Weighted Content Similarity Feature for Software Architecture Anti-Patterns Prediction

Somayeh Kalhor [a], Mohammad Reza Keyvanpour*[b]

[a] Department of Computer and Information Technology Engineering, Islamic Azad University, Qazvin Branch, Qazvin, Iran; Kalhor.somayeh@iau.ac.ir

[b] Department of Computer Engineering, Faculty of Engineering, Alzahra University, Tehran, Iran; keyvanpour@alzahra.ac.ir

ABSTRACT

As user needs change frequently over time, software systems must evolve; therefore, increased software complexity inevitably violates software engineering principles. The violations of these principles are called anti-patterns, which differ from bugs and faults, and can occur at various levels of abstraction; finally, they reduce software quality. Anti-patterns can occur in various software, including web applications, and their prediction can effectively help prevent their occurrence. The anti-patterns prediction process at different levels of abstraction utilizes software features, whose threshold values impact the accuracy of this process. This study presents an improved component-level feature, called weighted content similarity, to more accurately detect component dependencies by minimizing the influence of common words that are often used in comments but are worthless in identifying the relationship between components. Therefore, the comment words are weighted using TF-IDF values. F-Measure values are calculated to show the greater impact of our proposed weighted feature compared to structural, topological, and content similarity features on detecting dependencies between components of an open-source system. The prediction of component anti-patterns, such as cyclic and hub-like dependencies, will be possible with the help of dependency detection. The average F-Measure of topological features in OpenJPA 2.0.0 software is 0.73, content similarity features is 0.76, and weighted content similarity features is 0.88. Therefore, the F-Measure of our weighted content similarity feature is 0.12 higher than the unweighted content similarity feature and is 0.15 higher than the topological feature. So, it is more effective than these two features in predicting dependencies between components using machine learning algorithms.

***Keywords*— Architectural Anti-Patterns, Dependency-Based Smells, Effective Features in Dependencies.**

1. Introduction

Software systems increasingly encounter various anti-patterns throughout their lifecycle. They are conditions that contradict established software engineering principles, thereby complicating system maintenance and reducing the overall quality of the software [1]. Anti-patterns may occur in various systems, including distributed systems, which have been investigated in some studies [2]. They can be at different abstraction levels: statement, class, and component [3,4]. This study focuses on component-level or architectural-level anti-patterns, which often stem from how systems are decomposed into components, the configuration of these components, and the nature of their interrelationships [5]. They negatively impact essential system attributes, such as correctness, performance, reliability, and maintainability. Prior research has sought to assess the impact of architectural anti-patterns on system performance [6]. Anti-patterns in web applications are also inevitable because they consist of connected components. Anti-pattern detection plays a crucial role in improving software quality, and it can rely on deep learning (DL) techniques, whose effectiveness is highly dependent on the quality of the training data. However, little attention has been paid to analyzing the data preparation process, but some research analyzes the data preparation techniques used in anti-pattern detection methods based on DL [7]. Several anti-pattern detection tools have been proposed that are based on metrics (features) and have been compared to the gold standard in some studies [8]. Furthermore, our previous work contributed to this field by evaluating various metric-based automatic detection methods [9]. In this research, the values of specific features in the architecture are calculated, and their impact on the dependencies between

http://dx.doi.org/10.22133/ijwr.2025.525060.1284

Citation S. Kalhor, M.R. Keyvanpour, "Weighted Content Similarity Feature for Software Architecture Anti-Patterns Prediction", *International Journal of Web Research*, vol.8, no.3,pp.33-43, 2025, doi: http://dx.doi.org/10.22133/ijwr.2025.525060.1284.

*Coressponding Author
Article History: Received: 20 April 2025; Revised: 1 June 2025 ; Accepted: 20 June 2025.

software components is analyzed. Furthermore, a weighted content similarity feature is proposed that weights some bag-of-words by using their TF-IDF values [10,11]. This analysis facilitates the prediction of potential architectural anti-patterns, such as cyclic dependencies or hub-like components, in future system versions.

To assess the influence of component-level feature values on component dependencies, we use the Apache OpenJPA project. This project was developed by an open-source community with an extensive pool of contributors, thereby reducing concerns related to individual programming preferences and the potential for coding errors. Therefore, we are sure that our findings are both valid and dependable. In the first step, software relationships between components are identified. Software dependencies can be identified at different levels of abstraction through tools such as LattixDSM [12], Sonargraph [13], and JDeps [14], among others. These tools can extract dependencies between components and methods (at two different levels of abstraction) from the source code. For this analysis, relationships were identified using the JDeps tool [14]. In the second step, the values of the selected features—structure, topology, content similarity, and weighted content similarity—have been computed for all system components. Ultimately, the study explores the impact of these feature values on the presence or absence of inter-component relationships.

Some studies utilized software features to predict component dependencies [15, 16] and employed precision and recall metrics [17, 18]. Neither of these criteria alone is suitable for evaluation because the precision criterion excludes the component dependencies within the sample space that cannot be identified by component-level features (false negatives). On the other hand, the recall metric doesn't consider dependencies that are erroneously identified as existing based on the values of component-level features when they do not (false positives) [17,18]. Consequently, both metrics exhibit errors in evaluating component dependencies detection methods. Therefore, combining these two metrics through the F-Measure provides software developers with a more precise evaluation of component-level features. Also, no previous research has independently assessed the impact of component-level feature values in the presence or absence of component relationships. These studies typically examine the precision and recall metrics for predicting component relationships based on a combination of features, without separately analyzing the significance of each feature. Consequently, the findings presented in this study can be used to identify the importance of each feature in the context of component dependency detection. Additionally, the F-Measure of the new proposed feature (weighted content similarity), which eliminates the negative impact of non-important replicated words in the comments bag-of-words, is calculated and compared to other features.

The structure of the paper is as follows: Section 2 describes a formal problem definition, Section 3 reviews relevant works. Section 4 outlines the software features used to predict component dependencies and our proposed feature. Section 5 examines why OpenJPA software is considered in this study and explores the methods for identifying the relationships between its components. Section 6 provides an overview of the feature values; afterwards, their evaluation is in Section 7. Finally, Section 8 discusses the research, and Section 9 concludes the paper.

2. Problem Definition

If software developers lack insight into the system design, they may change the source code in an undesirable way. This negatively affects the quality of the software structure [1]. On the other hand, understanding the complexity of the relationships between different source code components in a software system is important [15].

Predicting undesirable dependencies between components based on software features and their elimination is a way to deal with structural complexity. The appropriate selection of component-level features and the impact of their values on the decision regarding the existence or absence of a dependency are crucial for accurate dependency predictions. Some component-level features can be defined in such a way that the lower their values, the more likely there is a dependency between components, and some other features can be defined in such a way that the higher their values, the more likely there is a dependency. On the other hand, the efficiency of the dependency prediction method can be increased by weighting each criterion based on its impact on the dependency between two components. This formal definition is shown by Equation(1).

$$DF\left(c_i, c_j\right) = min(W_1 * FL_1, W_2 * FL_2, \dots, W_n * FL_n) \\ and \ \max(W_1 * FH_1, W_2 * FH_2, \dots, W_m * FH_m) \quad (1)$$

DF is a decision function that predicts whether a pair of components, c_i and c_j, is dependent or not. This function must be calculated for all pairs of components in the software. The Wi determines for each feature based on its effect on dependency. The features whose lower values for a pair increase the probability of their dependence are denoted by FL, which can be n in number. Similarly, the features whose higher values for a pair of components increase the probability of their dependence are denoted by FH, which can be m in number.

3. Related Works

The impact of architectural anti-patterns on software maintainability, such as modularity and testability, has been analyzed using empirical evidence by Jolak et al. The effects of seven anti-patterns on modularity and testability in many open-source software projects have been examined separately [50]. Architectural anti-patterns occurring in software components can be detected using two categories of approaches: automated and non-automated [3]. Nowadays, everyone tends to do everything without human intervention and independently of their skill level. Therefore, the focus of anti-pattern detection methods is on automated techniques; this section will focus exclusively on these. The automatic detection of architectural anti-patterns is typically categorized into two broad approaches: 1) clustering-based techniques, and 2) non-clustering-based methods [3], as illustrated in Figure 1. Clustering-based detection methods address the balance between cohesion and coupling criteria, identifying components that exhibit anti-patterns [19]. Adding or removing classes or methods from the clusters can generate the desired software architecture [20, 21]. When software re-modularization entities are represented as classes, these methods operate at the component level. Some studies have employed search-based approaches for re-modularization techniques [22]. A multi-objective search-based fitting function is proposed to formulate the remodulation as an optimization problem. The results show that the proposed method significantly improves the remodulation compared to other methods in terms of Modularization quality (MQ) [43] and Non-Extreme Distribution (NED) [44] metrics, especially for smaller software [42].

Non-clustering-based methods are categorized into two types: 1) detecting by code smell agglomeration rules, and 2) detecting by graph and design structure matrix rules. In Detection by code smell agglomeration rules, source code smells can serve as indicators of architectural anti-patterns, prompting some detection methods to identify architectural issues by ranking groups of related code smells [23]. Code smell agglomeration refers to clusters of related code smells (e.g., smells within the same inheritance hierarchy), likely to signify underlying architectural problems. Detection by graph and design structure matrix rules approaches explore the relationships between components at a higher level of abstraction, utilizing data derived from issue-tracking systems that software development projects use to manage bugs and modifications, as well as commercial reverse engineering tools [24]. This data is used to generate a design structure matrix (DSM), where both the rows and columns correspond to the names of the project's source files. Each cell within the DSM represents a dependency between the file in the row and the file in the column. Furthermore, the historical changes made to the project's files are also analyzed, and based on this information, a distinct "Historical DSM" is constructed.

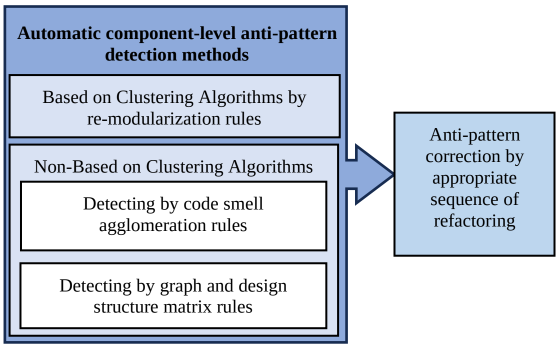

Automatic component-level anti-pattern detection methods
Based on Clustering Algorithms by re-modularization rules
Non-Based on Clustering Algorithms
Detecting by code smell agglomeration rules
Detecting by graph and design structure matrix rules
Anti-pattern correction by appropriate sequence of refactoring

Figure 1. Various component-level anti-pattern detection approaches [3]

The Design Rule Hierarchy (DRH) algorithm [25] organizes Dependency Structure Matrix (DSM) files into two categories: 1) structural dependencies, and 2) historical dependencies, while also reordering the rows and columns. By analyzing the clustered DSMs, various anti-patterns such as unstable interfaces, modularity violations, improper inheritance, cross-module cycles, and cross-package cycles can be identified [26]. Certain anti-pattern detection techniques develop a novel history coupling probability (HCP) matrix that quantifies the likelihood of a file being modified when another file changes, and utilize this matrix to detect anti-patterns [27,28]. The relationships between components can be represented as a graph, upon which graph analysis algorithms are applied to identify dependency-based anti-patterns [29, 30].

A catalog of micro frontend (MFE) [45] (an architectural style inspired by microservices architecture to decompose a monolithic application into smaller parts that can be developed, deployed, and updated independently, and are more flexible and maintainable) anti-patterns is presented [46]. To verify whether the identified problems in MFE architectures are prevalent and whether the proposed solutions effectively address them, a survey was conducted among 20 industry experts.

Additionally, some methods employ social network analysis (SNA) graphs [31, 15] to predict potential links. Through this approach, dependency-based architectural smells, such as cyclic dependencies, hub-like nodes, and nodes with unstable dependencies, can be forecasted. These anti-patterns are discussed in further detail in the subsequent section. Some studies focus on software features to predict these anti-patterns [15], yet none of the existing literature has evaluated the F-Measure criterion for the considered features.

Dependency-based anti-patterns occur when one or more components violate design principles and create undesirable dependencies [32]. Two such examples are Cyclic Dependency (CD) [33] and Hub-Like Dependency (HLD) [34], which can manifest at the class or component level [35]. This study investigates the effectiveness of various criteria in identifying relationships between components, laying the groundwork for predicting these anti-patterns at the component level. A cyclic dependency occurs when a closed loop of dependencies is formed between components, meaning changes to one element in the cycle are likely to impact others. This high degree of interconnectivity between components violates the principle of modularity. An example of a cyclic dependency involving four components is depicted in Figure 2, where the dependencies between classes are represented by arrows, with the cycle highlighted by red arrows.

The Hub-Like Dependency (HLD) anti-pattern arises when a component, which has input and output dependencies with many others, has excessive responsibility [34]. An example of a Hub-like dependency is also shown in Fig. 2, where Component 1 has numerous input and output relationships with other component classes, represented by green arrows. Some studies [47] have tried to help programmers develop their high-quality code. For this purpose, they have presented a pipeline based on the ArchUnit [48] and SonarQube [49] tools, respectively, to find architectural anti-patterns and code smells.

4. Component-Level Software Features

The hemophilia principle, which is used as an important principle to detect dependencies in social networks [36], has also been used to identify dependencies between components in a software system. According to this principle, highly similar individuals have a higher connection rate. In other words, more similar components have a higher chance of forming dependencies with each other. The similarity of components can be calculated based on their common neighbors or their content similarity. This section briefly describes three effective features for identifying potential connections between components in subsequent software versions, which have been investigated in previous studies [15]. Every feature that is more accurate in detecting component dependencies is more efficient in predicting them in future software versions. Therefore, we introduce a more precise feature, namely, weighted content similarity, which is explained in Subsection 4.4.

4.1. Structural Feature

The number of component classes is effective in component relationships.

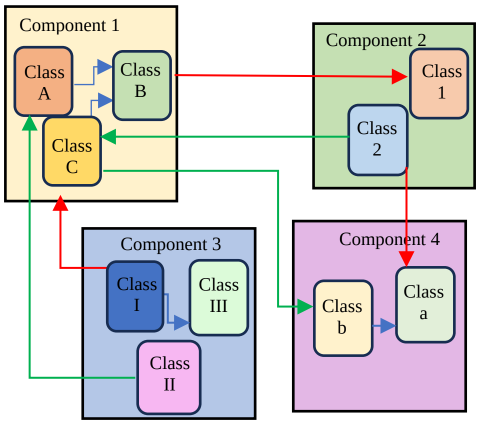


Figure 2. Example of the Cyclic Dependency among four components and Hub-like Dependence [32,33]

4.2. Topological Feature

This feature is calculated from the dependency graph links. The similarity between two nodes can be assessed by the number of common neighbors (the number of common adjacent nodes). In other words, the more neighboring components a pair of components shares, the more similar they are.

4.3. Content Similarity Feature

The source codes of components are checked using this criterion. Converting texts into bags of words [37] in the natural language processing routines used to calculate similarity. Software programs were converted into three sets (bags-of-words) [38] to check the similarity components. These collections are represented by class attribute names (fields), method names, or comments. The similarity evaluation criterion for the two texts is the cosine similarity. The words are first converted into vectors using the Word2vec algorithm, and then, based on Equation(2), this similarity is calculated.

$$\boldsymbol{Cosine\ Similarity}(\boldsymbol{A},\boldsymbol{B}) = \frac{\boldsymbol{A}.\boldsymbol{B}}{||\boldsymbol{A}||.||\boldsymbol{B}||} \qquad (2)$$

A and B are two components that aim to check their similarity. The similarity score for each representation of a bag of words is calculated using this formula. All the words in all the classes of a component, and the nested sub-components in it, are considered in its bag of words. Cosine similarity is calculated separately based on the bag-of-words of feature name, method name, and comment.

4.4. Weighted Content Similarity Feature (Our Proposed Feature)

In the calculation of the content similarity feature to represent bag-of-words, the weights of all word vectors are equal [38]. But we know the contribution of each word in these bags is different. Our general framework for the proposed feature calculation is shown in Figure 3. So, we use a word-weighting

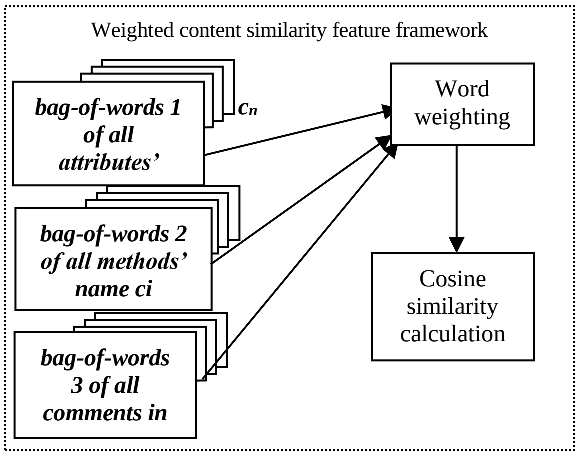

Weighted content similarity feature framework
bag-of-words 1 of all attributes'
cn
bag-of-words 2 of all methods' name ci
bag-of-words 3 of all comments in
Word weighting
Cosine similarity calculation

Figure 3. General framework to calculate our proposed feature

element to assign weight to each word based on its situation.

A. *Bags of Words for each Software Component*

The first step to calculate the weighted content similarity feature is the creation of three bags-of-words for each component in the considered software. A set of all components in the considered software indicated with C that contains n components *{c₁, c₂, ..., cₙ}*. Each component is shown with c_i, in this set, where the i value is 1 to *n*. These bags contain class attribute names (fields), method names, or comments. That must be created for each component denoted by c_i in Equation(3). In this equation, three created bag words are indicated based on their index, i.e., j.

$$b_j^i = \begin{cases} b_1^i & bag\ words\ of\ all\ atributes'names\ of\ classes\ in\ c_i \\ b_2^i & bag\ words\ of\ all\ methods'names\ in\ all\ c_i\ classes \\ b_3^i & bag\ words\ of\ all\ comments\ in\ all\ c_i\ methods\ and\ classes \end{cases} \quad (3)$$

B. *Word Weighting*

The second step in our method is weighting each word based on its situation. Programmers usually pay more attention to naming variables and functions than comments, because they are shorter. Comments are longer and usually in sentence form. Therefore, some common words between them, although frequent, are also worthless for finding connections between components. As a result, we have minimized their importance in our method by giving them the appropriate weight. This process is indicated by Equation (4).

$$\boldsymbol{w_i^{b_j^i}} = \begin{cases} \mathbf{1} & \boldsymbol{if\ j = 1\ all\ words\ in\ c_i} \\ \mathbf{1} & \boldsymbol{if\ j = 2\ all\ words\ in\ c_i} \\ \boldsymbol{TF - IDF} & \boldsymbol{if\ j = 3\ all\ words\ in\ c_i} \end{cases}$$
(4)

Therefore, if the bags of words are related to attributes' names of classes in ci (indicated with b1i) or methods' names for classes in ci (indicated with b2i), we assign 1 for weighting each word. But if the bag of words is related to all comments in all methods and classes in the component indicated by ci (shown with b3i), our method weights words from their TF-IDF values [10,11]. Term Frequency (TF) measures the number of times a term occurs in a component's comments. Since the total length of comments in a component can be short to long, each term may be repeated more in a component with more comments than in a component with fewer comments [11]. Therefore, to solve this problem, term frequency is calculated based on Equation (5). In this formula, c represents all comments in all classes of one component.

$$\boldsymbol{TF(t,c)} = \frac{\boldsymbol{Number\ of\ term\ t\ that\ occurs\ in\ c}}{\boldsymbol{Total\ number\ of\ terms\ in\ c}} \quad (5)$$

Inverse Document Frequency (IDF) reduces the weight of common words across multiple documents while increasing the weight of rare words. If a term appears in fewer documents, it is more likely to be meaningful and specific [11]. The formula for calculating IDF is shown in Equation (6).

$$IDF(t,C) = log\ \frac{Total\ number\ of\ considered\ components}{Number\ of\ components\ containing\ term\ t} \quad (6)$$

TF-IDF is calculated by multiplying Equation (5) and Equation (6) as shown in Equation (7).

$$TF - IDF(t,c,C)\ =\ TF(t,c) * IDF\ (t,C) \quad (7)$$

To increase the impact of important words in comments, should be assigned different coefficients and multiplied by their word vector. The word embedding matrix created based on comments in all classes present in a component is shown as: $W = \{w_1, w_2, ..., w_n\}$, where $w_i \in W$ is the *i-th* line of the embedding matrix W created based on word2vec, and n is the number of words in the comments bag-of-words. We weighted this matrix based on the TF-IDF value of each word, so $\{f_1, f_2, ..., f_n\}$ are coefficients. Finally, the weighted vector of bag-of-words is represented as $U=\{f_1w_1, f_2w_2, ..., f_nw_n\}$. Figure 4 shows the process of calculating our proposed feature.

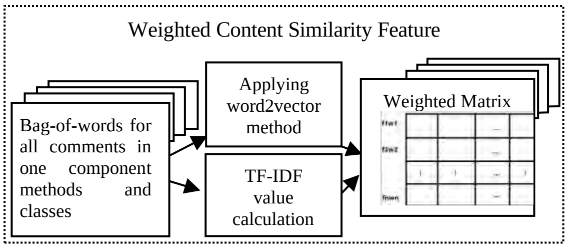

Weighted Content Similarity Feature
Bag-of-words for all comments in one component methods and classes
Applying word2vector method
TF-IDF value calculation
Weighted Matrix

Figure 4. Calculation of Weighted Content Similarity Feature

C. Cosine Similarity Calculation

To calculate it, the words are first converted into vectors by the Word2vec algorithm, and then, based on Subsection B, their weights are determined. Finally, based on (2) as mentioned in Section 4.3, the cosine similarity between each pair of components is calculated for all pairs of components in the open-source software.

5. Why OpenJPA Software

The OpenJPA has been selected in this study among all available open-source software on the Apache Software Foundation site [39]. The Apache Foundation proposed high-quality Java code programs, and most of the proposed architectural anti-pattern detection methods, including the paper by Diaz-Pace et al. [15] and the research by Duque-Min et al. [40], have been used to analyze the performance of their proposed method. On the other hand, OpenJPA has been used in previous studies of anti-pattern detection methods. Another reason is the moderate size of this software (appropriate for research work). The final reason is that different software versions are available on the site.

In this study, the component-level feature values of the OpenJPA software are calculated, and the presence or absence of component dependencies between every component is analyzed using the JDeps tool [14]. This tool, provided by Oracle Corporation (an American computer technology company that designs and manufactures computer hardware and database development tools), was added to JDK version 8 and later to allow developers to identify relationships between different software components. This static analysis tool is command-based and displays dependencies in text format in a console environment. It can display package-level or class-level dependencies by examining Java class files and JAR files.

The results obtained in our presented research, which analyzes the Apache OpenJPA project, are valid and generalizable to other software because this software was not designed by a programming team with a limited number of programmers, but was built by the open-source community for the open-source community. Therefore, concerns about programmer taste and the possibility of programming bugs are minimized. The second reason is the acceptable quality of this software because many versions have been provided that have been reviewed and bugs fixed by the open-source community. Additionally, it is produced by the Apache Software Foundation (ASF, a non-profit corporation) [39], a company with a good reputation in programming. Therefore, this software certainly adheres well to the principles of software engineering.

6. Effect of Software Features on Component Dependencies

The considered software feature values are calculated for OpenJPA 2.0.0, and their charts for this version are shown. Our study aims to introduce a more effective feature at the component level in detecting component dependencies. To prove that our proposed feature is more effective. First, we separately calculate the value of structure, topology, and content similarity features at the component level. Then the existence or non-existence of dependencies between two components is analyzed, and the results are presented in subsections 6.1, 6.2, and 6.3 by diagrams. Furthermore, the value of our proposed feature is calculated and presented in Subsection 6.4. Examining these charts helps researchers to easily compare these features, as their effects have not been comprehensively investigated in previous works. Various software tools, such as JDeps [14] and Python methods [41], were used to analyze the OpenJPA 2.0.0 software for the validity and reliability of the results. In this study, to minimize concerns about researcher skills and human errors, the JDeps tool was applied to identify dependencies between different software components in text format. Additionally, a library function in the Python programming language [41] was used to calculate the similarity of each component with others in the considered open-source project.

6.1. Effect of Structural Feature

The following calculates the structural feature values and their effect on the connection between components. Therefore, the number of classes in each program component is calculated. Also, the number of connected components for each component in the software is counted using the JDeps tool [14]. These values are shown in Figure 5. As can be seen, a component with more classes has more connections. When the number of classes in the component decreases, the probability of connections with other components will decrease.

6.2. Effects of Topological Feature

To calculate the topological feature values, the results of the JDeps tool reveal all component neighbors for each component in the software, separately. Therefore, the relationship between the common neighbors of the components and the direct relationship between them is shown in Figure 6. The number of common neighbors that each software

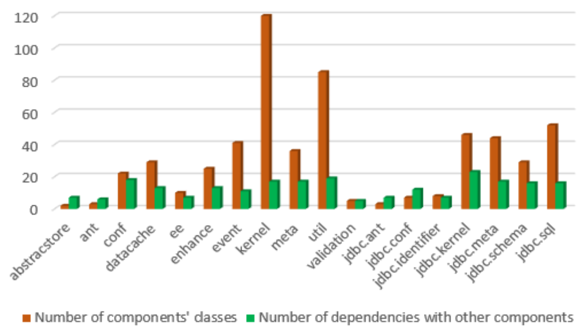


Figure 5. The number of connections between components and the class number of each component

component has with other components is shown in separate rows with different colors (specified in the color chart guide used to display the number of neighbors of each component). The name of each component whose relationship with other components in the software is being examined is displayed as the row name on the left. A green star next to some row is placed, which indicates the direct relationship between the component whose name is written on the left and the component shown in a row in a specific color. The rows without a star next to them indicate that there is no direct relationship between the two components. According to the values shown in the diagram, the probability of a direct connection between two components will increase as the number of common neighbors increases. Therefore, increasing the common neighbors between two components has also increased the possibility of creating a relationship between them in later software versions.

6.3. Content similarity Feature Values

To analyze the effect of the content similarity feature on the relationship between components, the cosine similarity values of our dataset are calculated for each component with the others, separately, as shown in Figure 7. The average cosine similarity is used based on a set of words, including method names, property names in classes, and comments in packages. The cosine similarity value is calculated using methods in Python [41], a powerful programming language; therefore, their results are reliable for analyzing the content similarity feature. The name of each software component is listed on the left side of the graph, and its similarity to each of the other components is shown in different rows with different colors, as displayed with the values specified in the columns. The green star next to the row is the cosine similarity value, indicating a relationship between those two components. Columns without a green star next to them indicate the absence of a relationship between those two components. The probability of a relationship between two things increases as the similarity between them increases.

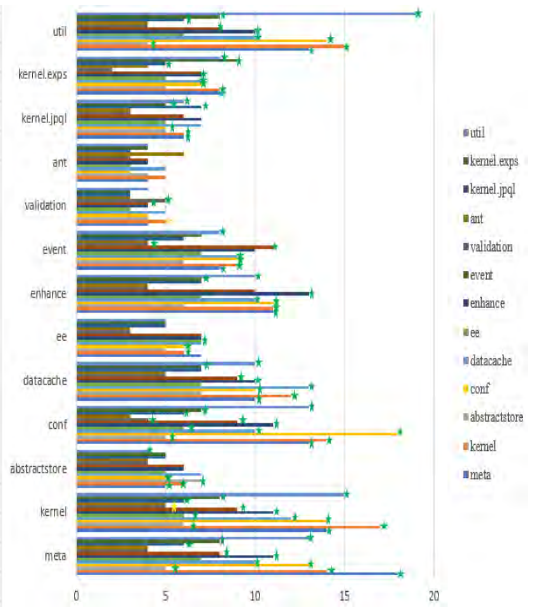


Figure 6. The number of common neighbors components and the component dependencies

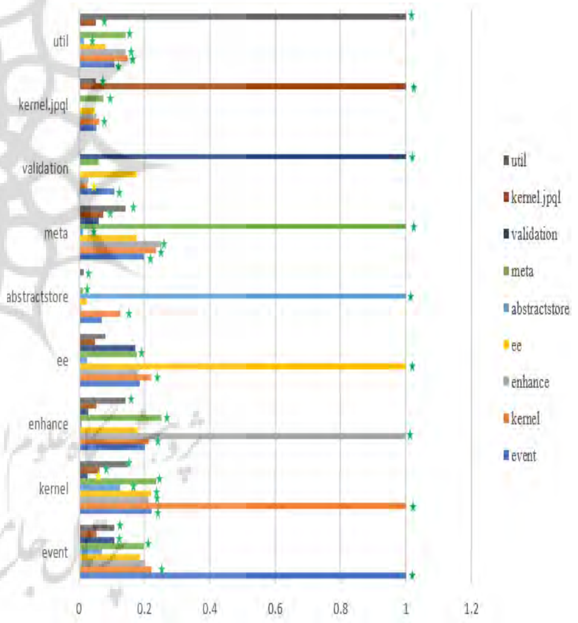


Figure 7. Content similarity values of components and the component dependencies

6.4. Weighted Content Similarity Feature Values

The weighted content similarity feature is the final calculated feature, and its effect on communication between components is analyzed. Figure 8 shows the cosine similarity of each component pair in our dataset separately. Also, the dependence between these values and the existence of a relationship between components is shown. This feature is calculated based on the average cosine similarity of bag-of-words, including method names, property names in classes, and weighted terms of comments in software packages. The graph shows that the probability of a relationship between components with higher weighted content similarity feature values increases.

7. Evaluation

Three evaluation metrics commonly used in various studies to analyze the performance of anti-pattern detection methods are Recall, Precision, and F-Measure [17, 18]. However, previous research has only calculated and compared the Precision and Recall values to investigate the impact of component-level features on detecting dependencies between components. The formulas for calculating these two metrics do not consider all the information in the confusion matrix [17]; therefore, they can lead to errors in evaluating the performance of a method.

The Precision metric ignores dependencies between software components in the sample space that component-level features may fail to identify (false negatives) [17,18]. Similarly, the Recall metric does not consider incorrectly detected dependencies when they do not exist (false positives) [17,18]. This study uses the F-Measure criterion to eliminate these errors and improve the evaluation. To calculate the F-Measure criterion to evaluate the influence of different features on the dependencies of the components, the "precision" and "recall" criteria [17] formulated in Equation (8) must first be calculated.

$$\boldsymbol{precision} = \frac{\boldsymbol{TP}}{\boldsymbol{TP+FP}}, \boldsymbol{Recall} = \frac{\boldsymbol{TP}}{\boldsymbol{TP+FN}} \qquad (8)$$

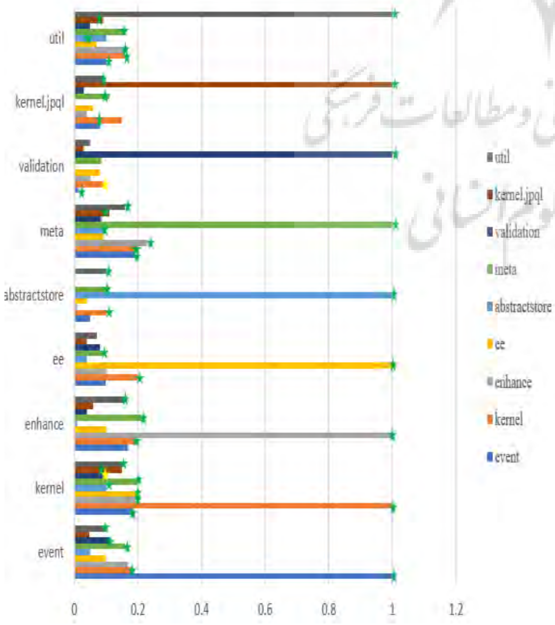


Figure 8. Weighted content similarity feature values of components and the component dependencies

In the precision and recall formulas shown in (8), the number of connected components detected by the JDeps tool [14] and the value of the features presented by TP (true positive). Additionally, the number of connected components that are not detected by JDeps but are incorrectly identified as connected components using feature values is represented by FP (false positive). Finally, the number of connected components specified by JDeps (actually exists), but they haven't been identified by feature values denoted by FN (false negatives).

To demonstrate that our proposed feature has higher accuracy, we compare the confusion matrix of the content similarity feature with the weighted content similarity on each component separately. To calculate the TP, FP, and FN values, firstly, the similarity feature values for a component with other components are calculated, and then the presence or absence of dependence between them is checked according to their thresholds. Finally, according to the results of the JDeps tool, TP, FP, and FN are calculated, as shown in Figure 9. As can be argued from the previous definitions, if the TP value of a feature is high, it will be desirable, while high values of FP and FN will be undesirable.

In Figure 9, W-TP, W-FP, and W-FN represent the true positive, false positive, and false negative values of the weighted content similarity feature (the proposed new feature) for each component, respectively. On the other hand, TP, FP, and FN represent the values of the content similarity feature for each component, respectively. These values have been used to calculate the precision and recall criteria. F-Measure values provide a more accurate view by combining these two criteria to evaluate the features as calculated based on Equation (9).

$$F - Measure = \frac{2*precision*Recall}{precision+Recall} \qquad (9)$$

In this study, these features are analyzed by considering their thresholds. The threshold values for topology, content similarity, and weighted content

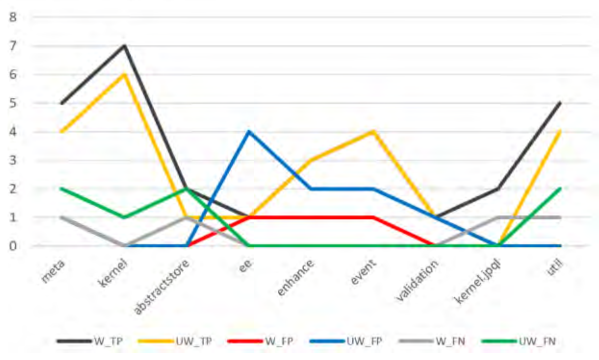


Figure 9. Comparing the confusion matrix values of the content similarity feature with the weighted content similarity

similarity features are set to 4, 0.1, and 0.1, respectively. The F-Measure values of topology and content similarity features in different software components are compared with the weighted content similarity feature in Figure 10 to understand the higher accuracy of our proposed feature. The average F-Measure for these features is calculated across different components; the average F-Measure values for topology, content similarity, and weighted content similarity features are 0,73, 0.76, and 0.88, respectively. According to the calculated average F-Measure values, it can be seen that the weighted content similarity feature has 0.15 higher accuracy than the topological feature and 0.12 higher accuracy than the unweighted content similarity feature in identifying the dependence between components. Therefore, the weighted content similarity feature has better accuracy in identifying component dependencies.

8. Discussion

Four features, namely structure, topology, content similarity, and weighted content similarity, have been investigated for different components of the open-source software, and their values have been calculated. The extent of the impact of each feature on the relationships between components has been examined and determined separately. By analyzing these values based on the presence or absence of relationships between components in this software version, it can be concluded that the impact of the topology, content similarity, and weighted content similarity features increases on the dependencies between components, respectively. In other words, the topological feature has the lowest F-Measure value, and the weighted content similarity feature has the highest F-Measure value. This increase in the content similarity feature impact is certainly due to the use of TF-IDF for weighting comment words in our proposed method. With this method, the weight of common words between components, which, despite their high repetition, are worthless for finding connections between them, is minimized.

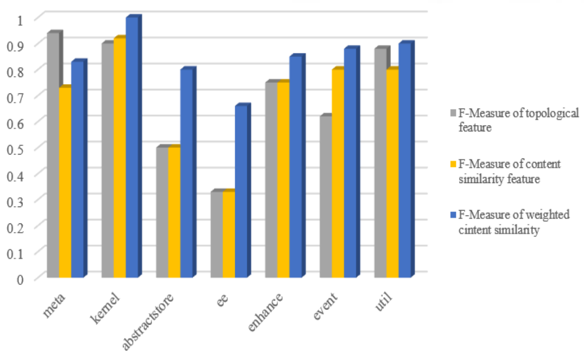


Figure 10. F-Measure of topological, content similarity, and weighted content similarity features

The results obtained from our research can be generalized to similar software systems because the selection of features is based on software engineering principles and the hemophilia principle, which is an important principle used in identifying dependencies in social networks. On the other hand, OpenJPA software was not designed by a programming team, but by the open-source community for the open-source community. Furthermore, the OpenJPA software is of acceptable quality due to its various versions, bugs fixed by the open-source community, and support by the Apache Software Foundation (ASF, a non-profit corporation) [39]. Therefore, by appropriately weighting each of the component-level features, the prediction of dependencies between components using machine learning methods in future software versions will be done accurately. Ultimately, better prediction of cyclic and hub-like dependency anti-patterns will be possible.

9. Conclusion

The present study introduces a new feature at the component level that affects component dependencies. The accuracy of this feature in detecting dependencies was compared with the topology and content similarity features presented in previous studies. To present a valid comparison, since precision and recall measures can lead to errors in evaluating the performance of a method, the F-Measure values for each feature have been calculated. Therefore, researchers in this field can have an error-free comparison of various components in the considered software.

These F-Measure values indicate that the weighted content similarity feature can be more effective than other features for detecting component dependencies. Future research can identify and introduce other effective features in component dependencies detection. Additionally, they can present new methods for predicting cyclic and hub-like dependency anti-patterns with high accuracy that use our proposed feature and other features simultaneously. By introducing various prediction methods based on feature values in recent years and applying them in different research fields, it is hoped that favorable results will also be achieved in the field of software anti-pattern prediction.

Declarations

Funding

This research did not receive any grant from funding agencies in the public, commercial, or non-profit sectors.

Authors' contributions

SK: Study design, acquisition of data, interpretation of the results, statistical analysis, drafting the manuscript;

M.RK: Study design, interpretation of the results, and revision of the manuscript.

Conflict of interest

The authors declare that no conflicts of interest exist.

References

[1] A. Avritzer, A. Janes, C. Trubiani, H. Rodrigues, Y. Cai, D. Menasche, and A. Oliveira, "Architecture and Performance Anti-patterns Correlation in Microservice Architectures," in *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*, Odense, Denmark, 2025, pp. 60-71, https://doi.org/10.1109/ICSA65012.2025.00016.

[2] A. Sekar, "Distributed systems patterns and anti-patterns: A comprehensive framework for scalable and reliable architectures," *World Journal of Advanced Engineering Technology and Sciences*, vol. 15, no. 1, pp. 667-676, 2025, https://doi.org/10.30574/wjaets,2025.15.1.0197.

[3] S. Kalhor, M.R. Keyvanpour, and A. Salajegheh, "A systematic review of refactoring opportunities by software anti-pattern detection," *Automated Software Engineering*, vol. 31, no. 2, p. 42, 2024. https://doi.org/10.1007/s10515-024-00443-y

[4] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. "Toward a catalogue of architectural bad smells," *In QoSA*, vol. 5581, pp. 146–162, Springer, 2009. https://doi.org/10.1007/978-3-642-02351-4_10

[5] R. C. Martin, *Clean architecture: A craftsman's guide to software structure and design*, Prentice Hall Press, 2017.

[6] F. A. Fontana, M. Camilli, D. Rendina, A. G. Taraboi, and C. Trubiani, "Impact of Architectural Smells on Software Performance: an Exploratory Study," *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, 2023, pp. 22-31, https://doi.org/10.1145/3593434.3593442

[7] F. Zhang, Z. Zhang, J. W. Keung, X. Tang, Z. Yang, X. Yu, and W. Hu, "Data Preparation for Deep Learning based Code Smell Detection: A Systematic Literature Review," *Journal of Systems and Software*, vol. 216, p. 112131, 2024, https://doi.org/10.1016/j.jss.2024.112131.

[8] S. Kalhor, M. R. Keyvanpour, and A. Salajegheh, "Experimental evaluation and comparison of anti-pattern detection tools by the gold standard," *In Proceedings of the 12th International Conference on Computer and Knowledge Engineering (ICCKE 2022)*, Ferdowsi University of Mashhad, Mashhad, Iran, 2022, pp. 361-366, https://doi.org/10.1109/ICCKE57176.2022.9960137.

[9] S. Kalhor, M. R. Keyvanpour, and A. Salajegheh, "Comparison of different automatic metric-based code smell detection methods," *In Proceedings of the 10th International Conference on Artificial Intelligence and Robotics-QICAR2024*, Qazvin Islamic Azad University, Feb. 29, 2024. http://393.demo-cd.ir/papers/r_117_240201155858.pdf

[10] C. Sammut and G. I. Webb, *Encyclopedia of Machine Learning*, Springer, Boston, 2010, https://doi.org/10.1007/978-0-387-30164-8_832

[11] S. Qaiser and R. Ali, "Text Mining: Use of TF-IDF to Examine the Relevance of Words to Documents," *International Journal of Computer Applications*, vol. 181. no. 1, pp. 25-29, 2018, https://doi.org/10.5120/ijca2018917395.

[12] https://www.Lattix.com/solutions/architecture-analysis

[13] http://www.hello2morrow.com/products/sonargraph

[14] https://docs.oracle.com/en/java/javase/11/tools/jdeps.html

[15] J. Andres Diaz-Pace, A. Tommasel, and D. Godoy, "Towards Anticipation of Architectural Smells using Link Prediction Techniques," *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Madrid, Spain, 2018, pp. 62-71, https://doi.org/10.1109/SCAM.2018.00015.

[16] A. Tommasel, "Applying social network analysis techniques to architectural smell prediction," *In Proceedings of the International Conference on Software Architecture Companion*, IEEE, 2019, pp. 254–261, https://tommantonela.github.io/icsa2019.pdf

[17] C. Catal, "Performance Evaluation Metrics for Software Fault Prediction Studies," Acta Polytechnica Hungarica, vol. 9, no. 4, 193-236, 2012. https://acta.uni-obuda.hu/Catal_36.pdf

[18] N. Moha, Y. G. Gu´eh´eneuc, L. Duchien, A. F. Le Meur, "Decor: a method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010, https://doi.org/10.1109/TSE.2009.50

[19] A. Alkhalid, M. Alshayeb, and S. Mahmoud, "Software refactoring at the package level using clustering techniques," *IET Software*, vol. 5, no. 3, pp. 276–284, 2011. https://doi.org/10.1049/iet-sen.2010.0070

[20] N. Shafiei and M. R. Keyvanpour, "Challenges Classification in Search-Based Refactoring," *In Proceedings of the 6th International Conference on Web Research (ICWR)*, Tehran, Iran, 2020, pp. 106-112, https://doi.org/10.1109/ICWR49608.2020.9122271.

[21] Z. Razani, and M. R. Keyvanpour, "SBSR Solution Evaluation: Methods and Challenges Classification," *In Proceedings of the 5th Conference on Knowledge-Based Engineering and Innovation (KBEI)*, Tehran, Iran, 2019, pp. 181-188, https://doi.org/10.1109/KBEI.2019.8734937.

[22] W. Mkaouer, M. Kessentini, A. Shaout, P. Kontchou, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using NSGA-III," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 3, pp. 1–45, 2015, https://doi.org/10.1145/2729974

[23] S. Vidal, W. Oizumi, A. Garcia, A.D. Pace, and C. Marcos, "Ranking architecturally critical agglomerations of code smells," *Science of Computer Programming*, vol. 182, pp. 64–85, 2019. https://doi.org/10.1016/j.scico.2019.07.003

[24] Y. Cai, and R. Kazman, "Software architecture health monitor," *In Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, ACM, New York, USA, 2016, pp. 18–21, https://doi.org/10.1145/2896935.2896940

[25] Y. Cai, H. Wang, S. Wong, and L. Wang, "Leveraging design rules to improve software architecture recovery," *In Proceedings of the 9th International ACM SIGSOFT conference on Quality of Software Architectures*, 2013, pp. 133-142. https://doi.org/10.1145/2465478.2465480

[26] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells," In Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture, IEEE, pp. 51–60, 2015, https://doi.org/10.1109/WICSA.2015.12

[27] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," *In Proceedings of the 38th IEEE/ACM International Conference on Software Engineering*, ACM, 2016, pp. 488–498, https://doi.org/10.1145/2884781.2884822

[28] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Architecture anti-patterns: Automatically detectable violations of design principles," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 1008–1028, 2019. https://doi.org/10.1109/TSE.2019.2910856

[29] M. Goldstein, and I. Segall, "Automatic and continuous software architecture validation," *In Proceedings of the 37th IEEE International Conference on Software Engineering,*

IEEE, 2015, vol. 2, pp. 59–68, https://doi.org/10.1109/ICSE.2015.135.

[30] T. Hübener, M. R. V. Chaudron, Y. Luo, P. Vallen, J. van der Kogel and T. Liefheid, “Automatic Anti-Pattern Detection in Microservice Architectures Based on Distributed Tracing,” *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Pittsburgh, PA, USA, 2022, pp. 75-76, https://doi.org/10.1145/3510457.3513066.

[31] A. Tommasel, “Applying social network analysis techniques to architectural smell prediction,” *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, Hamburg, Germany, 2019, pp. 254-261, https://doi.org/10.1109/ICSA-C.2019.00053

[32] L. Hochstein, and M. Lindvall. “Combating architectural degeneration: A survey,” *Inf. Softw. Technol*., vol. 47, no. 10, pp. 643–656, July 2005, https://doi.org/10.1016/j.infsof.2004.11.005

[33] R. Marinescu. “Assessing technical debt by identifying design flaws in software systems,” *IBM Journal of Research and Dev*, vol. 56, no. 5, pp. 528-540, 2012. https://doi.org/10.1147/JRD.2012.2204512.

[34] W. Tracz, “Refactoring for software design smells: Managing technical debt by Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma,” *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 6, p. 36, 2015. https://doi.org/10.1145/2830719.2830739

[35] F. Arcelli Fontana, I. Pigazzini, R. Roveda, D.A. Tamburri, M. Zanoni, and E. Di Nitto. “Arcan: A tool for architectural smells detection,” *In 2017 IEEE ICSA Workshops 2017, Gothenburg*, Sweden, April 5-7, 2017, pp. 282–285, https://doi.org/10.1109/ICSAW.2017.16

[36] M. McPherson, L. Smith-Lovin, and J.M. Cook. “Birds of a Feather: Homophily in social networks”. *Annu. Rev. Sociol*, vol. 27, no. 1, pp. 415–444, 2001. https://doi.org/10.1146/annurev.soc.27.1.415

[37] G. Salton, and M. J. McGill. “Introduction to Modern Information Retrieval”. McGraw-Hill, Inc., New York, NY, USA, 1986.

[38] W. Abdulazeez Qader, M. M. Ameen and B. L. Ahmed, “An Overview of Bag of Words: Importance, Implementation, Applications, and Challenges,” *2019 International Engineering Conference (IEC)*, Erbil, IRAQ, 2019, pp. 200-204. https://doi.org/10.1109/IEC47844.2019.8950616.

[39] https://httpd.apache.org

[40] L. D. Minh, D. Link, A. Shahbazian, and N. Medvidović. “An Empirical Study of Architectural Decay in Open-Source Software,” *In Proceedings of the IEEE International Conference on Software Architecture (ICSA)*, Seattle, WA, USA, 2018, pp. 176-17609, https://doi.org/10.1109/ICSA.2018.00027

[41] http://www.python.org

[42] M. R. Keyvanpour, Z. Zandian and F. Morsali, “Software Re-Modularization Method Based on Many-Objective Function”, *International Journal of Information and Communication Technology Research*, vol. 16, pp. 28-41,2024. https://doi.org/10.61186/itrc.16.1.28

[43] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, “Using automatic clustering to produce high-level system organizations of source code,” *in Proceedings. 6th International Workshop on Program Comprehension. IWPC’98 (Cat. No. 98TB100242)*, Ischia, Italy, 1998, pp. 45–52. https://doi.org/10.1109/WPC.1998.693283

[44] J. Wu, A. E. Hassan, and R. C. Holt, “Comparison of clustering algorithms in the context of software evolution,” in 21st IEEE International Conference on Software Maintenance (ICSM’05), Budapest, Hungary, 2005, pp. 525–535. https://doi.org/10.1109/ICSM.2005.31

[45] ThoughtWorks, “Micro frontends,” ThoughtWorks Technology Radar, 2016. [Online]. https://www.thoughtworks.com/pt-br/radar/techniques/micro-frontends

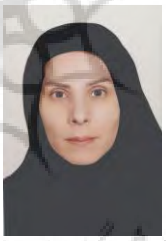
[46] N. Silva, E. Rodrigues, and T. Conte, "A Catalog of Micro Frontends Anti-patterns," *In Proceedings of the 47th International Conference on Software Engineering (ICSE)*, IEEE/ACM, Ottawa, ON, Canada, 2025, pp. 616-616, https://doi.org/10.1109/ICSE55347.2025.00079.

[47] M. D. Luca, S. D. Meglio, A. R. Fasolino, L. Libero Lucio Starace, and P. Tramontana, “Automatic Assessment of Architectural Anti-patterns and Code Smells in Student Software Projects,” *In Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024)*, June 18–21, 2024, Salerno, Italy. ACM, New York, NY, USA, pp. 565-569, https://doi.org/10.1145/3661167.3661290

[48] ArchUnit documentation, available at https://www.archunit.org/

[49] SonarQube documentation, available at https://docs.sonarsource.com/sonarqube/latest/

[50] R .Jolak, S. Karlsson, and F. Dobslaw, “An empirical investigation of the impact of architectural smells on software maintainability,” Journal of Systems and Software, vol. 225, 112382, 2025, https://doi.org/10.1016/j.jss.2025.112382



Somayeh Kalhor is a faculty member at Islamic Azad University, Qazvin Branch. She is researching for her doctoral thesis at Islamic Azad University, South Tehran Branch, Iran, on software anti-pattern detection methods and has presented papers in scientific journals and conferences.

Mohammad Reza Keyvanpour is a Professor at Alzahra University, Tehran, Iran. He received his B.Sc. degree in software engineering from Iran University of Science & Technology, Tehran, Iran. He received his M.Sc. and Ph.D degrees in software engineering from Tarbiat Modares, Tehran, Iran. His research interests include Software Engineering and Data Mining.