CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1. Research scope and motivation

The domain of the proposed thesis is Automated Software Engineering. The thesis will develop methods for the analysis of legacy software systems, focusing on using historical information describing the evolution of the systems extracted from the versioning systems. The methods for analysis will integrate techniques based on computational algorithms as well as data-mining. As proof-of-concept, tool prototypes will implement the proposed methods and validate them by extensive experimentation on several cases of real-life systems.

1.2. Objectives of the thesis

1.3. Structure of the thesis

1.4. Main Contributions

2. STATE OF THE ART IN SOFTWARE DEPENDENCIES

2.1. Structural dependencies

A dependency is created by two elements that are in a relationship and indicates that an element of the relationship, in some manner, depends on the other element of the relationship [1], [2].

Structural dependencies can be found by analyzing the source code [3], [4]. There are several types of relationships between these source code entities and all those create *structural dependencies*:

Data Item Dependencies

Data items can be variables, records or structures. A dependency is created between two data items when the value held in the first data item is used or affects the value from the second.

Data Type Dependencies

Data items are declared to be of a specific data type. Besides the built-in data types that every programming language has, developers can also create new types that they can use. Each time the data type definition is changed it will affect all the data items that are declared to be of that type.

Subprogram Dependencies

A subprogram is a sequence of instructions that performs a certain task. Depending on the programming language a subprogram may also be called a routine, a method, a function or a procedure. When a subprogram is changed, the developer must check the effects of that change in all the places that are calling that subprogram. Subprograms may also have dependencies with the data items that they receive as input or the data items that they are computing.

2.2. Logical dependencies

2.2.1. Software change

Software has distinctive stages during its life: Initial development, Evolution, Servicing, Phase out and Close down [5], [6].

In the *evolution stage* iterative changes are made. By changes, we mean additions (new software features), modifications (changes of requirements or misunderstood requirements) or deletions. There are two main reasons for the change: the learning process of the software team and new requests from the client.

If new changes are no longer easy to be made or are very difficult and time-consuming, the software enters the *servicing stage* also called aging software, decayed software and legacy [5], [7].

The main difference between changes made in the evolution stage and changes made in the servicing stage is the effort of making changes. In the evolution stage, software changes are made easily and do not require much effort while in the servicing stage only a limited number of changes are made and require a lot of effort, so are really time-consuming [8], [9].

The change mini-cycle consists of the following phases [10]:

- Phase 1: The request for change. This usually comes from the users of the software and it can also be a bug report or a request for an additional functionality.
- Phase 2: The planning phase, this includes program comprehension and change impact analysis. Program comprehension is a mandatory prerequisite of the change while change impact analysis indicates how costly the change is going to be. [11]
- Phase 3: The change implementation, restructuring for change and change propagation.
- Phase 4: Verification and validation.
- Phase 5: Re-documentation.

2.2.2. Version control systems

Software evolution implies change which can be triggered either by new feature requests or bug reports [12]. As presented also in section 2.2.1, one phase of the change mini-cycle consists of change implementation and propagation (changing source code files). Usually, developers use version control when it comes to software development. Version control is a system that records changes to a file or set of files over time so that developers can recall specific versions of those files later [13]. Distributed version control systems (such as Git, Mercurial, Bazaar or Darcs) allows many developers to collaboratively develop their projects [14].

2.2.3. Logical dedependencies in software systems

Logical dependencies (a.k.a logical coupling) can be found by software history analysis and can reveal relationships that are not always present in the source code (structural dependencies).

Software engineering practice has shown that sometimes modules which do not present structural dependencies still appear to be related. Co-evolution represents the phenomenon when one component changes in response to a change in another component [15], [16]. Those changes can be found in the software history maintained by the versioning system. Gall [17], [18] identified as logical coupling between two modules the fact that these modules *repeatedly* change together during the historical evolution of the software system [19].

The concepts of logical coupling and logical dependencies were first used in different analysis tasks, all related to changes: for software change impact analysis [20], for identifying the potential ripple effects caused by software changes during software maintenance and evolution [21], [22], [23], [24] or for their link to deffects [25], [26].

The current trend recommends that general dependency management methods and tools should also include logical dependencies besides the structural dependencies [22], [27].

2.2.4. Current status of research

Oliva and Gerosa [22], [21] have found first that the set of co-changed classes was much larger compared to the set of structurally coupled classes. They identified structural and logical dependencies from 150000 revisions from the Apache Software Foundation SVN repository. Also they concluded that in at least 91% of the cases, logical dependencies involve files that are not structurally related. This implies that not all of the change dependencies are related to structural dependencies and there could be other reasons for software artifacts to be change dependent.

Ajienka and Capiluppi also studied the interplay between logical and structural coupling of software classes. In [27] they perform experiments on 79 open source systems: for each system, they determine the sets of structural dependencies, the set of logical dependencies and the intersections of these sets. They quantify the overlapping or intersection of these sets, coming to the conclusion that not all co-changed class pairs (classes with logical dependencies) are also linked by structural dependencies. One other interesting aspect which has not been investigated by the authors in [27] is the total number of logical dependencies, reported to the total number of structural dependencies of a software systems. However, they provide the raw data of their measurements and we calculated the ratio between the number of logical dependencies and the number of structural dependencies for all the projects analyzed by them: the average ratio resulted 12. This means that, using their method of detecting logical dependencies for a system, the number of logical dependencies outnumbers by one order of magnitude the number of structural dependencies. We consider that such a big number of logical dependencies needs additional filtering.

Another kind of non-structural dependencies are the semantic or conceptual dependencies [23], [24]. Semantic coupling is given by the degree to which the identifiers and comments from different classes are similar to each other. Semantic coupling could be an indicator for logical dependencies, as studied by Ajienka et al in [28]. The experiments showed that a large number of co-evolving classes do not present semantic coupling, adding to the earlier research which showed that a large number of co-evolving classes do not present structural coupling. All these experimental findings rise the question whether it is a legitimate approach to accept all co-evolving classes as logical coupling.

Zimmermann et al [26] introduced data mining techniques to obtain association rules from version histories. The mined association rules have a probabilistic interpretation based on the amount of evidence in the transactions they are derived from. This amount of evidence is determined by two measures: support and confidence. They developed a tool to predict future or missing changes.

Different applications based on dependency analysis could be improved if, beyond structural dependencies, they also take into account the hidden non-structural dependencies. For example, works which investigate different methods for architectural reconstruction [29], [30], [31], all of them based on the information provided by structural dependencies, could enrich their dependency models by taking into account also logical dependencies. However, a thorough survey [32] shows that historical information has been rarely used in architectural reconstruction.

Another survey [33] mentions one possible explanation why historical information have been rarely used in architectural reconstruction: the size of the extracted information. One problem is the size of the extraction process, which has to analyze many versions from the historical evolution of the system. Another problem is the big number of pairs of classes which record co-changes and how they relate to the number of pairs of classes with structural dependencies.

The software architecture is important in order to understand and maintain a system. Often code updates are made without checking or updating the architecture. This kind of updates cause the architecture to drift from the reality of the code over time [32]. So reconstructing the architecture and verifying if still matches the reality is important [34].

Surveys also show that architectural reconstruction is mainly made based on structural dependencies [33], [32], the main reason why historical information is rarely used in architectural reconstruction is the size of the extracted information.

Logical dependencies should integrate harmoniously with structural dependencies in an unitary dependency model: valid logical dependencies should not be omitted from the dependency model, but structural dependencies should not be engulfed by questionable logical dependencies generated by casual co-changes. Thus, in order to add logical dependencies besides structural dependencies in dependency models, class co-changes must be filtered until they remain only a reduced but relevant set of valid logical dependencies.

Currently there is no set of rules or best practices that can be applied to the extracted class co-changes and can guarantee their filtering into a set of valid logical dependencies. This is mainly because not all the updates made in the versioning sys-

tem are code related. For example a commit that has as participants a big number of files can indicate that a merge with another branch or a folder renaming has been made. In this case, a series of irrelevant co-changing pairs of entities can be introduced. So, in order to exclude this kind of situations the information extracted from the versioning system has to be filtered first and then used.

Other works have tried to filter co-changes [22], [27]. One of the used co-changes filter is the commit size.The commit size is the number of code files changed in that particular commit. Ajienka and Capiluppi established a threshold of 10 for the maximum accepted size for a commit [27]. This means that all the commits that had more than 10 code files changed where discarded from the research. But setting a harcoded threshold for the commit size is debatable because in order to say that a commit is big or small you have to look first at the size of the system and at the trends from the versioning system. Even thought the best practices encourage small and often commits, the developers culture is the one that influences the most the trending size of commits from one system.

Filtering only after commit size is not enough, this type of filtering can indeed have an impact on the total number of extracted co-changes, but will only shrink the number of co-changes extracted without actually guaranteeing that the remaining ones have more relevancy and are more logical linked.

Although, some unrelated files can be updated by human error in small commits, for example: one file was forgot to be commited in the current commit and will be commited in the next one among some unrelated files. This kind of situation can introduce a set of co-changing pairs that are definitely not logical liked. In order to avoid this kind of situation a filter for the occurrence rate of co-changing pairs must be introduced. Co-changing pairs that occur multiple times are more prone to be logically dependent than the ones that occur only once. Currently there are no concrete examples of how the threshold for this type of filter can be calculated. In order to do that, incrementing the threshold by a certain step will be the start and then studying the impact on the remaining co-changing pairs for different systems.

Taking into account also structural dependencies from all the revisions of the system was not made in previous works, this step is important in order to filter out the old, out-of-date logical dependencies. Some logical dependencies may have been also structural in previous revisions of the system but not in the current one. If we take into consideration also structural dependencies from previous revisions then the overlapping rate between logical and structural dependencies could probably increase. Another way to investigate this problem could be to study the trend of concurrences of co-changes: if co-changes between a pair of classes used to happen more often in the remote past than in the more recent past, it may be a sign that the problem causing the logical coupling has been removed in the mean time.

2.3. Applications of software dependencies

2.3.1. Reverse engineering

The term reverse engineering was first defined by Chikofsky and Cross [35] as the *"process of analyzing a system to (i) identify the system's components and their inter-relationships and (ii) create representations of the system in another form or at a higher level of abstraction."*

Reverse engineering is viewed as a two step process: information extraction and abstraction. [36] The firs step, information extraction, is made by source code analysis which generates dependencies between software artifacts. So, reverse engineering uses dependencies in order to create new representations of the system or provide a higher level of abstraction [**?**], [37].

2.3.2. Architecture reconstruction

Currently, the software systems contain tens of thousands of lines of code and are updated multiple times a day by multiple developers. The software architecture is important in order to understand and maintain a system. Often code updates are made without checking or updating the architecture. This kind of updates cause the architecture to drift from the reality of the code over time. So reconstructing the architecture and verifying if still matches the reality is important. [32],[31], [38] ,[39], [6].

2.3.3. Identifying clones

Research suggests that a considerable part (around 5-10%) of the source code of large-scale software is duplicate code ("clones"). Source code is often duplicated for a variety of reasons, programmers may simply be reusing a piece of code by copy and paste or they may be "reinventing the wheel" [40], [41]. Detection and removal of clones can decrease software maintenance costs [42], [43].

2.3.4. Code smells

Code smells have been defined by Fowler [44] and describe patterns that are generally associated with bad design and bad programming practices. Originally, code smells are used to find the places in software may need refactoring [45]. Studies have found that smells may affect comprehension and possibly increase change and fault proneness [46], [47], [48]. Examples of code smells:

- Large Class: one class with many fields.
- Feature Envy: methods that access more methods and fields of another class than of its own class.
- Data Class: classes that only fields and do not contain functionality.
- Refused Bequest: classes that leave many of the fields and methods they inherit

unused
- Parallel Inheritance: every time you make a subclass of one class you also have to make a subclass of the other.
- Shotgun Surgery: one method is changing together with other methods contained other classes.

Previous studies already explored the idea of using history information in order to detect code smells [49].

2.3.5. Comprehension

Software comprehension is the process of gaining knowledge about a software system. An increased knowledge of the software system help activities such as bug correction, enhancement, reuse and documentation [50], [51], [52]. Previous studies show that the proportion of resources and time allocated to maintenance may vary from 50% to 75% [53]. Regarding maintenance, the greatest part of the software maintenance process is the activity of understanding the system. Thanking into consideration the previous statements we can say that if we want to improve software maintenance we have to improve software comprehension [54].

2.3.6. Fault location

Debugging software is an expensive and mostly manual process. Of all debugging activities, fault localization, is the most expensive [55].

Software developers locate faults in their programs using a manual process. This process begins when the developers observe failures in the program. The developers choose a set of data to inject in the system(a set of data that most likely replicate previous failures or may generate new ones) and place breakpoints using a debugger. Then they observe the system's state until an failed state is reached, and then backtrack until the fault is found.

As we said, this process has high costs so because of this high cost, any improvement in the process can decrease the cost of debugging.[56] [57]

2.3.7. Error proneness

Research has shown that based on the software error history and simple metrics related to the amount of data and the structural complexity of software, modules that are most likely to contain errors can be identified [58], [59].

2.3.8. Empirical software engineering research

Empirical research tries to explore, describe, predict, and explain natural or social phenomena by using evidence based on observation or experience. It involves obtaining and interpreting evidence by experimentation, systematic observation, interviews, surveys, or by the careful examination of documents and artifacts. [60]

3. EXTRACTING SOFTWARE DEPENDENCIES

3.1. Extracting structural dependencies

A dependency is created between two elements that are in a relationship and indicates that an element of the relationship, in some manner, depends on the other element of the relationship [1], [2].

Structural dependencies can be found by analyzing the source code [3], [4], [61]. A structural dependency between two classes A and B is given by the fact that A statically depends on B, meaning that A cannot be compiled without knowing about B. In object oriented systems, this dependency can be given by many types of relationships between the two classes: A extends B, A implements B, A has attributes of type B, A has methods which have type B in their signature, A uses local variables of type B, A calls methods of B.

We use an external tool called srcML [62] to convert all source code files from the current release into XML files. All the information about classes, methods, calls to other classes are extracted by parsing the XML files and building a dependency data structure [52], [63]. We choose the srcML format because it has the same markup for different programming languages and can ease the parsing of source code written in various programming languages such as Java, C++, and C#.

3.2. Extracting co-changing pairs

Logical dependencies (a.k.a logical coupling) can be found by software history analysis and can reveal relationships that are not always present in the source code (structural dependencies).

The concepts of logical coupling and logical dependencies were first used in different analysis tasks, all related to changes: for software change impact analysis [20], for identifying the potential ripple effects caused by software changes during software maintenance and evolution [21], [22], [23], [24] or for their link to deffects [25], [26].

Software engineering practice has shown that sometimes modules which do not present structural dependencies still can be related [12]. Co-evolution represents the phenomenon when one component changes in response to a change in another component [15], [16]. Those changes can be found in the software change history from the versioning system. Gall [17], [18] identified as logical coupling between two modules the fact that these modules *repeatedly* change together during the historical evolution of the software system [19].

The versioning system contains the long-term change history of every file. Each project change made by an individual at a certain point of time is contained into a commit [14]. All the commits are stored in the versioning system chronologically and each commit has a parent. The parent commit is the baseline from which development began, the only exception to this rule is the first commit which has no parent [13].

Currently there is no set of rules or best practices that can be applied to the extracted class co-changes and can guarantee their filtering into a set of logical dependencies. This is mainly because not all the updates made in the versioning system are code related. For example, a commit that has as participants a big number of files can indicate that a merge with another branch or a folder renaming has been made. In this case, a series of irrelevant co-changing pairs of entities can be introduced. So, in order to exclude this kind of situations the information extracted from the versioning system has to be filtered first and then used. Surveys also show that historical information is rarely used due to the size of the extracted information [33], [32].

Other works have tried to filter co-changes [22], [27], [21]. One of the used co-changes filter is the commit size.The commit size is the number of code files changed in that particular commit. Ajienka and Capiluppi established a threshold of 10 for the maximum accepted size for a commit [27]. This means that all the commits that had more than 10 code files changed where discarded from the research. But setting a harcoded threshold for the commit size is debatable because in order to say that a commit is big or small you have to look first at the size of the system and at the trends from the versioning system. Even thought the best practices encourage small and often commits, the developers culture is the one that influences the most the trending size of commits from one system.

Filtering only after commit size is not enough, this type of filtering can indeed have an impact on the total number of extracted co-changes, but will only shrink the number of co-changes extracted without actually guaranteeing that the remaining ones have more relevancy and are more linked.

Although, some unrelated files can be updated by human error in small commits, for example: one file was forgot to be commited in the current commit and will be commited in the next one among some unrelated files. This kind of situation can introduce a set of co-changing pairs that are definitely not logical liked. In order to avoid this kind of situation a filter for the occurrence rate of co-changing pairs can be introduced. Co-changing pairs that occur multiple times are more prone to be logically dependent than the ones that occur only once. Currently there are no concrete examples of how the threshold for this type of filter can be calculated. In order to do that, incrementing the threshold by a certain step will be the start and then studying the impact on the remaining co-changing pairs for different systems.

Nevertheless, logical dependencies should integrate harmoniously with structural dependencies in an unitary dependency model: valid logical dependencies should not be omitted from the dependency model, but structural dependencies should not be engulfed by questionable logical dependencies generated by casual co-changes. Thus, in order to add logical dependencies besides structural dependencies in dependency models, class co-changes must be filtered until they remain only a reduced but relevant set of valid logical dependencies.

3.3. Tool for measuring software dependencies

To establish structural and logical dependencies, we developed a tool that takes as input the source code repository URL of a given system and extracts from it the software dependencies [64]. From a workflow point of view, we can identify 3 major types of activities that the tool does: downloads the required data from the git repository, extracts from the source code the structural dependencies and, extracts and filters the co-changing pairs from the repository's commit history. Figure 3.1 represents the activities mentioned above. Each block represents a different activity.

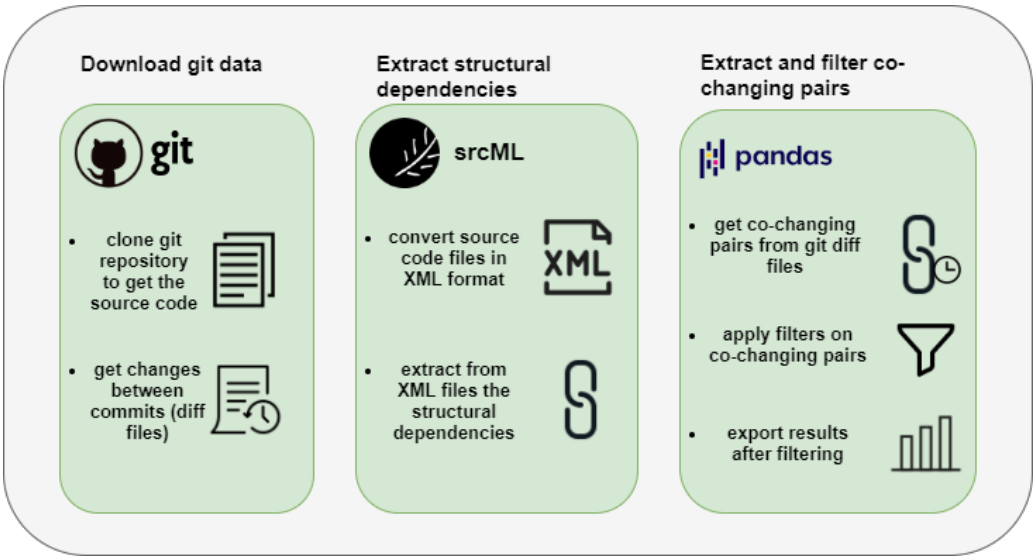

Figure 3.1: Tool workflow and major activities.

Download git data.

The source code repository provides us all the needed information to extract both types of dependencies. It holds the code of the system but also the change history of the system. We use the source code for structural dependencies extraction and the change history for co-changing pairs extraction. To get the source code files and the change history, we first need to know the repository URL from GitHub (GitHub is a Git repository cloud-based hosting service). With the GitHub URL and a series of Git commands, the tool can download all the necessary data for dependencies extraction.

As we can see in figure 3.2, the *"clone"* command will download a Git repository to your local computer, including the source code files. The *"diff"* command will get the differences between two existing commits in the Git repository. The tool gets the Git repository and the source code files by executing the "clone" command. Afterward, it gets all the existing commits within the Git repository. The commits are ordered by date, beginning with the oldest one and ending with the most recent one. The tool executes the "diff" command between each commit and its parent (the previous commit). The "diff" command generates a text file that contains the differences

between the two commits: code differences, the number of files changed and changed file names.

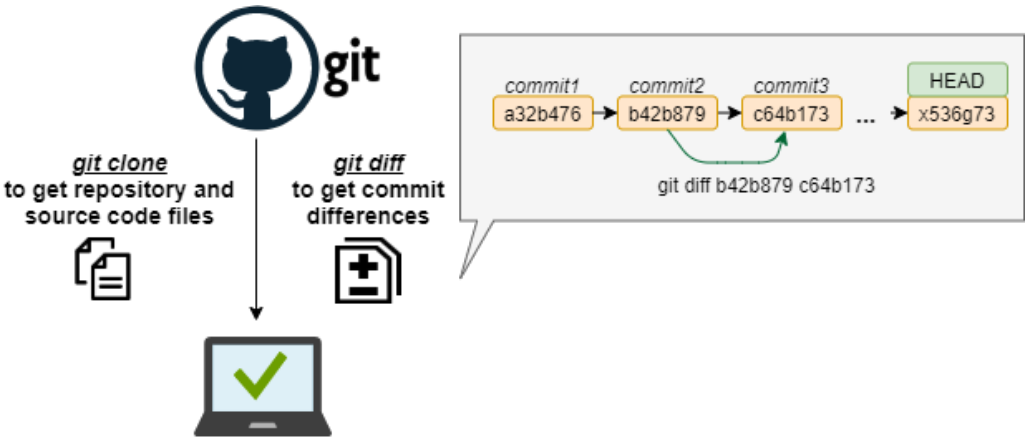

Figure 3.2: Commands used to download the required data from GitHub.

Extract structural dependencies.

To extract the structural dependencies from the source code files the tool converts each source code file into srcML format using an open-source tool called srcML. The srcML format is an XML representation for source code. Each markup tag identifies elements of the abstract syntax for the language [62]. After conversion, the tool parses each file and identifies all the defined entities (class, interface, enum, struct) within the file. It also identifies all the entities that are used by the entities defined. The connection between both types of entities mentioned above constitutes a structural dependency.

Extract and filter co-changing pairs.

The process of extracting and filtering the co-changing pairs is represented in figure 3.3. For co-changing pairs extraction, the tool parses each generated diff file. For each file, the tool gets the number of changed files and the name of the files. After structural dependencies extraction, the tool knows all the software entities contained in a file. Two entities from two changed files form a co-changing pair. After all the co-changing pairs of one diff file are extracted, the tool moves to the next diff file and extracts the set of co-changing pairs.

As will be presented in more details in sections 4.2, 4.3, and 4.4, not every co-changing pair extracted is a logical dependency. For a co-changing pair to be labeled as a logical dependency, it has to meet some criteria. Each criterion constitutes a filter that a co-changing pair has to pass in order to be called logical dependency. The filters are implemented in the tool and can be combined. The input for each filter is the set of co-changing pairs extracted, and the output is the remaining co-changing pairs that respect the filter criterion.

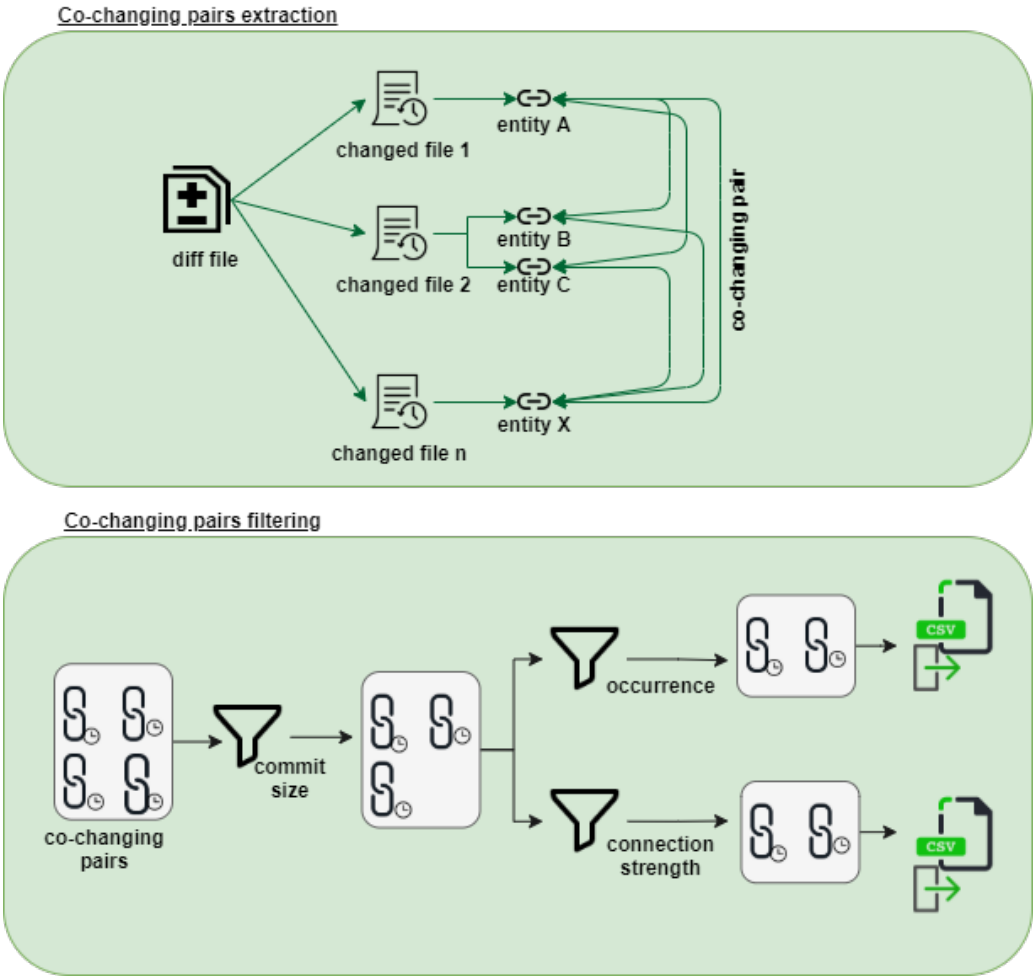

Figure 3.3: Co-changing pairs extraction and filtering.

4. FILTERING EXTRACTED CO-CHANGING PAIRS IN ORDER TO OBTAIN LOGICAL DEPENDENCIES

4.1. Data set used

We have analyzed a set of open-source projects found on GitHub[1] [34] in order to extract the structural and logical dependencies between classes. Table 4.1 enumerates all the systems studied. The 1st column assigns the projects IDs; 2nd column shows the project name; 3rd column shows the number of entities(classes and interfaces) extracted; 4th column shows the number of most recent commits analyzed from the active branch of each project and the 5th shows the language in which the project was developed.

Table 4.1: Summary of open source projects studied.

ID	Project	Nr. of entites	Nr. of commits	Type
1	bluecove	2685	894	java
2	aima-java	5232	1006	java
3	powermock	2801	949	java
4	restfb	3350	1391	java
5	rxjava	21097	4398	java
6	metro-jax-ws	6482	2927	java
7	mockito	5189	3330	java
8	grizzly	10687	3113	java
9	shipkit	639	1563	java
10	OpenClinica	9655	3276	java
11	robolectric	8922	5912	java
12	aeron	4159	5977	java
13	antlr4	4747	4431	java
14	mcidasv	3272	4136	java
15	ShareX	4289	5485	csharp
16	aspnetboilerplate	9712	4323	csharp
17	orleans	16963	3995	csharp
18	cli	2063	4488	csharp
19	cake	12260	2518	csharp
20	Avalonia	16732	5264	csharp
21	EntityFrameworkCore	50179	5210	csharp
22	jellyfin	8764	5433	csharp
23	PowerShell	2405	3250	csharp
24	WeiXinMPSDK	7075	5729	csharp
25	ArchiSteamFarm	702	2497	csharp
26	VisualStudio	4869	5039	csharp
27	CppSharp	17060	4522	csharp

[1] http://github.com/

4.2. Filtering based on the size of commit transactions

As presented in section 3.2, according to surveys, co-changing pairs are not used because of their size. One system can have millions of co-changing pairs. With this filtering type, we not only want to decrease the total size of the extracted co-changing pairs. But also to be one step closer to the identification of the logical dependencies among the co-changing pairs. In this step, we want to filter the co-changing pairs extracted after commit size (cs). This means that the co-changing pairs are extracted only from commits that involve fewer files than an established threshold number.

Different works have chosen fixed threshold values for the maximum number of files accepted in a commit. Cappiluppi and Ajienka, in their works [27], [28] only take into consideration commits with less then 10 source code files changed in building the logical dependencies.

The research of Beck et al [65] only takes in consideration transactions with up to 25 files. The research [22] provided also a quantitative analysis of the number of files per revision; Based on the analysis of 40,518 revisions, the mean value obtained for the number of files in a revision is 6 files. However, standard deviation value shows that the dispersion is high.

We analyzed the overall transaction size trend for 27 open-source csharp and java systems with a total of 74 332 commits. The results are presented in Figure 4.1 and in table 4.2, based on them we can say that 90% of the total commit transactions made are with less than 10 source code files changed. This percent allows us to say that setting a threshold of 10 files for the maximum size of the commit transactions will not affect so much the total number of commit transactions from the systems since it will still remain 90% of the commit transactions from where we can extract co-changing pairs [64].

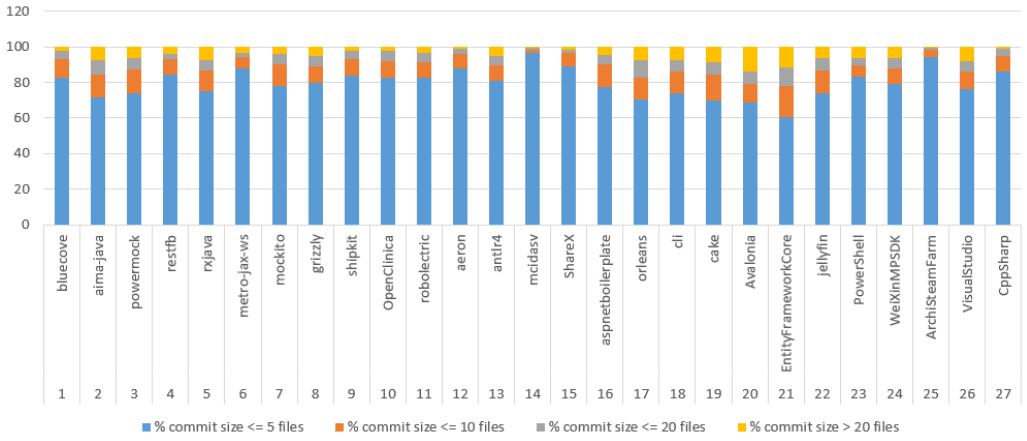

Figure 4.1: Commit transaction size(cs) trend in percentages.

As we can see in Figure 4.2 even though only 5% of the commit transactions have more than 20 files changed ($20 < cs < inf$) they generate in average 80% of

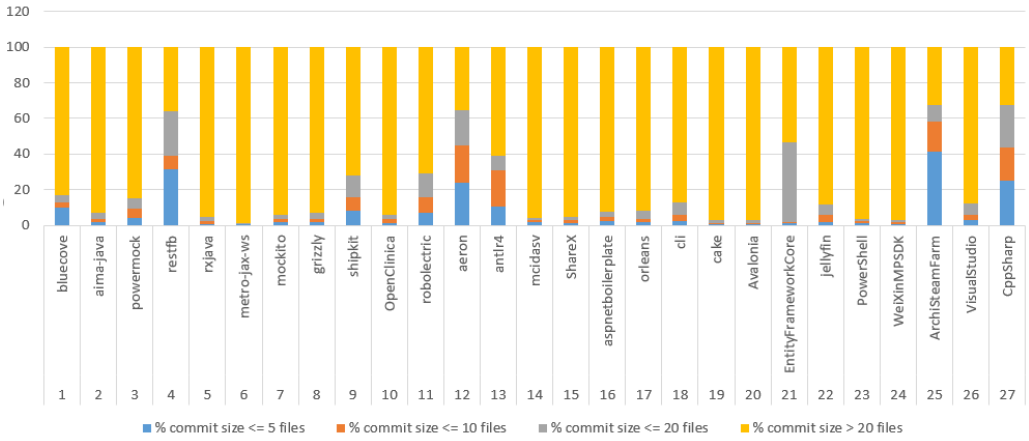

Figure 4.2: Percentages of LD extracted from each commit transaction size(cs) group.

the total amount of co-changing pairs extracted from the systems. The high number of co-changing pairs extracted from such a small number of commit transactions is caused by the number of files involved in those commit transactions.

One single commit transaction can lead to a large amount of co-changing pairs. For example in RxJava we have commit transactions with 1030 source code files, this means that those commits can generate ${}^{n}C_{k} = \frac{n!}{k!(n-k)!} = \frac{1030!}{2!(1028)!} = 529935$ logical dependencies. By setting a threshold on the commit transaction size we can avoid the introduction of those co-changing pairs into the system.

So filtering 10% of the total amount of commit transactions can lead to a significant decrease of the amount of co-changing pairs and that is why we choose the value of 10 files as our fixed threshold for the maximum size of a commit transaction [64].

4.3. Filtering based on number of occurrences

In the previous section, we filtered the co-changing pairs based on the commit size. Even though the number of extracted co-changing pairs was reduced, this type of filtering will not guarantee that the remaining co-changing pairs can pass as logical dependencies. One occurrence of a co-change pair can be a valid logical dependency, but can also be a coincidence.

Taking into consideration only co-changing pairs with multiple occurrences as valid dependencies can lead to more accurate results. But, if the project studied has a relatively small amount of commits, the probability to find multiple updates of the same classes at the same time is less likely to happen, so filtering after the number of occurrences can lead to filtering all the co-changes extracted.

We have performed a series of analyses on the test systems, incrementing the

Table 4.2: Commit transaction size(cs) trend and average per system.

Nr.	Project	$cs \leq 5$	$cs \leq 10$	$cs \leq 20$	$cs < \infty$	Avg
1	bluecove	738	97	37	22	4.9
2	aima-java	733	134	74	65	7.24
3	powermock	685	128	66	70	9.61
4	restfb	1160	127	44	60	9.9
5	rxjava	3395	447	253	303	8.46
6	metro-jax-ws	2583	198	78	68	4.33
7	mockito	2522	433	222	153	6.33
8	grizzly	2487	302	180	144	5.28
9	shipkit	1311	151	64	37	4.26
10	OpenClinica	2837	250	119	70	3.31
11	robolectric	4827	503	264	318	7.43
12	aeron	4844	684	300	149	4.6
13	antlr4	3426	437	304	264	8.5
14	mcidasv	3996	81	35	24	2.47
15	ShareX	4731	529	145	80	4.69
16	aspnetboilerplate	3208	569	321	225	6.61
17	orleans	2780	518	369	328	8.95
18	cli	3377	551	308	252	6.43
19	cake	1785	359	174	200	9.89
20	Avalonia	3806	641	371	446	8.43
21	EntityFrameworkCore	2866	878	644	822	15.38
22	jellyfin	4007	662	419	345	6.25
23	PowerShell	2702	224	133	191	7.33
24	WeiXinMPSDK	4604	526	296	303	9.01
25	ArchiSteamFarm	2357	92	28	20	2.24
26	VisualStudio	3902	521	295	321	6.71
27	CppSharp	3870	390	203	59	3.28

threshold value occurrence (occ) from 1 to 4. The co-changing pairs are extracted only for commits with the commit transaction size less or equal to 10. For each threshold mentioned above, the extracted co-changing pairs are filtered again by the occurrence threshold established. All the co-changing pairs that do not exceed the minimum number of occurrences are discarded.

The results of the analysis are presented in Table 4.3 as percentages of co-changing pairs that are also structural dependencies and Table 4.4 as ratio of the number of co-changing pairs to the number of structural dependencies (SD).

Based on Table 4.3 we can say that only a small percentage of the extracted co-changing pairs are also structural dependencies. This is consistent with the findings of related works [27], [28]. The percentage of co-changing pairs that are also structural dependencies increases with the minimum number of occurrences because the number of co-changing pairs from the systems decreases with the minimum number of occurrences. We calculate the overlapping between co-changing pairs and structural dependencies not only because we want to get an idea of how many structural dependencies are reflected in the versioning system through co-changing pairs, but also because we want to eliminate co-changing pairs that are structural dependencies since they don't bring any new information about the system.

We stopped the minimum occurrences threshold to 4 because we observed that for systems with ID 2, 6, 10, and 16 from Table 4.4 the ratio number is lower

Table 4.3: Percentage of co-changing pairs that are also structural dependencies.

ID	$occ \geq 1$	$occ \geq 2$	$occ \geq 3$	$occ \geq 4$
1	7,13	7,77	7,99	19,71
2	19,54	25,76	29,55	32,16
3	6,66	8,58	11,82	14,87
4	1,16	1,17	0,91	0,80
5	3,99	3,96	7,75	7,49
6	13,92	20,16	22,91	22,77
7	8,38	9,28	14,93	14,58
8	6,70	9,73	14,20	15,60
9	16,98	23,34	29,22	32,89
10	8,94	9,15	11,05	10,59
11	4,99	6,92	8,88	11,08
12	13,19	17,15	18,60	19,57
13	2,43	5,59	8,33	8,21
14	13,27	18,88	19,02	19,28
15	12,90	21,95	25,51	27,01
16	13,33	17,34	18,53	16,24
17	6,09	6,18	6,41	6,44
18	9,73	10,60	14,27	18,80
19	10,26	13,54	13,64	12,60
20	12,83	18,36	21,00	25,72
21	2,86	4,65	5,70	4,98
22	5,20	6,56	8,18	8,90
23	8,23	13,64	17,04	17,65
24	6,77	10,89	14,47	16,05
25	9,85	10,15	11,65	11,33
26	8,65	10,79	12,78	14,34
27	7,04	8,78	9,87	10,08
Avg	8,93	11,88	14,23	15,55

than 1, which means that the number of structural dependencies is higher than the number of co-changing pairs. On the other hand, for systems with ID 4, 11, 25, 27, the threshold of 4 for a minimum number of occurrences does not change the discrepancy between the number of co-changing pairs and structural dependencies.

If we try to go higher with the occurrences threshold, we will risk filtering all the existing co-changing pairs for some systems. So, filtering with a threshold of 4 for the minimum number of occurrences will indeed filter the logical dependencies, but for some of the systems, the remaining number of co-changing pairs will still be significantly higher compared to the number of structural dependencies.

4.4. Filtering based on connection strength

In section 4.2 we filtered the co-changing pairs extracted from the versioning system history based on the commit size. Based on the results obtained, we decided to filter out all co-changing pairs extracted from commits with more than 10 files changed.

In section 4.3, we added a new filtering rule based on the occurrence of a co-changing pair. The new filter is applied to the co-changing pairs resulted after

Table 4.4: Ratio of number of co-changing pairs to number of structural dependencies.

ID	$occ \geq 1$	$occ \geq 2$	$occ \geq 3$	$occ \geq 4$
1	4,13	1,94	1,23	0,26
2	0,81	0,33	0,16	0,10
3	5,12	1,93	0,78	0,38
4	53,36	42,00	38,31	36,30
5	4,27	2,90	0,88	0,72
6	1,07	0,46	0,30	0,23
7	4,09	2,38	0,99	0,73
8	4,06	1,57	0,76	0,49
9	3,64	2,03	1,14	0,77
10	1,41	1,01	0,47	0,34
11	7,91	4,47	2,93	2,03
12	3,92	2,15	1,47	1,07
13	10,15	3,18	1,22	1,03
14	3,07	1,53	1,16	0,97
15	2,34	0,84	0,48	0,33
16	1,21	0,47	0,26	0,19
17	2,99	1,83	1,11	0,84
18	2,26	1,37	0,67	0,40
19	2,32	1,38	0,76	0,67
20	1,24	0,58	0,35	0,18
21	5,33	2,12	1,27	1,05
22	3,38	1,88	0,99	0,74
23	3,62	1,22	0,76	0,37
24	2,57	1,22	0,67	0,46
25	7,47	5,36	4,16	3,73
26	4,03	2,16	1,50	1,15
27	7,46	4,26	2,99	2,43
Avg	5,67	3,43	2,51	2,15

commit size filtering. In this case, the filtering method proved insufficient due to the size diversity of the systems. One important conclusion drawn from the occurrence number filtering is that setting a hard threshold for a filter is not always a good idea. One threshold value can be too much for a small-sized system and too little for a medium-sized system.

To avoid the above problem, we decided to introduce another filter complementary to the commit size filter described in section 4.2. This filter focuses on the connection strength of a co-changing pair. In this section, we will filter out all the co-changing pairs that are not strongly connected.

To determine the connection strength of a pair, we first need to calculate the connection factors for both entities that form a co-changing pair. Assuming that we have a co-changing pair formed by entities A and B, the connection factor of entity A with entity B is the percentage from the total commits involving A that contains entity B. The connection factor of entity B with entity A is the percentage from the total commits involving B that contain also entity A.

$$connection\ factor\ for\ A = \frac{100 * commits\ involving\ A\ and\ B}{total\ nr\ of\ commits\ involving\ A} \tag{4.1}$$

$$connection\ factor\ for\ B = \frac{100 * commits\ involving\ A\ and\ B}{total\ nr\ of\ commits\ involving\ B} \tag{4.2}$$

As a practical example, if the pair formed by A and B update together 7 times and the total number of commits involving A is 20 and involving B is 7. The factor for A is 35 and for B is 100. The factor of 100 is the maximum factor that you can have and means that in all the commits involving B, also A is present.

Due to the fact that the factors obtained can vary from 0 to 100, for this filter, we begin with a threshold value of 10 and increment it by 10 until we reach 100.

The co-changing pairs are filtered out based on two scenarios:

- factor A and factor B $\geq threshold$%
- factor A or factor B $\geq threshold$%

Table 4.5: Ratio of number of filtered co-changing pairs to number of SD, when factor A and factor B $\geq threshold$%

Project	≥ 10%	≥ 20%	≥ 30%	≥ 40%	≥ 50%	≥ 60%	≥ 70%	≥ 80%	≥ 90%	≥ 100%
bluecove	1.326	0.658	0.433	0.401	0.244	0.199	0.195	0.022	0.011	0.011
aima-java	0.266	0.137	0.070	0.044	0.036	0.019	0.005	0.004	0.003	0.003
powermock	0.505	0.243	0.147	0.086	0.061	0.031	0.031	0.031	0.031	0.031
restfb	0.822	0.163	0.045	0.017	0.011	0.002	0.001	0.001	0.001	0.001
rxjava	0.234	0.119	0.054	0.037	0.034	0.018	0.013	0.011	0.007	0.007
metro-jax-ws	0.227	0.155	0.101	0.077	0.070	0.036	0.018	0.017	0.016	0.016
mockito	1.590	0.804	0.357	0.288	0.215	0.088	0.052	0.036	0.032	0.032
grizzly	2.073	0.293	0.170	0.111	0.093	0.050	0.039	0.034	0.021	0.007
shipkit	1.495	0.479	0.271	0.142	0.108	0.059	0.047	0.011	0.008	0.008
OpenClinica	0.253	0.135	0.093	0.078	0.062	0.042	0.024	0.019	0.019	0.017
robolectric	0.114	0.086	0.064	0.037	0.027	0.025	0.001	0.000	0.000	0.000
aeron	0.277	0.136	0.085	0.069	0.053	0.045	0.039	0.015	0.007	0.004
antlr4	11.363	0.721	0.031	0.010	0.007	0.004	0.000	0.000	0.000	0.000
mcidasv	3.225	0.805	0.660	0.533	0.493	0.454	0.386	0.356	0.005	0.005
ShareX	6.097	0.725	0.663	0.564	0.500	0.242	0.176	0.170	0.001	0.001
aspnetboilerplate	1.302	0.333	0.219	0.146	0.094	0.045	0.014	0.008	0.007	0.007
orleans	0.816	0.640	0.551	0.503	0.496	0.196	0.159	0.152	0.142	0.142
cli	1.676	0.233	0.159	0.118	0.102	0.062	0.058	0.029	0.026	0.026
cake	2.335	0.753	0.614	0.337	0.075	0.021	0.007	0.004	0.004	0.004
Avalonia	0.846	0.117	0.098	0.018	0.013	0.002	0.001	0.001	0.001	0.001
EntityFrameworkCore	3.377	1.691	1.608	1.584	1.576	1.310	0.001	0.001	0.001	0.001
jellyfin	0.132	0.006	0.003	0.002	0.002	0.000	0.000	0.000	0.000	0.000
PowerShell	1.732	1.299	0.158	0.053	0.007	0.001	0.000	0.000	0.000	0.000
WeiXinMPSDK	3.295	0.334	0.188	0.061	0.017	0.006	0.003	0.001	0.000	0.000
ArchiSteamFarm	0.897	0.479	0.429	0.423	0.412	0.403	0.339	0.009	0.001	0.000
VisualStudio	1.281	0.090	0.053	0.028	0.020	0.013	0.006	0.001	0.001	0.001
CppSharp	99.528	1.020	0.992	0.980	0.972	0.927	0.078	0.075	0.073	0.072

In table 4.5 we have on the columns the ratio between the number of structural dependencies and the number of co-changing pairs that resulted after filtering out pairs that have at least one factor below the specified threshold in the column header. In table 4.6 we have on the columns the ratio between the number of structural dependencies and the number of co-changing pairs that resulted after filtering out pairs that have both factors below the specified threshold in the column header.

We calculate the ratio number between the co-changing pairs and the structural dependencies because we want to evaluate the size of the extracted co-changing

Table 4.6: Ratio of number of filtered co-changing pairs to number of SD, when factor A or factor B $\geq threshold$%

Project	≥ 10%	≥ 20%	≥ 30%	≥ 40%	≥ 50%	≥ 60%	≥ 70%	≥ 80%	≥ 90%	≥ 100%
bluecove	1.312	1.181	0.700	0.599	0.419	0.235	0.219	0.046	0.045	0.045
aima-java	0.430	0.280	0.176	0.118	0.103	0.056	0.022	0.020	0.020	0.020
powermock	0.508	0.328	0.234	0.179	0.150	0.092	0.091	0.091	0.091	0.091
restfb	0.662	0.336	0.122	0.067	0.059	0.016	0.015	0.015	0.015	0.015
rxjava	0.279	0.206	0.145	0.100	0.099	0.047	0.044	0.039	0.034	0.034
metro-jax-ws	0.271	0.261	0.204	0.172	0.160	0.106	0.082	0.081	0.080	0.080
mockito	2.481	1.521	0.904	0.623	0.411	0.199	0.128	0.107	0.101	0.101
grizzly	1.332	0.838	0.515	0.320	0.288	0.142	0.117	0.106	0.090	0.076
shipkit	1.376	1.083	0.725	0.515	0.424	0.191	0.149	0.105	0.094	0.094
OpenClinica	0.830	0.434	0.314	0.256	0.217	0.130	0.093	0.082	0.080	0.072
robolectric	0.366	0.122	0.088	0.046	0.031	0.027	0.003	0.002	0.002	0.002
aeron	0.781	0.449	0.265	0.190	0.160	0.096	0.062	0.031	0.021	0.018
antlr4	11.363	0.798	0.055	0.022	0.011	0.007	0.002	0.002	0.002	0.002
mcidasv	1.932	1.203	0.858	0.682	0.579	0.473	0.396	0.365	0.013	0.013
ShareX	2.681	1.292	0.916	0.730	0.593	0.287	0.210	0.201	0.017	0.017
aspnetboilerplate	1.055	0.759	0.493	0.364	0.273	0.130	0.067	0.050	0.046	0.046
orleans	1.120	0.962	0.849	0.750	0.744	0.559	0.482	0.476	0.466	0.466
cli	1.676	0.762	0.560	0.434	0.375	0.269	0.237	0.149	0.142	0.142
cake	1.883	1.197	1.001	0.541	0.185	0.103	0.019	0.013	0.013	0.013
Avalonia	0.510	0.224	0.138	0.037	0.028	0.011	0.006	0.003	0.003	0.003
EntityFrameworkCore	2.636	1.888	1.695	1.623	1.608	1.317	0.006	0.006	0.006	0.006
jellyfin	0.132	0.030	0.016	0.011	0.008	0.003	0.002	0.002	0.002	0.002
PowerShell	3.454	1.648	0.232	0.081	0.021	0.004	0.003	0.003	0.003	0.003
WeiXinMPSDK	1.342	0.603	0.327	0.144	0.080	0.047	0.015	0.008	0.007	0.007
ArchiSteamFarm	5.472	1.416	0.830	0.677	0.575	0.450	0.353	0.023	0.016	0.014
VisualStudio	1.281	0.236	0.142	0.092	0.060	0.040	0.031	0.020	0.019	0.019
CppSharp	55.038	1.343	1.106	1.044	1.030	0.983	0.449	0.443	0.441	0.439

pairs compared to the size of the structural dependencies from the system. According to surveys [33], [32], the main reason why logical dependencies (a.k.a filtered co-changes) are not used together with structural dependencies is because of their size. So, it is important to us to get at each filtering step an overview regarding the ratio between co-changes size and structural dependencies size.

From the results presented in tables 4.5 and 4.6 we conclude that the number of co-changing pairs is drastically reduced. In most cases, the number of structural dependencies surpasses the number of co-changing pairs that remain after filtering. But, we do the filtering not only to reduce the size of the co-changing pairs extracted. We do the filtering of co-changing pairs extracted to make sure that the remaining co-changing pairs are indeed logically dependent.

If we filter out all the co-changing pairs that do not update at least half of the time together (factor A and factor B ≥ 50%) we remain with a decent quantity of co-changing pairs. Given the size of the output and the connection strength of the co-changing pairs, the remaining co-changing pairs can be considered, at this point, to be logically dependent.

4.5. Overlaps between structural and co-changing pairs

A logical dependency can be also a structural dependency and vice-versa, so studying the overlapping between logical and structural dependencies while filtering is important since the intention is to introduce those logical dependencies among with structural dependencies in architectural reconstruction systems. Current studies have shown a relatively small percentage of overlapping between them with and without any kind of filtering [27]. This means that a lot of non related entities update together in the versioning system, the goal here is to establish the factors that determine such a small percentage of overlapping [66].

Since we are first extracting co-changing pairs and only after various filters we call the remaining co-changing pairs logically dependent, we will be studying the overlapping between the remaining co-changing pairs after each filtering stage and the structural dependencies. For each system, we extracted the structural dependencies and the co-changing pairs and determined the overlap between the two dependencies sets, in various experimental conditions.

One variable experimental condition is whether changes located in comments contribute towards logical dependencies. This condition distinguishes between two different cases:

- with comments: a change in source code files is counted as a co-changing pair, even if the change is inside comments in all files
- without comments: commits that changed source code files only by editing comments are ignored

In all cases, we varied the following threshold values:

- commit size (cs): the maximum size of commit transactions which are accepted to generate co-changes. The values for this threshold were 5, 10, 20 and no threshold (infinity).
- number of occurrences (occ): the minimum number of repeated occurrences for a co-change to be counted as logical dependency. The values for this threshold were 1, 2, 3 and 4.

The six tables below present the synthesis of our experiments. We have computed the following values:

- the mean ratio of the number of co-changes to the number of structural dependencies (SD)
- the mean percentage of structural dependencies that are also co-changes (calculated from the number of overlaps divided to the number of structural dependencies)
- the mean percentage of co-changes that are also structural dependencies (calculated from the number of overlaps divided to the number of co-changes)

In all the six tables, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12 we have on columns the values used for the commit size cs, while on rows we have the values for the number of occurrences threshold occ. The tables contain median values obtained for experiments done under all combinations of the two threshold values, on all test systems. In all tables, the upper right corner corresponds to the most relaxed filtering conditions,

while the lower left corner corresponds to the most restrictive filtering conditions.

Table 4.7: Ratio of number of co-changes to number of SD, case with comments

	$cs \leq 5$	$cs \leq 10$	$cs \leq 20$	$cs < \infty$
$occ \geq 1$	3,39	5,67	9,00	80,31
$occ \geq 2$	2,24	3,47	5,02	60,14
$occ \geq 3$	1,04	2,53	3,52	44,68
$occ \geq 4$	0,90	2,16	2,88	33,47

Table 4.8: Ratio of number of co-changes to number of SD, case without comments

	$cs \leq 5$	$cs \leq 10$	$cs \leq 20$	$cs < \infty$
$occ \geq 1$	3,24	5,33	7,90	67,16
$occ \geq 2$	1,35	3,27	4,72	47,39
$occ \geq 3$	1,00	1,67	2,49	32,39
$occ \geq 4$	0,43	1,26	1,93	22,15

Table 4.9: Percentage of SD that are also co-changes, case with comments

	$cs \leq 5$	$cs \leq 10$	$cs \leq 20$	$cs < \infty$
$occ \geq 1$	19,75	29,86	39,29	76,59
$occ \geq 2$	12,50	20,20	27,68	66,11
$occ \geq 3$	8,49	14,22	19,94	55,99
$occ \geq 4$	6,58	10,95	15,76	47,12

Table 4.10: Percentage of SD that are also co-changes, case without comments

	$cs \leq 5$	$cs \leq 10$	$cs \leq 20$	$cs < \infty$
$occ \geq 1$	18,88	28,47	37,44	71,12
$occ \geq 2$	11,87	19,03	25,93	59,58
$occ \geq 3$	8,00	13,09	18,15	48,65
$occ \geq 4$	5,85	9,94	14,27	39,07

Table 4.11: Percentage of co-changes that are also SD, case with comments

	$cs \leq 5$	$cs \leq 10$	$cs \leq 20$	$cs < \infty$
$occ \geq 1$	12,02	8,86	6,72	1,79
$occ \geq 2$	15,05	11,71	9,38	2,21
$occ \geq 3$	17,45	13,97	11,57	2,86
$occ \geq 4$	18,96	15,28	12,94	3,67

In order to assess the influence of comments, we compare pairwise Tables 4.7 and 4.8, Tables 4.9 and 4.10 and Tables 4.11 and 4.12. We observe that, although there are some differences between pairs of measurements done in similar conditions with and without comments, the differences are not significant.

On the other hand, the overlap between structural and co-changes is given by the number of pairs of classes that have both structural and co-change dependencies. We evaluate this overlap as a percentage relative to the number of structural

Table 4.12: Percentage of co-changes that are also SD, case without comments

	$cs \leq 5$	$cs \leq 10$	$cs \leq 20$	$cs < \infty$
$occ \geq 1$	12,05	9,02	6,98	1,93
$occ \geq 2$	15,08	12,03	9,66	2,42
$occ \geq 3$	17,78	14,37	12,24	3,28
$occ \geq 4$	19,22	15,59	13,30	4,21

Table 4.13: Percentage of SD that are also co-changing pairs after connection strength filtering.

Condition	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$
factor A and factor B	11.20	6.80	4.44	3.25	2.58	1.74	1.16	0.57	0.35	0.33
factor A or factor B	15.94	11.02	7.56	5.59	4.52	2.90	2.00	1.33	1.04	1.02

Table 4.14: Percentage of co-changing pairs that are SD after connection strength filtering.

Condition	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$
factor A and factor B	10.95	20.61	23.73	26.75	28.57	33.31	33.43	38.34	42.52	39.41
factor A or factor B	12.19	16.85	19.41	20.70	21.63	22.84	21.86	23.08	24.00	22.73

dependencies in Tables 4.9,4.10 and 4.13, respectively as a percentage relative to the number of co-changes in Tables 4.11,4.12, 4.14.

A first observation from Tables 4.9, 4.10, and 4.13 is that not all pairs of classes with structural dependencies co-change. The biggest value for the percentage of structural dependencies that are also co-changes is 76.5% obtained in the case when no filterings are done.

From Tables 4.11, 4.12, and 4.14 we notice that the percentage of co-changes which are also structural is always low to very low. This means that most co-changes are recorded between classes that have no structural dependencies to each other [66].

5. STATE OF THE ART IN KEY CLASSES DETECTION AND RESULTS EVALUATION

5.1. Definition

Zaidman et al [67] were the first to introduce the concept of key classes and it refers to classes that can be found in documents written to provide an architectural overview of the system or an introduction to the system structure. Tahvildari and Kontogiannis have a more detailed definition regarding key classes concept: "Usually, the most important concepts of a system are implemented by very few key classes which can be characterized by the specific properties. These classes, which we refer to as key classes, manage many other classes or use them in order to implement their functionality. The key classes are tightly coupled with other parts of the system. Additionally, they tend to be rather complex, since they implement much of the legacy system's functionality" [68]. Also, other researchers use a similar concept as the one defined by Zaidman but under different terms like important classes [69] or central software classes [70].

The key class identification can be done by using different algorithms with different inputs. In the research of Osman et al., the key class identification is made by using a machine learning algorithm and class diagrams as input for the algorithm [71]. Thung et al. builds on top of Osman et al.'s approach and adds network metrics and optimistic classification in order to detect key classes [72].

Zaidman et al. use a webmining algorithm and dynamic analysis of the source code to identify the key classes [67].

Sora et al. use a page ranking algorithm for finding key classes and static analysis of the source code [31], [73], [74], [75]. In [76] the authors use in addition to the previous research also other class attributes to identify important classes. The page ranking algorithm is a customization of PageRank, the algorithm used to rank web pages [77]. The PageRank algorithm works based on a recommendation system. If one node has a connection with another node, then it recommends the second node. In previous works, connections are established based on structural dependencies extracted from static code analysis. If A has a structural dependency with B, then A recommends B, and also B recommends A.

The ranking algorithm ranks all the classes from the source code of the system analyzed according to their importance. To identify the important classes from the rest of the classes a threshold for TOP classes from the top of the ranking is set. The TOP threshold value can go from 1 to the total number of classes found in the system.

Some researchers [67], [78], [79] consider that 15% of the total number of classes of the system is a suited value for the TOP threshold. Other researchers [76] consider that 15% of the total number of classes is a too high value for the TOP

threshold and suggest that a value in the range of 20–30 is better.

5.2. Metrics for results evaluation

To evaluate the quality of the key classes ranking algorithm and solution produced, the key classes found by the algorithm are compared with a reference solution.

The reference solution is extracted from the developer documentation. Classes mentioned in the documentation are considered key classes and form the reference solution (ground truth) used for validation [80].

For the comparison between both solutions, is used a classification model. The quality of the solution produced is evaluated by using metrics that evaluate the performance of the classification model, such as Precision-Recall and Receiver Operating Characteristic Area Under Curve (ROC-AUC).

A classification model (or ”classifier”) is a mapping between expected results and predicted results [81], [82]. Both results can be labeled as positive or negative, which leads us to the confusion matrix from figure 5.1. The confusion matrix has the

Expected Result \ Predicted Result	Positive	Negative
Positive	*True Positive*	*False Positive*
Negative	*False Negative*	*True Negative*

Figure 5.1: Confusion matrix

following outcomes:

- *true positive*, if the expected result is positive and the predicted result is also positive.
- *false positive*, if the expected result is positive but the predicted result is negative.
- *false negative*, if the expected result is negative but the predicted result is positive.
- *true negative*, if the expected result is negative and the predicted result is also negative.

Precision-recall

Precision is the ratio of True Positives to all the positives of the result set.

$$precision = \frac{TP}{TP + FN} \tag{5.1}$$

The recall is the ratio of True Positives to all the positives of the reference set.

$$recall = \frac{TP}{TP + FP} \tag{5.2}$$

As mentioned in section 5.1, to distinguish the key classes from the rest of the classes a TOP threshold is used. Some researchers consider that 15% of the total classes is the best value for the TOP threshold and others consider that the value should be in the range of 20-30.

The precision-recall metric is suited if the threshold value is fixed. If the threshold value is variable, then metrics that capture the behavior over all possible values must be used. Such metric is the Receiver Operating Characteristic metric.

Receiver Operating Characteristic Area Under Curve

The ROC graph is a two-dimensional graph that has on the X-axis plotted the false positive rate and on the Y-axis the true positive rate. By plotting the true positive rate and the false positive rate at thresholds that vary between a minimum and a maximum possible value we obtain the ROC curve. The area under the ROC curve is called Area Under the Curve (AUC).

The true positive rate of a classifier is calculated as the division between the number of true positive results identified and all the positive results identified:

$$True\ positive\ rate(TPR) = \frac{TP}{TP + FN} \tag{5.3}$$

The false positive rate of a classifier is calculated as the division between the number of false positive results identified and all the negative results identified:

$$False\ positive\ rate(FPR) = \frac{FP}{FP + TN} \tag{5.4}$$

In multiple related works, the ROC-AUC metric has been used to evaluate the results for finding key classes of software systems. For a classifier to be considered good, its ROC-AUC metric value should be as close to 1 as possible, when the value is 1 then the classifier is considered to be perfect.

Osman et al. obtained in their research an average Area Under the Receiver Operating Characteristic Curve (ROC-AUC) score of 0.750 [71]. Thung et al. obtained an average ROC-AUC score of 0.825 [72] and Sora et al. obtained an average ROC-AUC score of 0.894 [76].

5.3. Baseline approach

We use the research of I. Sora et al [76] as a baseline for our research involving the usage of logical dependencies to find key classes. The baseline approach uses a tool that takes as an input the source code of the system and applies ranking strategies to rank the classes according to their importance.

In order to rank the classes according to their importance, different class metrics are used [78], [67], [79]. Below are presented some of the class metrics used in the baseline approach in order to rank the classes according to their importance.

Class attributes that characterize key classes

The metrics used in the baseline research can be grouped into the following categories:

- class size metrics: number of fields (NoF), number of methods (NoM), global size (Size = NoF+NoM).
- class connection metrics, any structural dependency between two classes:
  - CONN-IN, the number of distinct classes that use a class;
  - CONN-OUT, the total number of distinct classes that are used by a class;
  - CONN-TOTAL, the total number of distinct classes that a class uses or are used by a class (CONN-IN + CONN-OUT).
  - CONN-IN-W, the total weight of distinct classes that use a class.
  - CONN-OUT-W, the total weight of distinct classes that are used by a class.
  - CONN-TOTAL-W, the total weight of all connections of the class (CONN-IN-W + CONN-OUT-W) [76].
- class pagerank values, previous research use pagerank values computed on both directed and undirected, weighted and unweighted graphs:
  - PR - value computed on the directed and unweighted graph;
  - PR-W - value computed on the directed and weighted graph;
  - PR-U - value computed on the undirected and unweighted graph;
  - PR-U-W - value computed on the undirected and weighted graph;
  - PR-U2-W - value computed on the weighted graph with back-recommendations [31], [73], [76], [75].

Based on the class attributes presented, all the classes of the system are ranked. To differentiate the important (key) classes from the rest of the classes, a TOP threshold for the top classes found is set. The threshold vary between 20 and 30 classes.

The baseline approach not only identifies the key classes but also evaluates the performance of the solution produced. The same approach as the one presented in section 5.2 is used for the evaluation of the results. The key classes found by the ranking algorithm are compared with a reference solution that is extracted from the developer documentation by using a classification model.

The true positives (TP) are the classes found in the reference solution and also in the top TOP ranked classes. False positives (FP) are the classes that are not in the reference solution but are in the TOP ranked classes. True Negatives (TN) are classes

that are found neither in the reference solution nor in the TOP ranked classes. False Negatives (FN) are classes that are found in the reference solution but not found in the TOP ranked classes.

Due to the fact that the TOP threshold is varied, the Receiver Operating Characteristic Area Under Curve metric is used for the evaluation of the results.

The entire workflow of the baseline approach that was presented above is also presented in figure 5.2.

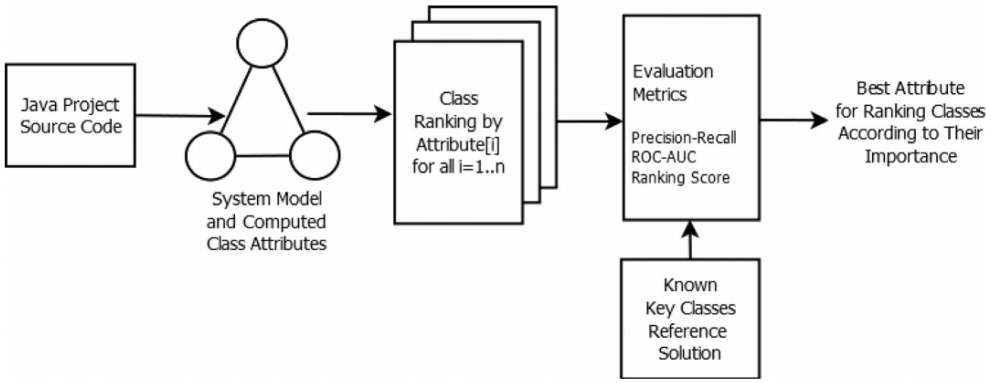

Figure 5.2: Overview of the baseline approach. Reprinted from "Finding key classes in object-oriented software systems by techniques based on static analysis." by Ioana Sora and Ciprian-Bogdan Chirila, 2019, Information and Software Technology, 116:106176. Reprinted with permission.

6. KEY CLASSES DETECTION USING LOGICAL DEPENDENCIES

6.1. Data set used

In this section, we will look over all the systems studied in the baseline research presented in section 5.3, and we will try to identify the systems that could be used also in our current research involving logical dependencies.

The research of I. Sora et al [76] takes into consideration structural public dependencies that are extracted using static analysis techniques and was performed on the object-oriented systems presented in table 6.1.

The requirements for a system to qualify as suited for investigations using logical dependencies are: has to be on GitHub, has to have release tags to identify the version, and also has to have an increased number of commits. From the total of 14 object-oriented systems listed in the paper [76], 13 of them have repositories in Github 6.2. And from the found repositories we identified only 6 repositories that have the same release tag as the specified version from table 6.1. It is important to identify the correct release tag for each repository to limit the commits further analyzed by date. Only commits that were made until the specified release are considered and analyzed. The commits number found on the remaining 6 repositories varies from 19108 commits for Tomcat Catalina to 149 commits for JHotDraw. In order to have more accurate results, we need a significant number of commits, so we reached the conclusion that only 3 systems can be used for key classes detection using logical dependencies: Apache Ant, Hibernate, and Tomcat Catalina. From all the systems mentioned in table 6.1 Apache Ant is the most used and analyzed in other works [66], [83], [84], [85].

Table 6.1: Analyzed software systems in previous research paper.

ID	System	Description
Sl	Apache Ant	Java library and command line tool that drive the build processes as targets and e
S2	Argo UML	UML modelling tool with support for all UML di
S3	GWT Portlets	Open source web framework for building GWT (Google Web
S4	Hibernate	Persistence framework for Java.
S5	javaclient	Java distributed application for playing with
S6	jEdit	Java mature text editor for programmer
S7	JGAP	Genetic Algorithms and Genetic Programming Ja
S8	JHotDraw	JHotDraw is a two-dimensional graphics framework for structured drawi
S9	JMeter	JMeter is a Java application designed to load test functional behavio
S10	Log4j	Logging Service
S11	Mars	The Mars Simulation Project is a Java project that models and simulates
S12	Maze	The Maze-solver project simulates an artificial intelligence
S13	Neuroph	Neuroph is a Java neural network framew
S14	Tomcat Catalina	The Apache Tomcat project is an open-source implementation of JavaServl
S15	Wro4J	The Wro4J is a web resource (JS and CSS) optimiz

Table 6.2: Found systems and versions of the systems in GitHub.

ID	System	Version	Release Tag name	Commits number
Sl	Apache Ant	1.6.1	rel/1.6.1	6713
S2	Argo UML	0.9.5	not found	0
S3	GWT Portlets	0.9.5 beta	not found	0
S4	Hibernate	5.2.12	5.2.12	6733
S5	javaclient	2.0.0	not found	0
S6	jEdit	5.1.0	not found	0
S7	JGAP	3.6.3	not found	0
S8	JHotDraw	6.0b.1	not found	149
S9	JMeter	2.0.1	v2_1_1	2506
S10	Log4j	2.10.0	v1_2_10-recalled	634
S11	Mars	3.06.0	not found	0
S12	Maze	1.0.0	not found	0
S13	Neuroph	2.2.0	not found	0
S14	Tomcat Catalina	9.0.4	9.0.4	19108
S15	Wro4J	1.6.3	v1.6.3	2871

6.2. Measurements using logical dependencies

As we mentioned in the beginning the purpose is to check if the logical dependencies can improve key class detection.

As presented in section 5.3, and section 5.1 the key class detection was done by using structural dependencies of the system. In this section, we will use the same tool used in the baseline approach presented in section 5.3, and we will add a new input to it, the logical dependencies.

Below is a comparison between the new approach and baseline approach, how we collect the logical dependencies, the results obtained previously, and the new results obtained. The new results are separated into two categories, the results obtained

by using structural and logical dependencies and the results obtained by using only logical dependencies.

6.2.1. Comparison with the baseline approach

The baseline approach uses a tool that takes as input the source code of the system to identify the key classes and the reference solution to evaluate the quality of the solution. We modified the tool such that it can also take as input the logical dependencies.

In order to rank the classes according to their importance, the tool uses different class metrics. The list of the metrics used in the baseline approach is presented in section 5.3. The difference in the metrics used compared with the baseline approach is that we use a subset of those metrics. The reason why we are not using all the metrics is that the extracted logical dependencies are undirected. The metrics used by the current approach are CONN-TOTAL, CONN-TOTAL-W, PR-U, PR-U-W, and PR-U2-W.

We did not change the rest of the workflow of the tool. Meaning that the TOP threshold is varied between 20 and 30 and the resulting solution is evaluated by using the ROC-AUC metric. The goal being a ROC-AUC (Receiver Operating Characteristic - Area Under the Curve) metric value as close to 1 as possible.

	Baseline approach	Current approach
Input data:	- source code for key class identification - reference solution for result evaluation	- source code ***and logical dependencies*** for key class identification - reference solution for result evaluation
Attributes for key class identification:	- all attributes presented in section 2.1.3	- a subset of the baseline approach attributes
Results evaluation:	- the quality of the results is evaluated by using a classification model and ROC-AUC metric	- same as in the baseline approach

Figure 6.1: Comparison between the new approach and the baseline

6.2.2. Logical dependencies collection and current workflow used

The logical dependencies are those co-changing pairs extracted from the versioning system history that remain after filtering. The filtering part consists of applying two filters: the filter based on commit size and the filter based on connection strength.

To determine the connection strength of a pair, we first need to calculate the connection factors for both entities that form a co-changing pair. Assuming that we

have a co-changing pair formed by entities A and B, the connection factor of entity A with entity B is the percentage from the total commits involving A that contains entity B. The connection factor of entity B with entity A is the percentage from the total commits involving B that contain also entity A.

$$connection\ factor\ for\ A = \frac{100 * commits\ involving\ A\ and\ B}{total\ nr\ of\ commits\ involving\ A} \tag{6.1}$$

$$connection\ factor\ for\ B = \frac{100 * commits\ involving\ A\ and\ B}{total\ nr\ of\ commits\ involving\ B} \tag{6.2}$$

We calculated the connection factor for each entity involved in a co-changing pair and filtered the co-changing pairs based on it. The rule set is that both entities had to have a connection factor with each other greater than the threshold value.

After the filtering part, the remaining co-changing pairs, now called logical dependencies, are exported in CSV files.

The entire process of extracting co-changing pairs from the versioning system, filter them, and export the remaining ones into CSV files is done with a tool written in Python.

The next step is to use the exported logical dependencies for key classes detection. In order to do that we used the same key class detection tool used in the previous research presented in section 5.3. We adapted the tool to be able to process also logical dependencies because previously the tool used only structural dependencies extracted from the source code of the software systems. The workflow is presented in figure 6.2

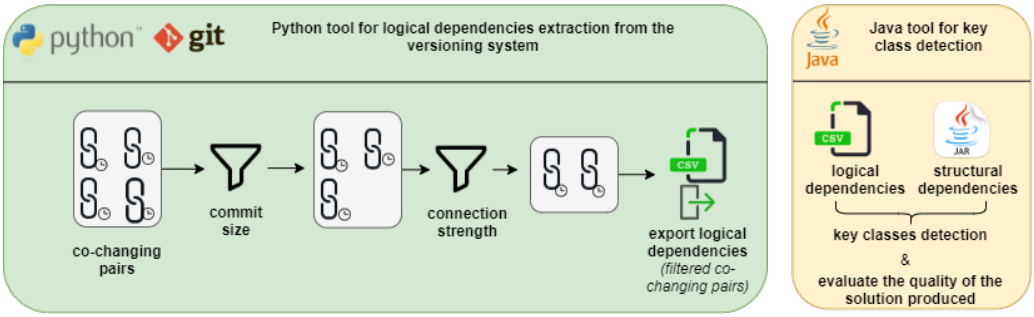

Figure 6.2: Workflow for key classes detection

6.2.3. Measurements using only the baseline approach

In table 6.3 are presented the ROC-AUC values for different attributes computed for the systems Ant, Tomcat Catalina, and Hibernate by using the baseline approach. We intend to compare these values with the new values obtained by using also logical dependencies in key class detection.

Table 6.3: ROC-AUC metric values extracted.

Metrics	Ant	Tomcat Catalina	Hibernate
PR_U2_W	0.95823	0.92341	0.95823
PR	0.94944	0.92670	0.94944
PR_U	0.95060	0.93220	0.95060
CONN_TOTAL_W	0.94437	0.92595	0.94437
CONN_TOTAL	0.94630	0.93903	0.94630

6.2.4. Measurements using combined structural and logical dependencies

The tool used in the baseline approach runs a graph-ranking algorithm. The graph used contains the structural dependencies extracted from static source code analysis. Each edge in the graph represents a dependency, the entities that form a structural dependency are represented as vertices in the graph. As mentioned in section 6.2.1, we modified the tool to read also logical dependencies and add them to the graph. In this section, we add in the graph the logical dependencies together with the structural dependencies.

In tables 6.4, 6.5, and 6.6, on each line, we have the metric that is calculated and on each column, we have the connection strength threshold that was applied to the logical dependencies used in identifying the key classes. We started with logical dependencies that have a connection strength greater than 10%, which means that in at least 10% of the commits involving A or B, A and B update together. Then we increased the threshold value by 10 until we remained only with entities that update in all the commits together. The last column contains the results obtained previously by the tool by only using structural dependencies.

As for the new results obtained by combining structural and logical dependencies, highlighted with orange are the values that are close to the previously registered values but did not surpass them. Highlighted with green are values that are better than the previously registered values. At this step, we can also observe that for all three systems measured in tables 6.4, 6.5, and 6.6, the best values obtained are for connection strength between 40-70%.

Table 6.4: Measurements for Ant using structural and logical dependencies combined

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Baseline
PR_U2_W	0.924	0.925	0.926	0.927	0.927	0.927	0.929	0.928	0.928	0.928	0.929
PR	0.914	0.854	0.851	0.866	0.876	0.882	0.887	0.854	0.852	0.852	0.855
PR_U	0.910	0.930	0.933	0.933	0.935	0.934	0.939	0.933	0.933	0.933	0.933
CON_T_W	0.924	0.928	0.931	0.932	0.933	0.934	0.936	0.934	0.934	0.934	0.934
CON_T	0.840	0.886	0.904	0.909	0.915	0.923	0.932	0.935	0.936	0.936	0.942

6.2.5. Measurements using only logical dependencies

In the previous section, we added in the graph based on which the ranking algorithm works the logical and structural dependencies. In the current section, we will add only the logical dependencies to the graph.

Table 6.5: Measurements for Tomcat using structural and logical dependencies combined

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Baseline
PR_U2_W	0.910	0.917	0.923	0.924	0.924	0.924	0.924	0.924	0.924	0.924	0.923
PR	0.811	0.800	0.815	0.834	0.847	0.852	0.853	0.858	0.858	0.858	0.927
PR_U	0.910	0.921	0.931	0.933	0.933	0.932	0.933	0.932	0.932	0.932	0.932
CON_T_W	0.914	0.920	0.924	0.926	0.926	0.926	0.926	0.926	0.926	0.926	0.926
CON_T	0.868	0.906	0.930	0.936	0.937	0.938	0.938	0.938	0.938	0.938	0.939

Table 6.6: Measurements for Hibernate using structural and logical dependencies combined

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Baseline
PR_U2_W	0.954	0.957	0.958	0.958	0.958	0.958	0.958	0.958	0.958	0.958	0.958
PR	0.929	0.929	0.933	0.939	0.939	0.946	0.947	0.947	0.947	0.947	0.949
PR_U	0.942	0.947	0.948	0.949	0.949	0.950	0.950	0.950	0.950	0.950	0.951
CON_T_W	0.939	0.942	0.943	0.944	0.944	0.945	0.945	0.945	0.945	0.945	0.944
CON_T	0.924	0.933	0.938	0.941	0.941	0.944	0.945	0.945	0.945	0.945	0.946

In tables 6.7, 6.8, and 6.9, are presented the results obtained by using only logical dependencies to detect key classes. The measurements obtained are not as good as using logical and structural dependencies combined or using only structural dependencies. But, all the values obtained are above 0.5, which means that a good part of the key classes is detected by only using logical dependencies. As mentioned in section 5.2, a classifier is good if it has the ROC-AUC value as close to 1 as possible.

One possible explanation for the less performing results is that the key classes may have a better design than the rest of the classes, which means that are less prone to change. If the key classes are less prone to change, this implies that the number of dependencies extracted from the versioning system can be less than for other classes.

Table 6.7: Measurements for Ant using only logical dependencies

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Baseline
PR_U2_W	0.720	0.627	0.718	0.703	0.732	0.824	0.852	0.881	0.876	0.876	0.929
PR	0.720	0.627	0.718	0.703	0.732	0.824	0.852	0.881	0.876	0.876	0.855
PR_U	0.720	0.627	0.718	0.703	0.732	0.824	0.852	0.881	0.876	0.876	0.933
CON_T_W	0.722	0.581	0.644	0.676	0.727	0.819	0.842	0.874	0.876	0.876	0.934
CON_T	0.722	0.581	0.644	0.676	0.727	0.819	0.842	0.874	0.876	0.876	0.942

Table 6.8: Measurements for Tomcat using only logical dependencies

Metrics	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$	Previous
PR_U2_W	0.672	0.656	0.645	0.697	0.754	0.776	0.786	0.799	0.799	0.799	0.923
PR	0.685	0.643	0.642	0.697	0.754	0.776	0.786	0.799	0.799	0.799	0.927
PR_U	0.685	0.643	0.644	0.697	0.754	0.776	0.786	0.799	0.799	0.799	0.932
CON_T_W	0.694	0.636	0.636	0.697	0.754	0.776	0.786	0.799	0.799	0.799	0.926
CON_T	0.654	0.611	0.636	0.697	0.754	0.776	0.786	0.799	0.799	0.799	0.939

Table 6.9: Measurements for Hibernate using only logical dependencies

Metrics	≥ 10%	≥ 20%	≥ 30%	≥ 40%	≥ 50%	≥ 60%	≥ 70%	≥ 80%	≥ 90%	≥ 100%	Baseline
PR_U2_W	0.657	0.564	0.601	0.619	0.622	0.650	0.653	0.654	0.654	0.654	0.958
PR	0.644	0.564	0.601	0.619	0.622	0.650	0.653	0.654	0.654	0.654	0.949
PR_U	0.644	0.564	0.601	0.619	0.622	0.650	0.653	0.654	0.654	0.654	0.951
CON_T_W	0.649	0.564	0.601	0.619	0.622	0.650	0.653	0.654	0.654	0.654	0.944
CON_T	0.644	0.564	0.601	0.619	0.622	0.650	0.653	0.654	0.654	0.654	0.946

6.3. Correlation between details of the systems and results

In this section, we discuss about the correlation between the details of the systems and the results obtained in section 6.2.

The reason why we are doing this correlation is to find if there are some links between the details of the systems and the results obtained.

The results obtained are presented in figures 6.3 - 6.8. We are using plots to display the results obtained to have a clearer view of how the results fluctuate over different thresholds values.

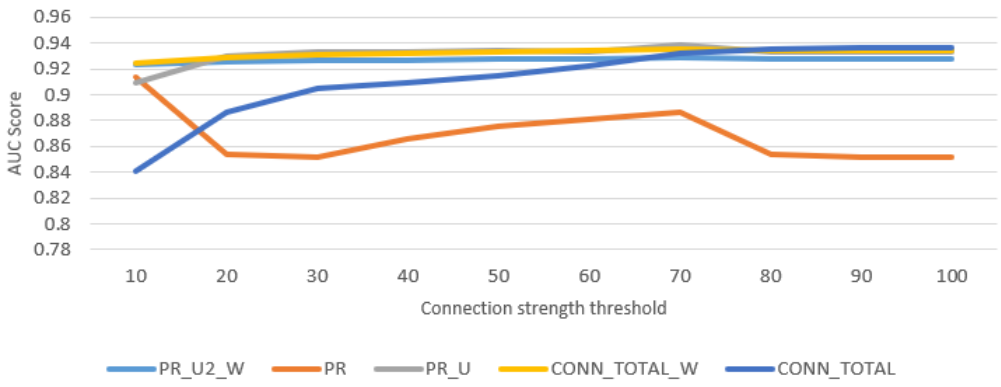

Figure 6.3: Variation of AUC score when varying connection strength threshold for Ant. Results for structural and logical dependencies combined.

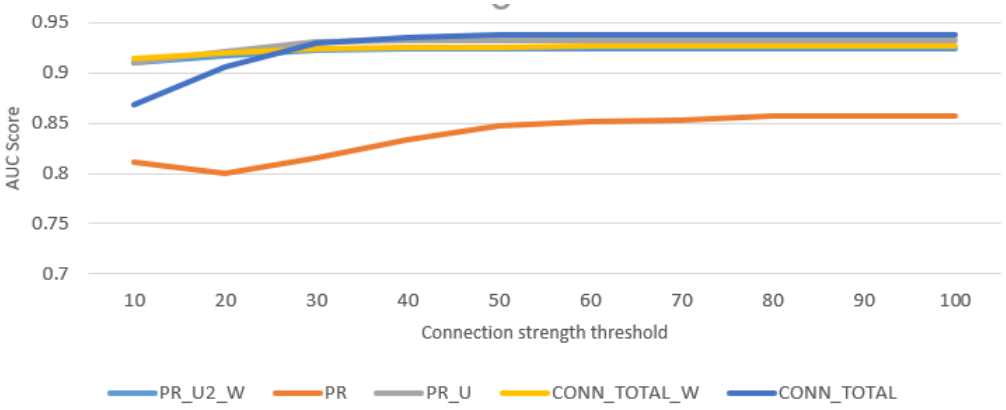

Figure 6.4: Variation of AUC score when varying connection strength threshold for Tomcat. Results for structural and logical dependencies combined.

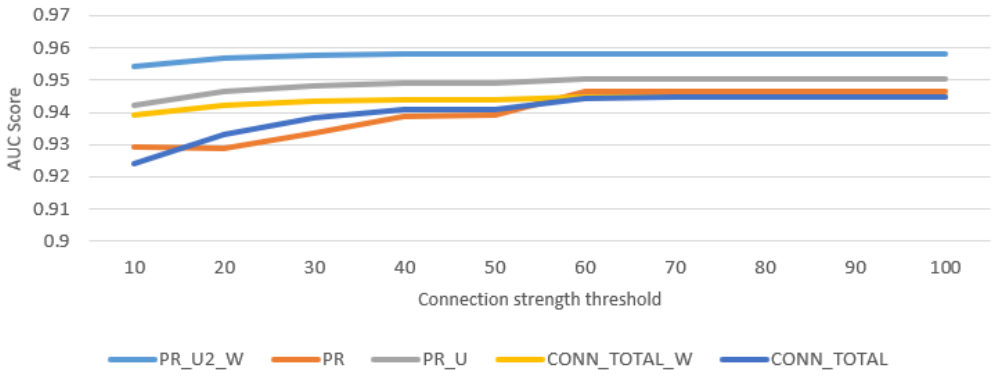

Figure 6.5: Variation of AUC score when varying connection strength threshold for Hibernate. Results for structural and logical dependencies combined.

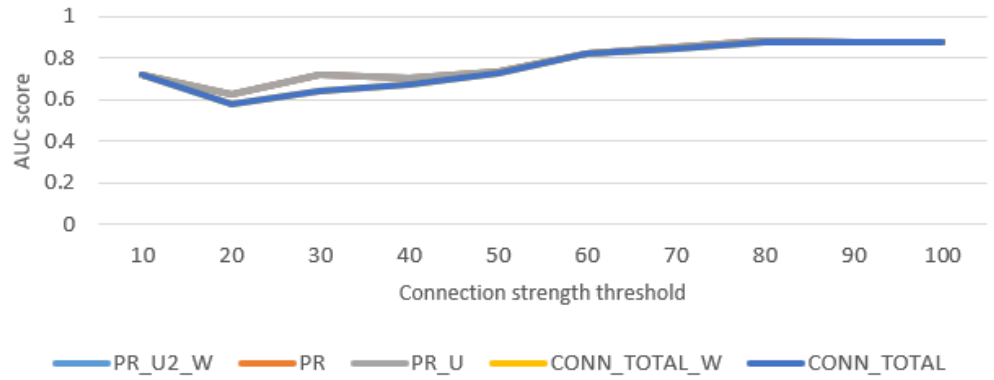

Figure 6.6: Variation of AUC score when varying connection strength threshold for Ant. Results for logical dependencies only.

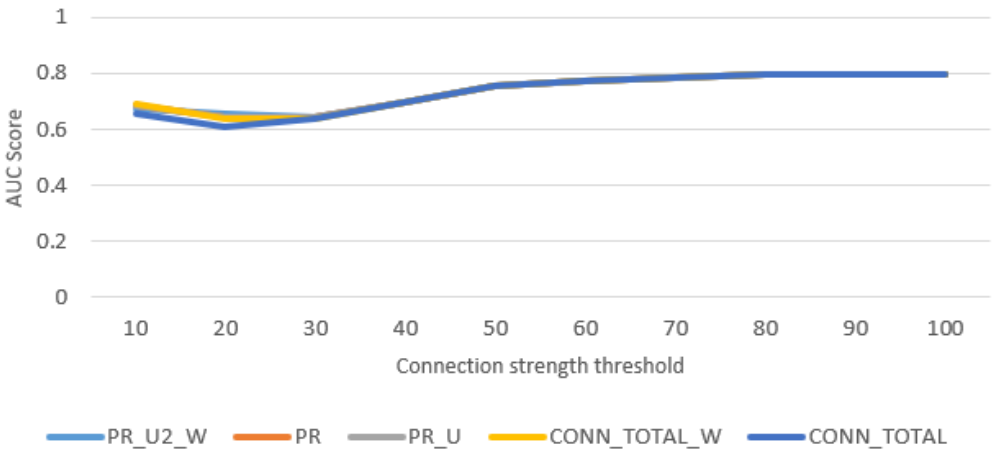

Figure 6.7: Variation of AUC score when varying connection strength threshold for Tomcat. Results for logical dependencies only.

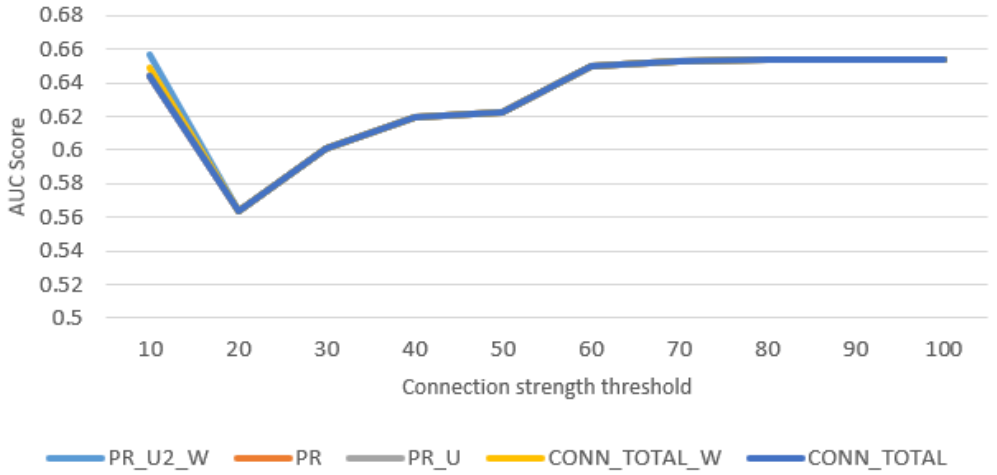

Figure 6.8: Variation of AUC score when varying connection strength threshold for Hibernate. Results for logical dependencies only.

The details of the systems are presented in two tables. In table 6.10 are the overlappings between structural and logical dependencies expressed in percentages. Each column represents the percentage of logical dependencies that are also structural, for each column the logical dependencies are obtained by applying a different connection strength filter. The connection strength filter begins at 10, meaning that in at least 10 % of the total commits involving two entities, the entities update together. We increase the connection strength filter by 10 up until we reach 100, meaning that in all the commits that involve one entity, the other entity is present also.

In table 6.11 are the ratio numbers between structural dependencies and logical dependencies. We added this table in order to highlight how different the total number of both dependencies is.

Table 6.10: Percentage of logical dependencies that are also structural dependencies

System	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$
Ant	25.202	34.419	36.385	34.656	33.528	33.333	28.659	33.333	35.294	35.294
Tomcat Catalina	4.059	22.089	25.000	25.758	25.926	37.525	47.368	55.285	75.000	76.923
Hibernate	6.546	26.607	29.565	32.374	32.543	45.170	44.980	42.473	42.473	42.473

Table 6.11: Ratio between structural and logical dependencies (SD/LD)

System	$\geq 10\%$	$\geq 20\%$	$\geq 30\%$	$\geq 40\%$	$\geq 50\%$	$\geq 60\%$	$\geq 70\%$	$\geq 80\%$	$\geq 90\%$	$\geq 100\%$
Ant	1.315	3.284	4.972	5.603	6.175	10.697	12.915	27.154	41.529	41.529
Tomcat Catalina	0.120	0.923	1.313	1.531	1.619	3.177	7.092	13.146	67.375	124.385
Hibernate	1.037	6.391	10.037	14.947	18.940	54.248	83.442	111.704	111.704	111.704

In figures 6.3, 6.4 and 6.5 are the measurements obtained by using structural and logical dependencies combined. In all three figures, the measurements at the beginning are smaller than the rest. Once with the increasing of the threshold value

also the measurements begin to increase. Meaning that better results for key class detection are found. The best measurements are when the threshold value is between 40 and 60, after that, the measurements tend to decrease a little bit and stay at that fixed value.

A possible explanation of the results fluctuation and then capping is that if we are looking at table 6.11 we can see that at the beginning, the total number of logical dependencies used is close to the number of existing structural dependencies. The high volume of logical dependencies introduced might cause an erroneous detection of the key classes, in consequence, smaller measurements. When the threshold begins to be more restrictive and the total number of logical dependencies used begins to decrease, the key classes detection starts to improve. This improvement stops after the threshold value reaches 60%. If we look again at table 6.11 we can see that after 60% the number of structural dependencies outnumbers the number of logical dependencies up to 124 times in some cases. In addition, if we look at table 6.10 we can see that the remaining logical dependencies overlap a lot with the structural dependencies, so we are not introducing too much new information.

So, the number of logical dependencies used is so small that it doesn't influence the key class identification. Since the structural dependencies used don't change, we obtain the same results for different threshold values.

In figures 6.6, 6.7 and 6.8 are the measurements obtained by using only logical dependencies. Initially, we expected to see a Gaussian curve, but instead, we see a bell curve. We think that in the beginning, we use a high number of logical dependencies in key class detection, among those logical dependencies is an important number of key classes and also an important number of other classes. But the number of other classes does not influence the key classes detection. When we start to increase the value of the threshold and filter more the logical dependencies, we also filter some of the initial detected key classes and remain with a significant number of other classes. In this case, the other classes that remain influence the measurements, causing the worst-performing solutions. Some of the key classes are strongly connected in the versioning system, and even for higher threshold values don't get filtered out. Meanwhile, the rest of the classes that are not key classes get filtered out for higher threshold values which leads to better performing measurements when the threshold value are above 60%.

6.4. Comparison of the extracted data with fan-in and fan-out metric

Fan-in and fan-out are coupling metrics. The fan-in of entity A is the total number of entities that call functions of A. The fan-out of A is the total number of entities called by A [86].

In tables 6.12, 6.13, and 6.14 we can find the metrics detalis for each documented key class of each system. The first column represents the name of each key class, the second column represents the fan_in values for each key class, the third column represents the fan_out values, the fourth column represents the number of entities that call functions of that key class plus the number of entities that are called

by the key class (fan_in and fan_out combined), and the fifth column represents the number of logical dependencies in which an entity is involved.

For Ant, we can see in table 6.12 that all the key classes have logical dependencies with other classes. The LD_NUMBER means the number of logical dependencies of an entity. The key classes with the most LD number are Project and IntrospectionHelper, these two entities can be found also in table 6.15 in which we did a top 10 entities that have a logical dependency with other entities. This means that some key classes are involved in software change quite often and can be observed via system history.

Table 6.12: Measurements for Ant key classes

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NUMBER
1	Project	191	23	214	157
2	Target	28	6	34	78
3	UnknownElement	17	13	30	90
4	RuntimeConfigurable	17	13	30	118
5	IntrospectionHelper	18	24	42	143
6	Main	1	13	14	82
7	TaskContainer	11	1	12	21
8	ProjectHelper2$ElementHandler	1	12	13	30
9	Task	110	7	117	88
10	ProjectHelper	16	8	24	101

For Tomcat Catalina, same as for Ant, we can see in table 6.13 that all the key classes have logical dependencies. The key classes with the most LD number are StandardContext and Request, these two entities can also be found in table 6.16 in which we did a top 10 entities that have the most logical dependencies with other entities for Tomcat Catalina.

For Hibernate things are a little bit different, as we can see in table 6.14, key classes like Criterion, Projection, or Transaction have 0 logical dependencies, meaning that those key classes are not involved in any software change. One possible explanation for this is that for Hibernate the architecture is designed in such way that the core is not often touched by change.

In tables 6.15, 6.16, and 6.17 we can find the top 10 entities with logical dependencies. The first column represents the name of each top 10 entity, the second column represents the fan_in values, the third column represents the fan_out values, the fourth column represents the fan_in and fan_out combined, and the fifth column represents the number of logical dependencies in which the entity is involved.

We did these top 10 tables to offer an overview of the highest registered numbers for LD for each system. As we mentioned before, some of the key classes are also present in these tables, but not all of them.

In table 6.17 we can find the top 10 measurements for Hibernate, most of the table is occupied by inner classes of AbstractEntityPersister. This is expected behavior since class AbstractEntityPersister is also present. This behavior is caused by the impossibility to separate the updates done for a class from its inner classes in the versioning system. So, each time AbstractEntityPersister records a change, also the inner classes are considered to have changed.

Table 6.13: Measurements for Tomcat Catalina key classes.

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NUMBER
1	Context	74	8	82	126
2	Request	48	28	76	215
3	Container	51	8	59	64
4	Response	38	12	50	90
5	StandardContext	11	38	49	216
6	FANector	23	9	32	89
7	Session	29	2	31	28
8	Valve	29	2	31	19
9	Wrapper	29	1	30	36
10	Manager	25	3	28	31
11	Host	26	1	27	44
12	Service	20	6	26	51
13	Engine	23	2	25	1
14	Realm	18	6	24	21
15	CoyoteAdapter	1	22	23	140
16	StandardHost	8	15	23	88
17	LifecycleListener	21	1	22	3
18	StandardEngine	2	19	21	57
19	Pipeline	19	2	21	20
20	Server	16	4	20	49
21	HostConfig	3	15	18	79
22	StandardWrapper	5	13	18	92
23	StandardService	3	12	15	81
24	Catalina	2	13	15	94
25	Loader	14	1	15	18
26	StandardServer	2	12	14	94
27	StandardPipeline	1	10	11	62
28	Bootstrap	3	3	6	41

Table 6.14: Measurements for Hibernate key classes.

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NUMBER
1	SessionFactoryImplementor	438	43	481	51
2	Type	444	5	449	0
3	Table	89	29	118	82
4	SessionImplementor	52	12	64	14
5	Criteria	45	12	57	15
6	Column	46	10	56	20
7	Session	31	21	52	52
8	Query	12	28	40	0
9	Configuration	1	38	39	115
10	SessionFactory	24	12	36	33
11	Criterion	30	3	33	0
12	Projection	11	3	14	0
13	FANectionProvider	12	2	14	0
14	Transaction	11	1	12	0

Overall, by looking at the comparisons between FAN_IN, FAN_OUT, FAN_TOTAL, and the logical dependencies in which a class is involved we could not determine a direct connection between them. Nither we can say that one influences the other. We consider that even though the metrics are not related directly, they could be all used together to get a better view of the system connections.

Table 6.15: Top 10 measurements for Ant.

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NUMBER
1	Project	191	23	214	157
2	Project$AntRefTable	1	2	3	157
3	Path	39	13	52	147
4	Path$PathElement	3	2	5	147
5	IntrospectionHelper	18	24	42	143
6	IntrospectionHelper$AttributeSetter	8	1	9	143
7	IntrospectionHelper$Creator	3	5	8	143
8	IntrospectionHelper$NestedCreator	7	1	8	143
9	Ant	2	15	17	136
10	Ant$Reference	3	1	4	136

Table 6.16: Top 10 measurements for Tomcat Catalina.

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NUMBER
1	StandardContext	11	38	49	216
2	StandardContext$ContextFilterMaps	0	0	0	216
3	StandardContext$NoPluggabilityServletContext	0	0	0	216
4	Request	48	28	76	215
5	Request$SpecialAttributeAdapter	0	0	0	215
6	ApplicationContext	3	22	25	158
7	ApplicationContext$DispatchData	0	0	0	158
8	ContextConfig	3	26	29	143
9	ContextConfig$DefaultWebXmlCacheEntry	0	0	0	143
10	ContextConfig$JavaClassCacheEntry	0	0	0	143

Table 6.17: Top 10 measurements for Hibernate.

Nr.	Classname	FAN_IN	FAN_OUT	FAN_TOTAL	LD_NR
1	AvailableSettings	1	0	1	205
2	AbstractEntityPersister	9	143	152	190
3	AbstractEntityPersister$CacheEntryHelper	0	0	0	190
4	AbstractEntityPersister$InclusionChecker	0	0	0	190
5	AbstractEntityPersister$NoopCacheEntryHelper	0	0	0	190
6	AbstractEntityPersister$ReferenceCacheEntryHelper	0	0	0	190
7	AbstractEntityPersister$StandardCacheEntryHelper	0	0	0	190
8	AbstractEntityPersister$StructuredCacheEntryHelper	0	0	0	190
9	Dialect	265	104	369	176
10	SessionFactoryImpl$SessionBuilderImpl	1	25	26	167

BIBLIOGRAPHY

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[62] srcml; www.srcml.org.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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