TÉCNICO LISBOA

Code Vectorization and Sequence of Accesses Strategies for Monolith Microservices Identification

Vasco Faustino de Faria

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. António Rito Silva

Examination Committee

Chairperson: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Supervisor: Prof. António Rito Silva
Member of the Committee: Prof. Filipe Alexandre Pais de Figueiredo Correia

October, 2022

Acknowledgments

First of all, I would like to thank my supervisor, Prof. António Rito Silva, for his commitment to his students, for all the support provided along the way, and for never doubting my capacities.

Also, this journey would not be possible without the hard work of my parents, who gave me this opportunity, invested a lot in my academic training, and have always motivated me to focus on my studies, which has allowed me to get this far.

I would also like to thank my girlfriend who has been very supportive and motivated me in this most demanding phase of my academic journey.

Lastly, I would like to thank all my family and friends for all the support along the way.

Institutional Acknowledgment

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) through projects UIDB/50021/2020 (INESC-ID) and PTDC/CCI-COM/2156/2021 (DACOMICO).

Abstract

Migrating a monolith application into a microservices architecture can benefit from automation methods, which speed up the migration and improve the decomposition results. One of the current approaches that guide software architects on the migration is to group monolith domain entities into microservices, using the sequences of accesses of the monolith functionalities to the domain entities. In this paper, we enrich the sequence of accesses solution by applying code vectorization to the monolith, using the Code2Vec neural network model. In the related work, we go through an evolution of the lexical analysis for automating monolith decompositions, from code tokenization approaches to the use of machine learning models. We apply Code2Vec to vectorize the monolith functionalities. We propose two strategies to represent a functionality, one by aggregating its call graph methods vectors, and the other by extending the sequence of accesses approach with vectorization of the accessed entities. To evaluate these strategies, we compare the proposed strategies with the sequence of accesses strategy, and an existing approach that use class vectorization. We run all these strategies over a large set of codebases, and then compare the results of their decompositions in terms of cohesion, coupling, and complexity.

Keywords

Monolith, Microservices, Microservices Identification, Static Analysis, Machine Learning, Architecture Migration

Resumo

A migração de uma aplicação monólita para uma arquitetura de microserviços pode beneficiar bastante da sua automatização, acelerando a migração e melhorando os resultados da decomposição. Uma das abordagens atuais para ajudar os arquitetos de software a realizar a migração consiste em agrupar as entidades de domínio do monólito para o decompor em micro-serviços, utilizando as sequências de acessos às entidades do domínio pelas funcionalidades do monólito. Neste trabalho, enriquecemos a solução da sequência de acessos utilizando uma análise lexical do código monólito baseada na vetorização do código utilizando o modelo de rede neural Code2Vec. No trabalho relacionado, descrevemos a evolução da análise léxica para a automatização das decomposições monolíticas, desde abordagens que utilizam apeans os tokens presentes no código até à utilização de modelos de aprendizagem de máquina. O Code2Vec é utilizado para vectorizar as funcionalidades do monólito, de modo a que a distância entre esses vectores possa ser utilizada para identificar monólitos. São propostas duas estratégias para representar uma funcionalidade, uma agregando os vetores dos métodos utilizados na sua execução, e a outra alargando a abordagem de sequência de acessos com entidades vetorizadas. Para avaliar estas estratégias, comparamos as estratégias propostas com a estratégia de sequência de acessos, e uma abordagem de decomposição com a perspetiva de classes que também utiliza o Code2Vec. Executamos todas estas estratégias sobre um grande conjunto de repositórios, e depois comparamos os resultados das suas decomposições em termos de coesão, coupling, e complexidade.

Palavras Chave

Monólito; Microserviços; Identificação de Microserviços, Análise Estática; Apendizagem Máquina; Migração de Arquitetura

Contents

List of Figures

List of Tables

Listings

Acronyms

AST	Abstract Syntax Tree
CV	Class Vectorization
DTO	Data Transfer Object
EV	Entity Vectorization
FVCG	Functionality Vectorization by Call Graph
FVSA	Functionality Vectorization by Sequences of Accesses
ML	Machine Learning
MST	Minimum Spanning Tree
NLP	Natural Language Processor
OLS	Ordinary Least Squares
ORM	Object-Relational Mapper
SA	Sequences of Accesses

1

Introduction

Contents

As microservices architectures prove their value over monoliths, an increasing number of applications are decomposing their architecture into microservices, which provides significant benefits in terms of scalability and maintainability.

Despite these advantages, and depending on the size and complexity of a codebase, this migration process can become very complex and expensive, making its automation a great option to save time and effort.

1.1 Context

Abdellatif et al. [2] present, in a survey on the modernization approaches of legacy systems, several migration strategies are classified by their inputs, processes, and outputs. These approaches work on a codebase's model obtained by applying collection tools, which are static if they do not require the system execution, or dynamic otherwise.

However, according to this study, it can be observed that the majority of the approaches for a system migration perform a static analysis of the source code, followed by a clustering algorithm in conjunction with similarity metrics that differ depending on the strategy.

In what concerns the collection part, the static analysis requires an effort that is not generalizable, because it depends on the particular programming languages and frameworks used in the monolith implementation.

1.2 Problem

The goal of this paper is to study whether an approach that does not require a complex data collection can achieve good decompositions. This would remove some bottlenecks of previous work since the collector wouldn't be restricted to a particular programming language, web development stack, and object-relational mapper.

This approach is inspired by Al-Debagy and Martinek's work [3], which analyze the monolith code like a Natural Language Processing problem (Natural Language Processor (NLP)). They use a neural network model called *Code2Vec* for microservices identification. This model takes advantage of a method abstract syntax tree (Abstract Syntax Tree (AST)) and the lexical interpretation of its tokens to calculate a numerical vector, representing as much information about that method as possible. With this tool, they generate vectors associated with the monolith codebase classes and measure the quality of the decomposition in terms of cohesion and coupling metrics.

On the other hand, in our previous work [4], the monolith codebase analysis is done from a perspective of the monolith functionalities, and the generation of a decomposition that minimizes the number of

distributed transactions that are required to implement a functionality. The results of this approach are measured in terms of the transactional complexity of the candidate decompositions.

1.3 Research Questions and Contributions

In this paper, we leverage on [3, 4], by integrating their perspective to verify, using a larger number of codebases, whether:

1. The use of *Code2Vec* with the functionality perspective provides better results than sequences of accesses in [4];
2. The application of the functionality perspective provides better results than Al-Debagy and Martinek [3];
3. The input parameters of the proposed strategies impact the results of the evaluation metrics.

The proposed solution starts with a Data Collection phase, where a new collector is used to extract all the methods of a codebase, along with all their information (package, class, type, source code, and method calls). During this phase, the *Code2Vec* model is used to generate each method's respective vector. After, we test different approaches to the extracted vectors in order to derive the functionalities vectors.

For evaluation, we apply the different strategies to a large set of codebases, and then compare the results using cohesion, coupling, and complexity metrics.

1.4 Organization of the Document

After this section, Chapter 2 discusses work related to the application of machine learning techniques in software migration, followed by Chapter 3 which explains the proposed solutions to improve the decomposition process, and Chapter 4 with the implementation process and the overall architecture of the solution. Chapter 5 evaluates and compares the new approaches with previous work, and finally, Chapter 6 consists of the conclusions of this work.

2

Related Work

Since the emergence of microservices architectures, migrating monoliths to these architectures has been an increasingly active topic [2].

There are approaches [4–6] that use the monolith functionalities sequences of accesses to the monolith domain entities to feed an aggregation algorithm that proposes candidate decomposition for microservices. These approaches can use static analysis of the monolith code, e.g. [4], or dynamic execution of the monolith to collect the sequences of accesses. In [7] it is compared the use of static and dynamic collection in several monolith systems. They conclude that, while in static analysis the data collection needs to be adapted to each programming language or framework, which requires tool adaption effort for each new programming language of full-stack technology, in a dynamic collection of data shown to have worse coverage, though generating a huge amount of data. Based on these results, our research intends to explore the use of lexical analysis, a form of static analysis that requires less effort because it is more language and technology independent.

On the other hand, several approaches for the identification of microservices and other usecases for code aggregation use lexical analysis.

Hammad and Banat [8] propose a technique that utilizes the K-Means clustering algorithm [9] to group a set of classes into packages, where the similarity measure presented consists of how many relevant tokens two classes have in common.

Their methodology consists of the following phases:

1. **Tokenization:** Where all tokens are extracted based on the operators of the programming language.

2. **Extraction of Identifiers:** Filter reserved words (get, set, i, j, etc.), digits, and operators to keep only the relevant tokens such as the names of used libraries, attributes, variables, and methods.

3. **Generation of Term Matrix:** Generate a matrix of Token / Class, in which the cell value depends on the existence of the token (1 if exists else 0).

4. **Clustering:** K-Means clustering with k set to 5 where the distance is the percentage of tokens the classes have in common with each other.

The results of this method are related to the size of each package, packages with a large number of classes have better chances to be automatically grouped, while small packages are generally poorly grouped, also this approach didn't achieve good results in terms of modularity, because the tokens must be identical in order to find a similarity between two parts of the code, which ignores all words that belong to the same semantic or lexical field.

Mazlami et al. [10] present three formal coupling strategies to generate a weighted graph from the meta-information of a monolithic codebase.

Their approach starts by extracting all the class files that constitute the monolith, the developers' team, and the history of modifications (commits) to the project from the version control system used to implement the given codebase.

Then, the extracted information is used to build a graph where the vertices represent classes, and the edges result from a weight function. They present three different weight functions, where each one is a different strategy to represent how strong the coupling between the two classes is.

1. **Logical Coupling Strategy**: Consists of assigning a weight of one to edges between classes that belong to the same modification, or otherwise a weight of zero.

2. **Semantic Coupling Strategy**: Couples classes that contain the same tokens, and to each edge assign a weight that depends on how many tokens they have in common and their frequency.

3. **Contributor Coupling Strategy**: The weight of each edge is determined by the developers who modified the two classes that are connected by that edge.

Finally, the generated graph is cut into subgraphs using the Minimum Spanning Tree (Minimum Spanning Tree (MST)) clustering algorithm, where each one of them is a set of classes that represent a microservice.

The *Semantic Coupling Strategy* follows the same logic of the previously mentioned approach [8], based on coupling two classes containing the same tokens but considering each one's frequency. Although this strategy presents a worse execution time when compared with their other approaches, this one show better results in what concerns the team size reduction and the average domain redundancy.

Brito et al. [11] use topic modeling to identify services according to domain terms (words with higher probabilities indicate a possible good topic).

In the first part of their approach, all the relevant tokens and structural dependencies are extracted from the Abstract Syntax Tree of the Monolith source code. They also refer that this step could be easier with a pure Natural Language Processor (NLP), but the results would be worse.

Next, they use Latent Dirichlet Allocation (LDA), a generative probabilistic unsupervised model, to categorize documents by topics. In this context, a document represents a class as a collection of words, and these words are the respective lexical items that compose variables and methods names used in each class source file.

Then, they create an edge-weighted graph G by combining the structural dependencies and topic distribution. The vertices of G correspond to each class/module, and each edge shows how strong the association is based on the topic distribution.

Finally, the Louvain clustering algorithm is used to group the classes/modules in order to define the microservices.

This topic modeling approach is also agnostic of the development stack, but the results depend on an optimal lexical token extraction, using specific parsers for each language to extract and process the ASTs as input to the model, which means increasing its complexity. As a result of their work, the model shows good cohesion values of the identified microservices with the trade-off of generating a high number of clusters in order to achieve good values in a metric that evaluates whether microservices follow the Single Responsibility principle [12].

Nowadays, there are not many approaches besides clustering when it comes to machine learning techniques to decompose monolith applications into microservices. However, the use of NLPs has been increasing due to their significant progress in performing lexical analyses, making the Data Collection phase easier and more generic.

Ma et al. [13] propose a solution based on *Word2Vec* [14], a widely-used machine learning method in natural language processing, to match existing microservices to new requirements. Their approach only works for applications where scenarios are written in a common language describing the features of the target system and that already follow a microservices architecture since their goal is to discover where to place new requirements.

During the data collection phase, they gather and preprocess the OpenAPI Specifications (OAS [1]) and the documents containing the scenarios written in Gherkin syntax. After some data preparation techniques, the pre-trained *Word2Vec* model is used to construct the corresponding BDD vectors of each scenario.

These vectors are then used to calculate the similarity between scenarios and each microservice. If the similarity score is bigger than a given threshold, the respective microservice is qualified as one of the candidates for that requirement. And finally, these candidates are arranged according to their scores.

Leveraging on the *Word2Vec* [14] work, Alon et al. [1] created *Code2Vec*, a neural network model trained to represent methods as fixed-length numerical vectors, also called code embeddings.

Al-Debagy and Martinek [3] propose an approach to decompose a monolith application into microservices using *Code2Vec* [1] to extract the methods' code embeddings.

First, they extract all the methods, and the respective code, from the monolith application and convert them into code embeddings, snippets of code characterized as a vector-based representation, using the *Code2Vec* model.

Using these vectors, they define a class embedding as the aggregation of its methods' embeddings. After testing, they found that the mean is the most suitable aggregation function to define a class embedding.

Finally, they have each class represented as a vector and by using a clustering algorithm they identify the microservices candidates, a group of semantically similar classes together.

[1] https://swagger.io/

The results of this novel approach show high cohesion values since all the semantically similar classes are grouped in a microservice, making this solution achieve even better results than the other approaches considered in their evaluation.

Overall, although there is some work on the use of *Code2Vec* for the identification of microservices in a monolith, it does not follow an approach where the data collected from the monolith is based on the functionalities accesses to domain entities. Additionally, there is a lack of studies that compare the approaches for a large number of codebases, using different quality metrics.

There are other approaches that associate Machine Learning (ML) techniques to the logs of the monolith execution. Taibi and Systä [15] approach the decomposition problem based on runtime behavior instead of static dependencies, identifying microservices candidates based on process mining performed on log files collected with Elastic search[2]. These log files record the user activities conducted from the user interface, the access to any system entry point, and information about each class and method visited. With this approach, not only does it help the architect to obtain decompositions with a low level of coupling, but it can also identify which services are used the most, a piece of important information for the decomposition process.

[2] https://www.elastic.co/

3

Solution

Contents

After keeping up with both types of code analysis (static and dynamic) and exploring machine learning techniques to automate the monolith decomposition process, we choose to follow a static and lexical analysis recurring to a machine learning model (*Code2Vec* [1]) that vectorizes code snippets. With this model, we apply different similarity metrics to cluster domain entities and functionalities based on their respective vectors. The distance between these vectors represents how close two pieces of code are lexically and semantically.

3.1 Code2Vec

Code2Vec [1] is a neural network model trained to represent methods as fixed-length numerical vectors, also called code embeddings. In machine learning, an embedding is a low-dimensional vector that represents high-dimensional data preserving the most information possible. Although the model is designed for method naming, the learned code embeddings can be used for several other applications.

The first stage of *Code2Vec* consists of transforming a code snippet into abstract syntax tree (AST) paths since it improves scalability while training the model avoiding the costs of learning the language syntax itself.

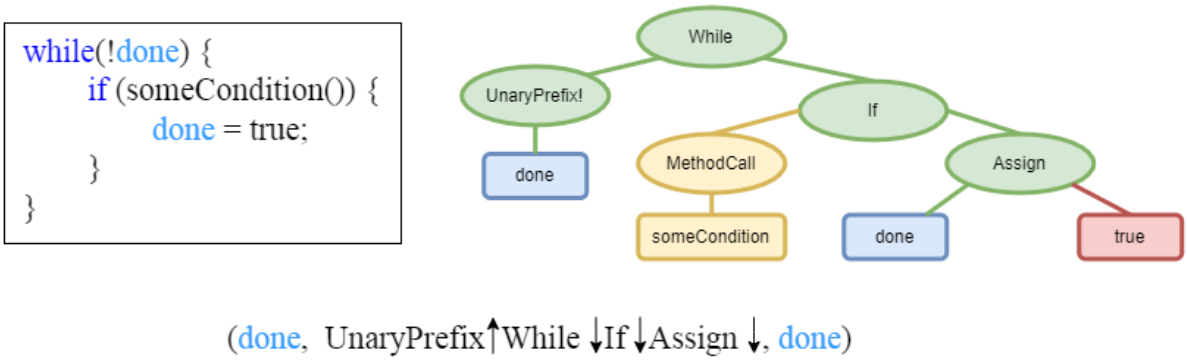


Figure 3.1: Extraction of the AST paths from *Code2Vec* paper [1] example.

Following the extraction of the AST paths (Figure 3.1), each path is mapped into a three-value tuple composed of the path's start node, intermediate expressions, and the final node. Then, each part of the tuple is converted to a real-valued representation, creating a three-dimension numerical vector known as a context vector acting as input to the path-attention network.

A neural attention network architecture is used to overcome the data sparsity problem of similar methods having different ASTs paths. With this attention mechanism, the model also learns the importance of each path, applying higher weights to the most important ones.

By applying the learned weights and the hyperbolic tangent function on the input vectors (Figure 3.2) the code embedding is computed using the attention weights to calculate a weighted average of all the

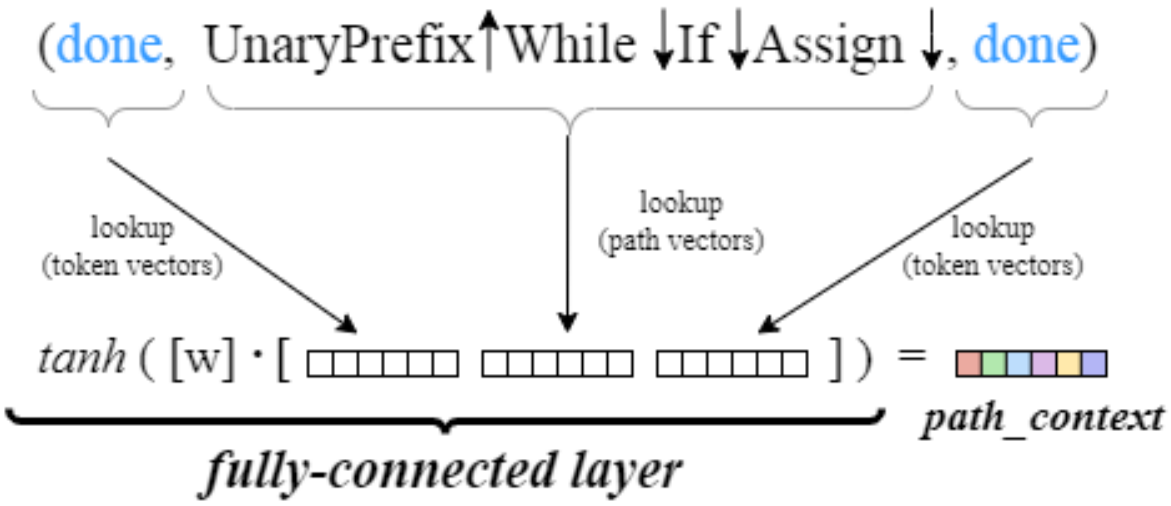


Figure 3.2: Path context vector representation from *Code2Vec* paper [1].

combined context vectors.

3.2 Data Collection

The first step of this approach consists of extracting all the necessary information from the monolith codebase and preparing it. This is done using the JavaParser library[1], which is a popular static analysis tool used to parse and modify java code by generating an interactive abstract syntax tree and providing a symbol resolution module. JavaParser also provides a type resolution module (symbol-solver) that can combine different type solvers to increase the capability of solving complex references like superclass methods.

The relevant type solvers for this work are the `JavaParserTypeSolver`, which given the source folder of the codebase, looks for the java files inferring its types and packages, and the `ReflectionTypeSolver`, which recognizes the Java language base types like `java.lang.Object`. There is also a `JarTypeSolver` that can be useful for codebases with embedded jar modules.

For the data collection, we explore all the codebase files, recurring to the JavaParser type solvers. For each java file, our parser starts by identifying the package, the class/interface name, the annotations, and all the present methods as well as checking if the class extends another.

Every time the parser founds a new method, its respective body is converted into a code embedding by the *Code2Vec* model, which we save along with the method signature. Also, inside each method body, the parser looks for all methods invocations' and tries to solve their signature using the type solvers. If

[1] https://javaparser.org/

the invoked method belongs to external libraries of the codebase, those invocations are discarded.

Since the evaluation is applied to monoliths implemented using Spring-Boot and an Object-Relational Mapper (Object-Relational Mapper (ORM)) each code embedding is characterized in terms of Spring-Boot architectural elements: Controller, Entity, Service, Repository, and Configuration classes. This categorization is used to verify whether some parts of the monolith code can provide more accurate results, and to identify the starting point of each functionality (Controller) and what are the monolith persistent domain entities (Entity).

3.3 Functionality Vectorization Strategies

We propose various functionality vectorization strategies to represent a functionality as an embedding by using the functionality call graph, or the functionality sequence of accesses to domain entities. The purpose of these strategies is to represent each microservice as a set of functionalities and thus understand which functionalities should be implemented in the same microservice.

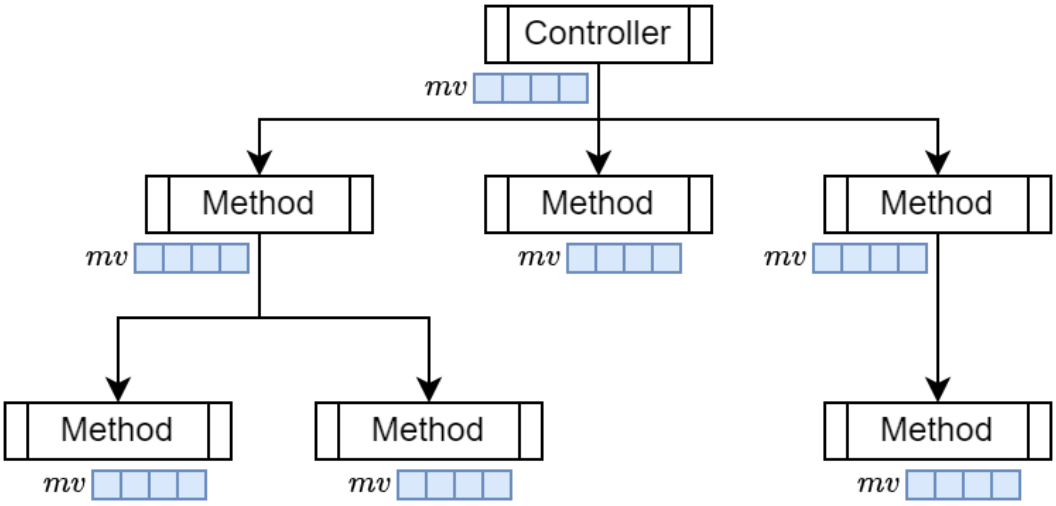


Figure 3.3: Extraction of a functionality call graph vector

Figure 3.3 presents the **Functionality Vectorization by Call Graph** (Functionality Vectorization by Call Graph (FVCG)) strategy, which represents each functionality as the call graph of its methods invocations, where the first method is the (Spring-Boot) controller where the functionality starts executing. By traversing a method call graph it is possible to reach loops, so to overcome this problem, a maximum depth parameter on the call graph is considered to compute the vector.

After discovering all the methods and the respective code embeddings, represented in Figure 3.3 by the mv vectors, that belong to the call graph of a functionality for a given depth, we apply the mean weighted function to those embeddings in order to achieve the functionality representing embedding.

The method annotations are used to infer each method type.

The following weights for method types are considered:

- $\underline{w_c}$: The controllers weight;
- $\underline{w_s}$: The services weight;
- $\underline{w_e}$: The entities weight;
- $\underline{w_i}$: The remaining methods (e.g., auxiliary or unclassified methods) weight, which will be referred as intermediate.

The weights are positive values that should sum 100 ($w_c + w_s + w_e + w_i = 100$).

$$cgv(f) = \frac{\sum_{mv \in f.cg(d).C} w_c \times mv + \sum_{mv \in f.cg(d).S} w_s \times mv + \sum_{mv \in f.cg(d).I} w_i \times mv + \sum_{mv \in f.cg(d).E} w_e \times mv}{\sum_{mv \in f.cg(d).C} w_c + \sum_{mv \in f.cg(d).S} w_s + \sum_{mv \in f.cg(d).I} w_i + \sum_{mv \in f.cg(d).E} w_e} \tag{3.1}$$

The vector is computed according to equation 3.1, where $f.cg(d)$ denotes the functionality (f) call graph, generated with depth d, due to possible recursive invocations, and $.C, .S, .I, .E$, denote, the call graph nodes that are of type, respectively, controller, service, intermediate and entity. Note that, additionally to the weight parameters, d parameter on the call graph depth determines the number of method vectors to consider. The purpose of these parameters is to study their impact on the quality of the result, and how they affect the evaluation metrics results. This study will help to understand the level of computational effort required in the construction of the vectors. For instance, if vectors computed using low depth provide good results, it will significantly reduce the computational effort. On the other hand, if the weights are irrelevant, the data collector will not need to recognize the type of each method, being a positive aspect to make the collector framework and architecture agnostic.

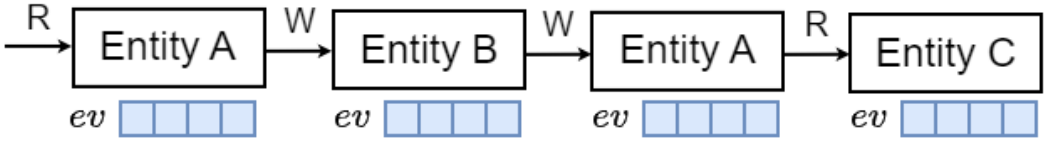


Figure 3.4: Extraction of a functionality sequence of accesses vector

Figure 3.4 presents the **Functionality Vectorization by Sequences of Accesses** (Functionality Vectorization by Sequences of Accesses (FVSA)) strategy, which represents each functionality as the sequence of its accesses to domain entities, where read accesses are distinguished from write accesses. It uses the sequences of accesses done by a functionality, and associates to each access the embedded vector of the accesses entity, ev. The entity embedded vector is computed by first identifying all the methods related to each entity, and then calculating the mean of that method's embeddings.

In order to represent entities as the mean of its methods' embeddings, the empty classes wouldn't have any embedding, but actually, this type of classes extends some other class or use annotations to generate their methods in compile time. This problem makes this solution not compatible with libraries like Lombok[2]. To mitigate this problem, we include inheritance in each class embedding by using all the top hierarchy classes' methods in the aggregation function.

$$sav(f) = \frac{\sum_{ev \in f.sa.R} w_r \times ev + \sum_{ev \in f.sa.W} w_w \times ev}{\sum_{ev \in f.sa.R} w_r + \sum_{ev \in f.sa.W} w_w} \tag{3.2}$$

Having functionality sequences of accesses and the entities' embeddings, the functionality embedding is the weighted average of the entities' embeddings of all the entities possibly accessed during the functionality execution, as presented in equation 3.2. In the equation, the entities read by functionality f in its sequence of accesses are denoted by $f.sa.R$, while $f.sa.W$ denotes the entities written. The parameters w_r and w_w, represent, respectively, the weight associated with the type of access. The weight values are positive and should sum to 100. Note that, as in the previous vectorization, the parameters will be used to assess the impact of distinguishing reads from write accesses in the quality of the decomposition.

3.4 Strategy Comparison

The strategy by Al-Debagy and Martinek [3] represents a microservice as a set of classes. They use class vectorization (Class Vectorization (CV)), where each class has an embedding calculated as the mean of its methods embeddings, a method already applied in the FVSA strategy.

Nevertheless, there are approaches where microservices are represented by monolith domain entities, instead of their classes, to highlight that the main aspect of a microservice is the independence of its database from other microservices databases. Therefore, we use another strategy adapted from the CV strategy in which, rather than representing a microservice as a set of classes, it is represented as a set of entities. The **Entity Vectorization** (Entity Vectorization (EV)) strategy only considers the classes in CV strategy which are entities.

There are four similarity measures based on the sequences of access [16]. They aggregate the monolith domain entities which are accessed by the same functionalities. The main idea behind these measures is that in a microservices architecture it is necessary to minimize the number of distributed transactions. Therefore, by having all the domain entities that are accessed by a functionality in the same cluster, the functionality can execute as a single transaction. These similarity measures represent the distance between two domain entities by using the sequences of accesses strategy (Sequences

[2] https://projectlombok.org/

of Accesses (SA)) of the functionalities that access them. Therefore, each of the similarity measures between entities e_i and e_j are defined as the following:

1. *Access*: Given a set of functionalities that access, both read or write, entity e_i, is the percentage of those who also access entity e_j.
2. *Read*: Given a set of functionalities that read entity e_i, is the percentage of those who also read entity e_j.
3. *Write*: Given a set of functionalities that write entity e_i, is the percentage of those who also write entity e_j.
4. *Sequence*: The percentage of the number of consecutive accesses to e_i and e_j entities over the maximum number of consecutive accesses for two domain entities.

Note that these measures, except the sequence, are not symmetric.

The SA strategy uses the four similarity measures by assigning weights to each one of them, such that their sum should be 100.

The strategies produce three different types of decompositions. SA and EV strategies generate clusters of entities, FVCG and FVSA strategies clusters of functionalities, and the CV strategy clusters of classes. Therefore, to compare the results, it is necessary to convert a decomposition type into the other. Since the metrics to be used in the evaluation are defined for decompositions of clusters of domain entities, the decompositions are converted into decompositions with clusters of entities.

To convert a cluster of classes into an entity's clusters, it is only necessary to remove all the non-entity classes from the clusters, which can lead to empty clusters and so we discard those clusters.

The functionalities clusters are converted into clusters of entities by counting the functionalities entity accesses present in each cluster. This is, for each domain entity's access by a functionality of a given cluster, the probability of that entity belonging to that respective cluster increases. Then, for each domain entity, we look for the cluster that accesses it the most to assign the entity to that respective cluster. Afterward, since this conversion may also result in empty clusters, those are discarded.

4

Architecture

Contents

4.1 Mono2Micro Structure

The proposed solution was implemented on top of the Mono2Micro tool architecture, which is composed of a set of static and dynamic collectors, a user-friendly interface, a *Spring Boot*[1] *Server*, and a *Fast API*[2] *Server*.

The collectors are responsible for collecting all the necessary data, either with static or dynamic analysis, from a given codebase used to identify possible decompositions. Then, the collected data is used to register a new codebase in the system, where this step is already done in the user interface, which requests the *Spring Boot Server* to persist the codebase in the file system.

Once a codebase is created it's possible to create a dendrogram that represents the distances between the monolith elements to be considered in the decomposition. For this operation, the *Spring Boot Server* connects with the *Fast API Server* to request the creation of the dendrogram.

With the codebase represented as a dendrogram, it becomes possible to cut it in order to generate a candidate decomposition of the codebase into microservices, where each microservice is represented as a set of entities.

For analyzing each decomposition, the user interface provides functionalities to visualize and manipulate that data. It is possible to visualize each decomposition from different views, retrieve the metrics values of the decomposition under analysis, and perform several operations over the decomposition, like merging, splitting, or transferring entities between clusters.

The Mono2Micro tool also provides a microservice analysis feature to compare two different candidate decompositions, a refactorization tool that automatically computes microservice saga orchestrations, and an analyzer module that can be used to explore all the variable parameters from a strategy in order to find which parameters result in better decompositions.

4.2 Architecture Design

Figure 4.1 represents a partial architecture of the Mono2Micro tool with only the relevant components for this work and the updates that had to be made.

To integrate the proposed solution into the Mono2Micro tool, it was necessary to create a new collector called *javaparser-callgraph*, to perform a static analysis over a given codebase. In addition to this collector, the *spoon-callgraph* collector present in the tool was also used to extract the sequences of accesses to domain entities from a codebase.

The proposed strategies logic was implemented in the *Spring Boot Server*, and it was also necessary to extend the *Fast API Server* because the new strategies generate functionalities' clusters instead of

[1] https://spring.io/
[2] https://fastapi.tiangolo.com/

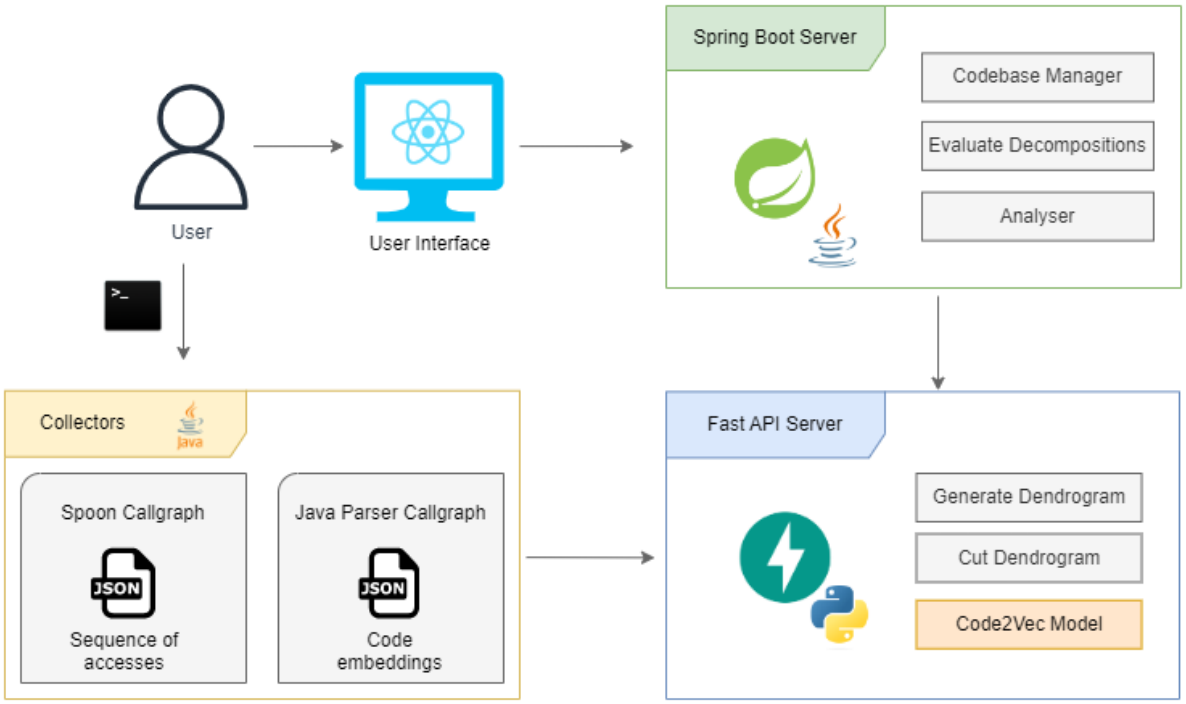


Figure 4.1: Partial architecture of Mono2Micro with the relevant components for this work.

entities' clusters.

Since the *Code2Vec* model has a similar technology stack as the *Fast API Server*, the programming language, and some libraries, it was decided to put the model into this server, exposing its functionality as a new endpoint.

4.2.1 Data Extraction

The first step of the implementation is to extract all the information from a codebase, necessary to execute the decomposition strategies. For each codebase, it is necessary to generate two files, one with the sequences of accesses to the domain entities, and another with all the categorized methods of the codebase and their embeddings.

The sequences of accesses to the entities are extracted through the *spoon-callgraph*, an existing collector in the Mono2Micro tool developed in previous work. To extract the code embeddings it was necessary to create a new parser, called *javaparser-callgraph*.

The *javaparser-callgraph* represents each codebase as a set of methods (Figure 4.2), grouped by packages and classes, that are represented with their signature, which contains the name of the method and its parameters. Besides the signature, each method is also represented by its type, a list of all methods invoked there, and a numerical vector generated by the *Code2Vec* model to represent that

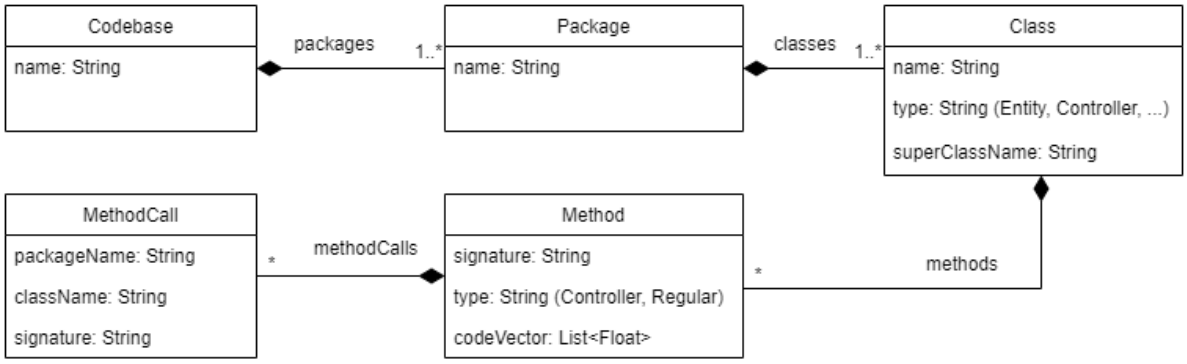


Figure 4.2: Representation of a codebase during the methods extraction with javaparser-callgraph

method.

Given the path for the codebase to analyze, the collector starts to search every file with the *.java* extension. Then, for each file, using the `StaticJavaParser` of the `JavaParser` library, the collector searches for the package, and the class declarations to retrieve the package name, the class name, the class annotations, and the extended superclass if any.

With the class annotations from the Spring Framework, it's possible to infer the class type, which can be cataloged as Controller, Service, Entity, Repository, or Configuration.

Next, the collector looks for methods and constructors declarations present in the class, again using the `StaticJavaParser`, to retrieve each method signature, the body, the method annotations, and the method call expressions.

The method annotations are used to infer if the presented method is a Controller method, by checking if the method has any of the Spring Framework Controller annotations.

The body of the methods is used to invoke the *Code2Vec* model service present in the *Fast API Server*, which generates the respective code vector for that code snippet.

Finally, to infer the methods signature, class, and package of the invoked methods through the method call expressions, it's necessary to use the `JavaParser` library type solvers to infer from each class and package the invoked methods belong to. The chosen type solver for this was the `CombinedTypeSolver`, which can aggregate multiple type solvers, making it possible to use the `ReflectionTypeSolver`, which uses the java reflection properties, and the `JavaParserTypeSolver`, which operates through the AST of the codebase, together to make a better type resolution.

For the constructor methods, the process was practically the same, with just the nuance that the *Code2Vec* doesn't accept constructor-like methods. To overcome this issue, the constructors were converted into regular methods by adding the `void` data type before the constructor name and replacing the `super` token inside the body with just `sup`, such that it can be accepted by the *Code2Vec* model java parser.

4.2.2 Code2Vec Integration

To integrate the *Code2Vec* model with the Fast API server, the source code available on their website[3] was downloaded, from which it was necessary to adapt the `InteractivePredictor` class, which was prepared to run the model through the terminal input.

Additionally, it was created a controller file exposing the `/predict` endpoint via the *Fast API* router and then registering the new route on the main file of the Fast API server. This endpoint receives the body of the method to be vectorized and returns the code vector extracted from the `InteractivePredictor` class prediction results.

Also, inside the *Code2Vec* source code, there is an `Extractor` class, which is responsible for parsing the given code snippet into the AST paths, which can be changed to support other language parsers.

Following the instructions file of the *Code2Vec* repository, it's possible to find a variety of trained models ready to configure from which we need to choose and download. For this work, it was decided to use the released trained model[4] by the largest dataset, because this dataset contains a total of 9500 top-starred Java projects from GitHub and about 16 million examples of code snippets. The *Code2Vec* also provides the possibility of downloading a dataset and training the model from scratch.

4.2.3 Codebase Manager

The Codebase Manager, which is located inside the *Spring Boot Server*, is responsible for the creation, persistence, and management of all codebases related data inside the file system. The process of creating a codebase consists of the user submission of the sequence of accesses, and the code embeddings files along with the name of the codebase. During this process, data from both submitted files are matched to complement the code embeddings entities identification, since the *spoon-callgraph* parser does a more detailed entity identification.

4.2.4 FVCG Strategy

To implement the FVCG strategy, it was first created a new service on the *Spring Boot Server* to generate dendrograms of functionalities with this strategy similarity function. The service starts by loading the selected codebase from the Codebase Manager and then generates all functionalities vectors from the call graph methods.

To generate these vectors, the code embeddings are loaded from the codebase, and then the service goes through every package and its classes looking for the methods identified as controllers, that represent the starting point of each functionality, as can be seen in the attached function A.1.

[3] https://code2vec.org/
[4] https://code2vec.s3.amazonaws.com/model/java-large-released-model.tar.gz

A recursive function A.2 was built to sum all the vectors from the controller's callgraph multiplied by the user requested weight parameters in order to compute the functionality vector from each controller.

The base case of the recursive function is when the depth achieves a value of zero, or when it reaches a leave of the callgraph tree, and so there are no more call methods to analyze. Otherwise, the function just keeps computing the invoked methods vectors and summing the used weights so that in the end it is possible to divide the sum of all weighted vectors by the sum of the weights as in equation 3.1.

4.2.5 FVSA Strategy

To implement the FVSA strategy, it was also created a new service on the *Spring Boot Server* to generate dendrograms of functionalities with this strategy similarity function. The service starts by loading the selected codebase from the Codebase Manager and then generates all functionalities vectors from the functionalities represented as a sequence of accesses, also called entities' traces, as can be seen in the attached function A.3.

To generate these vectors, the code embeddings are loaded from the codebase, and then the service goes through every package looking for the entities' classes. Every time an entity class is found, its vectorization process begins (A.5), which consists of retrieving all the code vectors from the hierarchy ascended classes' methods and from the respective class methods, and then is calculated the mean vector of the collected vectors to define the entity vector.

With all entity vectors computed, the service iterates over the sequence of accesses list, where each sequence of accesses represents a functionality. Then, the accesses present in each sequence of accesses are used to compute the respective functionality vector. Each access consists of the entity that was accessed and the type of access (read or write). To calculate the vector of each functionality all the vectors of the accessed entities are used, where each vector is multiplied by the requested weight assigned through the type of access. Then, the weighted mean of these vectors is used to define the functionality vector like in equation 3.2.

4.2.6 CV Strategy

To implement the CV strategy, a new service on the *Spring Boot Server* was created to generate dendrograms of classes with this strategy similarity function. The service starts by loading the selected codebase from the Codebase Manager and then generates all classes vectors by a class vectorization process that relies on each class methods vectors, as can be seen in the attached function A.6.

To generate the class vectors, the code embeddings are loaded from the codebase, and then the service goes through every package in order to vectorize all classes. For each class, the vectorization process (A.5) uses the same vectorization function as the FVSA when vectorizing its entities. All the

code vectors from the ascended hierarchy classes are grouped with the class methods from the class to vectorize, and the mean of all of these vectors is calculated to represent the class vector.

4.2.7 EV Strategy

To implement the EV strategy, a new service on the *Spring Boot Server* was created to generate dendrograms of entities with this strategy similarity function. The service starts by loading the selected codebase from the Codebase Manager and then generates all entities vectors by the same class vectorization process as the CV strategy, as can be seen in the attached function A.7.

4.2.8 Dendrogram Creation

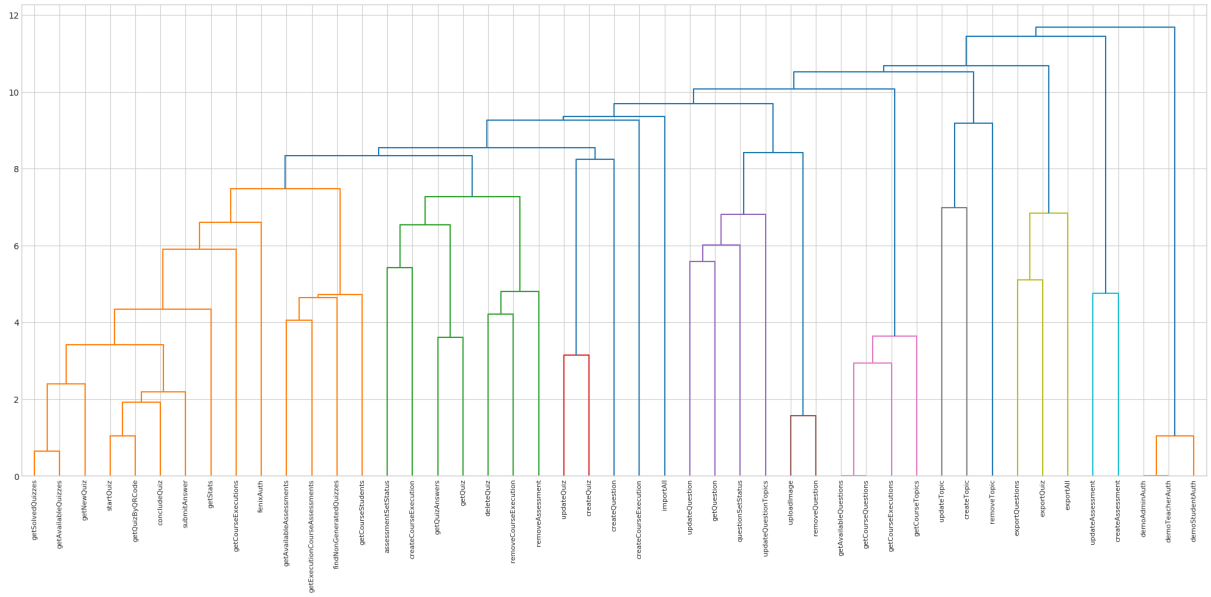


Figure 4.3: Example of a dendrogram created by the FVCG strategy over the quizzes-tutor[5]codebase.

After computing all functionality/classes/entities vectors, depending on the used strategy, these vectors are written in a temporary file, and the *Spring Boot Server* requests the *Fast API Server* to create a dendrogram 4.3 using the distance between the vectors as the similarity measure.

4.2.9 Decomposition Process

With the dendrogram generated, it is possible to experiment various cuts in order to generate possible decompositions.

[5] https://quizzes-tutor.tecnico.ulisboa.pt/

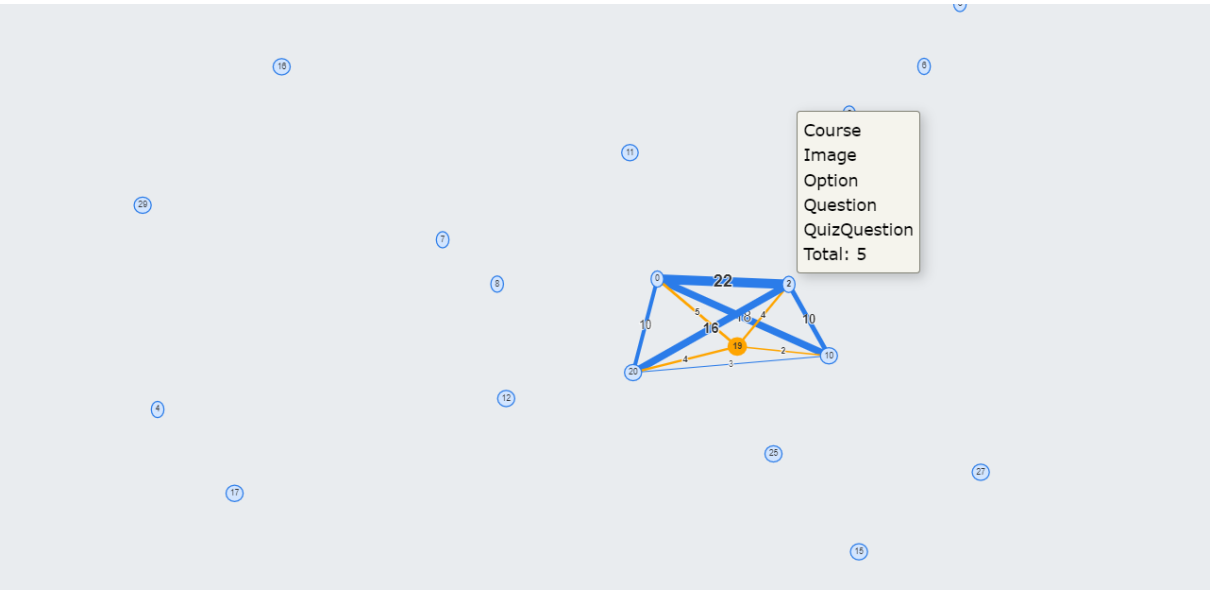


Figure 4.4: Cluster view example of a decomposition performed by cutting the previous dendrogram 4.3

When making a cut in the functionality dendrogram, functionality clusters are generated, but as the Mono2Micro tool works with entity clusters, it is necessary to transform functionality clusters into entity clusters.

This transformation process is done in the *Fast API Server*, where the sequences of accesses to the entities of each functionality are used to identify for each entity which is the functionality that does more accesses. To do so, the number of times each entity is accessed by each feature is calculated, which then serves to assign that entity to the cluster where the functionality is located. Because of this transformation process, some functionality clusters may become empty, when there is no entity assigned to a cluster as shown in figure 4.4.

4.2.10 Analyzer

This module, present on the *Spring Boot Server*, has been modified to support all the strategies mentioned above, where it generates all the possible combinations of parameters for each one, and generates the respective dendrograms and decompositions using the *Fast API Server*. For each decomposition generated, the metrics mentioned in the evaluation section 5 are calculated, which is done by *Spring Boot Server*.

5

Evaluation

Contents

To evaluate the research questions, we compare the *Code2Vec* decompositions generated using the *Code2Vec* similarity measures built on the monolith functionalities with the decompositions generated using sequences of access, as in [16], the decompositions generated using vectors for classes built with *Code2Vec*, as in [3], and the decompositions that only consider vectors for entities, which is a sub-category of the previous strategy.

5.1 Decomposition Generation

To evaluate the strategies it is necessary to generate a significant number of decompositions, varying the number of clusters and the strategies weights. In terms of the number of clusters, for codebases up to 10 entities, a maximum of 3 microservices are generated, between 10 and 20 a maximum of 5 microservices, and for more than 20 the maximum number of microservices is 10.

We start at a minimum of 3 microservices and generate all possible decompositions by varying each strategy's parameters using a step of one until reaching the maximum number of microservices. Since the strategies that don't represent a microservice by a set of entities may result in empty clusters, the real number of clusters of the generated decompositions is smaller than the requested one. To overcome this issue, we continue to increase the requested number of microservices and generate the respective decompositions until we achieve one that results in a number of clusters bigger than the maximum value.

The number of decompositions generated for each strategy depends on the number of its parameters since we explore all the possible combinations. For the weight parameters, we need to create all the combinations where the sum of the weights equals 100, using intervals of 10. In the FVCG strategy, we decided to vary the depth parameter from 1 to 6.

The **Hierarchical Clustering** algorithm is applied to the strategies vectors and distances, using the euclidean distance. A dendrogram is generated, which is cut to generate decompositions with different numbers of clusters. Since the hierarchical clustering algorithm supports different types of linkage criteria to determine the distances of the clusters, they are used as variations in the evaluation. The three criteria considered are:

- Single-linkage clustering: Distance between the closest entities of the measured clusters.
- Complete-linkage clustering: Distance between the furthest entities of the measured clusters.
- Average-linkage clustering: Average of the distances between each entity of one cluster and the entities of the other.

This linkage type parameter will also be varied during the generation of decompositions.

5.2 Evaluation Metrics

Three metrics are used to evaluate the quality of a generated decomposition: coupling, cohesion, and complexity.

5.2.1 Cohesion Metric

The cohesion measures the single responsibility principle [12]. The cohesion of a decomposition is computed using the cohesion of each one of its clusters. The cluster cohesion is percentage of the cluster's entities accessed by the respective functionalities. Therefore, a cluster has high cohesion if the accesses done by functionalities interact with all the entities in the cluster. And so, it has low cohesion if each functionality, that access the cluster, only accesses a small subset of the cluster entities. In a formal notation:

$$cohesion(c) = \frac{\sum_{f \in funct(c)} \frac{\#\{e \in c.entities : e \in G_f.accesses.entities\}}{\#c.entities}}{\#funct(c)} \tag{5.1}$$

where the functionalities of the cluster c are denoted as $funct(c)$, the cluster's entities are denoted as $c.entities$, and $G_f.accesses.entities$ represents the entities accessed by feature f, extracted from the feature call graph G_f. Then, the overall decomposition cohesion is the aggregation of all clusters' cohesions using the mean function.

5.2.2 Coupling Metric

The coupling reflects the interdependence between microservices. This is measured by the percentage of entities a cluster has to know of another. A cluster knows the entity of another cluster if there is a functionality that immediately after accessing an entity in the first cluster accesses an entity in the second cluster. Note that coupling is not a symmetric property because it depends on the order of accesses. On the other hand, it differs from cohesion because only the pairs of accesses where two clusters are involved are relevant. The coupling of a decomposition is the average of the coupling between all pairs of the decomposition cluster. The previous work defines how coupled two clusters c_i and c_j are, using the percentage of the c_j cluster's entities that c_i needs to access to perform its features, which is denoted as:

$$coupling(c_i, c_j) = \frac{\#\{e \in c_j : \exists_{ri \in RI(c_i, c_j)} e = ri[2].e\}}{\#c_j.e} \tag{5.2}$$

with $RI(c_i, c_j)$ being the remote invocations from c_i to c_j.

The resulting coupling value for each cluster c is the average of the coupling itself with every other cluster c':

$$coupling(c) = \frac{\sum_{c' \in D, c' \neq c} coupling(c, c')}{\#(D.clusters) - 1} \tag{5.3}$$

being D the system decomposition. And, the coupling of the decomposition consists on also the average of all the clusters' coupling values.

5.2.3 Complexity Metric

Predict how much effort is present in each feature migration. The migrated features become a collection of local transactions (LT) within a cluster and may require remote invocations (RI) to other clusters, resulting in distributed transactions.

The complexity of a feature f is defined as:

$$complexity(f, D) = \sum_{lt \in partition(G_f, D)} complexity(lt, D) \tag{5.4}$$

where the $partition(G_f, D)$ returns a set of local transactions LT and remote invocations RI given the call graph G_f of the feature and a decomposition D.

Complexity measures the effort required to migrate a functionality from a monolith to a microservices architecture [16]. This complexity results from the need to introduce a set of distributed transactions to implement the functionality. Since the distributed transactions execution needs to be implemented using eventual consistency, due to scalability [17], it is necessary to change the business logic to consider intermediate states of the domain entities, which is a consequence of the lack of isolation. Therefore, the complexity depends on the number of distributed transactions required to implement a functionality, and the number of intermediate states they introduce. The former is calculated by how many times the functionality sequence of accesses is split between clusters, each split is a distributed transaction. The latter is calculated by identifying the read and write accesses done by the distributed transactions. The complexity of a decomposition is the sum of the decomposition of the complexity of the functionalities.

The complexity of a local transaction lt is determined by the number of distributed functionalities that access relevant domain entities with the inverted access mode made by the local transaction. The inverted access mode of a read is a write access and vice versa.

$$\begin{aligned} complexity(lt, D) = \#\cup_{a_i \in prune(lt)} \\ \{f_i \neq lt.f : dist(f_i, D) \wedge a_i^{-1} \in prune(f_i, D)\} \end{aligned} \tag{5.5}$$

given the decomposition D, a function $dist(f, D)$ that identifies distributed functionalities, ai^{-1} denotes the inverted access mode, and a $prune$ function that filters the relevant domain accesses by removing

repeated accesses, and read accesses that happen after a write within the same local transaction.

5.2.4 Combined Metric

An additional metric is built using the three metrics to evaluate which decompositions have a better balance between them, as presented in equation 5.6. Note that, the complexity is divided by the maximum complexity of all decomposition to obtain a value between 0 and 1, this is called uniform complexity.

$$combined(d) = \frac{1 + \frac{complexity(d)}{max\_complexity(D)} + coupling(d) - cohesion(d)}{3} \tag{5.6}$$

5.3 Codebase Sample

To gather the codebases sample for this experiment, a list of GitHub repositories that depend on the Spring Data JPA library[1] was filtered to exclude codebases with less than five domain entities and controller classes. After that, the remaining codebases were sorted by the number of GitHub stars and manually selected from the top in order to keep the sample quite diverse in terms of codebase sizes. From these codebases we still had to exclude a few due to the dependence on libraries that generate methods from annotations on compile time, making these methods not available for a static analysis.

The selection process led to a relatively large number of monolith codebases (85), with an average number of code lines around 25 thousand and a standard deviation of 33 thousand lines of code, indicating a high variation of the codebases size. Also, it is possible to observe the distribution of the number of controllers and domain entities in figure 5.1.

5.4 Statistical Analysis

To validate the research questions we start to compare the strategies for the cohesion, coupling, complexity, and combined metrics, using decompositions for the 85 codebases chosen for different numbers of clusters. To measure whether the differences in the results of the strategies are statistically significant, we apply Welch's t-test [18].

Welch's t-test [18] is a two-sample location test used in statistics to test the hypothesis that two populations have equal means and it is more reliable when the two samples have unequal variances and possibly unequal sample sizes, which is the case. The hypotheses of the Welch's t-test are the following:

- H_0: $\mu_1 = \mu_2$, the samples have equal means;

[1] https://github.com/spring-projects/spring-data-jpa/network/dependents

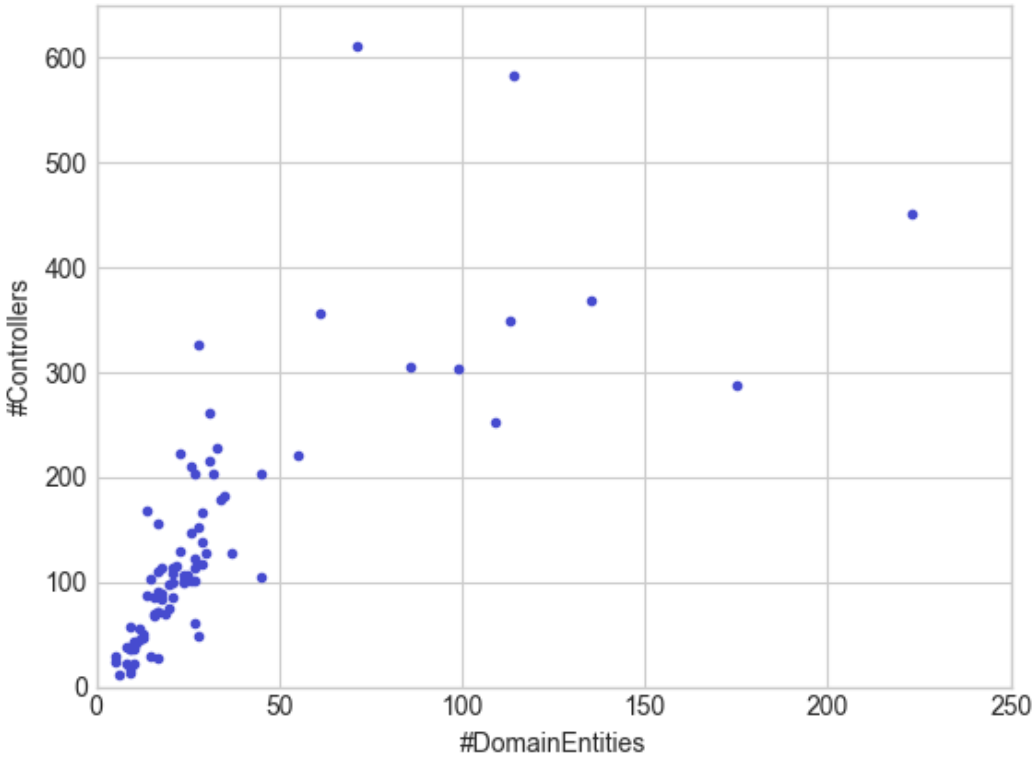


Figure 5.1: Representation of the 85 codebases used in the evaluation.

- H_1: $\mu_1 \neq \mu_2$, the samples have distinct means.

To reject or accept the presented null hypotheses, we use a significance level of 0.05.

In addition, we also analyze each proposed strategy individually to study the impact of the strategy parameters on metrics values. To do so, we run regressions for each type of parameter applying the ordinary least squares (Ordinary Least Squares (OLS)) method to choose the regression parameters, β_i and $cons$ of the equation 5.7.

$$metric(d) = \sum_{i \in parameters} \beta_i \times w_i + cons \tag{5.7}$$

To test these regressions, we also use a significance level of 0.05 to accept or reject the following hypotheses:

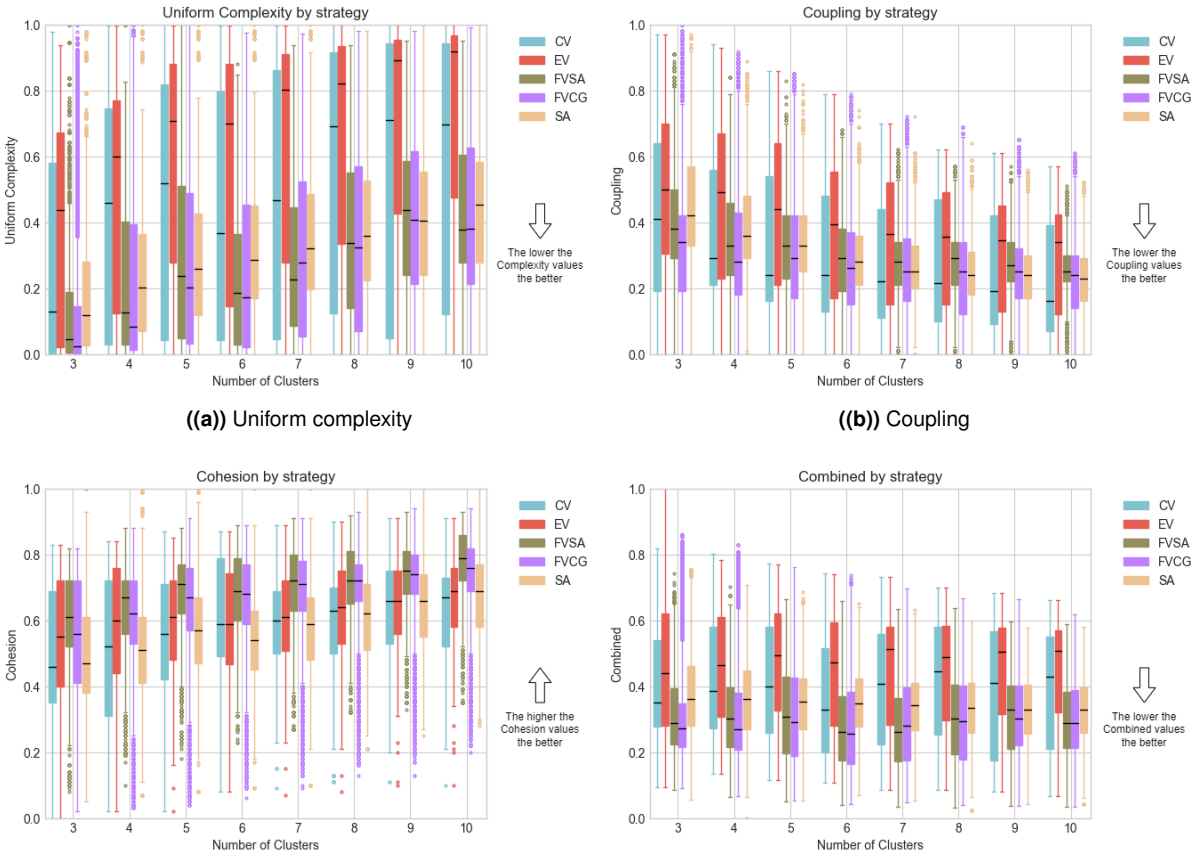
- H_0: $\beta_i = 0 \ \forall i \in \ ]0, \#parameters]$, the evaluation metrics does not have any relation with the parameters under analysis;

- H_1: $\beta_i \neq 0 \ \exists i \in \ ]0, \#parameters]$, the evaluation metrics are at least affected by one of the parameters under analysis.

For the feature vectorization strategies, since they use parameter weights, it will be necessary to study the problem of multicollinearity, since the weights depend on each other by adding up to 100. To overcome this problem, we repeat the regression analysis without one of the dependent parameters, and by doing this for each parameter we can retrieve better coefficient values. During the evaluation, the parameters of the clustering algorithm will also be considered as parameters of the strategies to understand the impact of the linkage criteria.

5.5 Results

To answer the research questions, we went through all the generated decompositions to calculate the respective values for cohesion, coupling, complexity, and combined metrics. With these values, it's possible to compare the strategies and look for any correlation between the proposed strategies' parameters and the metrics values. Figure 5.2 presents the results.



((a)) Uniform complexity

((b)) Coupling

((c)) Cohesion

((d)) Combined

Figure 5.2: Evaluation Metrics applied to the 85 codebases

Two strategies can be compared at the level of each metric alone, or in general through the combined

metric. A good strategy generates decompositions with low values of coupling and complexity, but with high values of cohesion. When comparing strategies with the combined metric, the best decompositions are the ones with lower values because the metric itself uses the coupling, complexity, and symmetric value of cohesion.

We start by comparing the results for strategies FVSA and FVCG. For complexity, the Welch's t-test rejects the hypothesis of having the same mean values except when the number of clusters is 9 and 10, and through Figure 5.2(a) it is possible to notice that most of the FVCG strategy values are most of the times lower than those of the FVSA, since the median is lower. Thus, it can be concluded that the decompositions generated by the FVCG strategy are in general less complex than those generated by the FVSA strategy.

Considering coupling, the hypothesis of having the same median values in each number of clusters is also rejected by Welch's t-test. From Figure 5.2(b) it is possible to observe that the coupling values for the FVCG strategy are lower than those of the FVSA strategy.

As for cohesion, (Figure 5.2(c)), the FVSA seems to obtain best results than FVCG since Welch's t-test rejects the hypothesis of having the same cohesion mean values and most of those results are higher than the ones generated by the FVCG.

In addition to these metrics, it is interesting to analyze the combined metric in figure Figure 5.2(d) that represents the balance between the previous ones. Welch's t-test only accepts the hypothesis of both strategies have the same mean when the number of clusters is 6 and 9, and as the values of the FVCG strategies are lower than the ones of FVSA, the FVCG achieve the best-balanced results.

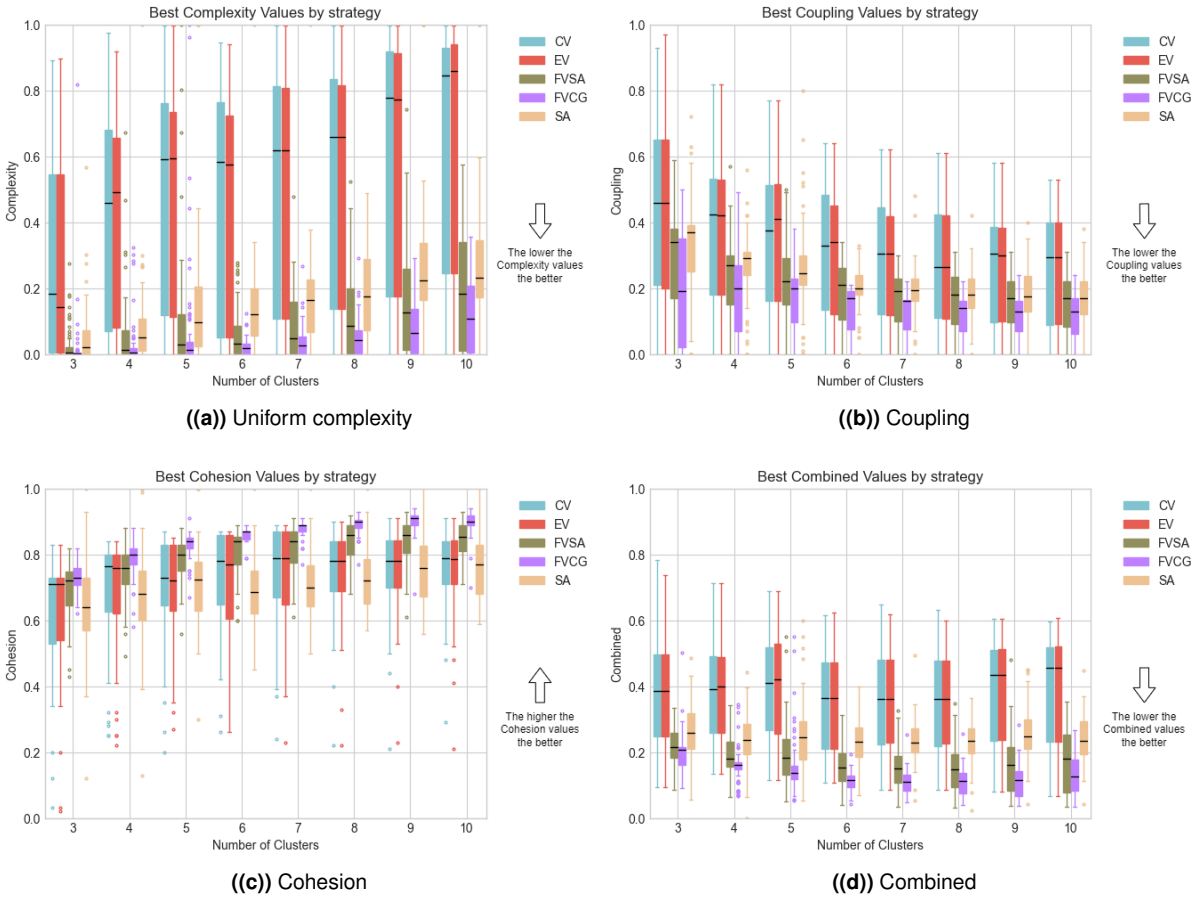
Table 5.1: Average number of decompositions and duration of each strategy when generating all decompositions by permuting each strategy parameters

	CV	FVCG	FVSA	SA	EV
#Decompositions Mean	71	131168	503	1514	17
Performance Time Mean (s)	62	2830	350	39	32

Since each strategy generated a different number of decompositions (table 5.1) derived from the number of parameters and from the conversion of functionalities' clusters to entities' clusters, it was decided to perform a second analysis in which only the best decompositions of each codebase are used for every strategy and number of clusters. This way, the same number of decompositions are considered for each strategy. Additionally, and as is shown in Table 5.1, the number of decompositions associated with strategy FVCG is significantly larger, which has an impact on the performance. Therefore, it is relevant to understand which parameters can be discarded, if any, to minimize the number of decompositions that need to be generated.

By decreasing the number of decompositions, the results' dispersion of the strategies that generated a higher number of decompositions decreased substantially along with the number of outliers, as shown

in Figure 5.3.



((a)) Uniform complexity

((b)) Coupling

((c)) Cohesion

((d)) Combined

Figure 5.3: Evaluation Metrics applied to the best decompositions of the 85 codebases for each metric

By using only the best decompositions for each codebase, the results of the functionality vectorization strategies improved significantly. For complexity, Welch's t-test accepts the hypothesis that they have the same mean when the number of clusters is 3 and 5. Regarding cohesion, coupling, and the combined metrics, the t-test continues to reject the hypothesis for any number of clusters. Overall, when looking at figure 5.3, the FVCG strategy distinguishes itself from the FVSA by having better results for cohesion, coupling, complexity, and so for the combined metric. Also, the FVCG strategy proves to be more interesting because it does not require such an in-depth analysis of the code as the FVSA strategy, and is more independent of the technology stack.

5.5.1 Does the use of *Code2Vec* with the functionality perspective provides better results than sequences of accesses?

To answer the first research question, the proposed strategies that rely on feature vectorization and the *Code2Vec* model (FVSA, FVCG) are compared with the SA strategy, which clusters entities by their access sequences. By comparing the FVSA strategy with SA, it will be possible to conclude the impact of the *Code2Vec* model on the sequence of entity accesses. And the comparison of FVCG and SA strategies will indicate if only the use of *Code2Vec* can achieve better results than a very detailed analysis used in the SA strategy.

In terms of complexity, Welch's t-test only accepts the hypothesis of two strategies having the same mean values when comparing the FVSA and SA strategies and the number of clusters is 4 or 5. In all other cases, including the FVCG strategy, as shown in Figure 5.3(a), most of the values of the proposed strategies are lower than those of the SA strategy, which leads to the conclusion that using the *Code2Vec* model with a functionality perspective generates less complex decompositions.

Regarding coupling, Figure 5.3(b) the FVSA and the SA strategy have very similar results which can be validated with the results of Welch's t-test, that accepts the hypothesis that the strategies have the same average coupling values for every number of clusters except for 3 and 5. The FVCG strategy obtains better results than the SA strategy since the Welch's t-test rejects that both strategies have the same mean for every number of clusters and the majority of the FVCG coupling results are lower than the ones of the SA strategy.

When it comes to the cohesiveness of the proposed strategies 5.3(c), the values are better compared to the SA strategy. Welch's t-test rejects all the hypotheses that the FVCG and the FVSA strategies have the same mean cohesion values when compared to the SA strategy. This implies that the decompositions generated by the *Code2Vec* proposed strategies have highly cohesive microservices.

Overall, when applying the combined metric (Figure 5.3(d)) to these strategies, the results of Welch's t-test also reject the hypothesis that the strategies have the same mean values for every comparison between the proposed strategies (FVCG and FVSA) and the SA strategy. With these results, it's possible to conclude that the appliance of the *Code2Vec* model with a functionality perspective to the sequence of accesses analysis improves the results, but when using just the functionalities vectorization without the sequence of accesses it's possible to achieve even better results.

5.5.2 Does the application of the functionality perspective provides better results than Al-Debagy and Martinek's class perpective?

To answer the second research question, the proposed strategies (FVSA, FVCG) are compared with the CV strategy, proposed by Al-Debagy and Martinek, and the EV strategy, which is an adaptation of the

CV.

Calculating Welch's t-test between the proposed strategies and the CV strategy it's possible to reject the hypothesis of having the same complexity, cohesion, coupling, and combined means for every number of clusters. These results show that these strategies are quite different as can be seen in figure 5.3 and that the results of the CV strategy are a lot worst for every metric than the ones of the FVCG and FVSA strategies.

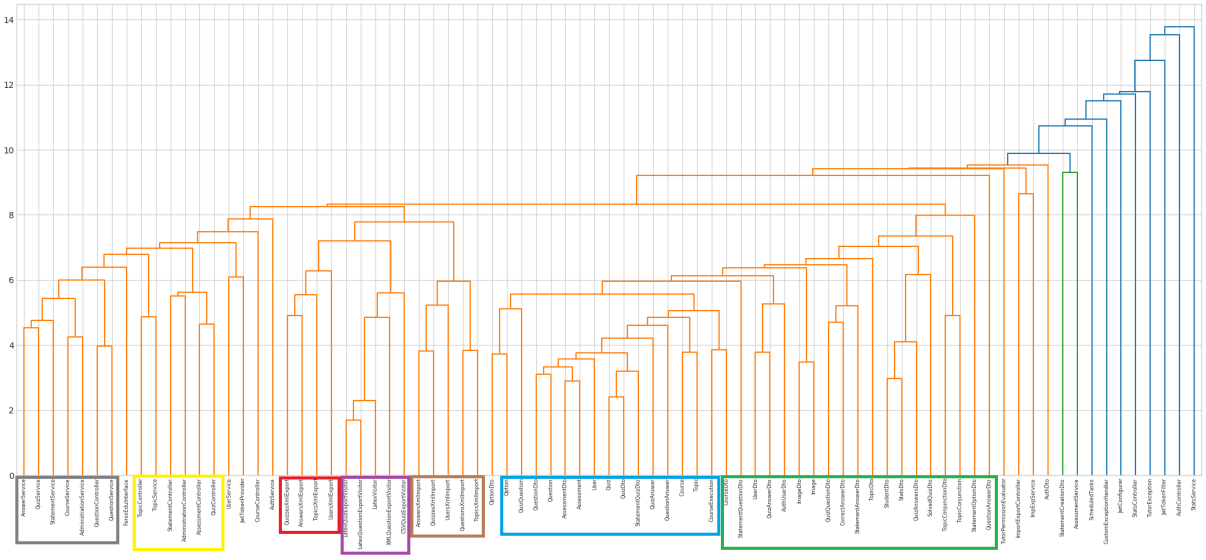


Figure 5.4: Class dendrogram of the quizzes-tutor codebase using the CV strategy.

The results of the CV strategy derive from clustering the vectorized classes, which leads to them being grouped by the various class types (Domain entities, Service, Controller, Configuration) as it's possible to observe in figure 5.4. Thus, when class clusters are converted to entity clusters, most of the domain entities are grouped in the same cluster leading to decompositions with several microservices with only one entity.

Trying to avoid this behavior, the EV strategy was implemented, which only considers classes that represent domain entities. But, this strategy ended up getting the same results as the CV strategy, since Welch's t-test accepts that it has the same average across all strategies and for all numbers of clusters, which led to Welch's t-test also rejecting the hypothesis that this strategy has the same means as the proposed strategies across all metrics and for all numbers of clusters.

This evaluation shows that the use of *Code2Vec* with a class perspective generates worse decompositions, since the class vectors are heavily influenced by each class type because of its respective lexical tokens, but could be a good approach to cluster classes into packages to organize the code by classes types like in [8].

5.5.3 Does the input parameters of the proposed strategies impact the results of the evaluation metrics?

To answer the third research question we will analyze the parameters of each of the proposed strategies.

Starting with the FVCG strategy, there are six parameters to analyze, the maximum depth (d) the call graph is explored, the four weights to control which method types are more relevant, and the linkage type used in the clustering algorithm.

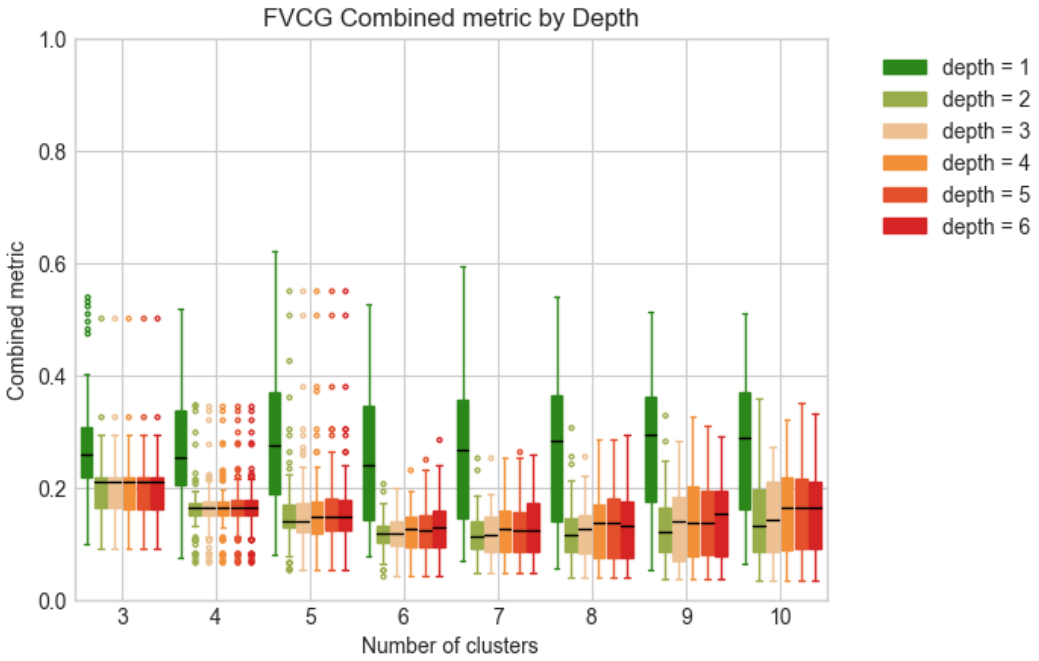


Figure 5.5: Regression of the depth parameter per the combined metric values.

Welch's t-test between a depth of 1 and a depth of 2 shows that there is a significant difference between the results presented in figure 5.5, and so depth 1 provides worse results than greater depths. But, when calculating an OLS regression for the depths greater than 1, allows us to reject the hypotheses that by increasing the depth more than 2 better results are obtained because the *p-value* is smaller than the significance level.

Therefore, it's impossible to conclude, for depths greater than 1, that any given depth is better than another. Since smaller depths require less computation, it's possible to rely just on a depth of two, in

which only the controller method input method of the functionality (controller) and the methods it invokes there are used.

This leads us to conclude that when using a lexical approach, only the first methods of each functionality are needed since they present most of the lexical tokens present in the entire functionality call graph.

Table 5.2: Regresssion coefficients of the method types weights with the combined metric

Weight	Without w_c	Without w_e	Without w_s	Without w_i
Controllers (w_c)	-	-5.9e-05	0.0004	3.5e-05
Entities (w_e)	5.9e-05	-	0.0004	9.4e-05
Services (w_s)	-0.0004	-0.0004	-	-0.0003
Intermediate (w_i)	-3.5e-05	-9.4e-05	0.0003	-

The regression between the method types' weights and the combined metric rejects the hypothesis that a different combination of the method types' weights affects the evaluation metric results since the *p-value* is less than the significance level.

Since the weights depend on each other, in a regression covering all four weight parameters it was noticed a multicollinearity problem. To overcome this problem, four additional regressions were made, each one without one of the weights. Although the new coefficients had much more reliable values as shown in the table 5.2, they also lead to the rejection of the above-mentioned hypothesis.

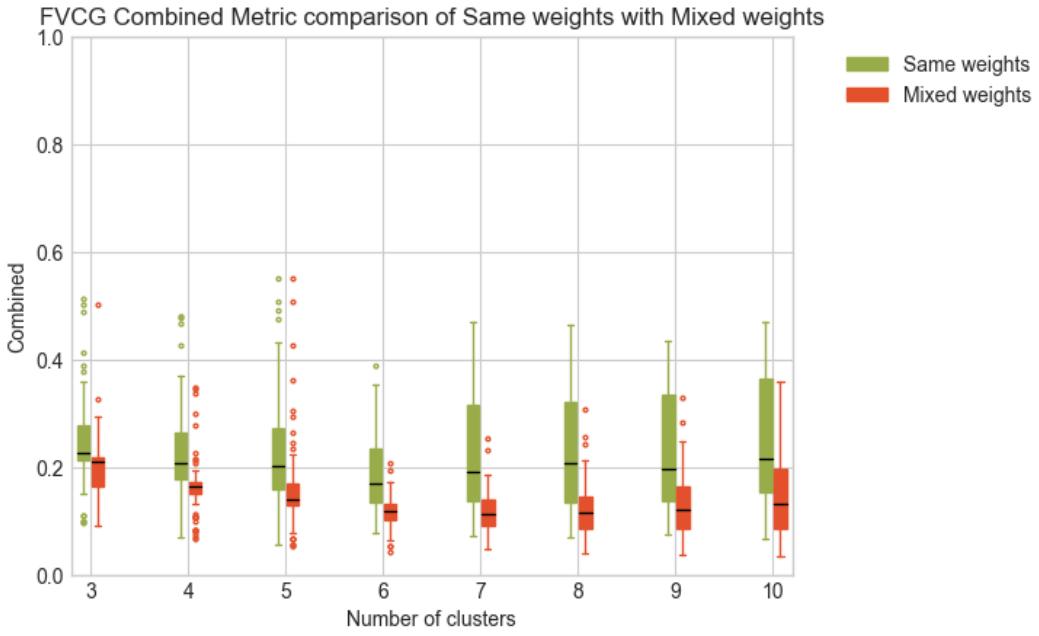


Figure 5.6: FVCG Comparison of the best decompositions combined metric results when the weights are equally distributed versus the best decompositions when using all possible weights distributions.

As it's not possible to find a perfect combination of method types' weights, an additional analysis of

the best decompositions for each codebase and number of clusters was made to understand if the use of the same weights for all method types can achieve good results as all possible weights combinations (Figure 5.6). But, Welch's t-test results rejected the hypothesis that the combined metric mean values of the same weights and mixed weights have distinct values, and the results obtained by mixing all the weights possibilities achieve better results when looking for the best decompositions.

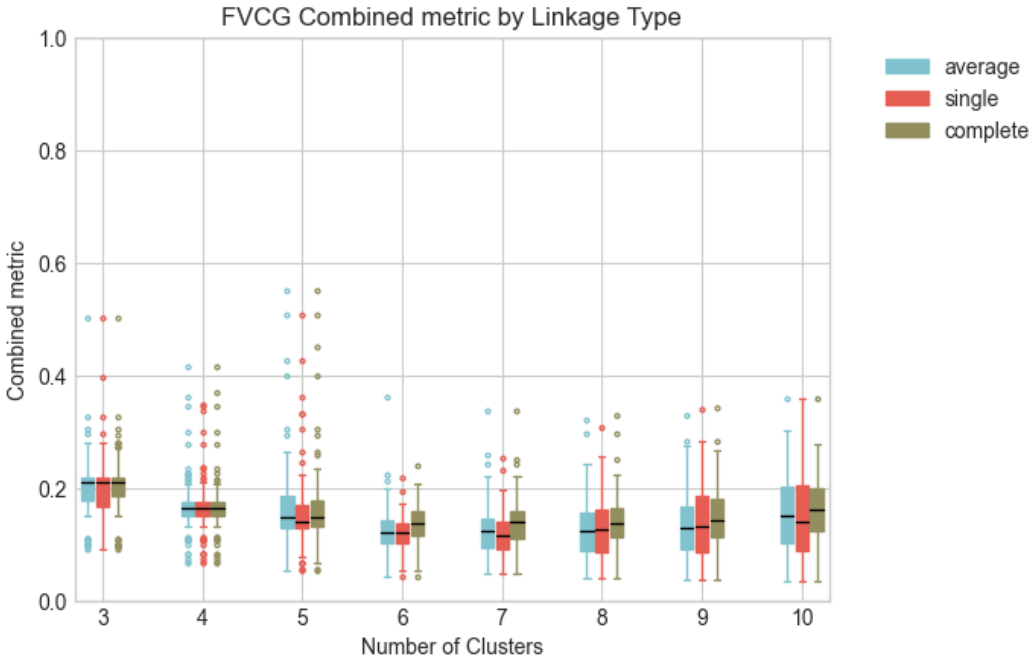


Figure 5.7: Regression of the Linkage Type parameter per the FVCG combined metric values.

Regarding the linkage criteria, to understand the impact of the cluster algorithm parameter over the combined results, it was used a depth of two when comparing the best results for each linkage type (average, simple, complete).

By comparing the different linkage types for each number of clusters(Figure 5.7), Welch's t-test allows us to state that the results of both three linkage types have the same means of the combined metric, with just two exceptions where the number of clusters is 6 and 7 when comparing the single type versus the complete linkage type, but even those **p-values** are very close to the significance level. This indicates, that the choice of the linkage type is relevant when clustering the functionality vectors generated with the FVCG strategy.

For the FVSA strategy, there are only three parameters to analyze, the two types of accesses' weights (write, and read), and also the linkage type.

The regression between the accesses types' weights and the combined metric allows us to reject the hypothesis that a different combination of the accesses types' weights affects the evaluation metric results since the *p-value* is less than the significance level. Once again, as these weights depend on

Table 5.3: Regresssion coefficients of the accesses types weights with the combined metric

Weight	Without w_r	Without w_w
Read Access (w_r)	-	0.0001
Write Access (w_w)	0.0001	-

each other, to avoid the problem of multicollinearity, two regressions were made separately (5.3), one with just the weights of read accesses, and one with the weights of write accesses. Both regressions reject the hypothesis that the weights have any statistically significant impact on the combined metric results.

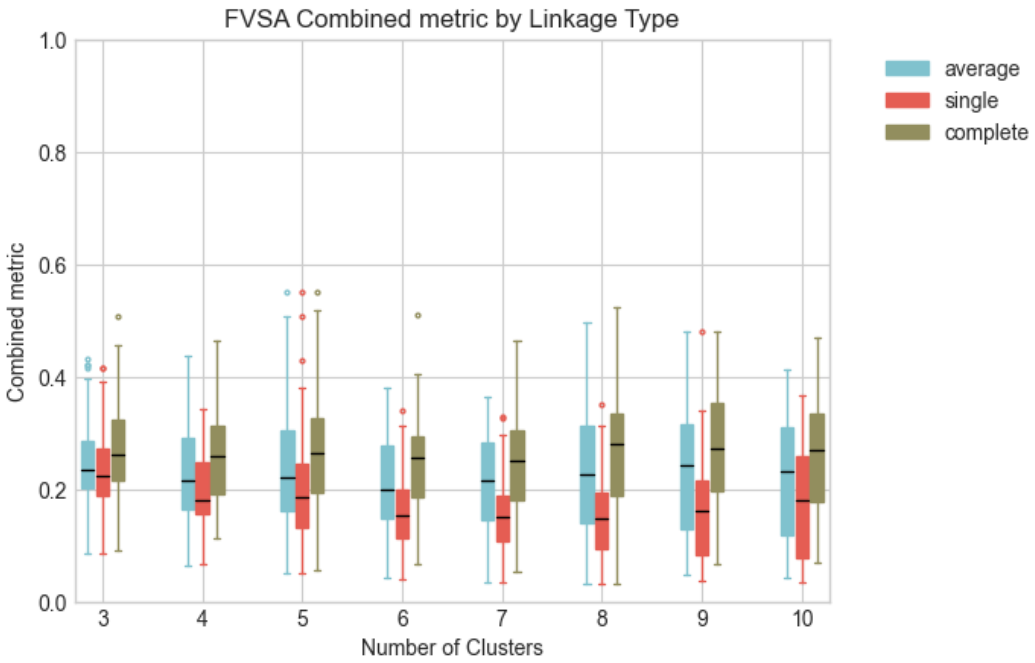


Figure 5.8: Regression of the Linkage Type parameter per the FVSA combined metric values.

When it comes to the linkage criteria, Welch's t-test rejects the hypothesis that the single type has the same mean as the average type for all cluster sizes except for 3. It accepts the hypothesis that the complete and average types when the number of clusters is 5, 7, 8, 9, and 10, which indicates that for these two linkage types the results obtained are very similar but worst than the single linkage type, which is the one that obtains, in general, the best results for the FVSA strategy.

5.6 Lessons Learned

In summary, what is possible to learn from this work is the following:

- It is possible to perform a lexical analysis of the AST with a neural network model and obtain

better results than a complex static analysis that captures the functionalities sequences of access to domain entities.

- Adding a neural network model to the static analysis of entity accesses (FVSA strategy) improves its results.
- Classes vectorization is shown to lead to decompositions where the classes are grouped by their type.
- The FVCG strategy is shown to provide the best results, when compared with the sequence of accesses strategy, and only using a depth of 2.

5.7 Threats to Validity

The FVCG strategy was only implemented to support java codebases, but since it is possible to change *Code2Vec* to accept more languages, it can be easily generalizable, just by creating a new parser for each language.

Due to the selection codebases selection process, we believe that the 85 selected codebases are representative of monolith systems. Although all codebases use the Spring framework, it does not bias the results, because these frameworks apply the same architectural patterns.

There may be some correlation between coupling and complexity metrics, so the results of the proposed new combined metric may be biased. Nevertheless, the results still promising when analyzing each metric separately.

The conversion of functionality clusters to entity clusters may bias the results. However, the strategies that applied this conversion have shown better results.

6

Conclusion

Contents

As the majority of monolith decomposition approaches perform a static analysis of the source code followed by a clustering algorithm, this work aimed to simplify and generalize this process by recurring to a lexical analysis independent of the technology stack.

The *Code2Vec* model was used to understand that a simple lexical analysis strategy can overcome in terms of complexity, coupling, and cohesion, a more complex analysis that has to extract all functionalities domain entities accesses sequences.

Analyzing monoliths as a set of functionalities was shown to provide better results than the monolith class vectorization strategy, which led to clusters of classes of the same type.

We conclude that the FVCG strategy, which only relies on the call graph for functionality vectorization, provides the best results. Additionally, it is possible to reduce the number of parameter combinations to choose the best decomposition by only using depth 2 of the call graph generation.

As an additional contribution, all the code of this work is available in this branch [1] of the *Mono2Micro* GitHub repository.

6.1 Future Work

Due to the results of this work, it would be interesting to explore the following topics for future work:

- Explore the FVCG strategy weight parameters with a Gradient boosting technique to infer in what circumstances a certain weight combination is better than another.
- Experiment a new decomposition approach, which only focuses on the vectorization of the controller method vector and the respective Data Transfer Object (DTO) classes, to confirm whether it is possible to use only the contracts from each functionality to generate a good decomposition.

[1] https://github.com/socialsoftware/mono2micro/tree/feature/code2vec

Bibliography

[1] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[2] M. Abdellatif, A. Shatnawi, H. Mili, N. Moha, G. E. Boussaidi, G. Hecht, J. Privat, and Y.-G. Guéhéneuc, "A taxonomy of service identification approaches for legacy software systems modernization," *Journal of Systems and Software*, vol. 173, p. 110868, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302582

[3] O. Al-Debagy and P. Martinek, "A Microservice Decomposition Method Through Using Distributed Representation of Source Code," *Scalable Computing: Practice and Experience*, vol. 22, no. 1, pp. 39–52, 2021.

[4] S. Santos and A. R. Silva, "Microservices identification in monolith systems: Functionality redesign complexity and evaluation of similarity measures," *Journal of Web Engineering*, Aug. 2022. [Online]. Available: https://doi.org/10.13052/jwe1540-9589.2158

[5] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service candidate identification from monolithic systems based on execution traces," *IEEE Transactions on Software Engineering*, vol. 47, no. 5, pp. 987–1007, 2021.

[6] L. Nunes, N. Santos, and A. Rito Silva, "From a monolith to a microservices architecture: An approach based on transactional contexts," in *Software Architecture*, T. Bures, L. Duchien, and P. Inverardi, Eds. Cham: Springer International Publishing, 2019, pp. 37–52.

[7] B. Andrade, S. Santos, and A. R. Silva, "From monolith to microservices: Static and dynamic analysis comparison," 2022. [Online]. Available: https://arxiv.org/abs/2204.11844

[8] M. Hammad and R. H. Banat, "Automatic Class Decomposition using Clustering," *Proceedings - 2021 IEEE 18th International Conference on Software Architecture Companion, ICSA-C 2021*, pp. 78–81, 2021.

[9] S. P. Lloyd, “Least squares quantization in pcm,” *IEEE Trans. Inf. Theory*, vol. 28, pp. 129–136, 1982.

[10] G. Mazlami, J. Cito, and P. Leitner, “Extraction of Microservices from Monolithic Software Architectures,” *Proceedings - 2017 IEEE 24th International Conference on Web Services, ICWS 2017*, pp. 524–531, 2017.

[11] M. Brito, J. Cunha, and J. Saraiva, “Identification of microservices from monolithic applications through topic modelling,” *Proceedings of the ACM Symposium on Applied Computing*, pp. 1409–1418, 2021.

[12] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. USA: Prentice Hall PTR, 2005, vol. 46, no. 2.

[13] S. P. Ma, Y. Chuang, C. W. Lan, H. M. Chen, C. Y. Huang, and C. Y. Li, “Scenario-Based Microservice Retrieval Using Word2Vec,” *Proceedings - 2018 IEEE 15th International Conference on e-Business Engineering, ICEBE 2018*, no. Ddd, pp. 239–244, 2018.

[14] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” 2013. [Online]. Available: https://arxiv.org/abs/1301.3781

[15] D. Taibi and K. Systä, “From monolithic systems to microservices: A decomposition framework based on process mining,” in *CLOSER 2019 - Proceedings of the 9th International Conference on Cloud Computing and Services Science*. SciTePress, 2019, pp. 153–164.

[16] N. Santos and A. Rito Silva, “A complexity metric for microservices architecture migration,” in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 169–178.

[17] A. Fox and E. A. Brewer, “Harvest, yield, and scalable tolerant systems,” in *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, ser. HOTOS ’99. USA: IEEE Computer Society, 1999, p. 174.

[18] B. L. Welch, “The Generalization of ‘Student’s’ problem when several different population variances are involved,” *Biometrika*, vol. 34, no. 1-2, pp. 28–35, 01 1947. [Online]. Available: https://doi.org/10.1093/biomet/34.1-2.28

A

Code of Project

Listing A.1: FVCG strategy Function

```
public void fvcgStrategy(
    FvcgRequestDTO requestParameters,
    List<Package> packages
) {
    for (Package pkg : packages) {
        for (Class cls : pkg.getClasses() {
            for (Method method : cls.getMethods()) {
                if (method.getType() == CONTROLLER_TYPE) {
                    Acumulator acumulator = getMethodCallsVectors(
                        requestParameters,
                        packages,
                        method,
                        requestParameters.getDepth()
                    );

                    List<Double> functionalityVector = vectorDivision(
                        acumulator.getVector(),
                        acumulator.getWeights()
                    );

                    saveFunctionalityVector(
                        method.getSignature(),
                        functionalityVector
                    );
                }
            }
        }
    }
}
```

Listing A.2: `GetMethodCallsVectors Recursion Function`

```
public Acumulator getMethodCallsVectors(
    FvcgRequestDTO requestParameters,
    List<Package> packages,
    Method method,
    int depth
) {
    int weights = GetMethodWeightByType(
        requestParameters,
        method.getType(),
        method.getClassType()
    );
    ArrayList<Double> vector = vectorMultiplication(
        method.getCodeVector(),
        weights
    );
    if (depth == 0 || method.getMethodCalls().length() == 0) {
        return new Acumulator(vector, weights);
    }
    for (MethodCall methodCall : method.getMethodCalls()) {
        Method invokedMethod = searchInvokedMethodBySignature(
            packages,
            methodCall.getPackageName(),
            methodCall.getClassName(),
            methodCall.getSignature()
        );
        Acumulator acumulator = getMethodCallsVectors(
            requestParameters,
            packages,
            invokedMethod,
            depth - 1
        );
        vectorSum(vector, acumulator.getVector());
        weights += acumulator.getWeight();
    }
    return new Acumulator(vector, weights);
}
```

Listing A.3: FVSA strategy Function

```
public void fvsaStrategy(
    FvsaRequestDTO requestParameters,
    List<Package> packages,
    List<SequenceOfAccesses> sequenceOfAccessesList
) {
    for (Package pkg : packages) {
        for (Class cls : pkg.getClasses() {
            if (cls.getType() == ENTITY_TYPE) {
                List<Double> entityVector = classVectorization(cls, packages);
                saveEntityVector(cls.getName(), entityVector.getMeanVector());
            }
        }
    }
    for (SequenceOfAccesses sequenceOfAccesses : sequenceOfAccessesList) {
        int weightSum = 0;
        Vector functionalityVector = new Vector();
        for (EntityAccess access : sequenceOfAccesses.getEntitiesAccesses()) {
            int weight = access.getType() == READ_TYPE ?
                requestParameters.getReadWeight() :
                requestParameters.getWriteWeight();
            List<Double> entityVector = getEntityVector(access.getEntityName());
            entityVector = vectorMultiplication(entityVector, weight);
            weightSum += weight;
            functionalityVector.addVector(entity_vector);
        }
        functionalityVector = vectorMultiplication(
            functionalityVector,
            weightSum
        );
        saveFunctionalityVector(
            functionalityTrace.getFunctionalityName(),
            functionalityVector
        );
    }
}
```

Listing A.4: `GetAscendedClassesMethodsCodeVectors Function`

```
public Vector getAscendedClassesMethodsCodeVectors(
    List<Package> packages,
    Vector classVector,
    String superClassName
) {
    if (superClassName.isEmpty()) {
        return classVector;
    }
    Class cls = findClassByQualifiedName(superClassName);
    for (List<Double> methodCodeVector : getClassMethodsCodeVectors(cls)) {
        classVector.addVector(methodCodeVector);
    }
    return getAscendedClassesMethodsCodeVectors(
        packages,
        classVector,
        cls.getSuperClassName()
    );
}
```

Listing A.5: `Class Vectorization Function`

```
public List<Double> classVectorization(
    Class cls,
    List<Package> packages,
) {
    Vector classVector = new Vector();
    classVector = getAscendedClassesMethodsCodeVectors(
        packages,
        classVector,
        cls.getSuperClassName()
    );
    for (Method method : cls.getMethods()) {
        classVector.addVector(method.getCodeVector());
    }
    return classVector.getMeanVector();
}
```

Listing A.6: CV strategy Function

```
public void cvStrategy(List<Package> packages) {
    for (Package pkg : packages) {
        for (Class cls : pkg.getClasses() {
            List<Double> classVector = classVectorization(cls, packages);
            saveClassVector(cls.getName(), classVector.getMeanVector());
        }
    }
}
```

Listing A.7: EV strategy Function

```
public void evStrategy(List<Package> packages) {
    for (Package pkg : packages) {
        for (Class cls : pkg.getClasses() {
            if (cls.getType() == ENTITY_TYPE) {
                List<Double> entityVector = classVectorization(cls, packages);
                saveEntityVector(cls.getName(), entityVector.getMeanVector());
            }
        }
    }
}
```