Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

An automated extract method refactoring approach to correct the long method code smell

Mahnoosh Shahidi, Mehrdad Ashtiani [*], Morteza Zakeri-Nasrabadi

School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran

ARTICLE INFO

Article history:
Received 23 May 2021
Received in revised form 12 December 2021
Accepted 9 January 2022
Available online 17 January 2022

Keywords:
Software refactoring
Extract method
Long method
Code smell
Graph analysis

ABSTRACT

Long Method is amongst the most common code smells in software systems. Despite various attempts to detect the long method code smell, few automated approaches are presented to refactor this smell. Extract Method refactoring is mainly applied to eliminate the Long Method smell. However, current approaches still face serious problems such as insufficient accuracy in detecting refactoring opportunities, limitations on correction types, the need for human intervention in the refactoring process, and lack of attention to object-oriented principles, mainly single responsibility and cohesion–coupling principles. This paper aims to automatically identify and refactor the long method smells in Java codes using advanced graph analysis techniques, addressing the aforementioned difficulties. First, a graph representing project entities is created. Then, long method smells are detected, considering the methods' dependencies and sizes. All possible refactorings are then extracted and ranked by a modularity metric, emphasizing high cohesion and low coupling classes for the detected methods. Finally, a proper name is assigned to the extracted method based on its responsibility. Subsequently, the best destination class is determined such that design modularity is maximized. Experts' opinion is used to evaluate the proposed approach on five different Java projects. The results show the applicability of the proposed method in establishing the single responsibility principle with a 21% improvement compared to the state-of-the-art extract method refactoring approaches.

1. Introduction

Reaching high-quality design is of great concern in developing today's software systems where flexibility, reusability, and maintainability are essential (Subramaniam and Zulzalil, 2012; Yamashita and Moonen, a). Correctly adhering to object-oriented principles is a preliminary approach to make quality software. Developers usually leverage the principles such as single responsibility and modularity to achieve a good design. However, in some cases, they neglect or violate these principles due to the lack of time or budget (Taibi et al., 2017; Yamashita and Moonen, b), which results in confusing, complex, and problematic structures in code, known as code smells (Palomba et al., 2018; Fowler and Beck, 2018; Khomh et al., 2012). The lack of attention to these principles imposes various costs to the software maintenance phase (Kaur and Singh, 2019; Tsantalis et al.). Code smells are symptoms of poor design and implementation choices (Yamashita and Moonen, b; Tufano et al., 2017; Sobrinho et al., 2021). They can be identified and fixed through the software refactoring process, performed either manually by the developers (Fowler and Beck, 2018) or automatically by tools (Baqais and Alshayeb, 2020). Refactoring is the process of making modifications to the internal structure of the code without altering its external behavior (Fowler and Beck, 2018). According to recent studies (Tsantalis et al.; Lacerda et al., 2020; Agnihotri and Chug, 2020), the long method is between the topmost five code smells recognized by researchers and practitioners. However, only a few approaches have been tried to treat the long method code smell (Maruyama, 2001; Tsantalis and Chatzigeorgiou, 2011; Charalampidou et al., 2017), which are not fully automated and require manual operations to complete the refactoring process.

The approach proposed by Maruyama (2001) based on static backward slicing requires the developer to determine a target variable for slicing. In contrast, the approach proposed by Tsantalis and Chatzigeorgiou (2011) generates all possible code slices which can be extracted and reported to the developers to select the best one. Approaches based on program slicing also do not consider the quality of extracted parts in terms of quality attributes such as cohesion and single responsibility during the refactoring process. SEMI (Charalampidou et al., 2017) identifies the largest possible cohesive sets of instructions and suggests

* Correspondence to: School of Computer Engineering, Iran University of Science and Technology, Hengam St., Resalat Sq., Postal Code: 16846-13114, Tehran, Iran.

E-mail addresses: ma_shahidi@comp.iust.ac.ir (M. Shahidi), m_ashtiani@iust.ac.ir (M. Ashtiani), morteza_zakeri@comp.iust.ac.ir (M. Zakeri-Nasrabadi).

https://doi.org/10.1016/j.jss.2022.111221
0164-1212/© 2022 Elsevier Inc. All rights reserved.

their extraction to achieve single responsibility. However, this approach generates an exhaustive set of statements that may not comply with the goal of single responsibility or introduce new smells such as the long-parameters list after applying the refactoring.

One crucial challenge that has not been addressed by previous researches is the co-existence of smells (Tsantalis et al.). Lozano et al. have shown that almost all methods suffering from feature envy reside in long methods, and feature envy methods are often located in god classes. It is also highly possible to introduce new code smells when applying a single refactoring to fix a specific code smell (Bibiano et al.). For instance, extract method refactoring may simply result in code duplication, long-parameter list, or feature envy, which are among the most common code smells. Therefore, it is essential to avoid the occurrence of the aforementioned smells when applying the extract method refactoring. In addition, the extract method could be exploited to simultaneously find and fix other refactoring opportunities such as feature envy. It is shown that combining Extract Method with Move Method transformations removes Feature Envy instances thoroughly (Fowler and Beck, 2018; Cedrim et al.) while developers are possibly sub-exploring batches to remove these smells in the manual refactoring (Bibiano et al.).

This paper proposes a fully automated long method identification and extract method refactoring suggestion while preventing new code smells and improving the overall software quality. To this aim, the possible refactoring candidates are generated, and the best one is selected considering single responsibility, cohesion, and coupling principles in object-oriented design. Additional move method refactoring is suggested if feature envy code smell is identified after extract method refactoring is applied. Moreover, the best possible name for the newly extracted method is determined to automate the entire refactoring process.

The relationship between the existing parts in a software program could be best described through graphs. The proposed approach is based on using advanced graph analysis techniques in various levels of the program entities with different granularities. For each analysis, a corresponding graph is defined and constructed from relevant entities in the source code, including classes, methods, attributes, and statements. The graph properties such as betweenness centrality, modularity, and clustering coefficient, are powerful means to identify the refactoring opportunities and operations. The whole process of the proposed refactoring approach consists of five steps. In the first step, the long method instances are identified by analyzing external and internal method dependencies and the number of lines of code in methods' bodies. In the second step, the possibility of extracting the meaningful pieces of codes from a long method is checked by modularity analysis of the method's control and data dependency graph. This graph is then used in the third step to generate and rank the extract method candidates by clustering coefficient and modularity measurement. The candidate generation algorithm avoids creating long-parameter list methods or a method that violates the single responsibility principle while considering simple heuristics. Candidate ranking strategy makes it possible to have a fully automated refactoring pipeline. In step four, a proper name is automatically assigned to the extracted method by analyzing the method's responsibility to preserve the program readability.

Finally, we address the possible feature envy code smell, which may be created or discovered after applying the extract method refactoring, in step five. In this step, the best destination class for the newly extracted method is identified such that the class cohesion is improved after moving the extracted method. We intend to maximize the internal cohesion of classes and minimize their external coupling. The best refactoring option with its proper method name and destination class will be suggested to the developer. In summary, the following contributions are proposed in this paper:

1. Automatically identifying the long method instances in a given software project using both the source code metrics and dependency graph analysis.
2. Investigating the possibility of applying automated extract method refactoring in long methods and highlighting the refactoring opportunities according to the modularity of data and control dependency graph.
3. Generating and ranking meaningful and applicable extract method candidates while avoiding long-parameter list code smell.
4. Suggesting a proper name for the newly extracted method based on its operation and responsibility.
5. Finding the best destination class, improving the class cohesion for the newly extracted method to fix any feature envy smell after applying the extract method refactoring.

The implementation of the proposed approach is publicly available on the GitHub repository: https://github.com/mahnooshshd/extract_method. We have examined the proposed approach on five Java open-source projects with considerable sizes and in different domains. Evaluations are performed by measuring different software metrics and quality attributes, including changes in lines of code, class cohesion, and coupling. The improvement of the overall quality of software in terms of the above metrics confirms the applicability of the proposed approach. To assess the single responsibility, cohesiveness, functionality, and readability of extracted methods, we have asked several experienced developers to review and score the suggestions made by the proposed approach. Based on their feedback, the quality of code has improved by 21% in terms of the single responsibility principle (SRP) compared to the state-of-the-art extract method refactoring approaches (Hubert, 2019).

The rest of this paper is organized as follows. In Section 2, we describe and review the related work of this research. Section 3 presents the proposed approach. Section 4 evaluates the approach through different experiments. Section 5 discusses the limitations of the proposed approach and the performed experiments. Threats to validity are discussed in Section 6. Finally, in Section 7, the conclusion and future works are discussed.

2. Related work

This section reviews and discusses the related literature in the two areas of the long method code smell detection and the automatic extract method refactoring, as the most appropriate refactoring resolving long method code smell (Fowler and Beck, 2018). The long method, also referred to as the brain method (Sharma and Spinellis, 2018), has been recognized as one of the most prevalent smells in source code (Agnihotri and Chug, 2020). Conspicuous approaches to long method code smell detection are metrics-based and machine learning-based, mainly relying on source code metrics. In their famous book, Lanza and Marinescu (2006) have used a combination of four source code metrics, including lines of code, cyclomatic complexity, maximum nesting level of code blocks, and the number of access variables to detect long methods. The thresholds for each metric are computed at different ranges using statistical analysis on 45 Java projects. They manually selected specific thresholds for these metrics to identify long methods. Fontana et al. have used a similar approach to prioritize the detected smells at different levels of criticality. However, manual detection of metrics and their threshold is controversial and not accurate.

Machine learning models can be used to find the best threshold for each metric automatically. Lanza and Marinescu (2006), Fontana et al. have applied machine learning classification models to detect four code smells, including the long method and feature envy. They concluded that only a hundred training examples are needed to reach at least 95% accuracy, which has been criticized by other researchers due to the small number of instances (Di Nucci et al.). Indeed, machine learning approaches require a large and precise manually labeled dataset of code smells to produce reliable results (Caram et al., 2019). Some studies have shown that machine learning techniques are not always suitable for code smell detection due to the highly imbalanced nature of the smelly samples in available datasets (Pecorelli et al., 2020). In almost all metrics-based long method smell detection strategies, the line of code is considered the essential metric that reveals this smell (Lanza and Marinescu, 2006; Fontana et al.; Arcelli Fontana et al., 2016).

Palomba et al. have used information retrieval techniques to detect five code smells, including the long method and feature envy. They compute the textual similarity between different code segments as the probability of being smelly. However, they use blank lines to segment the method body for the long method, which only considers the consecutive lines as a separate functionality (Wang et al.). We observed that the dependencies of a method in its enclosing class and other classes are also good indicators of long method and feature envy smells besides source code metrics and used dependency analysis to improve the detection of smelly methods.

Despite numerous approaches to detect the long methods, few studies focus on automatic extract method refactoring to resolve this smell. Due to the inherent difficulty of the extract method refactoring, it is not used in search-based refactoring approaches (Mkaouer et al., 2016; Mansoor et al., 2017; Mohan and Greer, 2019). Maruyama (2001) has proposed an approach based on inter-program slicing, namely block-based slicing, to extract a complete slice of a given variable from a specific region in the method body. The programmer manually determines the variable whose slice should be extracted. As a result, only a limited number of refactoring opportunities can be found by this approach, which is equal to local variables existing in the selected method.

Tsantalis and Chatzigeorgiou (2011) have extended the Maruyama approach (Maruyama, 2001) by automatically calculating all program's block-based slices for variable or class fields in the assignment statements of the method. For class attributes whose state changes in the method body, they use object state slice, which is a subset of statements affecting the object's state in the program dependency graph. Their method comes up with a list of candidate refactoring suggested to the developer. The disadvantages of this method are that it is not fully automated, and extracted candidates might have many duplications since block-based slicing requires some statements to be repeated in both the initial and extracted methods to preserve the program behavior (Hubert, 2019).

Silva et al. have used a model representing the code as a block structure containing statements that follow a linear control flow to automate extract method refactoring. Every combination of blocks in this structure represents a feasible refactoring opportunity if three conditions of syntactical preservation, behavior preservation, and quality improvement are met. The candidate blocks are finally ranked by using their structural dependencies to the rest of the method. Generating and filtering many code blocks with this approach for large projects is very time-consuming.

Charalampidou et al. (2017) state that long methods often violate the single responsibility principle. They have proposed a tool, SRP-based extract method identification (SEMI), to find code fragments with high cohesion and employed them to suggest refactorings to improve the SRP rate of the original method. SEMI generates extract method candidates based on coherence between statements using three different criteria to measure cohesion. Statements within a method are considered more coherent if they access the same variables, call methods of the same object or call the same methods on different objects with the same type. The lack of cohesion in methods (LCOM) (Al Dallal, 2011) measurement and the size of the candidate are used to rank the generated candidates. Extracting lines of code solely based on the variable names can be problematic when variables of the same name are defined in different scopes. For instance, the names of the loops' iterators are usually the same. In this case, two loops that are unrelated to each other may be in the same category due to the same iterator name.

Hubert (2019) has proposed an approach that extracts methods based on quality analysis findings. The approach receives a long method as input and generates an exhaustive list of refactoring candidates by transferring the source code into a block structure called Statement Graph (Silva et al.). In this graph, each node or block represents one line (statement) in the method, and the nesting depth of the code is the same in the statement graph. The statement graph simply stores data edges between statements containing the same variables. The root node is the method definition that contains method body statements. For candidate generation, nodes of the statement graph are combined under several preconditions such that continuous candidates are built. The ranking algorithm is based on the number of lines of code, the number of parameters, the number of blank lines, and comments in the code. The blank lines and comments are used since programmers usually separate or mark the beginning and end of each responsibility to explain it with comments. Therefore, the pieces of code that are written between the comments are performing a responsibility. However, it makes the ranking approach very dependent on the style of writing the code. Developers may not adhere to such coding and commenting principles. If this style of commenting is not followed in projects, this approach will not be effective in ranking candidates appropriately.

Compared to related works, the proposed approach in this paper relies on intensive graph analysis to generate the best possible refactoring candidate and rank them such that overall program quality is maximized. The cohesion and coupling properties of source code entities at different levels, including statements and methods, are considered to achieve better refactoring results. Previous works do not address the problem of refactoring new code smells introduced by extract method refactoring or only identified after applying extract method refactoring (Tsantalis et al.). A single transformation may not suffice to remove all code smells that reveal poor code structures. Bibiano et al. have stated that 51% of batch refactorings ended up introducing new code smell instances. They have reported two important observations: First, developers correctly applied Extract Method on the "envious" code, but they did not apply Move Method precisely on the extracted (and "envious") methods. Second, many "envious" methods were created via the Extract Method refactoring, but not all of them were moved. Inspired by these observations, the proposed approach focuses on the synergy between consecutive applications of Extract Method and Move Method refactorings to resolve the Long Method and Feature Envy code smells in a fully automated fashion.

3. Methodology

In this section, we discuss the proposed approach to automatically find and fix long method instances in a software project. We first provide an overview of the proposed methodology based on graph representation and analysis of source code and then describe each step in detail.

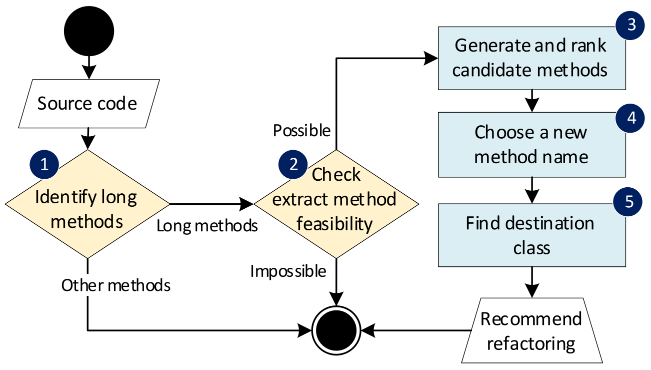


Fig. 1. The overall process of the long method identification and refactoring recommendation.

3.1. Overview

The best paradigm to automatically identify extract method opportunities is to recognize the root causes of long method occurrences. Long methods are disharmonies in code that appear when a method performs more than one operation or, in other words, violates the single responsibility principle (Fowler and Beck, 2018; Silva et al.; Al Dallal, 2011). In such a condition, the cohesion between the method's lines of code is decreased. Such methods are often highly coupled to other methods or fields outside its enclosing class, which results in a low cohesive class.

The graph representation of different relationships at different granularities can describe and quantify the concepts of responsibility, cohesion, and coupling in a natural way. Therefore, the proposed approach is built upon creating and analyzing graphs representing source code in high-level abstractions. The proposed methodology for long method identification and refactoring suggestion consists of five steps, shown in Fig. 1.

In the first step, the extract method opportunities are identified using the lines of code as a concrete proxy of code size and cohesion and coupling measures. In the second step, the feasibility of extract method refactoring is investigated formally by analyzing the modularity of data and control dependency graph, constructed from the lines of code for the long method's body. If there are meaningful and separated responsibilities within the long method's body, which can be extracted automatically, the method is passed to the next step. Otherwise, the proposed approach stops at this step to avoid suggesting any wrong refactoring solution. In the third step, the parts of codes that can be extracted from the main body of a given long method are recognized using graph analysis, and a list of candidate methods for the extract method refactoring is generated. The list of candidate methods is then ranked based on modularity metrics, and the best candidate method is selected. In step four, a proper name according to the input and output parameters is assigned to the extracted method. Finally, in step five, we determine the destination class based on the program dependencies, taking into account object-oriented design principles. Determining the destination class is very important to prevent new code smells due to the new calls created in the program. The last three steps are not supported by existing extract method refactoring tools. The detailed description of each step is discussed in the following subsections.

3.2. Long method detection

The refactoring opportunities for the extract method refactoring could be best identified by finding long method smells at each class of the program. We consider too many lines of code (LOC) and, at the same time, too much dependence on other classes and methods as the common characteristics of long methods (Fowler and Beck, 2018; Hubert, 2019). A combination of these two characteristics is used to assign a long method score to each method and determine the occurrence of the long method.

Eq. (1) assigns a *long method score*, $S_{LM}(M)$, to a given method M, where $LOC(M)$ denotes the lines of code of the method, M. $D(M)$ denotes the method dependency in terms of method cohesion and coupling described in Section 3.2.2. Two parameters are used to regulate the impact of the method's lines of code and dependency when scoring a method using Eq. (1). The parameter τ denotes the number of lines of code for which a method is not assumed as a long method. Therefore, the difference between this threshold and the LOC of a given method can be considered for detecting the long method smell. The parameter α controls the impact of dependency in scoring long methods.

$$S_{LM}(M) = e^{(LOC(M)-\tau)} + \alpha D\left(M\right) \tag{1}$$

In most cases, the high value of LOC leads to the long method smell (Hubert, 2019; Fontana et al.). For this reason, Eq. (1) has been designed such that the method's lines of code have a more significant impact on determining the smelliness of a method. According to Eq. (1), $S_{LM}(M)$ increases exponentially in terms of $LOC(M)$ and linearly in terms of $D(M)$.

3.2.1. Method's lines of code analysis

The appropriate number of lines of code for each method does not have a specific standard and varies between different Refs. Lanza and Marinescu (2006), Fontana et al., Arcelli Fontana et al. (2016). It also highly depends on the programming language. For example, implementing the same algorithm in a very high-level programming language such as Python requires fewer lines than implementing the same algorithm in a low-level language such as C or C++. Lippert and Roock state that methods larger than 30 lines of code are more error-prone than other methods (Martin Lippert, 2006). The maximum allowed length of a Java method is assumed to be around 40 (Steidl and Eder). We have experimented with different thresholds for Java methods' lines of code, τ, when detecting long method instances and find the best threshold empirically by comparing the accuracy of results.

3.2.2. Method call graph analysis

One can argue that a method with many lines of code should always be refactored to small methods. However, in some conditions, methods with a reasonable line of code may expose more than one functionality or responsibility. Therefore, lines of code cannot be used as the only metric to identify long method instances. In such cases, the number of dependencies in a method affects its responsibilities. Graph analysis is used to calculate the dependencies for each method in the program.

The principles of cohesion and coupling in object-oriented programs state that the number of internal dependencies of a software component, *e.g.*, a method, must be greater than its external dependencies. We construct two graphs of internal and external method calls for each method in a given program to measure the method's cohesion and coupling and compute the final dependency score $D(M)$. In both internal and external call graphs, the nodes are methods, and the edges between each pair of methods, M_1 and M_2, denote that method M_1 calls M_2 in its body. If there is no method call between M_1 and M_1, there is no edge in the corresponding dependency graph.

The *internal call graph*, $ICG(M)$, of a method M, in a class X, contains only the methods calls from M to other methods in the class X. In contrast, the *external call graph*, $ECG(M)$, of the method M, contains method calls to every method outside the class X.

Betweenness centrality (Newman, 2010) is used to analyze the dependencies of a method after constructing its dependency graphs. In graph theory, the betweenness centrality for each node is the number of shortest paths for every pair of nodes in a connected graph that passes through that node. Betweenness centrality $c_B^G(z)$ for a given node z of a graph G, is calculated using the following equation:

$$c_B^G(z) = \sum_{x,y,z \in V_G} \frac{\sigma(x,y,z)}{\sigma(x,y)} \quad (2)$$

In Eq. (2), V is the set of all nodes in the graph, G, $\sigma(x,y,z)$ is the number of shortest paths between nodes x and y that passes through node z, and $\sigma(x,y)$ is the number of *all* shortest paths between x and y. Eq. (2) can be used to measure the degree of centrality of each method in both the external and internal call graphs. The ratio of these two values determines the degree of dependence of a given method considering its context. After the calculation of the betweenness centrality for each node, the method dependency $D(M)$ for method M, is calculated by Eq. (3).

$$D(M) = \frac{c_B^{ECG(M)}(M) + 1}{c_B^{ICG(M)}(M) + 1} \quad (3)$$

where $c_B^{ICG(M)}(M)$ and $c_B^{ECG(M)}(M)$ are the betweenness centralities of the method M in the internal and external calls graphs of this method, respectively. The number one is added to each centrality value to avoid an undefined fraction. The parameter $D(M)$ is a relative measure that depends on different method calls within each method. The higher values of $D(M)$ denote that the method, M, struggles with different responsibilities. Therefore, it can be considered an orthogonal dimension next to the method's lines of code to detect the long methods.

3.3. Extract method refactoring feasibility

One of the challenges in refactoring the long method code smell is to spot the possibility of breaking the method into two smaller methods (Tsantalis and Chatzigeorgiou, 2011). If the method body has been written such that there are substantial dependencies between all statements, it is impossible to break the method into separate parts. We have examined the selected long methods to find how statements are related to each other before applying any refactoring to avoid any extra calculations and improve the efficiency of the refactoring process. Indeed, as shown in Fig. 1, steps 3, 4, and 5 are only performed if the automatic application of extract method refactoring is possible for a given long method.

3.3.1. Method dependency graph

A graph representation of the method's source code at the statements level called dependency graph is defined to investigate the possibility of extract method refactoring. In a method dependency graph, each line of code represents a node in the graph, and each edge of the graph indicates the dependency between two lines of code. There are two types of *data* and *control* dependencies between lines of code (Aho et al., 2006). Therefore, each edge in the method dependency graph is labeled with "1" for control dependency or "2" for data dependency.

Fig. 2 shows a simple code snippet with its corresponding dependency graph. A variable used in a line has a data dependence on the line that has recently changed the value of the same variable. If the execution of a line depends on the execution of another line, there is a control dependency between the two lines. The direction of each edge is from the dependent node to its parent node.

We start from the first line of the method after the method signature to build the dependency graph. Each line of code adds a node to the graph. We extract all the variables that are used or defined in the selected line. If the variable's value is modified or a variable is defined in this line, we save the line number along with the variable name as the line that has changed this variable for the last time. We also find the number of lines that have changed the variables used in this node, add them to a list, and create a data dependency edge from this node to the nodes in the list. If the selected line is within a conditional block, it would have a control dependence on the definition line of this block. Programming keywords including "if", "else", "switch", "case", "for", "while", and "catch" make the control dependencies when building the method dependency graph.

3.3.2. Modularity analysis of the method dependency graph

The modularity of the dependency graph, described in Section 3.3.1, is used to decide whether a method can be automatically separated into two methods or not. The modularity measure is vastly used for community detection in social networks (Newman and Girvan, 2004; Leicht and Newman, 2008; Fortunato, 2010). The concept of modularity in a subgraph determines the degree to which nodes within the subgraph are related to each other and related to nodes outside the subgraph (Newman and Girvan, 2004). The main idea of modularity is based on the difference between the probable number of edges that could exist in a graph and the actual edges that currently exist. The closer these two values are to each other, the more the structure of the graph would be uniform. In such a case, the number of edges inside the extracted subgraph is not significantly different from the outer edges. If the actual value is much greater than the probable value, the modularity value will increase. Conversely, the zero modularity expresses that there is no community to extract in the graph.

Subgraphs with a higher positive modularity value have a better modular structure. In other words, the number of inter-edges of these subgraphs is significantly different from the number of intra-edges. We have created the modularity matrix B, for the method dependency graph DG, based on the difference between the current edges in the dependency graph and the expected edges. We have calculated the modularity of a given method as follows. For dependency graph, DG with m edges, the probability that the end of an edge connects to a node i with the degree of k_i is equal to $p_i = k_i/2m$ since each edge has two connected ends. Therefore, the probability of connecting two nodes i and j will be $p_i p_j$. Multiplying this probability by the total number of edges (*i.e.*, $2m$), the expected value for the edge between i and j is obtained. Finally, each element of the modularity matrix B is calculated by the following equation (Fortunato, 2010):

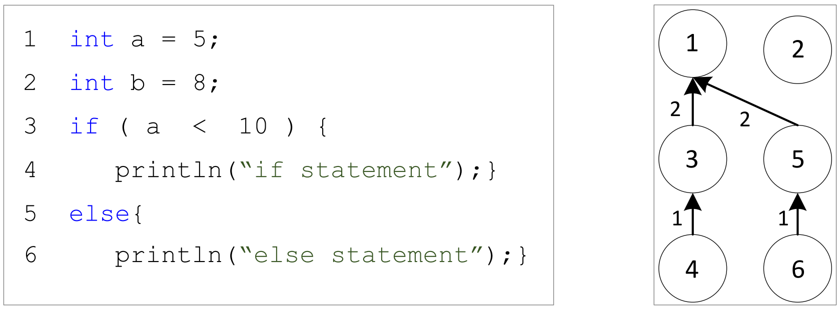
$$B_{ij} = A_{ij} - \frac{k_i k_j}{2m} \quad (4)$$

where A_{ij} is the adjacency matrix of the dependency graph DG. Eq. (5) defines the modularity for the directed graph as follows:

$$Q = \frac{1}{2m} \sum_{ij} (A_{ij} - \frac{k_i k_j}{2m}) \delta(C_i C_j) \quad (5)$$

In Eq. (5), C_i is the subgraph to which node i belongs. The parameter δ is equal to 1 if nodes i and j belong to the same subgraph. Otherwise, it is equal to -1. If the graph is divided into only two subgraphs, E, and F, Eq. (5) can be written as:

$$Q = \frac{1}{4m} \sum_{ij} (A_{ij} - \frac{k_i k_j}{2m})(s_i s_j + 1) \quad (6)$$



```
1  int a = 5;
2  int b = 8;
3  if ( a  <  10 ) {
4     println("if statement");}
5  else{
6     println("else statement");}
```

Fig. 2. An example of the dependency graph.

where s is the vector representing any partition of the graph in two clusters E and F. The parameter s_i is equal to $+1$ if i belongs to E, and -1 if i belongs to F. Substituting $A_{ij} - \frac{k_i k_j}{2m}$ in the above equation with B_{ij} according to Eq. (4), results in:

$$Q = \frac{1}{4m}\sum_{ij} B_{ij} s_i s_j = \frac{1}{4m} s^T B s \tag{7}$$

The vector s can be declared based on the eigenvector u_i of the modularity matrix B as $s = \sum_i a_i u_i$ where $a_i = u^T.s$ which result in the following equation:

$$Q = \frac{1}{4m}\sum_i a_i u_i^T B \sum_j a_j u_j = \frac{1}{4m}\sum_i (u_i^T \cdot s)^2 \beta_i \tag{8}$$

In Eq. (8), β_i is the corresponding eigenvalue of eigenvector u_i. If the modularity matrix B, does not have any positive eigenvalue, it concludes that the dependency graph cannot be split into smaller subgraphs. In other words, the dependencies between the lines of code in the method are high enough such that we could not break the method and move some lines to another method. In contrast, if the dependency graph contains modular subgraphs, those subgraphs can be extracted from the code. Therefore, for a given long method M the possibility of extract method refactoring is determined by calculating eigenvalues of the modularity matrix before applying the candidate generation algorithm.

3.4. Refactorings candidate generation and ranking

The dependency graph defined in Section 3.3.1 is processed to identify the subsets of statements in the body of a long method as extract method candidates. The extracted method must expose a separate functionality with negligible dependences on the long method to be useful and meaningful (Tsantalis and Chatzigeorgiou, 2011; Charalampidou et al., 2017). To this aim, lines of codes that are mostly related to each other and are less related to the other parts of the program are extracted from a given long method.

We deal with two challenges when generating and ranking extract method refactoring candidates. The first challenge is that the application of the extract method refactoring should not change the program's behavior (Fowler and Beck, 2018). It also should not cause a compilation error in the program. In other words, the refactored code must be utterly transparent from the viewpoint of the program end-users (Mens and Tourwe, 2004). The second challenge is to avoid introducing other code smell after performing an extract method refactoring on a given long method. The main possible messy code side effect of an improper extract method refactoring is to increase the method dependencies, leading to the long parameter list code smell (Fowler and Beck, 2018). The other highly possible code smell side effect is feature envy which may appear in the extracted method after application of extract method refactoring (Lozano et al.).

We have focused on addressing the aforementioned challenges in the proposed refactoring process. To address the first challenge, in addition to preserving data and control dependency when generating a refactoring candidate, a set of refactoring rules and postconditions (*i.e.*, conditions that are checked after applying the refactoring operation) are proposed in Section 3.4.3. To deal with the second challenge, in addition to controlling the dependencies between the original method and extracted method, the possibility of move method refactoring is considered after selecting the best candidate. Indeed, as discussed in Section 3.6, the best destination class for the extracted method is identified to resolve the possible feature envy code smell.

3.4.1. Candidate generation algorithm

The proposed candidate generation algorithm uses graph analysis on the adjacency matrix, A, of a method dependency graph to produce a list of extract method refactoring candidates. Fig. 3 shows the candidate generation algorithm. It receives the source code SC, and adjacency matrix A of a long method as input and returns an extraction list consisting of extracted method candidates as output. Each element within the adjacency matrix A contains a value of 0, 1, or 2, which indicate the absence of dependency, control dependency, or data dependency, respectively. Fig. 4 shows the adjacency matrix of the dependency graph in Fig. 2.

The first step in extracting statements as a candidate method is to determine the *starting points*. The starting point is the first line that can be separated from a method. To avoid any compilation error, we start the extraction from lines that do not have inseparable control dependence on any line. Statements with inseparable control dependence are the multiple parts conditional statements, including "if-else", "switch-case", and "try-catch" statements. Other statements can be considered as starting points. The algorithm finds all starting points and adds them to the starting point list. Each starting point will make a separate candidate.

In the next step, for each starting point, all the nodes that have a control dependency to it are added to a corresponding extraction list. The reason is to preserve the behavior of the program while avoiding possible compilation errors. For instance, if a line with an "if" condition is added to the extraction list, its corresponding "then" block must also be added to the list. This operation is performed for each node in the extraction list. The last node added to the extraction list is selected as the current node, and its corresponding column in the adjacency matrix is detected. Then, all rows with the value of 1 in this column are added to the extraction list that the current node belongs to.

For nodes with a data dependency to the current node, the value of the clustering coefficient help decides to add them to the extraction list of the current node. The clustering coefficient of each node indicates the degree to which the neighbors of that node are related (Newman, 2010). If we select a node with a

```
Algorithm: ExtractCandidateMethods
Inputs: method_source_code SC, adjacency_matrix A
Outputs: extract_list
   starting_points = []
   foreach row in A {
             if SC[row] does not have inseparable control dependence {
                         starting_points.append(row)
        }
  }
   foreach point in starting_points {
             extract_list = []
             extract_list.append(point)
             while not termination conditions {
                         clustering_coefficients = []
                         current_node = last(extract_list)
                         for row in A {
                                if row has control dependency to current_node {
                                        extract_list.append(row)
                        }
                                else if row has data dependency to current_node {
                                        C = clustering_coefficient(row)
                                        clustering_coefficients.append(C)
                        }
                   }
                         exctract_list.append(Max(clustering_coefficients))
         }
  }
```

Fig. 3. The extract candidate methods algorithm.

larger clustering factor, it allows us to extract the more inter-dependent parts of the code. In addition, we obtain more lines of codes that make the extracted candidate as meaningful as possible to be a new method. Eq. (9) is used to calculate the clustering coefficient of node i as below:

$$CC_i = \frac{1}{(k_i)(k_i-1)/2}\sum_{j,k} a_{ij}a_{ik}a_{jk} \tag{9}$$

where CC_i, is the clustering coefficient of node i and k_i represents the maximum number of edges possible for each node. The parameter a_{ij} is the element in row i and column j of the adjacency matrix A.

The clustering coefficient value is calculated for all nodes with a data dependency to the selected node, and the node with the highest value is added to the extraction list. The process of adding lines to the extraction list continues until at least one of the following conditions is met:

1. The number of input parameters for the candidate method exceeds four variables (Bloch, 2017). The reason is that a large number of input parameters introduce the long parameter list code smell and increase the dependency between the new method and the long method (Fowler and Beck, 2018; Silva et al.).
2. The number of extracted lines exceeds two-thirds of the long method lines. Extract method refactoring aims to reduce the number of lines in a long method and ultimately increase the quality of the code. The extracted method must not introduce a new long method or duplicated method.

$$\begin{array}{c} 1\\2\\3\\4\\5\\6 \end{array}\begin{bmatrix} 0&0&0&0&0&0\\ 0&0&0&0&0&0\\ 2&0&0&0&0&0\\ 0&0&1&0&0&0\\ 2&0&0&0&0&0\\ 0&0&0&0&1&0 \end{bmatrix}$$

Fig. 4. Adjacency matrix for the dependency graph in Fig. 2.

The candidate extraction algorithm returns a list of lines of code that can be extracted from a method. These extracted lines are possible options to refactor the smelly method.

3.4.2. Ranking candidate refactorings

The candidate extraction algorithm, discussed in Section 3.4.1, results in many possible refactorings which might not have the same usefulness from the programmer's viewpoint. Some automated extract method refactoring approaches leave the task of appropriate refactoring selection to the programmer (Maruyama, 2001; Tsantalis and Chatzigeorgiou, 2011), which avoids creating a fully automated refactoring pipeline. We propose a ranking strategy to rank the candidate methods based on their usefulness considering the object-oriented principles. Then, select the best candidate solution to provide a fully automated extract method refactoring.

The external dependencies of the methods in a high-quality object-oriented design are as small as possible. Moreover, each method must have a single and independent responsibility (Martin, 2009). If the lines of codes within a method body are highly cohesive and dependent on each other, they all perform a unique task or implement a single responsibility. In summary, a low coupled and highly cohesive code snippet is a suitable candidate for the extract method. The modularity defined in Eq. (5) best describes this circumstance for each extract method candidate. For each extraction candidate generated by the algorithm in Fig. 3, the lines of codes in the extraction part and remaining part constitute two subgraphs of the method's body dependency graph. The modularity is calculated considering these subgraphs for the corresponding candidate. The candidate extraction list is then arranged based on the modularity values. The candidate which results in the highest modularity after the extract method refactoring is selected as the most suitable candidate in the list. The modularity-based extract method candidate selection conducts the refactoring process to establish the single responsibility principle and improve the overall modularity of a component. It is also easy to calculate and time-efficient when the number of possible refactorings is too much or the size of the dependency graph is large.

3.4.3. Refactoring rules and conditions

As discussed earlier in Section 3.4, moving lines of code from a method to another one can cause runtime and even compile-time errors. We formally check the feasibility of automated refactoring as a separate precondition in Section 3.3. However, the moving statements must be precisely checked to prevent any errors and preserve the program behavior after the refactoring candidate is selected. To this aim, the following rules are considered for the selected code snippet to extract from a method's body:

1. If a value is returned in a new method, any call to this method from the caller method must be assigned to a corresponding variable in a caller method.
2. If the line containing the return value is in the list of extracted lines, the extracted method must be called at the return statement of the caller method.

In addition to the above rules, we apply a postcondition specific to the Java programming language. In Java, each method can return only one value. Therefore, the data dependency of other parts of the code on the extracted method should only contain one value. Otherwise, it is impossible to perform the refactoring process automatically. This postcondition is highly specific to the syntax of the programming language. For instance, in Python, a method can return any number of variables within a return statement.

3.5. Selecting a name for the new method

The readability of code is important when extract method refactoring is applied since it introduced a new method that requires an identifier. The name of a method must be such that it appropriately reflects the operation and responsibility of the method (Martin, 2009). We automatically assign a proper name to an extracted method based on the available information, mainly the names of variable types. For instance, if the new method returns a value and its type is numeric, the method name can be created as "calculate_{variableName}".

Fig. 5 shows the method name creation ruleset as a decision tree. For each status that may exist in the method, its name is selected based on the operation and the associated variable. At the root of the tree, a decision is made according to the method's return type. In the subsequent decision level, if the method is not void and returns a value, three different templates are used based on the type of return value. If a method is void and sets a variable, the method's name is prefixed with "set_global_". Otherwise, the method may print a variable or perform some operation on its input parameters. For the earlier case, the name is prefixed with "print_", and for the latter case, the method's name is prefixed with "handle_".

3.6. Identification of the destination class

A novel aspect of the proposed refactoring methodology is to prevent the introduction of new code smells after applying the extract method refactoring. This problem is observed in previous automated extract method approaches (Tsantalis and Chatzigeorgiou, 2011; Charalampidou et al., 2017; Bibiano et al.). In Section 3.4.1, we avoid the long-parameters list code smell (Fowler and Beck, 2018) by applying a simple precondition. We also take into account the modularity, which may lead to smells such as low cohesion and high coupling between methods in a class. This section investigates the most proper class for the extracted method to avoid feature envy code smell (Fowler and Beck, 2018). The new method must be in a class with the highest internal dependency or cohesion and the lowest external dependency or coupling after the changes are applied. Indeed, the target class X, is chosen such that the cohesion/ coupling ratio, so-called the *dependency ratio*, is maximized for this class after moving the extracted method.

Eq. (10) expresses the formula for the dependency ratio calculation. For each class X, which has at least one dependency on the extracted method, this ratio is calculated assuming the extracted method has been placed in class X. In the end, a class with the maximum dependency ratio is selected as the destination class for the extracted method.

$$DependencyRatio\,(X) = \frac{Cohesion\,(X)}{Coupling\,(X)} \tag{10}$$

Cohesion and coupling are calculated by creating the corresponding graphs and applying appropriate graph analysis. Fig. 6 shows the flowchart of the destination class selection process. In the first step, all classes associated with the new method are extracted and added to a list according to the project class diagram automatically extracted from the source code. These connections can be a method call or using a field of a class. In addition to associated classes, the current class of extracted methods is also added to the related classes list. The reason is that the new method is called in the class where the long method is located. This makes at least one dependency on the current class. At the end of this step, we have a list of candidate classes where the new method can be placed.

In the next step, the cohesion and coupling of the class X in the candidate classes list are calculated assuming that the new

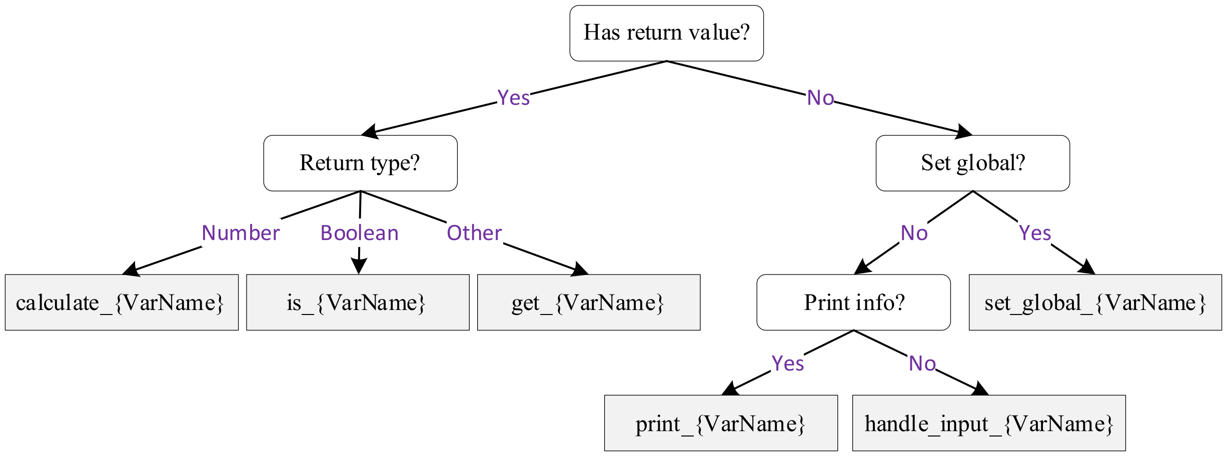


Fig. 5. Method name suggestion ruleset.

method is moved to class X. We build a graph named *cohesion graph* to calculate the *cohesion*(X). Assuming that each class contains some methods and attributes, the cohesion graph of class X is a graph whose nodes are both the methods and attributes of class X. There are three different types of relationships between nodes in the cohesion graph as follows:

1. *Attribute–attribute relationship:* If two class attributes are used by the same method, there is an edge between the two attributes in the corresponding cohesion graph.
2. *Method–method relationship*: If two methods call each other or share one or more common attributes, there is an edge between their two corresponding nodes in the cohesion graph.
3. *Method–attribute relationship*: If a method uses an attribute directly, there is an edge between their two corresponding nodes in the cohesion graph.

As discussed in Section 3.4.1, the clustering coefficient indicates the extent to which a group of neighbors is related to each other. The clustering coefficient analysis is used to measure the cohesion of the class X. The clustering coefficient for each node is calculated by Eq. (9), and the average clustering coefficient of all nodes in the cohesion graph of class X, determines the overall cohesion of this class:

$$cohesion(X) = \frac{1}{n}\sum_{\mathrm{i=0}}^{\mathrm{n}} cc_i \tag{11}$$

In Eq. (4), n is the number of nodes in the cohesion graph of class X, and cc_i is the clustering coefficient of the node i in this graph. If the average clustering coefficient in the cohesion graph is high, the class components (*i.e.*, its method and attributes) have enough dependencies in the same class (Gu et al., 2017). The class diagram is converted to the *coupling graph* to calculate the coupling of each class in the candidate classes list. The nodes in the coupling graph represent the project classes, and the edges indicate the dependency between the classes. The dependencies include method-call, attribute usage, and inheritance relation. This graph will be weighted and directionless. Edge weight indicates the number of times classes use each other. The directions of the edges do not matter in coupling analysis, and only the dependencies between classes are essential. As discussed in Section 3.2.2, the betweenness centrality of each node in the dependency graph is suitable for analyzing the degree of communication and dependencies between nodes (Savić et al., 2017). We calculate the coupling of the class X by measuring

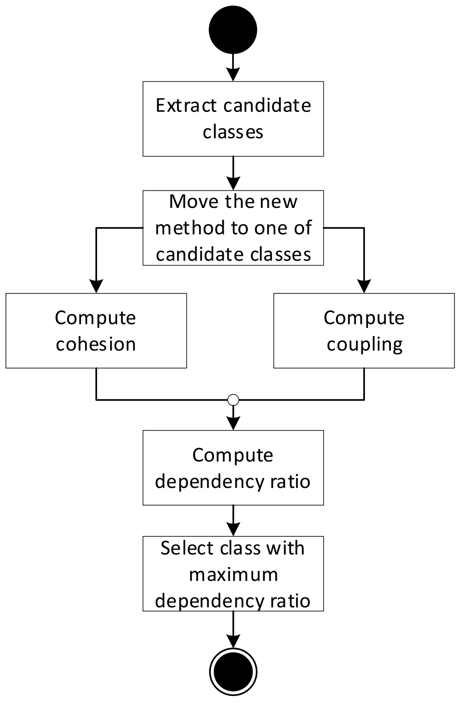

Extract candidate classes
Move the new method to one of candidate classes
Compute cohesion
Compute coupling
Compute dependency ratio
Select class with maximum dependency ratio

Fig. 6. The destination class selection.

the betweenness centrality $c_B^{CG}(X)$ of the class X in the coupling graph CG:

$$coupling\ (X) = c_B^{CG}(X) \tag{12}$$

In Eq. (12), $c_B^{CG}(X)$ is the betweenness centrality of the class X calculated by Eq. (2). After calculating the cohesion and coupling for each candidate class X the list of candidate classes is sorted by the value of their dependency ratio. Finally, the class with the highest value is selected as the destination class.

3.7. Implementation details

The proposed approach has been supported by an open-source tool, Extract Method, implemented in Python 3. The Extract Method tool receives the project's source code directory and project JAR file as input and produces a JSON file containing the required information of all applicable refactorings as output. Each record of the JSON file includes the long method name and

```
1   public void caseALogicalExpressionTail(ALogicalExpressionTail node) {
2       Object left = val;
3       val = null;
4       inALogicalExpressionTail(node);
5       if (node.getLogicalOperator() != null) {
6           node.getLogicalOperator().apply(this);
7       }
8       if (node.getRelationalExpression() != null) {
9           node.getRelationalExpression().apply(this);}
10      Object op = node.getLogicalOperator();
11      Object right = val;
12      val = null;
13      if (op != null) {
14          if (op instanceof AAndLogicalOperator) {
15              if (left != null && left instanceof Boolean
16                      && !((Boolean) left)) {
17                  val = false;
18              } else if (right != null && right instanceof Boolean
19                      && !((Boolean) right)) {
20                  val = false;
21              } else {
22                  val = asBoolean(left, node) && asBoolean(right, node);
23              }
24          } else if (op instanceof AImpliesLogicalOperator) {
25              val = !asBoolean(left, node) || asBoolean(right, node);
26          } else if (op instanceof AOrLogicalOperator) {
27              if (left != null && left instanceof Boolean
28                      && ((Boolean) left)) {
29                  val = true;
30              } else if (right != null && right instanceof Boolean
31                      && ((Boolean) right)) {
32                  val = true;
33              } else {
34                  val = asBoolean(left, node) || asBoolean(right, node);
35              }
36          } else if (op instanceof AXorLogicalOperator) {
37              val = !asBoolean(left, node) ^ asBoolean(right, node);
38          } else {
39              error(node);
40          }
41      } else {
42          error(node);
43      }
44      outALogicalExpressionTail(node);
45  }
```

Fig. 7. Extract Method opportunity identified by the proposed approach in `org.argouml.profile.internal.ocl. EvaluateExpression` class of the ArgoUML project.

the enclosing class name, along with the extracted method lines, instructions, name, and destination class. The full implementation of the proposed extract method tool is available on the GitHub repository: https://github.com/mahnooshshd/extract_method.

The JavaCallGraph library (Gousios et al., 2018) has been used to extract the program call graph. This library generates a textual file containing a table of caller–callee relationships, along with several static method calls. Each line in this file contains the caller method identifier, callee method identifier, and corresponding classes. The JavaCallGraph library is integrated with the proposed tool, and the internal and external method call graphs are extracted from the program call graph as subgraphs for each method.

The JavaParser library (Anon, 2019) has been applied to extract code lines and dependencies between code lines within the methods' bodies. This library generates and traverses the Java source code parse tree while looking for the interesting patterns defined by the developer. The patterns used in the proposed tool are the data and control dependencies of statements in the method body and the relationships between classes in a project. Graph analysis and the corresponding calculations have been implemented on top of NetworkX (NetworkX), the well-known graph library in Python.

3.8. A running example

To demonstrate the application of the proposed approach on real software projects, we have considered an example for a long method code smell and have identified and refactored it using the implemented tool. Fig. 7 shows the `caseALogicalExpressionTail` method with 45 lines of code in class `org.argouml.profile.internal.ocl. EvaluateExpression` of the ArgoUML project. This method is detected as a long method by the proposed tool. The identified lines of code to be extracted as the new method have been highlighted in the figure. From line 14 to line 40 is considered to be extracted as a new method. It is observed that these lines of code are used to set the value of the field in the `EvaluateExpression` class named `val`, which is a separate responsibility compared to the rest of the method's body.

The recommended name for the extracted method is `set_global_val` which is highly relevant to the method's operation. The selected destination class for the new method is the same class that encloses the original method according to the dependency ratio. Indeed, the extracted method operates on the fields and methods of the current class, specifically the `val` variable.

Table 1
Projects used in the performed experiments.

Project	Version	Application domain	Classes	Methods	LOC
JEdit (Pestov, 2021)	4.5.1	Free software text editor with syntax highlighting.	1,141	6,663	107,212
FreeMind (Müller, 2021)	0.9.0	Mind mapping application to edit a hierarchical set of ideas.	696	4,583	40,933
ArgoUML (Robbins, 2021)	0.34	UML diagramming application.	2,539	17,485	249,538
JFreeChart (Gilbert, 2020)	1.0.14	A Java chart library.	619	8,630	222,814
jVLT (jVLT - a vocabulary learning tool, 2021)	1.3.2	A vocabulary learning tool.	420	2,036	29,161

4. Evaluation

In this section, we discuss the evaluation of the proposed methodology. To this aim, we have examined each of the steps of the proposed methodology separately on five different software systems. The first experiment considers detecting the long method code smell instances as the extract method refactoring opportunities. The second experiment deals with identifying and ranking the extract method candidates for the corresponding long methods and their impact on different quality attributes, mainly the single responsibility principle. We have also evaluated the method name relevancy of automatically recommended names for the extracted methods. The third experiment compares the results of the proposed extracted method refactoring approach with one of the state-of-the-art approaches in this area (Hubert, 2019). Finally, the last experiment is dedicated to designating the suitable destination class for the new method. First, we explain the experimental setup and case studies.

4.1. Experimental setup

All experiments were performed on an Ubuntu Linux machine with an Intel® Core i7™ processor and 16 Gigabytes of RAM. To ensure the reliability and correctness of the results, we have repeated each experiment 5 times with the same setup and reported the average value.

4.1.1. Benchmark projects

We have used five Java open-source projects as the benchmark in the experiments. These projects have been used in previous research on extract method refactoring, which allows us to compare the results with other existing approaches. Table 1 shows the details of the benchmark projects. All projects contain at least 400 Java classes, 2,000 methods, and 29,000 lines of code (LOC), revealing that they are complex projects of large sizes. In total, we have evaluated the proposed tool on 39,397 Java methods.

4.2. Long method detection accuracy

We have applied the proposed long method smell detection strategy to find long method instances for each project in the benchmark. The results are compared with results in the code smell dataset recently proposed by Di Nucci et al. to evaluate the performance of the smell detection approach.

There are two mandatory parameters τ and α in Eq. (1), used to score methods. For τ or the minimum number of lines of code, we have experimented with two values of 40 and 60. We have also set the value of α or the coefficient of the method's dependency to 1. To determine the best threshold for which a given method should be detected as the long method, we have evaluated five values of 0.5, 1.0, 2.0, 3.0, and 4.0 to compare with $S_{LM}(M)$ calculated by Eq. (1). In total, ten different experiments were designed and performed. We have measured the recall metric for each configuration of thresholds, which is defined as the ratio of detected long methods to all existing long methods.

Tables 2 to 6 show the results of smell detection respectively for JEdit, FreeMind, ArgoUML, JFreeChart, and jVLT projects. The maximum recall for each project is bolded in the tables. The most accurate result of the long method smell detection belongs to the jVLT project, and the lowest recall belongs to the FreeMind project. In general, it is observed that if a smaller number is chosen for the $S_{LM}(M)$ threshold, the recall of this method will be higher. Nevertheless, more methods will be examined, resulting in more execution times. Indeed, choosing the appropriate threshold value for $S_{LM}(M)$ is very important. The best threshold obtained for $S_{LM}(M)$ by the performed experiments is 0.5. The results also show that the lines of code threshold, τ, do not considerably influence the recall of smell detection. For almost all experiments, the recall value for lines of code of 40 is the same for lines of code of 60. On average, the proposed long method detection strategy has a recall of 84.23%, which is acceptable to be used in practice compared to more complex machine learning-based approaches.

4.3. Extract method refactoring results

We have evaluated the impact of the extract method refactoring candidate identification on different aspects of code quality, including the single responsibility principle, cohesion, functionality of extracted methods, and changes in the number of lines of code in the refactored methods. We have also evaluated the relevancy of the method name suggested by the proposed approach for the extracted method.

4.3.1. Impacts on code quality

An important metric we wanted to evaluate was whether the method is split correctly and each of the new methods performs exactly a single responsibility or not. However, an exact parametric criterion cannot be defined to determine the code quality after refactoring long method code smell. In other words, it is not possible to evaluate all quality attributes automatically. Therefore, the best way to evaluate the meaningfulness, usefulness, and functionality of recommended refactorings is to ask for experts' opinions.

For our purposes in this experiment, we have asked four different expert developers, each with at least five years of professional programming experience, to carefully review the extract method refactoring suggestions. The developers were asked to assign a score between 1 and 10 for each suggested refactoring considering the following four quality aspects:

1. *Single responsibility at method-level:* How does the suggested refactoring adhere to the single responsibility principle (SRP) for both the extracted and original methods? Can the presented refactoring separate the responsibilities correctly?
2. *Code cohesion:* Is there high cohesiveness between the statements in both the extracted and original methods?
3. *Functionality:* Does the application of suggested refactoring preserve the behavior of the program?
4. *Method name relevancy:* Is a proper name chosen for the new method?

The scores are integer values ranging from 1 to 10, where 1 and 10 denote the lowest and highest approval rates. Manual

Table 2
Detected long methods in JEdit.

$S_{LM}(M)$ threshold	Line of code = 40		Line of code = 60	
	Number of detected long methods	Recall (%)	Number of detected long methods	Recall (%)
0.5	127	**90.62**	127	**90.62**
1.0	101	81.25	101	81.25
2.0	82	71.87	82	71.87
3.0	80	71.87	80	71.87
4.0	77	71.87	77	71.87

Table 3
Detected long methods in FreeMind.

$S_{LM}(M)$ threshold	Line of code = 40		Line of code = 60	
	Number of detected long methods	Recall (%)	Number of detected long methods	Recall (%)
0.5	69	**69.23**	69	**69.23**
1.0	55	61.53	55	61.53
2.0	32	53.84	32	53.84
3.0	32	53.84	32	53.84
4.0	30	53.84	28	46.15

Table 4
Detected long methods in ArgoUML.

$S_{LM}(M)$ threshold	Line of code = 40		Line of code = 60	
	Number of detected long methods	Recall (%)	Number of detected long methods	Recall (%)
0.5	95	**76.92**	95	**76.92**
1.0	88	61.53	88	61.53
2.0	63	53.84	63	53.84
3.0	59	53.84	59	53.84
4.0	58	53.84	51	46.15

Table 5
Detected long methods in JFreeChart.

$S_{LM}(M)$ threshold	Line of code = 40		Line of code = 60	
	Number of detected long methods	Recall (%)	Number of detected long methods	Recall (%)
0.5	103	**85.24**	103	**85.24**
1.0	101	**85.24**	101	**85.24**
2.0	74	73.77	71	65.57
3.0	74	73.77	69	65.57
4.0	70	68.65	69	65.57

Table 6
Detected long methods in jVLT.

$S_{LM}(M)$ threshold	Line of code = 40		Line of code = 60	
	Number of detected long methods	Recall (%)	Number of detected long methods	Recall (%)
0.5	217	**96.15**	217	**96.15**
1.0	193	88.46	193	88.46
2.0	140	84.61	140	84.61
3.0	132	84.61	130	84.61
4.0	112	84.61	112	84.61

evaluation by experts requires considerable effort and time. Due to the high number of refactored methods in some projects, *e.g.*, JEdit with more than 80 cases, we have randomly selected 20 refactored methods to be evaluated by an expert. For projects with less than 20 refactored methods, *e.g.*, JFreeChart with 16 cases, all refactored methods were evaluated by the experts.

Tables 7 to 11 show the average scores obtained for each project considering all quality aspects. The proposed approach has achieved a mean score of 7.1 for all quality attributes in all projects, indicating its appropriateness based on the experts' opinions. The maximum score given by the experts belongs to the jVLT project, and the minimum score belongs to the ArgoUML. However, a very low variance between scores indicates that the performance of our proposed extract method refactoring approach is not significantly affected by a specific project or evaluator.

The maximum average score between quality aspects is functionality which shows that the proposed approach preserves the program's functionality as much as possible. Concerning functionality, we have also performed regression testing according to the available tests in the benchmark projects. To this aim, we have applied 10 recommended refactorings for the benchmark projects and executed the unit tests for them. Unfortunately, no unit tests were associated with the JEdit and jVLT projects. We have measured the pass rates of the executed unit tests for the remaining three projects. The percentage of the passed tests are 100%, 80%, and 50% for the FreeMind, ArgoUML, and JFreechart, respectively. On overage, 76.67% of all tests are passed after applying the extract method refactoring. We have also observed that only a minor patch in the refactored methods is required to pass most of the failed tests.

The minimum average score between quality attributes belongs to the method name relevancy. The reason is that automatic recommendation of method names is inherently a difficult task (Alon et al., 2019). However, we have observed that the

Table 7
The average score of experts' opinions about extract method refactoring in JEdit.

Expert	Single responsibility	Code cohesion	Functionality	Method name relevancy	Average
expert #1	6.2	7	8.5	5	**6.8**
expert #2	5	6.5	9.2	4	**6.2**
expert #3	8	7.3	9	4.5	**7.2**
expert #4	7.5	7	7.5	6	**7.0**
Average	**6.7**	**7.0**	**8.6**	**4.9**	**6.8**

Table 8
The average score of experts' opinions about extract method refactoring in FreeMind.

Expert	Single responsibility	Code cohesion	Functionality	Method name relevancy	Average
expert #1	7	8	8.5	5	**7.1**
expert #2	6	7.5	9	5.5	**7.0**
expert #3	8	8	7.5	6	**7.4**
expert #4	7.5	6.2	8	6.5	**7.0**
Average	**7.1**	**7.4**	**8.3**	**5.8**	**7.1**

Table 9
The average score of experts' opinions about extract method refactoring in ArgoUML.

Expert	Single responsibility	Code cohesion	Functionality	Method name relevancy	Average
expert #1	6	6	8	5	**6.2**
expert #2	5	6.5	7	5.5	**6.0**
expert #3	7.2	7	8.2	6	**7.1**
expert #4	7.5	7	8	6.5	**7.3**
Average	**6.4**	**6.6**	**7.8**	**5.8**	**6.7**

Table 10
The average score of experts' opinions about extract method refactoring in JFreeChart.

Expert	Single responsibility	Code cohesion	Functionality	Method name relevancy	Average
expert #1	7.2	7.7	8	6	**7.2**
expert #2	8	7.5	8	6	**7.4**
expert #3	7.7	7.8	7.8	6	**7.3**
expert #4	7	7.7	7.8	5.5	**7.0**
Average	**7.5**	**7.7**	**7.9**	**5.9**	**7.2**

Table 11
The average score of experts' opinions about extract method refactoring in jVLT.

Expert	Single responsibility	Code cohesion	Functionality	Method name relevancy	Average
expert #1	6.9	7.3	8.4	5.5	**7.0**
expert #2	7.9	7.5	8	6	**7.4**
expert #3	7.8	7.7	8	6.2	**7.4**
expert #4	7	7.7	8	6	**7.2**
Average	**7.4**	**7.6**	**8.1**	**5.9**	**7.3**

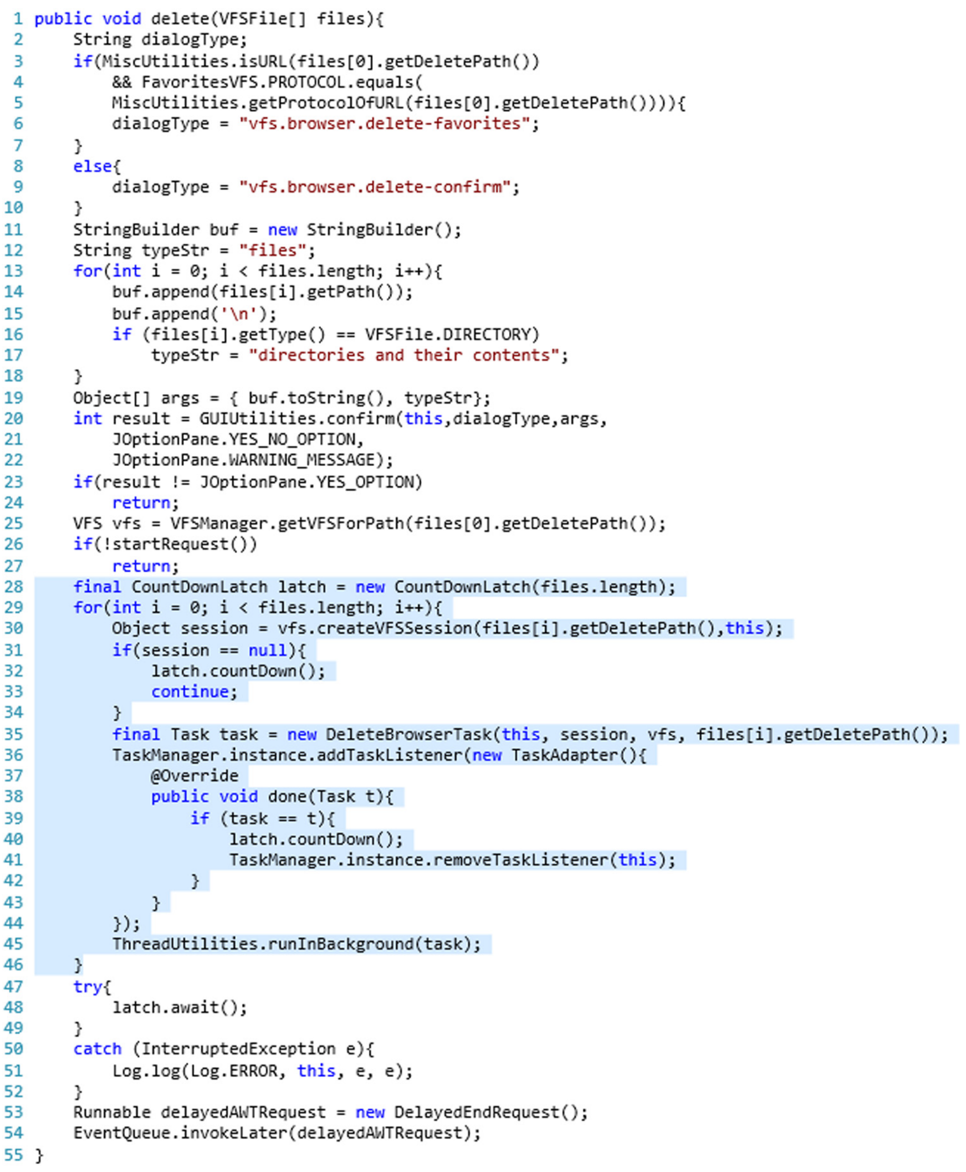
recommended name is often acceptable. For instance, in Fig. 7, the name `set_global_val` is highly relevant to the extracted method's operation since it actually set the value of the global variable `val`. As future work, the proposed rule-based approaches can be combined with more advanced method name recommendation techniques (Alon et al., 2019; Jiang et al.; Zaitsev et al.) to improve the methods' name relevancy.

Fig. 8 shows the method `delete` in `org.gjt.sp.jedit.browser. VFSBrowser` class of the JEdit project. The best-extracted candidate method by the proposed approach has been highlighted in the figure (lines 28 to 46) with a recommended name of "hanle_files" and destination class of "`org.gjt.sp.util. TaskManager`". The responsibility of the extracted method is the creation of `DeleteBrowserTask` instances and executing them in the background, which is performed in the `delete` method before refactoring. By applying the extract method, this responsibility is separated from the responsibility of configuring the delete file types.

We have also compared the proposed methodology with the most recent work on extract method refactoring by Hubert (2019) using manual evaluation by experts. As discussed in Section 2, in Hubert's work, a long method is received as input. Then, a piece of code that can be extracted from that method is returned to the user as output. To compare the proposed approach with the work of Hubert (2019), we have fed the methods identified by the proposed long method detector as input to Hubert's tool and created a refactored version of the benchmark projects. Then, we asked the experts to check the code quality after refactoring in both Hubert's approach and the proposed approach.

Four software developer experts with at least five years of programming experience participated in this experiment independently. They evaluated the two approaches concerning the three quality aspects, namely single responsibility, functionality, and code cohesion for the refactored methods. The same selected methods in the previous experiment are used to refactor with Hubert's tool (Hubert, 2019). In the cases where Hubert's tool could not apply any extract method refactorings, experts did not evaluate that method for Hubert's approach. The quality aspects evaluated by the expert are the same aspects defined and used in the previous experiment with the same scoring scheme.

The average experts' evaluation scores for each project are shown in Figs. 9 to 13. It is observed that the proposed approach outperforms the Hubert approach (Hubert, 2019) in the case of single responsibility and code cohesion. On average, the single responsibility score for the proposed approach is 21% higher than the single responsibility score for the Hubert approach (Hubert,



```
public void delete(VFSFile[] files){
    String dialogType;
    if(MiscUtilities.isURL(files[0].getDeletePath())
        && FavoritesVFS.PROTOCOL.equals(
        MiscUtilities.getProtocolOfURL(files[0].getDeletePath()))){
        dialogType = "vfs.browser.delete-favorites";
    }
    else{
        dialogType = "vfs.browser.delete-confirm";
    }
    StringBuilder buf = new StringBuilder();
    String typeStr = "files";
    for(int i = 0; i < files.length; i++){
        buf.append(files[i].getPath());
        buf.append('\n');
        if (files[i].getType() == VFSFile.DIRECTORY)
            typeStr = "directories and their contents";
    }
    Object[] args = { buf.toString(), typeStr};
    int result = GUIUtilities.confirm(this,dialogType,args,
        JOptionPane.YES_NO_OPTION,
        JOptionPane.WARNING_MESSAGE);
    if(result != JOptionPane.YES_OPTION)
        return;
    VFS vfs = VFSManager.getVFSForPath(files[0].getDeletePath());
    if(!startRequest())
        return;
    final CountDownLatch latch = new CountDownLatch(files.length);
    for(int i = 0; i < files.length; i++){
        Object session = vfs.createVFSSession(files[i].getDeletePath(),this);
        if(session == null){
            latch.countDown();
            continue;
        }
        final Task task = new DeleteBrowserTask(this, session, vfs, files[i].getDeletePath());
        TaskManager.instance.addTaskListener(new TaskAdapter(){
            @Override
            public void done(Task t){
                if (task == t){
                    latch.countDown();
                    TaskManager.instance.removeTaskListener(this);
                }
            }
        });
        ThreadUtilities.runInBackground(task);
    }
    try{
        latch.await();
    }
    catch (InterruptedException e){
        Log.log(Log.ERROR, this, e, e);
    }
    Runnable delayedAWTRequest = new DelayedEndRequest();
    EventQueue.invokeLater(delayedAWTRequest);
}
```

Fig. 8. Extract Method opportunity identified by the proposed approach in `org.gjt.sp.jedit.browser. VFSBrowser` class of the JEdit project.

Table 12
The results of the t-test.

Project	P-value (alpha = 0.05)		
	SRP	Functionality	Code cohesion
JEdit	0.0084	0.5	0.00080
FreeMind	0.0009	0.98780	0.00002
ArgoUML	0.0003	0.97320	0.00004
JFreeChart	0.0027	0.98230	0.00040
jVLT	0.4207	0.00410	0.00930

2019). However, in the case of functionality, the Hubert method has performed better, specifically on the FreeMind, ArgoUML, and JFreeChart projects. The reason is that the Hubert method applies very strict preconditions to ensure preserving the functionality of the extracted method under various conditions. It results in overly strong preconditions that prevent applying many beneficial transformations (Mongiovi et al., 2018). For example, the Hubert approach does not suggest any refactoring for the method shown in Fig. 8. By relaxing refactoring preconditions, we have improved the potential application of the proposed tool in practice.

The one-tailed two-sample independent t-test was used to determine whether the means of scores obtained by our approach and the Hubert method (Hubert, 2019) were significantly different. The null hypothesis is defined as there is no difference between the mean of obtained scores for a quality attribute of the two approaches. The alternative hypothesis expresses that the mean of scores obtained by our approach for a specific quality attribute is greater than the Hubert approach. The p-values for SRP, functionality, and cohesion are shown in Table 12. Considering the significance level, alpha = 0.05, the mean score of SRP obtained by the proposed approach is significantly higher than the SRP score obtained by the Hubert approach for all projects except the jVLT. Moreover, for the code cohesion, the proposed approach has performed significantly better. As mentioned earlier, the functionality of the proposed approach is not better than the Hubert approach except for the jVLT project. Therefore, the null hypothesis is rejected for the SRP and code cohesion, but it is accepted for the functionally.

4.3.2. *Impacts on the lines of code*

Undoubtedly, the most common symptom of the long method code smell is the high number of lines of code, which makes the

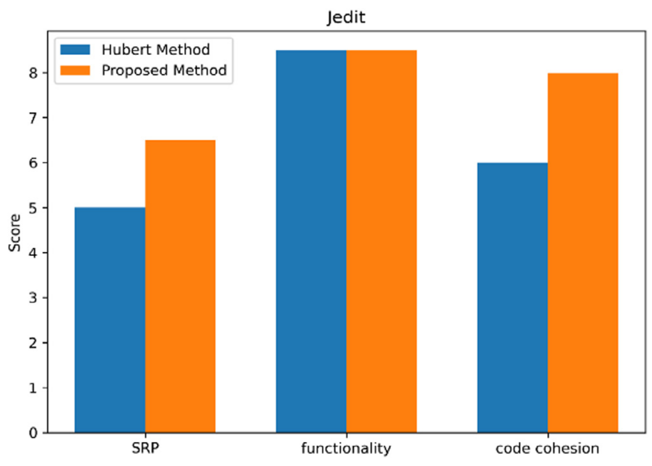


Fig. 9. Comparison of the proposed extract method refactoring approach with the Hubert approach on JEdit.

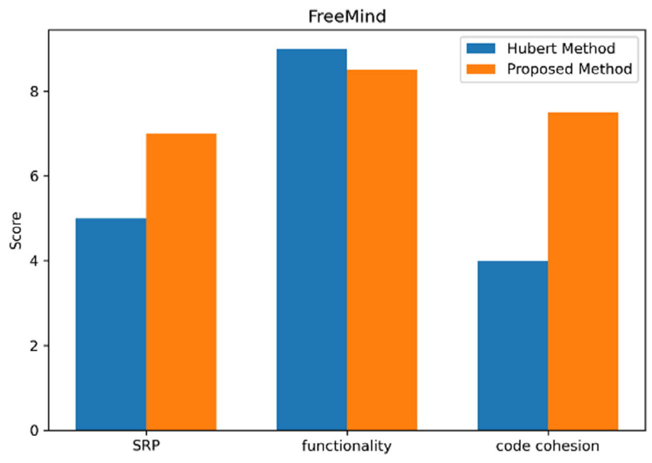


Fig. 10. Comparison of the proposed extract method refactoring approach with the Hubert approach on FreeMind.

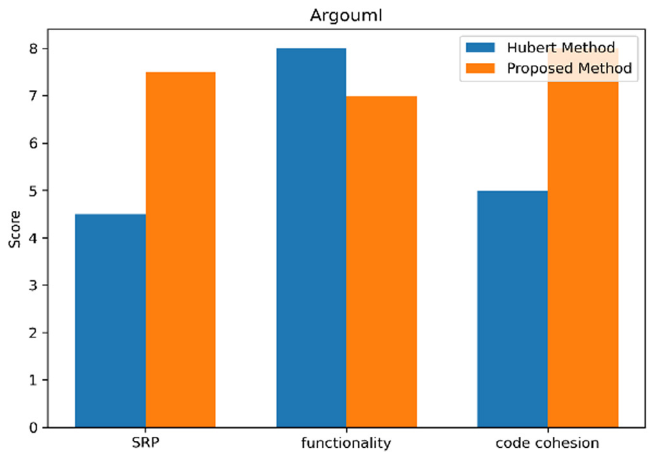


Fig. 11. Comparison of our extract method refactoring approach with Hubert approach on ArgoUML.

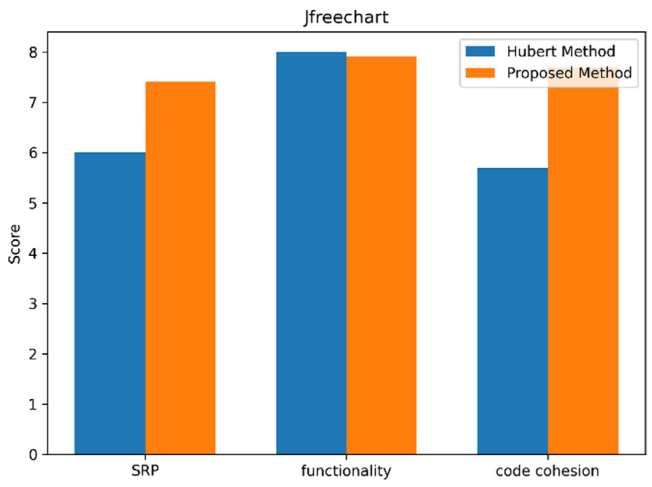


Fig. 12. Comparison of our extract method refactoring approach with Hubert approach on JFreeChart.

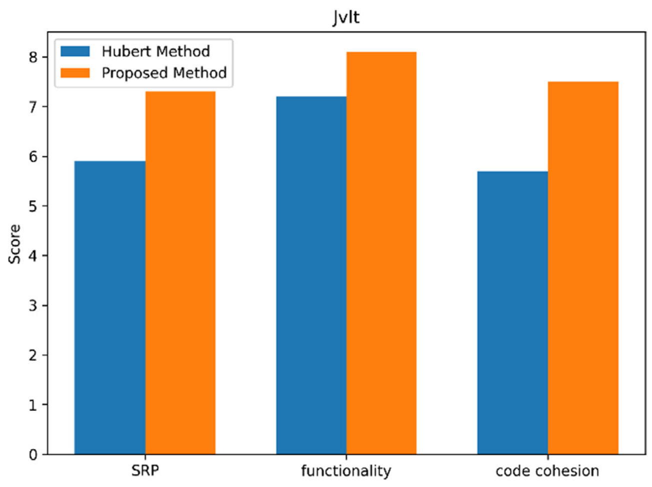


Fig. 13. Comparison of our extract method refactoring approach with Hubert approach on jVLT.

code less readable and very difficult to maintain. The primary purpose of extract method refactoring is to reduce the number of lines of code and thus increase the code quality by fixing the relevant code smell. For this reason, it is crucial to evaluate the proposed approach with respect to changes in methods' lines of code. We have examined the proposed extract method refactorings regarding the changes in the number of lines of code. For this experiment, the threshold value for the number of lines of code was set to 40. Then, the identified long methods are refactored.

Figs. 14 to 18 show the comparison between the lines of code before and after applying all suggested refactorings. The vertical axis in each diagram represents the number of lines of code, and each point on the horizontal axis represents a method. We have observed that the number of lines of code is reduced by more than half in some of the long methods. For example, the lines of code for the long method shown in Fig. 7 reduce from 45 to 18 after refactoring.

On the other hand, in some cases, the method's lines of code have not decreased significantly. For instance, the lines of code of the method shown in Fig. 8 reduce from 55 to 36. In another case, in the ArgoUML project, the lines of code for a long method that initially had 120 lines have decreased to 92 lines after refactoring. We also observed that this method still suffers from the long method code smell. The main reason is that the lines of code in this method are highly interdependent. If we separate some statements and place them in another method, the high dependence of the two methods would create another code smell in the program. Indeed, we have to pass many variables required by

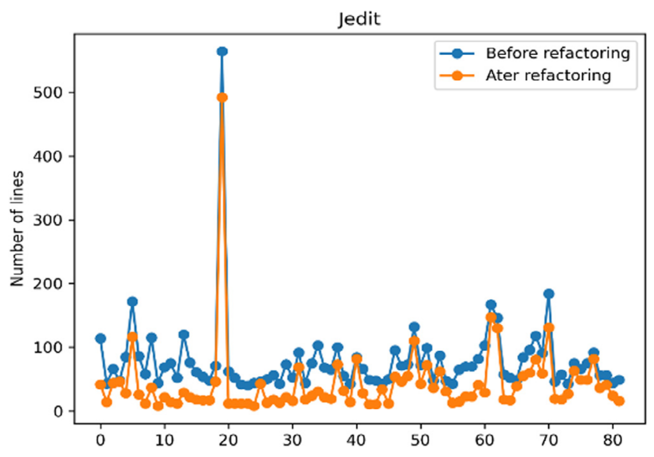


Fig. 14. Comparison of lines of code for long methods in JEdit before and after applying the extract method refactoring.

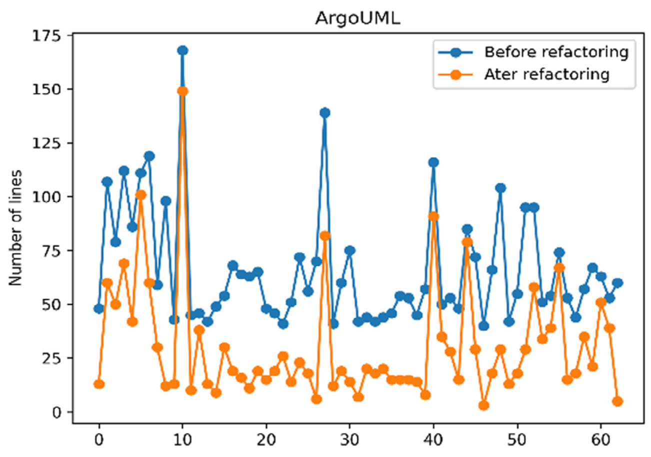


Fig. 16. Comparison of lines of code for long methods in ArgoUML before and after applying extract method refactoring.

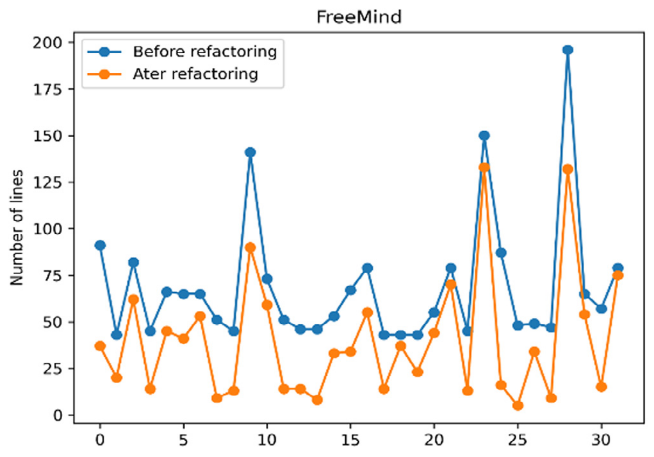


Fig. 15. Comparison of lines of code for long methods in FreeMind before and after applying extract method refactoring.

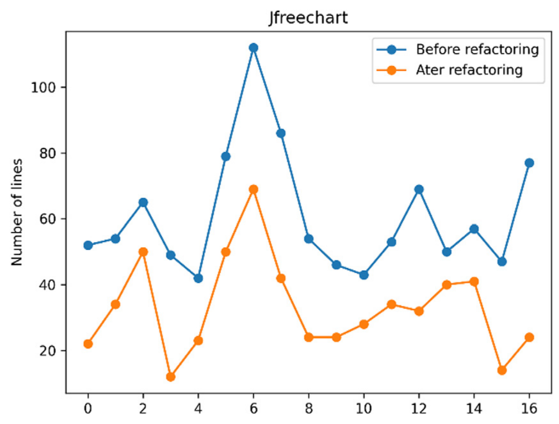


Fig. 17. Comparison of lines of code for long methods in JFreeChart before and after applying extract method refactoring.

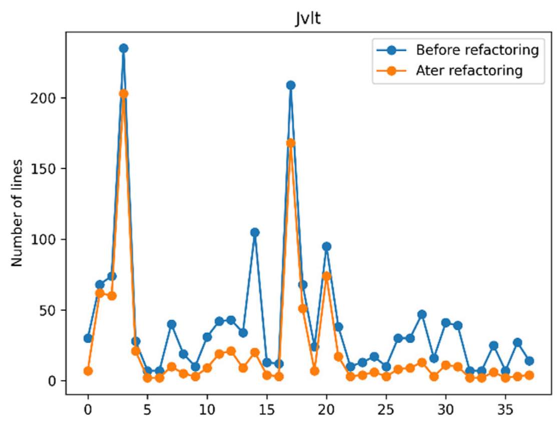


Fig. 18. Comparison of lines of code for long methods in jVLT before and after applying extract method refactoring.

the new method as its input parameters, resulting in a method with the long-parameter list code smell (Fowler and Beck, 2018; Bloch, 2017). As discussed earlier, one of our primary goals is to prevent introducing new code smells to programs after automatically applying the refactoring operation. For long methods whose statements are highly dependent on each other, automatic extraction of a meaningful and useful part would be impossible or challenging. A high portion of these methods is filtered when the feasibility of extract method refactoring is examined.

4.4. Destination class evaluation

In this section, we evaluate the designation of the destination class for the newly extracted method from a long method detected and refactored by the proposed approach. The goal of determining the most appropriate class for the new method is to prevent feature envy code smell after refactoring a long method instance. Feature envy mostly destroys class cohesion and increases the coupling between classes (Fowler and Beck, 2018). We have measured and compared the cohesion of target classes before and after adding a new extracted method. The cohesion value is calculated by Eq. (11), described in Section 3.6 for each destination class. Eq. (11) measured the cohesion based on the clustering coefficient of the class method and attributes. The higher the average of the clustering coefficient, the more cohesive the class would be. Figs. 19 to 23 illustrate the cohesion value for each class selected as the destination for the new method by the proposed approach before and after the move method refactoring. The vertical axis in each diagram shows the degree of cohesion, and each point on the horizontal axis shows a destination class.

In most cases, the class cohesion has either remained without any changes or increased. More precisely, in the JEdit project, the cohesion value in 85% of the classes has remained fixed or

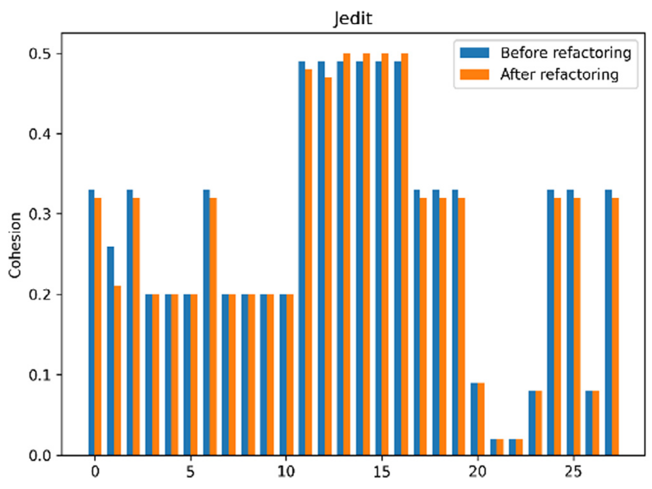


Fig. 19. Class cohesion before and after applying all extract method refactoring opportunities on JEdit.

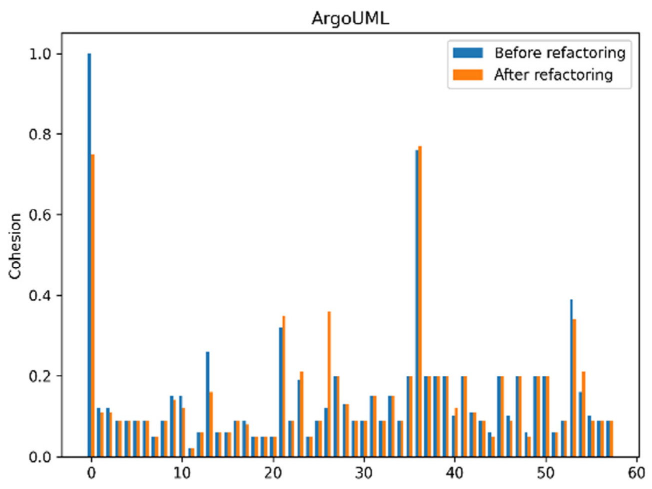


Fig. 21. Class cohesion before and after applying all extract method refactoring opportunities on ArgoUML.

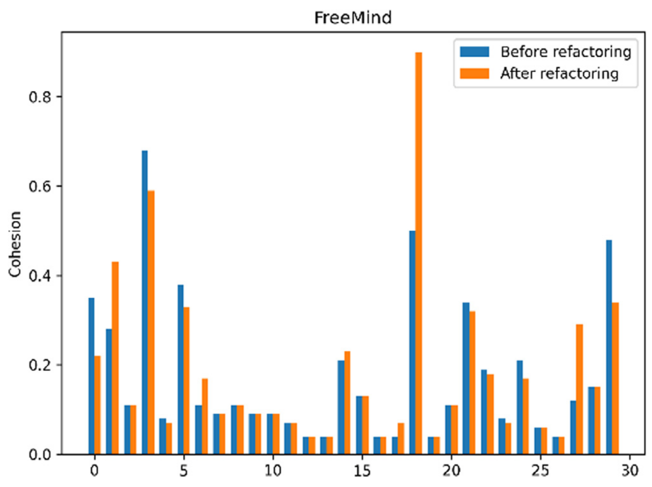


Fig. 20. Class cohesion before and after applying all extract method refactoring opportunities on FreeMind.

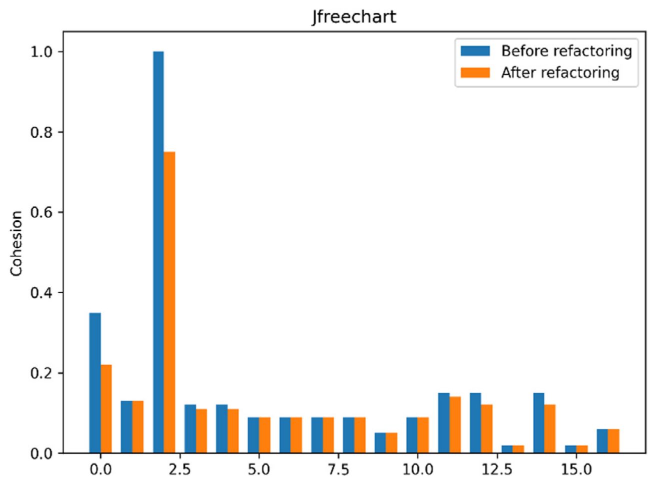


Fig. 22. Class cohesion before and after applying all extract method refactoring opportunities on JFreeChart.

improved. For FreeMind and ArgoUML projects, this rate is 71% and 80%, respectively.

An overall comparison of the class cohesion before and after the refactoring indicates that the destination classes are correctly selected for most of the extracted methods. In some cases, the class cohesion has remained unchanged since the long method's enclosing class has also been selected as the destination class for the extracted method. Therefore, the number of dependencies and the average clustering coefficient have remained fixed. For example, the destination class for the extracted method shown in Fig. 7 is the same since the extracted method does not call any method in others classes.

In cases where the class cohesion has increased, it indicates that the dependencies of the new method on the selected destination class have been high. Therefore, by moving the extracted method to this class, cohesion is improved. For instance, the proposed approaches determined the destination class for the extracted method shown in Fig. 8 `org.gjt.sp.util. TaskManager`. The cohesion of the destination class is changed from 0.76 to 0.77 after applying the move method refactoring, which indicates an improvement in the cohesion.

Although in the proposed method, we aimed to preserve or increase the cohesion degree of the classes as much as possible, as shown in the diagrams, in some cases, the cohesion not only did not increase but also decreased. The reason is the method's inappropriate design and implementation. The existence of many calls to different classes and methods within the method's body or the presence of a long-parameter list are indicators of such inappropriate design choices.

If the code of such methods is broken in any way, inside each of the remaining code snippets, there still exist many calls to other classes. In such cases, if the number of calls and dependencies on two or more classes are equal, we have to select one of the classes randomly and suggest it to the developer, which presumably results in decreasing class cohesion. We have mostly observed these cases for the methods in UI classes since the UI classes are often highly coupled and less coherent compared to classes implementing business logic. Finally, the results demonstrate that the move method refactoring must be applied to preserve or improve class cohesion after applying any extract method refactoring.

5. Practical limitations

The proposed approach in this paper addresses both the identification and application of automated extract method refactoring. It also automatically set a name for the extracted method

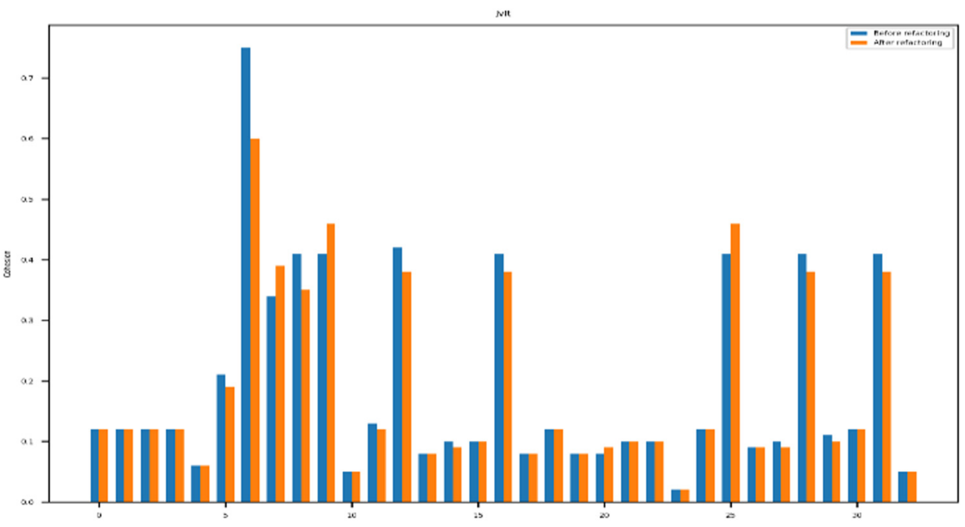


Fig. 23. Class cohesion before and after applying all extract method refactoring opportunities on jVLT.

to avoid any human intervention during the refactoring process. However, it comes up with several limitations which can be considered as future work in this area. We have briefly discussed some of these limitations in this section.

One critical aspect of any automated refactoring is the ability to preserve the program's behavior. The proposed approach extracts part of the long method's body according to the data and control dependencies between the lines of code, assuming that the lines that are dependent on each other perform a single unit of work. However, there is no definite guarantee that the proposed approach always preserves the behavior considering only the data and control dependencies (Mens and Tourwe, 2004). Although the generated refactorings are evaluated and approved by experts for five Java open source projects, in general, the problem of behavior preservation through the program transformation is undecidable (Mens and Tourwe, 2004). Excessive preconditions may lead to the loss of many appropriate refactoring opportunities. While ignoring all preconditions leads to the destruction of the program's behavior. Establishing a trade-off between the two ends of this spectrum in automated refactoring requires further investigation. Additional checks, including regression testing, can help to ensure behavior preservation. The proposed approach ranks the generated candidates only based on the modularity of the original method and the extracted part. The modularity value for the generated candidates may be very close or even equal. In such cases, other metrics such as lines of code can be used to select the best candidate instead of a random selection.

Automated refactoring can expose code chunks that might lead to exploitable vulnerabilities. For instance, creating an extracted method that includes and manipulates variables defined as final in the initial method might risk the integrity of the overall system. The proposed extract method tool adds a security flag, "potentially unwanted security vulnerability", to recommended refactorings' output to be considered by the user when applying the refactorings. The flag is set to true for long methods whose access modifier is not "private", *i.e.*, can be called from the outside of the enclosing class, and their extracted parts contain final variables. However, automatically detecting all possible security issues exposed by automated refactoring is challenging and requires further attention. We suggest the best applicable refactorings with this security flag to the user and do not change the program's source code. Therefore, we ensure that the proposed refactorings do not directly affect the system's security.

We have implemented and evaluated the proposed approach only for Java programs. To confirm the generalization of the results, we evaluated the proposed approach on five sizable Java, including 39,397 Java methods projects in different application domains. On average, the processing time of each method in a given program takes about 20 s in our settings (*i.e.* Ubuntu Linux machine with an Intel® Core i7™processor and 16 Gigabytes of RAM), which is an acceptable time. However, evaluating the performance of this approach on other programming languages requires specific implementations due to some of the programming language structures considered in the proposed method. Finally, if the lines of code in the long method body are highly dependent on each other, the proposed approach fails to extract the appropriate candidates.

6. Threats to validity

Several factors threaten the validity of the proposed approach and evaluations used in this paper. The main threat to construct validity is the overestimation of call graph edges due to the polymorphic instances and methods. This can impact the validity of the steps for generating candidate methods and finding destination classes. Java is one of the leading OOP languages, and thus, polymorphism is one of its core features. We have used a third-party tool, JavaCallGraph, to calculate the Java call graph developed by Gousios et al. (2018). JavaCallGraph can generate both the static and dynamic call graphs for Java programs. We used the static call graph generation module of the Java call graph tool, which generates the program's static call graph by analyzing JVM byte codes. Unfortunately, not all types can be inferred with the static analysis. On the other hand, the quality and precision of the dynamic call graphs are highly influenced by the size and quality of the corresponding test suite, which often is not available for most open-source projects (Jász et al.). Static call graphs are more suitable for refactoring since the refactoring is about the static structure of the program.

Hyperparameters, including the long method detection threshold for $S_{LM}(M)$ in Eq. (1) and the number of extracted method arguments may threaten the internal validity of the proposed approach's evaluation. However, we have experimented with various thresholds to obtain the best value with the highest long method detection recall. We have also cared about the number of arguments such that the long parameter list does not appear after the extract method refactoring is performed. Another possible threat is that the evaluations confirmed the behavior preservation only by considering the input–output of the refactored program, which may not be sufficient for some types of systems, mainly embedded and real-time systems.

Concerning the external validity, a potential threat to the generalization of the results is the limited number of selected projects and human experts. The experts might also not be familiar with projects analyzed in the experiments. The selected independent experts spent considerable time and effort investigating the projects and evaluating the refactorings recommended by the proposed tool to reduce the possible external threats. We have also selected open-source projects which are well documented and have an acceptable level of understandability. The benchmark projects are large enough to cover the different shapes of the long method, and feature envy code smells.

7. Conclusions

It is possible to successfully identify and refactor the existing long methods by applying various advanced graph analysis techniques such that the principles of object-oriented design are met while extracting pieces of code. This paper proposes a sequence of static analysis operations to detect and suggest the best refactoring solution for fixing long method code smells in Java source codes without introducing the new smells. We have demonstrated the applicability of the proposed approach in identifying and refactoring long methods on five different open-source Java projects, with a total of 39,397 methods distributed amongst 5,415 classes. Betweenness centrality of internal and external dependencies of a given method is used to assign a long method score. The possibility of refactoring is checked by modularity analysis on the dependency graph of the method's lines of code. The candidate methods for extraction are generated and ranked respectively by clustering coefficient and modularity analysis on the data and control dependency graph of the method's body. The candidate extraction algorithm avoids generating a low cohesive code snippet or a long-parameters list by limiting the number of extracted lines and required parameters. An appropriate name is automatically generated and assigned to a newly extracted method to make a fully automated refactoring pipeline and prevent readability issues. Finally, the best destination class for the newly created method is determined by selecting a class with the highest cohesion and coupling ratios when enclosing the target method. Therefore, the approach prevents the generation of feature envy code smell and preserves the program's modularity. We have concluded that the move method refactoring must be applied to preserve or improve class cohesion after applying the extract method refactoring.

By performing various experiments and using experts' opinions and object-oriented code metrics for verification, the results reveal the proposed method's applicability in refactoring long methods to meaningful, useful, functional, and readable methods. The results also indicate that the proposed approach outperforms the state-of-the-art automatic extract method refactoring approaches (Hubert, 2019) for the single responsibility principle by 21%. The betweenness centrality of the program dependency graph is a good metric to measure class coupling, and the clustering coefficient is a suitable metric to measure class cohesion. The modularity of the control and data dependency graph constructed from the method's body indicates possible extract method opportunities in a long method. Further analysis is still required to identify a set of applicable extract method candidates in the presence of highly dependent code segments. In addition, behavior preservation after refactoring in such cases is another area to investigate as future works. While the proposed approach automatically suggests the best possible refactorings, the complete proposed refactoring process must be automated to reduce the cost and time. There are several other opportunities to study in future works. First, the proposed approach can be used in a search-based refactoring process (Mariani and Vergilio, 2017) to find the best refactoring sequences that maximize software quality attributes. In this way, the synergy of applying different refactorings after the extract method refactoring is investigated automatically. Second, the responsibility of methods is mainly denoted by statements that expose the output of that method (*e.g.*, return or print statements). The candidate generation and ranking algorithms can consider the impact of such statements for the extract method refactorings to generate more accurate results. Finally, more advanced method name recommendation mechanisms (Alon et al., 2019; Jiang et al.; Zaitsev et al.) can be combined with the proposed rule-based approach to offer a more natural and developer-friendly name for the extracted method based on the functional relevance.

Compliance with ethical standards

This study has received no funding from any organization.

CRediT authorship contribution statement

Mahnoosh Shahidi: Conceptualization, Methodology, Validation, Writing – original draft. **Mehrdad Ashtiani:** Conceptualization, Writing – review & editing, Supervision. **Morteza Zakeri-Nasrabadi:** Validation, Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Ethical approval

This article does not contain any studies with human participants or animals performed by any of the authors.

References

Agnihotri, M., Chug, A., 2020. A systematic literature survey of software metrics, code smells and refactoring techniques. J. Inf. Process. Syst. 16 (4), 915–934.

Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D., 2006. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA.

Al Dallal, J., 2011. Measuring the discriminative power of object-oriented class cohesion metrics. IEEE Trans. Softw. Eng. 37 (6), 788–804. http://dx.doi.org/10.1109/TSE.2010.97.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. Code2vec: learning distributed representations of code. Proc. ACM Program. Lang. 3 (POPL), 1–29. http://dx.doi.org/10.1145/3290353.

Anon, 2019. Javaparser: analyze, transform and generate your java codebase. https://javaparser.org/ (accessed May 18, 2021).

Arcelli Fontana, F., Mäntylä, M.V., Zanoni, M., Marino, A., 2016. Comparing and experimenting machine learning techniques for code smell detection. Empir. Softw. Eng. 21 (3), 1143–1191. http://dx.doi.org/10.1007/s10664-015-9378-4.

Baqais, A.A.B., Alshayeb, M., 2020. Automatic software refactoring: A systematic literature review. Softw. Qual. J. 28 (2), 459–502. http://dx.doi.org/10.1007/s11219-019-09477-y.

Bibiano, A.C., et al., A quantitative study on characteristics and effect of batch refactoring on code smells, in: Proceedings of the 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Sep. 2019, 1–11, http://dx.doi.org/10.1109/ESEM.2019.8870183.

Bloch, J., 2017. Effective Java. Pearson Education.

Caram, F.L., Rodrigues, B.R.D.O., Campanelli, A.S., Parreiras, F.S., 2019. Machine learning techniques for code smells detection: A systematic mapping study. Int. J. Softw. Eng. Knowl. Eng. 29 (02), 285–316. http://dx.doi.org/10.1142/S021819401950013X.

Cedrim, D., et al., Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Aug. 2017, 465–475, http://dx.doi.org/10.1145/3106237.3106259.

Charalampidou, S., Ampatzoglou, A., Chatzigeorgiou, A., Gkortzis, A., Avgeriou, P., 2017. Identifying extract method refactoring opportunities based on functional relevance. IEEE Trans. Softw. Eng. 43 (10), 954–974. http://dx.doi.org/10.1109/TSE.2016.2645572.

Di Nucci, D., Palomba, F., Tamburri, D.A., Serebrenik, A., De Lucia, A., Detecting code smells using machine learning techniques: Are we there yet?, in: Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Mar. 2018, 612–621, http://dx.doi.org/10.1109/SANER.2018.8330266.

Fontana, F.A., Ferme, V., Zanoni, M., Roveda, R., Towards a prioritization of code debt: A code smell Intensity Index, in: Proceedings of the IEEE 7th International Workshop on Managing Technical Debt (MTD), Oct. 2015, 16–24, http://dx.doi.org/10.1109/MTD.2015.7332620.

Fortunato, S., 2010. Community detection in graphs. Phys. Rep. 486 (3–5), 75–174. http://dx.doi.org/10.1016/j.physrep.2009.11.002.

Fowler, M., Beck, K., 2018. Refactoring: Improving the Design of Existing Code, Second Edi. Addison-Wesley.

Gilbert, D., 2020. JFreeChart. http://www.jfree.org/jfreechart/ (accessed Sep. 30, 2020).

Gousios, G., Vergne, M., Laaber, C., 2018. Java-callgraph. https://github.com/gousiosg/java-callgraph (accessed May 18, 2021).

Gu, A., Zhou, X., Li, Z., Li, Q., Li, L., 2017. Measuring object-oriented class cohesion based on complex networks. Arab. J. Sci. Eng. 42 (8), 3551–3561. http://dx.doi.org/10.1007/s13369-017-2588-x.

Hubert, J., 2019. Implementation of an Automatic Extract Method Refactoring. University of Stuttgart.

Jász, J., Siket, I., Pengő, E., Ságodi, Z., Ferenc, R., Systematic comparison of six open-source Java call graph construction tools, in: Proceedings of the 14th International Conference on Software Technologies, 2019, 117–128, http://dx.doi.org/10.5220/0007929201170128.

Jiang, L., Liu, H., Jiang, H., Machine learning-based recommendation of method names: how far are we, in: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), Nov. 2019, 602–614, http://dx.doi.org/10.1109/ASE.2019.00062.

jVLT - a vocabulary learning tool, http://jvlt.sourceforge.net/ (accessed Sep. 15, 2021).

Kaur, S., Singh, P., 2019. How does object-oriented code refactoring influence software quality? research landscape and challenges. J. Syst. Softw. 157, 110394. http://dx.doi.org/10.1016/j.jss.2019.110394.

Khomh, F., Di Penta, M., Guéhéneuc, Y.G., Antoniol, G., 2012. An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness 17 (3), 243–275. http://dx.doi.org/10.1007/s10664-011-9171-y.

Lacerda, G., Petrillo, F., Pimenta, M., Guéhéneuc, Y.G., 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. J. Syst. Softw. 167, 110610. http://dx.doi.org/10.1016/j.jss.2020.110610.

Lanza, M., Marinescu, R., 2006. Object-Oriented Metrics in Practice: Using Software Metrics To Characterize, Evaluate, and Improve the Design of Object-Oriented Systems, 1st Ed.. Springer Publishing Company, Incorporated.

Leicht, E.A., Newman, M.E.J., 2008. Community structure in directed networks. Phys. Rev. Lett. 100 (11), 118703. http://dx.doi.org/10.1103/PhysRevLett.100.118703.

Lozano, A., Mens, K., Portugal, J., Analyzing code evolution to uncover relations, in: Proceedings of the IEEE 2nd International Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP), Mar. 2015, 1–4. http://dx.doi.org/10.1109/PPAP.2015.7076847.

Mansoor, U., Kessentini, M., Wimmer, M., Deb, K., 2017. Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm. Softw. Qual. J. 25 (2), 473–501. http://dx.doi.org/10.1007/s11219-015-9284-4.

Mariani, T., Vergilio, S.R., 2017. A systematic review on search-based refactoring. Inf. Softw. Technol. 83, 14–34. http://dx.doi.org/10.1016/j.infsof.2016.11.009.

Martin, R.C., 2009. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice-Hall.

Wiley & Sons, Martin Lippert, S.R., 2006. Refactoring in Large Software Projects: Performing Complex Restructurings Successfully, 1st Ed.

Maruyama, K., 2001. Automated method-extraction refactoring by using block-based slicing. ACM SIGSOFT Softw. Eng. Notes 26 (3), 31–40. http://dx.doi.org/10.1145/379377.375233.

Mens, T., Tourwe, T., 2004. A survey of software refactoring. IEEE Trans. Softw. Eng. 30 (2), 126–139. http://dx.doi.org/10.1109/TSE.2004.1265817.

Mkaouer, M.W., Kessentini, M., Bechikh, S., Cinnéide, M.Ó., Deb, K., 2016. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. Empir. Softw. Eng. 21 (6), 2503–2545. http://dx.doi.org/10.1007/s10664-015-9414-4.

Mohan, M., Greer, D., 2019. Using a many-objective approach to investigate automated refactoring. Inf. Softw. Technol. 112, 83–101. http://dx.doi.org/10.1016/j.infsof.2019.04.009.

Mongiovi, M., Gheyi, R., Soares, G., Ribeiro, M., Borba, P., Teixeira, L., 2018. Detecting overly strong preconditions in refactoring engines. IEEE Trans. Softw. Eng. 44 (5), 429–452. http://dx.doi.org/10.1109/TSE.2017.2693982.

Müller, J., 2021. Freemind - free mind mapping software. http://freemind.sourceforge.net/wiki/, (accessed Apr. 28, 2021).

NetworkX, NetworkX. https://networkx.github.io/ (accessed Apr. 26, 2019).

Newman, M., 2010. Networks: An Introduction. OUP Oxford.

Newman, M.E.J., Girvan, M., 2004. Finding and evaluating community structure in networks. Phys. Rev. E 69 (2), http://dx.doi.org/10.1103/PhysRevE.69.026113.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A., 2018. On the diffuseness and the impact on the maintainability of code smells: a large scale empirical investigation. Empir. Softw. Eng 23 (3), 1188–1221. http://dx.doi.org/10.1007/s10664-017-9535-z.

Palomba, F., Panichella, A., De Lucia, A., Oliveto, R., Zaidman, A., A textual-based technique for Smell Detection, in: Proceedings of the IEEE 24th International Conference on Program Comprehension (ICPC), 2016, 1–10. http://dx.doi.org/10.1109/ICPC.2016.7503704.

Pecorelli, F., Di Nucci, D., De Roover, C., De Lucia, A., 2020. A large empirical assessment of the role of data balancing in machine-learning-based code smell detection. J. Syst. Softw. 169, 110693. http://dx.doi.org/10.1016/j.jss.2020.110693.

Pestov, S., 2021. Jedit - programmer's text editor. http://www.jedit.org/ (accessed Apr. 28, 2021).

Robbins, J.E., 2021. Argouml. https://argouml.en.softonic.com/ (accessed Apr. 28, 2021).

Savić, M., Ivanović, M., Radovanović, M., 2017. Analysis of high structural class coupling in object-oriented software systems. Computing 99 (11), 1055–1079. http://dx.doi.org/10.1007/s00607-017-0549-6.

Sharma, T., Spinellis, D., 2018. A survey on software smells. J. Syst. Softw. 138, 158–173. http://dx.doi.org/10.1016/j.jss.2017.12.034.

Silva, D., Terra, R., Valente, M.T., Recommending automated extract method refactorings, in Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014, 2014, 146–156, http://dx.doi.org/10.1145/2597008.2597141.

Sobrinho, E.V.de P., De Lucia, A., de A. Maia, M., 2021. A systematic literature review on bad smells–5 w's: which, when, what, who, where. IEEE Trans. Softw. Eng. 47 (1), 17–66. http://dx.doi.org/10.1109/TSE.2018.2880977.

Steidl, D., Eder, S., Prioritizing maintainability defects based on refactoring recommendations, in: Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014, 2014, 168–176, http://dx.doi.org/10.1145/2597008.2597805.

Subramaniam, H., Zulzalil, H., 2012. Software quality assessment using flexibility: a systematic literature review. Int. Rev. Comput. Softw. 7 (5).

Taibi, D., Janes, A., Lenarduzzi, V., 2017. How developers perceive smells in source code: A replicated study. Inf. Softw. Technol. 92, 223–235. http://dx.doi.org/10.1016/j.infsof.2017.08.008.

Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A., Ten years of JDeodorant: lessons learned from the hunt for smells, in: Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Mar. 2018, 4–14. http://dx.doi.org/10.1109/SANER.2018.8330192.

Tsantalis, N., Chatzigeorgiou, A., 2011. Identification of extract method refactoring opportunities for the decomposition of methods. J. Syst. Softw. 84 (10), 1757–1782. http://dx.doi.org/10.1016/j.jss.2011.05.016.

Tufano, M., et al., 2017. When and why your code starts to smell bad (and whether the smells go away). IEEE Trans. Softw. Eng. 43 (11), 1063–1088. http://dx.doi.org/10.1109/TSE.2017.2653105.

Wang, X., Pollock, L., Vijay-Shanker, K., Automatic segmentation of method code into meaningful blocks to improve readability, in: Proceedings of the 18th Working Conference on Reverse Engineering, Oct. 2011, 35–44, http://dx.doi.org/10.1109/WCRE.2011.15.

Yamashita, A., Moonen, L., Do code smells reflect important maintainability aspects?, in: Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM), Sep. 2012, pp. 306–315 http://dx.doi.org/10.1109/ICSM.2012.6405287.

Yamashita, A., Moonen, L., Do developers care about code smells? an exploratory survey, in:Proceedings of the 20th Working Conference on Reverse Engineering (WCRE), Oct. 2013, 242–251, http://dx.doi.org/10.1109/WCRE.2013.6671299.

Zaitsev, O., Ducasse, S., Bergel, A., Eveillard, M., Suggesting Descriptive Method Names: An Exploratory Study of Two Machine Learning Approaches, in: Proceedings of the International Conference on the Quality of Information and Communications Technology, 2020, 93–106, http://dx.doi.org/10.1007/978-3-030-58793-2_8.

Mahnoosh Shahidi received her B.S. degree in computer engineering from Iran University of Science and Technology (2018), Tehran, Iran and her M.S. degree in software engineering from Iran University of Science and Technology (2021), Tehran, Iran. Her main research interests include automated software refactoring and code structure analysis.

Mehrdad Ashtiani received his B.S. degree in software engineering from Iran University of Science and Technology (2009), Tehran, Iran, the M.S. degree in software engineering from Iran University of Science and Technology (2011), Tehran, Iran and finally his Ph.D. degree from Iran University of Science and Technology (2015), Tehran, Iran. His main research interests include automatic refactoring and software modeling.

Morteza Zakeri-Nasrabadi received his M.Sc. degree in Software Engineering from the Iran University of Science and Technology (IUST) in 2018. He is a Ph.D. student in the School of Computer Engineering at Iran University of Science and Technology. His research interests are software engineering and machine learning. Currently, he works on applying machine learning techniques to software testing and refactoring.