesion of the original class. A preliminary evaluation, which was inspired by mutation testing (i.e. merging two classes and then trying to recreate the original classes using an extract class approach), was performed using two open source projects (ArgoUML and JHotDraw). The evaluation aimed at comparing: (a) the results derived using the Nash equilibrium and the Pareto optimum, as well as (b) the results of the proposed approach to state-of-the-art. The comparison has been performed based on F-measure [18], the applicability and the benefits of the proposed approach were demonstrated. The mean F-measure for the two projects was ranging from 84%-89%, exceling compared to the other two approaches.

2.3 Feature/ Functionality Identification

In this subsection, we present research efforts that attempt to identify parts of the source code that are providing a specific functionality through static analysis. Although in the literature there are several studies using information retrieval techniques aiming to connect features to computational units in the source code (e.g., [43], [44]), such a mapping has the opposite direction compared to our approach, and therefore, are omitted from this section. In addition to that, the majority of these studies use dynamic analysis in contrast to our approach which employs static analysis.

The approach proposed by Yoshida et al. [41] consists of three steps. The first involves syntax analysis of the source code into fragments, creating a syntax tree where the program syntax consist the nodes, and the code fragments the leaves. The second involves the extraction of functional elements, i.e., code fragments that work in cooperation. The extent to which code fragments cooperate is calculated using the Normalized Cohesion of Code fragments ($NCOCP_2$) metric and the results are compared to a threshold set in the same study. Finally, as last step, the approach proposes the combination of functional elements that show high cohesion. To verify the outcomes proposed by the approach, the authors conducted a case study using one software system of 3,641 lines of code, 70 classes, and 600 methods. The developer of the software was responsible for confirming the outcomes of the approach, which achieved to identify 51 out of the 80 functionalities (i.e., recall 63.7%), however, the precision of the approach is not provided by the authors.

Additionally, Antioniol et al. [4] compared the use of two different information retrieval approaches, one using a probabilistic and the other a vector space approach, aiming at associating high-level concepts with program concepts. To evaluate the two approaches they performed two case studies, one of which aimed at tracing source code to functional requirements using a Java system, consisting of 95 classes and about 20,000 lines of code. The validation of the study was performed based on experts who identified 58 correct functionalities among the 420 that had been suggested by the approach. The results of the study showed that both approaches can score about 13% - 48% precision for achieving a recall rate between 100% - 50%.

2.4 Comparison to Related Work

In this section, we compare SEMI with the approaches discussed in Section 2.1, from two perspectives: (a) in terms of the rationale of the approach, and (b) in terms of empirical validation.

Approach Rationale. First we discuss possible limitations of the approaches presented in Section 2.1, for extracting functionally coherent code blocks. We note that these limitations do not imply that the specific approaches are not adequate for suggesting relevant Extract Method opportunities, but we only discuss them against their fitness for creating SRP-compliant methods. To make this section more readable, we group the state-of-the-art approaches based on the rationale of their extraction algorithm, as follows:

- ***Approaches based on complete computation slicing*** (i.e., identification of code fragments that are cooperating in order to compute the value of a variable) [38]—The complete computation slice of a variable considers only cases when the variable changes value, without considering the lines where the variable is used, although such lines might participate in a code fragment serving a "larger" functionality. To our understanding, "large" functionalities are not easy to be offered by calculating only one variable, but sets of them. In that sense, the use of complete computation slicing is expected to only identify rather "small" functionalities, whereas the proposed approach can incorporate multiple calculations in the extracted fragments of code. We note that there are some slicing approaches taking also into account the use of variables (see e.g. [32]). However, none of these approaches has been exploited for the purpose of identifying extract method opportunities.
- ***Approaches based on code styling*** (e.g., blank lines in the code) [40]—Depending on code styling assumptions, like the separation of code fragments that concern unique functionalities using an empty line, is considered as a threat to the validity of the approach proposed by Yang et al. In particular, such approaches cannot be accurate for cases in which the assumption does not hold, e.g., a developer makes excessive or limited use of blank lines.
- ***Approaches based only on the abstract syntax tree*** (i.e., the iteration and decision nodes of the code) [41] and [30]—Approaches that are only based on the abstract syntax tree, might miss Extract Method opportunities, since some potentially large code fragments are considered as blocks and are not further examined. For example, consider the case that a method consists of multiple

statements, offering two different functionalities by every branch of an if-statement. In such cases, since these nodes are not further decomposed, potential Extract Method opportunities, which capture functionalities, may not be identified.

- ***Approaches based on the abstract syntax tree & all possible combinations of lines within the blocks*** [37]—An exhaustive set of all possible combinations of continuous lines within the syntax blocks may cause an enormous number of Extract Method opportunities, which have not been selected based on any quality characteristic. Although most of the functionalities will be identified, this exhaustive tactic is not considered as optimal.

Empirical Validation. In terms of empirical validation, we compare our study to existing state-of-the-art based on the following criteria: (a) research setting (e.g., industrial, open source, etc.), and (b) size of examined methods. The results of this comparison are presented in Table I.

TABLE I. Comparison to Related Work

Study	Research Setting	Average Examined Method Size
[38]	OSS	33.68
[40]	OSS	41.32
[30]	Illustration	46.00
[37]	Illustration	8.75
	OSS	48.40
Our Study	Industrial & OSS	525.00

Contributions. Therefore, our work advances the state-of-the-art, as follows:

- it is the first study that investigates the ***functional relevance*** of source code fragments to *identify Extract Method opportunities*. Extracting methods based on the offered functionality is considered a benefit, since it is conceptually closer to system design and modularization principles.
- it is the first study that is *empirically validated* with ***methods of hundreds of lines of code***. Validating an approach in a different order of magnitude is important for two reasons: (a) it tests the scalability of the approach, (b) it offers a more realistic validation environment than toy examples, as for methods of maximum 50 lines of code the assistance that a software engineer needs is minimum.
- it is the first study that is *empirically evaluated* in an ***industrial setting*** by *professional software engineers*. This aspect is important since industrial experts are more experienced, aware of the problems that specific methods have, and contribute to increasing the realism of the empirical setting.

3 THE SEMI APPROACH

In this section we discuss the proposed approach for identifying Extract Method opportunities, based on the single responsibility principle. The approach can be decomposed into two major parts that for simplicity are discussed in separate subsections: (a) the identification of candidate Extract Method opportunities, based on the functional relevance of code statements (see Section 3.1), and (b) the grouping and ranking of these candidates (see Section 3.2). Step (b) of the approach is important since the list of Extract Method opportunities can be large and may contain multiple overlapping suggestions.

3.1 Identification of candidate Extract Method opportunities

In the first part of the SEMI approach we are interested in identifying successive statements that are cooperating in order to provide a specific functionality to the system. According to De Marco et al. [17], cohesion is characterized as a proxy of the number of distinct functionalities that a module is responsible for. In this paper, we are interested in cohesion at method level and specifically in the coherence of statements. Therefore, as coherent we characterize two statements if they [14][4]:

- ***are accessing the same variable***. This choice is based on the definition of all method-attribute cohesion metrics [1], in which cohesion is calculated based on whether two methods are accessing a common class attribute. We note that in the context of this study, as variables we consider attributes, local variables and method parameters (i.e., every variable that is accessible through all statements in one method's body); or
- ***are calling a method for the same object***. This choice is based on the previous one, by taking into account the fact that objects are a special case of variables. This type of cohesion is named communication/information cohesion [42], according to which modules that are grouped together because they work on the same data, are coherent; or
- ***are calling the same method for a different object of the same type***. This choice is based on the definition of several cohesion metrics (e.g., LCOM4 (Lack of Cohesion Of Methods) [24], TCC (Tight Class Cohesion) and LCC (Loose Class Cohesion) [11], DCD (Degree of Cohesion-Direct) and DCI (Degree of Cohesion-Indirect) [5]) that consider two statements as coherent if they call the same method for a different object. The rationale of such metrics lies on the fact that although a function is performed on different data the two statements are related, since they are in need of the same service. Specifically, by calling the same method (e.g., `start()`) on two different objects (e.g., `rightAirplaneEngine` and `leftAirplaneEngine`), the same functionality is performed on different data. However, the two lines provide exactly the same functionality (in our example, starting first the right and then the left engine of the same

[4] This definition is in accordance to the cohesion among methods of a class, based on which two methods are coherent if they access the same attribute.

plane). Therefore, they should be considered functionally relevant (which is exactly the goal of our approach—i.e., identifying which lines are functionally coherent). We need to note that the two objects (`left` and `right`) are instances of the same class (e.g., `AirplaneEngine`), and therefore share the same set of possible method invocations. Finally, based on our previous work [14] that empirically explored the ability of cohesion metrics to predict the existence and the refactoring urgency of long method occurrences, $LCOM_4$ and DCD have been found to be among the most efficient indicators.

Based on this definition we identify all possible sets of successive statements that are coherent to each other (regardless of their size). To achieve this goal, we follow the process described in the flow chart of Fig. 1. We note that the final state of Fig. 1 does not correspond to the end of the approach, but only to the end of its first part (i.e., `Identification of candidate Extract Method opportunities`).

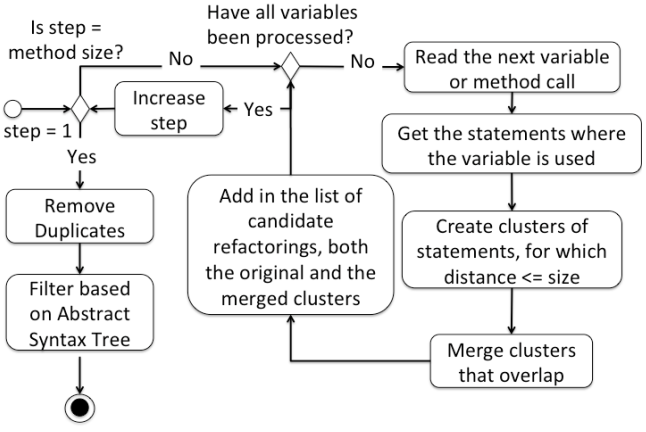

Fig. 1: Flow chart of Extract Method opportunities algorithm

A detailed explanation for each step of the aforementioned process will be presented through an illustrative example as follows: Suppose we are applying the proposed approach to the source code of a sample method, as presented in Fig. 2. In Fig. 2 all variables which are accessible by the method's statements (i.e., local variables, attributes, and parameters) and method calls have been underlined, in order to ease the calculation of the cohesion between statements. We note that we are only focusing on distinct accessible variables and method calls per statement, i.e., in cases that a variable or method call appears more than once in a single statement, we consider it only once. For example, the use of variable `i` in line `3` is underlined only once.

As an initialization step, a table that contains an index of used variables/called methods per statement is developed (see Table II). We note that, similarly to a program dependence graph for the special case of conditional statements, the `else` and the `else-if` statements include an indirect use of the variables used in the condition (e.g., the `else` statement in line `7` suggests that the value of variable `rcs` should be considered). Therefore, the variables or method calls used in conditions are copied to all branches of the statement.

```
1.public Resource[][] grabManifests(Resource[] rcs) {
2.  Resource[][] manifests = new Resource[rcs.length][] ;
3.  for(int i=0; i<rcs.length; i++) {
4.    Resource[][] rec = null;
5.    if(rcs[i] instanceof FileSet) {
6.      rec = grabRes(new FileSet[] {(FileSet)rcs[i]});
7.    } else {
8.      rec = grabNonFileSetRes(new Resource []{ rcs[i] });
9.    }
10.   for(int j=0; j< rec[0].length; j++) {
11.     String name = rec[0][j].getName().replace('\\','/');
12.     if(rcs[i] instanceof ArchiveFileSet) {
13.         ArchiveFileSet afs = (ArchiveFileSet) rcs[i];
14.         if (!"".equals(afs.getFullpath(getProj()))) {
15.            name.afs.getFullpath(getProj());
16.         } else if(!"".equals(afs.getPref(getProj()))) {
17.            String pr = afs.getPref(getProj());
18.            if(!pr.endsWith("/") &&  !pr.endsWith("\\")) {
19.               pr += "/";
20.            }
21.            name = pr + name;
22.         }
23.      }
24.      if (name.equalsIgnoreCase(MANIFEST_NAME)) {
25.        manifests[i] = new Resource[] {rec[0][j]};
26.        break;
27.      }
28.   }
29.   if (manifests[i] == null) {
30.      manifests[i] = new Resource[0];
31.   }
32. }
33. return manifests;
34.}
```

Fig. 2: Example Code

TABLE II. Variable/Method Call Index in Example

#Line	Accessed Variables / Called Method
2	manifests; rcs.length; rcs; length
3	i; rcs.length; rcs; length
4	rec
5	rcs; i
6	rec; grabRes; rcs; i
7	rcs; i
8	rec; grabNonFileSetRes; rcs; i
10	j; rec.length; rec; length
11	name; rec.getName.replace; j; rec; getName.replace; getName;replace
12	rcs; i
13	afs; rcs; i
14	rcs; i; equals; afs.getFullpath; getProj; afs; getFullpath
15	name.afs.getFullpath; getProj; name; afs.getFullpath; afs; getFullpath

#Line	Accessed Variables / Called Method
16	rcs; i; equals; afs.getFullpath; getProj; afs.getPref; afs; getFullpath; getPref
17	pr; afs.getPref; getProj; afs; getPref
18	rcs; i; equals; afs.getFullpath; getProj; afs.getPref; afs; getFullpath; getPref; pr.endsWith; pr; endsWith
19	pr
21	name; pr
24	name.equalsIgnoreCase; name; equalsIgnoreCase
25	manifests; i; rec; j
29	manifests; i
30	manifests; i
33	Manifests

Next, and in order to ease the comprehension of the next steps of the algorithm, we visualize the information of Table II in a matrix (see Fig. 3). In the matrix, as lines we add all accessible variables and called methods, as columns the corresponding source code line (for simplicity, in the example we assume that each line has only one statement), whereas in the cells we denote the use of a specific variable or method call in the corresponding statement.

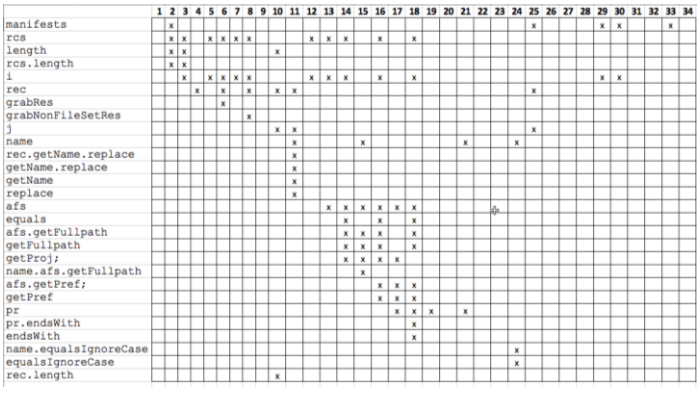

Fig. 3: Matrix visualization of accessible variables and method calls per statement

In the initialization of the iterative part of the algorithm, we begin with a step that equals one (step=1). With this step, the algorithm creates clusters of all the successive statements that access at least one common variable or call the same method, as shown in Fig. 4.

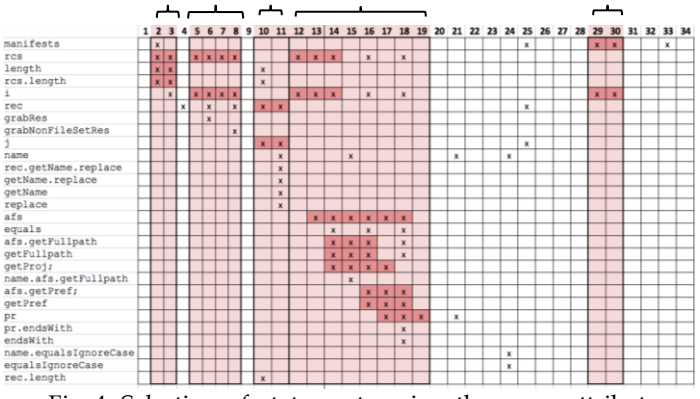

Fig. 4: Selection of statements using the same attribute or calling the same method, with step=1

The identification of Extract Method opportunities continues by increasing the step by one in each iteration. So, with step=2, new clusters are formed by treating statements with distance equal to 2 as successive. The newly clustered lines are presented in Fig. 5 with dark shading. Next, the algorithm performs a merging activity based on the agglomerative hierarchical clustering approach [23]. The criterion used for merging two clusters is the existence of an overlap between statements. In other words, the algorithm merges clusters that include even one common statement. To derive these Extract Method opportunities, the overlapping sets of statements are merged, as presented in Fig. 6. Concerning merging, as an example, we can look at that the cluster including statements 2-8 and the cluster including statements 4-11. The clusters are merged in a larger cluster of statements, since statements 4-8 are common in both clusters. We note that as candidate Extract Method opportunities we include both the original (i.e., 2-8 and 4-11) and the merged (2-11) clusters. This process can merge sets of statements that are only indirectly relevant. For example, statements 2-11 are only indirectly related, through the use of variable `rec` and method call `rcs`.

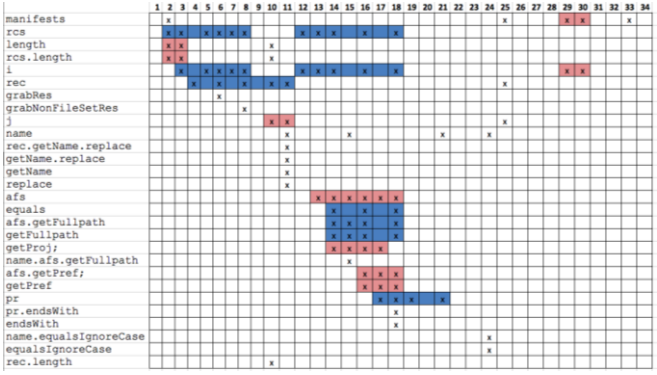

Fig. 5: Selection of statements accessing the same variable or calling the same method, with step=2

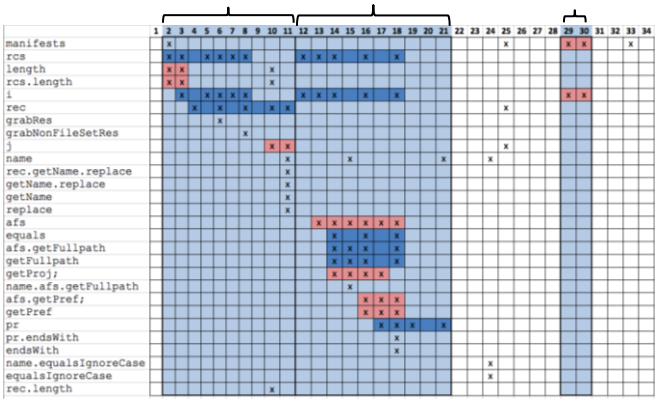

Fig. 6: Extract Method opportunities, derived with step=2

The algorithm continues to iterate until we reach the maximum step, i.e. method size. After all possible Extract Method opportunities have been identified, the algorithm removes the duplicate and the invalid clusters. This second task is very important because in many cases extracting a set of statements from a code would create

compile errors by violating syntactical or semantic preconditions, or behavioral inconsistencies [37].

The syntactical preconditions taken into consideration require that the selected fragment to be extracted should consist only by complete blocks of sequential statements. For example, if we want to extract statements A and B, but statement A is just before the block of an *if statement* and statement B inside the block of this *if statement*, then the extracted code should include all statements starting from statement A, until the closing statement of the *if block*. These preconditions guarantee that the recommendations provided by our approach can be directly applied to methods, without statement reordering. In addition, preservation of the syntax in combination with the fact that the extracted continuous statements are replaced by a method invocation, eliminate the possibility of breaking program semantics. In particular, according to the definition of Komondoor et al. [26] two methods are syntactically equivalent, if when they are called in the same state (i.e., same values for all variables) they produce the same output; this is true for our approach, since the sequence of statement execution and variable values are not altered compared to the original method. Finally, a set of behavioral preconditions should apply to ensure the preservation of functionality. For example, it should not be possible to extract a fragment in which two or more primitive variables are assigned that are also used by other statements out of this fragment. The reason behind this precondition is that due to Java restrictions, it is not possible to return the value of two variables.

The rationale of checking if a set of statements is valid for extraction has been exhaustively discussed in the literature (e.g., [37], [38]) and is for simplicity not discussed in this section. An example of such a case, is shown in Fig. 2, where the proposed set of statements suggested to be extracted (i.e., 25 - 33) is not valid, because it does not include complete blocks of code. Similarly, to Silva et al. [37], as blocks of code we refer to a sequence of continuous statements that follow a linear control flow. In particular, blocks 24-27 and 2-33 are only partially included. We note that in order to assist in the process of identifying the input and output parameters of the proposed extract method opportunity, the tool makes all required calculations, so that the values of the variables are not lost when invoking the new method.

3.2 Extract Method Opportunity Grouping/ Ranking

Once a list of all candidate Extract Method opportunities is created, the SEMI algorithm first groups them and then ranks them. The main idea for grouping Extract Method opportunities is that every two opportunities that are heavily overlapping and are of similar size[5] are highly probable to offer the same functionality. In particular, we expect that sets of statements of different size (i.e., number of statements) are not able to provide the same functionality. For example, suppose a set of 100 instructions that rotate a matrix clock-wise, perform a transformation on it, and then rotate it counter-clockwise, so as to bring it in the original position. Let us assume that a set of 30 instructions that perform the clockwise rotation overlaps with the identified set of 100 instructions. The opportunity to extract the 30 instructions cannot be considered as an alternative opportunity to the extraction of the entire set of 100 instructions, since it is not reasonable to assume that these 30 instructions can deliver the same functionality.

```
1. FOR each opportunity IN opportunity_list
2.   IF (opportunity.isAlreadyAnAlternative()) THEN
3.     SKIP to next opportunity
4.   END IF
5.   FOR each other_opp IN opportunity_list
6.     IF
7.       NotSimilarSize(opportunity, other_opp) AND
8.       SignificantlyOverlapping(opportunity, other_opp)
9.       AND other_opp.isAlreadyAnAlternative()==false)
10.    THEN
11.      IF
12.        (opportunity.HasMoreBenefitThan(other_opp))
13.      THEN
14.        opportunity.alternatives.Add(other_opp)
15.        set other_opp.isAlternative = true
16.      ELSE
17.        other_opp.alternatives.Add(opportunity)
18.        set opportunity.isAlternative = true
23.      END IF
24.    END IF
25.  END FOR
26. END FOR
```

Fig. 7: Extract Method Opportunity Grouping Algorithm

For every group of Extract Method opportunities, the optimal opportunity is set as the primary suggestion for extraction, and the rest are characterized as its alternatives. As optimal opportunity we consider the one that offers the highest benefit in terms of a specific fitness function (the selection of this fitness function is discussed in detail later in this section). The definition of the benefit that a software engineer would get from splitting a long method cannot be strictly defined, since it heavily depends on his perception. In particular, the benefit can range from purely measurable source code quality aspects (such as size, lack of cohesion, etc.) to more abstract ones (e.g., understandability, maintainability, etc.). This approach is based on measureable aspects, such as the cohesion metrics discussed in Section 3.1, which nevertheless affect the more abstract ones. The steps followed for executing this process are outlined in the pseudocode of Fig. 7. The pseudocode of Fig. 7, includes five parameters provided by the user at the execution time:

max_size_difference: The maximum allowed difference in size between two opportunities so as to be considered valid for grouping (see `NotSimilarSize`—statement 7). The difference in size is calculated as the ratio of absolute

[5] The thresholds for characterizing two extract method opportunities as heavily overlapping and being similar in size are parameters of the algorithm. These two, along with other parameters of the algorithm are discussed just after its high-level description.

difference of the two Extract Method opportunities, over the size of the smaller method:

$$Difference\_in\_Size(A, B) = \frac{|A.LoC - B.LoC|}{MIN(A.LoC, B.LoC)}$$

For example, if max_size_difference is set to 0.2, and the size of the two opportunities is 15 and 10, respectively, the `difference in size` can be calculated as (15-10) / 10 = 0.5, which is larger than the maximum allowed difference. As a default `max_size_difference` in this paper we use 0.2, i.e., a method is considered to be of similar size if it is ±20% larger or smaller. The use of a smaller default value (e.g., ±10%) would not be fitting for a rather small opportunity, since opportunities of size<10 would not be able to group with any other opportunity. The fact that the selection of these thresholds does not heavily influence the achieved accuracy of the proposed approach is discussed in Section 5.1 and the threats to validity section.

min_overlap: The minimum allowed overlap in the range of two opportunities so as to be considered valid for grouping (see `SignificantlyOverlapping`– statement 8). The overlap between two Extract Method opportunities is calculated as the percentage of overlapping statements, as follows:

$$overlap(A,B) = \begin{cases} |B.end - B.start + 1|, \\ A.start \le B.start \wedge A.end \ge B.end \\ \\ |A.end - A.start + 1|, \\ A.start \ge B.start \wedge A.end \le B.end \\ \\ |A.end - B.start|, \\ A.start \le B.start \wedge A.start \le B.end \wedge \\ A.end \ge B.start \\ \\ |A.start - B.end|, \\ A.start \le B.start \wedge A.end \ge B.start \wedge \\ A.start \le B.end \end{cases}$$

$$Overlap = \frac{overlap(A,B)}{MAX(A.LoC, B.LoC)}$$

We note that (A|B).start and (A|B).end correspond to the starting and ending statement numbers. To better facilitate the understanding of the four cases in which Extract Method opportunities A and B can overlap, we visualize all possible relations in Fig. 8. In this work as a default value for `min_overlap` we set 0.1. Therefore, even slightly overlapping opportunities can be grouped. This decision has been taken so as to reduce as much as possible the suggestions that are provided to the users.

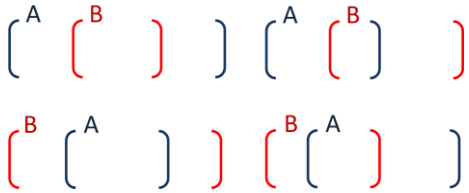

Fig. 8: Cases of Extract Method Overlap

significant_difference_threshold: The minimum difference in the benefit incurred by the two opportunities, so as to decide which one is the optimal. There are two measures of benefit outlined below (a primary and a secondary one). First, we check the difference between the primary benefit scores by calculating the normalized absolute difference:

$$Difference\_Between\_Benefits = \frac{|A.benefit - B.benefit|}{MAX(A.benefit, B.benefit)}$$

In case it is lower than the threshold for characterizing differences as significant, the secondary measure is used. In this study, we used 0.01 as the default value for the `significant difference threshold`. The value has been selected as the default strict value for checking significance in most statistical tests.

primary_measure_of_benefit: The method body cohesion metric that is used for comparing two opportunities. The term method body cohesion metric refers to measures that quantify the relevance/coherence of statements inside a single method [14]. We note that the selection of one metric as a primary measure of benefit is a choice of the software engineer, based on his personal intuition (a sample catalog is provided in [14]). However, for this study we selected to use $LCOM_2$[6] for the following reasons:

- it is a metric that although it ***assesses method cohesion***, it is ***correlated to method's size*** as well. This correlation is due to the way the metric is calculated, i.e., the upper limit of the metric score is the number of combinations by any two of the statements of method[7].
- it takes into account both ***cohesive*** and ***non-cohesive pairs*** of statements. Although both $LCOM_1$ and $LCOM_2$ conform to the aforementioned claim (i.e., they assess cohesion and are correlated to size), $LCOM_1$ is a count of only the non-cohesive pairs of statements. Such a calculation miss-assesses two methods of different sizes that have the same number of non-cohesive statements, but one has a bigger number of cohesive statements.
- it is among the ***top predictors for Long Methods identification***, based on a case study performed by Charalampidou et al. [14]. We expect that since the Long Method bad smell and the Extract Method refactoring are closely related, metrics that perform well in identifying the one, will be adequate for the other as well.

`Benefit` is calculated as the gain of applying the refactoring in terms of cohesion. Specifically, we use the worst case scenario for this calculation, by using the following

[6] We note that the numbering of LCOM metrics has been adopted from the overview by Al Dallal and Briand [1]. $LCOM_2$ has been tailored so as to assess cohesion at method level as follows: $LCOM_2 = P - Q$, if $P - Q \ge 0$ / otherwise $LCOM_2 = 0$, where P is the number of pairs of statements that do not share variables and Q is the number of pairs of lines that share variables.

[7] $LCOM_2 \in [0, \binom{LOC}{2}]$, where LOC equals the number of statements

formula. The rationale for using the MAX function is that we want to guarantee that none of the resulting methods has cohesion worse than that of the original method.

$$Benefit_{LCOM2} = LCOM2_{(original)} - MAX(LCOM2_{(opportunity)}, LCOM2_{(original\_after\_refactoring)})$$

secondary_measure_of_benefit: The secondary measure that we use is method size (in number of statements). To explain the choice of size as the secondary metric for comparing opportunities, we use the example of Fig. 9. In the left hand side of Fig. 9 we denote sets of statements that are 100% cohesive (i.e., all lines are cohesive to each other) within the same fill pattern (i.e., first and third sets of statements are cohesive). Also, we consider that statements with different fill patterns are 100% non-cohesive (i.e., no variable is shared). In this case, $LCOM_1$ for the left method[8] [14] is 38, and we compare two Extract Method opportunities: (Opp1) which extracts the block of 4 *LoC*, and (Opp2) which extracts the block of 2 *LoC*. We note that the extracted methods are totally cohesive and are not shown in the Figure. The remaining method from applying (Opp1) is a method with an $LCOM_1$ value equal to 10, whereas the remaining method of (Opp2) is a method with an $LCOM_1$ value equal to 20. Therefore, the benefit from extracting a larger number of statements (of same cohesion) is higher. Although this example describes an extreme scenario, the effect is similar in other cases.

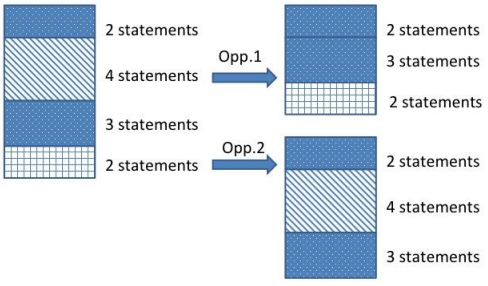

Fig. 9: Extract Method Benefit

After the grouping is completed, the created groups of opportunities are sorted based on the `primary_measure_of_benefit`. Similarly to grouping, if there is no significant difference between the primary suggestions of two groups, the larger Extract Method opportunity is prioritized. An illustrative example of the aforementioned process is presented below.

For example, consider the method of Fig. 2. By applying the *`Identification of candidate Extract Method opportunities`* part of the SEMI algorithm, we ended up with an opportunity list of 11 candidate refactorings, as presented in Table III. The first column of Table III is a simple identifier for the Extract Method candidate, the second column refers to the involved statements (line numbers), the third column is the `primary_measure_of_benefit` that is achieved by extracting the specified statements, whereas the last column represents the size (in number of statements) of the candidate Extract Method. For the sake of illustration, let's suppose that we use the default grouping parameters (described earlier in Section 3.2)

TABLE III. Initial extract method opportunities

	opportunity	Primary benefit	size
1	002 to 032	35	22
2	002 to 034	0	24
3	003 to 032	49	21
4	003 to 034	14	23
5	004 to 031	46	20
6	010 to 031	60	16
7	013 to 022	68	9
8	014 to 022	63	8
9	017 to 020	29	4
10	017 to 021	29	5
11	029 to 031	11	3

The *grouping algorithm* selects two opportunities of the aforementioned opportunity list and checks if they satisfy the grouping criteria (i.e., based on *max_size_difference* and *min_overlap*). For this first step we select opportunities 1 & 2 as shown in Table IV.

TABLE IV. Comparison of Extract Method Opportunities

	Opportunity	Primary benefit	size
1	002 to 032	35	22
2	002 to 034	0	24
3	003 to 032	49	21
4	003 to 034	14	23
5	004 to 031	46	20
6	010 to 031	60	16
7	013 to 022	68	9
8	014 to 022	63	8
9	017 to 020	29	4
10	017 to 021	29	5
11	029 to 031	11	3

By checking opportunities 1 and 2 we can observe that they satisfy the grouping criteria since *difference_in_size* = 0.09 (i.e., 2/22) < *max_size_difference* (0.2) and *overlap* = 0.91 (i.e., 22/24)> *min_overlap* (0.1). Thus opportunities 1 and 2 can create a group of opportunities. The next step is to find which opportunity will be the primary suggestion of the group. The current difference between the two opportunities, in terms of $Benefit_{LCOM2}$ is 1.00 (i.e., (35 - 0) / 35) > 0.01, which means that we need to compare the two opportunities based on their *primary_measure_of_benefit*. Therefore, and since $Benefit_{LCOM2}$ of opportunity 1 has a higher value compared to the *Bene-*

[8] Although $LCOM_1$ has been originally introduced at the class level by Chidamber and Kemerer [16], in this study we used the method-level definition as tailored by Charalampidou et al. [14]. In particular, $LCOM_1$ is calculated as $LCOM_1 = P$, where `P` is the number of pairs of statements that do not share variables.

fit_{LCOM2} of opportunity 2, opportunity 1 will be the primary suggestion of the group, and will "*include*" opportunity 2 as an alternative. For opportunities 1 and 3 the same criteria (max_size_difference and overlap) are met and thus, they can form a group. The current difference between the two clusters, in terms of $Benefit_{LCOM2}$, is 0.28 > 0.01. Therefore, in this case, and since opportunity 3 has a greater $Benefit_{LCOM2}$, it will be the primary suggestion of the group and will "*absorb*" opportunity 1 and its existing alternatives.

Next, by comparing opportunity 3 to opportunities 4 and 5, we can inspect that they both satisfy the grouping criteria and are also included as alternatives of opportunity 3. Opportunity 3 will thus be the primary suggestion with opportunities 1, 2, 4 and 5 as alternatives, as shown in Table V.

TABLE V. Grouping of Extract Method Opportunities

	opportunity	Primary benefit	size
3	003 to 032	49	21
1	002 to 032	35	22
2	002 to 034	0	24
4	003 to 034	14	23
5	004 to 031	46	20
6	010 to 031	60	16
7	013 to 022	68	9
8	014 to 022	63	8
9	017 to 020	29	4
10	017 to 021	29	5
11	029 to 031	11	3

The reason that we choose to store alternatives of Extract Method opportunities is to help software engineers in identifying slightly deviating opportunities. For example, suppose a Long Method of 600 statements, in such a method suppose a primary suggestion starting at statement 100 and finishing in statement 180. When the software engineer inspects the suggested refactoring, he/she notices that one functionality starts at statement 100, finishes near 180, but not exactly there. In that case, he/she can go through the alternatives and easily identify the most accurate source code part that offers the complete functionality.

The next comparison is between opportunities 3 and 6, which however does not satisfy one of the grouping criteria, namely *difference_in_size* = 0.22 (i.e., 11/49) > *max_size_difference* = 0.2. Therefore, these two opportunities cannot be grouped. This is also the result of the comparison of opportunity 3 with opportunities 7 to 11. The next step of the algorithm is to select the next opportunity that does not participate in any group and repeat the process. If we apply the same steps on the remaining opportunities and the sorting based on the primary benefit, the final result will be five groups of opportunities as shown in Table VI.

TABLE VI. Final Set of Extract Method Opportunities

	opportunity	Primary benefit	size
7	013 to 022	68	9
8	014 to 022	63	8
6	010 to 031	60	16
3	003 to 032	49	21
5	004 to 031	46	20
1	002 to 032	35	22
4	003 to 034	14	23
2	002 to 034	0	24
10	017 to 021	29	5
9	017 to 020	29	4
11	029 to 031	11	3

4 Industrial Case Study

In this section we present the design and the results of the industrial case study, which aimed at assessing if the extract method opportunities identified by SEMI, can be linked to specific functionalities. The case study has been performed within a large company producing printers, in the Netherlands.

4.1 Case Study Design

The goal of this case study, expressed with a GQM formulation [7], is to ***analyze*** *the SEMI approach* ***for the purpose*** *of evaluation,* ***with respect to*** *its ability to (a) accurately identify the functionalities of the original method, and (b) efficiently rank the candidate Extract Method opportunities based on their extraction benefit,* ***from the viewpoint*** *of software engineers in industry*. This case study is designed and reported according to the linear-analytic structure template suggested by Runeson et al. [36]. In particular in the next sections we present the four parts of our research design, i.e., research questions (see Section 4.1.1), case selection (see Section 4.1.2), data collection (see Section 4.1.3), and analysis (see Section 4.1.4).

4.1.1 Research Questions

According to the above-mentioned goal we have derived two research questions (RQ) that will guide the case study design and the reporting of the results:

RQ_1: *Is the proposed approach able to accurately identify the functionalities in a given method?*

This research question will explore the recall and precision rates [18]. Specifically, first we assess whether the approach has successfully identified all functionalities offered by the methods under study (i.e., the percentage of functionalities identified); and second the precision of the identification (i.e., the percentage of the identified Extract Method opportunities that match a specific functionality).

RQ_2: *Are the candidate refactorings, as proposed by the approach, ranked according to practitioners' perception of the benefit of applying those refactorings?*

This research question will explore the efficiency of the sorting part of the approach, based on the benefit obtained from extracting the candidate refactoring. The benefit is measured in terms of $LCOM_2$. The sorting algorithm should be able to prioritize the Extract Method opportunities that concern a specific functionality (which would constitute a coherent new method), among a number of candidates.

4.1.2 Case Selection

This study is a holistic multiple case study that has been conducted in a large company producing printers in the Netherlands. In this study as cases and corresponding units of analysis we consider the Long Methods. As case study participants we selected three software engineers that are currently working on improving the understandability of the explored methods.

In particular we have been provided with two Long Methods (for confidentiality reasons named as M_1 and M_2). M_1 consists of 408 lines of code and is responsible for preparing an image for printing, whereas M_2 consists of 642 lines of code and is responsible for processing the image while printing.

4.1.3 Data Collection

Collected Data. To answer the research questions mentioned in Section 4.1.1, we collected the following data items for each method:

- A list of the blocks of code that provide a specific functionality, based on the expert opinion of the participants. This list of functionalities is going to be used as the gold-standard for assessing the precision and recall of the SEMI approach (onwards referred to as *set of functionalities*).
- A sorted list of candidate refactorings (i.e., blocks of code) with respect to the **benefit** that can be obtained when these blocks are extracted as separate methods, based on the expert opinion of the participants (onwards referred to as *sorted refactoring opportunities*). The list of candidate refactorings opportunities (before sorting) has been obtained from SEMI to the two Long Methods of the company.

Collection Process. To collect the data required for our study, we conducted a workshop with three industrial practitioners, working for the company. The participants have been involved in the original construction and/or maintenance of the source code of the company, and more specifically with methods M_1 and M_2. The workshop was composed of two parts:

- S*tructured interviews*. According to [36] structured interviews consist basically of closed questions and can be similar to questionnaire-based surveys. For the needs of our study, we asked a set of closed questions (in some cases followed by an open question for explanation purposes). Due to the technical nature of the questions, the participants received the questions on paper and they were asked to write down their answers after working on the respective tasks. The researchers were present during the whole process, so the method can be compared to a supervised questionnaire-based survey [34]. The presence of the researchers in the room aimed at eliminating the disadvantages of simply distributing a questionnaire, like the lack of clarifications.
- *Focus group.* During the focus group the answers provided in the first part were discussed, giving the opportunity to clarify potentially different points of view or disagreements between the participants. The focus group could not bias the participants, because it was conducted at the end of the workshop, after the participants had submitted their completed questionnaires. The main goal of the focus group was to discuss and finalize starting and ending points of statement clusters that provide functionalities. For example, one engineer might have suggested that a specific functionality starts on statement 72 and another that the same functionality starts on statement 75. In this case, only one starting point was assigned to the cluster.

As preparation for the workshop we applied the proposed approach using as input the source code of methods M_1 and M_2. Therefore, two sets of *ranked candidate Extract Method opportunities* have been identified (a set of 33 Extract Method opportunities for M_1 and a set of 25 for M_2). The questions of both interviews and the focus group regarded the functionalities existing in the methods, the accuracy and completeness of the candidate refactorings, and the importance of extracting them to new methods.

Collecting the Set of Functionalities: First we asked the participants to identify as many functionalities as possible in the source code of M_1 and M_2, as well as the parts of the code that implement these functionalities. With this task we aimed at exploring if all functionalities have been identified by our approach (**RQ_1**). In particular, the participants identified eight functionalities for M_1, and six functionalities for M_2. The extracted functionalities are listed below:

- **M1**: (a) Fill swath entry, (b) Prepare plane, (c) Extract mask, (d) Mask swath, (e) Fill working buffer, (f) Update swath position, (g) Perform nozzle failure correction, and (h) Column reduction
- **M2**: (a) Calculate Y-offset, (b) Y-correction, (c) Rotate, (d) Update dot counter, (e) Determine array range, and (f) Scramble swath

Collecting the Sorted Refactoring Opportunities: Next, we provided the participants of the case study with a shuffled subset of the aforementioned Extract Method opportunities[9] and asked the participants to assign a score (1=min-5=max) based on the benefit of extracting each of the candidate methods into a new method. The benefit is evaluated based on two components: (a) the extent to which the approach identifies a complete functionality

[9] We were not able to provide practitioners with the whole set of refactoring opportunities (96 for M_1 and 65 for M_2, including their alternatives), due to time limitation. For selecting the 21 extract method opportunities we randomly selected opportunities from all parts of the list (7 from first 1/3 of the ranked list, and so on).

and (b) the design benefit gained from the extraction[10]. These two components of the scale are based on the two pillars of the proposed approach: each extract method opportunity should correspond to one functionality (compliance to SRP); and it should improve the design quality of the resulting system (the enhancement of quality is a basic advantage that should be offered by any refactoring [17]). The used scale is described as follows:

1. ***No Benefit (Not a functionality)***: The part of the code can't be mapped to a concrete functionality.
2. ***Limited Benefit (Not complete functionality)***: The part of the code does not provide a functionality (as a whole), but an adequate part of it.
3. ***Partial Benefit (Almost complete functionality)***: The part of the code could provide a functionality, by adding or deleting a small number of lines.
4. ***Only Functional Benefit (Complete functionality – No design improvement):*** The part of the code provides a functionality, but the benefit of extracting is not clear (e.g., it is highly coupled to the rest of the method, it is too small/large in size, etc.).
5. ***Optimal Benefit (Complete functionality - Design Benefit)***: The part of the code provides a functionality and its extraction as a different method provides benefits for the design of the system.

With this task we aimed at investigating the efficiency of the ranking approach, which is responsible for prioritizing the Extract Method opportunities with respect to their benefit for extraction (**RQ_2**). The ranking of the opportunities for the two methods is presented in Table VII. We note that each opportunity is assigned an id, composed by the method name and the number of the opportunity (i.e. $O_{1.1.}$ is the first opportunity of method M_1).

For assessing the evaluators' agreement, we used inter-rater reliability calculated through the intra-class correlation coefficient (ICC) [18]. The reliability for M1 has been calculated as 0.81 and as 0.29 for M2. The low reliability score for M2 suggests that the three reviewers expressed different opinions on the proposed ranking. A possible interpretation for this is the large size of M_2 (approx. 200 lines more than M1), which hindered its understanding, rendering the evaluation of refactoring opportunities more difficult. Thus, refactoring M2 has proven to be challenging even for experienced software engineers, with expertize on the specific method. For this reason, specifically for M2, we decided to consider only the opinion of the most experienced reviewer. The workshop organization and the questions used in the interviews and focus group are presented in the Appendix.

4.1.4 Data Analysis

In this section, we present the data analysis process that has been used for answering the research questions described in Section 4.1.1.

[10] The term *design benefit* has been intentionally provided to the participants in such an abstract form, since we were not aiming at a specific quality attribute, but only to the generic feeling of the software engineer, on whether the design would improve after the refactoring.

TABLE VII. Sorted Refactoring Opportunities

Method and Opportunity		Mean Score (SD)	Method and Opportunity		Mean Score (SD)
M_1	$O_{1.1}$	5.00 (0.00)	M_2	$O_{2.1}$	4.33(1.15)
	$O_{1.2}$	5.00 (0.00)		$O_{2.2}$	3.67 (2.31)
	$O_{1.3}$	5.00 (0.00)		$O_{2.3}$	3.67 (1.15)
	$O_{1.4}$	4.67 (0.58)		$O_{2.4}$	3.67(1.15)
	$O_{1.5}$	4.67(0.58)		$O_{2.5}$	3.33(0.58)
	$O_{1.6}$	4.33 (1.15)		$O_{2.6}$	3.33 (2.08)
	$O_{1.7}$	4.33 (0.58)		$O_{2.7}$	3.33(0.58)
	$O_{1.8}$	4.33 (0.58)		$O_{2.8}$	3.33(2.08)
	$O_{1.9}$	4.33 (1.15)		$O_{2.9}$	3.00(2.83)
	$O_{1.10}$	4.33 (1.15)		$O_{2.10}$	3.00(0.00)
	$O_{1.11}$	4.00 (1.00)		$O_{2.11}$	3.00(2.00)
	$O_{1.12}$	4.00 (1.41)		$O_{2.12}$	3.00(2.00)
	$O_{1.13}$	3.00(1.00)		$O_{2.13}$	3.00(1.73)
	$O_{1.14}$	3.00 (1.00)		$O_{2.14}$	2.67(1.53)
	$O_{1.15}$	2.33 (1.53)		$O_{2.15}$	2.33(0.58)
	$O_{1.16}$	2.33 (1.53)		$O_{2.16}$	2.33(1.53)
	$O_{1.17}$	2.33 (1.53)		$O_{2.17}$	2.33(1.53)
	$O_{1.18}$	1.67 (0.58)		$O_{2.18}$	2.00(1.00)
	$O_{1.19}$	1.33 (0.58)		$O_{2.19}$	1.67(0.58)
	$O_{1.20}$	1.33 (0.58)		$O_{2.20}$	1.33(0.58)
	$O_{1.21}$	1.00 (0.00)		$O_{2.21}$	1.33(0.58)

Identification Accuracy: For answering RQ_1 we will use three well-known metrics: namely F-measure, recall and precision [18]. All metrics are calculated three times for every method, by varying the tolerance in the approach. Specifically, we use the following tolerance values: 1%, 2%, and 3%. We note that although the accurate identification of functionalities is desirable there are cases that a functionality might be approximately identified. However, especially in large methods (e.g., 500 lines), pointing to a functionality with an accuracy of ±15 statements (i.e. 3% tolerance) is still expected to be beneficial for the software engineer. Further increasing the tolerance would lead to even higher precision and recall; however, we preferred to be strict in the evaluation of our approach so as to present a "worst-case scenario".

Recall is calculated as the fraction of correctly identified functionalities over the total number of functionalities that exist in the method according to experts' opinion (i.e., eight for M_1 and six for M_2). Similarly, *precision* is calculated as the ratio of correctly identified functionalities over the total number of identified refactoring opportunities (i.e., 33 for M_1 and 25 for M_2).

Ranking Accuracy: For answering RQ_2, we calculate the *Spearman Rank Correlation* between the expert ranking of refactoring opportunities, presented in Table VII, and the ranking that the approach offers for the same list of opportunities ($O_{1.x}$ and $O_{2.x}$, as discussed in Section 4.1.3).

We note that the correlation analysis is only based on the ranking and not the actual values that are assigned on the one side from the experts and on the other side from the approach. Therefore, the difference in nature of the two assessments (i.e., a scale for evaluators and cohesion for SEMI) is not biasing the results.

4.2 Results

In this section we present the results of our study organized by research question. In this section we compare our results to the literature and provide initial interpretations. A joint discussion of the results of both case studies is provided in Section 6.1.

4.2.1 Identification Accuracy (RQ1)

In this section we evaluate the proposed approach with respect to its accuracy when identifying Extract Method opportunities. As explained in Section 4.1.4, we will present results for each method separately and for three tolerance values. Therefore, in Table VIII we present the recall and precision of the SEMI approach for M_1 and M_2.

TABLE VIII. Approach Accuracy

	#Funcs	Total EMO[11]	Tolerance	Correct Func[12]	Recall	Precision	F-measure
M_1	8	33	1%	5	62.5%	15.1%	24.32%
		33	2%	8	100.0%	24.2%	38.97%
		33	3%	8	100.0%	24.2%	38.97%
M_2	6	25	1%	3	33.3%	12.0%	17.64%
		25	2%	5	83.3%	20.0%	32.26%
		25	3%	5	83.3%	20.0%	32.26%
Total	14	58	1%	8	57.1%	13.8%	22.23%
		58	2%	13	92.8%	22.4%	36.09%
		58	3%	13	92.8%	22.4%	36.09%

As it can be observed from Table VIII, the recall rate of the proposed approach ranges from 57% (i.e., 8 out of the 14 functionalities offered by both methods) to 93 % (i.e., 13 out of the 14 functionalities offered by both methods). Precision ranges from approximately 14% to 22%, i.e., the algorithm identifies 58 Extract Method opportunities, and through these opportunities 8-13 functionalities (depending on the tolerance) are retrieved. Compared to related work (i.e., other techniques that aim at the identification of Extract Method opportunities), the proposed approach achieves the highest recall rate, since the highest recall until now was 63%-75%, achieved by jDeodorant [38]. Concerning studies aiming at feature location, the recall rate of the proposed approach is comparable to the one of Antoniol et al. [4] and higher than the recall rate of the approach of Yosida et al. [41]. With regard to precision, our approach presents a rather low rate, compared to the highest rates in one study (i.e., jDeodorant achieves approximately 50% precision), but is comparable to the rest of the studies (see [4], [37]). However, an independent evaluation of the Extract Method algorithm provided by jDeodorant, suggested that its precision rate is closer to the average precision of similar techniques (i.e., lower than 10% [37]). We note that such comparisons are coarse-grained, in the sense that different datasets have been used in the compared studies. A fair comparison of the approaches will be provided in Section 5, where we present the results of a comparative case study using a uniform dataset and golden standard for the involved approaches. Therefore, no interpretations on the outcome of the comparison are provided in this section.

A possible interpretation of the lower recall and precision rates for M_2 can be the larger size of the method per se. The difficulty of functionality identification inside larger methods is also evident by the differences in the expert responses, as implied by the low reliability rates (see end of Section 4.1.3). To investigate the scalability of the proposed approach in Section 5.2.2 we further investigate the differences of recall and precision rates when investigating methods of various sizes.

4.2.2 Ranking Accuracy (RQ2)

To evaluate the ability of the approach to rank Extract Method opportunities, we present the results of the Spearman correlation test that we performed between the algorithm ranking, and experts' opinion. The results shown in Table IX concern the correlation between the ranking as obtained by our approach and the opinion of experts as presented in Table VII.

TABLE IX. Raking Accuracy

	M_1	M_2
Correlation Coefficient	0.479	0.477
Sig.	< 0.02	< 0.02

The results of Table IX suggest that the ranking offered by the approach is moderately correlated with the ranking based on experts' opinion [22][13]. Thus, if a software engineer starts evaluating the proposed opportunities by following the suggested order, he/she might be able to identify all relevant Extract Method opportunities without exhaustively parsing the list. A similar approach has been employed also by Silva et al. [37]. We note that, for the special case of M2 (i.e., low agreement rate among raters), the correlation of the rankings becomes 0.63, if we consider, only the opinion of the software engineer with the highest expertise on the specific method.

Furthermore, the results suggest that the ranking of candidate extract method opportunities based on the benefit in terms of cohesion can help improve the identification accuracy. In particular, we can observe that 93% of the functionalities have been identified in the Top-33 suggestions concerning M_1, and the Top-25 suggestions concerning M_2 (see Table VIII). This ability of the ranking algorithm to reduce the searching space for applying extract method opportunities, by 63 and 40 opportunities

[11] Total EMO: Total Number of Identified Extract Method Opportunities

[12] Correct Func: Correctly located functionalities

[13] According to Marg et al. [28] a correlation is characterized as strong if the correlation coefficient ranges from 0.40 to 0.69.

respectively, leads to an increased precision rate. The increase of precision by using a ranking/prioritization of Extract Method opportunities has also been reported by Silva et al. [37], and therefore is considered as an expected finding.

Finally, since we acknowledge that software engineers would more probably inspect only a limited number of suggestions (e.g., Top-10), we highlight that the approach is still able of identifying 7 out of 14 industrial functionalities, with a precision rate of 35% (F-measure: 0.41). Thus, based on F-measure, retaining the Top-10 suggestions from SEMI we achieve the better combination of precision and recall, compared to retaining Top-33 and Top-25 suggestions for M_1 and M_2. Summing up, although the correlation scores in Table IX do not suggest a strong relationship, but only a moderate one, precision and recall of the approach increases, when parsing only the top-X extract method refactoring opportunities. In particular by retaining only the top-10 suggestions for both methods, SEMI achieves an accuracy of 41% based on the F-measure, which is increased compared to retaining top-25 and top-33 suggestions (F-measure: 36%).

5 Comparative Case Study

In this section we present the design and the results of a case study on open source software projects, which aims at comparing the accuracy of SEMI to other state-of-the-art approaches. The case study has been performed on projects that have already been used in the literature as a benchmark for extract method identification approaches.

5.1 Case Study Design

The goal of this case study, expressed with a GQM (Goal Question Metric) formulation [7], is to ***analyze*** *the SEMI approach* ***for the purpose*** *of evaluation* ***with respect to****: (a) its ability to accurately identify extract method opportunities, and (b) the scaling of SEMI's accuracy when investigating longer methods* ***from the viewpoint*** *of software engineers*. Similarly to Section 4, this case study is also designed and reported according to the linear-analytic structure template suggested by Runeson et al. [36], and the same sub-section structure is used.

5.1.1 Research Questions

According to the above-mentioned goal we have derived two research questions (RQs):

RQ_1: *How does the accuracy of the SEMI approach compare to other state-of-the art tools/approaches?*

SEMI is not the only approach/tool proposed in the literature for suggesting extract method refactoring opportunities. Therefore, the aim of this research question is to compare the accuracy of SEMI to the accuracy of two state-of-the-art approaches: JDeodorant [37] and JExtract [38]. We note that we have preferred not to perform the comparison within the industrial setting presented in Section 4, because the industrial data could not be made available, a fact that would weaken the presentation and the replicability of this case study. To answer this research question we will use the same metrics as in Section 4 (i.e., F-measure, recall and precision rates [18]). In order to be able to provide a fair comparison among the approaches/tools we execute all of them in the same software projects/methods.

RQ_2: *How does the scalability of the SEMI approach compare to other state-of-the-art approaches?*

All existing approaches for extract method opportunities identification (including SEMI—as presented in Section 4.2) suffer from either low precision or low recall. In addition to that, in Section 4.2 we have discussed that the accuracy of SEMI is slightly decreased when applied to the longer industrial method. This can be considered as an expected finding, in the sense that the difficultly of analyzing a longer method is considered a more complex task. Thus, an interesting point of investigation is the scability of a method extraction approach, i.e. the ability of the approach to retain a certain level of accuracy as the size of the examined method increases. This research question aims at assessing the scalability of all the approaches compared in RQ_1, and investigating the expected decrease in accuracy.

5.1.2 Case Selection

This study is a holistic multiple case study that has been conducted on five open source software (OSS) projects. In this study as cases (and therefore also units of analysis) we consider a subset of the OSS projects' methods. The selection of methods and projects has been based on the original studies in which JDeodorant and JExtract have been evaluated (i.e., [38] and [37] respectively). We preferred to not limit our investigation to only one of the two approaches' benchmarks, so as not to bias our validation in favor of one of the two approaches. The examined OSS projects are: (a) Wikidev, (b) MyPlanner, (c) MyWebMarket, (d) Junit, and (e) JHotDraw.

In total, among the methods of the five projects the authors of the original studies have isolated 132 methods, in which they have identified 155 extract method opportunities. To identify the extract method opportunities Tsantalis et al. (projects: MyPlanner and Wikidev) have contacted projects' main developers to get their expert opinions, whereas Silva et al. used the first author's opinion for MyWebMarket (he is the developer of the project), and artificially created long methods for JUnit and JHotDraw, by merging methods that were invoked inside others (for more details see [39]).

5.1.3 Data Collection

To answer the RQs mentioned in Section 5.1.1, for every method we have recorded the following data items:

- ***LoC*** – Method Size in statements.
- ***Golden Standard*** – Lines of code to be extracted in a new method, as suggested either by experts or by the technique followed by Silva et al. [37].
- ***Best Matching Opportunity Identified in the Top-5 suggestions of SEMI***
- ***Best Matching Opportunity Identified in the Top-5 suggestions of JDeodorant***

- ***Best Matching Opportunity Identified in the Top-5 suggestions of JExtract***

We note that we have selected to retain the Top-5 suggested opportunities for all tools, since practitioners are expected not to investigate all opportunities provided by the tools. The decision to limit the size of the set with retained opportunities to five, was driven by our intention to be as strict as possible while evaluating the examined approaches/tools.

To collect the data, SEMI and JDeodorant have been executed with their default parameters, whereas JExtract has been configured so as to: (a) suggest extract method opportunities of minimum size of 2 lines, (b) provide unlimited number of recommendations, and (c) suggest only the extraction of continuous code fragments. This configuration has been performed so as to ensure the fair comparison to the other two tools/approaches. Finally, to ensure the replicability of our case study, we have made our dataset and golden standard available online[14].

5.1.4 Data Analysis

In this section we present the data analysis process that has been followed for answering the research questions described in Section 5.1.1.

- **Identification Accuracy**: For answering RQ_1 we will use the three metrics used in Section 4, namely F-measure, recall and precision [18]. Similarly to Section 4, all metrics are calculated three times for every method, by varying the tolerance of the approach (1%, 2% and 3%).
- **Scalability of the Accuracy**: For answering RQ_2, we calculate F-measure, recall and precision for a subset of the benchmark methods, including only the longest ones. Therefore, we isolated 15 methods with more than 30 statements[15] and compared the accuracy of the approaches in the complete dataset (on average approximately 18 LoC/method) with the accuracy of the approaches in only the longer ones (on average approximately 58 LoC/method).

5.2 Results

In this section we present the results of our comparative case study organized by research question. The number of extract method opportunities identified by each approach for all projects is presented in Table X.

TABLE X. Identified Opportunities

Tool	All Methods (132 cases)	Longer Methods (15 cases)
SEMI	737	228
JDeodorant	137	28
JExtract	7,612	4,057

From the results of Table X we can observe that the most conservative tool/approach is JDeodorant that makes on average 1 suggestion for smaller methods and less than 2 for longer methods, whereas the exhaustive approach of JExtract identifies approximately 55 opportunities for all methods and approximately 270 opportunities for the longer methods. Nevertheless, by retaining the Top-5 suggestions from JExtract and SEMI, the number of opportunities is limited to 638 and 75 respectively for all and long methods.

5.2.1 Identification Accuracy (RQ_1)

In this section we compare the accuracy of SEMI to state-of-the-art approaches. Therefore, in Table XI we present the F-measure, recall and precision of the three examined approaches for all methods. In Table XI with grey cell shading we denote the approach that presents the best accuracy.

TABLE XI. Approach Accuracy (all methods)

Tools	Tolerance	Recall	Precision	F-Measure
SEMI	1%	38,0%	12,9%	19,2%
	2%	47,0%	14,6%	22,3%
	3%	55,5%	18,8%	28,1%
JDeodorant	1%	14,8%	17,4%	16,0%
	2%	18,4%	21,1%	19,7%
	3%	23,8%	28,0%	25,7%
JExtract	1%	52,2%	12,6%	20,4%
	2%	59,3%	13,1%	21,5%
	3%	61,9%	15,0%	24,2%

Based on the results of Table XI, we can observe that SEMI presents the most accurate approach in terms of F-measure, whereas JDeodorant in terms of precision and JExtract in terms of recall. The fact that jDeodorant is presenting the highest precision rate is probably due to the conservative strategy in identifying extract method opportunities, which leads to a low number of false positives. In particular, the slicing algorithm of jDeodorant calculates the computational slice of exactly one variable, without any provision for merging extract method opportunities into larger ones. Nevertheless, this strategy limits recall as well. We note that by considering the whole list of extract method opportunities suggested by all tools (not retaining only Top-5 suggestions), JExtract achieves a recall rate of nearly 96% (with 3% tolerance), but with a very limited precision (approx. 2%), leading to an F-measure of 4%. In the same setting SEMI and JDeodorant present a similar F-measure (23% and 25% respectively).

5.2.2 Scalability of the Accuracy (RQ_2)

By focusing on only the long methods of our dataset, the accuracy of the examined tools/approaches is substantially differentiated, as presented in Table XII.

TABLE XII. Approach Accuracy in Long Methods

Tools	Tolerance	Recall	Precision	F-Measure
SEMI	1%	38,7%	16,4%	23,0%
	2%	41,9%	17,9%	25,0%
	3%	45,1%	19,1%	26,9%

[14] http://www.cs.rug.nl/search/uploads/Resources/TSEdataset.xls

[15] The intuition that methods with more than 30 lines are expected to be long has been suggested by Lippert and Roock [27].

Tools	Tolerance	Recall	Precision	F-Measure
JDeodorant	1%	9,6%	12,0%	10,7%
	2%	12,9%	14,3%	13,5%
	3%	12,9%	16,0%	14,2%
JExtract	1%	16,1%	6,6%	9,4%
	2%	19,3%	8,0%	11,3%
	3%	19,3%	8,0%	11,3%

From the results of Table XII, we can observe that by focusing only on methods with more than 30 lines of code, SEMI presents the best precision, recall, and F-measure for all levels of tolerance. Another, interesting finding is that SEMI performs very similarly for long and small methods, in contrast to other approaches/tools. In particular, by comparing the results of Table XI and Table XII, we can observe that in longer methods JDeodorant performs approx. 36% worse in F-measure, whereas JExtract approx. 51% worse. On the contrary, the recall of SEMI is decreased by only 9%, the precision increases 17%, and the F-measure increases by 9%.

6 Discussion

6.1 Interpretation of Results

The results of the performed case studies suggest that the SEMI approach achieves top F-measure rates compared to the state-of-the-art approaches on both Extract Method opportunities identification (see Section 3.1) and feature/functionality identification (see Section 3.2). This result becomes more evident in cases that extract method opportunities identification is performed on longer methods (i.e., more than 30 statements). This outcome suggests that *the single responsibility principle can be applied inside the body of a method*. SRP has been originally introduced at the design level and specifically for the extraction of classes from other larger ones. However, the results of this study suggest that multiple functionalities offered by the same method can be identified using the same approach. The extraction of such sets of statements to separate methods has been validated as useful by the experts participating in our case study. In addition to that, the results strongly suggest that *the use of method body cohesion metrics for identifying Extract Method opportunities* is accurate. In particular, the proposed approach has in total identified 13 out of 14 functionalities offered by two very long methods (approx. 500 lines of code) that we have examined, as indicated by the software engineers working on them (i.e., a recall rate of 92.8%). Furthermore, although the results of our case study suggest that the proposed approach is not achieving high precision rates, this is can be explained as follows:

Expected trade-off between precision and recall. In every classification approach the two measures of accuracy (i.e., precision and recall) are contradicting. Therefore, since the goal of this algorithm is to identify as many functionalities/Extract Method opportunities as possible, lower precision rate is preferable, in order to achieve top recall. The same outcome can be observed also in the study of Antoniol et al. [4], where in order to achieve 100% recall, the precision dropped to 13%. A possible reason for the improved precision of slicing-based approaches is that they have a much narrower scope, since they aim at extracting statements affecting a variable or in the best case scenario, the entire calculation of a variable. Such a goal is significantly more *bounded* than the selection of arbitrary functionalities involving numerous variables. Nevertheless, we note that the comparative case study has revealed that other existing approaches for extract method opportunities identification suffer from the same problem. To combine precision and recall in a single measure that takes into account this trade-off, we have used F-measure. The results suggest that based on F-measure, SEMI can handle this trade-off more efficiently compared to other approaches (highest F-measure rates).

The order of magnitude in the method size. As explained in Section 2 (see Table I), our industrial case study has tested SEMI on substantially long methods than any other approach. Since the size of the problem searching space increases exponentially with the growth of the method size, it is expected that the longer the method, the harder it is to accurately identify all functionalities. We expect especially precision to be influenced from this factor since the number of identified opportunities increases in longer methods. Nevertheless, the evidence obtained by the comparative case study, suggested that SEMI scales better than existing techniques in terms of precision, when the size of the method is increasing.

6.2 Reliability of Results

To test if the use of a different cohesion metrics differentiates the previously discussed results, we replicated the data collection for the industrial case study by using another cohesion metric (namely, Class Cohesion - CC [13]) in the calculation of the primary benefit measure. The selection of CC has been based on the facts that: (a) it is the top predictor of the existence of the long method bad smell [14], and (b) it has different properties compared to $LCOM_2$ in the sense that it is a normalized measure, it is not correlated to size, and it captures cohesion instead of its lack. The precision and recall of the proposed approach when using CC are presented in Table XIII.

TABLE XIII. Approach Accuracy (Metric: CC)

	#Funcs	Total EMO	Tolerance	Correct Func	Recall	Precision	F-measure
M_1	8	41	1%	5	62.5%	12.2%	20.41%
		42	2%	7	87.5%	16.7%	28.05%
		42	3%	8	100.0%	19.0%	31.93%
M_2	6	72	1%	4	66.6%	5.6%	10.33%
		72	2%	5	83.3%	6.9%	12.74%
		72	3%	5	83.3%	6.9%	12.74%
Total	14	113	1%	9	64.3%	7.9%	14.07%
		114	2%	12	85.7%	10.5%	18.71%
		114	3%	13	92.8%	11.4%	20.31%

The results suggest that using CC:

- ***recall*** is increased (64.3% - 92.8%) and thus *produces top results compared to the state of the art*; and
- ***precision*** is decreased (7.9% - 11.4%), but is still comparable to research state of the art (precision of such approaches is approximately 10%).

6.3 Implications for Researchers & Practitioners

The results of this study are expected to prove useful to both researchers and practitioners. Concerning practitioners, we expect that the proposed approach and the corresponding tool (see Section 6.3) will help them to ***improve the design-time quality of their code***. This improvement comes from two characteristics, namely the generic benefits of Extract Method refactoring and the benefits of applying the SRP:

- ***Generic benefits***. The refactoring of Long Methods and the consequent improvement of cohesion, caused by applying the SEMI approach has been related to improving quality attributes (e.g., maintainability [16] and reusability [6]).
- ***SRP-based benefits***. The Extract Method opportunities derived based on the Single Responsibility Principle [29] are expected to provide additional benefits in terms of modularity. In particular, the fact that each functionality is going to be encapsulated in a separate method decouples the axes of change for a specific class [29]. Furthermore, the enhanced modularity is expected to further boost reusability, not only in terms of ease in adjusting the reusable part of the code into the target system [25], but also in terms of "cleanly" reusing only the desired code, without needless repetition [29]. Finally, resolving modularity issues is expected to reduce the amount of technical debt[16] accumulated in software systems, since according to Alves et al. [2] modularity violations and code smells are its most common indicators.

Regarding researchers, the study led to some interesting implications and future work directions. First, the benchmark created for our comparative case study can be useful both in the domain of feature location and refactorings identification, which currently lack a set of methods with identified functionalities/extraction opportunities. The provision of this benchmark will enable the fair comparison of future approaches and reduce deviations in recall and precision, caused by using different systems as objects. Second, the fact that SRP and cohesion are successfully tailored to apply at the method level, opens new research directions on how other principles can be transferred to different levels of granularity, e.g., architecture or code. Finally, the approach can be tailored to fit the identification of additional refactoring opportunities. We believe that such a tailoring constitutes an interesting future work, since different refactoring opportunities require completely different identification algorithms, checking of preconditions, ranking approaches and evaluation strategies. For example, even for refactorings of similar purpose (e.g., extract parts of the code in different levels of granularity—i.e., extract methods, extract class, etc.) the required approaches should be different: in extract class you need to investigate the clusters of methods and attributes that should be placed in the new class, whereas in the extract method you need to investigate which lines of code are functionally relevant, do not violate AST preconditions, determine the number of parameters for the new method, etc. Thus, despite the fact that in both cases a cohesion-based approach is required, the same approach cannot be directly transferred from the one code smell to the other.

6.4 Tool Support

The SplitLongMethod tool is comprised of two parts: the `Long Method detector` that identifies Long Methods in large codebases (as presented in our previous work—see [14]); and the `Extract Method opportunities` that presents the identification, grouping and ranking of all possible Extract Method opportunities identified in a single Long Method. We note that the user has the option to freely select the metric that will be used for assessing the cohesion among statements and prioritizing the Extract method refactoring opportunities.

Long Method Detector: The tool analyses Java classes and the results are presented in two components, as shown in Fig. 10:

- *Results Table* (see right side of the User Interface (UI)): Presents method names, the cohesion metric score, and a suggestion whether the method is in need of refactoring or not. The methods are ranked based on the selected metric.
- *Heatmap* (see left side of the UI): Visually represents the same information. The size of each box depends on the ranking of the method, whereas the color (binary value) if it needs refactoring or not.

Extract Method Detector. The second part of the tool (related to the SEMI approach) focuses on a specific method. The selected method is analyzed and Extract Method opportunities are identified. After the identification of the Extract Method opportunities, the grouping and ranking algorithm is executed. The obtained groups of Extract Method opportunities are ranked based on the selected cohesion metric and are presented in Fig. 11:

- *Extract Method Opportunities* (left side of the UI): Lists all the identified extract method opportunities, and their expected benefit, ranked by benefit.
- *Alternatives* (center of the UI): Lists all alternatives (i.e. similar opportunities with minor differences in terms of starting/ending LoC), for the selected extract method opportunity, ranked by benefit.
- *Source Code* (right side of the UI): Highlights the code of the selected extract method opportunity

[16] Technical debt is perceived as any compromise made in order to add business value to software systems (e.g., shrink product time to market). More details on technical debt research can be found in [3].

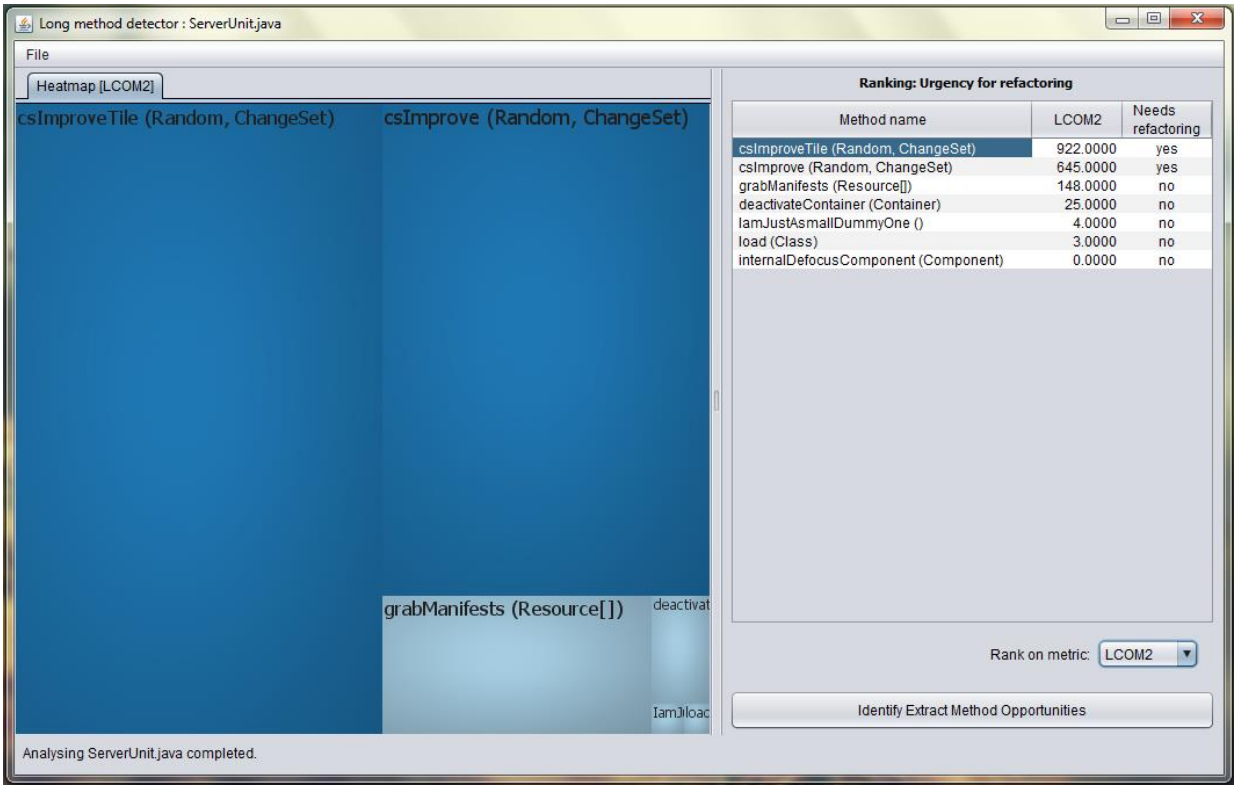

Fig. 10: Long Method Detector

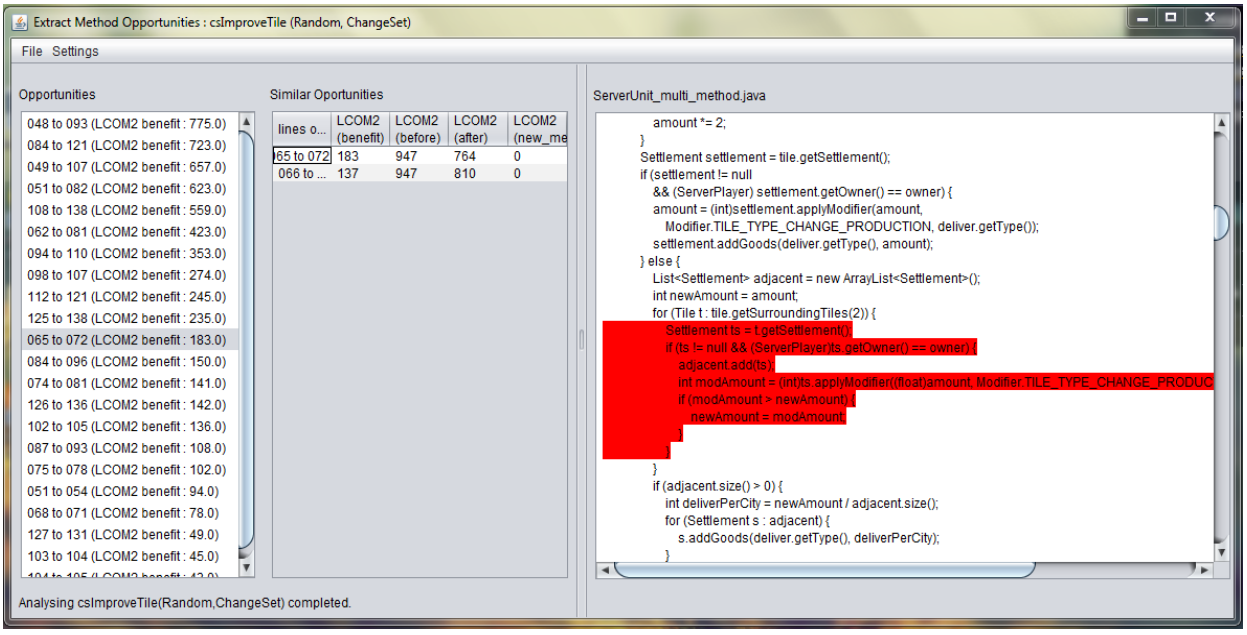

Fig. 11: Extract Method Opportunities UI

7 Threats to Validity

In this section, we present and discuss potential threats to the validity of our case studies [36]. Internal validity is not considered, since causal relations are not in its scope.

7.1 Construct Validity

A possible threat to construct validity is related to the accuracy of approaches that are used to identify and rank Extract Method opportunities. Such a threat is classified as construct validity in the sense that inaccurate results might lead to measuring a different phenomenon than the one originally intended to investigate.

The proposed algorithm for identifying, grouping, and ranking Extract Method opportunities is performed based on the assumption that common method calls and the use of common objects can indicate a potential relation between the corresponding statements. Thus, they

should be taken into account during the clustering process, which is performed considering cohesion. This assumption may pose a threat to validity since there are many definitions of cohesion [12] that do not take such relationships into account. However, the call of a same method, even with a different object denotes some similarity between statements, since they are in need of a same service (i.e., the one provided by the called method). Following a similar mindset, the use of the same object calling any method indicates the use of the same data and thus we can talk about data cohesion [42]. In addition to that, concerning the accuracy of the selected cohesion metric (i.e., $LCOM_2$), it is possible that the use of a different metric could affect the results of this study. However, we selected to use $LCOM_2$ based on our previous experience on method level cohesion metrics and their relation to Long Methods identification [14]. The main benefit of using LCOMs is its inherent correlation to both method's size and cohesion [14].

Moreover, the case study participants may have a different background and experience and thus influence the choice of selected functionalities and the ranking of refactoring opportunities. To avoid this threat, we involved three employees who were all familiar with the project under investigation. However, it is possible that participants have a different perspective of the methods, due to their different roles in the company (i.e., one refactorings expert and two developers). To mitigate this risk, we calculated their agreement rate (see Section 4.1). Specifically, we observed that concerning the first method the results show high agreement and thus constitute reliable results. On the other hand, regarding the second method the answers of the interviewees were not strongly correlated (but only moderately); this may suggest that more participants would be needed to obtain fully reliable results. However, this was not possible due to resource constraints in our industrial case study. To mitigate this threat we explore the ranking efficiency of SEMI not only based on the aggregated opinion of all experts but also to the opinion of the most experienced one. The results suggested that in both cases SEMI is able to provide a moderate to strong rank correlation.

7.2 Reliability

With regard to reliability, we consider any possible researchers' bias, during the data collection and data analysis process. The design of the study concerning data collection, does not contain threats, since the material provided to the participants included the source code of the company and clusters of code that had been created automatically by a tool. Additionally, the researchers themselves were not required to interpret the results at any point, since the participants were answering the tasks on paper. Moreover, with respect to the data analysis process, to mitigate any potential threats to reliability, two researchers were involved in the process, aiming at double checking the work performed and thus reducing the chances of reliability threats.

7.3 External Validity

Concerning external validity, a potential threat to generalization is the possibility that performing the study on different methods of different companies might affect the precision and recall rates. However, we believe that the selected industrial case, given its size and complexity, represents a realistic industrial system. Nevertheless, acknowledging the fact that this threat exists not only for this work, but also for all related previous studies, we emphasize the need for creating a benchmark for assessing such approaches (see Section 6.2).

Additionally, although the precision and recall of the proposed approach might change with the use of different parameters (e.g., a different cohesion metric in the calculation of the primary benefit measure), a sample experimentation (see Table XIII) has shown that these rates are not significantly influenced. A detailed discussion of these findings can be seen in Section 6.1.

8 Conclusion

This study proposes an approach for identifying Extract Method opportunities in the source code of Long Methods (namely SEMI), and ranking them according to the benefit that they yield in terms of cohesion. The proposed approach is based on the Single Responsibility Principle and its inherent relation to cohesion. Therefore, the approach identifies the largest possible cohesive sets of instructions and suggests their extraction.

To evaluate the approach, we conducted two case studies: (a) using two industrial Long Methods, which consisted of a total of 1,000 lines of code, and used experts' opinion as a golden standard, and (b) using open-source data to compare SEMI to state-of-the-art approaches/tools. To the best of our knowledge, the industrial case study is the largest one with regard to the size of the methods investigated. The results of the industrial case study indicate that the proposed approach can adequately identify functionalities inside the body of long methods, with high recall rates. In particular, we have been able to locate (with a ±3% tolerance) approximately 90% of functionalities that exist in these methods and suggest their extraction into different methods. The results of the comparative case study have indicated that SEMI outperforms existing approaches in terms of F-measure (i.e., a combination of precision and recall), and that it is the only approach that scales (i.e., retains high levels of F-measure) for methods of different sizes (ranging from 18 – 500 lines of code). In order to ease the adoption of our approach, we have developed a tool that automates its application for Java classes. Finally, we argue that the proposed approach can be useful to practitioners for applying the Extract Method refactoring.

Acknowledgment

This research has been partially funded by the ITEA2 project 11013 PROMES. We would like to thank Océ and the three engineers who participated in the study.

REFERENCES

[1] J. Al Dallal, "Measuring the Discriminative Power of Object-Oriented Class Cohesion Metrics", *Transactions on Software Engineering*. IEEE Computer Society, 37(6) , pp. 788 – 804. November/December 2011.

[2] N. S. R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, C. Seaman, "Identification and management of technical debt: A systematic mapping study", Information and Software Technology, Volume 70, pp.100-121, February 2016.

[3] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, "The financial aspect of managing technical debt: A systematic literature review", Information and Software Technology, Volume 64, August 2015, pp 52-73.

[4] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering Traceability Links between Code and Documentation". IEEE Trans. Softw. Eng. 28, 10. (October 2002), pp. 970-983.

[5] L. Badri, M. Badri, "A Proposal of a new class cohesion criterion: an empirical study", *Journal of Object Technology*. 3, 4 (April 2004), pp.145-159.

[6] J. Bansiya and C.G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment" IEEE Trans. Softw. Eng. 28, 1 (January 2002), pp. 4-17.

[7] V. Basili, G. Caldiera, D. Rombach, "The Goal Question Metric Approach", Encyclopedia of Software Engineering, John Wiley & Sons, pp. 528-532. 1994

[8] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures", J. Syst. Softw. 84, 3 (March 2011), pp.397-414.

[9] G. Bavota, R. Oliveto, A. De Lucia, G. Antoniol and Y. G. Guéhéneuc, "Playing with refactoring: Identifying extract class opportunities through game theory," Software Maintenance (ICSM), 2010 IEEE International Conference on, Timisoara, 2010, pp. 1-5.

[10] K. Beck, "Make It Run, Make It Right: Design Through Refactoring", SIGS Publications, 6 (4), 1997

[11] J. M. Bieman, B. Kang "Cohesion and reuse in an object-oriented system", *Proceedings of the 1st Symposium on Software Reusability* (Seattle, USA, 29 – 30 April 1995). SSR'95. ACM Press, pp. 259-262. 1995

[12] L. C. Briand and J. Daly, "A unified framework for cohesion measurement in object-oriented systems." *Empirical Software Engineering*. March 1998. Springer, 3, 1, pp. 65-117.

[13] C. Bonja, and E. Kidanmariam, "Metrics for class cohesion and similarity between methods", Proceedings of the 44th Annual Southeast Regional Conference. (ACMSE'06). Melbourne, Florida, 10-12 March 2006.. ACM Press, pp. 91-95.

[14] S. Charalampidou, A.Ampatzoglou, and P. 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). ACM, New York, NY, USA, Article 8, 10 pages. 2015.

[15] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of code smells in object-oriented systems", Innov. Syst. Softw. Eng. 10(1), pp. 3-18. March 2014.

[16] S. R. Chidamber and C. F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. IEEE Trans. Softw. Eng. 20, 6 (June 1994), 476-493.

[17] T. De Marco, "Structured Analysis and System Specification", Yourdon Press Computing Series. 1979.

[18] A. Field, 2013. Discovering Statistics using IBM SPSS Statistics. *SAGE Publications Ltd*.

[19] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. Identification and application of Extract Class refactorings in object-oriented systems. J. Syst. Softw. 85, 10 (October 2012), pp.2241-2260. 2012.

[20] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D.Roberts, "Refactoring: Improving the Design of Existing Code", Addison-Wesley Professional, 1 edition. July 1999.

[21] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, "Identifying Architectural Bad Smells," 13th European Conference on Software Maintenance and Reengineering (CSMR '09), pp.255-258. 24-27 March 2009

[22] D. Gregg, J. F. Power, and J. Waldron, "A method-level comparison of the Java Grande and SPEC JVM98 benchmark suites". Concurrency and Computation Practice and Experience 17(7-8), pp.757-773, 2005

[23] T. Hastie, R. Tibshirani, J. Friedman, "The Elements of Statistical Learning" , Springer New York Inc. , New York, NY, USA. 2001 .

[24] M. Hitz, and B. Montazeri, "Measuring coupling and cohesion in object oriented systems" *Proceedings of the International Symposium on Applied Corporate Computing* (Monterrey, Mexico, 25-27 October 1995). ISACC'95. Pp. 25-27. 1995

[25] G. Kakarontzas, E. Constantinou, A. Ampatzoglou, I. Stamelos, "Layer assessment of object-oriented software: A metric facilitating white-box reuse", Journal of Systems and Software, Volume 86, Issue 2, February 2013, Pages 349-366.

[26] R. Komondoor and S. Horwitz. "Semantics-preserving procedure extraction". 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '00). ACM, 155-169. 2000.

[27] M. Lippert and S. Roock, "Refactoring in Large Software Projects", 1st edition. Wiley & Sons. 2006.

[28] L. Marg, L.C. Luri, E. O'Curran, A, Mallett, "Rating Evaluation Methods through Correlation". *Proceedings of the 1st Workshop on Automatic and Manual Metrics for Operational Translation Evaluation* (Reykjavik, Iceland, 26 May 2014). MTE'14. 2014.

[29] R.C. Martin "Agile software development: principles, patterns and practices", Prentice Hall, New Jersey. 2003.

[30] P. Meananeatra, S. Rongviriyapanish, T. Apiwattanapong, "Using software metrics to select refactoring for long method bad smell", 8th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), pp.492-495. 17-19 May 2011.

[31] T. Mens, G. Taentzer, and O. Runge, "Analysing refactoring dependencies using graph transformation," Software and Systems Modeling, vol. 6, no. 3, pp. 269-285, 2007.

[32] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus. "How can I use this method?" In Proceedings of the 37th International Conference on Software Engineering (ICSE '15), Vol. 1. IEEE Press, Piscataway, NJ, USA, pp.880-890. 2015.

[33] E. Murphy-Hill, C. Parnin, A. P. Black, "How We Refactor, and How We Know It", Transactions on Software Engineering, IEEE Computer Society, 38 (1), pp. 5-18, January 2012.

[34] S. L. Pfleeger, B.A. Kitchenham, "Principles of survey research: part 1: turning lemons into lemonade", SIGSOFT Softw. Eng. Notes 26(6), pp.16-18. 2001.

[35] E. Piveta, J. Araujo, M. Pimenta, A. Moreira, P. Guerreiro, and R. T. Price, "Searching for Opportunities of Refactoring Sequences: Reducing the Search Space," 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2008), pp. 319-326, July 28-August 1, 2008.

[36] P. Runeson, M. Höst, A. Rainer, B. Regnell, "Case Study Research in Software Engineering: Guidelines and Examples", John Wiley and Sons, Inc. 2012.

[37] D. Silva, R. Terra, M. T. Valente, "Recommending automated extract method refactorings". In Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014), ACM, New York, NY, USA, pp.146-156. 2014.

[38] N. Tsantalis, A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods", Journal of Systems and Software, 84 (10), pp. 1757-1782. October 2011.

[39] N. Tsantalis and A. Chatzigeorgiou, "Ranking Refactoring Suggestions based on Historical Volatility", 15th European Conference on Software Maintenance and Reengineering (CSMR'2011), Oldenburg, Germany, March 1-4, 2011

[40] L. Yang, H. Liu, Z. Niu, "Identifying Fragments to be Extracted from Long Methods", Asia-Pacific Software Engineering Conference (APSEC '09), pp.43-49. 1-3 December 2009.

[41] N. Yoshida, M. Kinoshita, H. Iida, "A cohesion metric approach to dividing source code into functional segments to improve maintainability", 16th European Conference on Software Maintenance and Reengineering (CSMR), pp.365-370. 27-30 March 2012.

[42] E. Yourdon and L. L. Constantine., "Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design (1st Ed.)". Prentice-Hall, Inc., Upper Saddle River, NJ, USA. 1979.

[43] W. Zhao, L. Zhang, Y. Liu, J. Luo, J. Sun, "Understanding How the Requirements Are Implemented in Source Code", in the Tenth Asia-Pacific Software Engineering Conference, pp. 68-77, 10-12 Dec. 2003

[44] W. Zhao, L. Zhang, Y. Liu, J. Sun, F. Yang, "SNIAFL: Towards a Static Noninteractive Approach to Feature Location", Proceedings. 26th International Conference on in Software Engineering (ICSE 2004). Vol 15, no. 2, pp. 293-303, 23-28 May 2004.

Sofia Charalampidou is a PhD Student at the University of Groningen, the Netherlands, in the group of Software Engineering and Architecture. She holds an MSc degree in Software Engineering from Chalmers University of Technology, Sweden, and a BSc degree in Information Technology from the Technological Institute of Thessaloniki, Greece. Her research interests include software design, maintenance, and metrics.

Dr. Apostolos Ampatzoglou is a Guest Researcher in the Johann Bernoulli Institute for Mathematics and Computer Science of the University of Groningen (Netherlands), where he carries out research in the area of software engineering. He holds a BSc on Information Systems (2003), an MSc on Computer Systems (2005) and a PhD in Software Engineering by the Aristotle University of Thessaloniki (2012). His current research interests are focused on reverse engineering, software maintainability, software quality management, open source software engineering and software design. He has published more than 45 articles in international journals and conferences. He is/was involved in over 10 R&D ICT projects, with funding from national and international organizations.

Dr. Alexander Chatzigeorgiou is an associate professor of software engineering in the Department of Applied Informatics at the University of Macedonia, Thessaloniki, Greece. He received the Diploma in Electrical Engineering and the PhD degree in Computer Science from the Aristotle University of Thessaloniki, Greece, in 1996 and 2000, respectively. From 1997 to 1999 he was with Intracom S.A., Greece, as a telecommunications software designer. Since 2007, he is also a member of the teaching staff at the Hellenic Open University. His research interests include object-oriented design, software maintenance, and software evolution analysis. He has published more than 100 articles in international journals and conferences. He is a member of the Technical Chamber of Greece.

Antonis Gkortzis is a PhD Student at the Athens University of Economics and Business (Greece) in the Software Engineering and Security (SENSE) group. He holds an MSc degree in Software Engineering from University of Groningen (the Netherlands) and a BSc degree in Information Technology from the Technological Institute of Thessaloniki (Greece). His research interests include software security, object-oriented design, maintainability, and software quality assessment.

Dr. Paris Avgeriou is Professor of Software Engineering in the Johann Bernoulli Institute for Mathematics and Computer Science, University of Groningen, the Netherlands where he has led the Software Engineering research group since September 2006. Before joining Groningen, he was a post-doctoral Fellow of the European Research Consortium for Informatics and Mathematics (ERCIM). He has participated in a number of national and European research projects directly related to the European industry of Software-intensive systems. He has co-organized several international conferences and workshops (mainly at the International Conference on Software Engineering - ICSE). He sits on the editorial board of Springer Transactions on Pattern Languages of Programming (TPLOP). He has edited special issues in IEEE Software, Elsevier Journal of Systems and Software and Springer TPLOP. He has published more than 130 peer-reviewed articles in international journals, conference proceedings and books. His research interests lie in the area of software architecture, with strong emphasis on architecture modeling, knowledge, evolution, patterns and link to requirements.