Discovery of Modularity in Monolithic Java Project Codes Using Complex Networks

Marcos Cordeiro de Brito Jr*, Calebe P. Bianchini
and Leandro A. Silva

Graduate Program in Electrical Engineering and Computing, Mackenzie Presbyterian University, São Paulo, Brazil, 01.302-907
E-mail: marcos.brito@mackenzista.com.br; calebe.bianchini@mackenzie.br; leandroaugusto.silva@mackenzie.br
**Corresponding Author*

Received 12 July 2024; Accepted 15 May 2025

Abstract

Monolithic architecture is a software design which brings significant difficulties to system developers when it comes to maintenance or expanding the scope of a project. On the other hand, a modular project consists of several similar entities, or modules, which are the object of similar functions or processes that, applied repeatedly, have well-defined classes and smaller modules to work, bringing benefits such as reduced project development time and increased productivity for the system developers. This work proposes the use of complex networks through the *NetworkX* library in Python, using modularity detection algorithms for the static analysis of Java code. The goal is to discover modules by analyzing dependencies between classes, indicating the best way to identify code clusters to be treated as modules automatically. The outcomes of applying the Greedy Modularity, Louvain, K-Clique, and

Journal of Web Engineering, Vol. 24_6, 911–942.
doi: 10.13052/jwe1540-9589.2463

Girvan Newman algorithms to two open-source projects will be presented. A comparative analysis of these results will be illustrated using generated graphs and a distribution map, emphasizing the number of communities identified by each algorithm.

Keywords: Monolithic, modules, complex network, Java, complexity, static analysis.

1 Introduction

The software development landscape is increasingly characterized by the complexity of projects, requiring a deeper understanding and refined methodologies for effective management. This complexity is often fueled by various factors, including inadequate planning for managing the project's growth speed [20], a lack of clarity regarding the requirements [9], and high turnover rates among development teams [16]. These challenges frequently lead to the creation of complex and maintenance-intensive codebases, predominantly characterized by a monolithic architecture where a single code structure combines all system objects and classes with interlinked responsibilities [1].

Opting for a monolithic architecture in software development at the outset or maintaining it through the lifecycle of a project may be deemed appropriate depending on the project's specific traits and goals, as long as the project is clearly defined, with well-structured modules and their responsibilities not becoming conflated within the system [12]. However, when these projects start to accumulate too many responsibilities or when module dependencies become overly interconnected, the complexity of the code increases, posing significant challenges for maintenance and scalability.

Some proposals have been made to address the inherent complexities of monolithic projects by structuring them in alignment with the domain related to the problem they aim to solve [10]. This approach advocates for the modular organization of a project to clarify the system's responsibilities. Nevertheless, if a project remains monolithic, strategic planning is essential to ensure that modules are well-defined or that the project is divided into other architectures, such as layered systems or microservices.

Despite the known challenges in maintenance and scalability of monolithic architectures, many developers still opt for this structure [12], facing issues with high developer turnover in particular [16].

Various architectural approaches such as Clean Architecture [19], Hexagonal Architecture [14], and Microservices [17] have been proposed to

mitigate the limitations of monolithic structures, emphasizing the organization of the project into well-defined modules with clearly delineated responsibilities.

However, there are situations in which there are legacy systems made in monolithic architectures that need to be transformed into modules. The transitioning from monolithic code to a modular architecture is a complex process that requires a deep understanding of the existing project and meticulous code rewriting to identify potentially modularize segments.

There are no proposals in the literature that present solutions that help this software refactoring process. Given this gap, this article proposes to automate module detection in software projects by using complex network analysis, employing modularity detection algorithms available in the Python NetworkX library [23].

By leveraging complex network analysis to explore the challenges of monolithic architectures, this study aims to provide valuable insights rather than dictate a definitive modularization approach. The proposed methodology serves as a decision support tool, offering multiple community detection analyses as inputs for domain experts to evaluate. Instead of determining the optimal modularization automatically, the system presents structured information that enables specialists—those with deep knowledge of the project's architecture and requirements—to assess which analysis best aligns with the software's needs. Ultimately, the responsibility for selecting the most suitable modularization approach remains with the project expert, while the algorithms function as analytical aids to enhance the decision-making process.

To demonstrate the development of this project and the theoretical foundation used, this article is organized as follows. In Section 2, similar ideas that served as inspiration are presented. In Section 3, details about the solution's architecture, the chosen programming language for development, the language adopted for static analysis, and the project workflow and stages are discussed. In Section 4, the experiments conducted with Java source codes are presented, showing the results of the generated graphs and the identified communities. Finally, in Section 5, the conclusions from the experiments and suggestions for future work are provided.

2 Related Works

In this section, we discuss related works that present approaches for understanding the complexity of monolithic projects and various strategies for

decomposing them into microservices. These studies serve as the foundation for the detailed research in the following sections. The reviewed works illustrate diverse methodologies, such as the application of similarity measures for automatic decomposition, the use of deep graph clustering based on dual views, and the incorporation of multi-view clustering to improve decomposition quality. These insights significantly contribute to the ongoing exploration of effective techniques for transitioning from monolithic architectures to microservices.

Rademacher et al. [30] examine the automatic decomposition of monolithic systems into microservices to reduce redesign costs. The study employs similarity measures between domain entities and finds that no specific combination of these measures consistently yields optimal decomposition results. Furthermore, the research highlights the feasibility of incremental migrations and the positive correlation between coupling and complexity, suggesting that progressive modularization can be a practical approach for achieving smoother and more manageable transitions. The work provides detailed insights into evaluation techniques and redesign methods applied to complex monolithic architectures, laying a robust foundation for future studies on transforming intricate software architectures.

Qian et al. [28] propose an innovative methodology for extracting microservices from monolithic applications. This approach leverages two distinct views: structural dependency and business function. The fusion of these views facilitates more accurate and efficient microservice identification. Combining topic modeling and graph clustering techniques, the method produces a set of services derived from the original software. Applied to 200 open-source projects, the developed open-source tool demonstrated favorable cohesion and modularity metrics. The study significantly contributes to the field by offering a structured and quantitative approach to microservice identification and extraction, emphasizing the importance of structural and functional analysis for transitioning monolithic architectures to microservices.

Lars van Asseldonk [32] investigates the impact of using multiple perspectives on the quality of microservice decompositions. Utilizing a Python tool to extract static, semantic, and dynamic dependencies from a monolith and represent them in a weighted graph, the Louvain algorithm was applied to identify highly connected communities. Validated on seven open-source Python projects, the methodology showed that combining static and dynamic information improves modularity compared to using these information types individually. However, including semantic information significantly

decreased decomposition quality due to the increased dependencies, which favored semantics over static and dynamic dependencies. This study provides valuable insights into the complexities of decomposing monolithic systems into microservices and highlights the nuanced effects of using multi-view clustering approaches for achieving high-quality modularization.

The existing literature on decomposing monolithic systems into microservices highlights several approaches, each with unique advantages and limitations. Rademacher et al. [30] focus on similarity measures between domain entities, but this method does not capture all complexities of class dependencies. Qian et al. [28] employ structural dependency and business function views, which can be computationally intensive and are better suited for projects with clear structural separations. Van Asseldonk [30] uses a modified Louvain algorithm with multiple perspectives, finding that semantic information increases dependencies and reduces decomposition quality. In contrast, in this study leverages complex network analysis with Python's NetworkX library to detect modularity in monolithic codebases. This approach comprehensively analyzes class dependencies and uses efficient algorithms like Greedy Modularity and Louvain, offering straightforward implementation applicable to various projects without unnecessary complexity.

3 Material and Methods

3.1 Macro Architecture

The macro architecture of the approach proposed in this paper for finding dependencies and potential modularization points is shown in Figure 1. It presents the sequence of actions from the user's perspective, who performs the algorithm analysis, stores the results in the database, and finally visualizes the result through graphs.

In step 1, parametrization, the process begins with the user providing some parameters necessary for the algorithm to perform a complete analysis. These parameters include the project path to be analyzed and the name of the Java project package, which Java uses to identify its classes and will be important for helping to search for dependencies between the classes.

In step 2, the pre-processing starts with reading the project to be analyzed. During this stage, the algorithm searches all the project's classes and their dependencies. The class names and their dependencies are stored to generate the dataset for the next stage, step 3. The class names will be the nodes, and the dependencies will be the edges that will compose the graph to be

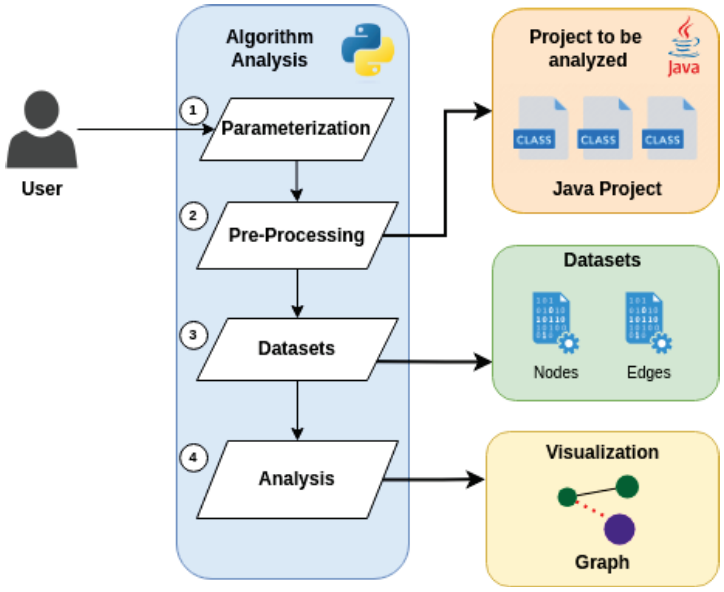

Figure 1 Macro architecture of the project.

analyzed. This information results in two files with nodes and edges that will create the graph for the subsequent analysis.

In step 4, analysis, the text files with nodes and edges are used to generate the graph, and community detection analyses are applied using modularity detection algorithms provided by the *NetworkX* library, as explained further in Section 3.4. The output includes visualizations of the graphs with the detected communities highlighted in different colors, as well as the separation points of the communities, where the edges are shown in red dashed lines to highlight these points. Graphs of each community are also generated separately, along with the identification of each class, to facilitate understanding which classes will comprise each microservice.

3.2 Process Flow

For a detailed understanding of the proposed approach, Figure 2 illustrates the flow of the entire functional process of the work, which is divided into three sections: the first shows the actions for which the user is responsible, the second describes the actions performed by the system, identified in the "Algorithm" layer, and the third section, called "Visualization", displays the results in the final step.

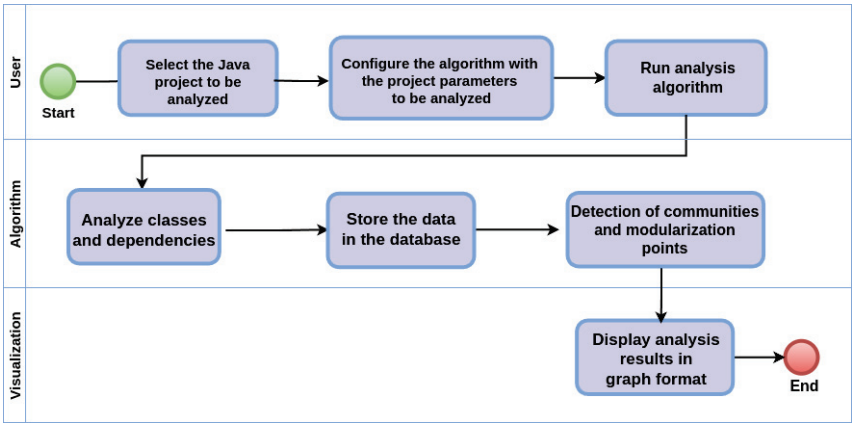

Figure 2 Process flow.

The process begins with the user selecting the project to be analyzed. Next, it is necessary to parameterize the algorithm of the Java package used at the beginning of each class because it is through this package name that the algorithm maps the interconnected classes. After adjusting the parameters, it is possible to start the analysis process. The algorithm begins to analyze the classes, searching for all the `.java` files and examining the dependencies of each one, storing this information in a relational database. In the next step, communities are identified, and the points where modules can be divided are determined. This information generates the visualization graphs of the communities and the graphs of each community separately, listing the classes that should remain together. To perform community detection, complex network algorithms are used.

Complex networks have applications in various areas of knowledge, and computing, they can be used to analyze code, identify vulnerabilities, and propose changes to projects based on the results found [26]. Community detection can indicate areas where complex code can be divided into smaller modules.

These qualities are essential for the algorithm proposed in this paper, which aims to identify classes that should remain together and the points of separation in the monolithic project. For community detection, four algorithms were employed, and their results were compared to determine the most suitable approach for the project under analysis. These algorithms are Greedy Modularity, Louvain, K-Clique, and Girvan–Newman, which will be explained in sequence.

3.2.1 Greedy Modularity

The Greedy Modularity algorithm is a widely used method for detecting communities in complex networks. It operates by maximizing the modularity of the network, a measure that quantifies the strength of division into communities by comparing the density of edges inside communities with that across different communities [6].

The algorithm begins by treating each node as an independent community. It then iteratively merges the pair of communities that results in the greatest increase in modularity. This merging process continues until no further increase in modularity is possible.

It is important to note that the Greedy Modularity algorithm follows a heuristic approach, meaning that while it is computationally efficient, it does not guarantee finding the optimal modularity value in all cases. Nevertheless, it is a widely adopted method due to its balance between efficiency and quality in community detection for large-scale networks [6].

3.2.2 Louvain

The Louvain method [4] is an advanced heuristic approach used for community detection in graphs. It aims to group the vertices of the graph into communities or groups that have a high density of internal connections and a lower density of external connections.

The method works iteratively, initially assigning each node as a separate community. In each iteration, vertices are reallocated between communities to maximize modularity, a measure that quantifies the quality of the division into communities:

$$Q = \frac{1}{2m} \sum_{ij} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \tag{1}$$

where:

- A_{ij} represents the adjacency matrix
- k_i and k_j denote the degrees of nodes i and j
- m is the total number of edges in the network
- $\delta(c_i, c_j)$ equals 1 if nodes i and j belong to the same community, otherwise 0.

After local optimization, the communities are aggregated into single nodes, forming a reduced network where the process is repeated. This iterative process continues until no further improvements in modularity are possible.

Compared to the Greedy Modularity approach, the Louvain method is computationally more efficient and well-suited for large networks. Due to its effectiveness, it has been widely applied in fields such as social network analysis, biology, and recommendation systems [15].

3.2.3 K-clique

A k-clique in a graph is a subgraph where the distance between any two vertices is not greater than k [5]. Mathematically, this condition can be represented as:

$$\forall i, j \in C, \quad d(i, j) \leq k \tag{2}$$

where $d(i, j)$ represents the shortest path between vertices i and j.

The special case of a 1-clique corresponds to a traditional clique, where all nodes are directly connected. A 2-clique is a maximal subgraph in which all nodes are connected by at most two edges. This characteristic is particularly relevant in social network analysis, where indirect relationships (e.g., "friend of a friend" connections) play an important role. Platforms like *LinkedIn*[1] leverage this concept to define second- and third-degree connections [5].

In the literature, the k-clique method has been widely used for community detection in large-scale networks, particularly in identifying overlapping communities [13]. Studies have applied it to social network analysis and other domains where community structures are not strictly disjoint but rather interwoven [2].

3.2.4 Girvan–Newman

The Girvan–Newman method is a divisive approach that uses the concept of edge betweenness centrality. Unlike modularity-based methods that attempt to merge nodes into communities, the Girvan–Newman algorithm progressively removes edges with the highest centrality scores, breaking the network into separate components. This measure is a generalization of vertex centrality, which measures the influence of a vertex on the others in the network. While vertex centrality counts the number of paths that pass through a particular vertex, edge centrality counts the number of paths that pass through the ends of the edge [8].

The edge betweenness centrality is defined as:

$$B(e) = \sum_{s \neq t} \frac{\sigma_{st}(e)}{\sigma_{st}} \tag{3}$$

[1] https://www.linkedin.com.

where:

- σ_{st} represents the total number of shortest paths between nodes s and t
- $\sigma_{st}(e)$ denotes the number of shortest paths passing through edge e.

The algorithm follows these steps:

1. Compute the betweenness centrality for all edges.
2. Remove the edge with the highest centrality.
3. Recalculate centrality values for the remaining edges.
4. Repeat until all edges are removed.

The removal of highly central edges progressively reveals underlying community structures. This method is particularly effective in networks where communities are connected by a few crucial bridges.

The Girvan–Newman algorithm has been applied in various domains, including social media link analysis [31] and biological network modularization [33]. Despite its effectiveness in detecting well-separated communities, its high computational complexity makes it less suitable for large networks compared to modularity-based methods.

3.2.5 Comparison summary

To facilitate the evaluation of the different community detection algorithms used in this study, Table 1 presents a comparative overview of their key characteristics. Each algorithm employs a distinct approach to identifying modular structures within complex networks, relying on different primary measures and optimization strategies. The comparison highlights their methodological differences, efficiency levels, and applicability in various contexts. This summary serves as a decision support reference, allowing domain experts to assess which algorithm best aligns with the specific modularization requirements of their project.

Table 1 Comparison summary between the algorithms

Algorithm	Primary measure	Characteristic
Louvain	Modularity (Q)	Heuristic method, efficient for large networks
Greedy Modularity	Modularity (Q)	Greedy method, less efficient than Louvain
k-clique	k-clique	Based on forming interlinked cliques
Girvan–Newman	Edge betweenness ($B(e)$)	Removes critical edges to reveal communities

3.3 Comparison of Algorithm Results

The process of *community detection* in complex networks involves multiple algorithms, each employing distinct heuristics and optimization criteria. To ensure a comprehensive evaluation, it is essential to compare the results obtained from different community detection methods and assess their practical relevance in real-world applications.

In this study, we compare the outcomes of the **Louvain, Greedy Modularity, k-clique, and Girvan–Newman algorithms**, focusing on key structural metrics such as *modularity score, number of detected communities, and community size distribution*. These metrics provide insight into the granularity of the detected communities and the extent to which each algorithm effectively captures meaningful structures within the network [11].

Beyond the quantitative comparison, the **validation of the identified modules' practical usefulness** is a crucial aspect of the analysis. This validation involves two complementary approaches:

3.3.1 Structural validation

The detected modules are evaluated based on their internal coherence and separation from other communities. A higher modularity score generally indicates well-defined communities; however, excessively high values may suggest over-partitioning, leading to fragmentation of functionally related components [24]. To account for this, we analyze whether the detected communities exhibit strong intra-community connections while maintaining minimal inter-community links [29].

Ultimately, the interpretation and validation of these results require domain expertise. The comparison of detected communities must be conducted by someone familiar with the project, as they possess the necessary contextual understanding to assess whether the identified modules align with the expected structure. This evaluation should be based on the algorithm's output, taking into account both quantitative metrics and qualitative insights derived from the system being analyzed.

3.3.2 Domain-specific validation

The identified communities are analyzed within the context of the studied system to determine their *functional significance*. In the case of software engineering applications, this involves assessing whether the detected modules align with logical software components, such as packages or architectural layers [21]. If the network represents a *social or biological system*, the

relevance of the communities is validated by examining known relationships, domain-specific metadata, or expert feedback [25].

By combining these evaluation methods, we aim to determine which algorithm provides the most **meaningful and interpretable** community structures for the given dataset. Furthermore, the insights gained from the comparative analysis contribute to a better understanding of how different community detection strategies impact the identification of modular structures in complex networks.

3.4 NetworkX Library

The *NetworkX* library[2] is widely used in *Python* for graph analysis and community detection in complex networks. It includes the implementation of various algorithms and tools for graph analysis and community detection [22].

This library implements algorithms such as *Greedy Modularity*, *k-clique*, *Louvain*, and many others. The *NetworkX* library offers functions for graph visualization and analysis of centrality, shortest paths, and other properties of complex networks.

3.5 Static Analysis

The algorithm developed in this study employs static analysis to scrutinize the source code written by programmers. This analysis is pivotal for mapping the dependencies among different classes within the project. Static analysis, as opposed to dynamic analysis, involves examining the code without actually executing it. This approach is effective in understanding the structural and syntactic aspects of the codebase, which is essential in identifying how various classes and components are interconnected.

Recent studies have highlighted the importance and accuracy of dependency analysis in static architecture compliance checking. For instance, Pruijt et al. [27] examined various tools for their ability to report dependencies accurately, finding significant differences in their performance and emphasizing the need for precise dependency mapping to avoid architectural erosion in software projects. Similarly, Beller et al. [3] conducted a large-scale evaluation of static analysis tools in open-source projects, revealing that while these tools are prevalent, their configurations often remain static and minimally customized after initial setup. This insight suggests that while static

[2]https://networkx.org/.

analysis tools are widely used, there is a gap in their dynamic application and continuous improvement in real-world projects.

These findings underline the necessity for continuous refinement and validation of static analysis tools to ensure they meet the evolving needs of software development environments. By leveraging robust static analysis methodologies, as explored in these studies, the algorithm in this research aims to provide a reliable mapping of class dependencies, forming a foundation for further architectural optimizations and refinements.

It's important to note that static analysis in this context is focused solely on dependency analysis and does not delve into identifying potential code issues like bugs or stylistic errors. However, for those purposes, the market offers a range of specialized tools designed for the *Java* language. Notable examples include:

- *FindBugs*[3]: A tool that examines Java bytecode to detect potential bugs in the code, offering insights into possible improvements.
- *Checkstyle*[4]: This tool focuses on code styling, ensuring that the code adheres to a set coding standard, which helps maintain consistency and readability across the codebase.
- *PMD*[5]: It scans Java source code to identify potential flaws, such as unused variables, empty catch blocks, unnecessary object creation, and more.

These tools complement the static analysis performed by the algorithm proposed here, as they provide an additional layer of code quality assurance, focusing on aspects beyond class dependencies. Referencing these tools, as mentioned by Louridas [18], highlights the comprehensive nature of code analysis, combining structural insights with quality checks. The integration of our dependency analysis algorithm with these tools can offer a holistic approach to understanding and improving the overall quality of the software project.

3.6 Programming Language Integration for Static Analysis

To analyze *Java*-based projects, it has been chosen to utilize the *Python* language for the development of our code analysis mechanism and to implement complex network techniques. The choice of *Python* is strategic, given its proficiency in text processing and analysis, which are crucial in

[3]http://findbugs.sourceforge.net/.
[4]http://checkstyle.sourceforge.net.
[5]http://pmd.sourceforge.net/.

comprehending the structure and relationships of the classes under scrutiny. Additionally, *Python*'s compatibility with complex network libraries further reinforces its suitability for this task.

The static analysis itself is conducted on the *Java* language, a decision motivated by *Java*'s inherent project structure standardization. This standardization greatly aids in the identification of classes and their interdependencies. Key structural elements in a *Java* project, such as the alignment of file names with class names, the mapping of file locations to class paths, and the explicit declaration of dependencies within the code, provide vital information for graph construction. This structured approach simplifies the identification of modules and inter-class interactions, enabling our algorithm to effectively discern the existing dependencies within the project.

The dependency search is executed by the *Python*-developed algorithm, which scans for `.java` files based on the initialization parameter of the application, corresponding to the project's package name. For instance, if the input package is `br.com.example`, the algorithm analyzes all `.java` files within the `br/com/example` directory of the project.

During this analysis, each file is meticulously examined. `Imports` beginning with `br.com.example` are identified as internal class references, signifying connections between the analyzed class and others within the project. These relationships are cataloged in a relational database, where the primary class is listed in the `name` field, and its associated classes are recorded in the `friend` field. This format mirrors a social network, where interconnected classes reflect the dynamics of a network of friendships, illuminating the web of interactions and dependencies within the project.

A practical illustration of this methodology is presented in Figure 3, showcasing a code snippet from an open-source *Flickr* project found on GitHub, exemplifying the typical structure of *Java* classes.

1. The reserved word `package` indicates the package to which this class belongs. All the information displayed afterward indicates the physical path where that file is located.
2. This information indicates the physical location of the Java file in the project structure, as shown in Figure 4.
3. The reserved word `import` is used to declare the dependencies that are imported into the class.
4. After the reserved word `import`, if the information matches that indicated after the `package`, it suggests that these dependencies are internal, i.e., other classes belonging to the same project.

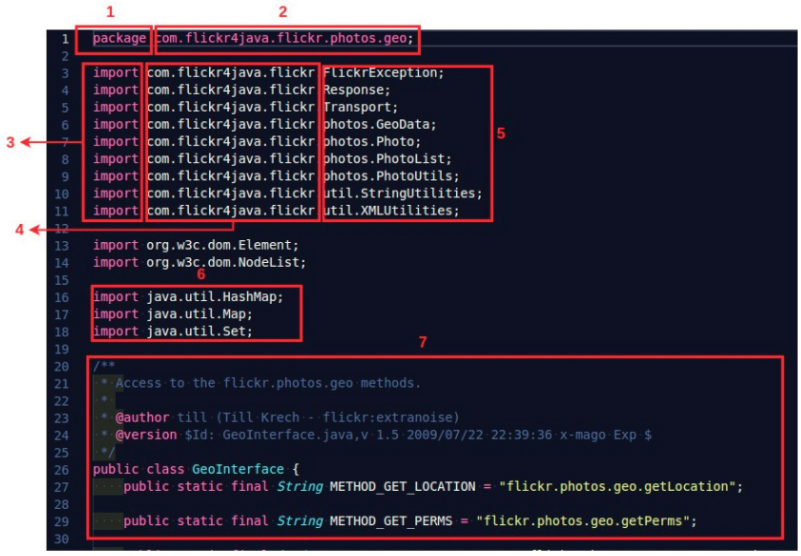

Figure 3 Java Code Example – *Flickr*.

5. The end of the information in the `imports` indicates potential new physical divisions of the project and the names of the files or classes that are imported.
6. Importing classes external to the project, or third-party libraries, the reserved word `import` is used followed by information different from that indicated after the `package`.
7. The beginning of the Java class, where all the instructions will be written by the developer.

3.7 Source Codes to be Analyzed

The source codes that were used for the analysis of the algorithm proposed in this work were extracted from *GitHub*[6] under open-source licensing or using codes voluntarily provided by their owners for use. Various projects with different sizes and structures were tested to validate the proposal of the developed algorithm.

The selection of projects for validating the proposed code was based on two main criteria. The first criterion was the availability of the projects on

[6]https://github.com/.

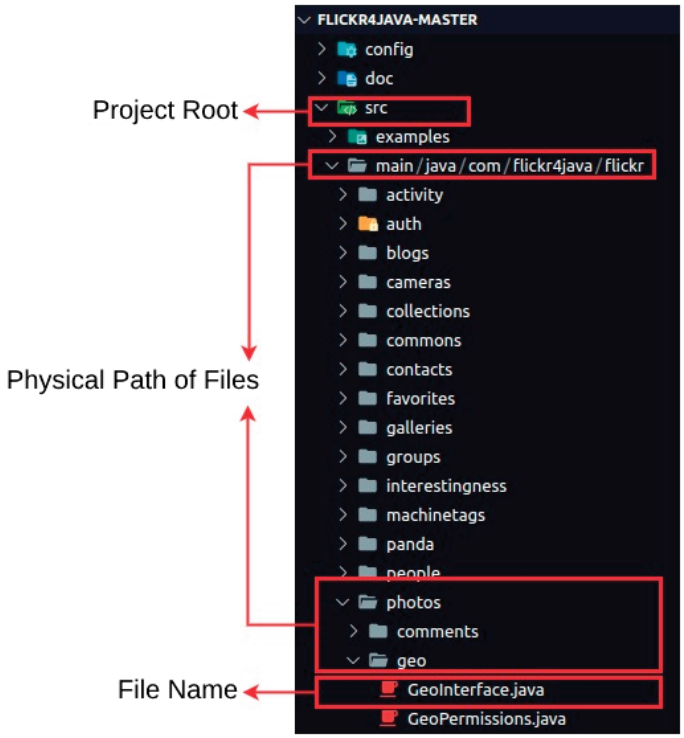

Figure 4 Physical structure of a Java project.

GitHub, ensuring full transparency and enabling their complete reproduction—an essential factor for the replicability of the experiments. The second criterion was the suitability of the projects for the study's objective, which focuses on identifying and separating monolithic Java code.

To ensure that the selected projects met the necessary characteristics, structural aspects such as codebase size and the number of classes in each repository were analyzed. These metrics serve as indicators of system complexity and modularity, allowing an assessment of the applicability of the proposed method in real-world scenarios.

4 Experimental Results

Experiments were conducted to validate the efficiency of the algorithm proposed in identifying dependencies of *Java* projects and in the automated detection of communities. Four different community detection algorithms in

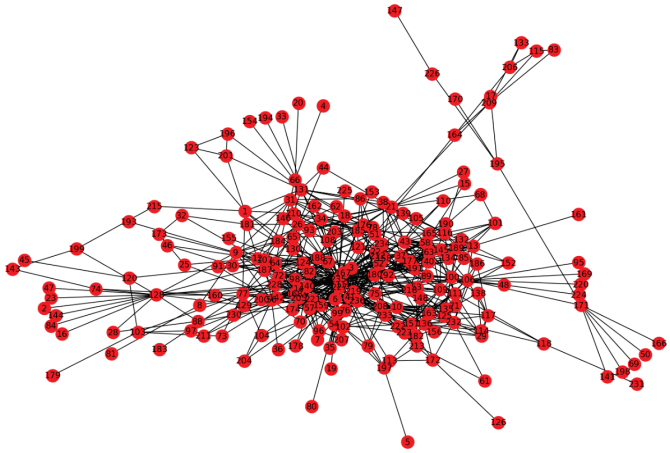

Figure 5 Linkage between classes.

Table 2 Modularity measure and quantitative analyses

Analysis	Result
Modularity measure:	0.07615483221665743
Number of graph nodes:	211
Number of graph edges:	679
Graph density:	0.0306477093206951
Average degree:	6.436018957345971
Maximum degree:	75
Graph clustering coefficient:	0.2672128326145455
Average graph path length:	3.249785601444369
Degree sequence:	[1, 2, 15, 5, 11, 11, 7, 15, 10, ...]
Community classes:	[1, 57, 3, 34, 55, 94, 103, 161, ...]

complex networks were used to identify the communities and present the modules that can be separated.

The results of the experiments provided a comprehensive analysis of the complete graph before community detection. This allowed for a full view of the existing connections between the classes of the project. Figure 5 illustrates the result of analyzing a *Java* code.

In Table 2, a graph analysis was generated with the following information: number of nodes, number of edges, graph density, average degree, maximum degree, clustering coefficient, average path length, modularity measure, degree sequence, and community classes. This information was used to obtain a more complete and detailed understanding of the structure and properties of the graph [7].

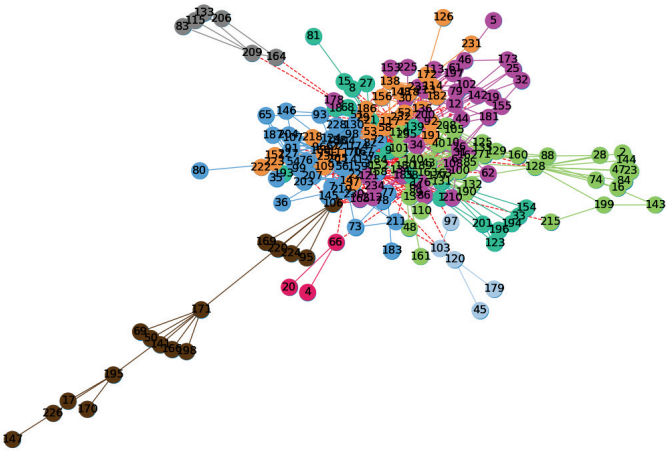

Figure 6 Detected communities.

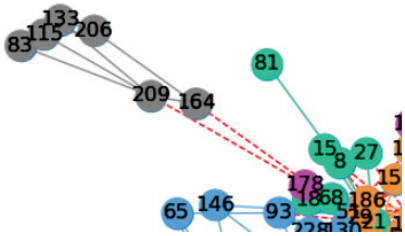

Figure 7 Connections that should be broken.

The analysis of the complex networks algorithm resulted in a graph, where the communities are identified and represented by different colors, as illustrated in Figure 6. Connections between communities that should be divided are highlighted by dashed and red edges, emphasizing the separation points, as shown in Figure 7.

Each discovered community was analyzed separately, only with the classes and dependencies, as done in Figure 8. In addition to the presented graph, for each found community, Table 3 shows the names of the classes and their connections that need to be separated.

To facilitate the comparison of results between different algorithms, distribution maps showing the number of communities found and the number of nodes in each community for each analyzed algorithm are presented, as illustrated in Figure 9.

These distribution maps provided a clear and concise visualization of the characteristics of the communities identified by each algorithm, allowing for

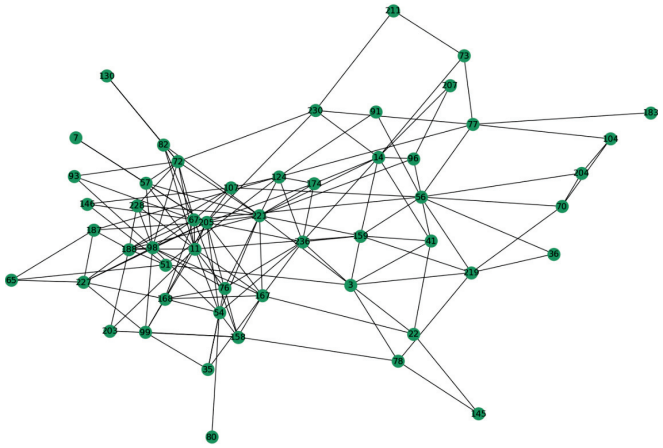

Figure 8 Connections that should be broken.

Table 3 Connections between classes to be split

Class	Split	Class
poc.consulta.ws.controller.testecontroller	–>	consulta.core.util.helper.messagehelper
poc.consulta.ws.controller.testecontroller	–>	consulta.core.model.enums.tipoconsumidor
consulta.core.util.helper.messagehelper	–>	consulta.ws.controller.consulentecontroller
consulta.core.util.helper.messagehelper	–>	consulta.core.exception.consultasemrespostaexception
consulta.core.util.helper.messagehelper	–>	framework.exception.handler.appexceptionhandler
consulta.core.util.helper.messagehelper	–>	consulta.ws.controller.consumidorcontroller
consulta.core.util.helper.messagehelper	–>	consulta.ws.controller.defaultcontroller
poc.consulta.infrastructure.repository.impala.operacaoclienteimpalatestpocrepository	–>	framework.jpa.impalarepository
poc.consulta.infrastructure.repository.impala.operacaoclienteimpalatestpocrepository	–>	consulta.core.model.enums.tipomovimento
poc.consulta.infrastructure.repository.impala.operacaoclienteimpalatestpocrepository	–>	consulta.core.model.enums.tipopagamentoavulso
poc.consulta.infrastructure.repository.impala.operacaoclienteimpalatestpocrepository	–>	consulta.core.model.enums.simnao
poc.consulta.infrastructure.repository.impala.operacaoclienteimpalatestpocrepository	–>	consulta.core.model.enums.tipopagamento
poc.consulta.infrastructure.repository.impala.operacaoclienteimpalatestpocrepository	–>	consulta.core.model.enums.tipoconsumidor
consulta.ws.controller.defaultcontroller	–>	consulta.core.model.enums.modalidadeoperacao
consulta.ws.controller.defaultcontroller	–>	consulta.core.exception.consultainvalidaexception
consulta.ws.controller.defaultcontroller	–>	consulta.core.model.enums.tipoconsumidor
consulta.core.model.enums.tipoconsumidor	–>	consulta.infrastructure.repository.impala.movimentoimpalarepository
consulta.core.model.enums.tipoconsumidor	–>	consulta.ws.controller.consulentecontroller

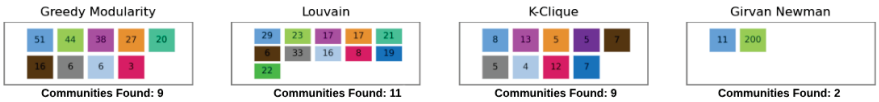

Figure 9 Connections that should be broken.

a direct comparison of the obtained results. It is possible to observe variations in the number of communities found and the size of the communities in each algorithm.

This comparative analysis is valuable for assessing the performance and differences between the algorithms, highlighting those that produce consistent results, identify distinct communities, or exhibit greater precision in identifying the nodes in each community.

By using these distribution maps as a measure of comparison, it is possible to gain important insights for selecting the most appropriate algorithm for the needs of the study in question, taking into account the structure and complexity of the analyzed network.

4.1 Case Study

To evaluate the effectiveness of the algorithm proposed in this work, experiments were conducted using two distinct open-source systems: *Flickr*[7] and *Jenkins*[8]. These experiments aim to determine which algorithm is most suitable for modularizing the code in question. The result does not seek to elect a winner, but to assist the developer in decision-making by comparing which result can better meet the project's needs more efficiently.

4.1.1 Graph analysis

Graphs representing the connections between the classes of the *Flickr* and *Jenkins* projects were generated before the community discovery analysis. In addition, a detailed analysis of the characteristics of these graphs was conducted. The graphs can be fully observed in Figures 10 and 11, respectively. Quantitative analyses, including measures such as the number of nodes, the number of edges, and other relevant metrics, are presented in Table 4. These pieces of information provide a comprehensive view of the graph structures and assist in understanding the connections between the classes in the respective projects.

Table 4 Graph comparison between *Flickr* and *Jenkins* projects

Analysis	Flickr	Jenkins
Number of graph nodes:	96	149
Number of graph edges:	351	210
Graph density:	0.07697368421052632	0.019045891529113006
Average degree:	7.3125	2.8187919463087248
Maximum degree:	40	82
Graph clustering coefficient:	0.062487829880921976	0.20762409618600694
Average graph path length:	2.9405701754385967	3.2275530564121166
Modularity measure:	0.06040941226126406	0.4539682539682541
Degree sequence:	[2, 4, 20, 36, 13, 13, ...]	[1, 82, 1, 30, 3, ...]
Community classes:	[2, 53, 144, 5, 8, ...]	[1, 138, 3, 113, 4, ...]

[7]https://github.com/boncey/Flickr4Java.

[8]https://github.com/jenkinsci/jenkins.

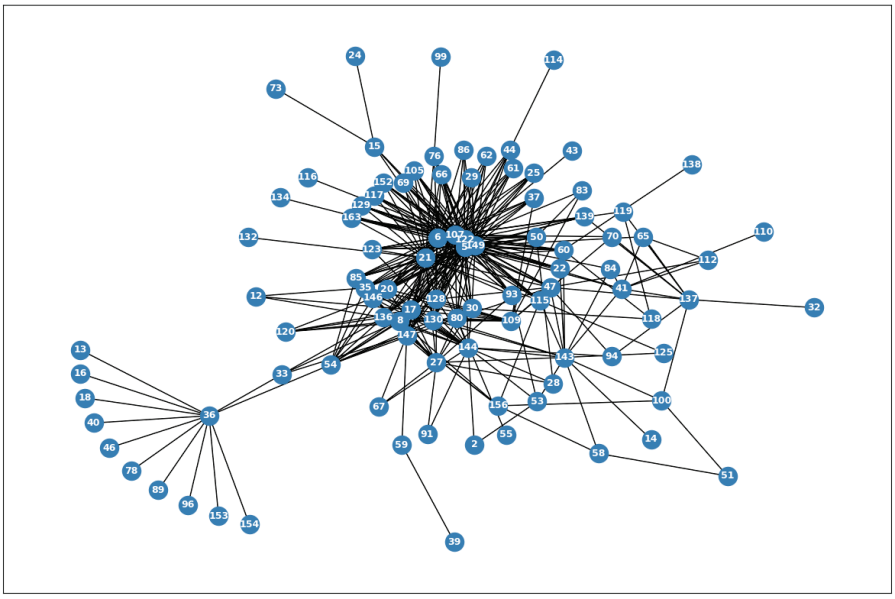

Figure 10 *Flickr*. Dependency graphs before community discovery.

Based on the presented data, the *Jenkins* project demonstrates having a well-defined and consistent graph. Although it contains a higher number of nodes, the number of edges is relatively lower, which suggests the presence of well-defined classes, with fewer connections to other classes.

Other indicators also point to a more densely connected graph, as evidenced by the clustering coefficient, indicating a greater tendency to form groups or communities. This information directly complements the modularity measure, which assesses the graph's structure and the effective division into distinct communities, presenting a value closer to 1 compared to the *Flickr* project.

4.1.2 Community discovery

The visual outcomes of the selected algorithms are distinctly presented. For the greedy modularity algorithm, the Flickr and Jenkins project graphs are shown in Figures 12 and 13, respectively. The *Louvain* method's results are depicted in Figures 14 (Flickr) and 15 (Jenkins), while the k-clique approach graphs are in Figures 16 (Flickr) and 17 (Jenkins). Finally, the *Girvan Newman* method's detailed results for Flickr and Jenkins are in Figures 18 and 19, respectively. These figures collectively provide a comprehensive perspective

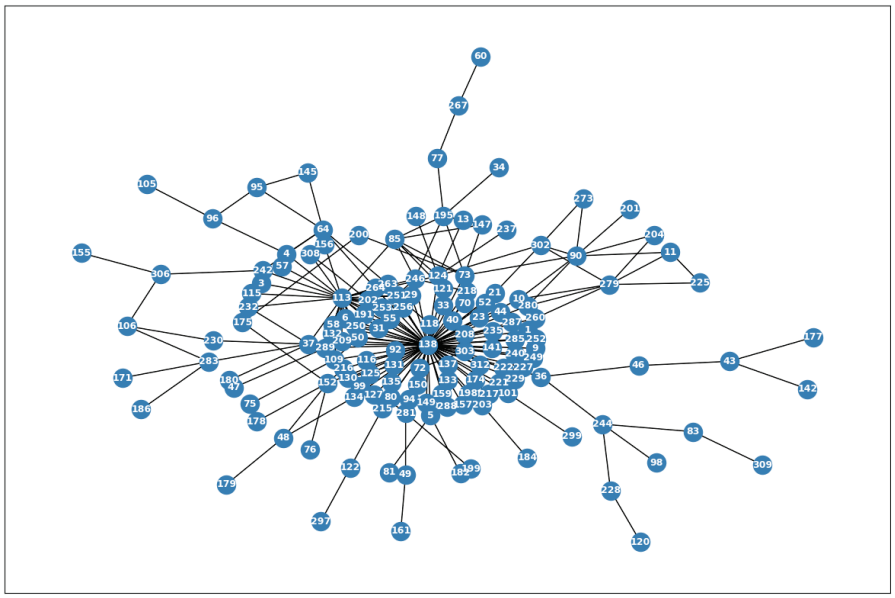

Figure 11 *Jenkins*. Dependency graphs before community discovery.

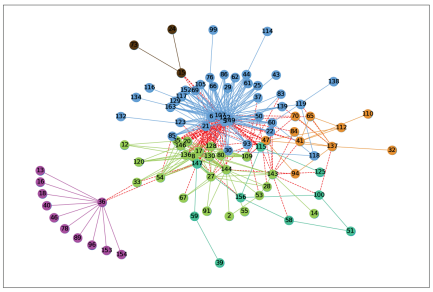

Figure 12 *Flickr – Greedy Modularity.*

on the structure and characteristics of each graph, highlighting the differences and similarities across the methodologies and their impacts on the projects.

4.1.3 Distribution map

For a comparative overview, the distribution map presents information about the number of communities found in each algorithm, the number of classes in each community, and the number of dependencies that need to be separated in each algorithm. This comparison provides a simplified representation of the same information present in the graphs in Section 4.1.2, facilitating

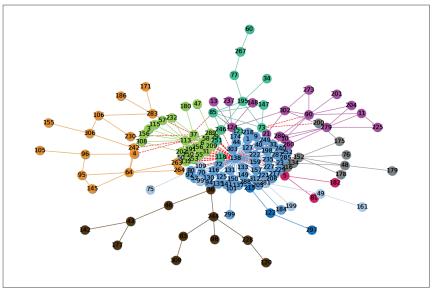

Figure 13 *Jenkins – Greedy Modularity.*

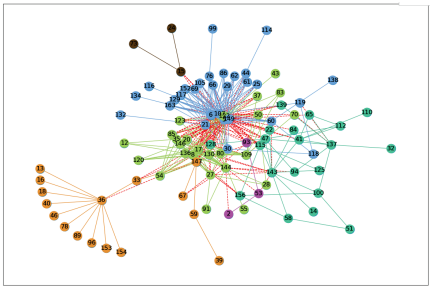

Figure 14 *Flickr – Louvain.*

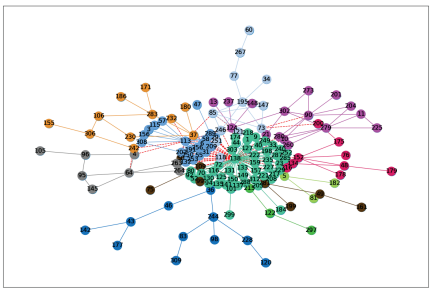

Figure 15 *Jenkins – Louvain.*

visualization and analysis. In the *Flickr* project, this information can be visualized in Figure 20, while in the *Jenkins* project, it can be observed in Figure 21. These distribution maps offer a summarized and intuitive view of the modularization characteristics of each algorithm in the respective projects.

Upon analyzing the generated maps, it was observed that the Greedy Modularity and *Louvain* algorithms showed similar performance in the

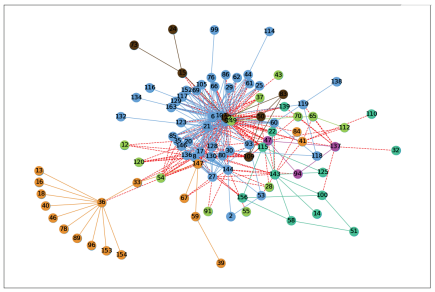

Figure 16 *Flickr – k-clique.*

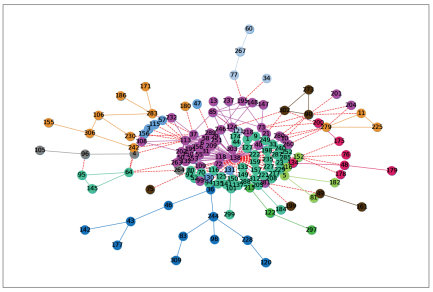

Figure 17 *Jenkins – k-clique.*

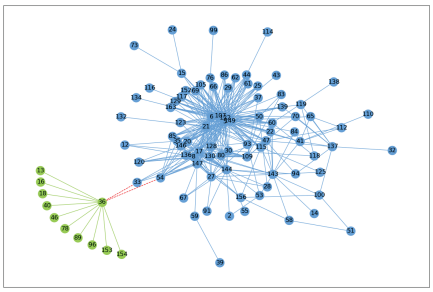

Figure 18 *Flickr – Girvan–Newman.*

analysis of both projects in terms of the number of communities, although they varied in the number of classes within each community.

Regarding the *k-clique* algorithm, it was configured to identify communities up to 3 cliques away. The results showed a closer alignment with the *Flickr* project compared to the other two algorithms in terms of the number of communities, but there was a significant difference in the number of classes.

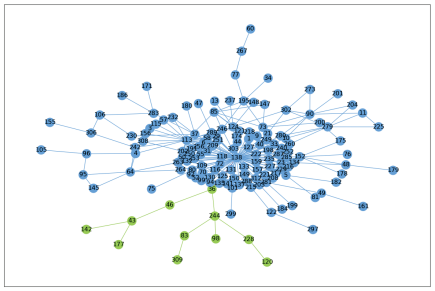

Figure 19 *Jenkins – Girvan–Newman.*

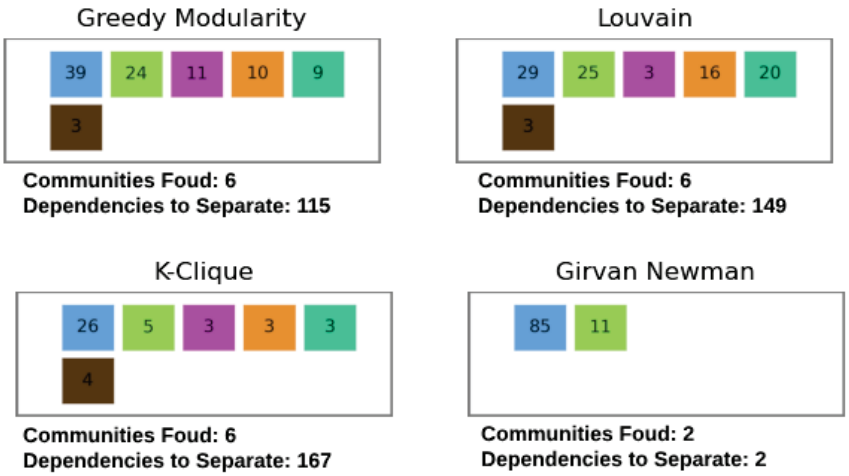

Figure 20 Analysis of the algorithms for the *Flickr* project.

However, for the *Jenkins* project, the results were quite different from the other analyses, even when compared to the *Flickr* results, both in terms of the number of communities and community classes.

In the analysis using the *Girvan–Newman* algorithm, it was observed that the number of communities in the projects was the same in both projects, with only two communities identified. This indicates that the performance of the algorithm, after a single execution, was not as efficient compared to other community detection methods.

In addition to the results presented, graphs corresponding to each community and each algorithm for all projects were generated, as illustrated in Figure 8. A table was created containing the classes that need to be separated, as demonstrated in Table 3.

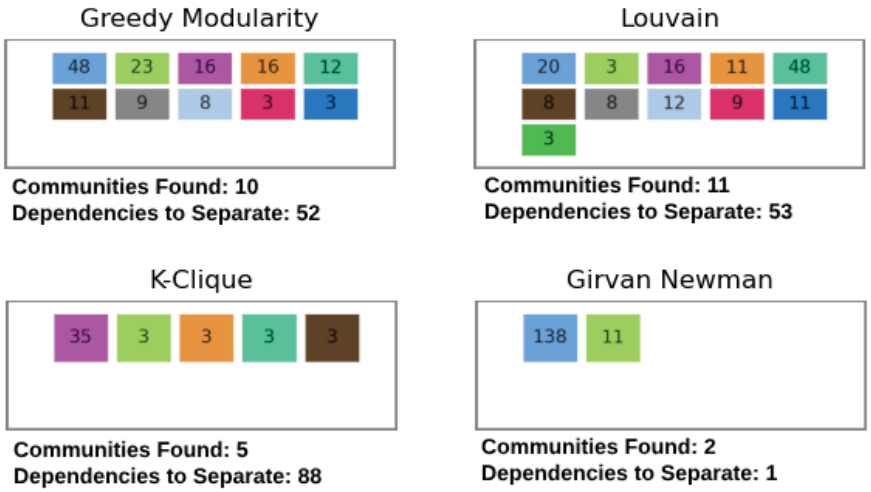

Figure 21 Analysis of the algorithms for the *Jenkins* project.

5 Conclusions and Future Work

This study presented an automated approach for modularizing monolithic Java projects by detecting communities in complex networks using the *NetworkX* library in *Python* and applying the *Greedy Modularity*, *Louvain*, *k-clique*, and *Girvan–Newman* algorithms. This technique aims to improve code maintainability and developer productivity by effectively segmenting the codebase into smaller, structured modules.

The results indicate that the *Greedy Modularity* and *Louvain* algorithms produced more consistent community distributions across different projects. *K-clique* exhibited variable performance, while *Girvan–Newman* yielded a low number of communities, requiring further refinements. To enhance accuracy, future research should explore recursive executions of *Girvan–Newman* on detected communities.

Further optimizations in *k-clique* and *Girvan–Newman* parameterization should be investigated to achieve better alignment with expected modularization patterns.

For *k-clique*, varying the clique size and distance parameters could provide deeper insights into class distribution. Similarly, refining the edge centrality threshold in *Girvan–Newman* may enhance community separation. These adjustments aim to improve precision in modularization and ensure a meaningful comparison with other algorithms.

Future work should also expand the analysis to additional programming languages, such as *Ruby, C#, and Python*, which share structural similarities with Java, facilitating adaptation.

Furthermore, incorporating quantitative metrics is crucial for evaluating community detection consistency and effectiveness. Statistical measures such as *normalized mutual information (NMI)* and *adjusted Rand index (ARI)* should be applied to quantify clustering stability and improve cross-algorithm comparisons.

To validate real-world applicability, this technique will be tested in real software projects, supervised by domain experts and the authors of this study. This assessment will measure its effectiveness in practical development environments, ensuring its usability and adaptability.

Finally, exploring *graph neural networks (GNNs)* could provide a more robust approach to analyzing graph structures. GNNs capture deeper relationships between code elements, offering a promising alternative to the complex network techniques used in this study.

References

[1] Omar Al-Debagy and Peter Martinek. A comparative review of microservices and monolithic architectures. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000149–000154. IEEE, 2018.

[2] Punam Bedi and Chhavi Sharma. Community detection in social networks. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 6(3):115–135, 2016.

[3] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481, 2016.

[4] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[5] Luís Cavique, Armando B Mendes, and Jorge Santos. An algorithm to discover the k-clique cover in networks. In *Portuguese Conference on Artificial Intelligence*, pages 363–373. Springer, 2009.

[6] Mingming Chen, Konstantin Kuzmin, and Boleslaw K Szymanski. Community detection via maximization of modularity and its variants. *IEEE Transactions on Computational Social Systems*, 1(1):46–65, 2014.

[7] Thamires Lopes das Mercês. Análise de métodos de detecção de comunidades em redes complexas.

[8] Ljiljana Despalatović, Tanja Vojković, and Damir Vukicević. Community structure in networks: Girvan-newman algorithm improvement. In *2014 37th international convention on information and communication technology, electronics and microelectronics (MIPRO)*, pages 997–1002. IEEE, 2014.

[9] Saru Dhir, Deepak Kumar, and VB Singh. Success and failure factors that impact on project implementation using agile software development methodology. In *Software engineering*, pages 647–654. Springer, 2019.

[10] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, United States of America; Massachusetts, 3 edition, 2004.

[11] Santo Fortunato and Darko Hric. Community detection in networks: A user guide. *Physics Reports*, 659:1–44, 2016.

[12] Martin Fowler. Monolithfirst, 2015.

[13] Enrico Gregori, Luciano Lenzini, and Simone Mainardi. Parallel k-clique community detection on large-scale networks. *IEEE Transactions on Parallel and Distributed Systems*, 24(8):1651–1660, 2012.

[14] Jesse Griffin. Hexagonal-driven development. In *Domain-Driven Laravel*, pages 521–544. Springer, 2021.

[15] Darko Hric, Richard K Darst, and Santo Fortunato. Community detection in networks: Structural communities versus ground truth. *Physical Review E*, 90(6):062805, 2014.

[16] Daniel Izquierdo-Cortazar, Gregorio Robles, Felipe Ortega, and Jesus M Gonzalez-Barahona. Using software archaeology to measure knowledge loss in software projects due to developer turnover. In *2009 42nd Hawaii International Conference on System Sciences*, pages 1–10. IEEE, 2009.

[17] James Lewis and Martin Fowler. Microservices, 2014.

[18] P. Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, 2006.

[19] Robert C Martin, James Grenning, Simon Brown, Kevlin Henney, and Jason Gorman. *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall, 2018.

[20] Harvey Maylor and Neil Turner. Understand, reduce, respond: project complexity management theory and practice. *International Journal of Operations & Production Management*, 2017.

[21] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[22] NetworkX Developers. Networkx. https://networkx.org/, 2023. Accessed: April 04, 2023.

[23] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[24] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.

[25] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818, 2005.

[26] Weifeng Pan, Bing Li, Yutao Ma, and Jing Liu. Multi-granularity evolution analysis of software using complex network theory. *Journal of Systems Science and Complexity*, 24(6):1068–1082, 2011.

[27] Leo Pruijt, Christian Köppe, Jan Martijn van der Werf, and Sjaak Brinkkemper. The accuracy of dependency analysis in static architecture compliance checking. *Software: practice and Experience*, 47(2):273–309, 2017.

[28] Lifeng Qian, Jing Li, Xudong He, Rongbin Gu, Jiawei Shao, and Yuqi Lu. Microservice extraction using graph deep clustering based on dual view fusion. *Information and Software Technology*, 158:107171, 2023.

[29] Martin Rosvall and Carl T. Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4):1118–1123, 2008.

[30] Samuel Santos and António Rito Silva. Microservices identification in monolith systems: functionality redesign complexity and evaluation of similarity measures. *Journal of Web Engineering*, 21(5):1543–1582, 2022.

[31] K Sathiyakumari and MS Vijaya. Community detection based on girvan newman algorithm and link analysis of social media. In *Digital Connectivity–Social Impact: 51st Annual Convention of the Computer Society of India, CSI 2016, Coimbatore, India, December 8-9, 2016, Proceedings 51*, pages 223–234. Springer, 2016.

[32] Lars van Asseldonk. From a monolith to microservices: the effect of multi-view clustering. Master's thesis, Utrecht University, 2021. Accessed: 2024-06-15.

[33] Ding Yanrui, Zhang Zhen, Wang Wenchao, and Cai Yujie. Identifying the communities in the metabolic network using'component'definition and girvan-newman algorithm. In *2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*, pages 42–45. IEEE, 2015.

Biographies

Marcos Cordeiro de Brito Jr received his Master's degree in Electrical Engineering and Computing from Universidade Presbiteriana Mackenzie in June 2023. Currently, he is the Head of Technology at SPC Grafeno, São Paulo, Brazil. He has had extensive experience in systems and software development since 2000. His skills include technical leadership, agile development, and continuous integration with tools like Jenkins, Docker, Kubernetes, and Git. Marcos is proficient in various programming languages, including Java, Python, and C#, and has strong knowledge of frameworks such as Spring Boot, Django, and Flask. Additionally, he has experience in agile methodologies, TDD, BDD, and cloud architectures using Google Cloud and AWS. He has also served as a professor at Paulista Faculty of Informatics and Administration (FIAP), teaching topics such as Spring Boot, MVC architecture, and Hibernate. Throughout his career, Marcos has participated in numerous national and international projects, significantly contributing to software innovation and quality in the organizations he has worked for.

Calebe P. Bianchini holds a Bachelor's degree in Computer Science from UFSCar, Brazil, where he also obtained his Master's degree. He completed his Ph.D. in Computer Engineering at POLI/USP, Brazil, in 2009. With a combined experience of over 19 years in high-performance computing, he has contributed to numerous research and development projects with Petrobras, Shell, Intel, and CERN, among others. Currently, he is an Associate Professor and researcher at Mackenzie Presbyterian University, where he also heads the MackCloud Lab, a Scientific Computing Center. He is an active member of the Brazilian High-Performance Computing community, particularly in organizing competitions and challenges in this domain. Additionally, he is an Ambassador for the NVIDIA Deep Learning Institute, where he conducts courses related to high-performance computing on NVIDIA GPUs.

Leandro A. Silva has a degree in Computer Engineering, a Master's degree and a Ph.D. from the USP Polytechnic School. He is currently a Professor at the Faculty of Computing and Informatics (FCI), in the Stricto-Sensu Academic Postgraduate Program in Electrical and Computer Engineering (PPGEEC), and in the Stricto-Sensu Professional Postgraduate Program in Applied Computing (PPGCA) and Coordinator Funding for Research (CFP) of the Dean of Research and Postgraduate Studies (PRPG) of Universidade

Presbiteriana Mackenzie (UPM). At UPM he has already held administrative activities as FCI Extension Coordinator and PPGEEC Coordinator. As for research activities, he participates in Program Committees for national and international conferences and also provides services as a technical reviewer for conferences and specialized journals. As a line of research, he has worked mainly in areas involving data science such as artificial neural networks, machine learning, data mining, big data, and pattern recognition. In these areas, he has regularly published scientific articles in the main national and international congresses, as well as in specialized journals. He is the main author of the textbook on Data Mining by Elsevier. Finally, Prof. Leandro Augusto co-leads the Laboratory and Research Group Big MAAp – Big Data and Applied Analytical Methods, where he develops Research and Development Projects with support from public agencies and companies from different segments of the production sector.