CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. REFINING SOFTWARE CLUSTERING: THE IMPACT OF CODE CO-CHANGES ON ARCHITECTURAL RECONSTRUCTION

we explore using code co-changes as input for software clustering for architectural reconstruction. Since structural dependencies are the most commonly used dependencies in software clustering, we investigate whether integrating them with code co-changes provides better results than using either dependency type alone.

Our experiments are applied to four open-source Java projects from GitHub. For each project, we apply three distinct clustering algorithms (Louvain, Leiden, and DBSCAN) and evaluate their performance using two clustering evaluation metrics. These metrics allow a comparison between clustering based solely on code co-changes and clustering that integrates both co-changes and structural dependencies, offering a better understanding of how these co-changes influence software architecture reconstruction.

1.1. Introduction

Software systems often need more documentation. Even if there was original documentation at the beginning of development, it may become outdated over the years. Additionally, the original developers may leave the company, taking with them knowledge about how the software was designed. This situation challenges the teams when it comes to maintenance or modernization. In this context, recovering the system's architecture is essential. Understanding the system's architecture helps developers evaluate better and understand the nature and impact of changes they must make. One technique to help in reconstructing the system architecture is software clustering. Software clustering involves creating cohesive groups (modules) of software entities based on their dependencies and interactions.

Among the dependencies that can be used for software clustering are structural dependencies (relationships between entities based on code analysis), lexical dependencies (relationships based on naming conventions), and code co-changes/logical dependencies (relationships between entities extracted from the version control system), and others.

This paper assesses the impact of logical dependencies in software clustering alone and combination with structural dependencies. The structural dependencies are used as they are extracted from static code analysis, while the logical dependencies are filtered co-changes obtained from the version control system [2]. The co-changes are filtered to enhance their reliability and remove noise caused by large commits with many files unrelated to development activities (e.g., formatting changes) or rare co-changes that may not indicate a true dependency [3].

The following research questions guide our investigation:

- **RQ1:** Does using structural dependencies (SD) combined with logical dependencies (LD) improve software clustering results compared to traditional approaches using only structural dependencies (SD)?
- **RQ2:** Can using only logical dependencies (LD) produce good software clustering results?
- **RQ3:** How do different filtering settings for logical dependencies (LD) impact clustering results, and which filtering settings provide the best performance?

To answer these research questions, we apply three different clustering algorithms (Louvain, Leiden, and DBSCAN) to different open-source projects. We then evaluate the results using two metrics: MQ (Modularization Quality) [4] and MoJoFM (Move and Join e**F**fectiveness **M**easure) [5]. The MoJoFM metric is used for external evaluation, evaluating against the perspective of the system's architect or developers. The MQ metric is used for internal evaluation based on the software structure itself. These two metrics allow us to compare the effectiveness of using structural and logical dependencies alone and combined. This comparison helps clarify how different dependencies and filtering choices affect clustering results.

In Section 1.2, we review the related work and previous studies that used various dependencies for software clustering and their metrics for evaluation. Section 1.3 provides an overview of structural and logical dependencies used in our approach, explaining how these dependencies are extracted. Section 1.4 details the workflow and implementation of our approach, including the extraction and filtering of dependencies and the clustering algorithm used. The plan and results of our experiments on four open-source projects are presented in Section 1.5. Section 1.6 evaluates our results using the Modularization Quality (MQ) metric and the MoJoFM metric. We also manually analyze some of the clustering solutions. Finally, Section **??** contains our conclusions and findings.

1.1.1. Abbreviations and Acronyms

The following abbreviations and acronyms are used throughout this article:

- **LD**: Logical Dependencies
- **SD**: Structural Dependencies
- **MQ**: Modularization Quality
- **MoJoFM**: Move and Join Effectiveness Measure

1.2. Related Work

Several studies have explored the use of different types of dependencies in software clustering, applying different algorithms to improve clustering results and using various metrics to evaluate the results obtained.

Tzerpos and Holt developed ACDC (Algorithm for Comprehension-Driven Clustering). This pattern-driven clustering algorithm uses subsystem structures such as source file patterns, directory patterns, system graph patterns, and support library patterns to detect similarities and create clusters [6]. For result evaluation, the au-

thors introduced the MoJo metric, which counts the minimum number of move and join operations required to transform one clustering result into another, assessing how close one clustering solution is to another [7], [8]. Later, Wen and Tzerpos introduced the MoJoFM metric, an enhanced version of the original MoJo distance metric for more effective measurements, as presented in more detail in subsection 1.4.2 [5].

Corazza et al. [9], [10] used lexical dependencies derived from code comments, class names, attribute names, and parameter names, applying Hierarchical Agglomerative Clustering (HAC) to group-related entities. For evaluating the results, the authors used a metric based on the MoJo distance metric and NED (Non-Extremity Cluster Distribution), which measures that the formed clusters are not too large or too small.

Andritsos and Tzerpos [8] used structural dependencies and nonstructural attributes, such as file names and developer names, and proposed the LIMBO algorithm, a hierarchical clustering algorithm for clustering software systems. They used the MoJo distance metric to evaluate the algorithm's output.

Anquetil et al. [11] also used lexical information, including file names, routine names, included files, and comments. They applied an n-gram-based clustering approach to detect semantic similarities between entities and evaluated the results using precision and recall metrics.

Maletic and Marcus [12] propose an approach to software clustering that uses semantic dependencies extracted using Latent Semantic Indexing (LSI), a technique for identifying similarities between software components. They apply the minimal spanning tree (MST) algorithm for clustering and evaluate the results using metrics based on both semantic and structural information.

Wu et al. [13] conducted a comparative study of six clustering algorithms using structural dependencies on five software systems. Four of the algorithms are based on agglomerative clustering, one on program comprehension patterns, and one algorithm is a customized version of Bunch [4]. The performance of these algorithms was evaluated using the MoJo metric and NED (Non-Extreme Distribution).

Mancoridis and Mitchell [4], [**?**], [14] developed the Bunch tool for software clustering and used structural dependencies as input. The tool applies clustering algorithms to the structural dependency graph and outputs the system's organization. For evaluation, the authors introduced the Modularization Quality (MQ) metric, described in more detail in Section 1.4.2, and is also used in our current experiments as an evaluation metric.

Prajapati et al. [15] propose a many-objective SBSR (search-based software remodularization) approach with an improved definition of objective functions based on lexical, structural, and change-history dependencies. The authors evaluate their approach on several open-source software systems using the MoJoFM metric for external evaluation and the MQ metric for internal evaluation.

Şora et al. [16], [17] developed the ARTs (Architecture Reconstruction Tool Suite) for their experiments on improving software architecture reconstruction through clustering. The tool suite implements various clustering algorithms, such as minimum spanning tree-based, metric-based, search-based, and hierarchical clustering, primar-

ily using structural dependencies as input. The research focuses on identifying the right factors for direct coupling between classes, indirect coupling, and layered architecture. The results of applying these different factors are evaluated using the MoJo distance metric.

Silva et al. [18] investigated using solely co-change dependencies as input for the Chameleon algorithm, an agglomerative hierarchical clustering method, to identify clusters. For evaluation, the authors used distribution maps to compare the clusters generated from co-change dependencies with the system's package structure.

1.3. Structural and Logical Dependencies

Software clustering relies on various dependencies to identify relationships between software entities. Structural dependencies have been mostly used due to their reliability [16]. However, recent research has started incorporating other types of dependencies besides structural dependencies [9], [11], [15]. This section will present an overview of structural and logical dependencies, focusing on how they are extracted.

1.3.1. Structural Dependencies

Structural dependencies are important for understanding the architecture of a software system because they reveal how different modules interact at the code level. In our research, we extract structural dependencies using a tool from our previous work [19]. This tool analyzes the source code to identify various relationships between software entities and exports them in CSV format.

Structural dependencies do not all have the same level of influence on a software system's architecture and behavior. For instance, the relationship between a variable and the class that uses it is not the same as the relationship between a class and the interface it implements. To reflect these differences, we assign different weights to each type of dependency.

The dependency types and weights were previously defined in related works on clustering [17], [1].

Table 1.1 shows the weights assigned to different categories of structural dependencies, as proposed in previous works.

Weight	Dependency types
4	Interface realization
3	Inheritance, parameter, return type, field, cast, type binding
2	Method call, field access, instantiation
1	Local variable

Table 1.1: Weights assigned to different structural dependency types. [1]

The weights are assigned based on the following considerations:

Weight 4 – Interface Realization: Assigned the highest weight because it signifies a strong architectural relationship. Implementing an interface means classes are expected to provide specific functionalities.

Weight 3 – Inheritance, Parameter, Return Type, Field, Cast, Type Binding: These dependencies represent significant connections between entities. They include inheritance relationships and shared data or types, which affect the behavior and properties of entities.

Weight 2 – Method Call, Field Access, Instantiation: These indicate interactions between classes but are less impactful than higher weights. They involve using methods or fields of other classes or creating instances. When a method call, field access, or instantiation occurs multiple times between the same pair of entities, the weight is multiplied by the number of occurrences. For example, if Class A calls a method in Class B three times, the assigned weight would be 6 (weight 2 multiplied by 3).

Weight 1 – Local Variable: Given the lowest weight, local variables are the most basic level of interaction.

1.3.2. Logical Dependencies

We refer to logical dependencies as the filtered co-changes between software entities. A co-change occurs when two or more software entities are modified together during the same commit in the version control system. Co-changes indicate that these entities are likely directly or indirectly related or dependent on each other.

Co-changes are associated with a degree of uncertainty. Compared to structural dependencies, where a dependency is certain, co-changes are less reliable. For example, if the system was migrated from one version control system to another, the first commit will include all the entities from the system at that point in time. Should we consider all these entities related to one another in this case? This would introduce false dependencies and reduce the likelihood of achieving accurate results when combining them with more reliable types of dependencies.

Even if we address the issue of the first commit, a developer can still resolve multiple unrelated issues in the same commit (even though development processes do not recommend this).

To solve this problem, in our previous works, we refined some filtering methods to ensure that the co-changes that remain after filtering are more reliable and suitable for use with other dependencies or individually [19], [20], [21]. Based on our previous results, the filters we decided to use further in our research are the commit size filter and the strength filter. Both filters are used together, and the result is the set of logical dependencies that we use to generate software clusters.

Commit Size Filter

The commit size filter filters out all co-changes that originate from commits that exceed a certain number of files.

We are interested in extracting dependencies from code commits that involve feature development or bug fixes because that is when developers change related code files. If multiple unrelated features or bug fixes are solved in a single commit, it will appear that all the entities in those files are related, even if they are not.

One scenario where this issue arises is the first commit of a software system when it is ported from one versioning system to another. This commit will contain many changed code files, but these changes do not originate from any functionality change, generating numerous irrelevant co-changes for the system.

A similar scenario occurs with merge commits. A merge commit is automatically created when developers perform a merge operation to integrate changes from one branch into another. After integration, all commits from the branch are added to the target branch, and on top of that, there is the merge commit containing all changes from the commits merged into a single commit. Since this commit contains only a merge of multiple smaller, related issues/features solved, it is better to gather information from the smaller commits rather than from the overall merge commit.

Both scenarios above have in common the large number of files involved in the commits. Based on our previous research and measurements regarding the number of files involved in a commit, we set a threshold of 20 files [19], [20]. Therefore, all co-changes originating from commits with more than 20 changed code files are filtered out.

Table 1.2 presents the commit statistics for the studied projects. The columns represent the percentage of commits with under 5 files modified, between 5 and 10 files, between 10 and 20 files, and above 20 files modified. We can observe that most commits have under 5 files changed, with Apache Tomcat having more than 90% of the commits with less than 5 files changed. On the opposite side, only a few commits involve more than 20 files changed, Hibernate ORM having the highest percentage at 8.39%. Overall, filtering based on commit size does not significantly reduce the number of commits considered.

Table 1.2: Commit statistics for studied projects

Project Name	**Number of files changed**			
	Under 5	**5-10**	**10-20**	**Above 20**
Apache Ant	83.83%	7.50%	4.17%	4.50%
Apache Tomcat	90.95%	5.44%	2.04%	1.58%
Hibernate ORM	71.74%	12.37%	7.50%	8.39%
Gson	83.63%	9.85%	3.70%	2.81%

Strength Filter

This filter focuses on the reliability of the co-changes. If a pair of co-changing entities appears only once in the system's history, it might be less reliable than a pair that

appears more frequently.

Zimmermann et al. introduced the support and confidence metrics to measure the significance of co-changes [22].

The *support metric* of a rule $(A \rightarrow B)$, where A is the antecedent and B is the consequent of the rule, is defined as the number of commits (transactions) in which both entities are changed together.

The *confidence metric* of $(A \rightarrow B)$, as defined in Equation (1.1), focuses on the antecedent of the rule and is the number of commits together of both entities divided by the total number of commits of (A).

$$\text{Confidence}(A \rightarrow B) = \frac{\text{Nr. of commits containing } A \text{ and } B}{\text{Nr. of commits containing } A} \tag{1.1}$$

The confidence metric favors entities that change less and more frequently together rather than entities that change more with a wider variation of other entities.

Assuming that (A) was changed in 10 commits and, of these 10 commits, 9 also included changes to (B), the confidence for the rule $(A \rightarrow B)$ is 0.9. On the other hand, if (C) was changed in 100 commits and, of these 100 commits, 50 also included changes to (D), the confidence for the rule $(C \rightarrow D)$ is 0.5. Therefore, in this scenario, we would have more confidence in the first pair $(A \rightarrow B)$ than in the second pair $(C \rightarrow D)$, even though the second pair has more than five times more updates together.

To favor entities involved in more commits together, we calculated a *system factor*. This system factor is the mean value of the support metric values for all entity pairs.

The system factor is multiplied by the calculated confidence metric value. In addition, since we plan to use the metric values as weights, together with the weights of the structural dependencies, we multiply by 100 to scale the metric value to be supraunitary, and we clip the results between 0 and 100.

We refer to this addition to the original calculation formula as the strength metric, and it is defined in Equation (1.2).

$$\text{strength}(A \rightarrow B) = \text{confidence}(A \rightarrow B) \times 100 \times \text{system factor} \tag{1.2}$$

Filter Application Process

Fig. 1.1 illustrates the overall filter application process. We begin by extracting all co-changes from the versioning system, and the first filter applied is the commit size filter. The commit size filter has a strict threshold of 20 files, meaning that any co-changes from commits involving more than 20 files are filtered out.

The co-changes that remain after applying the commit size filter are then pro-

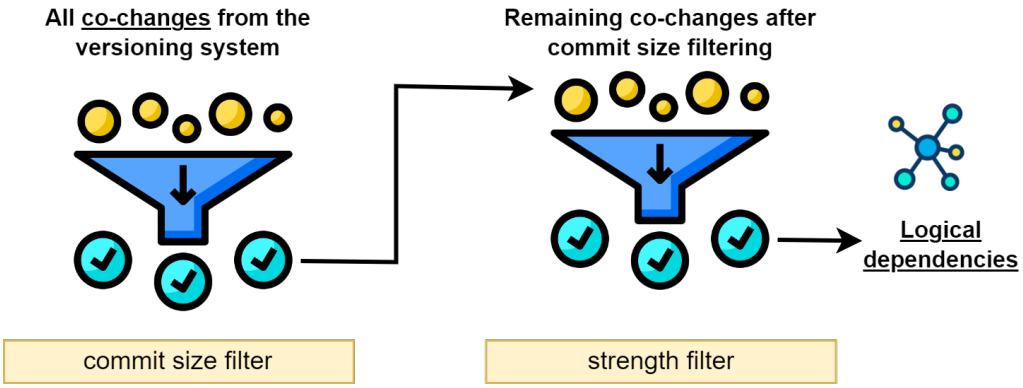

Figure 1.1: Filter application process

cessed using the strength filter. The strength filter uses multiple thresholds, precisely 10 different thresholds. We start with a threshold of 10 and increment it by 10 until we reach a maximum value of 100. We do not use a fixed threshold to assess how different strength thresholds affect our cluster generation.

Dependency Extraction and Filtering Tool

To extract and filter the co-changes, we used a previously developed tool [19]. This tool takes the GitHub repository address and the threshold values for commit and strength filters as input. The tool clones the repository, downloads all commit diffs starting from the first commit, examines all files changed in each commit to identify which entities have changed in those files, and creates undirected co-change dependencies between all changed entities within a commit.

The commit size filter is applied to these undirected co-change dependencies since the metric value for $(A \to B)$ is the same as for $(B \to A)$. For the strength filter, each co-change dependency is converted into a directed co-change dependency, so for each $(A \to B)$ dependency, we have both $(A \to B)$ and $(B \to A)$. This conversion is necessary because, as mentioned in the previous section, the confidence filter evaluates the rule's antecedent. Thus, the metric value for $(A \to B)$ differs from the metric value for $(B \to A)$.

After applying the filters, the remaining dependencies are exported to a CSV file for further use.

It is important to note that the strength metric is only used for filtering and is *not considered as a weight* of the dependencies. The *weight assigned to each dependency is the number of commits in which both entities were updated together*.

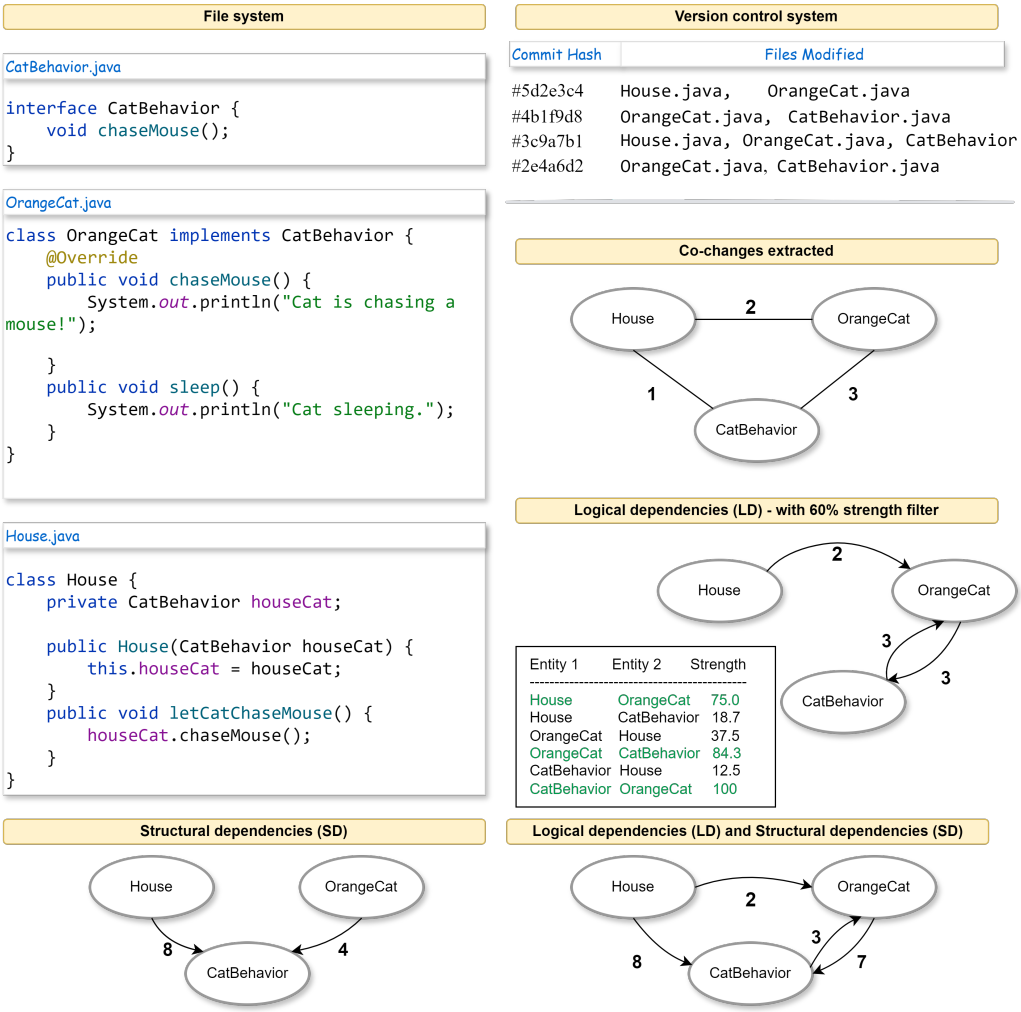

Figure 1.2: Dependency Graph: Combining structural and logical dependencies.

Combining Structural and Logical Dependencies

When structural dependencies (SD) and logical dependencies (LD) are combined in software clustering, both types of relationships are represented within the same graph.

Each entity in the system is represented as a node in the graph, and the dependencies between them are represented as directed weighted edges.

SD and LD weights are combined when the same pair of entities appear in both dependencies. In this case, the weights from SD and LD are summed, giving more influence to those entity pairs. When a pair of entities appear only in SD or only in LD, the edge is added to the graph together with its corresponding weight.

Figure 1.2 illustrates combining structural and logical dependencies in the same dependency graph. The structural dependencies between `House`, `OrangeCat`, and `CatBehavior` entities are visible from the source code analysis.

However, the combination of SD and LD reveals additional insights. One important observation is the logical dependency between `House` and `OrangeCat`, which is not observed from the structural analysis. This relation is extracted from version control and filtered using a 60% strength filter. The strength metric reveals that `House` and `OrangeCat` have a significant co-change value of 75.0, usually associated with a strong relationship.

When SD and LD overlap, such as between `OrangeCat` and `CatBehavior`, their weights are summed. This summation increases the weight of the dependency, making it more important in the dependency graph.

1.4. Methodology and Implementation

In this section, we present the methodology used to evaluate the impact of logical dependencies on the quality of software clustering solutions.

First, we describe the clustering algorithms used in our experiments: Louvain, Leiden, and DBSCAN. Next, we introduce the evaluation metrics used to assess the quality of the clustering results. Finally, we present the workflow and implementation of the tool developed for this research, which is built to process structural and logical dependencies, apply the selected clustering algorithms, and compute the evaluation metrics.

1.4.1. Clustering Algorithms

Louvain

The Louvain algorithm was originally developed by Blondel et al. and is used to find community partitions (clusters) in large networks. The algorithm begins with a weighted network of N nodes, initially assigning each node to its own cluster, resulting in N clusters. For each node, the algorithm evaluates the modularity gained from moving the node to the cluster of each of its neighbors. Based on the results, the node is moved to the cluster with the maximum positive modularity gain. This process is repeated for all nodes until no further improvement in modularity is possible [23], [24].

Leiden

The Leiden algorithm, developed by Traag et al., is an improvement over the Louvain algorithm for community detection in large networks. Like Louvain, the Leiden algorithm begins with each node assigned to its own cluster and iteratively moves nodes between clusters to optimize modularity. However, the Leiden algorithm addresses

some problems of the Louvain method, particularly regarding poorly connected communities and runtime performance issues [25] [26].

The Leiden algorithm introduces a refinement phase that ensures communities are locally optimally clustered and well-connected. This refinement step distinguishes the Leiden algorithm from Louvain.

DBSCAN

The Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm, introduced by Ester et al., is a density-based clustering algorithm for identifying clusters of arbitrary shape and detecting noise in data [27], [26].

DBSCAN operates based on two main parameters:

- **Eps**: It defines the radius within which to search for neighboring points.
- **MinPts**: The minimum number of points required for a dense region. It determines the minimum number of neighbors a point should have to be considered a core point.

The algorithm classifies points into three categories:

1. **Core Points**: Points that have at least *MinPts* neighbors within a radius of *Eps*. These points are located in the interior of a cluster.

2. **Border Points**: Points that have fewer than *MinPts* neighbors within a radius of *Eps* but are in the Eps-neighborhood of a core point. They are located on the edge of a cluster.

3. **Noise**: Points that are neither core points nor border points.

The DBSCAN algorithm starts by visiting an arbitrary point in the dataset. If the point is a core point, the algorithm starts a new cluster and retrieves all reachable points from this core point. All points are then marked as part of the cluster. If the point is a border point, it moves to the next point in the dataset. This process is repeated until all points have been visited.

DBSCAN can be applied for software clustering by considering software entities as data points. A distance measure based on dependency weights can be used to compute the neighborhood between entities.

1.4.2. Clustering Result Evaluation

We evaluate the clustering results using two metrics: the Modularity Quality (MQ) metric and the Move and Join Effectiveness Measure (MoJoFM) metric. Each provides a different perspective on the quality of the clustering solutions.

Modularity Quality Metric

Mancoridis et al. introduced the Modularity Quality (MQ) metric to evaluate the modularization quality of a clustering solution based on the interaction between modules (clusters) [**?**],[4]. It evaluates the difference between connections within clusters and connections between different clusters.

The MQ of a graph partitioned into k clusters, where A_i is the Intra-Connectivity of the i-th cluster and E_{ij} is the Inter-Connectivity between the i-th and j-th clusters, is calculated using Equation (1.3) [28].

$$MQ = \left(\frac{1}{k}\sum_{i=1}^{k} A_i\right) - \left(\frac{1}{k(k-1)}\sum_{i,j=1}^{k} E_{ij}\right) \tag{1.3}$$

The MQ metric's value ranges between -1 and 1. A value of -1 means that the clusters have more connections between the clusters than within the clusters, while a value of 1 means that there are more connections within clusters than between clusters. A good clustering solution should have an MQ value close to 1, since this indicates that the clusters are more cohesive internally and have fewer connections to other clusters.

The MQ metric is useful because it does not require additional input besides the clustering result. It relies on the structure of the clustered entities and their interactions.

MoJoFM Metric

Wen and Tzerpos introduced the MoJoFM metric to evaluate the similarity between two different software clustering results [5]. The metric is based on the MoJo metric, which measures the absolute minimum number of *Move* and *Join* operations required to transform one clustering solution into another [7], [5]. However, MoJoFM provides a similarity measure ranging between 0% and 100%, where 100% indicates identical clustering solutions.

The MoJoFM metric is calculated using Equation (1.4):

$$\text{MoJoFM}(A,B) = \left(1 - \frac{\text{mno}(A,B)}{\text{max}(\text{mno}(\forall A,B))}\right) \times 100\% \tag{1.4}$$

Where:

- mno(A,B) is the minimum number of *Move* and *Join* operations required to transform clustering solution A into clustering solution B.
- max(mno$(\forall A,B)$) is the maximum possible number of such operations required to transform any clustering A into clustering B.

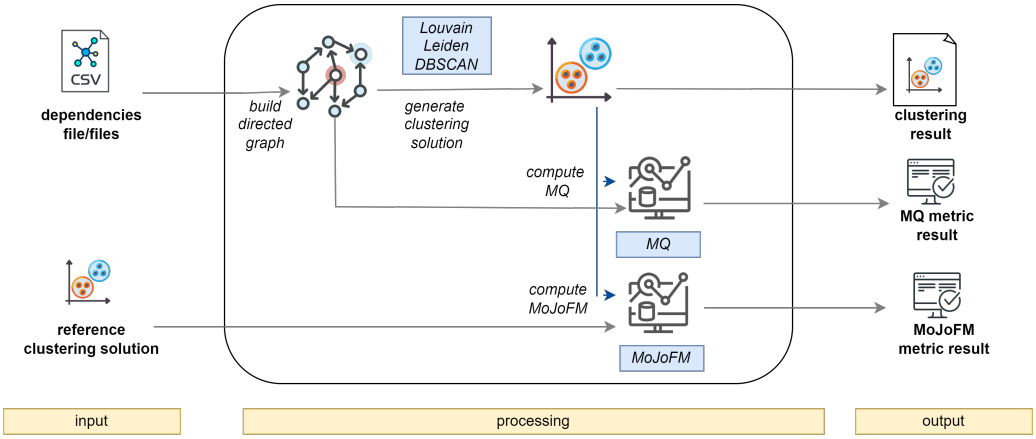

Figure 1.3: Tool workflow overview: input, processing and output.

To use the metric, we first need to generate a reference clustering solution for comparison. We manually created this reference based on our analysis of the codebase.

Using the MoJoFM metric, we can evaluate the similarity between the generated and reference clustering solutions. This metric is useful when combining multiple dependencies because it measures the similarity between the obtained clustering solutions and the same reference.

1.4.3. Workflow for Software Clustering and Evaluation

To evaluate how logical dependencies impact the quality of clustering solutions, we developed a Python tool capable of using any type of dependency, either alone or combined with other types of dependencies, as long as it is provided in CSV format. The tool clusters and evaluates software clustering solutions using the MQ or MoJoFM metrics.

Input

The tool takes one or multiple dependency CSV files as input and the reference solution required for the MoJoFM metric. We designed the tool to accept multiple dependency files so that we can generate clustering solutions based on either a single type of dependency (structural or logical) or a combination of both.

Since the MoJoFM metric requires a reference solution to evaluate the obtained clustering solutions, we manually inspected the code and created reference clustering solutions, which we then provided as input for the tool.

Processing

The dependencies are saved in the CSV file in the following format: `antecedent of a dependency, consequent of a dependency, weight`. The tool reads each line, adds the antecedent and consequent as nodes in a directed graph, and creates an edge between them, with the weight from the CSV file becoming the edge weight. The edge weights are summed if multiple dependency files are processed and the same dependency is found in multiple files.

The workflow of applying the clustering algorithms and performing the evaluations is shown in Figure 1.3. After all dependencies are read, the directed graph is passed to the clustering algorithms: Louvain, Leiden, and DBSCAN. Each algorithm generates its own clustering result. The results from each algorithm are then evaluated using the MQ metric and the MoJoFM metric. The MQ metric requires the directed graph and the clustering result, while the MoJoFM metric requires the reference clustering solution provided as input and the clustering result.

Output

After applying each clustering algorithm and completing both evaluations, we export the clustering result, the number of clusters from the clustering solution, and the MQ and MoJoFM metrics values.

1.5. Experimental Plan and Results

1.5.1. Experimental Plan

Overview of projects used

Table 1.3: Overview of projects used in experimental analysis

Project Name	Release Tag	Commits	GitHub Repository Link	Repository Description
Apache Ant	1.10.13	14,917	https://github.com/apache/ant	Java build tool for automating software tasks.
Apache Tomcat	8.5.93	22,698	https://github.com/apache/tomcat	Java web server and servlet container.
Hibernate ORM	6.2.14	16,609	https://github.com/hibernate/hibernate-orm	Java ORM framework for database management.
Gson	gson-parent-2.10.1	1,772	https://github.com/google/gson	Java library for JSON serialization and deserialization.

In Table 1.3, we have synthesized all the information about the four projects used in our experiments. The 'Project Name' column contains the names of the software projects sourced from GitHub. The 'Release Tag' column contains the specific

release tag of the project that was analyzed. We processed all the commits for logical dependency extraction, from the first commit to the commit associated with the specified tag. We extracted the dependencies from the code of that specific tag for structural dependencies. The 'Number of Commits' column provides the total number of commits used for logical dependencies extraction. The 'GitHub Repository Link' column includes the URL link to the project's repository on GitHub. Finally, the 'Repository Description' column briefly describes the project's purpose and functionality.

We mainly chose projects with more than 10,000 commits in their commit history so that the logical dependencies extraction can be done on a more extensive information base. However, we selected Gson, which has a relatively small commit history (1,772 commits), to determine if our experiments work with a smaller information base.

Tool runs

To assess the impact of logical dependencies and to answer the research questions from section 1.1, we run the tool presented in Section 1.4.3 in three different scenarios for all the projects from table 1.3. All three scenarios are illustrated in Fig. 1.4.

In the first scenario, we run the tool once, providing only the system's structural dependencies as input for the clustering algorithm.

In the second scenario, we run the tool ten times, using only logical dependencies as input. We perform ten runs because we generate logical dependencies with different threshold values for the strength filter. We start with a threshold of 10 and increase it in steps of 10 up to 100, where 100 is the maximum value for the threshold.

In the third scenario, we combine logical with structural dependencies. Similar to the second scenario, we ran the tool ten times using structural and logical dependencies generated with different strength thresholds.

1.5.2. Results

The experimental results are presented in this subsection in four tables, each corresponding to a different project. Table 1.4 presents the results for Apache Ant, Table 1.5 presents the results for Apache Tomcat, Table 1.6 presents the results for Hibernate ORM, and Table 1.7 presents the results for Gson.

Each table includes the following columns:

- **Dependency Type:** The types of dependencies used are as follows: SD for Structural Dependencies, LD for Logical Dependencies, and SD+LD for their combination. The strength threshold used is specified in parentheses right after LD.
- **Entities Count:** The total number of software entities (such as classes, interfaces, enums) involved in clustering.
- **System Coverage:** Considering that the total number of entities extracted from the codebase — which represents the entities forming structural depen-

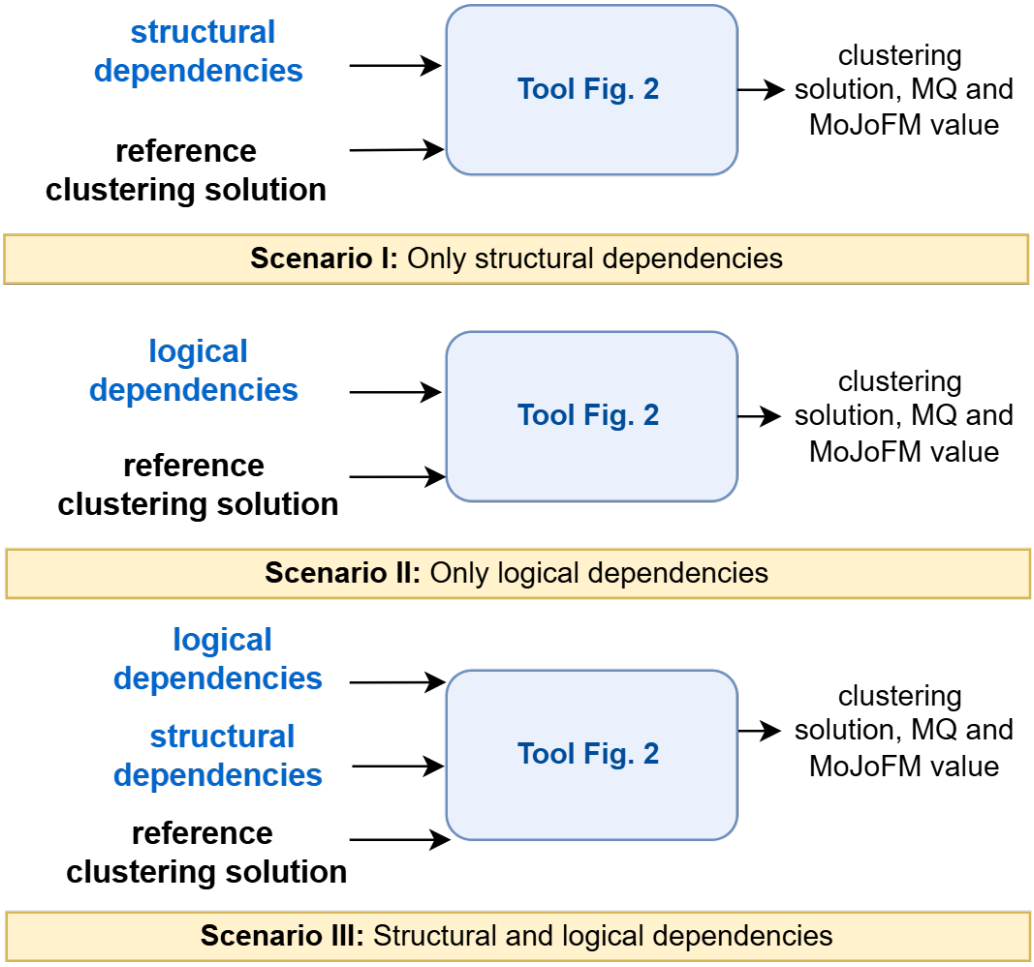

Figure 1.4: Experimental scenarios for analyzing the impact of logical dependencies on clustering quality

dencies (SD) — constitutes the entire set of entities in the system (the first line of each table), we calculated the percentage of entities present in the filtered logical dependencies (LD) relative to the total number of known codebase entities.

- **Louvain/ Leiden/ DBSCAN:** The clustering algorithms used in the experiments.
- **Nr. of Clusters:** The number of clusters from the clustering solution.
- **MQ (Modularization Quality):** The result obtained when applying the MQ evaluation metric to the clustering solution.
- **MoJoFM:** The result obtained when applying the MoJoFM evaluation metric to the clustering solution.

The rows in each table represent different dependency types and strength filter

thresholds used in the clustering experiments.

Table 1.4: Clustering results based on different dependency types and strength filter thresholds for repository: https://github.com/apache/ant

Dependency Type (strength threshold)	Entities Count	System Coverage (%)	Louvain			Leiden			DBSCAN		
			Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM
SD	517	100.00	14	0.114	46.02	14	0.101	52.99	34	0.144	25.1
LD (10)	320	61.89	55	0.506	65.57	55	0.506	65.57	30	0.435	39.02
LD (20)	215	41.58	53	0.547	68	53	0.547	68	23	0.505	53.5
LD (30)	174	33.65	44	0.558	71.7	44	0.558	71.7	19	0.585	50
LD (40)	152	29.40	40	0.580	71.53	40	0.580	71.53	19	0.602	53.06
LD (50)	138	26.69	35	0.604	73.98	35	0.604	73.98	17	0.633	56.1
LD (60)	120	23.21	34	0.587	70.48	34	0.587	70.48	14	0.650	51.43
LD (70)	106	20.50	32	0.577	71.43	32	0.577	71.43	11	0.661	51.65
LD (80)	92	17.79	29	0.576	70.13	29	0.576	70.13	9	0.709	50.65
LD (90)	79	15.28	24	0.606	71.88	24	0.606	71.88	8	0.705	56.6
LD (100)	64	12.37	19	0.611	75.51	19	0.611	75.51	6	0.691	56.93
SD+LD (10)	517	100.00	18	0.355	55.18	15	0.254	54.98	37	0.147	25.9
SD+LD (20)	517	100.00	17	0.318	52.39	19	0.365	53.78	32	0.149	26.49
SD+LD (30)	517	100.00	16	0.282	53.19	16	0.265	54.78	30	0.159	24.5
SD+LD (40)	517	100.00	17	0.340	51.99	17	0.317	53.19	31	0.146	24.7
SD+LD (50)	517	100.00	15	0.248	52.59	19	0.298	56.77	31	0.146	24.7
SD+LD (60)	517	100.00	16	0.244	50.8	16	0.271	54.38	32	0.155	25.1
SD+LD (70)	517	100.00	15	0.238	51.00	18	0.281	52.99	32	0.155	25.1
SD+LD (80)	517	100.00	13	0.246	45.22	15	0.255	45.82	32	0.155	25.1
SD+LD (90)	517	100.00	14	0.258	46.02	16	0.268	47.01	32	0.155	25.1
SD+LD (100)	517	100.00	15	0.214	50.8	15	0.227	50.4	32	0.155	25.1

Table 1.5: Clustering results based on different dependency types and strength filter thresholds for repository: https://github.com/apache/tomcat

Dependency Type (strength threshold)	Entities Count	System Coverage (%)	Louvain			Leiden			DBSCAN		
			Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM
SD	662	100.00	26	0.186	77.76	24	0.184	76.99	43	0.142	73.31
LD (10)	406	61.33	42	0.505	72.47	42	0.505	72.47	60	0.393	67.93
LD (20)	303	45.77	45	0.538	68.26	45	0.538	67.24	41	0.510	72.7
LD (30)	249	37.61	46	0.532	69.87	46	0.532	69.87	32	0.561	80.33
LD (40)	208	31.42	42	0.590	69.70	42	0.591	70.71	28	0.572	83.84
LD (50)	198	29.91	44	0.604	70.21	44	0.604	70.21	22	0.631	85.11
LD (60)	177	26.74	45	0.601	70.66	45	0.601	70.66	18	0.662	85.63
LD (70)	164	24.77	45	0.598	75.32	45	0.598	75.32	17	0.676	88.96
LD (80)	127	19.18	36	0.618	79.49	36	0.618	79.49	15	0.713	89.74
LD (90)	116	17.52	32	0.623	81.13	32	0.623	81.13	14	0.718	89.62
LD (100)	110	16.62	30	0.640	85.00	30	0.640	85.00	13	0.735	89.00
SD+LD(10)	662	100.00	28	0.324	78.99	28	0.324	78.99	40	0.161	74.23
SD+LD(20)	662	100.00	31	0.287	78.22	30	0.320	80.06	50	0.189	73.31
SD+LD(30)	662	100.00	32	0.296	79.92	32	0.277	75.77	45	0.209	73.47
SD+LD(40)	662	100.00	34	0.292	79.91	32	0.326	78.22	43	0.198	73.47
SD+LD(50)	662	100.00	33	0.294	76.53	35	0.301	76.23	43	0.196	73.31
SD+LD(60)	662	100.00	35	0.304	77.15	33	0.286	76.84	41	0.177	73.62
SD+LD(70)	662	100.00	34	0.282	76.69	34	0.292	77.45	41	0.166	73.62
SD+LD(80)	662	100.00	34	0.283	76.23	33	0.282	76.38	42	0.153	73.47
SD+LD(90)	662	100.00	31	0.311	78.99	31	0.311	78.99	43	0.153	73.31
SD+LD(100)	662	100.00	31	0.311	78.83	31	0.305	78.37	43	0.153	73.31

To better understand the impact of different dependency types on software clustering, we also analyzed the average weights assigned to structural dependencies (SD) and logical dependencies (LD) across the studied projects. Table 1.8 presents these average dependency weights. The first row shows the average weights for SD, which remain constant across all strength thresholds, while the other rows show the average weights for logical dependencies at different strength thresholds.

1.6. Evaluation

The overall analysis of all the results from subsection 1.5.2 indicates that combining structural and logical dependencies (SD+LD) provides better clustering solutions than using structural dependencies (SD) alone, covering 100% of the system, meaning that no entity is missed during cluster generation. On the other hand, logical dependencies

Table 1.6: Clustering results based on different dependency types and strength filter thresholds for repository: https://github.com/hibernate/hibernate-orm

https://github.com/hibernate/hibernate-orm

Dependency Type (strength threshold)	Entities Count	System Coverage (%)	Louvain			Leiden			DBSCAN		
			Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM
SD	4414	100.00	30	0.09	52.23	23	0.071	52.44	373	0.128	46.32
LD (10)	1450	32.85	44	0.389	57.22	45	0.39	58.22	99	0.395	57.08
LD (20)	1325	30.02	66	0.397	62.66	66	0.397	62.66	151	0.378	63.36
LD (30)	1222	27.68	66	0.38	62.45	67	0.38	63.04	148	0.378	65.42
LD (40)	915	20.73	84	0.417	63.68	85	0.412	63.56	110	0.382	66.9
LD (50)	900	20.39	84	0.409	64.56	84	0.409	64.56	105	0.386	67.02
LD (60)	848	19.21	82	0.406	63.26	81	0.41	63.39	104	0.379	65.13
LD (70)	459	10.40	89	0.516	69.08	89	0.516	69.08	41	0.467	58.21
LD (80)	450	10.19	91	0.506	68.64	91	0.506	68.64	39	0.479	60.49
LD (90)	432	9.79	92	0.492	66.93	92	0.492	66.93	40	0.473	58.66
LD (100)	356	8.07	81	0.524	65.92	81	0.524	65.92	29	0.537	58.2
SD+LD (10)	4414	100.00	19	0.096	53.93	19	0.099	52.28	282	0.121	46.01
SD+LD (20)	4414	100.00	21	0.126	52.85	23	0.122	56.21	309	0.135	47.4
SD+LD (30)	4414	100.00	26	0.121	55.76	26	0.15	54.54	317	0.135	49.45
SD+LD (40)	4414	100.00	27	0.182	54.57	28	0.163	55.89	350	0.134	49.35
SD+LD (50)	4414	100.00	26	0.16	52.37	24	0.147	53.31	350	0.134	49.37
SD+LD (60)	4414	100.00	26	0.161	52.35	27	0.153	53.19	352	0.135	49.31
SD+LD (70)	4414	100.00	28	0.139	52.78	29	0.154	54.34	366	0.13	47.13
SD+LD (80)	4414	100.00	28	0.142	52.83	28	0.147	53.35	366	0.13	47.72
SD+LD (90)	4414	100.00	28	0.136	52.62	30	0.153	53.83	365	0.13	47.72
SD+LD (100)	4414	100.00	30	0.128	52.78	28	0.114	55.23	365	0.128	47.75

Table 1.7: Clustering results based on different dependency types and strength filter thresholds for repository: https://github.com/google/gson

https://github.com/google/gson

Dependency Type (strength threshold)	Entities Count	System Cover (%)	Louvain			Leiden			DBSCAN		
			Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM
gson SD	210	100.00	10	0.139	53.47	9	0.129	55.94	23	0.127	51.88
gson LD (10)	66	31.43	10	0.565	62.07	9	0.572	60.34	19	0.399	68.97
gson LD (20)	50	23.81	11	0.547	64.29	11	0.547	64.29	9	0.523	59.52
gson LD (30)	41	19.52	12	0.544	63.64	12	0.544	63.64	6	0.606	66.67
gson LD (40)	31	14.76	8	0.635	69.57	8	0.635	69.57	6	0.612	69.57
gson LD (50)	31	14.76	8	0.600	69.57	8	0.600	69.57	6	0.565	60.87
gson LD (60)	28	13.33	8	0.552	65.00	8	0.552	65.00	5	0.584	60.00
gson LD (70)	26	12.38	7	0.579	66.67	7	0.579	66.67	5	0.586	55.56
gson LD (80)	18	8.57	5	0.590	60.00	5	0.590	60.00	4	0.544	40.00
gson LD (90)	18	8.57	5	0.590	60.00	5	0.590	60.00	4	0.544	40.00
gson LD (100)	18	8.57	5	0.590	60.00	5	0.590	60.00	4	0.544	40.00
gson SD+LD(10)	210	100.00	11	0.317	64.36	11	0.317	64.36	20	0.172	63.86
gson SD+LD(20)	210	100.00	11	0.259	61.39	11	0.259	61.39	17	0.136	53.96
gson SD+LD(30)	210	100.00	11	0.277	61.39	11	0.277	61.39	20	0.136	55.94
gson SD+LD(40)	210	100.00	10	0.277	61.39	10	0.277	61.39	20	0.135	55.94
gson SD+LD(50)	210	100.00	10	0.270	60.40	11	0.270	60.89	20	0.135	55.94
gson SD+LD(60)	210	100.00	9	0.296	61.39	10	0.290	61.88	20	0.135	55.94
gson SD+LD(70)	210	100.00	8	0.295	59.41	8	0.295	59.41	20	0.135	55.94
gson SD+LD(80)	210	100.00	7	0.267	58.91	8	0.263	59.41	21	0.134	55.45
gson SD+LD(90)	210	100.00	7	0.267	58.91	7	0.267	58.91	21	0.134	55.45
gson SD+LD(100)	210	100.00	7	0.267	58.91	8	0.263	59.41	21	0.134	55.45

Dependency type	Ant	Tomcat	Hibernate	Gson
SD	5.91	6.91	5.41	5.24
LD(10)	11.17	12.82	2.45	14.15
LD(20)	16.01	19.65	3.00	19.10
LD(30)	18.08	23.56	3.27	27.58
LD(40)	19.08	25.57	4.63	29.85
LD(50)	19.94	26.31	4.80	29.97
LD(60)	24.26	28.91	5.14	33.93
LD(70)	26.70	30.35	9.53	34.37
LD(80)	30.83	35.33	10.18	43.00
LD(90)	32.11	36.90	10.47	43.00
LD(100)	33.93	37.04	12.00	43.00

Table 1.8: Average weights of Structural Dependencies (SD) and Logical Dependencies (LD).

(LD) alone result in better clustering quality metrics compared to both SD and SD+LD, but they do not cover the entire system.

The best results for SD+LD are observed with a strength threshold between 10-40%. For LD only, the best results are obtained at a 100% strength threshold. The overall trend shows that for LD only, the MQ metric increases in value with a higher strength threshold, indicating more cohesive clusters, while the MoJo metric decreases, indicating that fewer transformations are needed to reach the expected clustering. For SD+LD, the best MQ and MoJo values are obtained at lower strength thresholds, and then both metrics indicate a less effective clustering solution obtained with higher strength thresholds.

We analyze each project in detail in the sections below and address the research questions.

1.6.1. Detailed results

Apache Ant

The clustering results for Apache Ant (Table 1.4) show that the combined structural and logical dependencies (SD+LD) achieved the best values with a strength threshold between 10% and 30%. The highest value for the MQ metric is reached with Leiden at a strength threshold of 20%, and the highest value for MoJoFM is also reached with Leiden at a strength threshold of 10%.

Compared with the SD-only results, all SD+LD clustering solutions for all algorithms show better MQ metric values, with the highest MQ value for SD+LD being more than three times greater than the corresponding SD-only value. Similarly, the MoJoFM metric shows better results than SD-only. However, it does not always outperform the MoJoFM metric applied to SD-only data.

Logical dependencies (LD) alone produced the highest MQ and MoJoFM values at the 100% strength threshold for both Leiden and Louvain, with the obtained metric values being higher than those of SD-only and SD+LD. However, the percentage of entities covered is significantly lower (LD(100) covers only 12.37% of the system). If we look at LD(10), where there is a 61.89% coverage of the system, which is more compared to LD(100), both metrics still perform better than SD-only and SD+LD(10). However, there is still a gap until 100% coverage.

From the clustering algorithm performance point of view, Leiden obtains the best evaluation metrics for all scenarios, followed by Louvain and DBSCAN.

One interesting observation is that, based on the LD-only results, where the metrics results improved with higher strength thresholds, the SD+LD results did not follow the same pattern. On the opposite, the SD+LD metric results decline with a higher strength threshold. An explanation for this behavior may lie in the overlap between structural and logical dependencies. As presented in Figure 1.5, the number of LD decreases with a stricter strength threshold compared to the number of SD, and the overlap between the two types of dependencies increases.

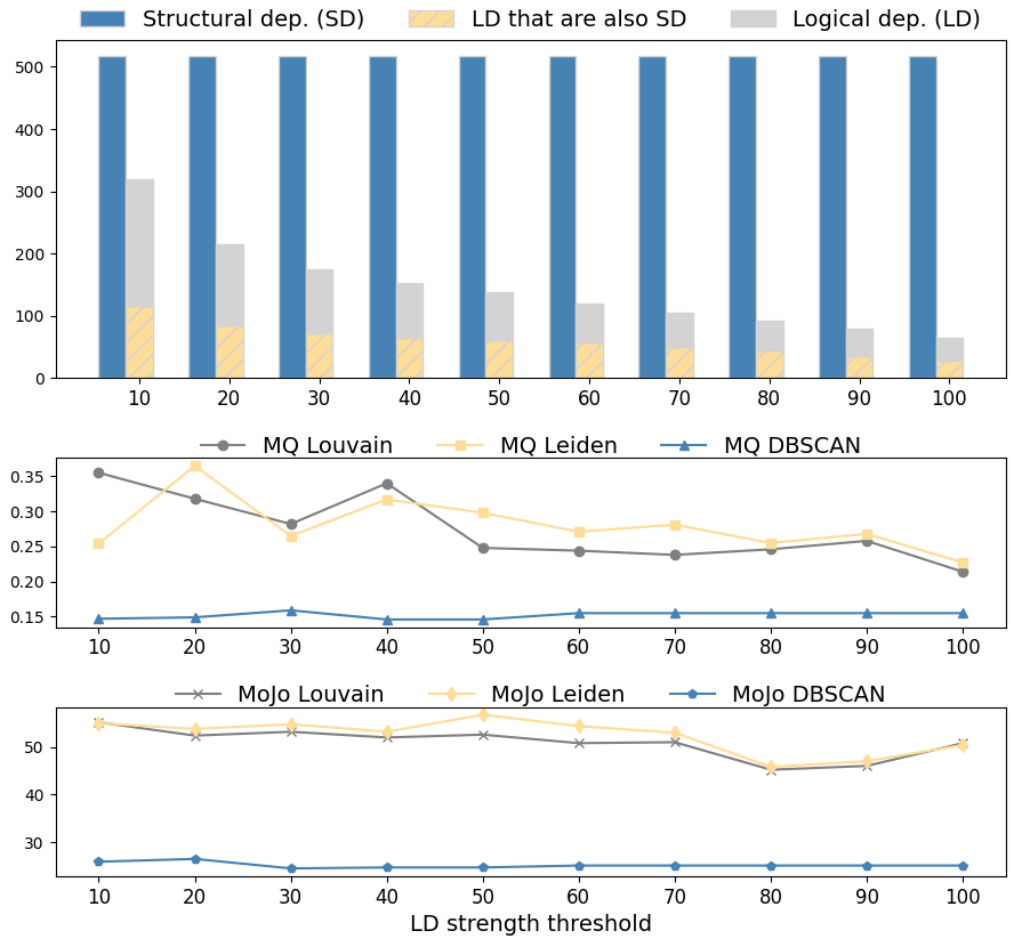

Figure 1.5: Apache Ant: Overlap between structural and logical dependencies and its correlation with clustering metrics.

In our previous works, we studied how these two types of dependencies overlap [19], [20]. The reason behind those studies was to check how much new information we can get from using logical dependencies and how much is already present via structural dependencies.

Our overall findings were that with stricter filtering of logical dependencies, we obtain a higher percentage of overlap between the two dependencies, reaching at most 50% of logical dependencies that are also structural dependencies.

So, we consider that the reason why SD+LD clustering solutions decline in performance with a higher strength threshold is that less and less new additional information is added to the system (logical dependencies that are not structural dependencies), causing the clustering solution to start resembling the performance of

the SD-only solution. In Figure 1.5, we can see that LD(10) represents 61% of the quantity of SD, while LD(100) is only at 12%, with half of them being duplicated with SD.

Apache Tomcat

For Apache Tomcat (Table 1.5), the best results for SD+LD were obtained with strength thresholds between 10% and 40% across all algorithms. The Leiden algorithm achieved the best result for the MQ metric at a strength threshold of 40%, while the best MoJoFM result was obtained at a threshold of 20%, also with the Leiden algorithm. Compared with the SD-only results, the peak MQ values almost double the SD-only values. Like Apache Ant, the MQ values for all strength thresholds are higher than those for SD-only. While MoJoFM is not better for all thresholds, it still improves compared with the SD-only results.

The LD-only results show the highest MQ and MoJoFM values at LD(100) for the Louvain and Leiden algorithms. However, as with the Apache Ant results, coverage remains an issue. LD(100) covers only 16.62% of the system, lower than the coverage from SD-only or SD+LD combinations. On the other hand, LD(10), which covers 61.33% of the system, still has better clustering solutions compared to SD-only, based on both MQ and MoJoFM results.

We observe the same decline in results with a stricter strength threshold for SD+LD. As with the previous system, these results can again be connected to the percentage of LD that also overlaps with SD and the decreasing number of LD compared to SD once the strength threshold becomes stricter. As shown in Figure 1.6, LD filtered with a 10% strength threshold overlaps with SD by approximately 22%, while at a 100% strength threshold, the overlap increases to approximately 39%.

To ensure that the decline in performance for SD+LD with a stricter strength threshold is indeed caused by the fact that LD are significantly fewer than SD, and SD duplicates that part of them at higher thresholds, we added an additional experiment to our study. In this experiment, whose results can be found in Table 1.9, we increased the weights associated with LD(100) for Apache Tomcat to confirm that we are dealing with an LD quantity problem rather than a weight problem.

Therefore, in this experiment, we increased the weight assigned to each logical dependency filtered with a 100% strength threshold from the Apache Tomcat system by values ranging from 1 to 5 and re-ran scenario III from Fig. 1.4.

To maintain consistency, we used the same columns as in the other result tables (1.4, 1.5, 1.6, 1.7), with the addition of two new columns:

- **Multiplication Factor:** The value by which each logical dependency weight is multiplied.
- **Avg Weight:** The average weight assigned to each type of dependency used.

In Table 1.8, which presents the average weights associated with the dependencies across all systems, we can see that for Tomcat, the average weight for LD(10) is already almost double the SD average weight. For LD(100), the average weight is

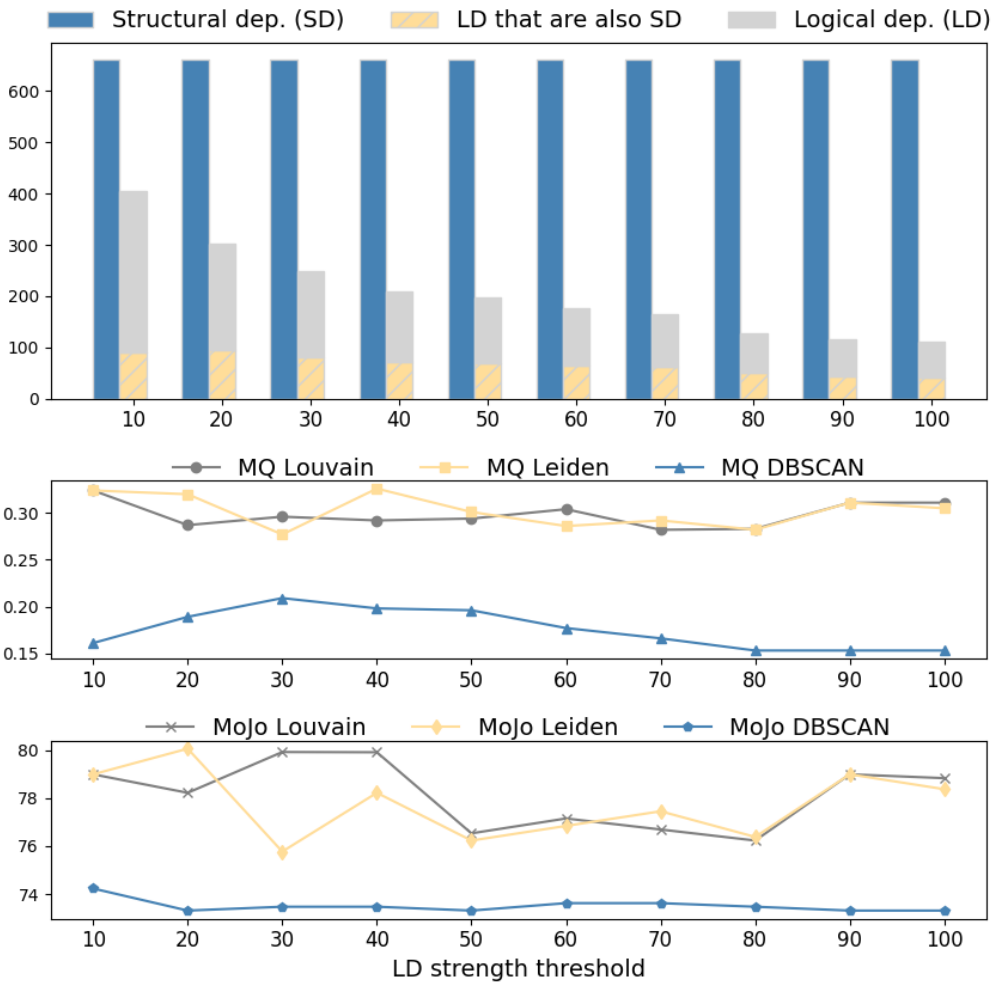

Figure 1.6: Apache Tomcat: Overlap between structural and logical dependencies and its correlation with clustering metrics.

approximately five times higher than that of SD.

Based on the metric values obtained for multiplication factors of 2 to 5, we can see that after increasing the weights assigned to LD, the metric values improve only slightly, with changes recorded at the second decimal: a 0.02 improvement for Louvain and 0.07 for Leiden. The results for DBSCAN remain unchanged due to the fixed values of MinPts and Eps and the already high LD weights for LD(100).

We can conclude from the experiment with weights that the issue is the quantity of dependencies. SD outnumbers LD, making LD information less impactful on the clustering solution.

Table 1.9: Impact of multiplication factors on clustering results for LD(100) in Apache Tomcat

Multiplication Factor	Avg. weight		Louvain			Leiden			DBSCAN		
	SD	LD	Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM	Nr. of clusters	MQ	MojoFM
1	6.91	37.04	31	0.311	78.83	31	0.305	78.37	43	0.153	73.31
2	6.91	74.08	33	0.295	73.57	30	0.301	72.33	43	0.153	73.31
3	6.91	111.12	34	0.313	74.19	33	0.309	72.80	43	0.153	73.31
4	6.91	148.16	34	0.312	73.88	33	0.312	72.49	43	0.153	73.31
5	6.91	185.20	34	0.306	73.88	33	0.308	72.18	43	0.153	73.31

Hibernate ORM

Hibernate ORM is the second largest system after Apache Tomcat regarding the number of commits analyzed, with 16,609 commits considered for LD extraction. Additionally, it is the largest in terms of system size, with 4,414 entities (Table 1.3).

Based on the results from Table 1.6, the SD+LD combination with a strength threshold of 40 performs best for this system. Louvain achieves the best MQ metric at this threshold, while Leiden achieves the best MoJoFM metric.

LD-only produced the best MQ values at 100% strength and the best MoJoFM values at 70% strength for both Louvain and Leiden. Compared to the previous systems, where both best values were recorded at the same strength threshold, Hibernate shows an earlier peak for MoJoFM. The system coverage is likely a factor contributing to this. Hibernate LD[100] covers only 8.07% of the system, the lowest percentage among all systems studied. This low percentage can be linked to the number of commits compared to the number of entities. For Apache Tomcat, there were 22,698 commits and 662 entities, while for Hibernate, there were 16,609 commits and 4,414 entities. This indicates that not all entities had a chance to be updated in the version control system.

This observation is also reflected in Table 1.8, where Hibernate has the lowest average weights for LD compared to SD across all systems. In other systems, LD(10) starts with almost double the average weight compared to SD, while Hibernate's LD(10) average weight is less than half of the SD average weight.

Hibernate has the lowest overlap percentage between LD and SD, as shown in Figure 1.7. Similar to the other systems, the performance for MQ and MoJoFM decreases for SD+LD as the strength threshold becomes stricter.

The results for Hibernate highlight the challenge of achieving better clustering in larger systems with fewer commits relative to their size.

Google Gson

Gson has the smallest number of commits analyzed, with 1,772 commits considered for LD extraction, and it is also the smallest in terms of system size, with 210 entities involved in clustering (Table 1.3).

Based on the results from Table 1.7, the SD+LD combination with a strength threshold of 10 achieved the best results for both MQ and MoJoFM. Like Apache Ant

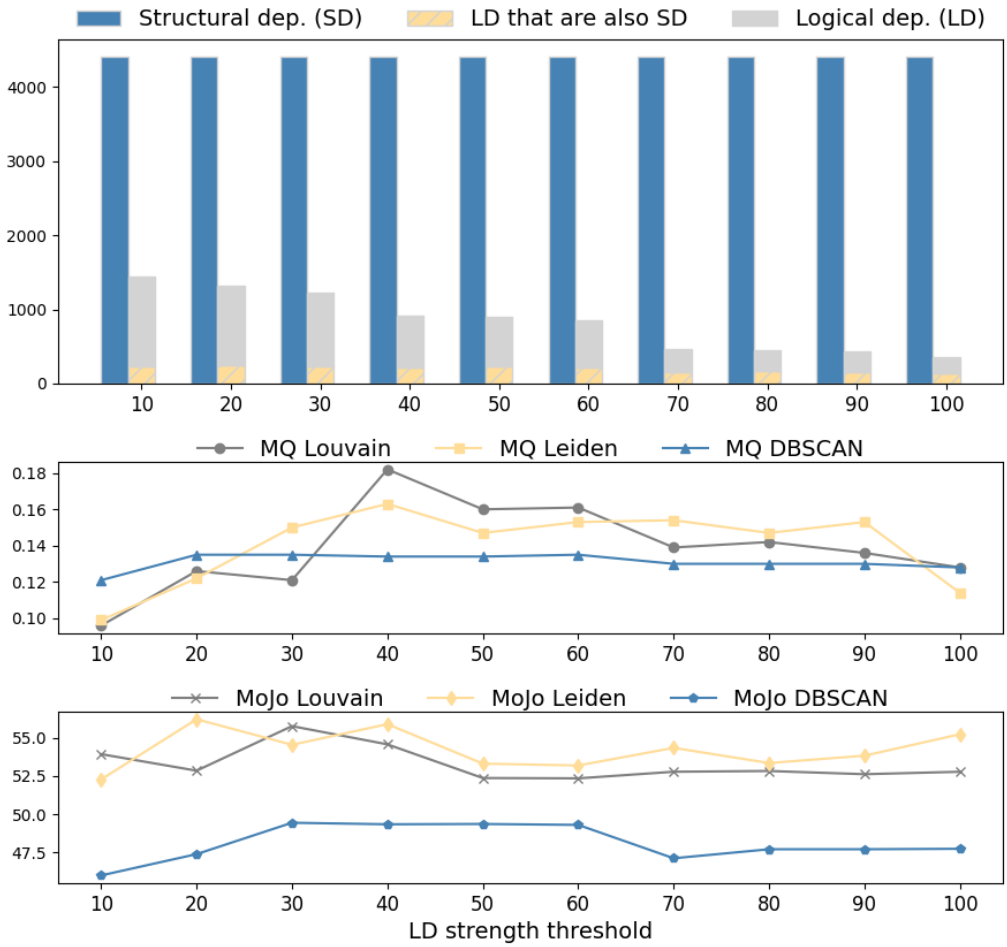

Figure 1.7: Hibernate ORM: Overlap between structural and logical dependencies and its correlation with clustering metrics.

and Tomcat, all SD+LD combinations achieve better MQ values than SD-only.

LD-only produced the best results for MQ at 40% strength for both Louvain and Leiden, and the best MoJoFM value was also observed at the same threshold for both algorithms. It is the only system where the best MQ result for LD-only occurs at a lower strength threshold than 100%. This is due to the very low number of entities remaining in the system at 100% (only 18 out of 210).

In this particular system, it is more visible that the Leiden clustering algorithm does not improve the Louvain algorithm in some scenarios. This observation is based on the fact that the values obtained for both MQ and MoJoFM metrics are the same in most cases for the Gson system for both algorithms.

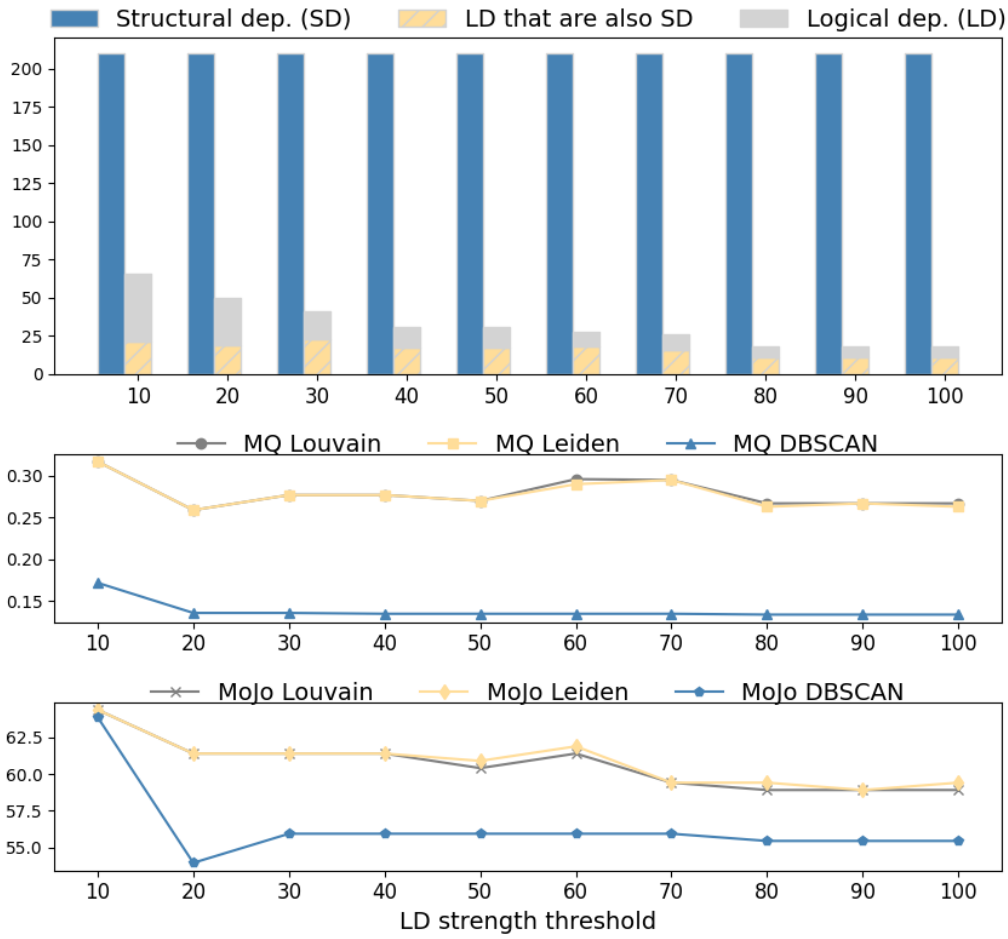

Figure 1.8: Google Gson: Overlap between structural and logical dependencies and its correlation with clustering metrics.

It can also be observed that Gson has identical metric values for MQ and MoJoFM across multiple strength thresholds. Again, the small number of commits and the size of the system contribute to the stability of these metrics.

Gson also has relatively high overlap rates between LD and SD compared to the other systems, as shown in Figure 1.8. Despite the constant values, the trend of decreasing performance for SD+LD with stricter strength filtering for LD is also present in Gson.

The results for this system highlight the difficulty of achieving better clustering solutions using logical dependencies in smaller systems with fewer commits. However, even for a small system like Gson, an improvement is still visible when using logical dependencies.

1.6.2. Research Questions and Findings

In this section, we will answer our research questions based on the results of our experiments.

RQ1: *Does using structural dependencies (SD) combined with logical dependencies (LD) improve software clustering results compared to traditional approaches using only structural dependencies (SD)?*

Based on the results from all four systems analyzed, the combination of SD and LD performed better than SD-only according to the MQ and MoJoFM metrics, confirming that using SD+LD improves software clustering results. Across different strength thresholds, SD+LD achieved higher MQ and MoJoFM values for all clustering algorithms, indicating better modularization. The filtering thresholds applied to logical dependencies influence the effectiveness of combining SD and LD. Lower strength thresholds (10% to 40%) resulted in better MQ and MoJoFM values across all clustering algorithms compared with higher thresholds.

At higher strength thresholds, the MQ and MoJoFM results decline. This decline occurs because stricter strength thresholds significantly reduce the amount of logical dependencies. This reduction leads to fewer new relationships introduced into the clustering process. Our previous research on the overlap between SD and LD showed that higher strength thresholds correlate with increased overlap between the two dependencies. This overlap indicates that at higher strength thresholds, not much new information is added to the system besides what is already introduced by structural dependencies, reducing the impact of logical dependencies on clustering results.

However, when considering a lower strength threshold, the relationship between the system size and the number of commits in the version history becomes an important factor.

In the case of Hibernate, a stricter strength threshold was needed to achieve the best metric values compared to other systems. Although the metrics obtained at a 10% strength threshold for LD are better than SD-only, the system reached peak metric values at 40% threshold across all algorithms.

With 16,609 commits and 4,414 entities, Hibernate has a significantly lower average number of commits per entity than Ant (14,917 commits and 517 entities) or Tomcat (22,698 commits and 662 entities). This lower ratio means that each entity in Hibernate is, on average, involved in fewer commits. As a result, the co-change data extracted for logical dependencies are sparser and contain more noise at lower strength thresholds.

A stricter strength threshold (e.g., 40%) filters out these weaker logical dependencies.

In contrast, systems like Ant and Tomcat, with higher ratios of commits to entities, obtain better results with logical dependencies at a 10% strength threshold because entities participate in more commits on average.

In conclusion, with an appropriate threshold, combining SD with LD leads to better clustering results than using SD alone.

RQ2: *Can using only logical dependencies (LD) produce good software clustering results?*

Using LD-only produced good clustering results, especially at higher strength thresholds. LD(100) produced the highest MQ and MoJoFM values for most systems compared with SD-only and SD+LD. However, the coverage of LD-only is significantly lower at these higher thresholds. For all systems, after filtering with 100% strength threshold, the system coverage of the remaining logical dependencies is less than 17% of the total known dependencies in the system.

Thus, while LD(100) provides the highest metric results compared to SD+LD and SD-only, it represents only a small subset of the system's dependencies.

On the other hand, LD(10) has, in most of the cases, better metric results than those for SD-only and SD+LD with better system coverage. Apache Ant and Tomcat LD(10) cover more than 60% of the system, while for Hibernate and Gson, the coverage is slightly above 30%.

Therefore, LD(10) can be an alternative to SD-only or SD+LD, especially if structural dependencies are not available.

It is also important to consider that LD-only performance at higher strength thresholds depends on the system's characteristics, such as the number of commits and entities. For Gson project, the performance at a 100% strength threshold is not as good as for the other systems, reaching its peak at a 40% threshold. This is due to the low number of entities remaining at higher thresholds, with only 18 entities at the 80% to 100% strength thresholds, and the relatively small number of commits considered.

In summary, while LD-only can produce good clustering results, especially at higher strength thresholds, its limited coverage reduces its usability, as the clustering is intended for the entire system, not just a small subset. LD-only offers a good alternative to SD-only at lower strength thresholds, providing acceptable coverage.

RQ3: *How do different filtering settings for logical dependencies (LD) impact clustering results, and which filtering settings provide the best performance?*

The impact of different filtering settings on clustering performance was observed across all systems. For LD-only clustering, lower strength thresholds like 10% provided good system coverage but had lower MQ and MoJoFM values compared to higher thresholds like 100%, where the best metric results were often achieved. However, at these higher thresholds, the system coverage was significantly reduced.

The best performance was generally observed with strength thresholds between 10% and 40% for the combination of SD and LD (SD+LD). At these thresholds, the clustering solutions achieved higher MQ and MoJoFM values than SD alone.

It is important to select the optimal filtering threshold. Logical dependencies filtered with lower strength thresholds include more relationships, introducing more knowledge that could improve clustering results. However, a too-low threshold may sometimes introduce noise, especially in systems with a low commits-to-entities ratio. On the other hand, higher thresholds significantly reduce noise but can exclude valuable dependencies and decrease coverage.

The optimal filtering threshold may vary depending on system characteristics (number of commits, size of the system), so it is important to consider these factors when filtering logical dependencies.

1.7. Discussion on Ant clustering

Based on the results from Table **??**, we can observe that the combined approach of structural dependencies and logical dependencies gives the highest Modularity Quality (MQ) metric of 0.227 with a strength threshold of 30%, which is an improvement over the 0.08 MQ metric obtained when considering only structural dependencies.

Beyond the positive result indicated by the MQ metric, we searched for further validation by human software engineering expert opinion. After thoroughly studying and understanding the analyzed system source code and documentation, we evaluated the remodularization proposals resulting from the two clustering solutions.

The two clustering solutions compared are the clustering solution obtained only from structural dependencies, in comparison to the clustering solution obtained from using both structural and logical dependencies, filtered with a threshold of 30% for strength.

The clustering solution relying solely on structural dependencies consists of 12 clusters, while the solution using both structural and logical dependencies consists of 15 clusters. Both solutions involve the same number of entities (517). The entities listed below are placed in different clusters:

- taskdefs.Available$FileDir
- taskdefs.Concat and its inner classes taskdefs.Concat$1, taskdefs.Concat$MultiReader, taskdefs.Concat$ TextElement
- taskdefs.Javadoc$AccessType
- util.WeakishReference and its inner class util.WeakishReference$HardReference
- taskdefs.Replace and its inner classes taskdefs.Replace$ NestedString, taskdefs.Replace$Replacefilter

The migration of entities between clusters is illustrated in Figure 1.9. Given that the inner classes are shifted from one cluster to another in the same way as the outer class, we omitted the inner classes from the diagram.

As the cluster number itself is not significant and may vary across different script runs (labels might vary), we will refer to each individual cluster resulting from structural dependencies as *Cluster A* and to the ones resulting from both logical and structural dependencies as *Cluster B*.

1.7.1. taskdefs.Concat and its inner classes

To have a better overview of how and why entities are transferred between clusters, we depicted Concat's logical and structural connections in Figure 1.10. Additionally, Figure 1.11 illustrates connections within Cluster A, while Figure 1.12 does the same for Cluster B.

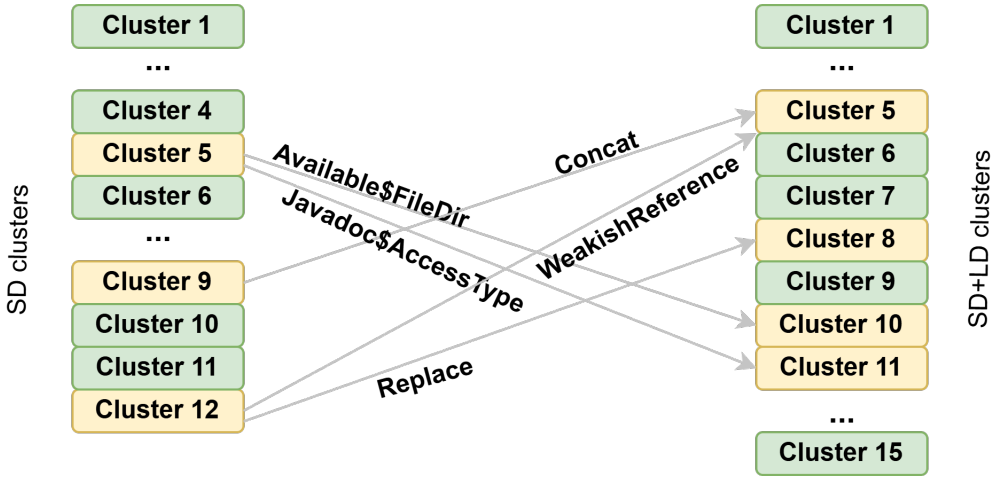

Figure 1.9: Migration of entities between clusters

In Cluster A, the Concat class and its inner classes (Concat$1, Concat$MultiReader, Concat$TextElement) are placed together with conditions like Available, And, Or, IsTrue, Equals, IsReference, Contains.

On the other hand, in Cluster B, they are placed with classes associated with file manipulation and archive operations such as Ear, Jar, War, and Zip, as well as utility classes for file handling like FileUtils and JavaEnvUtils, and entities for zip file processing (ZipEntry, ZipFile). This placement is due to the logical dependencies that Concat class has with FileUtils and FileSet in the versioning system.

To assess whether the placement of Concat in Cluster B is better than in Cluster A, we referred to the official Ant documentation. According to the documentation: "This class contains the 'concat' task, used to concatenate a series of files into a single stream" [**?**]. Therefore, judging by its usage and purpose according to the documentation, positioning the Concat class along with its inner classes in Cluster B is more suitable than in Cluster A.

1.7.2. taskdefs.Available$FileDir

In Cluster A the entity 'taskdefs.Available$FileDir' is in the same cluster with entities that are related to the build process (ProjectHelper, TaskAdapter, ComponentHelper), but not with entities that have any relation to condition checks or file existence evaluations or with its outer class.

Cluster B contains entities that are related to condition checking (And, Contains, Equals, Or, UpToDate), and also with the outer class of 'taskdefs.Available$FileDir', 'taskdefs.Available'. The movement of entities from Cluster A to Cluster B was influenced by the logical dependencies between 'taskdefs.Available' and 'taskdefs.Available$FileDir'.

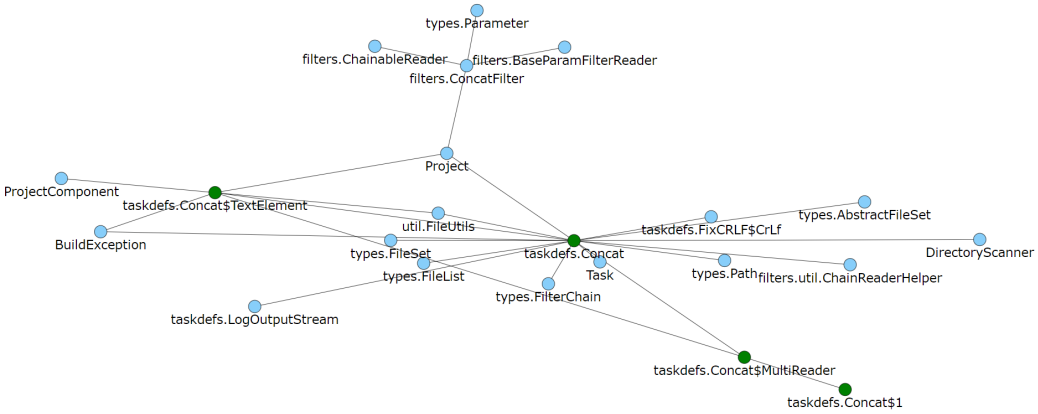

Figure 1.10: Dependencies (LD and SD) of Concat class

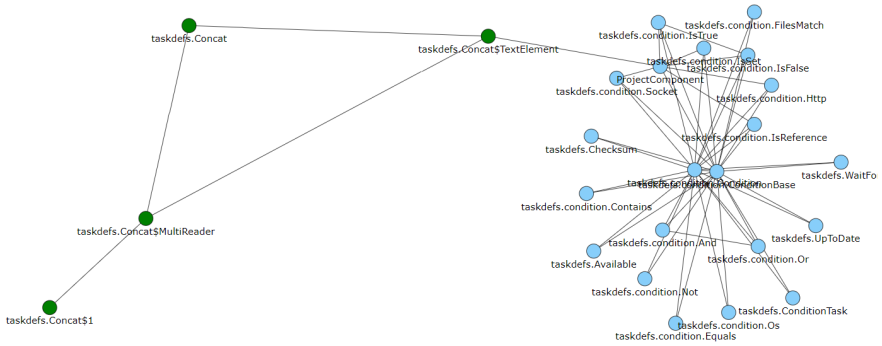

Figure 1.11: Placement of Concat in ClusterA (SD); cluster size: 25

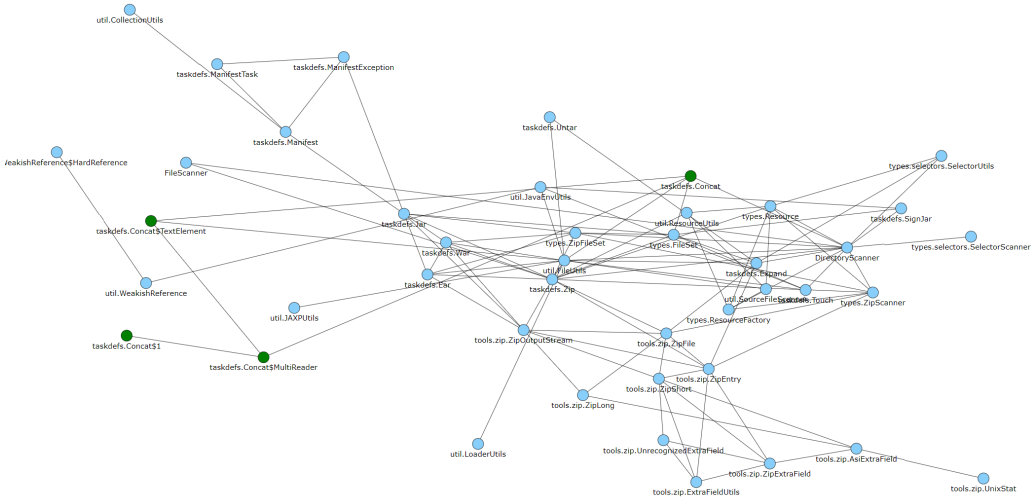

Figure 1.12: Placement of Concat in ClusterB (SD and LD); cluster size: 52

The documentation description for 'taskdefs.Available' states: "Will set the given property if the requested resource is available at runtime. This task may also be used as a condition by the condition task."[**?**]. This reinforces its placement in Cluster B, which is centered around task definitions and conditions related to build processes.

1.7.3. taskdefs.Replace and its inner classes

We depicted the logical and structural connections of Replace in Figure 1.13. Figure 1.14 illustrates connections within Cluster A, while Figure 1.15 does the same for Cluster B.

Replace and its inner classes are placed in Cluster B with entities with which they share common functionality and purpose, such as Copydir, Delete, DependSet, or MatchingTask. These entities are involved in similar tasks and operations. On the other hand, the placement in Cluster A, with WeakishReference and its inner class doesn't seem good, as these entities are completly unrelated.

The movement of entities from Cluster A to Cluster B was influenced by the strong logical dependency between 'taskdefs.Replace' and 'taskdefs.MatchingTask'.

The cluster from figure 1.14 is the 12th cluster from the clustering solution based on structural dependencies. This particular cluster is essentially comprised of two unrelated entities: Replace and WeakishReference. Due to their logical dependencies, both entities are grouped within larger clusters that share similar functionalities in the SD and LD solution.

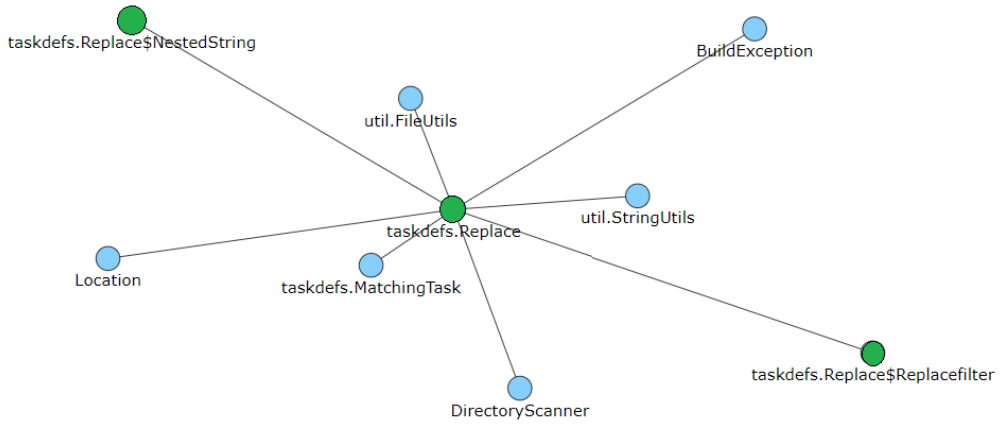

Figure 1.13: Ant dependencies (LD and SD) of Replace and its inner classes

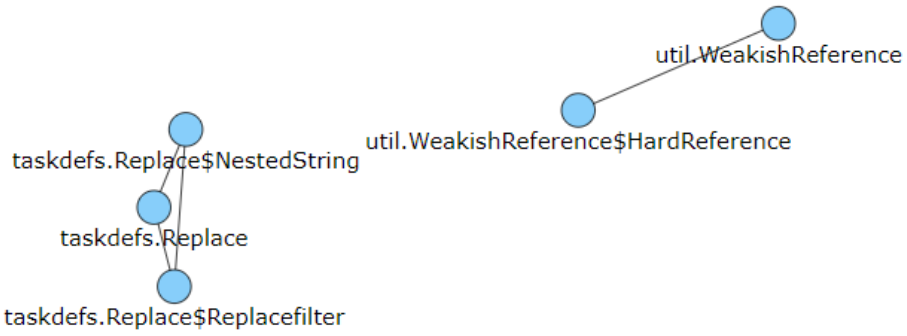

Figure 1.14: Placement of Replace in ClusterA (SD); cluster size: 5

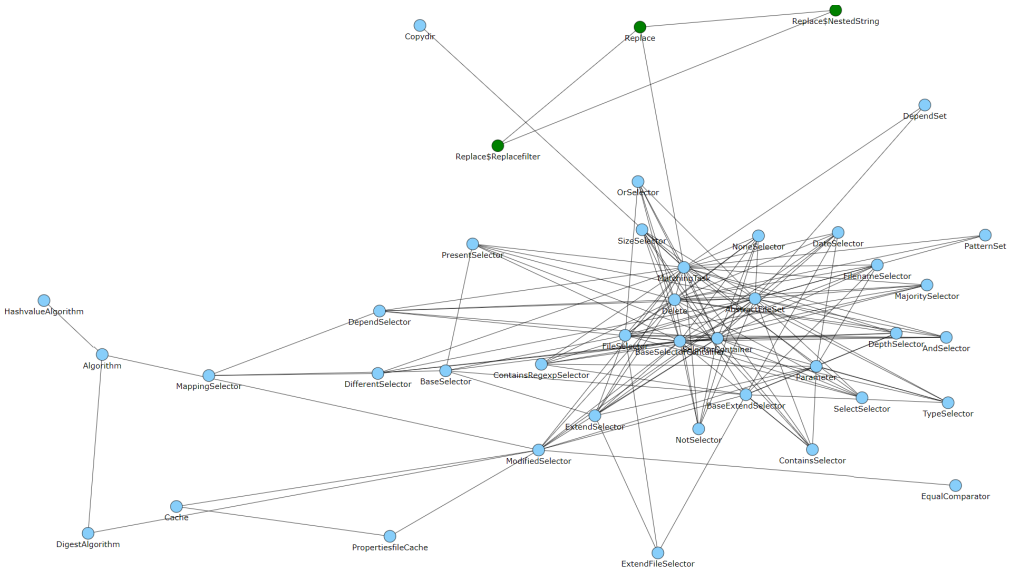

Figure 1.15: Placement of Replace in ClusterB (SD and LD); cluster size: 42

BIBLIOGRAPHY

[1] Ioana Şora and Ciprian-Bogdan Chirila. Finding key classes in object-oriented software systems by techniques based on static analysis. *Information and Software Technology*, 2019.

[2] Gustavo Oliva and Marco Aurelio Gerosa. A method for the identification of logical dependencies. In *Proceedings of the 7th International Conference on Global Software Engineering (ICGSEW)*, pages 70–72, 2012.

[3] Nemitari Ajienka and Andrea Capiluppi. Understanding the interplay between the logical and structural coupling of software classes. *Journal of Systems and Software*, 134, 2017.

[4] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'99)*, pages 50–59, 1999.

[5] Zhihua Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, pages 194–203, 2004.

[6] V. Tzerpos and R. C. Holt. Accd: an algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 258–267, 2000.

[7] V. Tzerpos and R. C. Holt. Mojo: a distance metric for software clusterings. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 187–193, 1999.

[8] P. Andritsos and V. Tzerpos. Information-theoretic software clustering. *IEEE Transactions on Software Engineering*, 31(2):150–165, February 2005.

[9] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello. Investigating the use of lexical information for software system clustering. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 35–44, 2011.

[10] Anna Corazza, Sergio Di Martino, and Giuseppe Scanniello. A probabilistic based approach towards software system clustering. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 88–96, 2010.

[11] Nicolas Anquetil and Timothy Lethbridge. File clustering using naming conventions for legacy systems. *Proceedings of the Second Working Conference on Reverse Engineering*, 1998.

[12] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic (lexical) and structural information. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, pages 103–112, 2001.

[13] A. E. Hassan Wu and R. C. Holt. Comparison of clustering algorithms in the context of software evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 525–535, 2005.

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

[15] Amarjeet Prajapati, Anshu Parashar, and Amit Rathee. Multi-dimensional information-driven many-objective software remodularization approach. *Frontiers of Computer Science in China*, 17(3):173209, 2023.

[16] Ioana Şora. Software architecture reconstruction through clustering: Finding the right similarity factors. In *Proceedings of the 2013 International Conference on Software Maintenance*, pages 45–54, 2013.

[17] I. Şora, G. Glodean, and M. Gligor. Software architecture reconstruction: An approach based on combining graph clustering and partitioning. In *2010 International Joint Conference on Computational Cybernetics and Technical Informatics*, pages 259–264, 2010.

[18] Luciana Silva, Marco Valente, and Marcelo Maia. Co-change clusters: Extraction and application on assessing software modularity. *Transactions on Aspect-Oriented Software Development*, 2015.

[19] Adelina-Diana Stana and Ioana Şora. Logical dependencies: Extraction from the versioning system and usage in key classes detection. *Computer Science and Information Systems*, 20:25–25, 2023.

[20] Adelina-Diana Stana and Ioana Şora. Analyzing information from versioning systems to detect logical dependencies in software systems. In *2019 IEEE 13th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 15–20, 2019.

[21] Adelina-Diana Stana and Ioana Şora. Identifying logical dependencies from co-changing classes. In *Proceedings of the 2019 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 486–493, 2019.

[22] T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 563–572, 2004.

[23] Vincent Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008.

[24] Andrea Lancichinetti and Santo Fortunato. Community detection algorithms: A comparative analysis. *Physical Review E, Statistical, Nonlinear, and Soft Matter Physics*, 80(5), 2009.

[25] V. Traag, L. Waltman, and Nees Jan van Eck. From louvain to leiden: guaranteeing well-connected communities. *Scientific Reports*, 9:5233, 2019.

[26] T. Bonald, N. de Lara, Q. Lutz, and B. Charpentier. Scikit-network: Graph analysis in python. *Journal of Machine Learning Research*, 21(185):1–6, 2020.

[27] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*, pages 226–231, 1996.

[28] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC'98)*, pages 45–52, 1998.

[29] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 167–176, New York, NY, USA, 2005. ACM.

[30] Trosky B. Callo Arias, Pieter van der Spek, and Paris Avgeriou. A practice-driven systematic review of dependency analysis solutions. *Empirical Software Engineering*, 16(5):544–586, Oct 2011.

[31] Adelina Diana Stana. and Ioana Şora. Identifying logical dependencies from co-changing classes. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*, pages 486–493. INSTICC, SciTePress, 2019.

[32] Nemitari Ajienka and Andrea Capiluppi. Understanding the interplay between the logical and structural coupling of software classes. *Journal of Systems and Software*, 134:120–137, 2017.

[33] David Binkley. Source code analysis: A road map. pages 104–119, 06 2007.

[34] Igor Scaliante Wiese, Rodrigo Takashi Kuroda, Reginaldo Re, Gustavo Ansaldi Oliva, and Marco Aurélio Gerosa. An empirical study of the relation between strong change coupling and defects using history and social metrics in the apache aries project. In Ernesto Damiani, Fulvio Frati, Dirk Riehle, and Anthony I. Wasserman, editors, *Open Source Systems: Adoption and Impact*, pages 3–12, Cham, 2015. Springer International Publishing.

[35] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control With Subversion for Subversion 1.6: The Official Guide And Reference Manual*. CreateSpace, Paramount, CA, 2010.

[36] Fabian Beck and Stephan Diehl. On the congruence of modularity and code coupling. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 354–364, New York, NY, USA, 2011. ACM.

[37] Nemitari Ajienka, Andrea Capiluppi, and Steve Counsell. An empirical study on the interplay between semantic coupling and co-change of software classes. *Empirical Software Engineering*, 23(3):1791–1825, 2018.

[38] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, Feb 2009.

[39] Gustavo Ansaldi Oliva and Marco Aurelio Gerosa. On the interplay between structural and logical dependencies in open-source software. In *Proceedings of the 2011 25th Brazilian Symposium on Software Engineering*, SBES '11, pages 144–153, Washington, DC, USA, 2011. IEEE Computer Society.

[40] Gustavo Ansaldi Oliva and Marco Aurélio Gerosa. Experience report: How do structural dependencies influence change propagation? an empirical study. In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*, pages 250–260, 2015.

[41] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM.

[42] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.

[43] Liguo Yu. Understanding component co-evolution with a study on linux. *Empirical Software Engineering*, 12(2):123–141, Apr 2007.

[44] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 190–, Washington, DC, USA, 1998. IEEE Computer Society.

[45] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, IWPSE '03, pages 13–, Washington, DC, USA, 2003. IEEE Computer Society.

[46] Mark Shtern and Vassilios Tzerpos. Clustering methodologies for software engineering. *Adv. Soft. Eng.*, 2012:1:1–1:1, January 2012.

[47] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009.

[48] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *2010 17th Working Conference on Reverse Engineering*, pages 119–128, Oct 2010.

[49] Ioana Şora. Software architecture reconstruction through clustering: Finding the right similarity factors. In *Proceedings of the 1st International Workshop in Software Evolution and Modernization - Volume 1: SEM, (ENASE 2013)*, pages 45–54. INSTICC, SciTePress, 2013.

[50] Ioana Şora. Helping program comprehension of large software systems by identifying their most important classes. In *Evaluation of Novel Approaches to Software Engineering - 10th International Conference, ENASE 2015, Barcelona, Spain, April 29-30, 2015, Revised Selected Papers*, pages 122–140. Springer International Publishing, 2015.

[51] Ioana Şora. A PageRank based recommender system for identifying key classes in software systems. In *2015 IEEE 10th Jubilee International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 495–500, May 2015.

[52] Ioana Şora. Helping program comprehension of large software systems by identifying their most important classes. In Leszek A. Maciaszek and Joaquim Filipe, editors, *Evaluation of Novel Approaches to Software Engineering*, pages 122–140, Cham, 2016. Springer International Publishing.

[53] Ioana Şora, Gabriel Glodean, and Mihai Gligor. Software architecture reconstruction: An approach based on combining graph clustering and partitioning. In *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on*, pages 259–264, May 2010.

[54] Andy Zaidman and Serge Demeyer. Automatic identification of key classes in a software system using webmining techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6):387–417, 2008.

[55] Yann-Gaël Guéhéneuc. A reverse engineering tool for precise class diagrams. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '04, pages 28–41. IBM Press, 2004.

[56] M. L. Collard, H. H. Kagdi, and J. I. Maletic. An XML-based lightweight C++ fact extractor. In *11th IEEE International Workshop on Program Comprehension, 2003.*, pages 134–143, May 2003.

[57] M. L. Collard, H. H. Kagdi, and J. I. Maletic. An XML-based lightweight C++ fact extractor. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 134–, Washington, DC, USA, 2003. IEEE Computer Society.

[58] Michael L. Collard, Michael J. Decker, and Jonathan I. Maletic. Lightweight transformation and fact extraction with the srcML toolkit. In *Proceedings of the 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, SCAM '11, pages 173–184, Washington, DC, USA, 2011. IEEE Computer Society.

[59] Adelina Diana Stana and Ioana Şora. Identifying logical dependencies from co-changing classes. In *Submitted to The 7th International Workshop on Software Mining (SoftwareMining) at ASE 2018*, 2018.

[60] Adelina Diana Stana. An analysis of the relationship between structural and logical dependencies in software systems. Master's thesis, Politehnica University Timisoara, Romania, June 2018.

[61] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071, Oct 2016.

[62] Xiaoxia Ren, B. G. Ryder, M. Stoerzer, and F. Tip. Chianti: a change impact analysis tool for java programs. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 664–665, May 2005.

[63] Grady Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[64] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35:864–878, 2009.

[65] Liguo Yu. Understanding component co-evolution with a study on linux. *Empirical Softw. Engg.*, 12(2):123–141, April 2007.

[66] P. Wang, J. Yang, L. Tan, R. Kroeger, and J. David Morgenthaler. Generating precise dependencies for large software. In *2013 4th International Workshop on Managing Technical Debt (MTD)*, pages 47–50, May 2013.

[67] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, SE-11(8):749–757, Aug 1985.

[68] P. K. Linos and V. Courtois. A tool for understanding object-oriented program dependencies. In *Proceedings 1994 IEEE 3rd Workshop on Program Comprehension- WPC '94*, pages 20–27, Nov 1994.

[69] Norman Wilde. Understanding program dependencies, 1990.

[70] Keith H. Bennett and Vaclav Rajlich. Software maintenance and evolution: a roadmap. pages 73–87, 05 2000.

[71] Hongji Yang and Martin Ward. *Successful Evolution of Software Systems*. Artech House, Inc., Norwood, MA, USA, 2003.

[72] Bennet P. Lientz and E. Burton Swanson. Problems in application software maintenance. *Commun. ACM*, 24(11):763–769, November 1981.

[73] Steven Fraser, Frederick Brooks, Jr, Martin Fowler, Ricardo Lopez, Aki Namioka, Linda M. Northrop, David Parnas, and Dave Thomas. "no silver bullet" reloaded: retrospective on "essence and accidents of software engineering". pages 1026–1030, 01 2007.

[74] Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[75] K H. Bennett, Dh Le, and Vaclav Rajlich. The staged model of the software lifecycle: A new perspective on software evolution. 05 2000.

[76] Franz Lehner. Software life cycle management based on a phase distinction method. *Microprocessing and Microprogramming*, 32:603–608, 08 1991.

[77] K. Bennett. Legacy systems: coping with success. *IEEE Software*, 12(1):19–23, Jan 1995.

[78] S. S. Yau, J. S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *The IEEE Computer Society's Second International Computer Software and Applications Conference, 1978. COMPSAC '78.*, pages 60–65, Nov 1978.

[79] Vaclav Rajlich. Modeling software evolution by evolving interoperation graphs. *Ann. Software Eng.*, 9:235–248, 05 2000.

[80] S A. Bohner and R S. Arnold. Software change impact analysis. *IEEE Computer Society*, 1, 01 1996.

[81] L Bass, P Clements, and Rick Kazman. Software architecture in practice 2nd edition. 01 2003.

[82] Kamran Sartipi. Software architecture recovery based on pattern matching. 09 2003.

[83] Gerardo Canfora and Massimiliano Di Penta. New frontiers of reverse engineering. pages 326 – 341, 06 2007.

[84] E.J. Chikofsky, Cross , and II . Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7:13–17, 02 1990.

[85] Ira Baxter, Andrew Yahin, Leonardo de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. volume 368-377, pages 368–377, 01 1998.

[86] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28:654– 670, 08 2002.

[87] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. pages 244–, 01 1996.

[88] Andrian Marcus and J.I. Maletic. Identification of high-level concept clones in source code. pages 107– 114, 12 2001.

[89] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. 11 2002.

[90] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. 01 1999.

[91] Spencer Rugaber. Program comprehension. 08 1997.

[92] Bennet Lientz, E Burton Swanson, and Gerry E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21:466–471, 06 1978.

[93] Dag Sjøberg, Tore Dybå, and Magne Jorgensen. The future of empirical methods in software engineering research. pages 358–378, 06 2007.

[94] James A. Jones and Mary Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. pages 273–282, 01 2005.

[95] Holger Cleve and Andreas Zeller. Locating causes of program failures. pages 342– 351, 06 2005.

[96] Hongji Yang and Martin Ward. Successful evolution of software systems. 01 2003.

[97] R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, Feb 1991.

[98] M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6):864–878, Nov 2009.

[99] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. An empirical study on the developers' perception of software coupling. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 692–701, May 2013.

[100] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, May 2015.

[101] S. Li, H. Tsukiji, and K. Takano. Analysis of software developer activity on a distributed version control system. In *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 701–707, March 2016.

[102] M. Abbes, F. Khomh, Y. Gueheneuc, and G. Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 181–190, March 2011.

[103] F. Khomh, M. Di Penta, and Y. Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84, Oct 2009.

[104] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17:243–275, 06 2012.

[105] Ruven E. Brooks. Towards a theory of the cognitive processes in computer programming. *Int. J. Hum.-Comput. Stud.*, 51:197–211, 08 1999.

[106] Iris Vessey. Expertise in debugging computer programs. 12 1984.

[107] V. Y. Shen, Tze-jie Yu, S. M. Thebaut, and L. R. Paulsen. Identifying error-prone software—an empirical study. *IEEE Transactions on Software Engineering*, SE-11(4):317–324, April 1985.

[108] Stana Adelina and Şora Ioana. Analyzing information from versioning systems to detect logical dependencies in software systems. In *International Symposium on Applied Computational Intelligence and Informatics (SACI)*, May 2019.

[109] Ioana Şora and Ciprian-Bogdan Chirila. Finding key classes in object-oriented software systems by techniques based on static analysis. *Information and Software Technology*, 116:106176, 2019.

[110] P. Meyer, H. Siy, and S. Bhowmick. Identifying important classes of large software systems through k-core decomposition. *Adv. Complex Syst.*, 17, 2014.

[111] D. Steidl, B. Hummel, and E. Juergens. Using network analysis for recommendation of central software classes. In *2012 19th Working Conference on Reverse Engineering*, pages 93–102, 2012.

[112] L. Tahvildari and K. Kontogiannis. Improving design quality using meta-pattern transformations: a metric-based approach. *J. Softw. Maintenance Res. Pract.*, 16:331–361, 2004.

[113] Ioana Şora. Finding the right needles in hay - helping program comprehension of large software systems. In *Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*, pages 129–140. INSTICC, SciTePress, 2015.

[114] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[115] Yi Ding, B. Li, and Peng He. An improved approach to identifying key classes in weighted software network. *Mathematical Problems in Engineering*, 2016:1–9, 2016.

[116] Weifeng Pan, Beibei Song, Kangshun Li, and Kejun Zhang. Identifying key classes in object-oriented software using generalized k-core decomposition. *Future Generation Computer Systems*, 81:188–202, 2018.

[117] L. do Nascimento Vale and M. de A. Maia. Keecle: Mining key architecturally relevant classes using dynamic analysis. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 566–570, 2015.

[118] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Ninth European Conference on Software Maintenance and Reengineering*, pages 134–142, 2005.

[119] M. Kamran, M. Ali, and B. Akbar. Identification of core architecture classes for object-oriented software systems. *Journal of Applied Computer Science & Mathematics*, 10:21–25, 2016.

[120] Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006. ROC Analysis in Pattern Recognition.

[121] Andrew P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997.

[122] M. H. Osman, M. R. V. Chaudron, and P. v. d. Putten. An analysis of machine learning algorithms for condensing reverse engineered class diagrams. In *2013 IEEE International Conference on Software Maintenance*, pages 140–149, 2013.

[123] Ferdian Thung, David Lo, Mohd Hafeez Osman, and Michel R. V. Chaudron. Condensing class diagrams by analyzing design and network metrics using optimistic classification. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, page 110–121, New York, NY, USA, 2014. Association for Computing Machinery.

[124] X. Yang, D. Lo, X. Xia, and J. Sun. Condensing class diagrams with minimal manual labeling cost. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 22–31, 2016.

[125] A. Mubarak, S. Counsell, and R. M. Hierons. An evolutionary study of fan-in and fan-out metrics in oss. In *2010 Fourth International Conference on Research Challenges in Information Science (RCIS)*, pages 473–482, 2010.

[126] srcml; www.srcml.org.

[127] Fionn Murtagh and Pedro Contreras. Algorithms for hierarchical clustering: An overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(2):86–97, 2012.