Measuring the degree of library dependency

Núria Bruch Tàrrega
nuria.bruch@gmail.com

November 2, 2020, 97 pages

Academic supervisor: Dr. Ana M. Oprescu, ana.oprescu@uva.nl
Host supervisor: Dr. Lodewijk Bergmans, l.bergmans@sig.eu
Host supervisor: Dr. Miroslav Živković, m.zivkovic@sig.eu
Host organisation: Software Improvement Group (SIG), www.softwareimprovementgroup.com

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

The usage of libraries, both commercial and open-source, provides the implementation of certain functionalities and is a widespread practice among developers. The usage of these libraries allows developers to avoid duplicating code by reusing it instead. However, when a developer uses a library in a software product, this creates a dependency. This dependency may result in transitive dependencies when the library depends on other libraries. The dependencies created when reusing a library can also carry problems — if a library has a security issue, it can be propagated to the software product, which depends on it, directly or indirectly. To deal with dependencies, developers can use package managers, which allow them to install and update the libraries they use. However, these package managers generally do a simple evaluation of the dependencies: either there is a dependency or not. Hence, a detailed evaluation of the dependencies is missing, which could help developers deal with vulnerabilities, breaking changes, and deprecated dependencies.

In this thesis, we propose a model for software dependencies, which can help to provide a fine-grained evaluation of them. The model includes three types of metrics: coupling, coverage, and usage per class. For each metric in the model, we provide a formal definition and a theoretical validation by proving the metrics' properties. We additionally implemented a proof-of-concept tool that, given a library from the *Maven Central Repository*, calculates the metrics of the model for each of the dependencies using bytecode analysis. Moreover, the proof-of-concept includes a visualization of the dependency tree, including the calculated metrics.

Finally, we conducted experiments to validate the model, the implementation of the proof-of-concept, and the visualization. The experiments include interviews with 15 professional developers who evaluated the clarity and actionability of the model's metrics and the proposed visualizations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Currently, many libraries are available for developers, both commercial and open-source. Using libraries is becoming more and more popular [1] since it allows reusing previously developed code and helps developers avoid implementing the same functionalities multiple times. For example, the Maven Repository Central, which contains the artifacts of a large amount of Java libraries, included 2.8M artifacts in 2018, which is 13.5x more artifacts than in 2011 [2].

When a developer uses a library in a project, it creates a dependency between the project and the library. It implies that a significant number of projects depend on other libraries. The Maven Repository Central includes more than 9M dependencies between artifacts, dated 2018 [2]. This adds the task of managing these dependencies to the maintenance tasks of the project — proper maintenance of the dependencies of a project is also part of the software applications' health and security. The management of dependencies is one of the problems that software engineering is trying to solve [3]. For instance, an update of a dependency may require changing part of the code if the update contains breaking changes [4].

The management and maintenance of the dependencies of a project is an important task. External libraries, just like any other software product, can have security vulnerabilities that may affect the projects that depend on these libraries. For example, some security vulnerabilities can become problems that can negatively impact a software product regarding integrity, privacy, or availability [1].

Currently, developers have package managers at their disposal to ease managing the dependencies of their projects. However, the dependency management available in these package managers only evaluates if a dependency exists or not, and a more detailed evaluation is missing [5]. For instance, there is no way to evaluate how much a project depends on a library or how much of the library is used. Therefore, there is no way to know how likely it is that the project is affected by a vulnerability in the library and which parts of the project's code would need updating in case of a breaking change in the library.

Therefore, this thesis aims to create a model to evaluate the dependencies to obtain information on the *actual usage* of the dependencies. A set of metrics is proposed to measure the dependencies between projects and the dependencies these have. The metrics are designed to evaluate the dependencies according to three different perspectives: (1) the code affected by the dependency, (2) how much of a library is used, and (3) the usage of the dependency per class.

This project has been carried out in collaboration with the company *Software Improvement Group (SIG)*, and it is motivated by the *FASTEN* project [2]. The *FASTEN* project aims to improve the quality of open-source development environments to make them more secure and reliable. For this reason, one of the goals is to analyze the software library dependencies in more detail.

1.1 Research questions

To address the problems described in the previous section, we formulate the following research questions:

RQ1: *How can we measure the degree of code dependency between two software products with a direct dependency?*

[1] `https://cve.mitre.org/cve/cna/rules.html#section_7-1_what_is_a_vulnerability`
[2] `https://www.fasten-project.eu/`

With this question, we want to propose a set of metrics to measure a dependency from the product's point of view that has a dependency on another product. We want to measure how much the project is affected by the dependency.

- **RQ1.1:** *What constitutes a dependency between two products?*
  First, we need to determine what creates a dependency — the type of connection between the products and how it can be measured.
- **RQ1.2:** *Which metrics can be used to measure the dependency?*
  We propose metrics to measure the dependency described in the previous subquestion. Existing metrics are considered, as well as new ones.
- **RQ1.3:** *How can we validate the proposed metrics?*
  There are many approaches to validate metrics used in the literature, some of which are used to validate the proposed metrics in this thesis, based on which apply to the proposed metrics.

RQ2: *How can we measure the degree of code dependency between two software products with a transitive dependency?*

Transitive dependencies involve more factors than direct dependencies, such as the propagation of the dependency and its impact. Therefore, the metrics proposed for the direct dependencies have to be adapted for the transitive ones.

RQ3: *How can we measure how much a library is used by a software product?*

For this question, we look at the dependency from another perspective. RQ1 focused on how much does the software product depends on a library. In this case, we measure how much of the library is being used by the software product.

RQ4: *How can we visualize the metrics designed to model the software dependencies?*

For the model to be usable by developers, it is necessary to create a way to visualize the results, which is usable for the target audience. The visualizations are presented to software developers to discuss their usefulness and actionability, based on the situation in which these would be used.

1.2 Research method

The main research method we use during this project is the *Technical Action Research* (TAR) [6]. This research method is artifact-based, which means that the first step is to produce the artifact meant to be used in certain situations envisioned by the researcher. The testing of this artifact, to see if it is effective in these situations, is done through several iterations. First, under ideal conditions, and subsequently changing the experiments step by step to reach a real-world situation. In this master thesis, the artifact is the proof-of-concept that implements the model for software dependencies and the proposed visualizations. Because of time constraints, performing multiple iterations is not possible. Therefore, we created a setting with an example input, in which the interviewees can use the artifact and evaluate it. However, there is the option of continuing with this part of the work in the future.

Furthermore, the research includes experiments. These experiments will be conducted as the empirical part of the validation of the metrics and the proof-of-concept implementation of both the model and the proposed visualizations.

1.3 Contributions

Considering the current state of the art in the domain of this thesis project, the main contributions made by this research are the following:

1. **Model for software dependencies:**
   We have created a model for both direct and transitive software dependencies. It contains three types of metrics, which measure the dependency from a different point of view: coupling, coverage, and usage per class. In total, there are eight metrics in the model. For each of the metrics, we provide a formal definition and a property-based theoretical validation.

2. **Proof-of-Concept tool:**
   To complement the model and be able to evaluate it, we create a proof-of-concept implementation. The tool can calculate all the model metrics for the dependency tree of a given library by using Java bytecode analysis. To calculate the metrics for each library in the dependency tree, all the libraries have to be available in the *Maven Central Repository*.
   The proof-of-concept includes a front-end which contains three visualizations of the model: a tree graph, a table, and a class distribution bar-chart.
3. **Validation:**
   To validate the other contributions, we have conducted five experiments. First, we have compared our results with the results of the research by Soto-Valero et al. [7]. Second, for the first type of metrics, the coupling metrics, we have evaluated their significance. Third, for the coupling metrics for transitive dependencies, which have a propagation factor, we have conducted a sensitivity analysis of this factor. Next, to validate the visualizations as well as the actionability and clarity of all the metrics, we have conducted expert interviews. We interviewed 15 professional developers who used the tool during the interview, considering certain usage scenarios. Finally, we have created a benchmark of the coupling metrics to understand their scale and distribution.

1.4 Outline

In Chapter 2, we describe the background of this thesis based on the literature of the domain. Chapter 3 describes the metrics created to model the dependencies between software products. The model is used in Chapter 4, which describes the creation of the proof-of-concept tool that calculates the metrics of the model. In Chapter 5, the setup and execution of the experiments are explained, and the results of the experiments are shown and discussed. The research questions are answered in chapter 6. Chapter 7 contains the work related to the domain of this thesis. Finally, we present our concluding remarks, as well as future work in Chapter 8.

Chapter 2

Background

In this chapter, we present the background information needed for the research conducted in this thesis. In addition, we also define some terminology used throughout the thesis.

2.1 Terminology

This section contains a review of the terms used in some related literature, particularly those papers that specify the terminology used. The different terms used are compared by explaining the differences. Finally, we specify the terms used in this thesis and its definition.

Term	[8]	[1]	[9]	[7]
Library	x		x	x
Package		x	x	
Application		x		
Project	x	x		x
Version		x	x	x
Instance	x			
Release			x	
Artifact				x
Package manager			x	x
Package repository			x	
Dependency	x	x	x	x
Reverse dependency		x		
Inherited dependency				x
Direct/Transitive dependency	x	x	x	x
Direct/Indirect dependency			x	
Deployed/non-deployed dependency	x		x	
Own/third-party dependency	x			
Halted dependency	x		x	
Bloated dependency				x
Dependency tree	x			x
Dependency network		x	x	
Ecosystem		x		

Table 2.1: Terminology from the literature

First, we discuss the differences between the first four terms of the table. A *library* is a software unit distributed independently and can be reused in other software components. The term *library* is replaced by the term *package* in some papers, when referring to certain software *ecosystems* or *package managers*

which use this term. The term *application* is used to differentiate the software components that are available in repositories to be reused (*libraries*) and those that are not (*applications*) [1]. The term *project* is used in different papers with different meanings. Pashchenko et al. [8] use it to refer to a group of *libraries* developed or maintained by the same team of developers. Kikas et al. [1] use it to refer to both *library* and *application*. Finally, Soto-Valero et al. use it to refer to Maven Projects.

Each *library*, available in a *package repository*, can have different *versions*. Apart from the word *version*, the literature also uses the terms *instance* and *release*, with the same meaning. The word *artifact* is used by Soto-Valero et al. [7] to refer to a Maven Artifact, which corresponds to a Maven Project version.

The papers that specifically define *package manager* or *package repository* in the terminology, it is because these tools have an important role in the research.

When describing the relationship in which a *library* uses another, relevant literature uses the term *dependency*. However, different types of *dependencies* are defined depending on the paper and the research done, which will be discussed below.

The *dependencies* declared in a *library* are always called *direct dependencies*, and those introduced by the *direct dependencies* are called *transitive* or *indirect*.

There is also the distinction between *deployed* and *non-deployed dependencies*. The *deployed* ones are those included in the software product once it is in production, and the *non-deployed* ones are those used for development tasks (e.g., testing *libraries*).

When considering the risk associated with a *dependency*, it is important to consider whether the *library* is *own* or *third-party*. An *own dependency* is maintained by the same team that develops the product that has the *dependency*.

Another particular case when considering risk is *halted dependencies*. These *dependencies* are not updated anymore, which means that if a vulnerability is found, it will not be fixed.

The last type of *dependency* is the *bloated dependency*. These *dependencies* can be *direct* or not, but have the characteristic that the *library* is never used and uselessly increase the *dependency* tree's size without the need.

Finally, the terms *dependency tree* and *dependency network* refer to the graph created by the *dependencies*. In these graphs, the *library versions* are the nodes, and the *dependencies* are the edges. The *ecosystem* includes all the *libraries* and *applications* involved in the *dependency network*.

Based on the terminology discussed above, we define the terms that are used within this thesis.

- **Library:** A software artifact that is distributed independently. It can have different implementations distinguished by *versions*. The model created in this thesis is meant to analyze the *dependencies* of any software product. However, for simplicity during the thesis, we always refer to *libraries*.
- **Version:** A *version* of a *library* that contains an implementation of the *library*. Each *version* has specific metadata associated to build the *version* successfully. Among other data, it specifies which *versions* of other *libraries* it is using.
- **Dependency:** When a *library version* uses another *library version*, it creates a relationship between the *versions* of the two *libraries*, a *dependency*. In particular, the first *library version* depends on the second one.
- **Direct and transitive dependencies:** A *direct dependency* is when a *library version* directly uses a *version* of another *library*. A *transitive dependency* means a *library version* uses another one indirectly, through other *library versions* that it depends on.
- **Unused dependencies:** We have decided to use the term unused dependency, instead of *bloated dependency* since we find it easier to understand. A *dependency* is unused when included in the *library version*'s *dependency* tree, but the *library* never reaches it. This could happen with both *direct* and *transitive dependencies* for different reasons.
- **Dependency network:** Graph that represents the *dependencies* between *library versions*. In a *dependency network*, the *library versions* are the nodes, and the *dependencies* between them are the edges.
- **Ecosystem:** A set of *libraries* that have *versions* with *dependencies*. When the *libraries* are updated (new *versions*), the *ecosystem* evolves.

2.2 Dependency management

There exist several dependency managers. In this thesis, we analyze Java libraries, and in particular, we will use projects in the Maven ecosystem, *Maven Central Repository*[1]. Therefore, the dependency manager used is the one included in Maven. This section describes the characteristics of Maven, including the types of dependencies and how these are declared.

2.2.1 Maven Dependencies

The configuration of a Maven Project, including the dependency management, is done in the *Project Object Model (POM)* file. Apart from defining the dependencies of a project, the POM file contains the project's description and the build plugins that it uses.

The following parameters define a project:

- **GroupID:** The identifier of the group or company that developed the project.
- **ArtifactID:** The identifier of the project itself.
- **Version:** The version of the implementation of the project.
- **Packaging:** The packaging method that the project uses. Although other packaging methods are available, *jar* files are the default ones and can be used to analyze the bytecode of the libraries.

Module hierarchy A Maven Project can be configured using two different strategies. First, as a single module, it will only have one *POM* file, and that only one packaging will result in the build of the project. The second option is to create a multi-module project. In this case, the project has multiple *POM* files. These *POM* files can have a defined hierarchy in which there is a parent *POM* that has children *POMs*, these children will *inherit* dependencies from the parent file. For the development of the PoC of this thesis, a module with a *POM* file, even if it has a parent module, is considered a library since it can be used in a different library as an individual dependency.

Dependencies and DependencyManagement There are two sections in a *POM* file that are used for dependency management purposes: `dependencies` and `dependencyManagement`.

The `dependencyManagement` section is used in multi-module projects. It is used to define certain dependency information (e.g., the version of the artifacts). It is used in the parent *POM* to simplify the dependency definition of the children's files.

The `dependencies` section is where the dependencies are declared. If a parent file has dependencies declared in this section, these will always be inherited by the children's files.

Maven uses both sections of the *POM* file to resolve the dependencies of a library.

Scope of the dependencies One of the main mechanisms that the Maven dependency manager offers is the dependency scope specified for each dependency included in a *POM* file. A direct dependency's scope affects how the transitive dependencies are treated, except the scope import. There are 6 different scopes:

- **Compile:** The default scope, all direct dependencies without a specified scope, have compiled scope. These dependencies are available in the library's classpath and will be propagated as transitive dependencies to the libraries that depend on the current library.
- **Provided:** This scope type means that the dependency is expected to be provided by the JDK or the container during runtime. The dependencies with scope provided do not propagate transitive dependencies and are only available in the classpath on compilation and test.
- **Runtime:** In this case, the dependency is only needed during the execution, and therefore not necessary during compilation. It is available in the classpath during runtime and test. Runtime dependencies are propagated as transitive dependencies.
- **Test:** This scope specifies a dependency that is only used for testing purposes. It is available in the classpath during test and execution phases. Test dependencies do not propagate as transitive dependencies.

[1] `https://repo1.maven.org/maven2/`

- **System:** Similar to *provided*, but it is necessary to indicate the path to the jar of the dependency. It may cause problems if the product is built in a machine where the indicated path does not match the actual one. This scope is not transitive.
- **Import:** This scope is only available for dependencies declared in the section *DependencyManagement* and with specified type *pom*. It indicates that the dependency should be replaced with the dependencies declared in its pom. Therefore, these dependencies are replaced and do not affect transitivity.

Optional dependencies and exclusions In Maven, it is possible to declare dependencies as *optional*. The primary use case for this is when a project is not divided into sub-modules, and specific dependencies are only used in certain parts of the project. In this case, it might be possible to use the project without using these parts, and the dependency might not be necessary. Declaring the dependency as optional allows saving both space and memory when the dependency is not used.

Another feature of the dependency management available in Maven is the dependency *exclusions*. This feature has the goal of saving part of the memory and space used by transitive dependencies. When a particular transitive dependency is not used in your project, it is possible to specify the transitive dependency exclusion. It could be useful if you use only a part of a direct dependency that does not need the transitive dependency. The excluded dependency will not be included in the project's dependency tree and will not be imported in the library's classpath.

Dependency resolution To resolve the dependencies, Maven uses the *POM* file and scopes recursively to create the dependency tree. However, another aspect of dependency resolution is the version of each of the dependencies. It is possible that a dependency tree contains the same dependency more than once, and maybe with different versions. Maven uses the dependency mediation algorithm to resolve the version of the dependencies. The algorithm's strategy is the "nearest definition," which consists of choosing the version of the dependency closer to the root in the dependency tree. If one artifact is declared twice with different versions and at the same level, the dependency mediation will choose the first declaration of the dependency.

2.3 Coupling

When assessing the quality of software, many aspects are considered and measured. One of these, in particular for Object-Oriented systems, is coupling. Coupling measures the degree of dependency between two different parts of a system. In the literature of coupling metrics, these have been used to measure the dependence of the same system elements or give an overview of the coupling within a system. However, in this thesis, we measure the coupling, or degree of dependency, between two systems.

Therefore, we propose revisiting the existing coupling metrics, meant to measure the coupling between units of the *same* project and adapt them to measure coupling *between* projects.

According to Poshyvanyk and Marcus in [10], there are six main groups of coupling metrics:

- **Structural coupling metrics:** Measured directly from static source code analysis. Largely studied by the literature about coupling [10–13].
- **Dynamic coupling measures:** Measured using dynamic code analysis. *"Introduced as the refinement to existing coupling measures due to gaps in addressing polymorphism, dynamic binding, and the presence of unused code by static structural coupling measures"* [10].
- **Evolutionary and Logical coupling:** According to Zimmermann and Diehl [14], the evolutionary coupling can: *"tell us which parts of the system are coupled by common changes or cochanges."*
- **Coupling measures based on information entropy approach:** Coupling metrics based on the information-theory approach, such as the metrics proposed by Allen and Khoshgoftaar in [15].
- **Conceptual coupling metrics:** Based on the semantic similarity between the elements. This is the focus of the work from Poshyvanyk and Marcus [10].
- **Coupling metrics for specific types of software applications:** Specialized coupling metrics for certain kinds of projects, such as knowledge-based systems or aspect-oriented approach.

Since this research is aimed to be independent of the domain, the last category is not considered in this thesis. Moreover, the evolutionary coupling is not possible to be applied in our context. Likely, the separate projects will not evolve simultaneously, given that the same team does not develop them. Finally, the research of this project, owing to the time limitation, will be centered on the structural metrics.

These metrics are going to be proposed as a first step to measure the degree of library dependency. Nevertheless, the metrics can be extended and calculated more accurately by adding dynamic coupling and information entropy approach metrics in future work.

There are many structural coupling metrics, each measuring a different type of coupling from a different perspective, depending on the purpose for which the metrics are needed. To define the necessary metrics to measure the dependency between products, we have used the framework described by Briand et al. [11], which unifies the frameworks defined by Eder et al. [13], Hitz and Montazeri [16], and Briand et al. [12].

According to the unified framework defined by Briand et al. [11], the coupling metrics have specific characteristics defining which type of coupling they are measuring. In particular, six criteria are defined:

- **Type of connection:** This criterion defines which mechanism creates coupling, which type of dependency is measured, how the two elements are connected. The different types of connection, as described by Briand et al. [11] can be found in Table 2.2.
- **Locus of impact:** If the coupling is import or export. In other words, if the class for which coupling is being measured is the client or the server of the relationship.
- **Granularity of the measure:** The detail at which the metric calculates coupling. It is defined by 1) The domain at which coupling is measured (e.g., class-level) and 2) how the metric counts the connections (e.g., evaluating whether two elements are connected or not, or counting each one of the connections individually). The six options to count connections as defined by Briand et al. [11], can be found in Table 2.3.
- **Stability of the server:** In the framework by Briand et al. [11], the servers are classified as unstable if these are "subject to development or modification in the project at hand" and stable if these "are not subject to change in the project at hand." The last one includes classes imported from libraries. According to Briand et al., coupling with an unstable class represents more risk than coupling with a stable class. However, the framework's studied metrics do not use this criterion and treat all classes with the same importance.
- **Direct and indirect coupling:** Does the connection between the two elements are direct or transitive (there is at least one other element connecting the two). The metrics that do not account for indirect coupling can be adapted by calculating the metric's transitive closure.
- **Inheritance:** In this criteria, Briand et al. [11] define the position of the metric respecting exceptional cases such as inheritance and polymorphism.

#	Client Item	Server Item	Description
1	attribute *a* of a class *c*	class *d*, d != c	class *d* is the type of *a*
2	method *m* of a class *c*	class *d*, d != c	class *d* is the type of a parameter of *m*, or the return type of *m*
3	method *m* of a class *c*	class *d*, d != c	class *d* is the type of a local variable of *m*
4	method *m* of a class *c*	class *d*, d != c	class *d* is the type of a parameter of a method invoked by *m*
5	method *m* of a class *c*	attribute *a* of a class *d*, d != c	*m* references *a*
6	method *m* of a class *c*	method *m'* of a class *d*, d != c	*m* invokes *m'*
7	class *c*	class *d*, d != c	high-level relationships between classes, such as *uses* or *consists-of*

Table 2.2: Types of connections, obtained from [11]

Counting connections	Level	Description
A	Method or attribute	count individual connections
B	Method or attribute	count the number of distinct items at the other end of the connections
C	Class	add up the number of connections counted as in A) for each method or attribute of the class
D	Class	add up the number of connections counted as in B) for each method or attribute of the class
E	Class	count the number of distinct items at the end of connections starting from or ending in methods or attributes of the class
F	Class	for a class c, count the number of other classes to which there is at least one connection

Table 2.3: Counting connections, obtained from [11]

Based on these criteria, Briand et al. classify the existing coupling metrics, according to their definitions [11]. The comparison of all the metrics, according to the criteria of the framework, can be found in Table 2.4. The criteria stability of the server has been excluded from the table since none of the metrics consider it.

Coupling Between Objects (CBO) This metric counts the number of other classes to which the client class coupled. This metric has two definitions: the original definition, CBO' in Table 2.4, which does not count inheritance. Then, there is the revised definition of the metric [17], which does include inheritance.

Response for Class (RFC) This metric calculates the response set of a class. According to Chidamber and Kemerer [17], *"The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class"*. The return set includes the methods called directly by the class, as well as the methods that are called by transitivity. In the framework by Briand et al. [11], it is considered that the inherited methods should be included in this set since they can be executed to respond to a message received in the class. Based on this definition, there are three defined metrics, the first one being RFC_α [18]. The α defines the number of nested levels of transitivity considered in the calculation of the metric. The other two metrics are particular cases of this one: RFC corresponds to when $\alpha = 1$, and RFC' when $\alpha = \infty$.

Message Passing Coupling (MPC) This metric, created by Li and Henry [19], counts invocations from the new methods of a class to methods of other classes. This means that the inherited methods are not considered, but it is unclear how it treats overridden methods or calls to inherited methods. Briand et al. [11], to eliminate ambiguity, redefined the metric as *"the number of static invocations of methods not implemented in c by methods implemented in c"*.

Data Abstraction Coupling (DAC) This metric was also defined by Li and Henry [19] as follows: *"number of ADTs defined in a class"*, where ADT is abstract data type. However, this definition does not specify how the metric should count the connections or consider inherited ADTs. Because of this ambiguity in the original definition of the metric, Briand et al. [11] redefined the metric: *"DAC is the number of not inherited attributes that have a class as their type. The number of the classes used as types for attributes is counted by DAC'"*.

Coupling Factor (COF) COF is the only metric of which the domain of measurement is the entire system and was defined by Abreu et al. [20]. COF calculates the number of relations between classes of the system, which are not related through inheritance. The relations are counted in a binary manner, only counted to how many other classes is a class related, instead of how many times these are connected.

Metric	Inheritance	Locus of impact	Types of connection	Domain of measure	Counting connections	Indirect coupling
CBO	both	both	5, 6	class	F	no
CBO'	no	both	5, 6	class	F	no
RFC_α	both	import	6	class	E	depends
RFC	both	import	6	class	E	no
RFC'	both	import	6	class	E	yes
MPC	both	import	6	class	C	no
DAC	both	import	1	class	C	no
DAC'	both	import	1	class	D	no
COF	no	both	5, 6	system	F	no
ICP	both	import	6	method, class, set	A, C	no
IH-ICP	only	import	6	method, class, set	A, C	no
NIH-ICP	no	import	6	method, class, set	A, C	no
IFCAIC	no	import	1	class	C	no
ACAIC	only	import	1	class	C	no
OCAIC	no	import	1	class	C	no
FCAEC	no	export	1	class	C	no
DCAEC	only	export	1	class	C	no
OCAEC	no	export	1	class	C	no
IFCMIC	no	import	2	class	C	no
ACMIC	only	import	2	class	C	no
OCMIC	no	import	2	class	C	no
FCMEC	no	export	6	class	C	no
DCMEC	only	export	6	class	C	no
OCMEC	no	export	6	class	C	no
OMMIC	no	import	6	class	C	no
IFMMIC	no	import	6	class	C	no
AMMIC	only	import	6	class	C	no
OMMEC	no	export	6	class	C	no
FMMEC	no	export	6	class	C	no
DMMEC	only	export	6	class	C	no

Table 2.4: Coupling metrics comparison

The coupling factor is normalized between 1 and 0 by dividing the number of relations by the system's maximum number of relations possible. This way, it is possible to compare systems of different sizes.

Information-flow-based Coupling (ICP) The original ICP metric counts *"for method m of class c, the number of polymorphically invoked methods of other classes, weighted by the number of parameters of the invoked method."* Sadly, we have not been able to obtain the original paper, but it is described by Briand et al. [11]. From this description, the metrics IH-ICP and NIH-ICP are defined. IH-ICP counts only inheritance-based coupling, whereas NIH-ICP counts coupling to those classes with no inheritance relationship. Finally, the metric ICP is the sum of the previous two.

Suite of metrics by Briand et al. This set of metrics was defined by Briand et al. with their previous framework for coupling metrics [12]. This framework was specially created for C++, and therefore, it has some extensions specific to this language. The metrics of the set are named according to three criteria: relationship, locus, and type of interaction. Each metric's name is composed in the following way: the initials of the relationship, the initials of the type of interaction, and the initials of the locus. These initials are described below.

There are three types of connections, listed below, which can be used to determine the coupling of a class *c*. All the definitions have been obtained from [12].

- Inheritance (A, D): Interactions from a class to its antecessors or descendants, depending on the locus.
- Friendship (F, IF): Extension for C++, interactions from class to all the classes declared as friends or the classes that declare it their friend (inverse friends), depending on the locus.
- Other (O): interaction with classes that do not have an inheritance or friendship relationship.

The three different types of interaction described by Briand et al. [12] are the following:

- Class-Attribute (CA): "There is a class-attribute (CA-) interaction from class c to class d if an attribute of class c is of type class d."
- Class-Method (CM): "There is a class-method (CM-) interaction from class c to class d if a newly defined method of class c has a parameter of type class d."
- Method-Method (MM): "There is a method-method (MM-) interaction from class c to class d, if a method implemented at class c statically invokes a method of class d (newly defined or overriding), or receives a pointer to such a method."

Finally, the two types of locus are:

- Export from a class (EC): "Change flows away from a class" related to the descendants (D) and the friends (F).
- Import to a class (IC): "Change flows towards a class" related to the ancestors (A) and the inverse friends (IF).

The suite of metrics is defined based on all possible combinations of these three criteria.

2.4 Metrics validation

This thesis includes evaluating and validating the metrics included in the proposed model by using the proof-of-concept. Since there is no unique way to validate metrics which is globally accepted and used, various approaches are adopted. In the paper [21], Srinivasan et al. explain that there are two fundamental approaches for metric validation: *theoretically* and *empirically*. Therefore, to provide a check of validity that is as broad as possible, a mixture of these two approaches will be used during this project. However, this research does not contain a full validation of the metrics due to time constraints.

The coupling metrics' theoretical validation is conducted according to the *Mathematical Properties of Measures for Coupling* [21]. Also, a subset of the aspects presented by Meenely et al. [22] are also used to validate all the metrics in the model; in particular, we focus on Actionability and Definition validity.

Chapter 3

Dependency evaluation model

This chapter contains a description of the metrics proposed to measure dependencies from various perspectives: coupling, coverage, and usage per class. Each metric is also validated theoretically by proving the metrics fulfill certain properties that the aspect being measured has.

3.1 Measuring the degree of dependency

Although we have not been able to find any other research which proposes metrics to measure the degree of code dependency in a dependency with a library, this has been previously done to measure the degree of dependency within a library or any other software product. The type of metrics used is coupling metrics. Therefore, we are going to define coupling metrics, which can be used in our use case. To create the metrics, we first need to define the coupling we measure. Then, we define the metrics which measure the type of coupling described. Finally, for the theoretical validation of the metrics, the five properties of coupling established in the literature [23], are proven for each one.

3.1.1 Definition of coupling

To measure the degree of dependency, we investigate the characteristics of the coupling between libraries. Following the framework defined by Briand et al. [11], described in Section 2.3, we define the coupling to measure according to the six criteria in the framework, which were defined based on the characteristics of existing coupling definitions.

Criterion 1 - Type of connection: This criterion defines which type of connection creates coupling between the two items. Several and clearly distinguishable mechanisms can create coupling, as defined by Briand et al. [11], listed below.

Given class a of library A, and class b of library B...

1. ... class a has an attribute of type b (Relationship of aggregation).
2. ... method of class a has a parameter of type b or has return type b.
3. ... method of class a has a local variable of type b.
4. ... method of class a calls a method which has a parameter of type b.
5. ... method of class a references an attribute of class b.
6. ... method of class a invokes a method of class b.
7. ... class a and class b have a relationship such as uses or consists-of.

Having a single metric measure more than one of these types of connections is not recommended for various reasons. To begin with, the strengths of every type of connection have to be justified: Has the coupling created by a local variable (type 5) the same strength as the one created by a method invocation (type 6)? How is the strength quantified? When mixing types of connections, there is information missing; it is impossible to know how much of the coupling is created by which type of

connection, and therefore which fix has a priority. Therefore, all relevant types of connections in the use case of RQ1, measuring the degree of dependencies between libraries, are measured by different metrics.

To decide which types of connections to measure, we reviewed the literature on coupling metrics to understand which connections are the most measured and why. Our findings are summarized in Table 3.1.

Reference	1	2	3	4	5	6	7
[13]	x	x	x	x		x	x
[16]	x	x	x		x	x	x
[12]	x	x				x	
[24]	x	x					
[25]	x	x	x	x		x	
[26]	x				x	x	
[27]	x	x	x	x	x	x	x
[28]					x	x	
[29]	x				x	x	
[30]	x				x	x	

Table 3.1: Literature usage of the types of connection

Types 1 and 6 are the most used in the literature and, in particular, method invocation coupling is hypothesized to be the most relevant type of connection by Briand et al. [11]. Therefore, we define a first metric to measure **type 6: method invocation**.

The second metric that we consider is **type 1: aggregation coupling**, for two key reasons. First, it is used as much as type 6 in the reviewed literature. Besides, in some cases, measuring method invocations may not be enough to understand how much impact a dependency may have on a library. There is the possibility that a class contains a field with the type of another class but never calls a method that belongs to that class.

The above-mentioned types of connections are those that we consider for these metrics, and we will explain them in greater detail in Section 3.1.2. Nevertheless, it might be necessary to include additional metrics in the future, to account for other connection types.We discuss this in the experiment 2 (see Section 5.2).

Criterion 2 - Locus of impact: As explained in Section 2.3, Briand et al. define two options for the locus of impact: import and export [11]. According to the definition of the problem, this measurement aims to know how much a library depends on another, from the point of view of the library that uses another one. Hence, the locus of impact of the coupling to be measured in this thesis is **import**. We measure the dependency from the point of view of the library that acts as a client of a server library.

Criterion 3 - Granularity of the measure: In this criterion, there are two aspects to define. (1) The aggregation level of the measure, and (2) how the metric counts the connections. We first discuss the aggregation level. Briand et al. [11] define the following levels:

- Attribute
- Method
- Class
- Set of classes
- System

The goal is to measure the coupling between the set of classes of the client library and the server library classes. The measurement is done by aggregating the coupling of the more fine-grained levels [11]. For consistency with the terminology used in this thesis, we name this aggregation level **library level**.

Next, we define how the metric has to count connections. The options for counting connections defined by Briand et al. [11] are explained in Table 2.3. The options B), D), E), and F) are not useful for our use case. These options count the distinct items at the other end of the connection, not considering how many times those items are at the other end of the connection.

The two other options are A) and C), which count individual connections. The difference between A and C is the aggregation level at which the connections are counted. Option C) counts the connections as in A), but adding the result for each class's method or attribute. Since the level of the domain of the metrics is not class but library, a new option for counting connections is defined, which would be option G. The definition is created following the same style as Briand et al.: **Add up the number of connections counted as in C) for each class of the library**. By following this method of aggregating the number of connections through aggregation levels, a fine-grained analysis is maintained for the aggregation level of the metrics.

Criterion 4 - Stability of the server: Briand et al. define stable classes as *"Classes that are not subject to change in the project at hand"* [11]. Following this definition, the server of the connection has to be stable in this case. Therefore, we are going to count connections from non-stable elements to **stable servers**. According to the previously defined locus of impact, the non-stable classes are the stable classes' clients.

However, in this thesis, the differentiation between stable and unstable classes is not enough. The goal is to measure coupling only with classes that are part of other libraries. Therefore, the classes that belong to standard libraries and the programming language types, although stable, will not be considered by these metrics.

Criterion 5 - Direct and indirect coupling: To decide whether the metrics count or not indirect coupling, we need to distinguish two alternative scenarios in which we want to measure coupling: Direct dependencies and transitive dependencies. When measuring direct dependencies, we want to measure only direct coupling between the libraries, whereas, for transitive dependencies, it is necessary to measure indirect coupling. Hence, **both types of coupling** will be measured, with two different metrics for each selected type of connection: One for direct dependencies and another for transitive dependencies.

Criterion 6 - Inheritance: There are three aspects to decide within this criterion: how, if at all, does the metric distinguish between inheritance-based coupling and noninheritance-based coupling? If the metric counts method invocations, does it account for polymorphism? Finally, what defines if a method or an attribute is part of a class or not?

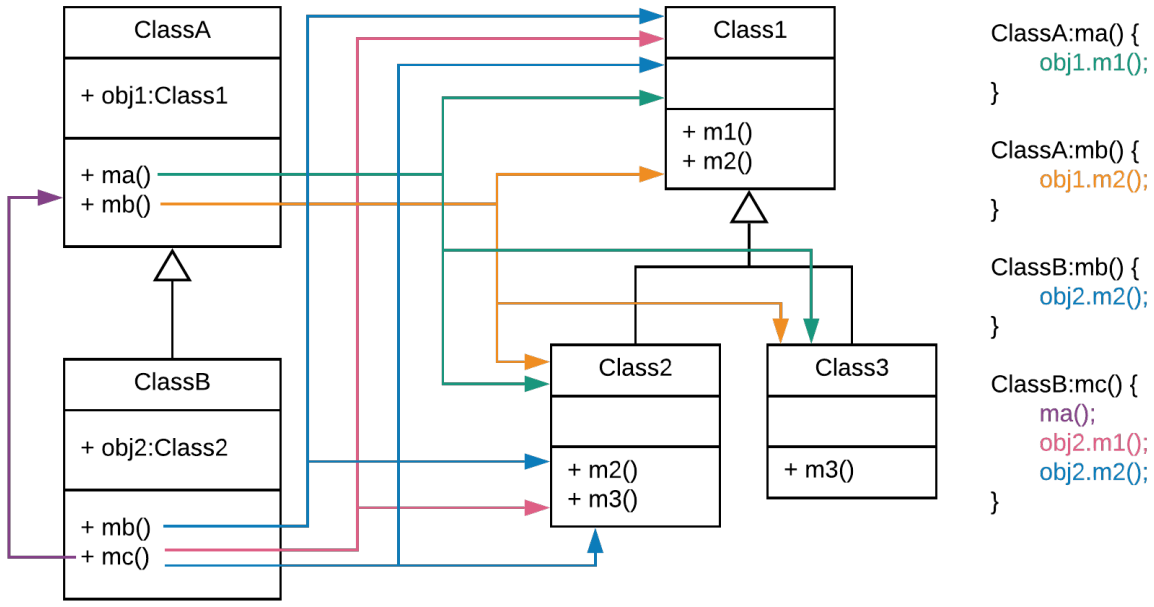

Figure 3.1: Example of coupling special cases, based on example from Briand et al. [11]

In order to answer the first question, we focus on the method `mc` of `ClassB` in Figure 3.1. This method invokes `ma` of `ClassA`, inherited by `ClassB`.

This is known as *inheritance-based* coupling and is sometimes considered as a special case of coupling. When there is a change of an inherited method that a class uses, it requires the same maintenance as the method that is not inherited. Therefore, our metrics **include inheritance-based coupling without distinction**.

In the case of the second question, polymorphism, we look at the methods of `ClassA`. This class contains an attribute of type `Class1`, which considering dynamic assignation of types could also be of type `Class2` or `Class3`.

We first analyze whether a call to a method of `Class1` would create coupling with `Class2` and `Class3`, and if it makes a difference when the method is overridden or not. The method `ma` invokes `m1`, which is not overridden by any of the descendants of `Class1`. When a change is made in `Class2` or `Class3` no change is required as the invoked method remains the same. In contrast, method `mb` calls `m2`, which is overridden in `Class2`. Here, the implementation of `m2` in `Class2` could be updated, and this may affect the way `ClassA` uses it, and therefore changes may be needed. Thus, it is necessary to **account for polymorphism**.

Lastly, we discuss about how to decide whether a method belongs to a class or not. We have two options: (1) a method belongs to the class that implements it (could be more than one since we account for polymorphism), or (2) a method belongs to the class that it is referenced from. An example of this can be found in the last two lines of the method `mc` of `ClassB` call method `m1` and `m2` on an object of type `Class2`. The difference is that `m1` is implemented in `Class1` and `m2` is overridden in `Class2`. From a maintenance perspective, when the method `m1` is updated in `Class1`, this probably requires update in `ClassB` as well. However, changes in `Class2` will not generate a need to update the method call `m1` in `ClassB`. When `m2` is updated in `Class1`, it will not make a difference for the call to `m2` in `ClassB` since it is not executing the implementation of `Class1`. Therefore, **a method call creates coupling with the class that contains the implementation**.

Summary Based on the criteria discussed above, we have four different definitions of coupling. Therefore, we create four metrics, each one measuring one type of coupling. The summary of the definitions of coupling created can be found in Table 3.2.

Metric	Type of connection	Locus of impact	Domain of measure	Counting connections	Direct/Indirect	Inheritance	Polymorphism	Item belongs to class
#1	6	Import	Library	Individual connections	Direct	Both	Yes	Implemented
#2	1	Import	Library	Individual connections	Direct	Both	Yes	Implemented
#3	6	Import	Library	Individual connections	Indirect	Both	Yes	Implemented
#4	1	Import	Library	Individual connections	Indirect	Both	Yes	Implemented

Table 3.2: Criteria of the set of metrics

3.1.2 Metrics for direct dependencies

This section begins with a brief discussion in which the definition of coupling of the proposed metrics is compared with the existing metrics in the literature described in Section 2.3. Next, there is the formal definition of each of our metrics. To end the section, the metrics' theoretical validation is done by proving the five properties of coupling metrics as defined by Briand et al. [23], for each of the new metrics.

Revisiting existing metrics

Once we have defined the coupling that is going to be measured by each of the metrics (see Table 3.2), we can compare it with the existing coupling metrics, according to Table 2.4 in Section 2.3, to decide

whether there are metrics measuring the same coupling we previously defined, or if it is necessary to create new metrics.

Metric #1 To compare the coupling defined for this first metric defined in Table 3.2 with the coupling measured by existing metrics, we focus on the ones that have the following characteristics:

- Type of connection: method invocations (type 6 in Table 2.2)
- Locus of impact: import
- Direct or indirect coupling: direct
- Counting connections: count individual connections (option A or C in Table 2.3).

The metrics that share these characteristics are *MPC*, the group *ICP*, and the metrics *AMMIC*, *IFMMIC* and *OMMIC*. However, *MPC* does not consider polymorphic implementations of the called methods. *IFMMIC* is a metric formulated specifically for C++ [12] and therefore is not useful for our model. The metric *IFMMIC* focuses on method invocations between classes with a *friend* relationship, which does not exist in most languages. Furthermore, in case the language is C++ no distinction should be made between *friend* and *non-friend* classes.

From the group of metrics *ICP*, the metric *ICP* considers both inheritance and non-inheritance coupling and therefore shares the definition of coupling with our metric. However, according to the definition of *ICP*, the coupling created by each method call is weighted by the number of parameters of the called method. In this aspect, it differs from metric #1 from Table 3.2.

Finally, the metrics *AMMIC* and *OMMIC* use the same definition of coupling as metric #1 except that *AMMIC* counts the method invocations to ancestors and *OMMIC* to other classes. Therefore, metric #1 is the sum of *AMMIC* and *OMMIC*, but aggregated at the library level instead of the class level.

Metric #2 In the case of the second metric, which measures the coupling as defined in Table 3.2, we focus on the metrics with the following characteristics:

- Type of connection: client class contains an attribute of type server class, aggregation coupling (type 1 in Table 2.2)
- Locus of impact: import
- Direct or indirect coupling: direct
- Counting connections: count individual connections (option A and C in Table 2.3, measure individual connections at the method and class level respectively)

According to Table 2.4, the metrics that share these characteristics are *DAC*, and from the suite of metrics by Briand et al. [12] the metrics *IFCAIC*, *ACAIC* and *OCAIC*. However, *IFCAIC* is an extension for C++ and will not be considered, for the same reasons for which we discarded *IFMMIC* for metric #1.

According to the definition of *DAC*, it counts the number of attributes of a class that have any other class as type. Therefore, instead of calculating the coupling between two classes, it calculates a class's coupling with every other class. However, metric #2 is used at the library level and calculates coupling between two libraries instead of the coupling of one library with all the others.

Finally, *ACAIC* and *OCAIC* consider aggregation coupling with ancestors and others respectively. Therefore, metric #2 is the sum of these two metrics but aggregated to the library level since the metrics are designed for class level.

Formal definitions

Metric #1: Direct method invocation coupling (`MIC`) The `MIC` metric measures the dependency between two libraries, one acting as a client (L_c) and the other as a server (L_s). Based on the granularity of the measure criterion discussed in Section 3.1.1, this metric is calculated for each of the classes implemented in L_c, and for each of the methods implemented $\texttt{M}(L_c)$ in these classes. For each implemented method $m_c \in \texttt{M}(L_c)$, we count the number of individual invocations to a method of L_s, denoted $\texttt{nII}(m_c, L_s)$. For each method invocation made by the methods implemented in L_c, we count

only the ones implemented in stable classes (not implemented in L_c). The set of stable methods invoked is denoted $\texttt{SIM}(m_c)$.

$$\texttt{MIC}(L_c, L_s) = \sum_{m_c \in \texttt{M}(L_c)} \texttt{nII}(m_c, L_s) \tag{3.1}$$

According to the criterion inheritance, it is necessary to consider all the polymorphic implementations of the invoked method that are implemented in L_s. Therefore, we intersect the set of polymorphic implementations of an invoked method $\texttt{PM}(m_s)$ with the set of methods $\texttt{M}(L_s)$ implemented in L_s. Finally, to obtain the number of individual invocations, $\texttt{nII}(m_c, L_s)$, we multiply the number of times a stable method ($m_s \in \texttt{SIM}(m_c)$) has been invoked, $\texttt{nI}(m_c, m_s)$ by the number of polymorphic implementations $\texttt{nP}(m_s, L_s)$ of the method in L_s.

$$\texttt{nII}(m_c, L_s) = \sum_{m_s \in \texttt{SIM}(m_c)} \texttt{nI}(m_c, m_s) * \texttt{nP}(m_s, L_s) \tag{3.2}$$

$$\texttt{nP}(m_s, L_s) = |\texttt{PM}(m_s) \cap \texttt{M}(L_s)| \tag{3.3}$$

Metric #2: Direct aggregation coupling (AC) The AC metric counts the number of times when a class of L_c has an attribute whose type is a class implemented in L_s. Therefore, the metric is calculated for each class implemented in L_c ($c_c \in \texttt{C}(L_c)$). We consider only those attributes types that are stable classes (not implemented in L_c) for each class c_c. The set of stable attribute types in a class c is $\texttt{SAT}(c_c)$.

To account for polymorphism (criterion *inheritance*), we count all the descendants of the class that are implemented in L_s. Therefore, we intersect the set of the descendants of the class, $\texttt{DC}(c_s)$, with the set of classes implemented in L_s ($\texttt{C}(L_s)$). Finally, to count the individual connections, we multiply the number of times a client class c_c has an attribute of type the server class c_s ($\texttt{NA}(c_c, c_s)$) by the number of class descendants (class included) implemented in L_s ($\texttt{nDC}(c_s, L_s)$).

$$\texttt{AC}(L_c, L_s) = \sum_{c_c \in \texttt{C}(L_c)} \sum_{c_s \in \texttt{SAT}(c_c)} \texttt{NA}(c_c, c_s) * \texttt{nDC}(c_s, L_s) \tag{3.4}$$

$$\texttt{nDC}(c_s, L_s) = |\texttt{DC}(c_s) \cap \texttt{C}(L_s)| \tag{3.5}$$

Theoretical validation

The theoretical validation of the metrics consists of demonstrating the properties of the metrics. Theoretical validation is necessary since it proves that the metrics share properties with the attribute measured; in this case, the attribute is coupling. In particular, for coupling metrics, there are five properties defined by Briand et al. [23], which have been largely used by literature [10, 15, 31]. First, we describe each of the properties, and then we prove them for each of the metrics.

1. **Nonnegativity:** The value of the coupling metrics should never be negative.
2. **Null value:** The value of the coupling is expected to be zero if there is no relationship from the client library to the server library.
3. **Monotonicity:** It is expected that if more relationships are added from the client library to the server library, the metric value does not decrease.
4. **Merging of classes:** If two classes of the client library are merged, the total coupling between the client library and the server library should not increase.
5. **Merging of unconnected classes:** When two classes of the client library, which do not share usage of the server library, are merged, the total coupling between the client library and the server library should remain equal.

To describe the properties in greater detail, we use $\texttt{Coupling}(L_c, L_s)$ to refer to both AC and MIC, and $\texttt{R}(L_c, L_s)$ to refer to the set of relationships between L_c and L_s, $\texttt{Coupling}(L_c, L_s)$ uses $\texttt{R}(L_c, L_s)$ to evaluate the coupling between the two elements, but the way it is used differs per metric. To refer to the relations between a class of the client library, c_c, and the server library, we use $\texttt{R}(c_c, L_s)$, and the coupling between the class and the server library is $\texttt{Coupling}(c_c, L_s)$. The description of the properties is based on the description done by Briand et al. [23], which was meant for coupling metrics that measure the

coupling within an element, or between an element and all the other elements. Therefore, the properties' description has been adapted for metrics that measure coupling between two different elements. Also, since all the newly introduced metrics measure import coupling, the properties' definitions are focused on this locus of impact.

Nonnegativity Let L_c be a client library and L_s be a server library. The coupling between the two libraries is non-negative, $\texttt{Coupling}(L_c, L_s) \geq 0$.

Null value Coupling is expected to be null (zero) when there is no import relationship between the client and the server libraries.

Let L_c be a client library and L_s be a server library. The coupling between the two libraries is null if the set of import relationships from L_c to L_s, $\texttt{R}(L_c, L_s)$, is empty. Therefore, $\texttt{R}(L_c, L_s) = \emptyset \implies \texttt{Coupling}(L_c, L_s) = 0$.

Monotonicity Considering the definition of coupling, it is expected that when more relationships are added between the libraries, coupling does not decrease.

Let L_c be a client library, L_s be a server library, and $c \in L_c$ be a class in L_c. We modify class c to form a new class c' which is identical to c except that $\texttt{R}(L_c, L_s) \subseteq \texttt{R}(L'_c, L_s)$. For example, some method invocations have been added from c to classes implemented in L_s. Let L'_c be a library identical to L_c but in which c has been replaced by c'. Then, $\texttt{Coupling}(L_c, L_s) \leq \texttt{Coupling}(L'_c, L_s)$.

Merging of classes The original definition of the property is created for metrics that measure coupling within a system, and therefore it is necessary to reformulate it. It is expected that if two classes of a system are merged, the system's coupling does not increase. If two classes are merged, the coupling between the two classes is subtracted from the system's total coupling.

When considering the coupling between a client library and a server library, if two client library classes are merged, the two libraries' coupling would not increase. It could decrease, depending on how the refactoring is performed. If the classes share usage of the server library, one of the usages may be removed.

Therefore, let L_c be a client library, L_s be a server library, and $c_1, c_2 \in L_c$ two classes in L_c. Let c' be the class that results from merging c_1 and c_2, and L'_c be the library resulting from L_c when c_1 and c_2 have been replaced by c'. Then, $\texttt{Coupling}(c_1, L_s) + \texttt{Coupling}(c_2, L_s) \geq \texttt{Coupling}(c', L_s)$ and $\texttt{Coupling}(L_c, L_s) \geq \texttt{Coupling}(L'_c, L_s)$.

Merging of unconnected classes This property is a variation of the previous one, and it has to be adapted to the use case of this thesis. It is expected that the system's coupling will stay the same if two classes of a system, which have no relationship, are merged. This is because the class that results in the merging will have the same number of relationships with other classes as the original two.

When measuring the coupling between a client library and a server library, we define two unconnected classes as classes that do not share usage of the server library. Therefore, none of the relationships with the server library can be merged when merging the two classes, and the coupling with the server library stays the same.

Let L_c be a client library, L_s a server library, and $c_1, c_2 \in L_c$ two classes from L_c which do not share the same relationship with L_s. Let c' be the class that is the union of c_1 and c_2, and L'_c be the library identical to L_c but in which c_1 and c_2 have been replaced by c'. If there are no relationships between c_1 and c_2, then, $\texttt{Coupling}(c_1, L_s) + \texttt{Coupling}(c_2, L_s) = \texttt{Coupling}(c', L_s)$ and $\texttt{Coupling}(L_c, L_s) = \texttt{Coupling}(L'_c, L_s)$.

Theoretical validation: MIC

Nonnegativity If we assume that the metric `MIC` does not fulfill the property Nonnegativity, there should be a client library L_s and a server library L_c such that $\texttt{MIC}(L_c, L_s) < 0$. According to the equation 3.1, this means that exists at least one client method $m_c \in \texttt{M}(L_c)$ such that $\texttt{nII}(m_c, L_c) < 0$. Following the equation of `nII` 3.2, this opens two possibilities.

First, that there is a server method $m_s \in \texttt{SIM}(m_c)$ such that $\texttt{nI}(m_c, m_s) < 0$. However, m_s is a method out of the set $\texttt{SIM}(m_c)$ which is the set of stable methods invoked by m_c, which means that $\texttt{nI}(m_c, m_s) > 0$ for all $m_s \in \texttt{SIM}(m_c)$, therefore it is a contradiction.

The other option is that there is a method $m_s \in \texttt{SIM}(m_c)$ such that $\texttt{nP}(m_s, L_s) < 0$. $\texttt{nP}(m_s, L_s)$, according to the equation 3.3, corresponds to the cardinality of the intersection between the set $\texttt{PM}(m_s)$

and $\texttt{M}(L_s)$. Therefore, the cardinality of the intersection has to be less than zero. However, the cardinality of the intersection is by definition greater or equal to zero. This constitutes a contradiction.

Therefore, the initial assumption is not true, and *Nonnegativity* holds for the metric MIC.

Null value Assuming there is no null value for metric MIC, there is a client library L_c and a server library L_s such that $\texttt{R}(L_c, L_s) = \emptyset$, and $\texttt{MIC}(L_c, L_s) \neq 0$. As non-negativity holds, we have that $\texttt{MIC}(L_c, L_s) \geq 0$. Therefore, $\texttt{MIC}(L_c, L_s) > 0$. Hence, following equation 3.1 there is a client method $m_c \in M(L_c)$ such that $\texttt{nII}(m_c, L_s) > 0$.

Thus, according to equation 3.2, there is a server method $m_s \in \texttt{SIM}(m_c)$, such that $\texttt{nI}(m_c, m_s) > 0$ and $\texttt{nP}(m_s, L_s) > 0$. Therefore, the method m_s is called at least one time by the method m_c from the client library L_c, and at the same time is implemented by the server library L_s, which means that there is a relationship between L_c and L_s, which contradicts the original assumption that $\texttt{R}(L_c, L_s) = 0$.

Consequently, there is a *null value* for metric MIC.

Monotonicity Let L_c be a client library that contains class c_c, and let c'_c be a class resulting from adding relationships with the server library L_s to the class c_c. Then, $\texttt{R}(c_c, L_s) \subseteq \texttt{R}(c'_c, L_s)$. Let L'_c be a client library identical to L_c but in which the class c_c has been replaced by c'_c. Therefore, $\texttt{R}(L_c, L_s) \subseteq \texttt{R}(L'_c, L_s)$.

Let's assume that the MIC metric does not fulfill the property monotonicity, this would mean that $\texttt{MIC}(L_c, L_s) > \texttt{MIC}(L'_c, L_s)$. Since the only difference between L_c and L'_c is the substitution of c_c by c'_c, then $\sum_{m_c \in \texttt{M}(c_c)} \texttt{nII}(m_c, L_s) > \sum_{m'_c \in \texttt{M}(c'_c)} \texttt{nII}(m'_c, L_s)$ (see equation 3.1). Therefore, the methods of class c_c have more individual invocations to L_s than the methods from class c'_c. This contradicts the initial assumption that $\texttt{R}(c_c, L_s) \subseteq \texttt{R}(c'_c, L_s)$.

Therefore, *Monotonicity* holds for the metric MIC.

Merging of classes Let L_c be a client library that includes the classes c_1 and c_2. Let c' be a class such that $c_1 + c_2 = c'$ and L'_c be a client library identical to L_c but where c_1 and c_2 have been replaced by c'. If we assume that the property merging of classes does not hold for metric MIC, it would mean that $\texttt{R}(c_1, L_s) \subseteq \texttt{R}(c', L_s) \wedge \texttt{R}(c_2, L_s) \subseteq \texttt{R}(c', L_s)$ and at the same time $\texttt{MIC}(L_c, L_s) > \texttt{MIC}(L'_c, L_s)$.

Therefore, there is a method m_c which is implemented in c_1 or c_2 such that contains a call to a method m_s which does not exist in any of the methods implemented in c'. This is a contradition with the initial affirmation $\texttt{R}(c_1, L_s) \subseteq \texttt{R}(c', L_s) \wedge \texttt{R}(c_2, L_s) \subseteq \texttt{R}(c', L_s)$. Therefore, the property *Merging of classes* holds for metric MIC.

Merging of unconnected classes Let L_c be a client library and L_s be a server library. Let c_1 and c_2 be classes implemented in L_c, such that $\texttt{R}(c_1, L_s) \cap \texttt{R}(c_2, L_s) = \emptyset$. Let c' be a class such that $c_1 + c_2 = c'$. Therefore, $\texttt{R}(c_1, L_s) + \texttt{R}(c_2, L_s) = \texttt{R}(c', L_s)$. Let L'_c be a client library identical to L_c but in which c_1 and c_2 have been replaced by c'. We assume that the metric MIC does not fulfill this property.

Therefore, $\texttt{MIC}(L_c, L_s) \neq \texttt{MIC}(L'_c, L_s)$. According to property *Merging of classes*, $\texttt{MIC}(L_c, L_s)$ cannot be less than $\texttt{MIC}(L'_c, L_s)$. Thus, $\texttt{MIC}(L_c, L_s) > \texttt{MIC}(L'_c, L_s)$. This means that there is a m_c implemented in c_1 or c_2 that contains an invocation to a method m_s implemented in L_s, which is not included in c'. This contradicts that $\texttt{R}(c_1, L_s) + \texttt{R}(c_2, L_s) = \texttt{R}(c', L_s)$.

Therefore, property *Merging of unconnected classes* holds for metric MIC.

Theoretical validation: AC

Nonnegativity Supose that the metric AC does not have the nonnegativity property. Thus, there is a client library L_c and a server library L_s such that $\texttt{AC}(L_c, L_s) < 0$. Then, according to equation 3.4 there is a client class $c_c \in \texttt{C}(L_c)$ and a server class $c_s \in \texttt{SAT}(c_c)$ such that either $\texttt{NA}(c_c, c_s)$ or $\texttt{nDC}(c_s, L_s)$ have a negative value.

Let's assume that $\texttt{NA}(c_c, c_s) < 0$. This means that the c_s is a class that is included in the set of stable classes declared as fields in c_c ($c_s \in \texttt{SAT}(c_c)$) and, at the same time is declared a negative number of times, which is a contradiction.

Therefore, $\texttt{nDC}(c_s, L_s)$ has to be negative. According to equation 3.5, $\texttt{nDC}(c_s, L_s)$ corresponds to the cardinality of the intersection between two sets. Even if the two sets do not share any element, by definition, the intersection will be the empty set, and the cardinality will be zero. Hence, $\texttt{nDC}(c_s, L_s)$ cannot have a negative value, and the initial assumption is false.

In conclusion, *Nonnegativity* holds for the metric AC.

Null value If we assume that property null value does not hold for metric AC, there has to be a client library L_c and a server library L_s such that have no relationship ($\mathtt{R}(L_c, L_s) = 0$) and $\mathtt{AC}(L_c, L_s) \neq 0$. Since AC has the property *Nonnegativity*, the result cannot be negative, which means that $\mathtt{AC}(L_c, L_s) > 0$. Hence, following equation 3.4, there is a client class $c_c \in \mathtt{C}(L_c)$ and a server class $c_s \in \mathtt{SAT}(c_c)$ such that $\mathtt{NA}(c_c, c_s) > 0$ and $\mathtt{nDC}(c_s, L_s) > 0$. This means that the class c_s is at the same time declared at least once by the client class c_c ($\mathtt{NA}(c_c, c_s) > 0$) and implemented in the server library L_s. However, this would create a relationship between L_c and L_s, which contradicts the initial assumption.

Therefore, the property *Null value* holds for metric AC.

Monotonicity Let L_c be a client library that contains class c_c, and let c'_c be a class identical to c_c but more relationships with the server library L_s. Then, $\mathtt{R}(c_c, L_s) \subseteq \mathtt{R}(c'_c, L_s)$. Let L'_c be a client library identical to L_c but in which the class c_c has been replaced by c'_c. Therefore, $\mathtt{R}(L_c, L_s) \subseteq \mathtt{R}(L'_c, L_s)$.

If we assume that the metric AC does not fulfill this property, means that $\mathtt{AC}(L_c, L_s) > \mathtt{AC}(L'_c, L_s)$. The only difference between these two calculations is the result of the calculation for c_c and c'_c. Therefore, $\sum_{c_s \in \mathtt{SAT}(c_c)} \mathtt{NA}(c_c, c_s) * \mathtt{nDC}(c_s, L_s) > \sum_{c_s \in \mathtt{SAT}(c'_c)} \mathtt{NA}(c'_c, c_s) * \mathtt{nDC}(c_s, L_s)$, see equation 3.4.

This means that there is a server class $c_s \in \mathtt{SAT}(c_c)$, that is implemented in L_s ($\mathtt{nDC}(c_s, L_s) > 0$) such that $\mathtt{NA}(c_c, c_s) > \mathtt{NA}(c'_c, c_s)$. This contradicts the original assumption that c'_c is constructed from c_c but with additional relationships with L_s. $\mathtt{NA}(c_c, c_s)$ will only be greater than $\mathtt{NA}(c'_c, c_s)$ if there is an attribute of type c_s in c_c (which is a relationship between c_c and L_s) that does not exist in c'_c.

Therefore, *Monotonicity* holds for the metric AC.

Merging of classes Let L_c be a client library that includes the classes c_1 and c_2. Let c' be a class such that $c_1 + c_2 = c'$ and L'_c be a client library identical to L_c but where c_1 and c_2 have been replaced by c'. We assume that the property merging of classes does not hold for metric AC. Therefore, $\mathtt{R}(c_1, L_s) \subseteq \mathtt{R}(c', L_s) \wedge \mathtt{R}(c_2, L_s) \subseteq \mathtt{R}(c', L_s)$ and, also $\mathtt{AC}(L_c, L_s) > \mathtt{AC}(L'_c, L_s)$.

Thus, either c_1 or c_2 contain an attribute of type c_s, such that c_s is implemented in L_s and it is not included c'. This creates a contradition with the initial affirmation $\mathtt{R}(c_1, L_s) \subseteq \mathtt{R}(c', L_s) \wedge \mathtt{R}(c_2, L_s) \subseteq \mathtt{R}(c', L_s)$, since the declaration of an attribute of type c_s is included in $\mathtt{R}(c_1, L_s)$ or $\mathtt{R}(c_2, L_s)$. Therefore, *Merging of classes* holds for metric AC.

Merging of unconnected classes Let L_c be a client library and L_s be a server library. Let c_1 and c_2 be classes implemented in L_c, such that $\mathtt{R}(c_1, L_s) \cap \mathtt{R}(c_2, L_s) = \emptyset$. Let c' be a class such that $c_1 + c_2 = c'$. Therefore, $\mathtt{R}(c_1, L_s) + \mathtt{R}(c_2, L_s) = \mathtt{R}(c', L_s)$. Let L'_c be a client library identical to L_c but in which c_1 and c_2 have been replaced by c'. We assume that the metric AC does not fulfill property *Merging of unconnected classes*.

Therefore, $\mathtt{AC}(L_c, L_s) \neq \mathtt{AC}(L'_c, L_s)$. According to property *Merging of classes*, it cannot happen that $\mathtt{AC}(L_c, L_s) < \mathtt{AC}(L'_c, L_s)$. Therefore, $\mathtt{AC}(L_c, L_s) > \mathtt{AC}(L'_c, L_s)$. The only way this is if there is an attribute of type c_s declared in c_1 or c_2 and implemented in L_s, such that is not included in c'. This creates a contradition with the initial affirmation that $\mathtt{R}(c_1, L_s) + \mathtt{R}(c_2, L_s) = \mathtt{R}(c', L_s)$.

Therefore, metric AC fulfills *Merging of unconnected classes*.

3.1.3 Metrics for transitive dependencies

In this section, the metrics to measure transitive dependencies are described. First, the characteristics of the metrics, according to the criteria previously discussed and summarized in Table 3.2, are compared to the existing metrics described in section 2.3. Next, some concepts involved in the formulation of the transitive metrics are explained. Then, there is a formal definition of the two metrics for transitive dependencies. Finally, the five properties of coupling metrics are demonstrated.

Revisiting existing metrics

In the set of metrics reviewed by Briand et al. [11], there is only one metric that does count indirect coupling, RFC' (see Table 2.4). This metric also counts inheritance-based coupling, and it is focused on the client element, just as the metrics #3 and #4 defined in Table 3.2. However, RFC' is calculated at the class aggregation level, whereas the metrics for this work are calculated at the library level. Furthermore, the strategy to count connections is E, which means that it counts the number of elements with which the class has a connection, not how many connections.

Therefore, both metrics #3 and #4 are entirely unrelated to those reviewed by Briand et al. [11].

Concepts related to transitive dependencies

Other factors have to be taken into account to define the coupling for transitive metrics, which are not needed for the direct metrics.

Reachability To measure the transitive dependencies, only those methods or classes of the transitive dependencies that are *reachable* from the analysed client library are considered. For a given call-graph, a method is reachable if there is a path from the client library to the method [32]. For example, in Figure 3.2, there is no path from `Lib1` (the client library) to the method `Method10`. Therefore, `Method10` is not reachable form `Lib1` and the call from `Method6` to `Method10` will not be considered when measuring the transitive dependency between `Lib1` and `Lib3`, in the case of method invocation coupling.

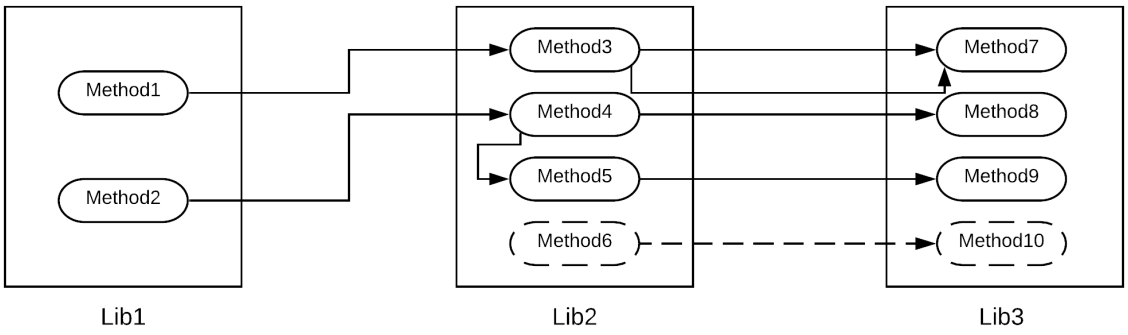

Figure 3.2: Reachability example

Propagation Factor The propagation represents how the impact of a change can spread across dependencies [9]. In Figure 3.2, a change in `Method3` would affect directly the client library (`Lib1`). However, a change in `Method7`, affects first `Method3`, and then it can spread to `Method1`, in case it is not mitigated in `Method3`. This possible mitigation is accounted for by the *Propagation Factor*.

Formal definition

Metric #3: Transitive method invocation coupling (`TMIC`) If we look at how the metric `MIC` is calculated, it could be summarized as follows: find all the methods in L_s that are reachable from L_c. Then, for each one, count how many method calls exist in L_c that reach this method, and sum up the results. The main difference between `MIC` and `TMIC` is that the calls in L_c will not directly execute a reachable method of L_s. The execution of a method in L_s is indirect since L_s is not a direct dependency of L_c.

Therefore, it is necessary to take into account the distance between L_c and L_s. In addition, it could happen that L_s is reachable from L_c at different distances. For instance, if L_s appeared twice in the dependency tree of L_c, this is is the case in Figure 3.3 if we take `Lib1` as L_c and `Lib4` as L_s.

Therefore, the coupling will be measured for a certain `distance`, denoted $\texttt{TMICD}(L_c, L_s, \texttt{distance})$. The value of the metric $\texttt{TMIC}(L_c, L_s)$, will be measured as follows. For each `distance` at which there is coupling between L_c and L_s, sum up the coupling measured by $\texttt{TMICD}(L_c, L_s, \texttt{distance})$, multiplied by a propagation factor (`PF`) to the power of the $\texttt{distance} - 1$, where $\texttt{PF} \in (0, 1)$. We have designed the formula by taking the propagation factor to the power of $\texttt{distance} - 1$, because this way, the coupling of the direct dependencies ($\texttt{distance} = 1$) is not mitigated. Also, then when $\texttt{distance} = 2$, which corresponds to the first level of transitivity, the coupling is mitigated only once.

$$\texttt{TMIC}(L_c, L_s) = \sum_{\texttt{distance}} \texttt{TMICD}(L_c, L_s, \texttt{distance}) * \texttt{PF}^{\texttt{distance}-1} \tag{3.6}$$

The transitive coupling between two libraries at a certain distance $\texttt{TMICD}(L_c, L_s, \texttt{distance})$ is calculated in the following manner. For each method from L_s that is reachable from L_c throught method calls at `distance` ($rm \in \texttt{RM}(L_c, L_s, \texttt{distance})$), we count the number of method invocations in L_c from

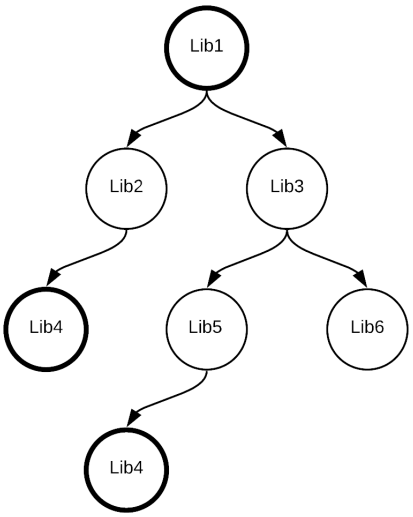

Figure 3.3: Example dependency tree

which rm is reachable, $\texttt{nIR}(rm, L_c)$. The number of method invocations is multiplied by the number of polymorphic implementations of rm in L_s ($\texttt{nDC}(rm, L_s)$).

$$\texttt{TMICD}(L_c, L_s, \texttt{distance}) = \sum_{rm \in \texttt{RM}(L_c, L_s, \texttt{distance})} \texttt{nIR}(rm, L_c) * \texttt{nP}(rm, L_s) \tag{3.7}$$

Metric #4: Transitive aggregation coupling (TAC) To calculate TAC, just as in the case of TMIC, the distance between L_c and L_s should be considered. Therefore, for each distance, we take the number measured by $\texttt{TACD}(L_c, L_s, \texttt{distance})$, and multiply it by a propagation factor PF to the power of the $\texttt{distance} - 1$.

$$\texttt{TAC}(L_c, L_s) = \sum_{\texttt{distance}} \texttt{TACD}(L_c, L_s, \texttt{distance}) * \texttt{PF}^{\texttt{distance}-1} \tag{3.8}$$

The transitive aggregation coupling per distance ($\texttt{TACD}(L_c, L_s, \texttt{distance})$), is calculated in the following way. For each class of L_s that is reachable from L_c through field declarations at distance ($rc \in \texttt{RC}(L_c, L_s, \texttt{distance})$), count all the field declarations from which it is reachable ($\texttt{nFR}(rc, L_c)$), and multiply it by the number of descendants of the reachable class ($\texttt{nDC}(rm, L_s)$).

$$\texttt{TACD}(L_c, L_s, \texttt{distance}) = \sum_{rc \in \texttt{RC}(L_c, L_s, \texttt{distance})} \texttt{nFR}(rc, L_c) * \texttt{nDC}(rm, L_s) \tag{3.9}$$

Theoretical validation: TMIC

Nonnegativity Assume that nonnegativity does not hold for metric TMIC. Then, there exists a client library L_c, and a server library L_s such that $\texttt{TMIC}(L_c, L_s) < 0$. According to equation 3.6, there is a distance for which either $\texttt{TMICD}(L_c, L_s, \texttt{distance}) < 0$ or $\texttt{PF}^{\texttt{distance - 1}} < 0$. Since distance is a positive integer, and $\texttt{PF} \in (0, 1)$, the second option is not possible.

Let us assume that $\texttt{TMICD}(L_c, L_s, \texttt{distance}) < 0$. Looking at the equation 3.7, we see that there has to be at least one method, rm, from L_s and reachable from L_c at a certain distance ($rm \in \texttt{RM}(L_c, L_s, \texttt{distance})$), such that $\texttt{nIR}(rm, L_c) < 0$ or $\texttt{nP}(rm, L_s) < 0$. Since $\texttt{nP}(rm, L_s)$ corresponds to the number of polymorphic implementations of rm in L_s, and we know that rm belongs to L_s, then $\texttt{nP}(rm, L_s) \geq 1$. Finally, for $\texttt{nIR}(rm, L_c) < 0$ to be true, there should be less than zero method invocations in L_c from which rm is reachable. However, since $rm \in \texttt{RM}(L_c, L_s, \texttt{distance})$, and $\texttt{RM}(L_c, L_s, \texttt{distance})$ corresponds to the set of methods from L_s that are reachable from L_c, we have that $\texttt{nIR}(rm, L_c) \geq 1$, which constitutes a contradition.

Therefore, the metric TMIC fulfills the property *Nonnegativity*.

Null value Assuming that TMIC does not fulfill property *Null value*, there exists a client library L_c, and a server library L_s such that $\texttt{R}(L_c, L_s) = \emptyset$ and $\texttt{TMIC}(L_c, L_s) \neq 0$.

For $\texttt{TMIC}(L_c, L_s) \neq 0$ to be true, according to equation 3.7, there has to be a method rm such that $rm \in \texttt{RM}(L_c, L_s, \texttt{distance})$. However, that would mean that rm belongs to L_s, and is reachable from L_c, which constitutes a relation between L_c and L_s, and contradicts $\texttt{R}(L_c, L_s) = \emptyset$.

Hence, the property *null value* holds for metric TMIC .

Monotonicity Let L_c be a client library containing class c_c, and c'_c be a class resulting from adding relationships with the server library L_s to class c_c. Then, $\texttt{R}(c_c, L_s) \subseteq \texttt{R}(c'_c, L_s)$. Let L'_c be a client library resulting from replacing class c_c by c'_c in L_c. Therefore, $\texttt{R}(L_c, L_s) \subseteq \texttt{R}(L'_c, L_s)$. If we assume that TMIC does not fulfill property *Monotonicity*, it would be true that $\texttt{TMIC}(L_c, L_s) > \texttt{TMIC}(L'_c, L_s)$. Therefore, for a certain distance, we have that $\texttt{TMICD}(L_c, L_s, \texttt{distance}) > \texttt{TMICD}(L'_c, L_s, \texttt{distance})$, according to equation 3.6. According to the equation 3.7, this opens two possibilities.

First, we have that $|\texttt{RM}(L_c, L_s, \texttt{distance})| > |\texttt{RM}(L'_c, L_s, \texttt{distance})|$, which means that there are more methods from L_s reachable from L_c than from L'_c. This is not possible since, the only difference between L_c and L'_c is the substitution of class c_c by class c'_c, which only adds relationships with L_s. The second option is that for a certain $rm \in \texttt{RM}(L_c, L_s, \texttt{distance})$, and therefore also $rm \in \texttt{RM}(L'_c, L_s, \texttt{distance})$, such that $\texttt{nIR}(rm, L_c) > \texttt{nIR}(rm, L'_c)$. However, that means that in L_c there are more method invocations that reach rm, than in L'_c. As discussed earlier, this constitutes a contradicion with the way L'_c is created.

Therefore, property 3 *Monotonicity* holds for TMIC.

Merging of classes Let L_c be a client library that includes the classes c_1 and c_2. Let c' be a class such that $c_1 + c_2 = c'$ and L'_c be a client library resulting from replacing c_1 and c_2 by c' in L_c. If we assume that the property merging of classes does not hold for TMIC, then $\texttt{R}(c_1, L_s) \subseteq \texttt{R}(c', L_s) \wedge \texttt{R}(c_2, L_s) \subseteq \texttt{R}(c', L_s)$ and $\texttt{TMIC}(L_c, L_s) > \texttt{TMIC}(L'_c, L_s)$.

Since the only difference between L_c and L'_c is the replacement of c_1 and c_2 by c', there has to be a method invocation from c_1 or c_2 to a method $rm \in \texttt{RM}(L_c, L_s, \texttt{distance})$, which is not included in c'. However, the method invocation has to be a relation included in $\texttt{R}(c_1, L_s)$ or $\texttt{R}(c_2, L_s)$, and we have that $\texttt{R}(c_1, L_s) \subseteq \texttt{R}(c', L_s) \wedge \texttt{R}(c_2, L_s) \subseteq \texttt{R}(c', L_s)$. Therefore it is a contradiction.

Hence, the property *Merging of classes* holds for TMIC.

Merging of unconnected classes Let L_c be a client library and L_s be a server library. Let c_1 and c_2 be classes implemented in L_c, such that $\texttt{R}(c_1, L_s) \cap \texttt{R}(c_2, L_s) = \emptyset$. Let c' be a class such that $c_1 + c_2 = c'$. Therefore, $\texttt{R}(c_1, L_s) + \texttt{R}(c_2, L_s) = \texttt{R}(c', L_s)$. Let L'_c be a client library identical to L_c but in which c_1 and c_2 have been replaced by c'. We assume that the metric TMIC does not fulfill this property.

Therefore, $\texttt{TMIC}(L_c, L_s) \neq \texttt{TMIC}(L'_c, L_s)$. According to property *Merging of classes*, $\texttt{TMIC}(L_c, L_s) \geq \texttt{TMIC}(L'_c, L_s)$. Hence, $\texttt{TMIC}(L_c, L_s) > \texttt{TMIC}(L'_c, L_s)$.

Then, there is a method invocation is c_1 or c_2 which is not included in c, which contradicts that $\texttt{R}(c_1, L_s) + \texttt{R}(c_2, L_s) = \texttt{R}(c', L_s)$.

We conclude that TMIC fulfills property *Merging of unconnected classes*.

Theoretical validation: TAC

Nonnegativity Assuming that nonnegativity does not hold for metric TAC, there exists a client library L_c, and a server library L_s such that $\texttt{TAC}(L_c, L_s) < 0$. In line with equation 3.8, there is a distance for which two things can happen. First, $\texttt{PF}^{\texttt{distance}-1} < 0$. However, since distance is a positive integer, and $\texttt{PF} \in (0, 1)$, this is not possible.

The second option is that $\texttt{TACD}(L_c, L_s, \texttt{distance}) < 0$. Looking at the equation 3.9, we see that there has to be at least one class, $rc \in \texttt{RC}(L_c, L_s, \texttt{distance})$, that belongs to L_s and is reachable from L_c, such that $\texttt{nFR}(rc, L_c) < 0$ or $\texttt{nDC}(rm, L_s) < 0$. $\texttt{nDC}(rm, L_s)$ is the number of descendants of rc in L_s. As we know that rc belongs to L_s, we have that $\texttt{nDC}(rm, L_s) \geq 1$.

Finally, if $\texttt{nFR}(rc, L_c) < 0$ is true, there are less than zero field declarations in L_c from which rc is reachable. However, since $rc \in \texttt{RC}(L_c, L_s, \texttt{distance})$, and $\texttt{RC}(L_c, L_s, \texttt{distance})$ corresponds to the set of classes from L_s that are reachable from L_c through field declarations, we have that $\texttt{nFR}(rc, L_c) \geq 1$, which constitutes a contradition.

Therefore, the property *Nonnegativity* holds for metric TAC.

Null value Let us assume that TAC does not fulfill property *Null value*. Therefore, there exists a client library L_c, and a server library L_s such that $\texttt{R}(L_c, L_s) = \emptyset$ and $\texttt{TAC}(L_c, L_s) \neq 0$.

If $\texttt{TAC}(L_c, L_s) \neq 0$ then, as stated in equation 3.9, there has to be a class rc such that $rc \in \texttt{RC}(L_c, L_s, \texttt{distance})$. However, if $rc \in \texttt{RC}(L_c, L_s, \texttt{distance})$, then rc belongs to L_s, and is reachable from L_c. This creates a relation between L_c and L_s, and therefore contradicts $\texttt{R}(L_c, L_s) = \emptyset$.

In conclusion, the property *Null value* holds for metric TAC.

Monotonicity Assuming that property *Monotonicity* does not hold for metric TAC, let L_c be a client library containing class c_c, and L_s be a server library. Also, let c'_c be the resulting class of adding new relationships with L_s to class c_c. Then, $\texttt{R}(c_c, L_s) \subseteq \texttt{R}(c'_c, L_s)$. Let L'_c be the client library resulting from replacing class c_c by c'_c in L_c. Therefore, $\texttt{R}(L_c, L_s) \subseteq \texttt{R}(L'_c, L_s)$.

Since TAC does not fulfill property *Monotonicity*, we have that $\texttt{TAC}(L_c, L_s) > \texttt{TAC}(L'_c, L_s)$. Hence, for a certain distance, it is true that $\texttt{TACD}(L_c, L_s, \texttt{distance}) > \texttt{TACD}(L'_c, L_s, \texttt{distance})$, according to equation 3.8. As equation 3.9 indicates, this can be true in two cases.

The first option is $|\texttt{RC}(L_c, L_s, \texttt{distance})| > |\texttt{RC}(L'_c, L_s, \texttt{distance})|$. In other words, there are more classes from L_s reachable from L_c than from L'_c. Since the only difference between L_c and L'_c is the replacement of class c_c by c'_c, and according to the definition of class c'_c, this is not possible.

Therefore, the last option is that there is a class $rc \in \texttt{RC}(L_c, L_s, \texttt{distance}) \wedge rc \in \texttt{RC}(L'_c, L_s, \texttt{distance})$, such that $\texttt{nFR}(rc, L_c) > \texttt{nFR}(rc, L'_c)$. This implies that the number of field declarations that reach rc in $L_c$is greater than in L'_c, which is a contradicion with the definition of L'_c.

Therefore, property *Monotonicity* holds for TAC.

Merging of classes Let L_c be a client library, and let classes c_1 and c_2 be classes implemented in L_c. Also, let c', created as follows $c' = c_1 + c_2$, and L'_c be a client library based on L_c in which c_1 and c_2 have been replaced by c'. Assuming *merging of classes* does not hold for TAC, we have that $\texttt{R}(c_1, L_s) \subseteq \texttt{R}(c', L_s) \wedge \texttt{R}(c_2, L_s) \subseteq \texttt{R}(c', L_s) \wedge \texttt{TAC}(L_c, L_s) > \texttt{TAC}(L'_c, L_s)$.

L_c and L'_c are only different in the replacement of c_1 and c_2 by c'. Hence, there has to be a field declaration from c_1 or c_2 which reaches a class $rc \in \texttt{RC}(L_c, L_s, \texttt{distance})$, and is not found in c'. However, the reachability through a field declaration is a relation included in $\texttt{R}(c_1, L_s)$ or $\texttt{R}(c_2, L_s)$, and we have that $\texttt{R}(c_1, L_s) \subseteq \texttt{R}(c', L_s) \wedge \texttt{R}(c_2, L_s) \subseteq \texttt{R}(c', L_s)$, which constitutes a contradiction.

Therefore, TAC fulfills property *Merging of classes*.

Merging of unconnected classes Let L_c and L_s be a client and a server library respecively. Let c_1 and c_2 be classes implemented in L_c, such that $\texttt{R}(c_1, L_s) \cap \texttt{R}(c_2, L_s) = \emptyset$. Let c' defined as $c' = c_1 + c_2$. Hence, $\texttt{R}(c_1, L_s) + \texttt{R}(c_2, L_s) = \texttt{R}(c', L_s)$. Let L'_c be a client library which is the result of replacing c_1 and c_2 by c' in L_c.

Assume that the TAC does not fulfill *Merging of unconnected classes*. This means that $\texttt{TAC}(L_c, L_s) \neq \texttt{TAC}(L'_c, L_s)$. Since TAC has the property *Merging of classes*, we know that $\texttt{TAC}(L_c, L_s) \geq \texttt{TAC}(L'_c, L_s)$, which leaves $\texttt{TAC}(L_c, L_s) > \texttt{TAC}(L'_c, L_s)$.

Then, there exists a field declaration in c_1 or c_2 which reaches a class included in L_s and is not included in c. However, we have that $\texttt{R}(c_1, L_s) + \texttt{R}(c_2, L_s) = \texttt{R}(c', L_s)$, which creates a contradiction.

Therefore, the metric TAC fulfills property *Merging of unconnected classes*.

3.2 Measuring coverage of the dependency

The metrics presented in this section measure the dependencies from a different perspective. Instead of measuring import coupling between the client library and the server library, we look at how much of the server library is used by the client library. With these metrics, the developers, for example, can estimate the probability that a breaking change in a library affects their code.

3.2.1 Definition of coverage

Just as with the coupling metrics, it is necessary to define the characteristics of the coverage that will be measured. In this case, there is no framework indicating which are the relevant characteristics to define and with which criteria.

Therefore, based on the criteria discussed for the definition of coupling, we have created the list criteria to define the coverage of a dependency, discarding those that are only relevant for coupling.

Type of connection For these metrics, since it is not about coupling but about how much of the server library is used by the client library, the metrics will not be focused solely on one type of connection. Instead, we will consider every type of connection in the measurement of coverage.

The types of connections discussed previously (see section 3.1.1) are some of the most common types of connection. Nevertheless, there are others which are also considered.

Given class a and class b...

- ... class a has an annotation of type b.
- ... class a has a declared field with an annotation of type b.
- ... class a has method m, which has an annotation of type b.
- ... class a has method m, which has a parameter with an annotation of type b.
- ... class a has method m, which throws an exception of type b.

Granularity In this case, we have to define a unit to measure which percentage of the server library units are being used by the client library. According to the type of connections, two different kinds of units can be used: methods and classes. For the goal of these metrics, it is also necessary to use both units of measure. For example, the breaking changes of a library can be at a metric or for an entire class, and we want to measure it for both cases. Therefore, we define two metrics, one for methods and another for classes. Nevertheless, the metrics' aggregation level is still at the library level since the goal is to measure the percentage of reachable units.

The last aspect of the granularity to consider is how the connections are evaluated. The goal is to know how many items are reached, but not how many times. Therefore, we count the distinct items at the other end of the connections.

Direct & Indirect Finally, we have to decide whether to consider indirect usage or not. Since we want to know the total percentage of the server library coverage, we will consider both the units that are directly used and those indirectly used.

Inheritance Although these metrics measure coverage and not coupling, we still have to decide how the metrics deal with inheritance. Just as in the case of the coupling metrics, all the polymorphic implementations of the methods and the descendants of the classes will be considered in the calculation of the coverage metrics. This is because all the possible implementations might be executed, and therefore should be considered as covered. Finally, we have defined that a method belongs to both the class that declares it and the class that implements it. Both classes are reachable from the client and should be considered as covered.

Summary The metrics resulting from the description of coverage can be found in Table 3.3.

Metric	Type of connection	Unit of measure	Aggregation level	Counting connections	Direct/Indirect	Inheritance
% Reachable classes	All	Class	Library	Distinct items	Both	Accounted for
% Reachable methods	All	Method	Library	Distinct items	Both	Accounted for

Table 3.3: Characteristics of the coverage metrics

3.2.2 Formal definition of the metrics

Percentage of reachable classes

This metric calculates the percentage of coverage of a dependency using classes as the unit of measure. Therefore, it calculates the cardinality of the set of classes implemented in the server library (L_s) that are reachable from the code of the client library (L_c), denoted $\texttt{RC}(L_c, L_s)$. The number of reachable classes is divided by the total number of classes in L_s.

As explained in section 3.2.1, all types of connections are considered for this metric. Hence, $\texttt{RC}(L_c, L_s)$ includes all the classes reachable through any of the types of connection or a combination of these, accounting for inheritance.

$$\texttt{\%ReachableClasses}(L_c, L_s) = \frac{|RC(L_c, L_s)|}{|C(L_s)|} \tag{3.10}$$

Pergentage of reachable methods

This metric works exactly as the previous one, but instead of using the class as the unit of measure, it uses methods. Therefore, it divides the number of elements in the set of methods from the server library (L_s) that are reachable from the code of the client library (L_c), $RM(L_c, L_s)$, by the total number of methods ($|M(L_s)|$).

For this metric, since the unit of measure is the method, the only type of connection through which a method is reachable is the method call or method invocation. Thus, the methods included in the reachable classes are not considered reachable by default since it is not sure if the method has been invoked. Nevertheless, for the reachable methods, the polymorphic implementations of these are also considered.

$$\texttt{\%ReachableMethods}(L_c, L_s) = \frac{|RM(L_c, L_s)|}{|M(L_s)|} \tag{3.11}$$

3.2.3 Theoretical validation

In this section, the theoretical validation of the metrics `%ReachableClasses` and `%ReachableMethods` is done by proving the properties that these metrics should fulfill. Given that the two metrics are highly similar, the proofs are done for the first metric, namely `%ReachableClasses`, but could easily be done for the second one with the same reasoning.

The properties chosen for these metrics are the ones that the aspect being measured should have. We based the properties on the work by Srinivasan and Devi [21], which reviews the methodologies to validate metrics in software engineering. We obtained the following list of properties, which apply to these metrics. The first two properties were originally described by Weyuker [33] and the last three by Briand et al. [23]. Some properties are originally defined using the class as the aggregation level. Therefore, we adapted the definition of the properties for metrics with library as the aggregation level.

1. **Noncoarseness:** Two different libraries can have different values for the same metric.
2. **Nonuniqueness:** There can exist different libraries with the same value.
3. **Nonnegativity:** The value of the metric should never be negative.
4. **Null value:** The value of the metric is expected to be zero if there is no usage of the server library in the client library code.
5. **Monotonicity:** It is expected that if more usage is added from the client library to the server library, the metric value does not decrease.

Noncoarseness Let us consider the two cases in Figure 3.4. In the first case, `Lib1` would be the client library, and `Lib2` the server library. In the example, we can see that the number of classes in the server library is: $|\texttt{C}(Lib2)| = 3$. Also, the number of classes of the server library, reached by the client library is $|RM(Lib1, Lib2)| = 2$. Therefore, following the equation 3.10, we have that $\texttt{\%ReachableClasses}(Lib1, Lib2) = \frac{2}{3} = 0.66$.

In the second case in Figure 3.4 we take `Lib3` as the client library, and `Lib4` as the server library. Therefore, we have $|\texttt{C}(Lib4)| = 3$ and $|RM(Lib3, Lib4)| = 3$, which means that the value of the metric is $\texttt{\%ReachableClasses}(Lib3, Lib4) = \frac{3}{3} = 1$.

Therefore, two different libraries can have different values for `%ReachableClasses`, which means that this metric fulfills the property *Noncoarseness*.

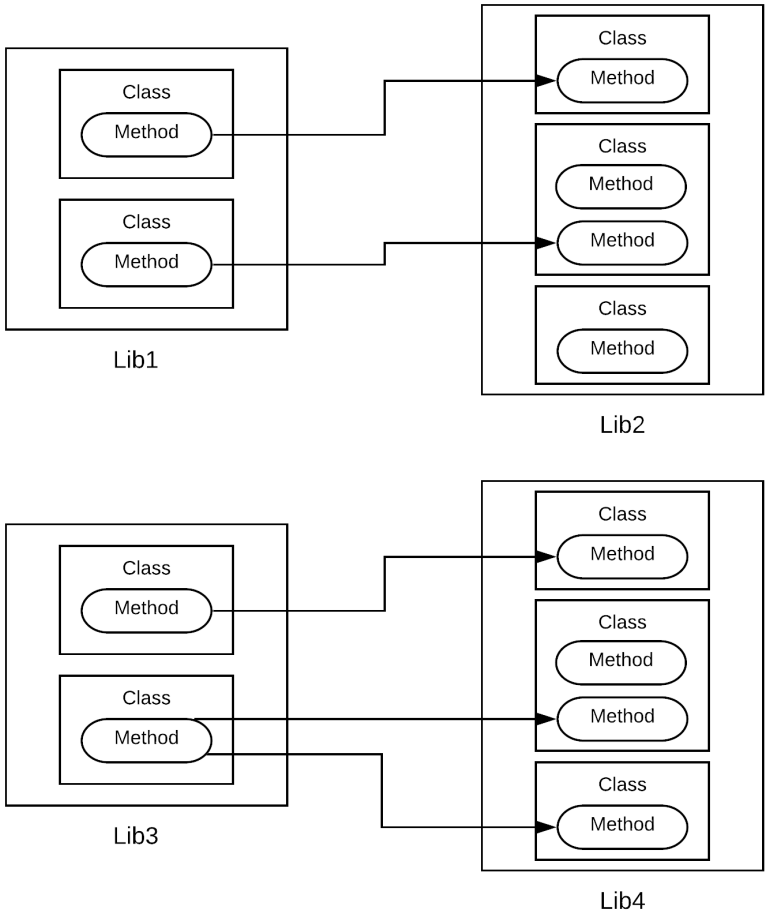

Figure 3.4: Example of noncoarseness, percentage of reachable classes

Nonuniqueness Following the first example in Figure 3.5 we take `Lib1` as the client library, and `Lib2` the server library. Therefore, we can see that $|\texttt{C}(Lib2)| = 3$, and $|RM(Lib1, Lib2)| = 2$. Hence, calculating the metric using equation 3.10, we obtain $\texttt{\%ReachableClasses}(Lib1, Lib2) = \frac{2}{3} = 0.66$.

In the second case in Figure 3.4 we have `Lib3` as the client library, and `Lib4` as the server library. Looking at the example, we know that $|\texttt{C}(Lib4)| = 3$ and $|RM(Lib3, Lib4)| = 2$. Therefore, the metric for these two libraries is $\texttt{\%ReachableClasses}(Lib3, Lib4) = \frac{2}{3} = 0.66$.

Therefore, two different libraries can have the same value for metric `%ReachableClasses`. Hence, the property *Nonuniqueness* holds for this metric.

Nonnegativity Assume that the metric `%ReachableClasses` does not fulfill this property. Let L_c be a client library, and L_s be a server library, such that $\texttt{\%ReachableClasses}(L_c, L_s) < 0$. Then, we have that $|RC(L_c, L_s)| < 0 \oplus |C(L_s)| < 0$. This means that the cardinality of a set is negative, which is not true by definition. Therefore, it is a contradition.

Hence, the metric fulfills property *Nonnegativity*.

Null value Let L_c be a client library, and L_s be a server library. There is no usage in L_c of library L_s, $RC(L_c, L_s) = \emptyset$. Assuming that the property null value does not hold for `%ReachableClasses`, then $\texttt{\%ReachableClasses}(L_c, L_s) \neq 0$. Since this metric has the property nonnegativity, it follows that

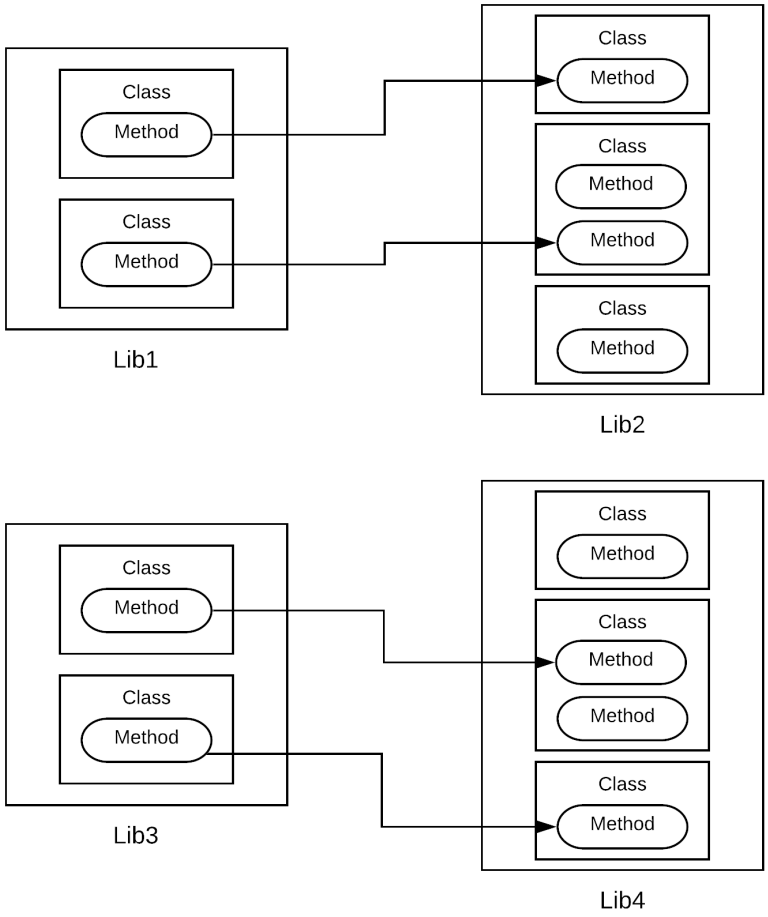

Figure 3.5: Example of nonuniqueness, percentage of reachable classes

`%ReachableClasses`$(L_c, L_s) > 0$. Therefore, $\frac{|RC(L_c,L_s)|}{|C(L_s)|} > 0$ and $|RC(L_c, L_s)| > 0$, which contradicts that $RC(L_c, L_s) = \emptyset$.

Therefore, the property *Null value* holds for metric `%ReachableClasses`.

Monotonicity Having L_c a client library, and L_s a server library. Let c_c be a class in L_c, and c'_c a class created as the result of adding more connections to L_s in c_c. Then, let L'_c be a client library resulting of replacing c_c by c'_c in L_c. Assuming that the property does not hold, we would have that `%ReachableClasses`$(L_c, L_s) >$ `%ReachableClasses`(L'_c, L_s). Therefore, since $|C(L_s)|$ is the same in both cases because L_s does not change, we have that $|RC(L_c, L_s)| > |RC(L'_c, L_s)|$. This means that there is some c_s in L_s, which is reachable from c_c and not reachable from c'_c. However, c'_c contains all the connections between c_c and L_s, and that is therefore not possible.

In conclusion, the metric `%ReachableClasses` has the property *Monotonicity*.

3.3 Measuring usage per class

In this case, the goal is to give low-level information on the usage of the server library of a dependency from the client library. It can be useful for a developer or a maintainer to know exactly which parts of the code of the client are using the server library. For example, if there is a breaking change or if the server library has to be replaced. Therefore, the goal is to know how much usage there is of a server library for each class of the client library.

This section describes how the usage is measured by these metrics and provides a formal definition of the metrics. Finally, we perform a theoretical validation of the metrics.

3.3.1 Definition of usage per class

To define the usage to be measured by these metrics, we use the relevant criteria for this case. We based the list on the criteria used to define coupling, removing those not applicable for the definition of usage.

Type of connection Just as with the coupling metrics, all the types of connection can be relevant, but the usage of each type of connection might have a different impact. Therefore, these should not be measured together. Since this work is meant to be the first approach, we select two possible connection types.

First, the *method invocation*, since it is the only type of connection that refers to the methods used in the dependencies, and therefore is the only one indicating where the server libraries' methods can be invoked from. Then, to represent how the classes of the server libraries are used, and for consistency with the previous metrics, we select the *field declaration*.

Locus of impact In the case of the usage per class metrics, the locus of impact is *import*. This is because we look at the usage from the client class's point of view, which uses a server library through a dependency.

Granularity As the title of the section indicates, these metrics are measuring the usage *per class*. Hence, the aggregation level of the metric is the *class* level for the client library. In other words, we calculate the metric taking a class of the client library and a server library.

The way the connections are counted for these metrics is not considered in the original list created by Briand et al. [11]. Instead, it is counted the number of places of the client class where a connection is originated. For instance, let's take `Lib1` from Figure 3.6 as the client library, and `Lib3` as the server library. From `Class1` to `Lib3` there are four different connections, but there are only two origins of these connections. The same happens with `Class2`, in which there are two connections with `Lib3`, but only one origin.

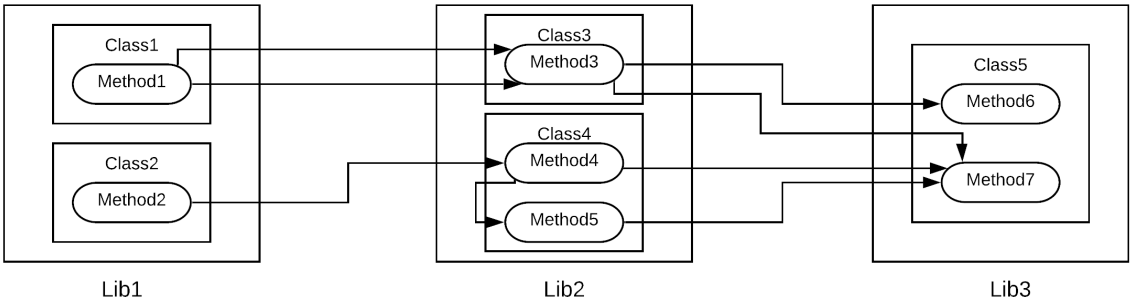

Figure 3.6: Example to calculate usage per class

Direct & Indirect The connection measured can be either direct or indirect, depending on the dependency between the client and the server library. However, this does not change the calculation of the metric. Hence, both types of dependencies are calculated with the same metric.

Inheritance To fully calculate whether a method invocation reaches a server library or not, it is necessary to consider inheritance when computing the reachability. A server library may be only reachable through one of the polymorphic implementations of a method or a descendant of a reachable class. Therefore, the usage per class metrics, do *account for inheritance*.

Summary Based on the criteria discussed above, we define two metrics. The summary of the characteristics of these two metrics can be found in Table 3.4.

Metric	Type of connection	Aggregation level	Counting connections	Direct/Indirect	Inheritance
#1	Method invocation	Class	#origin of connection	Both	Accounted for
#2	Field declaration	Class	#origin of connection	Both	Accounted for

Table 3.4: Criteria of the set of metrics

3.3.2 Formal definition of the metrics

In this section, we provide a formal definition of the metrics *method invocations per class* and *field declarations per class*.

Method invocations per class Having a client library (L_c), and a server library (L_s). The result of this metric for a class of the client library ($c_c \in \texttt{C}(L_c)$) corresponds to the number of method invocations contained in the methods of c_c ($m_c \in \texttt{M}(c_c)$) such that, the call graph created from these method invocations reach a method implemented in the client library ($m_s \in \texttt{M}(L_s)$). The set with the method invocations in m_c that reach a method in L_s is $\texttt{nIR}(m_c, L_s)$, where the computation of the reachability accounts for inheritance.

$$\texttt{\#MethodInvocations}(c_c, L_s) = \sum_{m_c \in \texttt{M}(c_c)} |\texttt{nIR}(m_c, L_s)| \tag{3.12}$$

Field declaration per class This metric is calculated in the following manner. For a class c_c of a client library L_c ($c_c \in \texttt{C}(L_c)$), and a server library L_s. The result of the metric for c_c, corresponds to the number of field declarations in c_c such that, through field declarations reach a class implemented in L_s. The set of field declarations that reach L_s is denoted $\texttt{nFR}(c_c, L_s)$, where the computation of the reachability of L_s considers inheritance.

$$\texttt{\#FieldDeclarations}(c_c, L_s) = |\texttt{nFR}(c_c, L_s)| \tag{3.13}$$

3.3.3 Theoretical validation

This section contains the proofs, for the metrics `#MethodInvocations` and `#FieldDeclarations`, of the properties that these should have. The proofs are done for one metric since they can be done for the other metric very similarly.

The properties chosen in this case are the same as the ones chosen in Section 3.2.3 since these properties can also be applied for these metrics. Nevertheless, the description of the properties has been adapted to fit the characteristics of the metrics.

1. **Noncoarseness:** Two different classes can have different values for the same metric.
2. **Nonuniqueness:** There can exist different classes with the same value.
3. **Nonnegativity:** The value of the metric should never be negative.
4. **Null value:** The value of the metric is expected to be zero if there is no connection of the type measured by the metric from the client class to the server library.
5. **Monotonicity:** It is expected that if more usage is added from the client class to the server library, the metric value does not decrease.

Noncoarseness Following the first case in Figure 3.7, we take textitLib1 as the client library, `Lib2` as the server library, and the only class in `Lib1` as c_c. We can see that the class has only one method, for which $|\texttt{nIR}(Method, Lib2)| = 1$. Following the equation 3.12, $\texttt{\#MethodInvocations}(c_c, Lib2) = 1$.

Now we move to the second case in Figure 3.7. In this case we take `Lib3` as the client library, `Lib4` as the server library, and the class in `Lib3` as c_c. Class c_c has only one method, for which $|\texttt{nIR}(Method, Lib4)| = 3$. Hence, $\texttt{\#MethodInvocations}(c_c, Lib4) = 3$.

In conclusion, two different classes can have different values for metric `#MethodInvocations`, which fulfills property *Noncoarseness*.

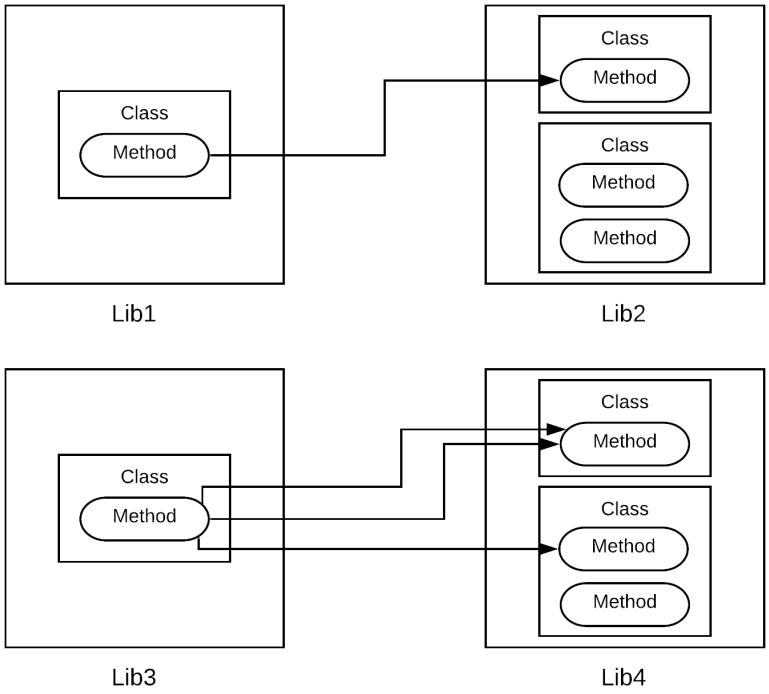

Figure 3.7: Example of noncoarseness, number of method invocations

Nonuniqueness Now we look at Figure 3.8, and in the first example we take `Lib1` as the client library, `Lib2` the server library, and the class in `Lib1` as the client class, c_c. The client class has only one method, for which $|\texttt{nIR}(Method, Lib2)| = 2$. Therefore, $\texttt{\#MethodInvocations}(c_c, Lib2) = 2$.

Focusing on the second case in Figure 3.8, we have that `Lib3` is the client library and `Lib4` the server library. Also, the client class, c_c, is the only class in `Lib3`. We can see that there is one method in c_c, which calls `Lib4` a total of 2 times, $|\texttt{nIR}(Method, Lib4)| = 2$. Hence, $\texttt{\#MethodInvocations}(c_c, Lib4) = 2$.

With this example, we see that two different classes can have the same value for `#MethodInvocations`. Therefore, this metric fulfills the property *Nonuniqueness*.

Nonnegativity If the property does not hold for `#MethodInvocations`, there exists a client library L_c containing c_c and a server library L_s, such that $\texttt{\#MethodInvocations}(c_c, L_s) < 0$. Therefore, a method $m_c \in \texttt{M}(c_c)$, exists such that $|\texttt{nIR}(m_c, L_s)| < 0$. However, the cardinality of a set cannot be negative by definition, and therefore it is a contradition.

Hence, the property *Nonnegativity* holds for metric `#MethodInvocations`.

Null value Assume that the metric does not fulfill property null value. Let L_c and L_s be a client library and a server library, adn c_c a class in L_c. Assume that c_c does not have any connection to L_s, and given that the metric does not fulfill *null value*, $\texttt{\#MethodInvocations}(c_c, L_s) \neq 0$. From the previous proof we know that $\texttt{\#MethodInvocations}(c_c, L_s) \geq 0$, then $\texttt{\#MethodInvocations}(c_c, L_s) > 0$. Hence, there is a method $m_c \in \texttt{M}(c_c)$ such that $|\texttt{nIR}(m_c, L_s)| > 0$. Since $\texttt{nIR}(m_c, L_s)$ is the set of method

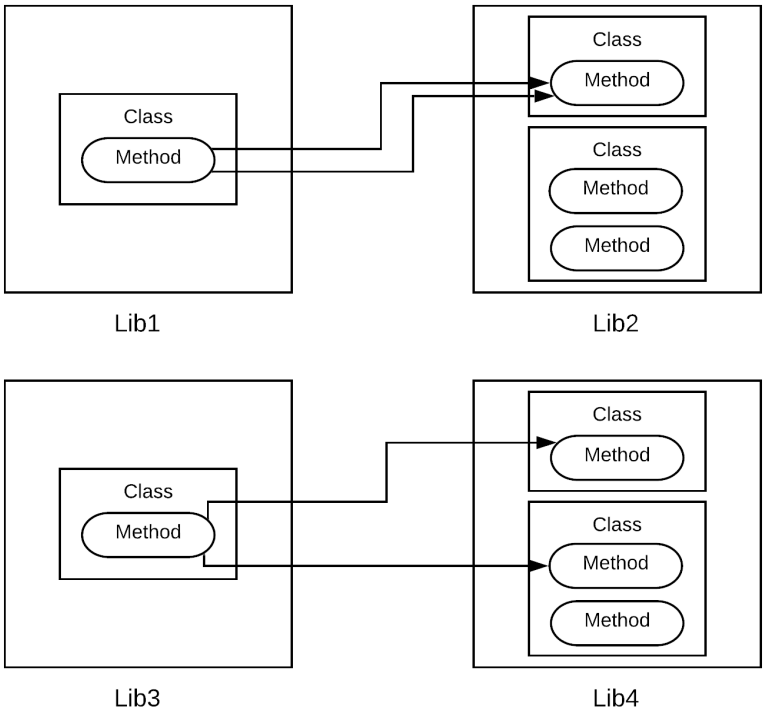

Figure 3.8: Example of nonuniqueness, number of method invocations

invocations in c_c that reach L_s, there has to be a connection between c_c and L_s, which constitutes a contradition.

Therefore, the metric fulfills property *Null value*.

Monotonicity Let L_c be a client library, and let L_s be a server library. Given a class c_c in L_c, we define c'_c as a copy c_c with added connections to L_s. Let L'_c be a client library identical to L_c,. but in which c_c has been replaced by c'_c. Assume that metric `#MethodInvocations` does not fulfill *Monotonicity*. Therefore, $\texttt{\#MethodInvocations}(c_c, L_s) > \texttt{\#MethodInvocations}(c_c, L_s)$. Then, for a given $m_c \in \texttt{M}(c_c)$, and its copy in c'_c, $m'_c \in \texttt{M}(c'_c)$, such that $|\texttt{nIR}(m_c, L_s)| > |\texttt{nIR}(m'_c, L_s)|$. This means that m_c contains a method invocation which reaches L_s and it is not included in m'_c. However, all the method invocations contained in c_c are maintained in c'_c. Therefore, it is a contradition.

Hence, property *Monotonicity* holds for metric `#MethodInvocations`.

Chapter 4

Proof of Concept

In this section, we describe the proof-of-concept implementation to calculate the metrics of the model, which can be found publicly in GitHub [1]. In order to empirically validate the metrics defined in the previous section, it is necessary to be able to calculate them. Therefore, we decided to create a proof-of-concept tool to calculate the metrics for real-world libraries and experiment with the results. First, we discuss the possible techniques to implement it and describe the chosen one. Then, there is an explanation of how each metric is calculated, including pseudo-code, to illustrate it.

4.1 Analysis technique

Several techniques could have been used to implement the PoC that calculates the different metrics proposed.

- Bytecode analysis
- Source code analysis
- Call-level dependency graph

After an initial effort, source code analysis has been discarded. The reason for it is that the source code is needed for both the client library and all its dependencies. Although it is sometimes available in Maven, it is not available as often as the bytecode. Furthermore, obtaining the code from GitHub created issues when resolving the dependency tree. In many cases, the version of the dependency being used in the repository was not yet available in Maven. Therefore, the dependency tree could not be resolved.

The option of call-level dependency graphs is useful for the metrics that measure method invocations. However, the information needed to calculate aggregation coupling is not contained in the call-level graphs. Therefore, the approach to developing this PoC has been bytecode analysis. Furthermore, obtaining the complete dependency tree's call-graphs is still a work in progress within the FASTEN project [2].

The model proposed in this research is meant to be language-agnostic. Nevertheless, the proof-of-concept scope is limited to Java since the bytecode analysis performed by the proof-of-concept is limited to this programming language. Furthermore, the PoC is focused on the libraries available in Maven. It has been decided to limit the scope to Java and Maven since it allows us to compare the results with the research by Soto-Valero et al. [7] (see Section 5.1). Furthermore, the work done by the FASTEN project is also focused on Maven. Therefore, it will be easier in the future to compare the results when measuring the metrics with bytecode analysis and call-graphs since the source will be the same.

4.2 Architecture

The proof-of-concept tool is divided into two parts: the frontend and the backend. The frontend contains the visualizations proposed to see the dependency tree and display the metrics of the model. Meanwhile,

[1] `https://github.com/NuriaBruchTarrega/alexandria`
[2] `https://www.fasten-project.eu/`

the backend receives requests to calculate the dependency tree for a given Maven library. The response to a request consists of the result of the calculation of the metrics for each server library in the dependency tree of the given client library. The implementation of the backend is supported by the libraries *Aether*[3], and *Javassist*[4].

Aether Aether is a library created by Eclipse, which allows us to fetch Maven artifacts from different repositories. Besides, it is also used to resolve the dependencies of a library and create the dependency tree. To create the dependency tree, Aether uses the same strategy to resolve the dependencies as Maven.

In the PoC, this library has been used for the initial steps. The request to calculate the metrics receives the identifiers of the library to analyze (*GroupId*, *ArtifactId*, and *version*). Aether is used to fetch the library *jar* and *POM* files from the Maven Central Repository.

Once the artifact of the client library is obtained, *Aether* is used to resolve the dependencies of the artifact. For the calculation of the metrics to be possible, all the client library dependencies should also be available in the Maven Central Repository. The dependency tree of the artifact is visited, and each of the dependencies is also obtained to have the *jar* files to analyze. Also, the dependency tree calculated with Aether is used to create a custom dependency tree using the custom class `DependencyTreeNode`, which stores the data about each of the libraries needed for calculating the metrics. The `DependencyTreeNode` will be described in greater detail later on.

Javassist Javassist is a library that allows the user to perform bytecode manipulation in a simple way. It has two levels, a source-level and a bytecode level. Using the source-code level, it is possible to perform bytecode analysis and manipulation without a deep knowledge of bytecode. Meanwhile, the bytecode level allows the user to manipulate bytecode directly. For this thesis, the level used is source-code.

The javassist library has been used to interpret and analyze the jars of the client library and all the server libraries. Once the jars of the client library and all its dependencies are available, the first step is to join all the *.class* files in a `ClassPool` object, which is the main object of *Javassist*. Once the `ClassPool` is created, it is used to obtain the classes from the client library, from which the different metrics are calculated. The process of calculating the metrics is explained in greater detail in the following.

Overview In Figure 4.1, one can see the process the proof-of-concept does to calculate the metrics of the model. First, obtaining the *.jar* files of each library included in the dependency tree. Then, by using *Javassist*, it iterates through the classes of the client library to find the usage of the direct dependencies. Next, based on the direct usage calculated, it iterates through the entire dependency tree to find the necessary data to calculate the metrics of the model. Finally, with all the necessary information stored in the data model, the metrics' value is calculated. The process of obtaining the data to store in the data model and the data model itself are described below.

4.2.1 Model of the dependency tree

In order to represent the dependency tree of the client library, the implementation uses the class `DependencyTreeNode`. Each `DependencyTreeNode` contains the information of the library it represents, namely *groupID*, *artifactID*, and *version*. Also, to represent the dependencies of the library represented by each `DependencyTreeNode`, there is a `List` of `DependencyTreeNode`.

To store the information needed to calculate the coupling metrics, there are two other classes: `MicBehaviors`, `AcClasses`, for `MIC` and `AC` respectively. `MicBehaviors` is a map, containing for each method or constructor (behavior) of the server library used to calculate `MIC`, a `Set` with all the method calls from which it is reachable. In the same way, `AcClasses` is a map, in which for each class of a server library used to calculate `AC`, there is a `Set` of all the field declarations from which the class is reachable. The behaviors and classes used to calculate `MIC` and `AC` are those which are reachable through the type of connection of the metric.

Each `DependencyTreeNode` has an object `MicBehaviors` and `AcClasses` for each of the distances at which the metric is measured, stored as a map, where the distance is the key and `MicBehaviors` or `AcClasses`, the value.

Also, for the metrics that measure the dependencies' coverage, two additional fields have been created: `ReachableClasses` and `ReachableBehaviors`. The first field is to calculate the `%ReachableClasses`,

[3] `https://wiki.eclipse.org/Aether/What_Is_Aether`
[4] `http://www.javassist.org/`

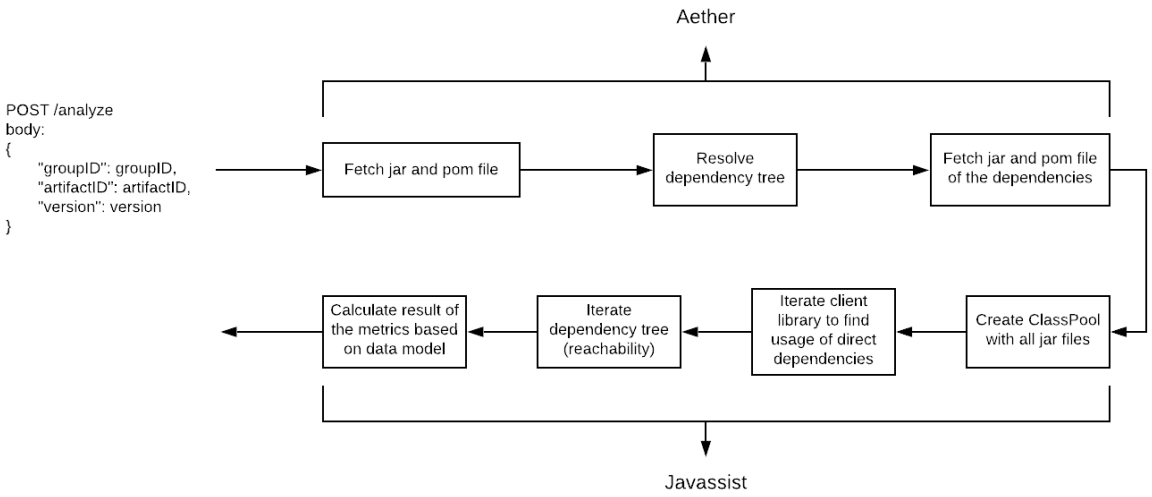

Figure 4.1: Overview of the proof-of-concept implementation of the calculation of the model

and the second one for `%ReachableMethods`. Both `ReachableClasses` and `ReachableBehaviors` are sets containing the classes, in the first case, and behaviors in the second case, of the library represented by the `DependencyTreeNode`, which are reachable from the client library.

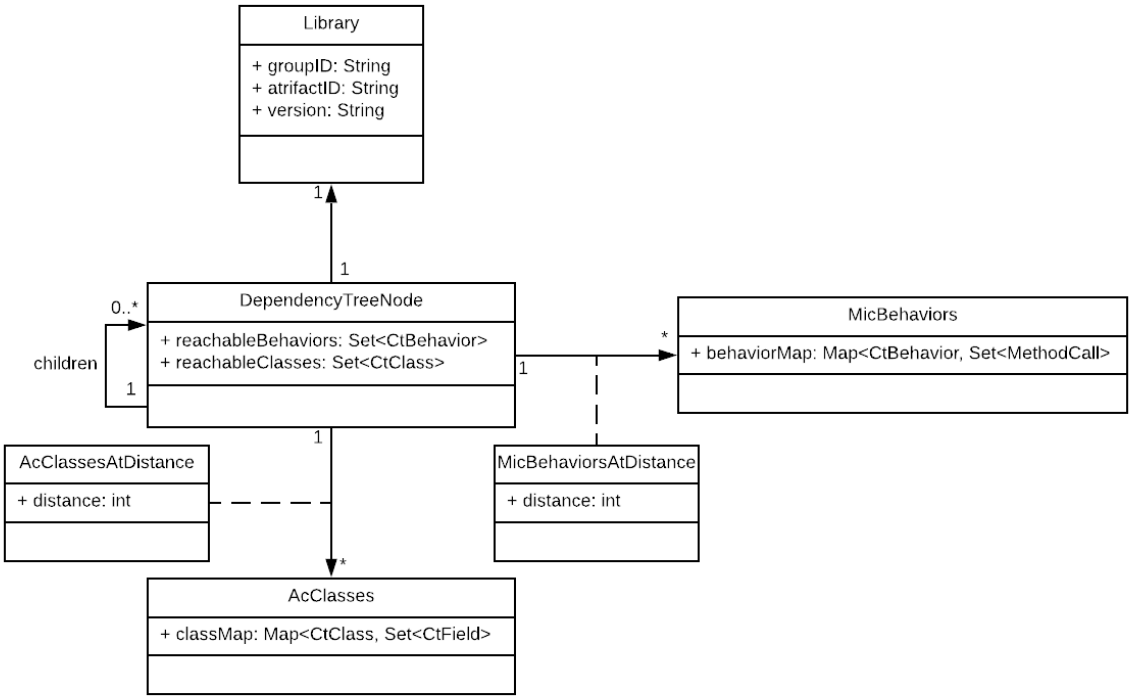

Figure 4.2: Class diagram, dependency tree model

4.3 Calculating coupling metrics

This section contains the description of how the metrics `MIC`, `AC`, `TMIC`, and `TAC` are calculated in the implementation of the PoC. During this section, we use the terminology of the library Javassist. For

example, to refer to methods and constructors, we say behaviors. Also, the classes used from Javassist are named as `Ct<name of the element>`, where `Ct` means compile-time. Therefore, the class representing a java class is `CtClass`.

4.3.1 Method Invocation Coupling

The pseudo-code in Figure 4.3 represents the algorithm used to calculate the metric *MIC* for each one of the direct dependencies of a given client library. The algorithm's output is a map containing the set of `BehaviorCalls` that call each `CtBehavior`. The map is generated for each of the direct dependencies and stored in the `DependencyTreeNode` of each library.

The class `CtBehavior` contains information about the behavior itself and about the class in which it is declared. Each of the `CtBehavior` is used later on to find the polymorphic implementations of the behavior and calculate the metrics for the transitive dependencies.

To calculate this first metric, the implementation iterates through all the classes of the client library (line 3). For each class, the behaviors are obtained (line 4).

Then, the tool iterates through all the behaviors (line 5) and calls the method `instrument`. The main use case of this method is to modify the bytecode of the method, but in this case, we use it to find the connections of interest for this metric. The method `instrument` receives an `ExprEditor` object, which is a class that can be extended to implement the methods for editing the bytecode, which are empty by default. To calculate this metric, the overridden methods are `edit(MethodCall mc)`, `edit(ConstructorCall cc)`, and `edit(NewExpr ne)`. These methods will be called for each method call or constructor call, existing in the method (line 6). The constructor calls in the form of `this()` or `super()` are captured by the method `edit(ConstructorCall cc)`. Meanwhile, the constructor calls in the form of `new Object()` are captured by `edit(NewExpr ne)`.

For each captured call to a behavior, it is checked whether the called behavior belongs to a server library (line 9). In case it does, the call is added to the `MicBehaviors` map of the server library (line 10), with 1 as the distance at which the connection is found since it is a direct dependency.

```
1  clientClasses = classPoolManager.getClientLibraryClasses()
2
3  for each clientClass in clientClasses
4    behaviors = clientClass.getDeclaredBehaviors()
5    for each behavior in behaviors
6      for each behaviorCall in behavior
7        serverBehavior = behaviorCall.getBehavior()
8        serverClass = serverBehavior.getDeclaringClass()
9        if (classPoolManager.belongsToDependency(serverClass))
10         dependencyTreeNode.addMicBehavior(serverBehavior, behaviorCall, distance = 1)
```

Figure 4.3: Pseudo-code of the algorithm to calculate MIC

4.3.2 Aggregation Coupling

The pseudo-code in Figure 4.4 represents the algorithm used to calculate the metric `AC`. The output of the algorithm is a map containing for each `CtClass` of the server library, the set of `CtField` that are of the type of the `CtClass`. The class `CtClass` represents a class and contains the information about the class itself and about the library where it is implemented. `CtClass` is used later on to find the descendants of the class and calculate the metric `TAC` for the transitive dependencies.

The algorithm for this second metric is similar to the previous one. First, the implementation iterates through all the classes of the client library (line 3). Then, for each class, it iterates through all the declared fields (line 5). Next, each of the fields containing a generic type is parsed separately (line 6) to obtain all the types included in the generic (line 7), which are treated individually (line 8). All the simple fields, if the class's implementation is found in a server library (line 13), are included in the calculation of the metric (line 14). The distance is set to 1 due to the server library being a direct dependency.

Inheritance Once the two algorithms (Figures 4.3 and 4.4) are finished, to follow the definition of the metrics as specified in section 3.1.2, the hierarchies in the server libraries are visited. For the metric `MIC`,

```
1  clientClasses = classPoolManager.getClientLibraryClasses()
2
3  for each c in clientClasses
4    fields = c.getDeclaredFields()
5    for each field in fields
6      if ( field .containsGeneric())
7        serverClasses = getAllClasses( field )
8        for each serverClass in serverClasses
9          if (classPoolManager.belongsToClientLibrary(serverClass))
10           dependencyTreeNode.addAcClass(serverClass, field)
11     else
12       serverClass = field .getType()
13       if (classPoolManager.belongsToDependency(serverClass))
14         dependencyTreeNode.addAcClass(serverClass, field, distance = 1)
```

Figure 4.4: Pseudo-code of the algorithm to calculate AC

all the polymorphic implementations of the behaviors are found, and for the metric `AC` all the descendants of the classes.

The algorithms to find the descendants and the polymorphic implementations of the methods are used with each of the libraries with which the client library has a direct dependency and for which coupling was found, either `MIC` or `AC`.

The pseudo-code used to find the polymorphic implementations of the methods can be found in Figure 4.5. The process is as follows: First, iterate through all the classes of the server library, given its `DependencyTreeNode` (line 5). Then, it iterates through the behaviors in the map contained in `MicBehaviors`. If the current library class is a descendant of the class containing the reachable behavior (line 8) and contains a behavior with the same signature as the reachable behavior (line 9), a polymorphic implementation of the behavior has been found. Therefore, the found behavior is added to the `MicBehaviors` with the same set of `BehaviorCall` and distance as the mic behavior (line 11).

```
1  serverLibrary = dependencyTreeNode.getLibrary()
2  micBehaviorsMap = dependencyTreeNode.getMicBehaviors()
3  serverLibraryClasses = classPoolManager.getLibraryClasses(serverLibrary)
4
5  for each serverClass in serverLibraryClasses
6    for each micBehavior in micBehaviorsMap
7      declaringClass = micBehavior.getDeclaringClass()
8      if (serverClass .isSubClassOf(declaringClass))
9        if (serverClass .containsBehavior(micBehavior.getSignature()))
10         serverBehavior = serverClass.getBehavior(reachableBehavior.getSignature())
11         dependencyTreeNode.addMicBehavior(serverBehavior, micBehavior.getBehaviorCalls(), distance)
```

Figure 4.5: Pseudo-code of the algorithm to find polymorphic implementations

The process to find descendants in the case of the metric `AC` is the same as the one in Figure 4.5, but iterating over the `reachableClassesMap` instead of the `reachableBehaviorsMap`. Therefore, if the server class is a sub-class of a reachable class, it is added to the `AcClasses` of the library.

As can be observed in the previous explanation of how the detection of inheritance is done, in this PoC implementation, it is not detected if a class is extended in the client library instead of in the server library.

4.3.3 Transitive Method Invocation Coupling

The calculation of `TMIC` takes place after calculating `MIC` for all the direct dependencies of the client library. The methods used in the calculation of `MIC` from these dependencies are used as a base to calculate `TMIC`.

To calculate the metric for every dependency in the tree, the dependency tree is traversed using a *breadth-first search (BFS)* on the `DependencyTreeNode`. The algorithm starts with the direct dependencies since the client library node has already been used for the calculation of `MIC`. A branch finishes the

traversing either when a node does not have more children or when no reachable methods have been found in a server library. The pseudo-code of the implemented algorithm can be found in Figure 4.6.

For each `DependencyTreeNode` visited, the method `calculateTransitiveMIC` is executed. The pseudo-code of this method can be found in Figure 4.7, which is going to be explained below.

```
1  toVisit = queue(clientLibraryNode.getChildren())
2
3  while (!toVisit.isEmpty())
4    visiting = toVisit.poll()
5    if (visiting.hasMicBehaviors())
6      findPolymorphicImplementations(visiting.getMicBehaviors())
7      if (visiting.hasChildren())
8        calculateTransitiveMIC(visiting)
9        toVisit.add(visiting.getChildren())
```

Figure 4.6: Pseudo-code of the *BFS* used for `TMIC`

```
1  micBehaviors = visitingTreeNode.getMicBehaviors()
2
3  for each micBehavior in micBehaviors
4    behaviorsToVisit = queue(micBehavior)
5    visitedBehaviors = ∅
6    while (!behaviorsToVisit.isEmpty())
7      visitingBehavior = behaviorsToVisit.poll()
8      if (visitedBehaviors.contains(visitingBehavior)) continue
9      visitedBehaviors.add(visitingBehavior)
10
11     for each behaviorCall in visitingBehavior
12       calledBehavior = behaviorCall.getBehavior()
13       calledClass = calledBehavior.getDeclaringClass()
14       if (classPoolManager.isStandardClass(calledClass))
15         continue
16       else if (classPoolManager.isClassInDependency(calledClass, visitingLibrary))
17         visitingLibrary.addMicBehaviorToDependency(calledClass, distance + 1)
18       else // calledClass is in current library
19         behaviorsToVisit.add(calledBehavior)
```

Figure 4.7: Pseudo-code of the algorithm to calculate `TMIC`

First, we obtain all the behaviors of the visited `DependencyTreeNode`, used for the calculation of `MIC` (line 1). For each one of these behaviors (line 3), the call graph of the behavior is iterated to find all the reachable behaviors of the dependencies of the library that is being visited.

A queue of the behaviors that have to be visited is created, containing initially only the current behavior (line 4). A set of all the previously visited behaviors is created (line 5) to avoid infinite loops.

For each behavior in the queue, all the `behaviorCall` are visited (line 11). Then, there are three different cases to consider. First, if the called behavior is implemented in a standard class, the call is ignored (line 14). Also, if the called behavior is implemented in a dependency of the current library, the behavior is added to the behaviors to consider for the dependency (line 16), with the same distance plus 1, since it is one level more of dependency than the current `DependencyTreeNode`. Finally, the last option is if the called behavior is implemented in the current library, in which case it is added to the queue of behaviors to visit (line 19). This way, the entire call graph of the relevant behaviors is visited.

4.3.4 Transitive Aggregation Coupling

The calculation of `TAC` is very similar to the calculation of `TMIC` but using the ac classes instead of the mic behaviors. The pseudo-code of the *BFS* for the `TAC` is in Figure 4.8.

The implementation of the method `calculateTransitiveAC` also follows a similar strategy as the `calculateTransitiveMIC`. However, instead of iterating the call graphs of the reachable methods, it iterates the field declarations of the classes. The pseudo-code can be found in Figure 4.9.

For each class in the field `AcClasses` of the current library (line 3), a queue is created with all the classes to be visited (line 4). The queue is declared containing only the current class. To avoid visiting

```
1  toVisit = queue(clientLibraryNode.getChildren())
2
3  while (!toVisit.isEmpty())
4    visiting = toVisit.poll()
5    if (visiting.hasAcClasses())
6      findDescendants(visiting.getAcClasses())
7      if (visiting.hasChildren())
8        calculateTransitiveAC(visiting)
9        toVisit.add(visiting.getChildren())
```

Figure 4.8: Pseudo-code of the *BFS* used for `TAC`

```
1  acClasses = visitingTreeNode.getAcClasses()
2
3  for each acClass in acClasses
4    classesToVisit = queue(acClass)
5    visitedClasses = ∅
6    while (!classesToVisit.isEmpty())
7      visitingClass = classesToVisit.poll()
8      if (visitedClasses.contains(visitingClass)) continue
9      visitedClasses.add(visitingClass)
10
11     fields = visitingClass.getDeclaredFields()
12     for each field in fields
13       if (field.containsGeneric())
14         classesInField = getAllClasses(field)
15         for each classInField in classesInFIeld
16           if (classPoolManager.isStandardClass(classInField)) continue
17           else if (classPoolManager.isClassInDependency(classInField, visitingLibrary))
18             visitingLibrary.addAcClassToDependency(classInField)
19           else // classInField is in current library
20             classesToVisit.add(classInField)
21       else
22         classInField = field.getType()
23         if (classPoolManager.isStandardClass(classInField)) continue
24         else if (classPoolManager.isClassInDependency(classInField, visitingLibrary))
25           visitingLibrary.addReachableClassToDependency(classInField, distance + 1)
26         else // classInField is in current library
27           classesToVisit.add(classInField)
```

Figure 4.9: Pseudo-code of the algorithm to calculate `TAC`

the same classes multiple times, a set with all the visited classes is also maintained (line 5). When a class is visited, all the fields declared in the class are obtained. For each of the declared fields, just as in the calculation of `AC` (Figure 4.4), the fields containing generic signature are parsed to obtain all the types included in the field (line 14).

Then, for each type in the fields, a similar process to the one used for `TMIC` is done. If the type is implemented in a standard class, it is ignored. Meanwhile, if the type is implemented in a class that belongs to a dependency of the current library, it is added to the relevant classes of the dependency, with the distance of the current `DependencyTreeNode` plus one. Finally, if the type is implemented in the current library, it is added to the queue of classes to visit.

Inheritance During the calculation of `TMIC` and `TAC`, it is possible that one of the classes visited during the algorithms described in Figures 4.7 and 4.9 is found to be abstract. In this case, the possible executed implementations of the class or methods are found by using the same strategy described in Figure 4.5.

4.3.5 Propagation Formula

To calculate the actual value of both, `TMIC` and `TAC`, we use the values stored in `MicBehaviorsAtDistance` and `AcClassesAtDistance` (see Figure 4.2). Then, using the value calculated at each distance, we apply the formula described in the equations 3.6 and 3.8. The only value of these formulas that cannot be extracted from the analysis of the bytecode, is the *propagation factor*. This variable depends on the real-world behavior of these metrics, and its value will be discussed in Chapter 5.

4.4 Calculating coverage metrics

This section describes how the metrics `%ReachableClasses` and `%ReachableMethods` are calculated in the proof-of-concept, including the pseudo-code of the implementation. Both metrics are calculated at the same time since some of the connections are entangled. For example, one class can be reachable because it is the return type of a method. The implementation is done in two steps: finding the classes and methods of the direct dependencies directly reachable from the client library, and a second one to find the rest of the reachable methods and classes of all the dependency tree.

4.4.1 Step 1

The pseudo-code to perform the first step of the calculation of the coverage metrics can be found in Figures 4.10, 4.11, and 4.12. First, in Figure 4.10, the implementation iterates through all the classes in the client library to find the usage of the dependencies (line 3). In the case of these metrics, the coverage can be due to any type of connection, in the methods of each of the classes (line 4) or in the class itself (line 5). In Figure 4.11, we show which connections are detected in the client library methods.

```
1 clientClasses = classPoolManager.getClientLibraryClasses()
2
3 for each clientClass in clientClasses
4   findDependencyUsageInBehaviors(clientClass)
5   findDependencyUsageInClass(clientClass)
```

Figure 4.10: Pseudo-code of the step 1 to calculate coverage of the dependencies (Part 1)

In the pseudo-code of Figure 4.11, the connections are detected in the client library methods. The code iterates through every behavior declared in the current client class (line 3). For each behavior, different types of connections are detected. If any of the types involve a class or a behavior implemented in one of the dependencies, it is added to the information of the dependency as reachable. The first type of connection detected is the method or constructor calls (line 4) and the field access performed in the methods (line 10). Also, the types of the parameters of the method (line 15) and the return type (line 20) are possible connections with the dependencies, as well as the exceptions thrown by the methods (line 24). Finally, the annotations contained in the method are also detected (line 29); these annotations include those specified in the method itself and the annotations of the parameters of the method.

Finally, the connections that happen at a class level are detected with the pseudo-code in Figure 4.12. The first one of these connections is, as was detected by metric `AC`, the field declarations (line 2), the parsing of the generic types, is skipped for simplicity. It is then detected as a connection, the superclass of the client class (line 8), and the interfaces implemented (line 12). Lastly, it is also detected as a connection, the annotations in the class (line 17). These annotations can be in the class itself or any of the declared fields.

4.4.2 Step 2

The pseudo-code of the second step can be found in Figures 4.13, 4.14, and 4.15. The code which iterates through all the dependencies is in Figure 4.13. It contains a *BFS*, which visits the entire dependency tree. For each one of the libraries in the tree, all the reachable methods and classes are found to calculate the coverage of the dependencies of the visited library. This is because the initial step only finds those methods directly used by the client library, but other methods and classes are indirectly reachable.

The code to find all the reachable methods and usage of dependencies is in Figure 4.14. To do so, the directly reachable methods of the library are iterated (line 4). For each of these methods, the call-graph created from the method is visited in the same way it is done for the metric `TMIC` (see Section 4.3.3). However, for each visited method (line 8), all the possible connections are detected (line 9). These connections are the same as the ones detected in Figure 4.11. In the first step, the difference is that for each connection, the only check is if the element on the other end is in a different library. Nevertheless, in this second step, it is also checked if the element at the other end of the connection is in the same library. In this case, it is added to the reachable method or classes. Besides, if it is a method, it is added to the methods to visit since it is part of the call-graph.

```
1  findDependencyUsageInBehaviors(clientClass)
2    behaviors = clientClass.getDeclaredBehaviors()
3    for each behavior in behaviors
4      for each behaviorCall in behavior
5        serverBehavior = behaviorCall.getBehavior()
6        serverClass = serverBehavior.getDeclaringClass()
7        if (classPoolManager.belongsToDependency(serverClass))
8          dependencyTreeNode.addReachableBehavior(serverBehavior)
9
10     for each fieldAccess in behavior
11       serverClass = fieldAccess.getField().getType()
12       if (classPoolManager.belongsToDependency(serverClass))
13         dependencyTreeNode.addReachableClass(serverClass)
14
15     for each parameter in behavior
16       serverClass = parameter.getType()
17       if (classPoolManager.belongsToDependency(serverClass))
18         dependencyTreeNode.addReachableClass(serverClass)
19
20     returnType = behavior.getReturnType()
21     if (classPoolManager.belongsToDependency(returnType))
22       dependencyTreeNode.addReachableClass(returnType)
23
24     exceptions = behavior.getThrowsExceptions()
25     for each exception in behavior
26       if (classPoolManager.belongsToDependency(exception))
27         dependencyTreeNode.addReachableClass(exception)
28
29     annotations = behavior.getAnnotations()
30     for each annotation in annotations
31       if (classPoolManager.belongsToDependency(annotation))
32         dependencyTreeNode.addReachableClass(annotation)
```

Figure 4.11: Pseudo-code of the step 1 to calculate coverage of the dependencies (Part 2)

```
1  findDependencyUsageInClass(clientClass)
2    fields = clientClass.getDeclaredFields()
3    for each field in fields
4      serverClass = field.getType()
5      if (classPoolManager.belongsToDependency(serverClass))
6        dependencyTreeNode.addReachableClass(serverClass)
7
8    superClass = clientClass.getSuperClass()
9    if (classPoolManager.belongsToDependency(superClass))
10     dependencyTreeNode.addReachableClass(superClass)
11
12   interfaces = clientClass.getImplementedInterfaces()
13   for each interfaceClass in interfaces
14     if (classPoolManager.belongsToDependency(interfaceClass))
15       dependencyTreeNode.addReachableClass(interfaceClass)
16
17   annotations = clientClass.getAnnotations()
18   for each annotation in annotations
19     if (classPoolManager.belongsToDependency(annotation))
20       dependencyTreeNode.addReachableClass(annotation)
```

Figure 4.12: Pseudo-code of the step 1 to calculate coverage of the dependencies (Part 3)

The last part is the code to find all the reachable classes and usage of the dependencies in the classes. The reachable classes found until this point are iterated (line 4), and all the reachable classes from those classes are visited. For each visited class, the different types of connections are checked (line 9). The connections checked are the same as in Figure 4.12. Also, if the class at the other end of the connection is in the same library, it is added to the library's reachable classes and in the `toVisit` to check the connections starting in this class.

```
1  toVisit = queue(clientLibraryNode.getChildren())
2
3  while (!toVisit.isEmpty())
4    visiting = toVisit.poll()
5
6    findAllReachableMethodsAndUsageOfDependencies(visiting)
7    findAllReachableClassesAndUsageOfDependencies(visiting)
8
9    toVisit.add(visiting.getChildren())
```

Figure 4.13: Pseudo-code of the step 2 to calculate coverage of the dependencies (Part 1)

```
1  findAllReachableMethodsAndUsageOfDependencies(dependencyTreeNode)
2    directlyReachableMethods = dependencyTreeNode.getReachableMethods()
3
4    for each directlyReachableMethod in directlyReachableMethods
5      behaviorsToVisit = queue(directlyReachableMethod)
6      visitedBehaviors = ∅
7      while (!behaviorsToVisit.isEmpty())
8        visiting = toVisit.poll()
9        findConnectionsInBehavior(toVisit)
10       visitedBehaviors.add(visiting)
```

Figure 4.14: Pseudo-code of the step 2 to calculate coverage of the dependencies (Part 2)

```
1  findAllReachableClassesAndUsageOfDependencies(dependencyTreeNode)
2    directlyReachableClasses = dependencyTreeNode.getReachableClasses()
3
4    for each directlyReachableClass in directlyReachableClasses
5      classesToVisit = queue(directlyReachableClass)
6      visitedClasses = ∅
7      while (!classesToVisit.isEmpty())
8        visiting = toVisit.poll()
9        findConnectionsInClass(toVisit)
10       visitedClasses.add(visiting)
```

Figure 4.15: Pseudo-code of the step 2 to calculate coverage of the dependencies (Part 3)

4.5 Calculating usage per class metrics

These two metrics are calculated after the calculation of the coupling metrics. In order to calculate the usage per class, the sets of `MethodCall` and `CtField` from `MicBehaviors` and `AcClasses`, are used (see Figure 4.2).

The pseudo-code used to calculate the metric `#MethodInvocations`, for a certain dependency and all the client classes from which it is reachable, can be found in Figure 4.16. The `micBehaviors` of the `DependencyTreeNode` are iterated (line 4). For each one of the `methodCall` involved (line 7), the class where the `methodCall` takes place is added to the map of the `#MethodInvocations` with value 1, or summed 1 to the value of the metric for that class, in case the class is already included in the map. In the case of the `#FieldDeclarations` the process is the same, but using the `CtField` stored in `AcClasses` instead.

4.6 Visualization

In this section, we explain how the visualization of the tool has been designed and implemented.

This section contains a brief description of the technologies and libraries used to develop the application's visualization and a description of each part of the visualization.

For each part of the visualization, we discuss different visual aspects of the visualization following the structure used by Kula et al. [3]. However, since the tool implemented for this thesis is meant to be interactive, we have added this new aspect. Therefore, the visual aspects considered are the following: Layout, shape, color, and interaction.

```
methodInvocationMap = map(class, int)
micBehaviorsMap = dependencyTreeNode.getMicBehaviorsMap()

for each entry in micBehaviorsMap
  methodCallSet = entry.getValue()

  for each methodCall in methodCallSet
    clientClass  = methodCall.fromClass()
    if methodInvocationMap.contains(clientClass)
      methodInvocationMap.update(clientClass, value + 1)
    else
      methodInvocationMap.add(clientClass, 1)
```

Figure 4.16: Pseudo-code of the calculation of the `#MethodInvocations` of a dependency

4.6.1 Technologies

The visualization has been implemented using the framework `Angular`[5]. Most of the UI elements used are obtained from `Angular Material`[6]. Finally, for graph representations we have used `vis.js`[7], and `ngx-charts`[8] to display charts.

4.6.2 Dependency Tree

The first visualization element's goal is to provide an overview of the client library's dependency tree after being resolved with the Maven algorithm. In this overview, the maintainer should be able to see the degree of dependency with each of the client library's dependencies. Furthermore, the unused dependencies, as well as the most used ones, should be easily identifiable. An example of this visualization for the client library *org.apache.flink:flink-core:1.9.1* can be found in Figure 4.17

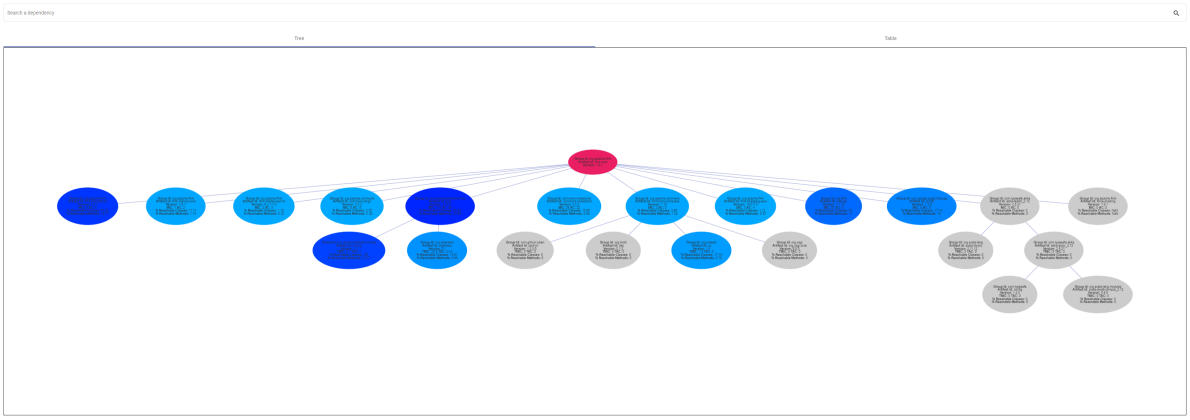

Figure 4.17: Example of the tree visualization

Layout Since this visualization displays the dependency tree of the client library, the chosen layout is a *graph*. In particular, it is a *tree*. In this graph, each node represents a library, and each edge a dependency between the nodes. The tree is organized by levels, such that the first level only contains the client library, and the second level the direct dependencies. The rest of the levels are organized according to the dependencies of the previous levels.

Each node displays the following information about the library: `GroupID`, `ArtifactID`, `version`, and the result of the metrics — `MIC` and `AC` for the direct dependencies, and `TMIC` and `TAC` for the transitive dependencies.

[5] `https://angular.io/`
[6] `https://material.angular.io/`
[7] `https://visjs.org/`
[8] `https://github.com/swimlane/ngx-charts`

Shape For this visualization, the shape of each of the nodes is the same, an ellipse. This is because the differentiation between the nodes is done with the node's color, not the shape. Furthermore, the ellipse is the shape that allows us to display all the necessary information in the node without taking too much extra space.

Color To indicate the state of the dependency with the nodes' color, we have used three different colors. The nodes representing libraries for which no coverage has been found are light grey. The rest of the dependencies have different blue shades, going from lighter blue for the less used and darker blue for the most used. Furthermore, the color of the node representing the client library is dark pink. Finally, when a node is selected (see next paragraph), the color of the node changes to light pink.

Interaction The main problem with the dependency tree visualization is that if the tree contains too many nodes, it is difficult to see its content. To fix this, all the nodes can be selected. When a node is selected, the visualization zooms in the selected node so that the user can see the content.

There are two ways to select a node. The first one is by clicking the node. The second option is by finding the node in the search bar. The search bar has been implemented to give suggestions containing all the libraries' names displayed in the tree. To clear the node selection, the user has to click in the graph view, outside of the nodes.

The second type of interaction has been implemented to display some additional information on the nodes. When a user hovers over a node, a *tooltip* appears. The *tooltip* contains the data of the library and the value of the coupling and coverage metrics. For the coupling metrics, two tables display the value measured and the distance at which the value was measured for each metric.

4.6.3 Dependency Table

The dependency tree visualization gives an overview of the dependencies. However, it is not useful to compare the values of the metrics among dependencies. Therefore, this second visualization is focused on seeing together all the values to sort and compare. Figure 4.18 shows the table visualization for the client library *org.apache.flink:flink-core:1.9.1*.

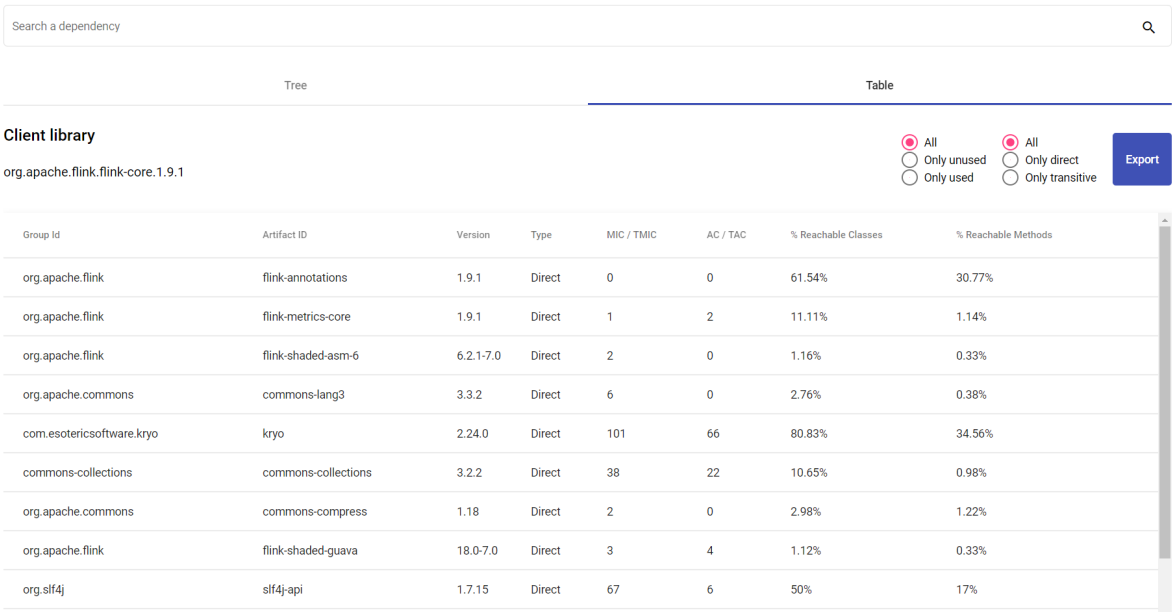

Figure 4.18: Example of the table visualization

Layout To be easy to compare the metrics' value among all the dependencies, the layout chosen for this visualization is a table. The table displays one dependency on each row, while the columns display information about the dependency and the coupling and coverage metrics calculated for the dependency.

For each library, there is a column for the *groupId*, the *artifactId*, the *version*, and whether the dependency is direct or transitive. Then, the rest of the columns the values displayed are `MIC` and `AC` (or `TMIC` and `TAC` for transitive dependencies), and the metrics `%ReachableClasses` and `%ReachableMethods`.

Shape For this visualization, the shape is already defined by the layout, which is a table.

Color The color is used in this visualization only to indicate which library has been selected by the user, in which case the row of the library has a pink background. The rest of the rows have a white background.

Interaction As explained earlier, this visualization is meant to compare the values easier, and therefore the first interaction implemented for this visualization is sorting. The rows of the table can be sorted according to the values of any of the columns. Also, it is possible to filter the content of the table. The rows can be filtered based on whether the dependencies are direct or transitive and if the client library uses the dependencies or not.

4.6.4 Distribution per class

With the previous visualizations, the user has an overview of the dependency tree. However, there is no way the user can have more detailed information about a specific library of the dependency tree. Therefore, and making use of the node selection implemented in the visualization described in section 4.6.2, we have created a visualization to display the usage per class metrics for the analyzed client library and the selected server library. An example of this visualization for the client library *org.apache.flink:flink-core:1.9.1* and the server library *com.esotericsoftware.kryo:kryo:2.24.0*, can be found in Figure 4.19.

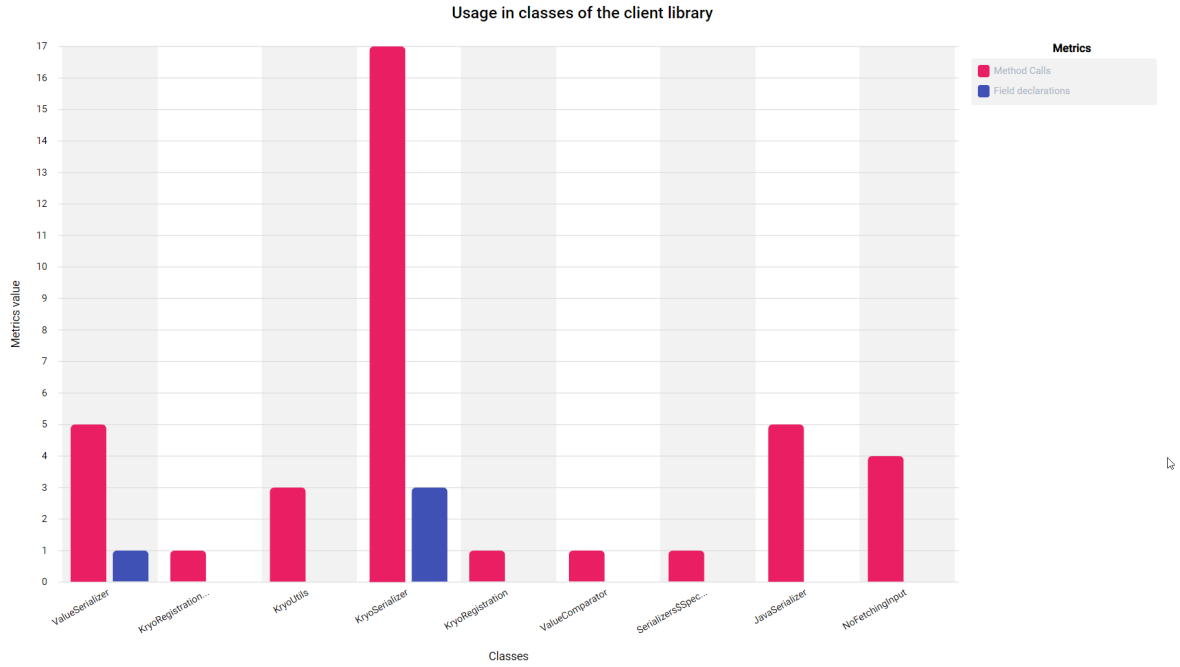

Figure 4.19: Example of the distribution per class visualization

Layout The chosen layout for this visualization is a multi-chart. The chart contains two values for each represented class, the metrics `#MethodInvocations` and `#FieldDeclarations`. Only the classes for which at least one of the two metrics is measured appear in the chart.

Shape The shape chosen for this chart is the bar. The line was discarded since it is not the goal to show progression between the different classes. Hence, the chart used is a multi-bar chart. At the x-axis

of the chart, only the simple names of the classes are displayed to avoid having too much text in the chart. In the y-axis, there is a numeric scale, which corresponds to the calculated `#MethodInvocations` and `#FieldDeclarations`.

Color To follow the palette used for the rest of this tool's frontend, we use the dark blue color for the bars displaying the metric `#MethodInvocations`, while the bars displaying the `#FieldDeclarations` are pink.

Interaction This visualization appears and disappears according to the node selection of the dependency tree and table visualizations.

Also, if the user hovers the cursor over a column, a *tooltip* appears. This *tooltip* contains the values of the calculated metrics since these values are not displayed in the chart. Furthermore, it contains the fully-qualified name of the class to indicate the user the exact path to find the class within the project.

Chapter 5

Experiments

In this section, we describe the experiments conducted with the dependency model created in this thesis, as well as the PoC. For each of the experiments, we explain the goal, the setup, the results, and finally, we discuss the results and the conclusions extracted from these.

5.1 Experiment 1: Comparison

The goal of this experiment is to validate the implementation in the PoC. We compare against the results of the study *"A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem"* by Soto-Valero et al. [7]. In this work, a bytecode analysis is performed on Maven Artifacts to detect which of these artifacts' dependencies are *bloated*, which we refer to as *unused*. Although the PoC created in this thesis does not perform the same kind of analysis, we consider that a dependency is *unused* if all the model's metrics detects no usage.

5.1.1 Experimental setup

Soto-Valero et al. perform a qualitative analysis, in which they analyze 31 libraries available as Maven artifacts. If unused dependencies are found during the analysis, a *Pull Request* is made to the *GitHub* repository of the artifact in which the unused dependencies are deleted from the *pom* file.

With the information in the paper, we collect the *GroupId* and *ArtifactId* of the 31 artifacts. Out of the 31, 2 could not be found in the *Maven Repository Central*. For the other 29, the *version* to use in the experiment is determined by finding the last version released before the experiment by Soto-Valero et al. was conducted — November of 2019.

Of the 29 artifacts, the PoC could not use 13 because either the artifact itself or some dependency could not be downloaded from *Maven Central*. Therefore, we have 16 libraries to analyze and compare the results with the results obtained by Soto-Valero et al. Table 5.1 contains the artifacts' identifiers used in this experiment.

The first idea was to create a request that computed the comparison automatically. However, finding why the results were not the same, in most cases, needed manual checking and research. That is why the analysis of the libraries in Table 5.1 has been executed one by one, and the comparison has been made manually.

5.1.2 Results

The results obtained from the comparison with the data from Soto-Valero et al. [7] is done in two parts: if all the libraries found as unused in the paper are also found as unused by the tool, and if all the other libraries are found as used. For the first part, there are several cases:

- **Correct:** All the dependencies detected as unused in the paper are also detected as unused by the PoC.
- **Testing:** At least one of the unused dependencies is a testing dependency, and therefore it is not detected by the PoC.

Group Id	Artifact Id	Version
org.mybatis	mybatis	3.5.3
org.apache.flink	flink-core	1.9.1
com.puppycrawl.tools	checkstyle	8.27
com.google.auto	auto-common	0.10
edu.stanford.nlp	stanford-corenlp	3.9.2
com.squareup.moshi	moshi-kotlin	1.9.2
org.neo4j	neo4j-collections	3.5.13
org.asynchttpclient	async-http-client	2.10.4
org.alluxio	alluxio-core-transport	2.1.0
com.github.javaparser	javaparser-symbol-solver-logic	3.15.5
io.undertow	undertow-benchmarks	2.0.27.Final
org.teavm	teavm-core	0.6.1
com.github.jknack	handlebars-markdown	4.1.2
ma.glasnost.orika	orika-eclipse-tools	1.5.4
fr.inria.gforge.spoon	spoon-core	8.0.0
org.jacop	jacop	4.7.0

Table 5.1: Identifiers of the Maven artifacts used for comparison

- **Parent:** At least one of the unused dependencies is from a parent module, and it does not appear in the artifact's tree.
- **Used:** At least one of the unused dependencies is found as used by the PoC.

The cases that can happen for the rest of the libraries, the ones that are found as used by Soto-Valero et al., are listed below:

- **Correct:** All the used dependencies, according to the paper, are also used according to the tool.
- **Shaded:** At least one dependency is detected as unused because it is shaded within the jar file of the client library in the building process.
- **Testing:** At least one dependency is found as unused since it is only used for testing but not marked with scope *testing*.
- **Unused:** At least one dependency found as used by the paper is unused in the tool's analysis.

In Table 5.2, there are the results for each one of the client libraries used in the experiment.

5.1.3 Discussion

To evaluate this experiment's results, we will discuss the meaning of the different cases involved in detecting unused dependencies. There are 9 cases in which the unused dependencies were not correctly detected. Out of these 9, 4 can completely be fixed if the tool could also analyze the tests defined in the client library. This could be fixed by doing source-code analysis, including the testing, instead of bytecode analysis. Also, the 2 cases in which the dependencies are inherited from the parent, and the algorithm used to resolve the dependencies does not include them. Finally, there are 3 cases in which there is at least one dependency, which should be unused according to the paper, but it was detected used by the PoC. Therefore, the PoC implementation is overestimating in a certain measure the usage of the dependencies.

Next, we take a look at the detection of used dependencies. There are 9 cases in which at least one of the dependencies that were supposed to be used has no usage detected. Two of these cases are due to the dependencies being shaded within the client library's jar during the build process. Also, one case includes a dependency used for testing. These scenarios could be fixed by doing source-code analysis since the dependencies would not be shaded yet, and the tests could be included in the analysis. Finally, the other 7 cases include dependencies detected as unused, according to the paper by Soto-Valero et al.

Library	Unused in paper	Used in paper
org.mybatis:mybatis:3.5.3	Testing	Shaded
org.apache.flink:flink-core:1.9.1	Testing	Unused
com.puppycrawl.tools:checkstyle:8.27	Testing	Correct
com.google.auto:auto-common:0.10	Testing	Unused
edu.stanford.nlp:stanford-corenlp:3.9.2	Correct	Unused
com.squareup.moshi:moshi-kotlin:1.9.2	Correct	Correct
org.neo4j:neo4j-collections:3.5.13	Correct	Unused
org.asynchttpclient:async-http-client:2.10.4	Used	Shaded
org.alluxio:alluxio-core-transport:2.1.0	Correct	Unused
com.github.javaparser:javaparser-symbol-solver-logic:3.15.5	Correct	Correct
io.undertow:undertow-benchmarks:2.0.27.Final	Parent	Correct
org.teavm:teavm-core:0.6.1	Parent	Correct
com.github.jknack:handlebars-markdown:4.1.2	Correct	Unused
ma.glasnost.orika:orika-eclipse-tools:1.5.4	Testing, Used	Correct
fr.inria.gforge.spoon:spoon-core:8.0.0	Used	Correct
org.jacop:jacop:4.7.0	Correct	Testing, Unused

Table 5.2: Results of the comparison with Soto-Valero et al. [7]

[7]. Therefore, there are some false negatives, which could be related to the way we defined an unused dependency: based on the model's metrics. There may still be some types of connections not detected by the tool (e.g., reflection constructs). In some other cases, it could be a particular type of server library involved, the usage of which cannot be detected on the code. For example, we researched the server libraries of these cases, and it includes a case in which the dependency is related to the compilation of the project. Another case involves a client library in which the part that uses the dependency is not shipped with Maven, and therefore it cannot be detected in the jar file. Finally, there are some empty dependencies - have no classes. A next step could be to investigate what these libraries are used for and how to detect it.

Finding 1: The scope of the PoC has limited the analysis in some cases, such as testing and shaded dependencies. This could be fixed by either including the tests in the jar files or doing source-code analysis instead.

Threats to validity The results of this experiment depend on the results of the research done by Soto-Valero et al. [7]. Therefore, if their implementation has a bug or fails to detect some usage, the results of this experiment might be affected. Moreover, as we have seen in the discussion of the results, there are some cases in which the results of our tool do not match the results obtained by Soto-Valero et al. due to testing dependencies. But without considering these cases, there are also false positives and negatives in our results. However, some of these can be due to testing dependencies — we have already found some cases in which a dependency used for testing is not declared with scope *testing*, which can alter our results. To fix this, it would be necessary to include the tests in the analysis.

5.2 Experiment 2: Coupling metrics significance

The goal of this experiment is to validate if the coupling metrics designed in the model, namely `MIC`, `AC`, `TMIC`, and `TAC`, are a good indicator of the usage of the dependencies by the clients. As a partial validation of the metrics' significance, we compare it with the results gathered from the usage metrics. We want to know how often a dependency is used, either by using classes or methods, and it is detected as uncoupled by the coupling metrics. This way, we know if there are many cases in which a dependency is only used with a type of connection other than method invocation or field declaration.

Data collection The original idea was to measure real-world data about how the clients update the dependencies and their impact on the code. We could then have seen the correlation of this impact with the degree of dependency measured with the coupling metrics. Different approaches were taken to obtain real-world data.

First, we tried to find GitHub commits in which there had been an update of a dependency. However, the search engine in GitHub does not allow to filter the results by the language of the commit. Therefore, most of the results obtained were not useful. Also, most of the updates are only patches, which require only a bump in the version number of the declared dependency.

Based on these findings, the second approach we took was to look for updates that contained breaking changes. To find the libraries, and the versions of these libraries, that had these types of changes, we used the *Maven Dependency Dataset* [34]. Raemaekers et al. used this dataset to analyze the use of semantic versioning and the possible impact of breaking changes [4]. It is possible to query this dataset to obtain libraries with breaking changes, with version numbers, and other libraries that depended on these. However, we need to find the commit of the client library in which the update containing a breaking change was made, and it is not always possible. We considered some of the requirements to be able to analyze a dependency with the PoC. For instance, we need all the dependencies of the client library available in Maven, and testing dependencies cannot be used since they are not analyzed by the tool. Considering all these requirements, obtaining enough data for the experiment from the *Maven Dependency Dataset* had to be done manually.

Next, we contacted the first author of the paper *"Why and How Java Developers Break APIs"* [35], which mines GitHub repositories to find possible breaking changes in APIs, to obtain the dataset of breaking changes created based on their findings. Brito, the author, shared the dataset with us. The dataset includes 24 commits containing breaking changes, which correspond to 19 different libraries. Out of the 25 commits, 12 are from Gradle libraries instead of Maven and cannot be used with the PoC. Besides, we could not find four of the commits in GitHub, and two others correspond to testing libraries, which are out of the scope of the analysis performed by the PoC. Therefore, there were only six breaking changes left, for which three the Maven artifact that these belong to had no dependants for which to do the analysis. The last three have only one dependant, and therefore is not possible to compare the impact of the breaking changes.

Finally, we tried to manually search for deprecated libraries and other libraries that used them — however, similar problems were encountered. Finding commits that replaced a deprecated dependency and the client library and all the dependencies are available in *Maven Central Repository* was a laborious manual task that eventually gave no results.

5.2.1 Experimental set up

In this experiment, we calculated the coupling and coverage metrics of the model for a set of Maven libraries (see list of libraries in Appendix A), to compare the results between the two types of metrics. To run this experiment, we prepared a new request in the API of the PoC. The request has to contain a path to a *.txt* file (tab-delimited). The file has to contain three columns (with headers): *Group Id*, *Artifact Id*, and *version*. For each one of the rows, the metrics are calculated for each of the dependencies. The result of each of the analyses is processed, summarizing all the analyses with the following information:

- **Total number of dependencies:** Number of dependencies of all the analyzed client libraries, including both direct and indirect.
- **Times coupling metrics were not enough:** Number of dependencies for which all the coupling metrics had value zero, but there were methods and classes found reachable by the usage metrics.
- **Times MIC/TMIC were not enough:** Number of times in which there was usage found, but MIC (or TMIC in the case of transitive dependencies) had value zero.
- **Times AC/TAC were not enough:** Number of times in which there was usage found, but MIC (or TMIC in the case of transitive dependencies) had value zero.
- **List server libraries coupling metrics not enough:** The list of *GroupId*, *ArtifactId*, and *version* of the server libraries for which all the coupling metrics were not enough to indicate if there is usage or not.
- **List server libraries MIC/TMIC not enough:** The list of *GroupId*, *ArtifactId*, and *version* of the server libraries for which the metrics `MIC` and `TMIC` were not enough to indicate if there is usage or not.

- **List server libraries AC/TAC not enough:** The list of *GroupId*, *ArtifactId*, and *version* of the server libraries for which the metrics `AC` and `TAC` were not enough to indicate if there is usage or not.

As can be seen, in addition to the number of times that a dependency was used and it was not detected by the metrics (or at least by one of them), the list of server libraries of these dependencies is also stored. This way, it is possible to analyze which types of libraries are those and why the coupling metrics are not enough to detect their usage.

The experiment was run with a file containing 67 client libraries from the *Maven Central Repository*. We selected the client libraries to use for this experiment with the following criteria. First, we used the same libraries as in the comparison experiment (see Section 5.1), but using the last version of each library. We decided to reuse these libraries because the criteria used to select these libraries by Soto-Valero et al. [7] is aligned with the needs of this experiment and are listed below:

- The library is relevant - has more than 100 stars on GitHub.
- The library can be built successfully with Maven.
- Has been developed recently - in the case of Soto-Valero et al., at least October 2019.
- The library has at least one dependency declared.
- It is indicated how to create a pull request.

Although some of the items on the list are not explicitly required for our experiment. We need that the library can be built and is available in Maven as well as that it has at least one relevant dependency (compile scope). Therefore, the libraries in this set are a good fit for the experiment.

To analyze more client libraries and, therefore, more dependencies, we extended the list of libraries. First, we visited the popular libraries list of the *Maven Central Repository* [1]. Also, we queried the dataset generated by Harrand et al. [36] with the 99 most popular libraries from Maven, according to the number of clients these have. For each of the libraries, we selected the last version available in Maven and filtered the resulting list according to the following criteria:

- The artifact of the last version of the library should have at least one dependency with scope compile.
- The artifact and all its dependencies can be obtained from the *Maven Central Repository*.

5.2.2 Results

The summary of the results of this experiment is shown in Table 5.3. We can see the total number of analyzed dependencies, the number of dependencies for which the coverage metrics have found usage, and the coupling metrics have not. The last two rows show the number of dependencies which have more than 0% coverage, and no coupling has been found by the metrics `MIC`/`TMIC` and `AC`/`TAC` respectively.

Analyzed dependencies	699
Metrics are not enough	35
MIC/TMIC are not enough	40
AC/TAC are not enough	147

Table 5.3: Summary of the significance experiment

The server libraries for which usage was found by the coverage metrics but not by the coupling metrics can be found in Table 5.4. The first two columns of this table are the *group id*, and the *artifact id* of the server libraries. Only by looking at the libraries' names one can see that some of them are libraries that contain `annotations`. We have checked the content of all the libraries to find out which include only `annotations`. The result of this search can be seen in the last column of Table 5.4.

[1] `https://mvnrepository.com/popular`

Group Id	Artifact Id	Type
com.fasterxml.jackson.core	jackson-annotations	Annotations
com.github.javaparser	javaparser-symbol-solver-model	Other
com.google.code.findbugs	findbugs-annotations	Annotations
com.google.code.findbugs	jsr305	Annotations
com.google.errorprone	error_prone_annotations	Annotations
com.google.j2objc	j2objc-annotations	Annotations
org.apache.flink	flink-annotations	Annotations
io.grpc	grpc-context	Other
io.netty	netty-codec-socks	Other
jakarta.activation	jakarta.activation-api	Other
org.apiguardian	apiguardian-api	Annotations
org.codehaus.mojo	animal-sniffer-annotations	Annotations
org.codehaus.plexus	plexus-component-annotations	Annotations
org.codehaus.woodstox	stax2-api	Other
org.glassfish.jaxb	jaxb-core	Other
org.jetbrains	annotations	Annotations
org.joda	joda-convert	Other
org.junit.jupiter	junit-jupiter-api	Other
org.neo4j	annotations	Annotations
org.yaml	snakeyaml	Other

Table 5.4: List of the server libraries for which the coupling metrics were not enough to indicate usage

5.2.3 Discussion

Based on the results shown in Table 5.3, we can see that the combination of the two types of coupling metrics allows us to detect used dependencies in 664/699 of the cases, approximately 95%. In addition, the metrics `MIC` and `TMIC` alone, detect 659/699 of the cases, which corresponds to a 94%. This means that `AC` and `TAC` are only really needed in 5 of these dependencies. However, this only talks about whether or not the metrics can define whether a dependency is used or not, which was not the goal of this thesis. Therefore, `AC` and `TAC` are also necessary to consider the type of coupling measured by this metric.

Nevertheless, if we compare the number of times that `AC/TAC` are not able to detect whether a dependency is used or not with the times that the same happens with `MIC/TMIC`, we can conclude that:

> **Finding 2:** Method invocation is enough to detect dependency usage in more cases than Field declaration. Therefore, it is a more significant metric.

This is in line with the intuition that method invocation is the most significant type of connection. Generally, if there is an object of a particular class type in the code, at some point, a method call will be done on this object. However, an object can exist in the code through mechanisms other than field declaration (e.g., return type of a method call or a variable).

Nevertheless, since there are cases in which the combination of both types of metrics was not enough, it indicates that there are some other types of connections to be considered and measured. Given that half of the server libraries used but not detected by the coupling metrics contain annotations, creating a metric to measure the coupling created by annotations would improve these results.

> **Finding 3:** A metric to measure coupling created by the use of annotations would improve the precision of the model. The usage of annotations is sometimes the only type of connection between a client and a server library.

During the process of this thesis, it was intended to create this type of coupling metric. However,

it is not a trivial problem for transitive dependencies. In the coupling metrics defined in this thesis, only one type of connection is considered throughout the whole dependency tree. For example, in the dependency tree in Figure 5.1, the client library uses direct dependency through a connection other than annotations. Then, the *non-annotation server library* uses some annotation from the *annotation server library*. Iterating the dependency tree through annotations connection would still not detect the usage of the transitive dependency.

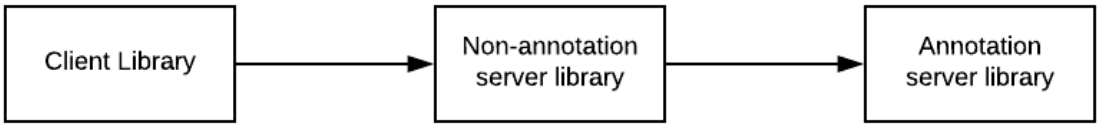

Figure 5.1: Example dependency tree involving annotations

Threats to validity It is important to remark that the results of this experiment, as indicated in the goal of the experiment, should not be interpreted as a full validation of the significance of these metrics. The experiment is checking whether these metrics are significant enough to detect coupling between a client library and a server library or not. However, there is no information about the coupling created by other types of connections. In addition, we have to consider the sample size of this experiment; a larger dataset could change the experiment results.

5.3 Experiment 3: Sensitivity Analysis

As explained before, we have not been able to obtain the real-world data to understand what the impact of the transitive dependencies is, or can be, and correlate it with our transitive coupling metrics `TMIC`, and `TAC`. Therefore, we cannot derive an actual value of the *propagation factor* for these two metrics. Instead, we conduct a sensitivity analysis of the *propagation factor* on these two metrics.

A *Sensitivity Analysis* consists of analyzing how much the output of a model depends on an input variable [37]. In this case, the input variable is the *propagation factor*, and the output is the value of the metrics.

5.3.1 Experimental set up

To run this analysis, we have set up a new request in the API of the PoC. This request receives a list of Maven artifacts in a *.txt* file (tab-delimited). The file includes three columns, containing for each artifact, the following information: *group id*, *artifact id*, and *version*.

The first step is to run the dependency model's calculation for each of the dependencies of the Maven artifacts included in the list. Then, for each one of the transitive dependencies with coupling, we run the sensitivity analysis. Since the *propagation factor* is a value in the range $(0, 1]$, we calculate the value of the metrics incrementing the propagation factor by 0.01 from 0.01 to 1.

We run this experiment with a randomly selected subset of the libraries used for the significance experiment. Then, out of the set of dependencies that can be used for the sensitivity analysis, we select a representative subset of 15 dependencies. This subset is chosen to have dependencies with different distances and values measured at each distance. The dependencies used for the sensitivity analysis can be found in Table 5.5.

5.3.2 Results

With the data obtained from running the experiment, we calculate the covariance of the metrics `TMIC` and `TAC` with the *propagation factor*, to understand how much the value of the metrics changes due to a change in the *propagation factor*. The values of the covariance are displayed in Table 5.6.

#	**Client library**	**Server library**
1	org.asynchttpclient async-http-client 2.12.1	io.netty netty-common 4.1.48.Final
2	org.asynchttpclient async-http-client 2.12.1	io.netty netty-buffer 4.1.48.Final
3	org.easymock easymock 4.2	org.hamcrest hamcrest-core 1.3
4	org.apache.maven maven-project 3.0-alpha-2	org.codehaus.plexus plexus-classworlds 1.3
5	org.springframework.boot spring-boot-autoconfigure 2.3.4.RELEASE	org.springframework spring-core 5.2.9.RELEASE
6	org.springframework.boot spring-boot-autoconfigure 2.3.4.RELEASE	org.springframework spring-jcl 5.2.9.RELEASE
7	org.springframework.boot spring-boot-autoconfigure 2.3.4.RELEASE	org.springframework spring-beans 5.2.9.RELEASE
8	org.eclipse.jetty jetty-server 11.0.0.beta1	org.eclipse.jetty jetty-util 11.0.0.beta1
9	org.asynchttpclient async-http-client 2.12.1	log4j log4j 1.2.17
10	org.alluxio alluxio-core-transport 2.3.0	com.google.protobuf protobuf-javalite 3.11.0
11	fr.inria.gforge.spoon spoon-core 8.2.0	org.eclipse.platform org.eclipse.osgi 3.16.0
12	fr.inria.gforge.spoon spoon-core 8.2.0	org.eclipse.platform org.eclipse.equinox.preferences 3.8.0
13	fr.inria.gforge.spoon spoon-core 8.2.0	org.eclipse.platform org.eclipse.equinox.common 3.13.0
14	com.puppycrawl.tools checkstyle 8.36.2	log4j log4j 1.2.17
15	com.puppycrawl.tools checkstyle 8.36.2	org.apache.geronimo.specs geronimo-jms 1.1_spec_1.0

Table 5.5: Sensitivity analysis, list of dependencies used

#	**TMIC Covariance**	**TAC Covariance**	**TMIC Correlation**	**TAC Correlation**
1	21.40	7.74	0.9921	0.9915
2	35.29	4.46	0.9999	0.9999
3	0.67	1.52	1	1
4	1.43	0.68	0.9999	0.9919
5	15.34	2.28	0.9993	0.9978
6	5.91	1.02	0.9951	0.9919
7	3.54	0.93	1	1
8	16.84	4.55	0.9999	1
9	0.34	2.01	0.9689	0.9446
10	35.94	6.23	0.9999	1
11	10.22	0.59	0.9412	0.9634
12	9.18	1.23	0.9797	0.9613
13	31.63	1.78	0.9742	0.9749
14	1.32	0.67	0.9777	0.9995
15	2.92	1.36	0.9471	0.9689

Table 5.6: Covariance and Pearson correlation of the metrics `TMIC` and `TAC` with the *propagation factor*, for all the dependencies used in the sensitivity analysis

In addition, we also calculate the *Pearson correlation coefficient* [38], since it is the most used when measuring the degree of relationship between two variables. The values of the correlation range from 0.941282 to 1 for `TMIC`, and from 0.9446 to 1 for `TAC`. Figures 5.2 and 5.3 show the a plot with the values of the metrics `TMIC` and `TAC` as a function of *propagation factor*. On the left side, for the client library *fr.inria.gforge.spoon:spoon-core:8.2.0* and the server library *org.eclipse.platform:org.eclipse.osgi:3.16.0*,

and on the right side for *org.easymock:easymock:4.2* and *org.hamcrest:hamcrest-core:1.3*.

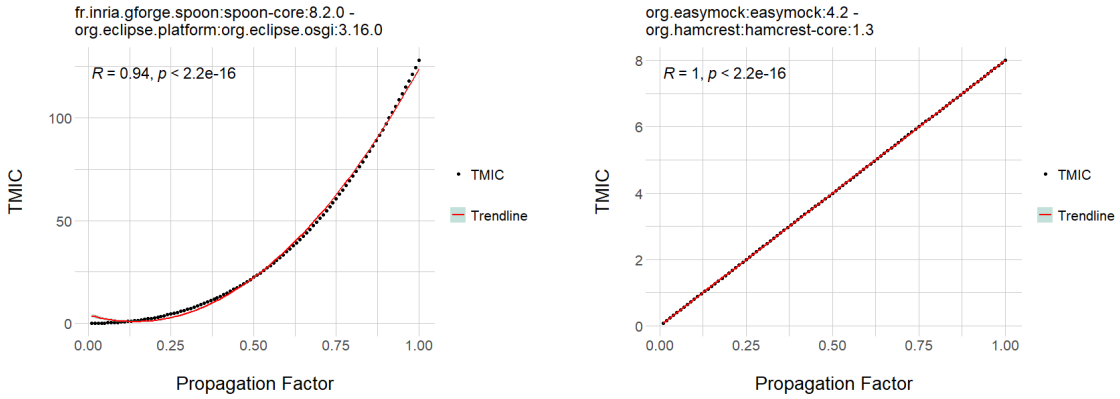

Figure 5.2: `TMIC` as a function of the *propagation factor*, with quadratic regression (left) and linear regression (right). *R* is the Pearson correlation coefficient, and *p* corresponds to the confidence interval

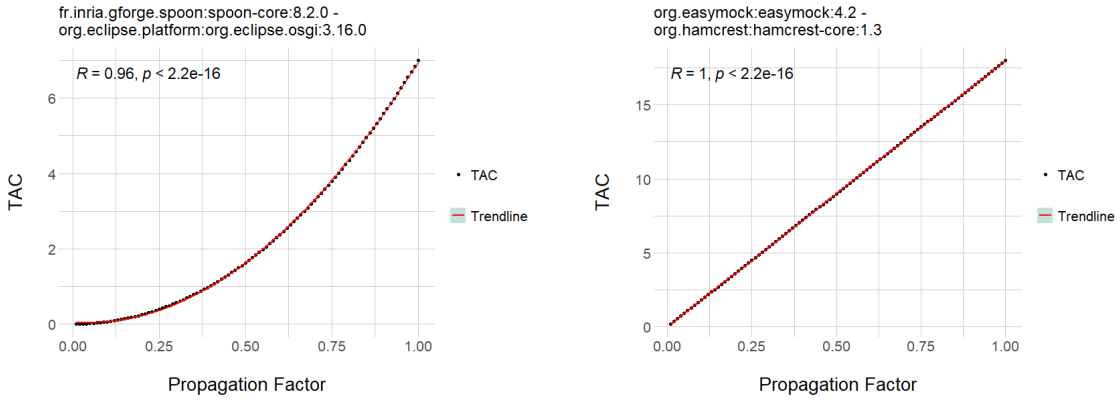

Figure 5.3: `TAC` as a function of the *propagation factor*, with quadratic regression (left) and linear regression (right). *R* is the Pearson correlation coefficient, and *p* corresponds to the confidence interval

5.3.3 Discussion

The sensitivity analysis results indicate a high sensitivity of the two coupling metrics to the propagation factor since there is a high correlation between these two values, the *Pearson correlation coefficient* is always higher than 0.94 (see Table 5.6).

> **Finding 4:** The value of the metrics `TMIC` and `TAC` is highly sensitive to the value of the *propagation factor*.

In Table 5.6, we can see that the values of the covariance for `TMIC` are generally greater than those of `TAC`. This seems to be related to the fact that the values measured for `TMIC` are greater than those measured for `TAC`. Also, the cases in which the covariance is greater than 30 seem to have greater coupling values at each distance. If we look at equation 3.6, the coupling measured at each distance (`TMICD`$(L_c, L_s,$ `distance`$)$), is the value multiplied by the *propagation factor*. Therefore if the *propagation factor* is increased, the total value of the metric will increase more in consequence if the value of

`TMICD`$(L_c, L_s,$ `distance`$)$ is greater. To confirm this intuition, we create a plot to compare the covariance with the sum of the coupling measured at each distance. These plots can be seen in Figure 5.4 and 5.5, for `TMIC` and `TAC` respectively.

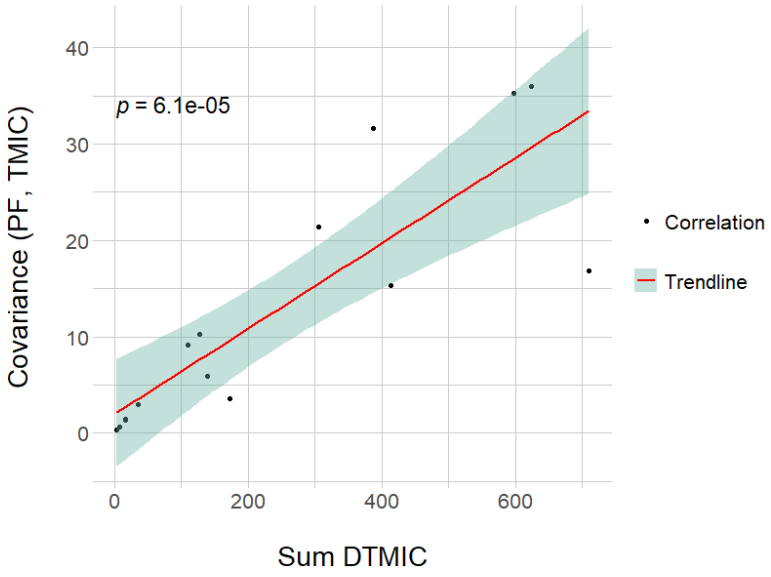

Figure 5.4: Covariance of *propagation factor* and `TMIC` as a function of the sumation of the coupling measured at each distance (`DTMIC`). p corresponds to the confidence interval

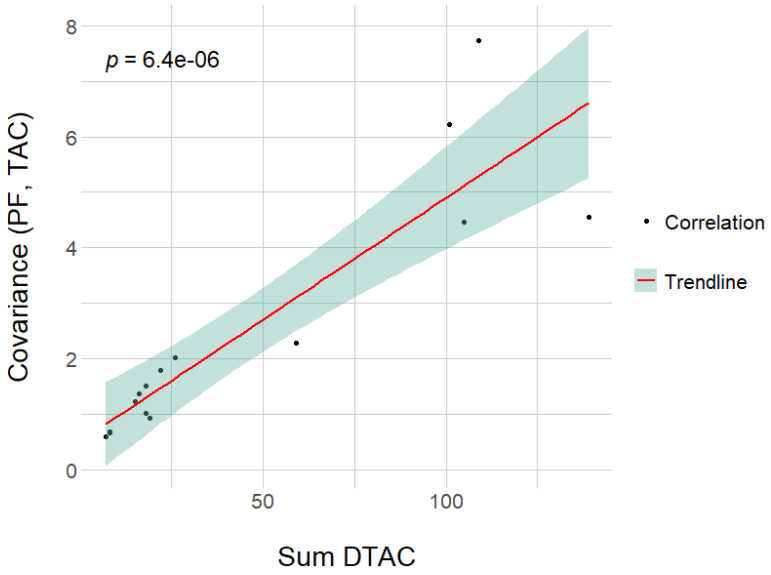

Figure 5.5: Covariance of *propagation factor* and `TAC` as a function of the sumation of the coupling measured at each distance (`DTAC`). p corresponds to the confidence interval

Finally, in the results of the correlation coefficient between the propagation factor and the metrics, we also observe that the cases with the lowest correlation coefficients tend to be dependencies in which there is more distance between the client library and the server library. To compare the correlation and the distances, we create a plot with the correlation coefficient calculated for the sensitivity analysis and

the maximum distance at which coupling is found. The plots for `TMIC`, and `TAC` can be found in Figures 5.6 and 5.7 respectively.

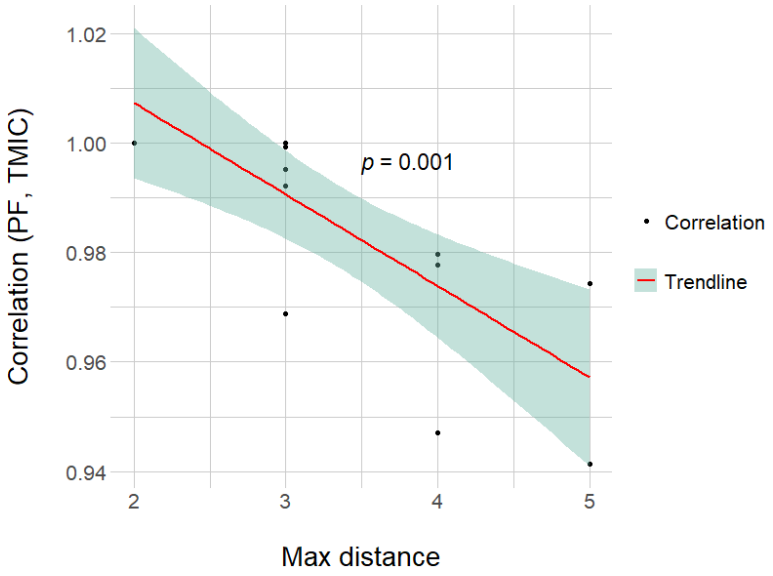

Figure 5.6: The correlation between *propagation factor* and `TMIC` as a function of the maximum distance at which coupling is measured. *p* corresponds to the confidence interval

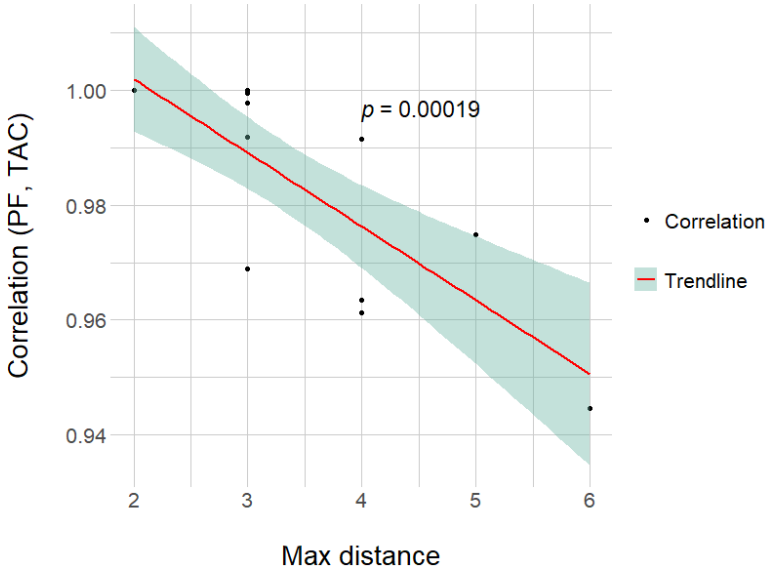

Figure 5.7: The correlation between *propagation factor* and `TAC` as a function of the maximum distance at which coupling is measured. *p* corresponds to the confidence interval

Threats to validity To evaluate the results of this experiment, it is important to consider the sample size of dependencies used. To ensure that the set of dependencies used was as relevant as possible, we selected a representative sample. The selection was made considering the distance between the client

and the server, the number of times the server appeared in the dependency tree of the client, and the coupling measured.

5.4 Experiment 4: Expert Interviews

This last experiment has various goals. The main one is to validate the design of the visualization. According to Munzner [39], there are four levels at which this validation can be done:

1. Domain Problem and Data Characterization
2. Operation and Data Type Abstraction
3. Visual Encoding and Interaction Design
4. Algorithm Design

In this case, we focus on the third option: Visual Encoding and Interaction Design. To carry out this validation, we designed an *Expert Review* through interviews. In addition, the second goal of this experiment is to evaluate the clarity and actionability of the metrics included in the model. Therefore, we added questions about clarity and actionability so that the participants could give their views. Therefore, these two aspects of the metrics, which are included in the set of validation criteria defined by Meneely et al. [22], can also be validated. It is important to mention that this experiment is not a complete validation of the metrics or the visualization in all scenarios. Instead, it is used to evaluate the clarity and actionability of the metrics and the usability of the visualizations in the discussed scenarios, not in a general sense.

5.4.1 Experimental set up

The interview consists of 19 questions and a demonstration of the PoC, with two proposed scenarios in which the interviewee uses the tool. The questions are divided into four sections, which, together with the demonstration, divide the interview into a total of five parts:

1. **Demographics:** The questions of this section are related to the interviewee's professional experience and current job.
2. **Dependency Management:** In this part, the questions are focused on the interviewee's experience with dependency management and the tools used for this purpose.
3. **Demonstration:** The third part is the demonstration of the tool, in which two scenarios are presented to the interviewee. During the discussion of the scenarios, the interviewee controls the mouse to interact directly with the tool.
4. **Visualizations:** The section after the demonstration contains questions about the tool itself and the designed visualizations.
5. **Metrics:** The last section focuses on the designed metrics, the clarity, and comprehensibility of these, as well as actionability.

The interviews contain three types of questions: open answer, binary, and scaled from 1 to 5. During every question, even the binary and scaled questions, the interviewee can make comments or discuss the answer. The list of questions contained in the interview can be found in Table 5.7.

The interviews were done via *Zoom*[2]. *Zoom* offers the possibility of sharing the control of the mouse with other participants and the option of recording the interview. The interviews are recorded to rewatch it afterward and take notes of the interviewees' answers. Therefore, the interview itself feels more like a normal conversation, and there are no pauses.

5.4.2 Results

In this section, we show the answers obtained during the interviews. The results will be discussed in section 5.4.3: the suitability of the visualizations, as well as the clarity and actionability of the metrics.

[2] `https://zoom.us/`

Question	Section	Type
1. What is your software development role?	Demographics	Open answer
2. How many years of experience do you have as a software developer?	Demographics	Open answer
3. Which programming language(s) do you usually use in your job?	Demographics	Open answer
4. Which type of projects do you usually work on?	Demographics	Open answer
5. Do you have experience with dependency management?	Dependency Management	Binary
6. To what extent is it important to you (or do you try) to have the dependencies up to date?	Dependency Management	Scaled
7. To what extent is it important to you to monitor the vulnerabilities that your dependencies may be exploiting?	Dependency Management	Scaled
8. Which tools (if any) do you use for dependency management?	Dependency Management	Open answer
9. To what extent do you think the tools you used so far are helping you to maintain your dependencies?	Dependency Management	Scaled
Scenario 1: You are a new maintainer of the library *org.apache.flink:flink-core*. Since you have not worked in this library's development, you want to see how the dependency tree looks like. What would you look for?	Demonstration	Scenario
Scenario 2: You realize that a library called *kryo* has a new version, which has been announced to contain breaking changes. How likely it would affect your library, and which classes are affected.	Demonstration	Scenario
10. How much do you agree that the tool is useful in the presented scenarios?	Visualizations	Scaled
11. How much do you agree that managing dependencies would be easier with the presented tool?	Visualizations	Scaled
12. With your job in mind, which (if any) are the most useful of the visualizations?	Visualizations	Open answer
13. How much do you agree that the presented tool would be useful in your job?	Visualizations	Scaled
14. Is there some other visualization or change you would like to see? For which cases do you think it would be useful?	Visualizations	Scaled
15. To what extent do you agree that the metrics are clear and comprehensible? (Answer per metric)	Metrics	Scaled
16. To what extent do you agree that the metrics are useful in the described scenarios	Metrics	Scaled
17. To what extent do you agree that the metrics are actionable in the sense that they give you the information you need to make a decision?	Metrics	Scaled
18. Which (if any) do you think are the most useful of the metrics? Based on the tasks that you usually do in your job.	Metrics	Open answer
19. Is there some other metric or change that you would like to be added to the model? In which scenarios do you think it could be useful?	Metrics	Open answer

Table 5.7: Questions of the interview

Demographics

The roles of the 15 participants in the interviews include: Software developer, Software engineer, Technology lead, Head of innovation, Head of development, and Head of product. In some of the results, we differentiate between the answers given by developers and non-developers. In the developers' group, we consider the interviewees who answered with the role of software developer and engineer, and the rest of the interviewees are considered non-developers. The key difference is that the non-developers have tasks related to architecture or management, and therefore their needs and their perspective is different.

The years of experience range from 1 to 20, with an average of 7.13. Half of the interviewees have worked in backend development and web services systems. In addition, some of the other types of projects include mobile applications and frontend development. The languages in which the interviewees have experience can be seen in Figure 5.8.

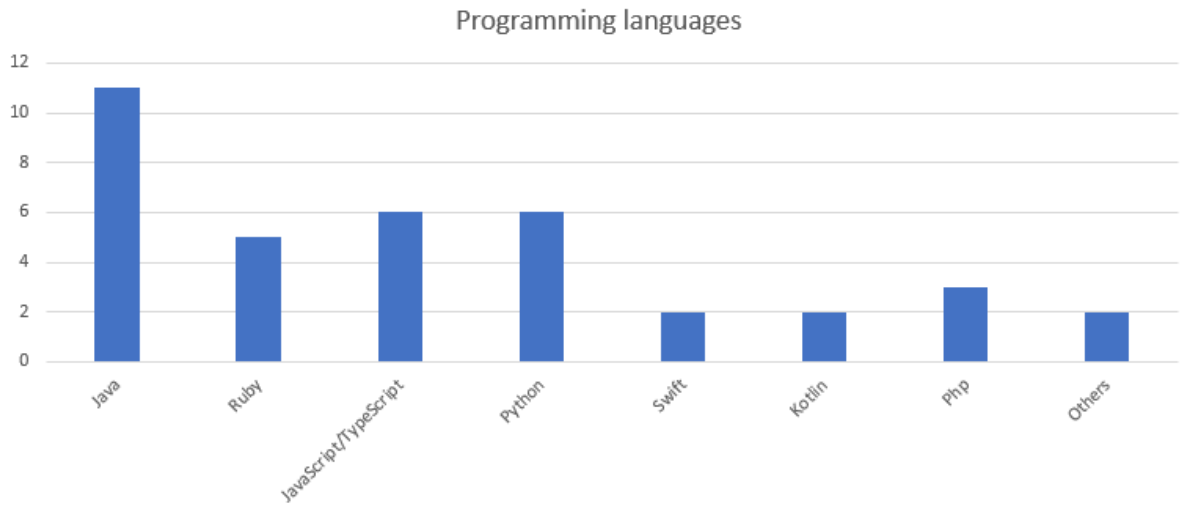

Figure 5.8: Answers to Question 3 of the interview

Dependency Management

The 15 interviewees have experience with dependency management. However, some of them indicated that it is not a task that they usually perform in their jobs, but rather in personal projects or time. Figure 5.9 shows the answers to question 6 regarding the importance of updating the dependencies. The reasons given by the interviewees answering *Neutral* and *Important* for not giving it more importance include: prioritizing the fact that the versions used are compatible, that there is no version incompatibilities with the current version used, and that the version used is stable.

Figure 5.10 shows the answers to question 7 about the importance of monitoring the dependencies' vulnerabilities. The interviewees considered that the importance of monitoring the dependencies' vulnerabilities was less than *Very important* reasoned about it. For example, some said that it is something that they do, but not regularly. They just update when there is a new version to ensure that the patch is always used if a vulnerability has been discovered. Finally, the last reason depends on the type of dependency — if it is not a customer-facing dependency, it is not that important.

In Figure 5.11, there are the answers to question 8. The interviewees gave more than one answer to the question, but always at least one explicitly included in the figure.

Finally, the answers to question 9, regarding how helpful are the tools that the interviewees use for dependency management. The interviewees who considered the tools to be really helpful (5) compared it to not using any tool. Whilst the interviewees giving lower marks (2-3), considered that features are missing. Mainly, they considered that the basic needs are covered. However, some more detailed information about how to manage the dependencies is not there.

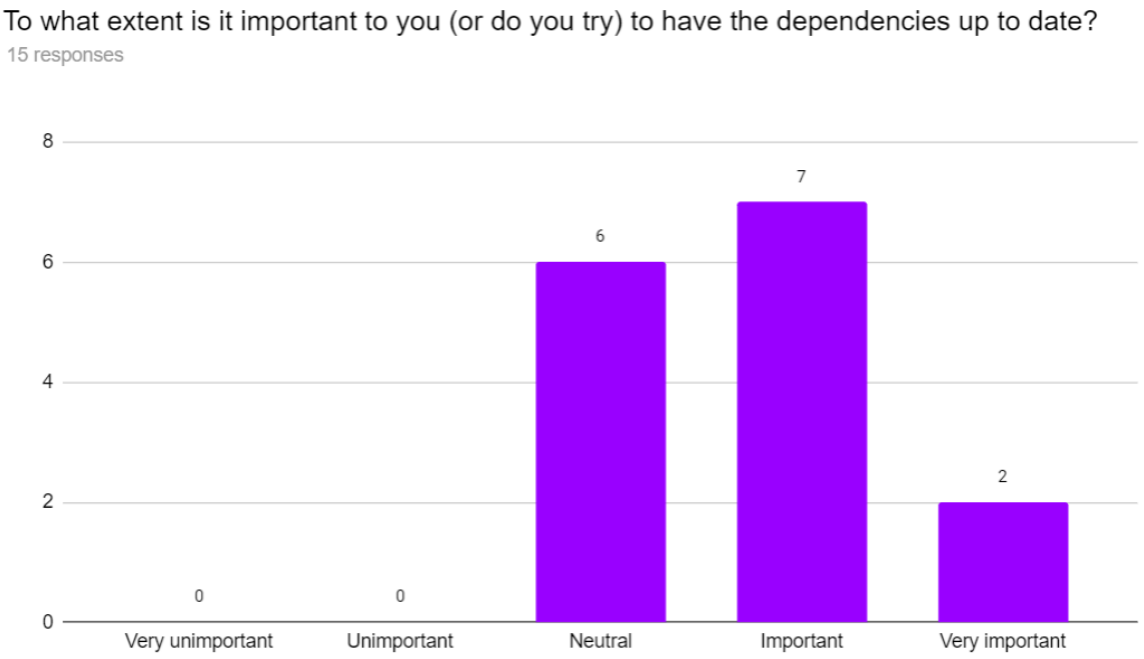

Figure 5.9: Answers to Question 6 of the interview

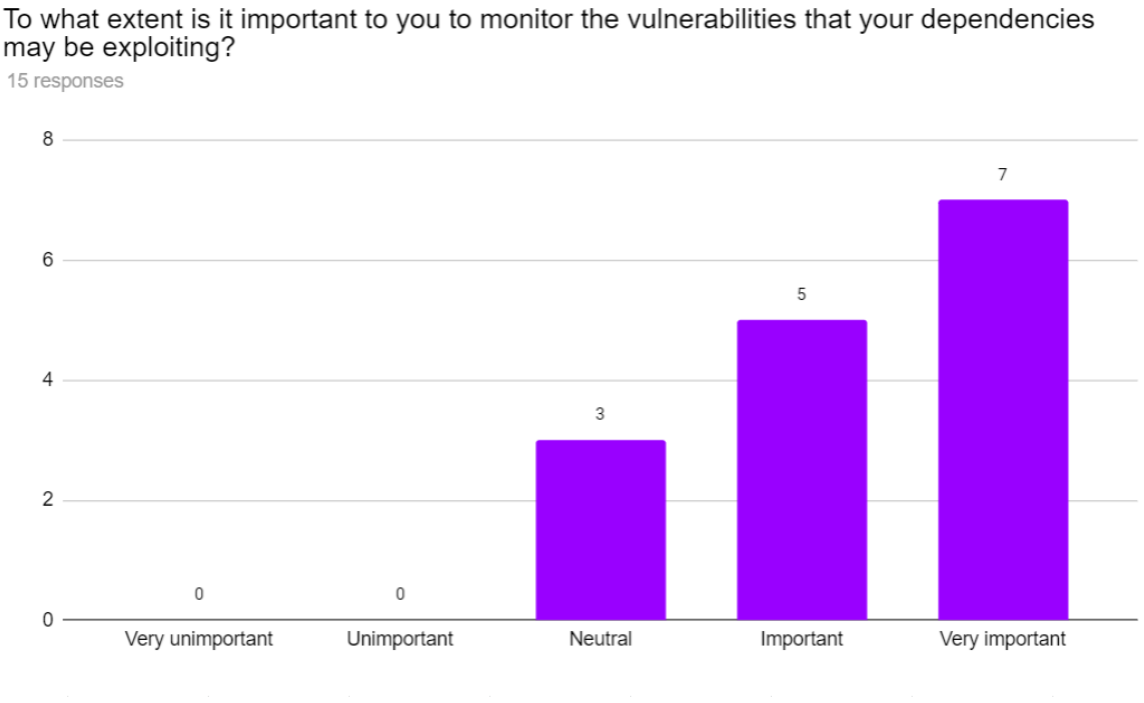

Figure 5.10: Answers to Question 7 of the interview

Visualizations

The interviewees' answers to question 10 about the tool's usefulness are displayed in Figure 5.13. Some of the interviewees' reasons for not giving it the maximum grade are the need for improvement in some aspects of both the visualization and the metrics and being useful for some particular scenarios.

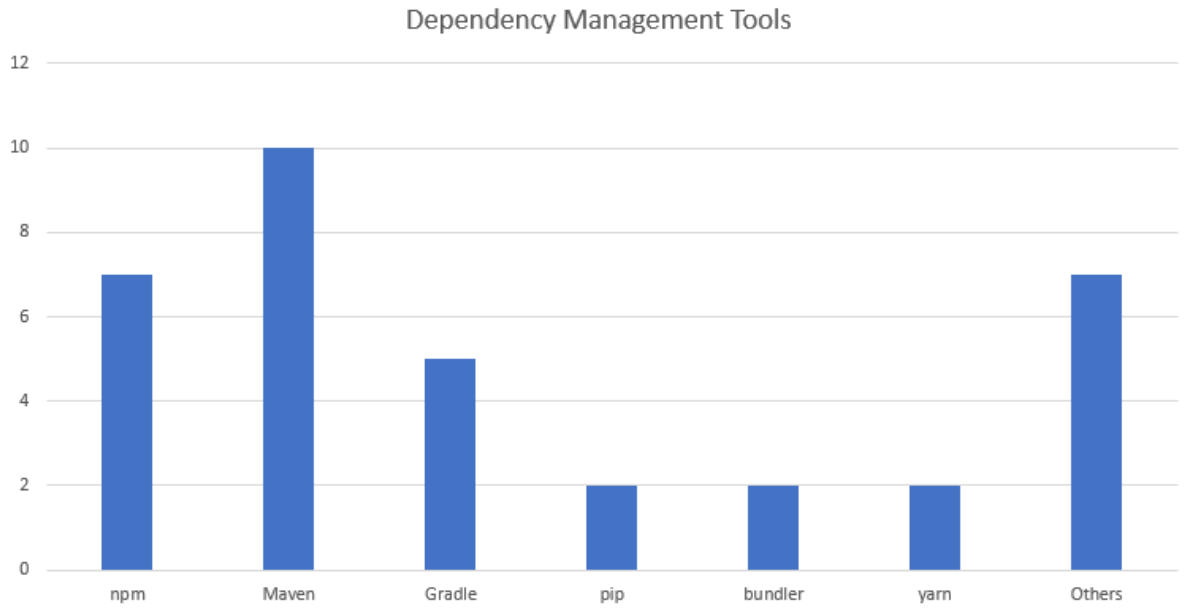

Figure 5.11: Answers to Question 8 of the interview

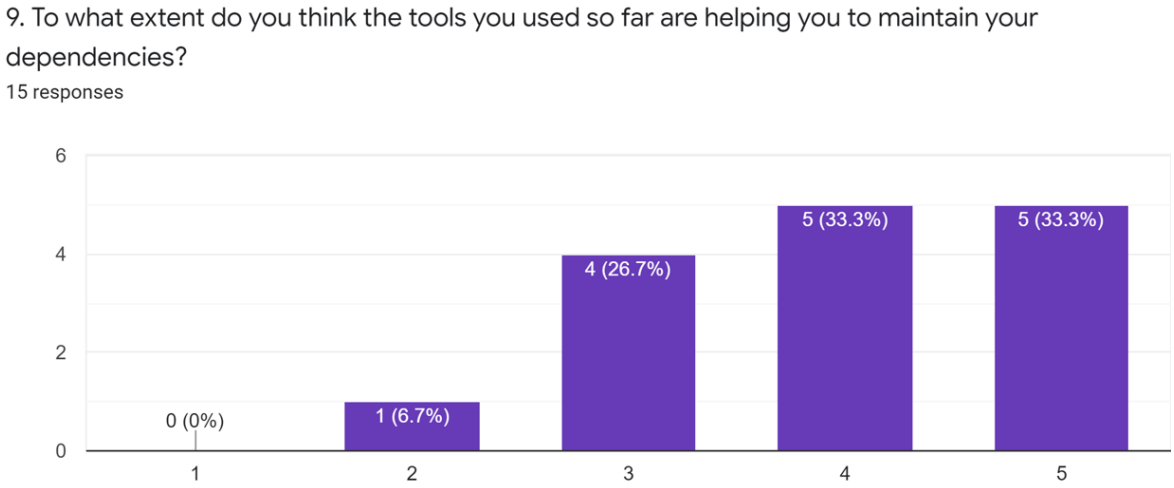

Figure 5.12: Answers to Question 9 of the interview

Figure 5.14 we can see the results for Question 11. In this case, the interviewees who answered *Disagree* or *Neutral* considered that the tool is meant for some particular cases, which are not very likely to be needed. The other interviewees agreed that the tool would probably not be used daily but would make some tasks easier, such as those discussed in the scenarios.

For Question 12, about which visualizations are most useful, most of the interviews answered more than one visualization. Table 5.8 summarizes the answers.

The answers to Question 13 are shown in Figure 5.15. Just as in Question 11, the reason given by the interviewees who answered *Disagree* or *Neutral* are that the tasks for which the tool is useful are not one of the regular tasks in their job.

With Question 14, the interviewees gave their suggestions for improvements to the current visualizations and completely new visualizations. The list of suggestions can be found below:

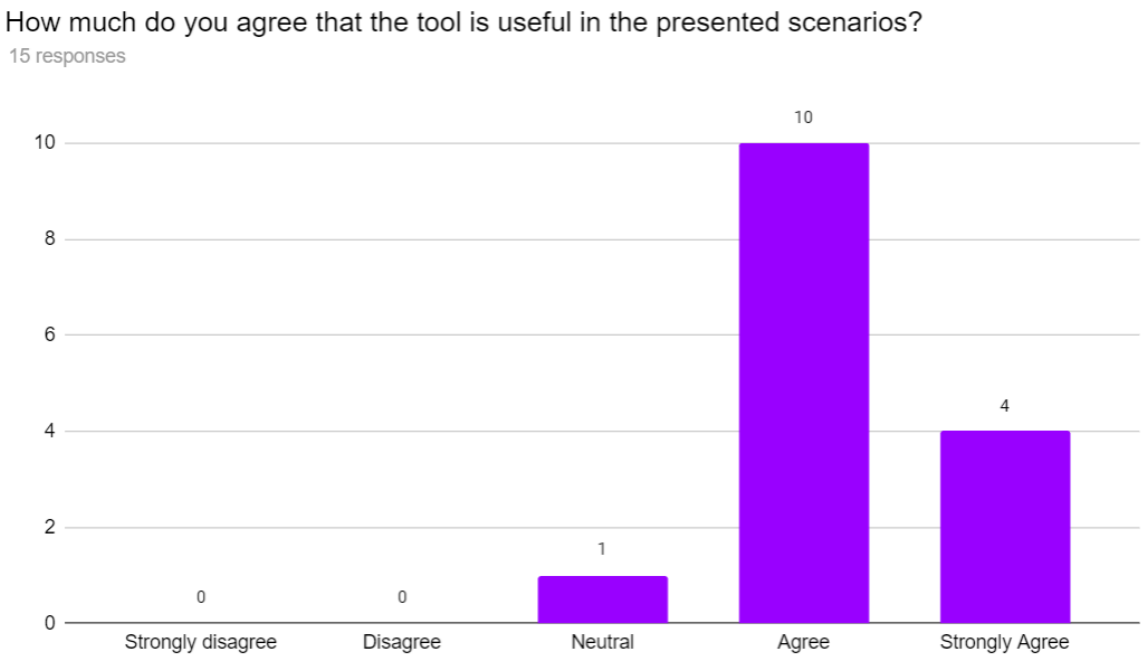

Figure 5.13: Answers to Question 10 of the interview

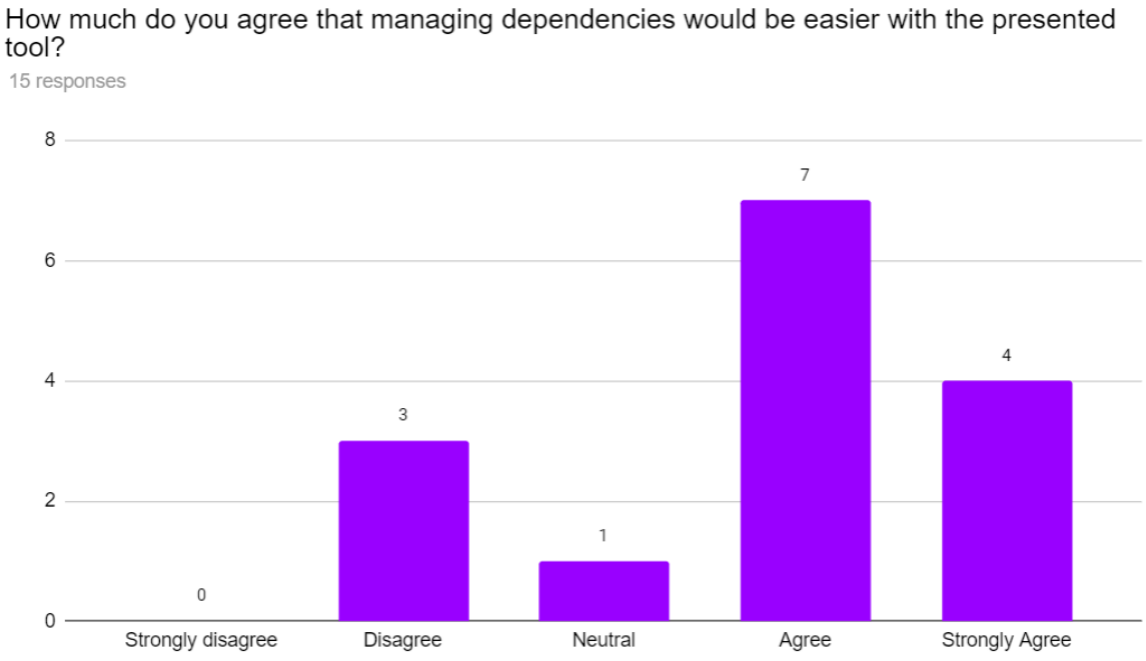

Figure 5.14: Answers to Question 11 of the interview

- Add tree visualization at the class level.
- List with the classes and methods used from a library.
- Add a decision-making model, indicating which actions should be taken.

Tree	Table	Barchart
⊙	⊕	
		⊕
⊕		⊕
⊕		
⊕	⊙	⊕
⊕		⊙
⊙	⊕	
⊕	⊕	⊕
		⊕
⊙	⊙	⊕
⊕	⊕	⊕
⊕		
⊕	⊕	⊕
⊙		⊕
⊕	⊕	⊕

Table 5.8: Answers to Question 12 of the interview. ⊕ indicates that the interviewee considered that visualization to be the most useful, the ⊙ is used when the visualization was considered useful but in a clear second position, and an empty cell means that the the visualization was not mentioned.

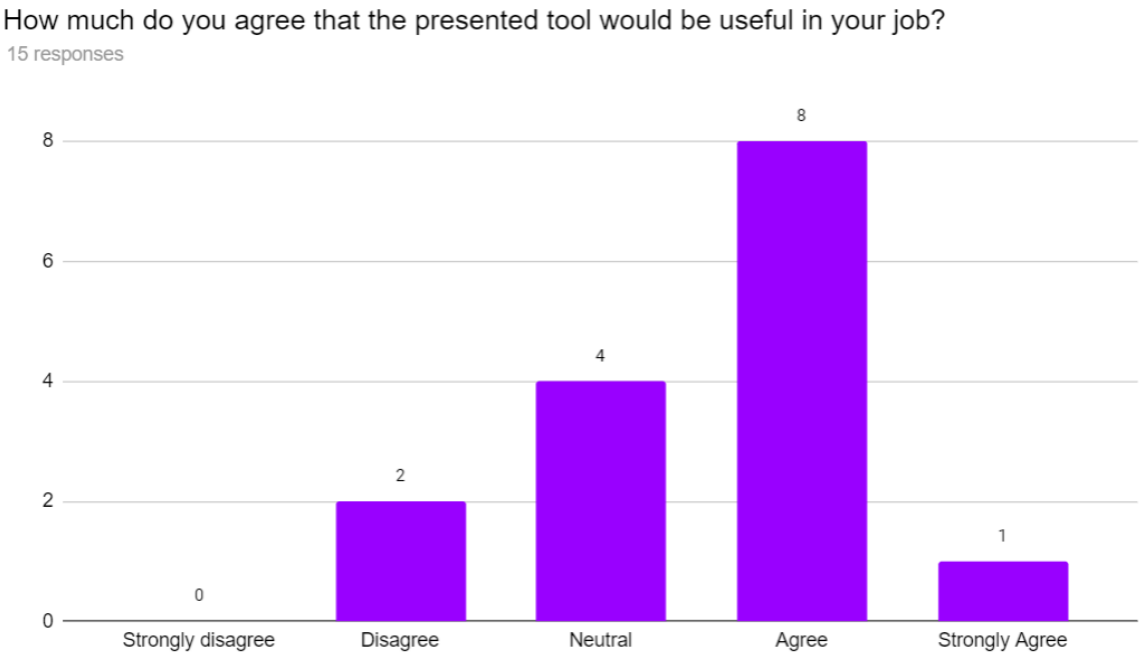

Figure 5.15: Answers to Question 13 of the interview

- Visualization to know how much of the system is depending on a library.
- Smart visualization displaying some potential problems: the freshness of the dependency, multiple versions of the same library being used.
- Color legend in the tree graph.

- Tooltip with a description of the metrics of the model.
- Display the licenses of the dependencies.
- Change the bar chart for a table.
- Possibility to move and reorganize the nodes of the tree.
- Turn the features into a command interface to be unattended running in the build pipeline.

Metrics

The answers to question 15 can be found in Figure 5.16; for each metric, the number of times an interviewee answered with each number of the options. In addition, Table 5.9 shows the average mark given to each metric, first considering only the marks given by developers, then by non-developers, and finally, the total average.

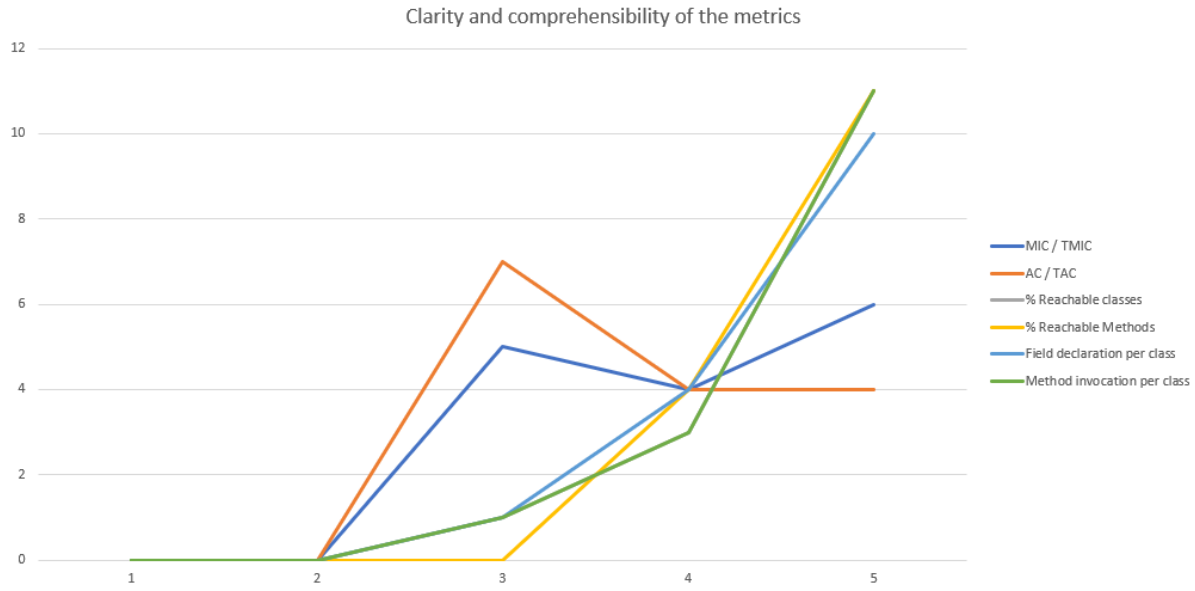

Figure 5.16: Answers to Question 15 of the interview

Metric	Average developers	Average non-developers	Total Average
MIC/TMIC	3.8	4.6	4.06
AC/TAC	3.6	4.2	3.8
% Reachable classes	4.6	4.8	4.66
% Reachable methods	4.8	4.6	4.73
Field declaration per class	4.8	4.2	4.6
Method invocation per class	4.8	4.4	4.66

Table 5.9: Results Question 15: Average marks of the metrics, given by developers, non-developers, and all

In Figure 5.17, there are the answers to question 16 of the interview. The reasons for not giving the metrics the best grade include that it would be more useful if the metrics suggested as missing in question 19 were included. It was also suggested that context is missing for some metrics (e.g., the absolute number of reachable methods and classes, risk assessment for the coupling metrics).

The answers to question 17 can be found in Figure 5.18. The interviewees who gave the grade *Neutral* reasoned that the metrics need some improvements to be truly actionable and that there is information that still can only be found in other places. The interviewees who answered *Agree* suggested that the model's metrics are a good starting point to know which actions to take to start with.

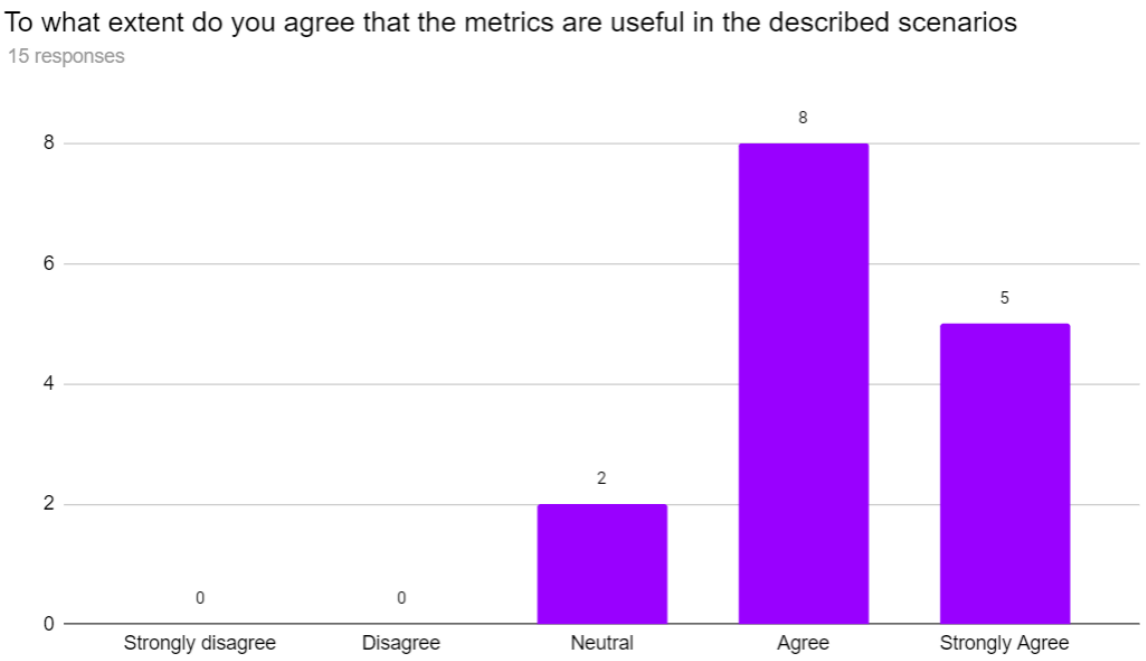

Figure 5.17: Answers to Question 16 of the interview

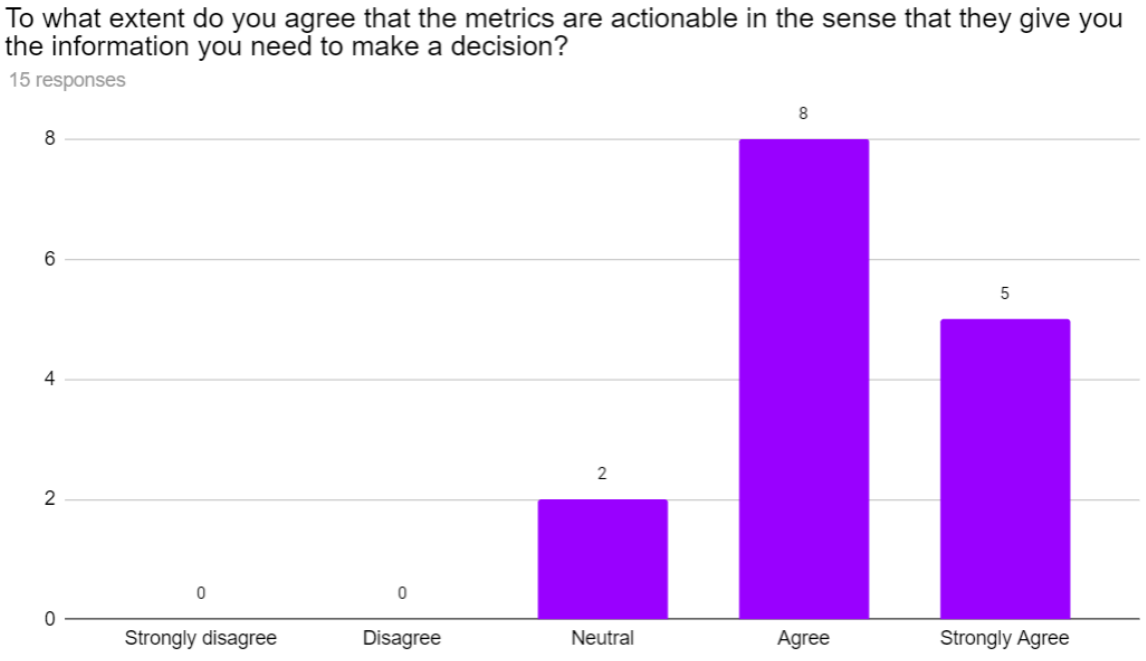

Figure 5.18: Answers to Question 17 of the interview

For question 18, the interviewees answered with which metrics they considered more useful. Their answers can be found in Table 5.10.

Finally, the suggestions made for improving the current metrics or adding new ones, in the answers to question 19, can be found in the list below:

MIC / TMIC	AC / TAC	% Reachable classes	% Reachable Methods	Field declaration per class	Method invocation per class	Role
⊕		⊕				Developer
		⊕	⊕			Developer
		⊕	⊕			Developer
				⊙	⊕	Developer
		⊕	⊕			Developer
		⊕	⊕			Developer
⊕		⊕	⊕			Non-developer
⊕						Non-developer
⊕	⊕	⊕	⊕			Developer
		⊕	⊕	⊕	⊕	Developer
⊕	⊕	⊕	⊕	⊕	⊕	Non-developer
⊕	⊕					Developer
⊕	⊕					Non-developer
		⊕	⊕			Developer
⊕		⊕				Non-developer

Table 5.10: Answers to Question 18 of the interview. ⊕ indicates that the interviewee considered the metric to be the most useful, the ⊙ is used when the metric was considered useful but in a clear second position, and an empty cell means that the the metric was not mentioned.

- Min, max, and mean of the coupling metrics.
- How much of the client is depending on the server library.
- A list of the reachable methods and classes.
- The absolute number of reachable methods and classes.
- Freshness indicator.
- Aggregate of the coupling metrics.
- Muber of files of the client using the server library.
- Lines of code of the client affected by the dependency.
- Code reuse: a combination of the metrics.

5.4.3 Discussion

In this section, we discuss the answers to the interviews, divided into sections. First, there is a discussion on the interviewees' general evaluation, according to their role.

We have found a difference in how the interviewees evaluate the model and the visualizations according to whether the interviewee's main focus is development or not.

The developers want more information that can be directly transformed into development actions. For example, they suggested seeing the list of reachable methods of the dependencies, which can answer

a vulnerable method is used or not. Another example is the lines of code where the calls to a dependency are made, so the developer can directly go to that line of code to make the necessary changes.

However, non-developers are interested in more high-level information. One of the suggestions made was to create a metric regarding how much of the client library depends on the server library. Also, how widespread the usage is in the client library. These two metrics would be related to the architecture of the system.

Furthermore, there is also an indicator of this difference in their answers to the preferred metric. The coupling metrics are more useful at an architectural level to understand how much a client depends on a server library. In Table 5.10, we see the the interviewees with non-developer role, always mentioned at least `MIC/TMIC` as the most useful metric. If we compare the average marks of the developers and non-developers in Table 5.9, we can see this same difference. The marks given by non-developers to the coupling metrics are higher than those given by developers.

Dependency Management With the questions answered in the *Dependency Management* section, we know that all the interviewees have experience with this type of task. Also, the interviewees considered it more important to monitor the dependencies' vulnerabilities than the update to the last version (see Figures 5.9, and 5.10). This is also confirmed by some interviewees' comments, saying that the main reason to update a dependency is to avoid being affected by possible vulnerabilities.

Visualizations When asked about the tool's usefulness, there is a general agreement that the tool is useful in the scenarios discussed during the interview, Figure 5.13. There are some negative answers to whether the tool can make dependency management easier. However, most of the comments about these answers are related to the tool being useful for particular cases. Therefore, the negative responses are not associated with the tool, not making the tasks easier.

> **Finding 5:** There is a consensus that the created tool is useful in certain scenarios related to dependency management.

In the answers to Question 12, only four interviewees answer only one of the visualizations (see Table 5.8). Some of the other interviewees also commented that there is value in combining the perspectives in the visualizations. Each visualization has a point of view, which adds to the value of the entire tool. The *Tree* visualization gives an understanding of the hierarchy of the dependencies and where the transitive dependencies come from. The *Table* visualization makes it possible to compare the metrics, and therefore, the degree of dependence. Finally, the *Barchart* gives a more detailed view of the impact of the dependencies in the client library.

> **Finding 6:** It is important to have a different perspective on a system's dependencies in the visualizations to have a complete understanding of the dependency tree.

With the last question about the visualizations, the interviewees suggested some improvements to be made. Certain suggestions are small improvements in terms of interaction with the visualizations and some other additions to make it easier to use. According to the research method, the next step would be implementing some of these suggestions and doing more interviews. However, due to time limitations, we cannot do these tasks.

Also, some interviewees expressed a need to integrate the different sources of information about dependency management. This data includes vulnerability data, the new versions available of the dependencies, and the licenses of the libraries used. Some related suggestions for making this tool's features into another type of tool are to make it an IDE Plugin and a command interface. Hence, there is a general interest in the tool, but it might be important to consider changing the format in which these are supplied.

Metrics Regarding the clarity and comprehensibility of the metrics, the interviewees' average grade for all the metrics is positive, being a 3.8/5, the lowest one, and a 4.73/5, the highest one, as shown in Table 5.9.

> **Finding 7:** There is a consensus that the metrics defined in this model are clear and comprehensible.

The metrics with the lowest grade are the coupling metrics. As confirmed by the comments on the model's actionability, most of the interviewees agreed that the coupling metrics are harder to understand since there is not a clear scale. Therefore, a number gives some information on the dependency itself and how it compares to the rest of the dependencies in the tree, but there is no clear meaning that indicates if a certain value is "good" or "bad."

Finding 8: The coupling metrics could be improved with a clear scale or rating evaluation.

The answers to Question 16, about the model's usefulness, are mainly positive (see Figure 5.17). Just as in the visualizations, the interviewees giving the lowest marks reasoned that the need to have such a detailed evaluation of the dependencies is not necessary daily.

Finding 9: The model is useful for dependency management. However, it is not needed for the most common and regular tasks.

With respect to question 18, regarding which are the most useful metrics in the model, the metrics that got more votes are the coverage metrics, as shown in Table 5.10. This result is consistent with the answers to Question 15, in which the coverage metrics have two of the highest average scores. Also, just as in the visualizations, some interviewees commented that it is the different perspectives given by each of the metrics making the model more useful.

In the last question of the interviews, we gathered suggestions about the model's metrics and can also be grouped into changes to current metrics and new metrics. One of the main suggestions obtained is to create a combination of the coupling metrics, have a general evaluation, and dive into the detail with the current metrics. The users would then have a general indicator, which can already point them to the dependencies that need a more detailed look.

Finding 10: There is an interest in a general metric indicating the general degree of dependency as a combination of the model's existing metrics.

We have also obtained some suggestions, which are modifications to the current metrics. For example, having the absolute number of reachable methods or classes and the list of their names. It should be evaluated, which is the need for this information, and in which cases it would be useful.

Finally, one of the suggestions to improve the coupling metrics' actionability is to create a risk profile of these metrics by evaluating the common values of these metrics and the outliers. This suggestion is further investigated in the following experiment.

Threats to validity Given the number of participants in the interviews, some other profiles which could also be potential users of the tool might not have been represented in this study. To avoid this threat, we have not limited the participants to the host organization of this thesis. Instead, we have included professionals from other organizations and backgrounds.

A common threat to validity when conducting interviews is the Hawthrone effect, according to which the interviewees act differently because they are being observed. To avoid this bias and prevent the interviewees from being biased towards evaluating the tool positively, each question was followed by a discussion. The interviewees could explain their opinion completely.

5.5 Experiment 5: Benchmarking

One of the main remarks received for the coupling metrics during the interviews is that there is no clear scale for those metrics. Therefore, the metrics' value can be hard to interpret since there is no indicator of which number is very high or very low.

Hence, we have benchmarked the values of the metrics `MIC`, `AC`, `TMIC`, and `TAC`. The goal is to understand, which is the distribution of the values and be able to indicate which are the outliers of these metrics.

5.5.1 Experimental set up

To execute this experiment, we have set a new request in the backend of the PoC. This request should contain the path to a *.txt* file, which includes three different columns, tab-delimited. Each row represents a Maven artifact, and for each column it indicates: *group id*, *artifact id*, and *version*.

Then, the calculation of the metrics of the model is performed. For each analyzed dependency, the value of the benchmarked metrics is stored. The result consists on 4 different *.csv* files. The first two contain all the different values of the `MIC` and `AC` metrics. The last files represent the values of the transitive coupling metrics, represented in three columns. The first, the dependency id, the second the distance, and the third the value calculated at that distance. Therefore, more than one row might be used to represent the `TMIC` or `TAC` of a dependency.

5.5.2 Results

We have analyzed a total of 299 direct dependencies and 470 transitive dependencies, obtained from analyzing a set of Maven libraries (see list of libraries in Appendix A). We filtered out zero values since there is no coupling in those cases, and we plot the rest of the values. In Figures 5.19 and 5.20, we show the histograms representing the distribution of the values of `MIC` and `AC` respectively.

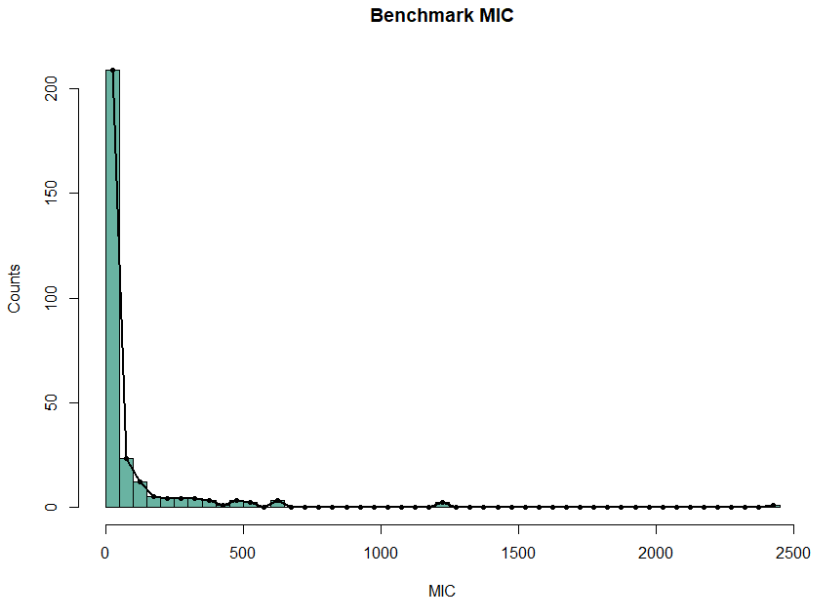

Figure 5.19: Histogram `MIC` benchmark, 60 bins

For the transitive metrics, namely `TMIC` and `TAC`, we have calculated the benchmarking with different values for the *propagation factor*. The histogram of the benchmarks for metrics `TMIC` and `TAC` with propagation factor 1, 0.5, and 0.1, can be found in Figures 5.21, 5.22, 5.23 and 5.24, 5.25, and 5.26, respectively.

Finally, we also generated 70th, 80th, and 90th percentiles of each of the metrics, which can be found in Table 5.11.

5.5.3 Discussion

In the results of all the metrics, one can see that the majority of the values are the lowest ones. In other words, many dependencies are loosely coupled, while there are a few which are highly coupled. Also, it is possible to observe a difference in the values of the metrics measuring method invocation coupling (`MIC` and `TMIC`), with those measuring aggregation coupling — namely `AC` and `TAC`. The method invocation metrics generally have higher values. This can also be intuitively perceived: for each field declared, multiple method calls can occur.

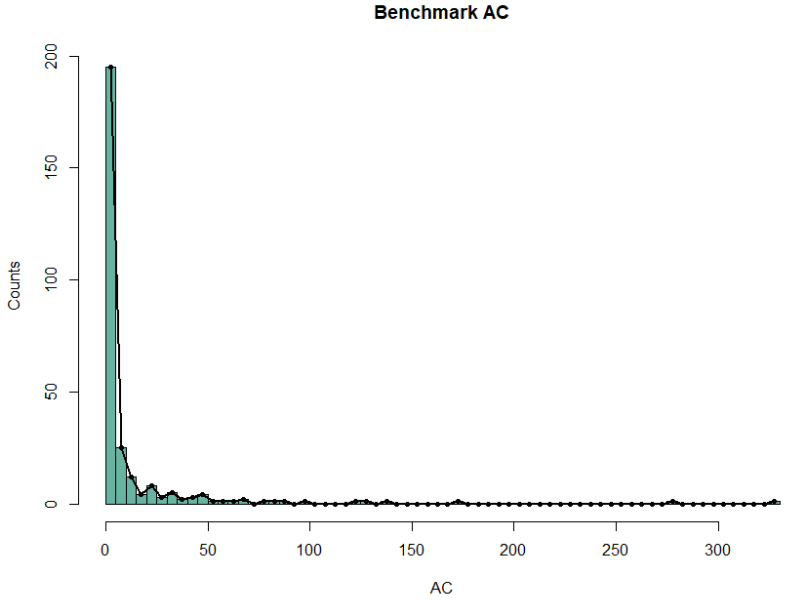

Figure 5.20: Histogram `AC` benchmark, 60 bins

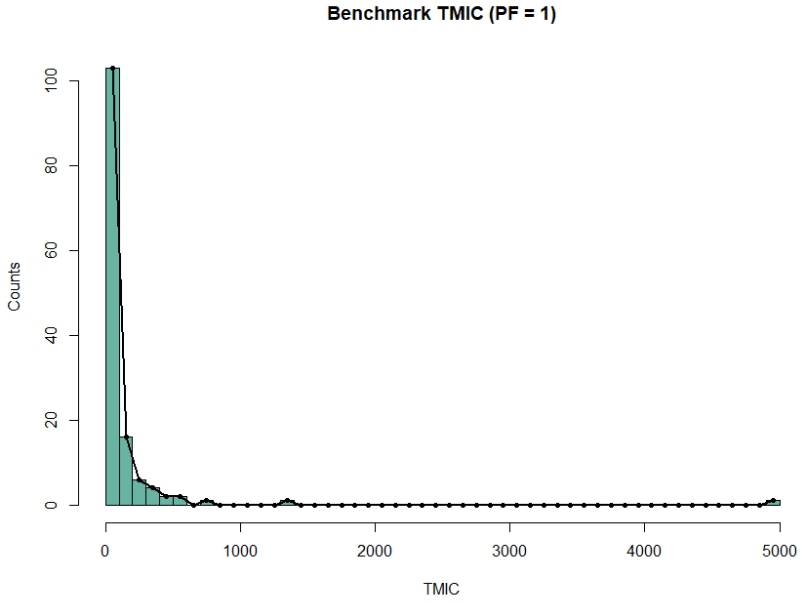

Figure 5.21: Histogram `TMIC` benchmark, propagation factor = 1, 60 bins

Finding 11: The four coupling metrics have a similar distribution: small values are common, while larger values are rare.

When comparing the results of the metrics `TMIC` and `TAC` with different values of *propagation factor*, the comparison is not exactly as it could be intuitively expected. In Figures 5.21 and 5.25 we can see the benchmark of `TMIC` with *propagation factor* 1 and 0.1, respectively. Given that one *propagation factor* is 10 times smaller than the other one, we would expect the results of the metrics to be 10 times smaller as well. However, we can see that on the right side of the plot; there is at least one dependency with `TMIC` around 5000 for both *propagation factors*. We looked at the values measured to understand why this

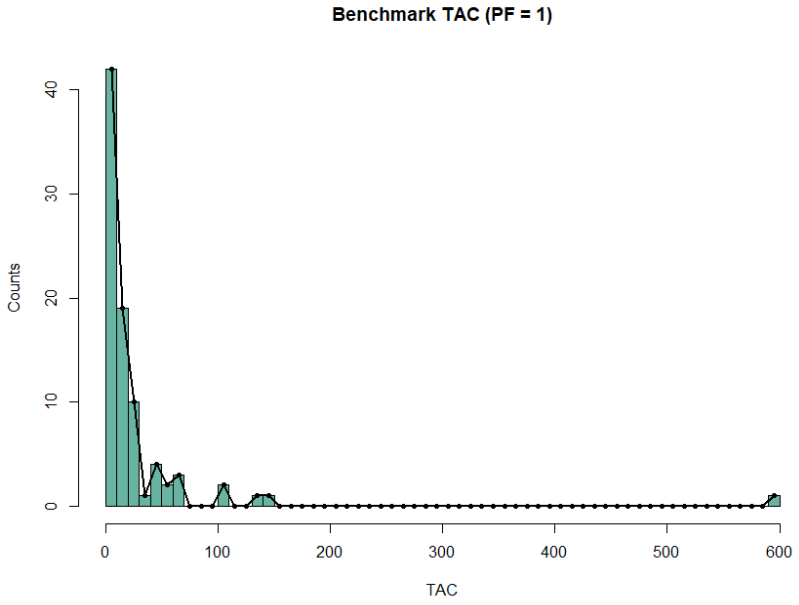

Figure 5.22: Histogram `TAC` benchmark, propagation factor = 1, 60 bins

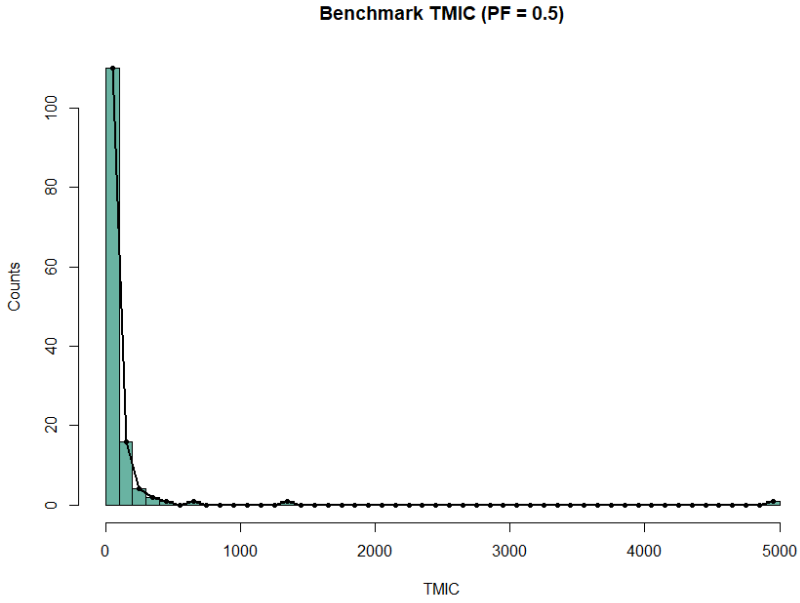

Figure 5.23: Histogram `TMIC` benchmark, propagation factor = 0.5, 60 bins

happened. The dependency has two distances at which coupling is measured: 1 and 2. These distances have a coupling of 4958 and 11, respectively. Following the equation 3.6, since the *propagation factor* is used to the power of `distance` − 1, it is not applied when distance is 1. This is also consistent with the fact that distance 1 means that the server library is, in this case, a direct dependency, and therefore there is no mitigation of the coupling. Hence, that is why this value appears with both *propagation factors* because it is only applied for distance 2, which has a small value, in comparison with distance 1. The same scenario can be seen in the case of `TAC` (Figures 5.22 and 5.26).

Furthermore, it is expected that with a lower propagation factor, the values of the metrics are pushed to zero. Therefore, the first bin of the graph should substantially increase. Looking at Figure 5.21, we can see that the first bin with 1 as the *propagation factor* is around 100. In Figure 5.23, we can see

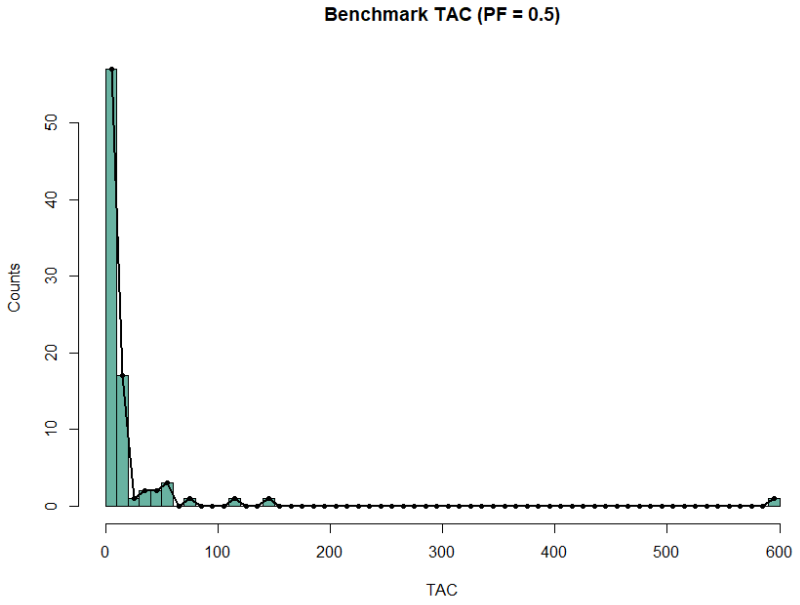

Figure 5.24: Histogram `TAC` benchmark, propagation factor = 0.5, 60 bins

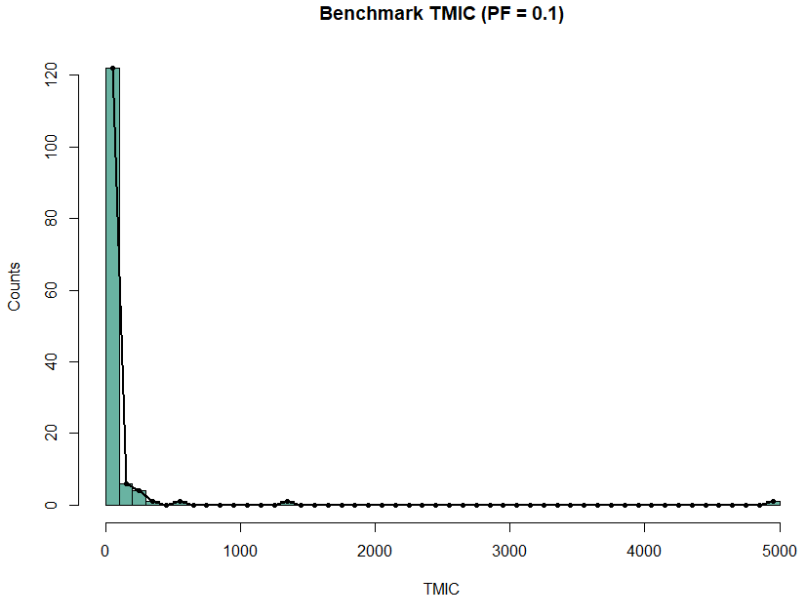

Figure 5.25: Histogram `TMIC` benchmark, propagation factor = 0.1, 60 bins

that the first bin increased to around 110, and in Figure 5.25, the count of the first bin is around 120. The increment is not as high as could be expected, considering the difference between the propagation factors. However, this is again due to some server libraries being at the same time direct and transitive dependencies; the coupling measured at distance 1 decreases the difference in the value of the metric. Nevertheless, the fact that a decrease in the *propagation factor* moves the distribution of the metric's values to zero can be seen in Table 5.11. In the table, we can see that the percentiles' values are closer to zero as the *propagation factor* decreases.

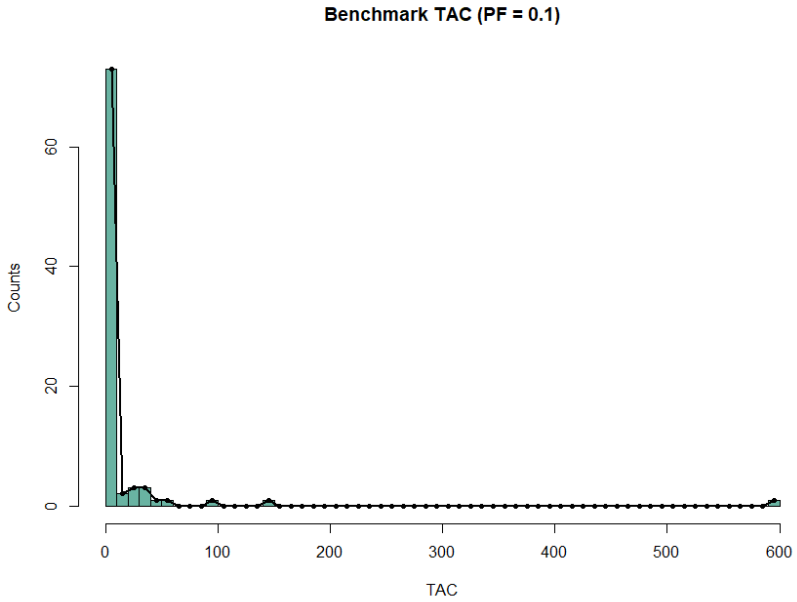

Figure 5.26: Histogram `TAC` benchmark, propagation factor = 0.1, 60 bins

Metric \ Percentile	70th	80th	90th
MIC	79.00	118.80	305.40
AC	21.50	35.00	61.00
TMIC (PF = 1)	59.50	129.00	215.00
TAC (PF = 1)	19.50	24.00	56.00
TMIC (PF = 0.5)	30.37	87.00	141.87
TAC (PF = 0.5)	10.50	12.25	39.12
TMIC (PF = 0.1)	11.40	33.00	105.70
TAC (PF = 0.1)	7.35	7.77	25.15

Table 5.11: 70th, 80th, and 90th percentiles of the coupling metrics

Finding 12: The values of `TMIC` and `TAC`, do not decrease as much as could be expected when the *propagation factor* is decreased. This is because of the libraries which are found in direct and transitive dependencies. The coupling created in the direct dependency remains equal regardless of which *propagation factor* is applied.

In Table 5.11, there are the 70th, 80th, and 90th percentiles of each of the metrics. These percentiles have been previously used for software metrics for risk assessment [40]. Considering the values in Table 5.11, the risk assessment for each of the metrics can be found in Table 5.12, with four risk profiles: low risk, medium risk, high risk, and very high risk. It is worth saying that the percentiles should be calculated with other datasets with different characteristics for this risk profile to be completely useful. Therefore, this risk profile is only valid for the current dataset.

Threats to validity Just as in the case of the experiment *coupling metrics significance*, when evaluating the results of this experiment is necessary to consider the dataset used. The results of the benchmark would change if a larger dataset were used. It is also worth mentioning that we have selected the client libraries to use based on the most popular libraries of the Maven Central Repository. However, it could be interesting to see how the results change if less popular or smaller client libraries are also included in

Metric \ Risk	Low risk	Medium risk	High risk	Very high risk
MIC	<79	<118.8	<305.4	>305.4
AC	<21.5	<35	<61	>61
TMIC (PF = 1)	<59.5	<129	<215	>215
TAC (PF = 1)	<19.5	<24	<56	>56
TMIC (PF = 0.1)	<11.4015	<33	<105.7	>105.7
TMIC (PF = 0.5)	<30.37	<87.00	<141.87	>>141.87
TAC (PF = 0.5)	<10.50	<12.25	<39.12	>39.12
TAC (PF = 0.1)	<7.35	<7.77	<25.15	>25.15

Table 5.12: Risk profile of coupling metrics

the dataset.

Chapter 6

Discussion

This chapter discusses the answers to the research questions formulated in this thesis and the process used to obtain these answers, and possible threats to validity.

6.1 RQ1: How can we measure the degree of code dependency between two software products with a direct dependency?

To answer this question, we have created the metrics `MIC` and `AC`. These metrics measure the coupling between the code of the client and the server library.

RQ1.1: What constitutes a dependency between two products? To define what constitutes code dependency, we discuss the meaning of coupling. To define coupling in the scenario of this research question, we use the framework created by Briand et al. [11] as described in Section 3.1.1. To correctly represent the code dependency scenario between two software products, we adapt the framework. For example, by adding a new aggregation level: the library level.

An important decision in this stage is to define two types of coupling to be considered: the method invocation coupling and aggregation coupling. As discussed in Section 5.2, these two types of coupling might not be sufficient to represent the coupling between two libraries accurately. As mentioned in Section 5.2, there are cases in which the two types of coupling metrics measured in this thesis might not be enough. This could be solved by analyzing which types of dependencies could add information about the dependency and design the metric.

RQ1.2: Which metrics can be used to measure the dependency? The answer to this question is described in Section 3.1.2. Based on the definition of coupling created to answer the previous question, we formally defined the metrics to measure the degree of code dependency for direct dependencies: Method Invocation Coupling (`MIC`), and Aggregation Coupling (`AC`).

RQ1.3: How can the proposed metrics be validated? To validate the metrics, we have taken different approaches. First, we have provided proof that both metrics fulfill the five properties of coupling metrics, defined by Briand et al. [23], which have been largely used in the literature. Then, from the set of validation criteria for software metrics described by Meneely et al. [22], we selected the actionability and clear definition. Professional developers have conducted the validation of these two criteria. In order to improve the actionability of these metrics, we have created a benchmark. This benchmark allows us to create a scale or a risk evaluation, which helps de users understand what a certain value for the metric means and how to react to it.

6.2 RQ2: How can we measure the degree of code dependency between two software products with a transitive dependency?

We have adapted the metrics designed for the previous question to measure code dependency of a transitive dependency. The result is the metrics Transitive Method Invocation Coupling `TMIC` and Transitive Aggregation Coupling `TAC`. These two metrics consider the distance between the client and the server library, and according to it, apply a *propagation factor*. The coupling's impact is reduced due to the libraries between the client library and the server library. This mitigation is modeled with the *propagation factor*. Moreover, the metrics `TMIC` and `TAC` use reachability to measure the coupling. Therefore, only the parts of the server library that are reachable are measured. We have provided a formal definition of both of the coupling metrics for transitive dependencies. However, the value of the *propagation factor* cannot be determined since we have not been able to measure the impact of coupling in the real world and how it behaves. The *propagation factor* may also change depending on why you are looking at the coupling metrics (e.g., vulnerabilities or breaking changes). Since we cannot set only one value for the *propagation factor*, we have conducted a sensitivity analysis, which indicated that the value of the metrics is highly sensitive to the value of the *propagation factor*.

The metrics are validated by proving that they fulfill the five properties of coupling metrics. And as the metrics for direct dependencies, these are included in the expert interviews to evaluate their clarity and actionability. Moreover, these two metrics also have a benchmarking. We have created it twice, one with *propagation factor* set to 1 and another one set to 0.1. This way, we have two extremes of the value, and we can see the differences.

6.3 RQ3: How can we measure how much of a dependency is used by a software product?

This question is answered by creating the coverage metrics: *Percentage of reachable classes* and *percentage of reachable methods*. These two metrics measure how much of a dependency is used by considering the reachable classes and methods compared to the total classes and methods of the dependency. The reachability is measured by considering all possible types of connections between the client and the server. Both of the coverage metrics, are formally defined in Section 3.2.

We have conducted a theoretical validation by proving a subset of software metrics' properties that apply to the coverage metrics. Moreover, these metrics were also evaluated by professional developers to validate their actionability and the clarity of their definition.

6.4 RQ4: How can we visualize the metrics designed to model the software dependencies?

To visualize the model created during the previous questions, we have added a front-end to the proof-of-concept tool, which contains three visualizations. The first one is a tree graph visualization, which shows the dependency tree's hierarchy, and the unused parts are easily identifiable. Each node of the tree displays the coupling and coverage metrics for the client library and the server library. The second visualization is a table visualization. It shows the data of the server library of each dependency and the metrics measured for each. The table allows the user to filter the dependencies to be displayed and sort according to any values of the table. Finally, there is a third visualization when a dependency is selected. This visualization shows the distribution per class of the server library usage by displaying the usage per class metrics.

This visualization has been validated by conducting expert interviews. The experts agreed that the tool and the visualizations are useful for specific scenarios. Moreover, when asked about the most valuable visualizations, the answers were diverse, indicating no clear favorite. Some of the interviewees said that the combination of the three visualizations is needed. The goal for the future work would be to improve the visualizations is to implement some of the changes suggested by the interviewees, and conduct interviews again in a more real-world setup.

When conducting the interviews, we also noticed that there are apparent differences in the opinions of the interviewees according to their roles. Therefore, a study of which perspectives are there, which are the needs of each user, and how to adapt the tool for them.

6.5 Proof-of-Concept

To give a complete answer to the previous questions, we have created the proof-of-concept tool to calculate the metrics of the model for an entire dependency tree, given a client library. The tool works with Maven libraries, obtaining the *.jar* files from the *Maven Central Repository*. This decision restricts the type of software product which can be analyzed. Moreover, it excludes the testing dependencies from the analysis since the tests are not included in the sources that can be obtained from Maven. However, using libraries available in Maven made possible the comparison with the results of the paper by Soto-Valero et al. [7].

Another decision was to do bytecode analysis, which, as has been already discussed, can be obtained more often from the *Maven Central Repository* than source-code. However, bytecode analysis has its limitations. The main one that we have encountered while developing the proof-of-concept is that the declaration of variables cannot be found in the bytecode. Declaration of variables is one of the types of connections defined by Briand et al. [11], and therefore should be considered in the calculation of the coverage metric *Percentage of reachable classes*. Nevertheless, the possible usages of a variable can be detected using bytecode. For example, if a method is called on the variable, the variable is declared based on the return of a method call, or if the variable is sent as a parameter on a method call. All these usages are detected by the bytecode and included in the calculation of the metric. Since the coverage metrics do not consider how many times a class or a method has been reached, if a local variable is used in any way, the type of the variable is counted as reached.

6.6 Limitations

The metrics The metrics we created in this thesis have been validated theoretically, with the properties and the clarity and actionability evaluated during the interviews, and empirically with the experiments. However, the empirical validation should not be considered as a complete validation of the metrics since an evaluation of how the metrics correlate with other aspects such as the maintainability is not conducted. This is because the data collection was not feasible in the context of this thesis, as has been explained in section 5.2. Nevertheless, we conducted other types of empirical validation to make it as complete as possible.

The coupling metrics we have defined in this thesis focus on two different types of connections: the method invocations and the field declarations. For the transitive dependencies, the metrics consider chains of these two types of connections and not chains of combined connections. Therefore, the coupling created by mixed connections is not considered. This is because the impact of the coupling created by each type of connection might be different, and we decided to focus on these two. Nevertheless, the combination of different types of connections is considered for the coverage metrics. For the coverage, it is not relevant how much of the coverage was created by each type of connection.

The dataset For the experiments conducted in this research, we have analyzed the dependency tree of Maven libraries and obtained the files from the Maven Central Repository. We have to consider that the results obtained in this research might be affected by these decisions. The files obtained from Maven contain production code only. It might be interesting to see the results if the non-production code was also included in the analysis, not only testing code but also any other code that is not included in the final product. However, the need to include non-production code in the analysis depends on the goal of the analysis; if the goal is to understand the impact a certain vulnerability might have, maybe non-production code might be directly excluded [8]. In addition, the results of the experiments (e.g., the benchmarking) could be different if proprietary software was analyzed, as well as using applications or other systems as clients instead of libraries. We think it could be valuable to use the tool with a different type of dataset to understand the differences.

Chapter 7

Related Work

In this chapter, we present the work related to the topic of this thesis. As far as we have been able to find, there are no papers that propose a way to model dependencies between two software products. However, there is related work in the area of dependency management and software ecosystem modeling, as well as in the topic of coupling metrics. We present the related work divided into two sections. For each topic, we discuss the papers and summarize the comparison of all the papers at the end of the section.

7.1 Software dependencies

In Dependencies We Trust: How vulnerable are dependencies in software modules? [41] The thesis investigates how the vulnerabilities in npm packages create a cascading-effect in the JavaScript ecosystem and how the vulnerabilities are fixed. This is done by studying the vulnerable packages' dependency chain through the packages that depend on these. One of the main contributions of this research is the tool used to find the dependencies between the packages, to determine the impact of the vulnerabilities in the ecosystem, `Rastogi.js`[1]. The research determined that although only a 1% of the modules are vulnerable, and that the dependency chain increased the number of vulnerable modules a 39.36%.

However, in contrast with the dependency metrics developed in this thesis, determining whether a package is affected by a vulnerability with `Rastorgy.js` is binary evaluation, performed by looking if the package is included as a dependency or not. Therefore, there is no confirmation of whether the dependency is used, or if the part containing the vulnerability is being exploited, since there is no fine-grained evaluation of the dependency. In the conclusions, Hejderup states *"On the other hand, reports of a vulnerable dependency are not an immediate sign of a security weakness in a module. There are several factors to this: the module is used in a development environment, the vulnerable functionality of the dependency is not used, or there is a little risk that the vulnerability can be triggered."*. This sentence points out the need for a more detailed analysis of the usage of the dependencies.

Impact Assessment for Vulnerabilities in Open-Source Software Libraries [42] Plate et al. create an approach to analyze whether an application, depending on a library that contains a vulnerability, is affected by it or not. Their methodology is meant to help assess the need to update the application with a version that does not use the library's vulnerable version.

The methodology consists of comparing the parts of the library used by the application with the parts updated in the library patch that fixes the vulnerability. It is assumed that those are the parts of the library containing the vulnerable code. The parts of the library used by the application are defined based on a dynamic analysis of the application and the bundled libraries.

In this work, the authors refer to application in the same way we have been using client library. Again, this work focuses only on the impact of vulnerabilities and how to fix them, while this thesis measures the degree of dependency. Moreover, instead of focusing on static analysis as in this thesis, they combine it with dynamic analysis to determine if the application uses the part of the library containing the vulnerability. Nevertheless, these two approaches can be combined to obtain more information about the dependency: while static analysis indicates how the code of the client uses the dependency, the

[1] https://github.com/jhejderup/rastogi.js

dynamic analysis shows the actual execution, as well as how many times is each part of the application, and the dependency is executed. Just as in this thesis, Plate et al. use *Javassist* for the analysis, since for their proof-of-concept, they focus on Java.

Modeling Library Dependencies and Updates in Large Software Repository Universes [43] In this paper, a model for library dependencies is created. The model is classified as a graph-based Software Universe Graph. It is focused on the updates of the dependencies and shows metrics such as the *wisdom-of-the-crowd*. In addition, it is extended to describe the *adoption-diffusion* and the *co-dependency*. The *adoption-diffusion* studies how the migrations to newer versions of a library are done. The *co-dependencies* are libraries which are usually employed together in an application. This last metric is used to compare different super repositories, Github, and Maven.

In contrast with this thesis, the model is not meant to give information about a particular dependency, but rather indicate the most popular libraries and versions, and how it changes. Therefore, the analysis done to determine whether there is a dependency is not fine-grained but binary. Just as in this thesis, Kula et al. focus on Java, particularly the Maven ecosystem.

PRÄZI: From Package-based to Precise Call-based Dependency Network Analyses [5] Hejderup et al. create an approach to generate fine-grained call-level dependency networks: *Präzi*. This approach evaluates more precisely the dependencies between software products than a package-level approach. As part of their work, the authors create an implementation of *Präzi* for the Rust's *Crates.io* ecosystem, called *RustPräzi*.

With *RustPräzi*, and to demonstrate how effective the approach is, they perform two case studies. First, a case study that focuses on the propagation of the vulnerabilities across software products, in which they prove the higher accuracy of their approach in comparison to a package-level dependency network. The second case study looks at the impact of deprecation in a library. In this case, with *Präzi*, they can perform an impact analysis of cleaning up deprecated functions.

Hejderup et al. use the word package in the same way we use library in this thesis. Moreover, instead of using the Maven ecosystem for their research, as we have done, and most of the other papers of the domain, they use Rust and *Crates.io*. Furthermore, to create the call-graph of each package, instead of doing a custom analysis, they use a previously existing tool (*LLVM*[2]).

Software ecosystem call graph for dependency management [44] In this paper, Hejderup et al. propose extending dependency networks with a versioned call-graph. The authors describe the algorithms used to create the network and how to perform an impact analysis of the changes in a library. Their technique uses the commit time of the products to resolve the dependencies' version ranges, aiming not to miss dependencies with previous library versions. The authors evaluate their dependency network by generating an impact analysis of a particular security bug. The output of the analysis is the part of the dependency network that is impacted by the bug.

Therefore, in this work, Hejderup et al. propose a different approach to create call-level dependency networks, which considers each library's historical data. The focus of the research is an impact analysis of changes and bugs. Nevertheless, the dependency network could also have other applications, such as calculating the metrics proposed in this thesis. The prototype created by Hejderup et al. is focused on JavaScript and the *npm* ecosystem. In this case, to create the call-graph, they used *Jalangi*[3], which uses dynamic analysis.

A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem [7] Soto-Valero et al. conducted a study of the unused dependencies included in other libraries' dependency tree in the Maven ecosystem. The research includes every type of dependency: direct, transitive, and inherited from the parent module.

To conduct this research, Soto-Valero et al. implemented `DepClean`[4]. The tool analyses the dependencies of an artifact by creating a call graph with the libraries involved to define if a dependency is used or not; it is done through bytecode analysis.

Just as this thesis, Soto-Valero et al. focus on Java and the ecosystem of Maven Central. The paper includes a comprehensive study of the unused dependencies in the Maven ecosystem, reported that 75.1%

[2]https://github.com/llvm/llvm-project
[3]https://github.com/Samsung/jalangi2
[4]https://github.com/castor-software/depclean

of the analyzed dependencies were unused dependencies. Nevertheless, the evaluation of the dependencies is still binary, but the strategy used to determine the existence of the dependency is fine-grained. This fine-grained analysis uses bytecode-analysis, just as the proof-of-concept developed in this thesis.

7.1.1 Summary

In Table 7.1, one can find a summary of the comparison between the different papers previously discussed in this section. The papers are first compared according to the terminology they use: library, package, artifact, or application. Then, according to the language and ecosystem used in their prototypes. Next, depending on the type of evaluation done of the dependencies, either binary or fine-grained, and which type of analysis was done. Finally, we also compare the goal, which was the main focus of the paper.

		[41]	[42]	[43]	[5]	[44]	[7]
Terminology	Library		x	x		x	
	Package	x			x		
	Artifact						x
	Application		x				
Ecosystem	Java (Maven)		x	x			x
	JavaScript (npm)	x				x	
	Rust (Crates.io)				x		
Evaluation	Binary	x		x			
	Fine-grained		x		x	x	x
Analysis	Source-code						
	Bytecode		x		x		x
	Dynamic		x			x	
	Dependencies file	x		x			
Goal	Dependency model			x		x	
	Impact vulnerabilities	x	x		x		
	Impact deprecation				x		
	Impact changes					x	
	Unused dependencies						x
	Library popularity			x			

Table 7.1: Summary comparison, software dependencies related work

As one can see in Table 7.1, the basic terminology, when researching about software dependencies, changes from paper to paper. The reason is that the terms *Library*, *Package*, and *Artifact*, can be used more or less with the same meaning. The difference is that some of these terms are specific for certain software ecosystems (e.g., Maven has artifacts). And the term *Application* is used to denote the software product that uses a library. In the context of this thesis, we have used client library and server library instead. The most used software ecosystems are npm for JavaScript and Maven for Java. This might be because these are two of the largest package repositories. This is precisely the reason for which Hejderup et al. [5] decided not to use these two and use *Crates.io* instead.

In evaluating the dependencies, we can see that there is interest in making it more precise. The papers that do a binary evaluation do it by parsing the file where the dependencies are declared and perform the same evaluation as the package manager does. The fine-grained evaluation is done with bytecode analysis or dynamic analysis. In the goals of the papers, we can see that there is interest in determining via a fine-grained evaluation, whether a vulnerability in a library is affecting the client that depends on this library. The same happens for deprecation and changes in a library.

7.2 Coupling metrics

Defining metrics for software components [45] Vernazza et al. create a set of metrics for software components. In the paper, software component is defined as a set of classes, similar to how we treat a library as a set of classes to define the coupling metrics. The set of metrics defined in the paper is based on the set of metrics by Chidamber and Kemerer, which include the coupling metrics *CBO* and *RFC*. To validate the metrics created, they use a theoretical approach by using the properties defined by Briand et al. [23], just as we did for our coupling metrics. In addition, the authors manually calculate the value of the metrics for various architecture patterns to illustrate their usage.

Detecting Indirect Coupling [25] Yang and Berrigan propose a way to measure indirect coupling, which is different from the one suggested by Briand et al. [11], which consists of a transitive closure of the direct coupling. Consider transitive coupling created by more than one type of connection, which differs from the transitive coupling metrics defined in this thesis, focusing on a kind of connection. The authors describe the concept of *use-def* indirect coupling, which is used to identify coupling between the definitions of variables (*def*) and the place where these variables are used (*use*).

Although there is no theoretical validation of the *use-def* indirect coupling, a tool is created to measure this type of coupling in Java projects using source-code analysis. The tool is used to measure the *use-def* indirect coupling in several projects, including the tool itself. With the tool, the authors validate the actionability of *use-def* indirect coupling by relating it to design issues and problems encountered during development.

Measuring Indirect Coupling [46] This thesis by Yang continues with the work done in the last paper [25]. In the thesis, Yang defines a set of indirect coupling metrics based on the *use-def* indirect coupling. In addition, the relation between the metrics and the maintenance effort is also modeled.

The metrics and their relation with maintenance are empirically validated by using a tool developed for this thesis. The tool uses static-code analysis to calculate the set of metrics. First, they calculate the metrics for a corpus of Java projects to see their distribution. The results obtained are similar to those obtained by us in the coupling metrics benchmarking: many loose coupled cases and a few with higher coupling.

The second part of the empirical validation consists of experiments to compare the value of the metrics for different software products and the time the participants needed to resolve certain tasks.

Deriving Coupling Metrics from Call Graphs [47] In this paper, Allier et al. use call-graphs to measure existing coupling metrics. Using call-graphs helps to improve the precision of the measurement since dynamic features used by a system can be accounted for. In particular, they focus on how are polymorphy and dynamic class loading accounted for. Allier et al. use the coupling metrics *CBO* and *RFC* (see Section 2.3), and modify them to account for the dynamic features. The metrics are calculated using call-graphs generated with different techniques. The results show that the call-graph used to calculate the metrics has an important impact on the values.

In this thesis, we have also created our own strategy to account for polymorphy in our coupling metrics. However, we have not included the detection of reflection constructs in the analysis. Using call-graphs to calculate the metrics designed in this thesis could help improve the precision of the measurement. Nevertheless, we would need to generate call-graphs for the entire dependency tree and combine them to create a general call graph. This is not currently available since it is a work in progress within the *FASTEN project*, but it is part of the future work.

Using indirect coupling metrics to predict package maintainability and testability [48] In this paper, Almugrin et al. calculated and validated the new indirect coupling metrics they defined in previous work [49], which are based on the metrics by Martin [50]. The metrics are empirically validated by showing their relationship with maintainability and testability. The experiments done for this validation used previously defined maintainability and testing metrics to compare these metrics' values with the values of the indirect coupling metrics. To run the experiments, they used a set of existing tools and one custom tool to obtain all the necessary data and calculate the values of all the metrics for the seven systems involved in their experiment. The authors calculate the linear regression and correlation between coupling metrics and the maintainability and testability metrics. The results show that the lower values of the coupling metrics tend to have better results for maintainability and testability.

7.2.1 Summary

The summary of the comparison of the papers included in the coupling metrics related work can be found in Table 7.2. The papers are compared according to whether there are new metrics defined or not and if these metrics account for indirect coupling, and the type of coupling measured. Then, we compare the technique used to measure the metrics and the type of validation conducted in the paper.

		[45]	[25]	[46]	[47]	[48]
Defined new metric(s)	Yes	x	x	x		x
	No				x	
Indirect coupling	Yes		x	x		x
	No	x			x	
Type of coupling	Method invocation	x			x	
	Field access	x			x	
	use-def		x	x		
	N/A					x
Measurement technique	Source-code analysis		x	x		x
	Call-graph analysis				x	
	Manual calculation	x				
Validation	Theoretical	x				
	Empirical		x	x	x	x

Table 7.2: Summary comparison, coupling metrics related work

In Table 7.2, it is possible to see that most of the papers included in the related work create new coupling metrics, two of them by adapting previously existing metrics [45, 48] and the other two are focused on a type of coupling which has not been measured before: *use-def* [25, 46]. Allier et al. [47], instead of creating new metrics, evaluate how much the value of the metrics change, depending on which technique is used to calculate them, they compare different call-graph generators based on how they deal with inheritance. There is source-code analysis in the related work about coupling metrics, which was not used in any of the papers related to software dependencies. This seems to be because the papers about coupling metrics analyze only one software product at a time. In contrast, the papers about software dependencies analyze dependency trees or even entire ecosystems, making other options such as bytecode or dynamic analysis a better option. Four out of the five papers do an empirical validation of their work. Two of the papers validate their metrics by looking at the effect these have in terms of maintainability [46, 48]. Yang and Berrigan [25], compare the places of their tool with *use-def* coupling, with the places where they had issues during development, and finally, Allier et al. [47] compare the results obtained with different call-graphs generators.

Chapter 8

Conclusion

In this work, we have created a model for the dependencies created when using a library. The model contains three types of metrics. The first one, to measure the degree of code dependency, is the coupling metrics. We have defined coupling metrics for direct and transitive dependencies, considering the coupling's reachability and propagation. Then, we have created coverage metrics to measure how much of a library is used, at the method and class level. Finally, we defined usage per class metrics to see which classes of the client are using the library and how much. We have provided a formal definition of each metric and conducted a theoretical validation of the metrics by proving the properties the metrics should have. These properties have been obtained from the literature.

To provide empirical validation of the metrics, we have created a proof-of-concept tool, which calculates the metrics for every dependency of a given client. The tool uses libraries available in the *Maven Central Repository*, which allowed us to validate the implementation by comparing the results with the literature. This comparison indicated some possible improvements, such as changing the way the bytecode is obtained, to be able to include testing dependencies, and avoid issues with shadowed dependencies.

Furthermore, we have conducted an experiment to evaluate the significance of the coupling metrics. This experiment indicated that the current metrics are sufficient to indicate whether there is coupling with a dependency in about 95% of the cases. Additionally, it also indicated that annotations are a special case in the context of software dependencies, which would need a specific metric to be measured.

We have also conducted a sensitivity analysis of the propagation factor used in the coupling metrics for transitive dependencies. The propagation factor represents the mitigation done by the libraries between the client and the server library. Our results indicate that the value of the metrics for transitive dependencies is highly sensitive to the propagation factor. Furthermore, we have created a benchmark of the coupling metrics to define a scale and risk evaluation of these metrics. We have found that all the coupling metrics' distribution is similar; most of the cases have low values, but there are a few with very high values.

Finally, the proof-of-concept includes three visualizations of the model. These visualizations have been validated with expert interviews. During the interviews, the experts evaluated the tool and its visualizations positively and made some suggestions for improvement. Moreover, the model's actionability and clarity were also assessed during the interviews, with a positive response to both aspects.

8.1 Future work

This section presents the future work that could be done based on the research done in this thesis. There are many other topics to be researched in the domain of software dependencies, but we focus on the model and the proof-of-concept we developed.

8.1.1 The model

The current model has had positive reactions from the professional developers interviewed during the thesis. Nevertheless, some improvements could be made. First, the coupling metrics of the model currently measure method invocation coupling and aggregation coupling. However, there are other types of connections that could be added to the set. In particular, as discovered with the significance

experiment results, the first type of connection to be considered is the usage of annotations. A metric to measure the coupling created by the usage of annotations could be used in combination with the current coupling metrics for a more precise measurement of all types of dependencies. A second step to improve the coupling metrics is to improve the benchmarking. Analyzing new client libraries to extend the benchmark data and add types of clients other than libraries could improve the quality of the benchmark and the risk evaluation of the metrics. A comparison of the results obtained from analyzing libraries and analyzing other types of software products is also possible.

In addition, during the interviews, it was suggested to create a general metric, combining the values of the current model's metrics. The general metric could give an overview of the state of the dependencies and indicate the developers which dependency needs more focused. However, the model would still contain all the other metrics to give more detailed information with the perspective of each metric.

Finally, as explained at the beginning of the thesis, another possibility is to add metrics based on dynamic analysis. Adding dynamic analysis could give more information on the actual usage of the client and the server libraries. It would allow us to validate the findings of the static coupling metrics. Dynamic analysis has already been used to evaluate if a client is affected or not by a vulnerability [42].

8.1.2 The proof-of-concept

The proof-of-concept developed for this thesis can calculate the metrics of the model for all the dependencies of the dependency tree of a given client library. The client and the server libraries have to be available in the *Maven Central Repository*. The main limitation of the PoC is that it does not include testing dependencies and cannot detect shaded dependencies. The first point could be fixed by changing the process to obtain both the client and the server library sources. Obtaining *.jar* files from Maven makes it impossible to include the tests in the analysis and adds issues with shaded dependencies within the *.jar* file of the client. Therefore, an initial step would be to add the possibility of obtaining the client and server libraries from a different repository, including the tests in the *.jar* file. Another option would be to change the type of analysis to source-code analysis, but this would require changing the analysis completely. Moreover, the issues with the parent module's inherited dependencies could be solved by changing the strategy to resolve the dependency tree and include the inherited dependencies in the analysis. Moreover, in the future, it could also be valuable to calculate the metrics by using previously generated call-graphs instead of bytecode analysis. Generating merged call-graphs of different libraries is a work in progress within the FASTEN project. Finally, since the model is meant to be language-agnostic, it would be good to look at other languages apart from Java for the implementation. It would be an interesting exercise to see which characteristics and edge cases other languages have and make both the implementation and the model more complete.

Acknowledgements

Firstly, I want to thank my family and friends for their support during all my studies, including the experience of this master thesis. In particular, to Jakob Löhnertz, for the discussions we have had about the topic, which gave me ideas to continue my work. Also, to Mar Badias, for always being there to listen to me when I needed it, working side-by-side during quarantine weeks really helped me.

I want to thank my supervisor from the University of Amsterdam, Ana Oprescu, and my supervisors within SIG Lodewijk Bergmans and Miroslav Živković. The weekly discussions we have had during the entire process and their feedback were crucial for the success of this thesis.

To the entire research team of SIG, thank you for having me during these last months, for the help offered during the weekly standups, and for the feedback received after my presentation.

Many thanks to all the participants in the interviews about the tool I developed during the thesis. The feedback and discussions during the interviews were truly interesting and provided valuable insights.

Finally, I would like to thank the members of the organization of the Seminar Series on Advanced Techniques & Tools for Software Evolution 2020, who allowed me to participate in the seminar. Being able to participate and organize *SATToSE 2020* was a great experience. Thanks again to Ana Oprescu and Miroslav Živković for co-authoring the paper with me; your contribution was truly valuable.

Bibliography

[1] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks", in *Proceedings of the 14th International Conference on Mining Software Repositories*, IEEE press, 2017, pp. 102–112.

[2] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The maven dependency graph: A temporal graph-based representation of maven central", *IEEE International Working Conference on Mining Software Repositories*, vol. 2019-May, pp. 344–348, 2019, ISSN: 21601860. DOI: 10.1109/MSR.2019.00060. arXiv: arXiv:1901.05392v1.

[3] R. G. Kula, C. De Roover, D. German, T. Ishio, and K. Inoue, "Visualizing the evolution of systems and their library dependencies", in *2014 Second IEEE Working Conference on Software Visualization*, IEEE, 2014, pp. 127–136.

[4] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the Maven repository", *Journal of Systems and Software*, vol. 129, pp. 140–158, Jul. 2017, ISSN: 01641212. DOI: 10.1016/j.jss.2016.04.008.

[5] J. Hejderup, M. Beller, and G. Gousios, *Prazi: From package-based to precise call-based dependency network analyses*, 2018.

[6] R. Wieringa and A. Moralı, "Technical action research as a validation method in information systems design science", in *International Conference on Design Science Research in Information Systems*, Springer, 2012, pp. 220–238.

[7] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the maven ecosystem", *arXiv preprint arXiv:2001.07808*, 2020.

[8] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: Counting those that matter", in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ACM, 2018, p. 42.

[9] C. Gao, L. Bergmans, and X. Schrijen, *A survey of property propagation and aggregation*, FASTEN (825328) - Fine-Grained Analysis of Software Ecosystems as Networks, 2019.

[10] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems", in *2006 22nd IEEE International Conference on Software Maintenance*, IEEE, 2006, pp. 469–478.

[11] L. C. Briand, J. W. Daly, and J. K. Wust, "A unified framework for coupling measurement in object-oriented systems", *IEEE Transactions on software Engineering*, vol. 25, no. 1, pp. 91–121, 1999.

[12] L. Briand, P. Devanbu, and W. Melo, "An investigation into coupling measures for c++", in *Proceedings of the 19th international conference on Software engineering*, 1997, pp. 412–421.

[13] J. Eder, G. Kappel, and M. Schrefl, "Coupling and cohesion in object-oriented systems", Citeseer, Tech. Rep., 1994.

[14] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes", *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[15] E. B. Allen and T. M. Khoshgoftaar, "Measuring coupling and cohesion: An information-theory approach", in *Proceedings Sixth International Software Metrics Symposium (Cat. No. PR00403)*, IEEE, 1999, pp. 119–127.

[16] M. Hitz and B. Montazeri, *Measuring coupling and cohesion in object-oriented systems*. Citeseer, 1995.

[17] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design", *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[18] N. I. Churcher and M. J. Shepperd, "Towards a conceptual framework for object oriented software metrics", *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 2, pp. 69–75, 1995.

[19] W. Li and S. M. Henry, *Object-oriented metrics which predict maintainability*, 1993.

[20] F. B. Abreu, M. Goulão, and R. Esteves, "Toward the design quality evaluation of object-oriented software systems", in *Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA*, 1995, pp. 44–57.

[21] K. Srinivasan and T. Devi, "Software metrics validation methodologies in software engineering", *International Journal of Software Engineering & Applications*, vol. 5, no. 6, p. 87, 2014.

[22] A. Meneely, B. Smith, and L. Williams, "Validating software metrics: A spectrum of philosophies", *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 4, 2012, ISSN: 1049331X. DOI: 10.1145/2377656.2377661.

[23] L. C. Briand, S. Morasca, and V. R. Basili, "Property-based software engineering measurement", *IEEE transactions on software Engineering*, vol. 22, no. 1, pp. 68–86, 1996.

[24] F. G. Wilkie and B. A. Kitchenham, "Coupling measures and change ripples in c++ application software", *Journal of Systems and Software*, vol. 52, no. 2-3, pp. 157–164, 2000.

[25] H. Y. Yang, E. Tempero, and R. Berrigan, "Detecting indirect coupling", in *2005 Australian Software Engineering Conference*, IEEE, 2005, pp. 212–221.

[26] G. Gui and P. D. Scott, "Ranking reusability of software components using coupling metrics", *Journal of Systems and Software*, vol. 80, no. 9, pp. 1450–1459, 2007.

[27] V. Gupta and J. K. Chhabra, "Package coupling measurement in object-oriented software", *Journal of computer science and technology*, vol. 24, no. 2, pp. 273–283, 2009.

[28] R. Harrison, S. Counsell, and R. Nithi, "Coupling metrics for object-oriented design", in *Proceedings Fifth International Software Metrics Symposium. Metrics (Cat. No. 98TB100262)*, IEEE, 1998, pp. 150–157.

[29] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring-improving coupling and cohesion of existing code", in *11th working conference on reverse engineering*, IEEE, 2004, pp. 144–151.

[30] F. Koetter, M. Kochanowski, M. Kintz, B. Kersjes, I. Bogicevic, and S. Wagner, "Assessing software quality of agile student projects by data-mining software repositories", in *Proceedings of the 11th International Conference on Computer Supported Education-Volume 2: CSEDU, INSTICC*, SciTePress, 2019, pp. 244–251.

[31] J. Zhao, "Measuring coupling in aspect-oriented systems", in *10th International Software Metrics Symposium (Metrics 04)*, 2004.

[32] C. Gao, L. Bergmans, and X. Schrijen, *A survey of property propagation and aggregation*, FASTEN (825328) - Fine-Grained Analysis of Software Ecosystems as Networks, 2019.

[33] E. J. Weyuker, "Evaluating software complexity measures", *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1357–1365, 1988.

[34] S. Raemaekers, A. Van Deursen, and J. Visser, "The maven repository dataset of metrics, changes, and dependencies", in *IEEE International Working Conference on Mining Software Repositories*, 2013, pp. 221–224, ISBN: 9781467329361. DOI: 10.1109/MSR.2013.6624031.

[35] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Why and how Java developers break APIs", *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings*, vol. 2018-March, no. Dcc, pp. 255–265, 2018. DOI: 10.1109/SANER.2018.8330214. arXiv: 1801.05198.

[36] N. Harrand, A. Benelallam, C. Soto-Valero, O. Barais, and B. Baudry, "Analyzing 2.3 Million Maven Dependencies to Reveal an Essential Core in APIs", no. August, 2019. arXiv: 1908.09757. [Online]. Available: http://arxiv.org/abs/1908.09757.

[37] B. Iooss and A. Saltelli, "Handbook of Uncertainty Quantification", *Handbook of Uncertainty Quantification*, pp. 1–20, 2016. DOI: 10.1007/978-3-319-11259-6.

[38] B. Everitt and A. Skrondal, *The Cambridge dictionary of statistics*. Cambridge University Press Cambridge, 2002, vol. 106.

[39] T. Munzner, "A nested model for visualization design and validation", *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 921–928, 2009, ISSN: 10772626. DOI: 10.1109/TVCG.2009.111.

[40] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data", in *2010 IEEE International Conference on Software Maintenance*, IEEE, 2010, pp. 1–10.

[41] J. Hejderup, *In dependencies we trust: How vulnerable are dependencies in software modules?*, 2015.

[42] H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries", in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2015, pp. 411–420.

[43] R. G. Kula, C. De Roover, D. M. German, T. Ishio, and K. Inoue, "Modeling Library Dependencies and Updates in Large Software Repository Universes", 2017. arXiv: 1709.04626. [Online]. Available: http://arxiv.org/abs/1709.04626.

[44] J. Hejderup, A. van Deursen, and G. Gousios, "Software ecosystem call graph for dependency management", in *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, IEEE, 2018, pp. 101–104.

[45] T. Vernazza, G. Succi, and G. Granatella, "Defining Metrics for Software Components", *submitted to ECOOP'2000*, vol. XI, no. July 2000, pp. 1–11, 2000.

[46] H. Y. Yang, "Measuring Indirect Coupling", p. 160, 2010. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.153.4609.

[47] S. Allier, S. Vaucher, B. Dufour, and H. Sahraoui, "Deriving coupling metrics from call graphs", *Proceedings - 10th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2010*, pp. 43–52, 2010. DOI: 10.1109/SCAM.2010.25.

[48] S. Almugrin, W. Albattah, and A. Melton, "Using indirect coupling metrics to predict package maintainability and testability", *Journal of Systems and Software*, vol. 121, pp. 298–310, 2016, ISSN: 01641212. DOI: 10.1016/j.jss.2016.02.024. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2016.02.024.

[49] S. Almugrin and A. Melton, "Indirect package coupling based on responsibility in an agile, object-oriented environment", in *2015 Second International Conference on Trustworthy Systems and Their Applications*, 2015, pp. 110–119. DOI: 10.1109/TSA.2015.26.

[50] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

Appendix A

Data set

The data set used for the experiments *Coupling metrics significance* and *Benchmarking* is displayed in Table A.1.

Group Id	Artifact Id	Version
org.apache.flink	flink-core	1.11.2
com.puppycrawl.tools	checkstyle	8.36.2
com.google.auto	auto-common	0.11
edu.stanford.nlp	stanford-corenlp	4.0.0
com.squareup.moshi	moshi-kotlin	1.10.0
org.neo4j	neo4j-collections	4.1.2
org.asynchttpclient	async-http-client	2.12.1
org.alluxio	alluxio-core-transport	2.3.0
com.github.javaparser	javaparser-symbol-solver-logic	3.15.15
io.undertow	undertow-benchmarks	2.2.0.Final
org.teavm	teavm-core	0.6.1
com.github.jknack	handlebars-markdown	4.2.0
ma.glasnost.orika	orika-eclipse-tools	1.5.4
fr.inria.gforge.spoon	spoon-core	8.2.0
org.jacop	jacop	4.7.0
com.google.guava	guava	29.0-jre
com.fasterxml.jackson.core	jackson-databind	2.11.2
org.clojure	clojure	1.10.1
org.apache.logging.log4j	log4j-core	2.13.3
javax.servlet	javax.servlet-api	4.0.1
org.mockito	mockito-core	3.5.11
org.apache.httpcomponents	httpclient	4.5.12
org.slf4j	slf4j-simple	1.7.30
org.junit.jupiter	junit-jupiter-api	5.7.0
org.slf4j	slf4j-log4j12	1.7.30
joda-time	joda-time	2.10.6
com.squareup.okhttp3	okhttp	4.9.0
mysql	mysql-connector-java	8.0.21
org.easymock	easymock	4.2
org.hamcrest	hamcrest-core	2.2
commons-beanutils	commons-beanutils	1.9.4
org.apache.logging.log4j	log4j-slf4j-impl	2.13.3
org.springframework	spring-core	5.2.9.RELEASE

org.slf4j	jul-to-slf4j	2.0.0-alpha1
org.eclipse.jetty	jetty-server	11.0.0.beta1
com.sun.xml.bind	jaxb-impl	3.0.0-M4
org.jetbrains.kotlin	kotlin-stdlib	1.4.10
org.apache.maven	maven-project	3.0-alpha-2
org.apache.maven	maven-artifact	3.6.3
org.apache.camel	camel-core	3.5.0
org.slf4j	jcl-over-slf4j	2.0.0-alpha1
org.reflections	reflections	0.9.12
org.apache.ant	ant	1.10.8
xerces	xercesImpl	2.12.0
org.slf4j	log4j-over-slf4j	2.0.0-alpha1
com.sun.xml.bind	jaxb-core	3.0.0-M4
org.scala-js	scalajs-library_2.12	1.2.0
commons-logging	commons-logging	1.2
commons-fileupload	commons-fileupload	1.4
org.springframework.boot	spring-boot-autoconfigure	2.3.4.RELEASE
org.springframework	spring-jdbc	5.2.9.RELEASE
org.springframework	spring-webmvc	5.2.9.RELEASE
org.springframework	spring-aop	5.2.9.RELEASE
org.springframework	spring-orm	5.2.9.RELEASE
org.springframework	spring-tx	5.2.9.RELEASE
org.springframework	spring-web	5.2.9.RELEASE
org.springframework	spring-beans	5.2.9.RELEASE
org.springframework	spring-context	5.2.9.RELEASE
org.springframework	spring-context-support	5.2.9.RELEASE
com.google.inject	guice	4.2.3
com.google.guava	guava	29.0-jre
com.google.code.findbugs	annotations	3.0.1
ch.qos.logback	logback-core	1.3.0-alpha5
ch.qos.logback	logback-classic	1.3.0-alpha5

Table A.1: List of libraries from Maven Central used in experiments 2 and 5.