Contents

Chapter 1

Presentation of the research topic

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.

In Chapter 2 is presented the state of the art for determining logical and structural dependencies and a series of applications of software dependencies. In Chapter 3 is presented a tool that is build to extract logical and structural dependencies from a set of open-source systems and a set of factors that can be used to filter the recordings of class co-changes such that valid logical dependencies can be identified. And finally, in Chapter 4 is presented the research content and the proposed stages of research.

Chapter 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 [12], [16].

Structural dependencies can be found by analyzing the source code [51], [13]. 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 [39], [32].

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 [39], [10].

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 [33], [48].

The change mini-cycle consists of the following phases [63]:

- 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. [1]
- 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 [62]. 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 [22]. Distributed version control systems (such as Git, Mercurial, Bazaar or Darcs) allows many developers to collaboratively develop their projects [40].

2.2.3 Definition

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 [64], [15]. Those changes can be found in the software history maintained by the versioning system. Gall [29], [30] identified as logical coupling between two modules the fact that these modules *repeatedly* change together during the historical evolution of the software system [7].

The concepts of logical coupling and logical dependencies were first used in different analysis tasks, all related to changes: for software change impact analysis [49], for identifying the potential ripple effects caused by software changes during software maintenance and evolution [45], [44], [47], [34] or for their link to deffects [61], [65].

The current trend recommends that general dependency management methods and tools should also include logical dependencies besides the structural dependencies [44], [4].

2.2.4 Current status of research

Oliva and Gerosa [44], [45] 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 [4] 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 [4] 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 [47], [34]. 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 [5]. 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 [65] 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 [25], [23], [24], 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 [26] shows that historical information has been rarely used in architectural reconstruction.

Another survey [55] 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 [26]. So reconstructing the architecture and verifying if still matches the reality is important [35].

Surveys also show that architectural reconstruction is mainly made based on structural dependencies [55], [26], 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 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.

Other works have tried to filter co-changes [44], [4]. 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 [4]. 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 [17] 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. [14] 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 [11], [31].

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. [26],[24], [6] ,[52],

[32].

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" [43], [42]. Detection and removal of clones can decrease software maintenance costs [8], [36].

2.3.4 Code smells

Code smells have been defined by Fowler [28] 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 [59]. Studies have found that smells may affect comprehension and possibly increase change and fault proneness [3], [37], [38]. 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 [46].

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], [19], [20]. Previous studies show that the proportion of resources and time allocated to maintenance may vary from 50% to 75% [41]. 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 [27].

2.3.6 Fault location

Debugging software is an expensive and mostly manual process. Of all debugging activities, fault localization, is the most expensive [60].

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.[2] [18]

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 [53], [54].

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. [56]

Chapter 3

Current status of doctoral research

3.1 Tool for measuring software dependencies

In order to build structural and logical dependencies we have developed a tool that takes as input the source code repository and builds the required software dependencies [58]. The workflow can be delimited by three major steps as it follows (Figure 3-1):

Step 1: *Extracting structural dependencies.*

Step 2: *Extracting logical dependencies.*

Step 3: *Processing the information extracted.*

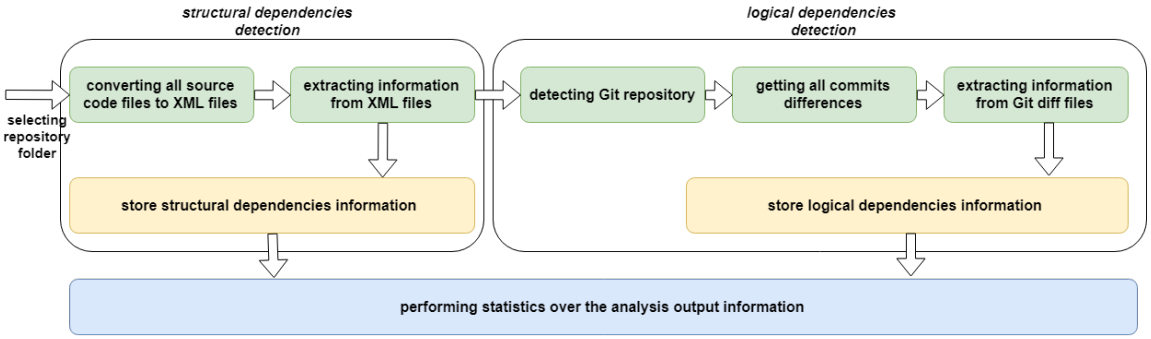


Figure 3-1: Processing phases

3.1.1 Extracting structural dependencies

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 system, 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 [20], [21] to convert all source code files from the current release into XML files. All the information about classes, methods, calls to other classes are afterwards extracted by our tool parsing the XML files and building a dependencies data structure. We have chosen to rely on srcML as a preprocessing tool because it reduces a significant number of syntactic differences from different programming languages and can make easier the parsing of source code written in different programming languages such as Java, C++ and C#.

3.1.2 Extracting logical dependencies

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 [22]. 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. We will take into consideration only *commits that have a parent* since the first commit can include source code files that are already in development (migration from one versioning system to another) and this can introduce redundant logical links [4].

The tool looks through the main branch of the project and gets all the existing commits. For each commit a diff against the parent will be made and stored. Here we have the option to ignore commits that contain more files than a threshold value for commit size. Also, we have the option to check whether the differences are in actual code or if they affect only parts of source files that are only comments. Finally after all the difference files are stored, all the files are parsed and logical dependencies are build. For a group of files that are committed together, logical dependencies are added between all pairs formed by members of the group. Adding a logical dependency increases an occurrence counter for the logical link.

We have analyzed a set of open-source projects found on GitHub[1] [35] in order to extract the structural and logical dependencies between classes. Table 3.1 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)

[1]http://github.com/

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 3.1: Summary of open source projects studied.

ID	Project	Nr. of entites	Nr. of commits	Type
1	bluecove	586	894	java
2	aima-java	987	818	java
3	powermock	1084	893	java
4	restfb	783	1188	java
5	rxjava	2673	2468	java
6	metro-jax-ws	1103	2222	java
7	mockito	1409	1572	java
8	grizzly	1592	3122	java
9	shipkit	242	1483	java
10	OpenClinica	1653	3749	java
11	robolectric	2050	5029	java
12	aeron	541	5101	java
13	antlr4	1381	3449	java
14	mcidasv	805	3668	java
15	ShareX	919	2505	csharp
16	aspnetboilerplate	2353	1615	csharp
17	orleans	3485	3353	csharp
18	cli	767	2397	csharp
19	cake	2250	1853	csharp
20	Avalonia	1677	2445	csharp
21	EntityFramework	7107	2443	csharp
22	jellyfin	2179	4065	csharp
23	PowerShell	861	2033	csharp
24	WeiXinMPSDK	2029	2723	csharp
25	ArchiSteamFarm	117	2181	csharp
26	VisualStudio	1016	4417	csharp
27	CppSharp	259	3882	csharp

3.2 Filtering based on the size of commit transactions

A big commit transaction can indicate that a merge with another branch or that a renaming has been made. In this case, a series of irrelevant logical dependencies can be introduced since not all the files are updated in the same time for a development reason. Different works have chosen fixed threshold values for the maximum number of files accepted in a commit. Cappiluppi and Ajienka, in their works [4], [5] only take into consideration commits with less then 10 source code files changed in building the logical dependencies.

The research of Beck et al [9] only takes in consideration transactions with up to 25 files. The research [44] 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 cpp and java systems. The results are presented in Figure 3-2, 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 logical dependencies [58].

As we can see in Figure 3-3 even though only 5% of the commit transactions have more than 20 files changed ($20 < cs < inf$) they generate in average 80% of the total amount of logical dependencies extracted from the systems. The high number

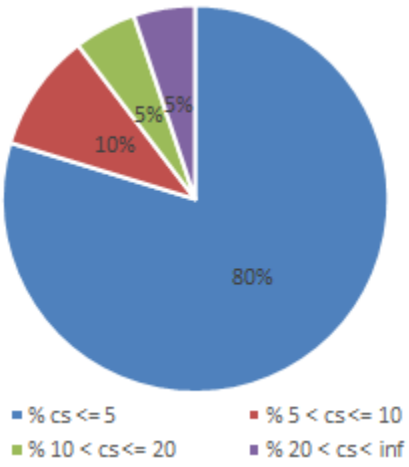


Figure 3-2: Commit transaction size(cs) trend in percentages

of logical dependencies extracted from such a small number of commit transactions is caused by big commit transactions. One single big commit transaction can lead to a large amount of logical dependencies. For example in RxJava we have a very few commit transactions with 1030 source code files, this means that those files 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 logical dependencies into the system.

So filtering 10% of the total amount of commit transactions can indeed lead to a significant decrease of the amount of logical dependencies and that is why we choose the value of 10 files as our fixed threshold for the maximum size of a commit transaction [58].

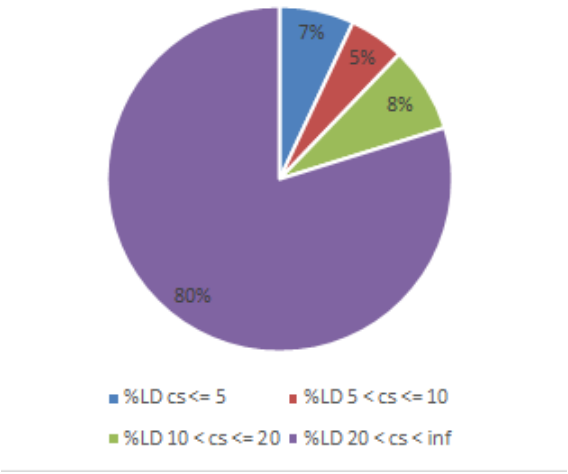


Figure 3-3: Percentages of LD extracted from each commit transaction size(cs) group

3.3 Filtering based on the number of occurrences

One occurrence of a co-change between two software entities can be a valid logical dependency, but can also be a coincidence. Taking into consideration only co-changes with multiple occurrences as valid dependencies can lead to more accurate logical dependencies and more accurate results. On the other hand, if the project studied has a relatively small amount of commits, the probability to find multiple updates of the same classes in the same time can be small, so filtering after the number of occurrences can lead to filtering all the logical dependencies extracted. Giving the fact that we will study multiple projects of different sizes and number of commits, we will analyze also the impact of this filtering on different projects.

We have performed a series of analysis on the test systems, incrementing the threshold value occ from 1 to 4. In each of the cases the extracted logical dependencies from commit transaction with less or equal to 10 changed source code files were also filtered by the minimum number of occurrences established and all the log-

ical dependencies that did not exceeded the minimum number of occurrences were discarded.

The results of the analysis are presented in Table 3.2 as percentages of logical dependencies (LD) that are also structural dependencies and Table 3.3 as ratio of the number of logical dependencies (LD) to the number of structural dependencies (SD).

Table 3.2: Percentage of LD that are also SD

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

Table 3.3: Ratio of number of LD to number of SD

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

Based on Table 3.2 we can say that only a small percentage of the extracted logical dependencies are also structural dependencies. This is consistent with the findings of related works [4], [5]. The percentage of LD which are also SD increases with the minimum number of occurrences because the number of logical dependencies from the systems decreases with the minimum number of occurrences. We calculate the overlapping between logical and structural dependencies not only because we want to get an idea of how many structural dependencies are reflected in the versioning system through logical dependencies but also because we want to eliminate logical dependencies that are also structural dependencies since they don't bring any new information to the systems.

We stopped the minimum occurrences threshold to 4 because we observed that for systems with ID 2, 6, 10 and 16 from Table 3.3 the ratio number is lower than 1 which means that the number of SD is higher than the number of LD. On the other hand for systems with ID 4, 11, 25, 27 the threshold of 4 for minimum number of occurrences does not change the discrepancy between the number of logical and structural dependencies. If we try to go higher with the occurrences threshold we will risk to filter all the existing logical dependencies for some of the 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 logical dependencies will still be significantly higher compared to the number of structural dependencies.

3.4 Overlaps between structural and logical dependencies

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 [4]. 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 [57].

In the main series of experiments, for each system, we extracted the structural dependencies and the logical dependencies 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 towards a logical dependency, even if the change is inside comments in all files
- without comments: commits that changed source code files only by editing comments are ignored as logical dependencies

In all cases, we varied the following threshold values:

- commit size (cs): the maximum size of commit transactions which are accepted to generate logical dependencies. 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 logical dependencies (LD) to the number of structural dependencies (SD)
- the mean percentage of structural dependencies that are also logical dependencies (calculated from the number of overlaps divided to the number of structural dependencies)
- the mean percentage of logical dependencies that are also structural dependencies (calculated from the number of overlaps divided to the number of logical dependencies)

In all the six tables, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9 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.

In order to assess the influence of comments, we compare pairwise Tables 3.4 and 3.5, Tables 3.6 and 3.7 and Tables 3.8 and 3.9. 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.

Table 3.4: Ratio of number of LD 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 3.5: Ratio of number of LD 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 3.6: Percentage of SD that are also LD, 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 3.7: Percentage of SD that are also LD, 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 3.8: Percentage of LD 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

Table 3.9: Percentage of LD 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

On the other hand, the overlap between structural and logical dependencies is given by the number of pairs of classes that have both structural and logical dependencies. We evaluate this overlap as a percentage relative to the number of structural dependencies in Tables 3.6 and 3.7, respectively as a percentage relative to the number of logical dependencies in Tables 3.8 and 3.9.

A first observation from Tables 3.6 and 3.7 is that not all pairs of classes with structural dependencies co-change. The biggest value for the percentage of structural dependencies that are also logical dependencies is 76.5% obtained in the case when no filterings are done.

From Tables 3.8 and 3.9 we notice that the percentage of logical dependencies 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 [57].

3.5 Conclusions and future work

Our experiments show that the most important factors which affect the quality of logical dependencies are: the maximum size of commit transactions which are accepted to generate logical dependencies, and the minimum number of repeated occurrences for a co-change to be counted as logical dependency.

We conclude that it is important to put a threshold on the maximum size of com-

mit transactions which are accepted to generate logical dependencies. Only small commit transactions (changing up to 5 source code files) can be reliably used for introducing logical dependencies. We have also determined that small commit transactions are the most frequent kind of transactions, representing in average 80% of all commit transactions. Under these conditions, we have determined that increasing the threshold for the minimum number of repeated occurrences for a co-change to be counted as a logical dependency reduces significantly the number of logical dependencies. In average, increasing with 1 the threshold for repeated occurrences determines a reduction to half for the number of logical dependencies. A value of 4 for the threshold for repeated occurrences, combined with the condition of accepting only small commit transactions, already keeps the number of logical dependencies in the same range as the number of structural dependencies. Future work will investigate further the issue of repeated occurrences, analyzing also their trend in time.

Currently we have extracted logical dependencies from all the revisions of the system, and structural dependencies from the last revision of the system. In future work we will take into account also structural dependencies from all the revisions of the system, in order to filter out the old, out-of-date logical dependencies. 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 occurrences 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.

The future work will be performed in the next stages of the doctoral program as presented in Chapter 4.

Chapter 4

Research content and stages of research

4.1 Proposed research stages

The research will be made by following the next stages of implementation:

Section 1: Development of content and tools

Stage 1: Build tool to extract structural dependencies from code and co-changes from git for a given set of projects.

Stage 2: Find filters for the co-changes extracted, the filters can be the ones already mentioned in previous works or new ones. Establish different thresholds for those filters.

Stage 3: Study the impact of those filters and the corresponding thresholds on the remaining quantity of co-changes for each system. Study the overlapping between the remaining pairs of co-changing entities and the structural dependencies extracted

[57].

Stage 4: Establish a dynamic way to determine the thresholds for filters in order to fit the best each studied system. Main focus on the threshold for number of occurrences of co-changing pairs. Use plots or other visual instruments in order to see the highest and the lowest rates for the numbers of occurrences among co-changing pairs. Also filter those rates into normal and abnormal ones and study what was the cause of the highest rates (code or human related).

Stage 5: Take into account also structural dependencies from all the revisions of the system to filter out the old, out-of-date logical dependencies. Study how this affects the remaining number of logical dependencies.Here an extra check is needed, it can be a case in which old structural dependencies that were also logically linked to continue to be logically linked even after the structural dependency was removed.

Section 2: Usage

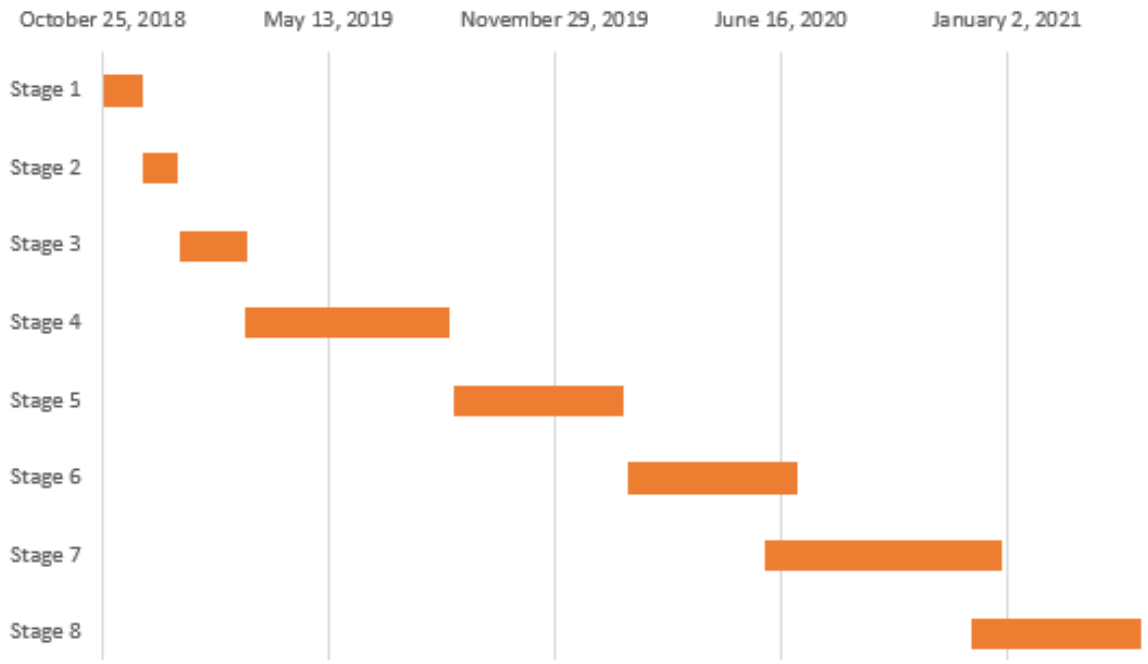
Stage 6: Export the remaining co-changes whom at this step we can call logical dependencies and use them among structural dependencies in tools for architectural reconstruction to evaluate the improvement.

Stage 7: Compare the number of logical dependencies with metrics like Fan Out, Fan In, Efferent Coupling (Ce), Afferent Coupling (Ca) and study their connections.

Stage 8: Identify other tools that use structural dependencies and evaluate the impact of co-changes filtering into logical dependencies for them.

4.2 Gantt chart

Gantt Chart				
START DATE	END DATE	DESCRIPTION	PAPER OUTPUT	DURATION (days)
10/25/18	11/30/18	Stage 1	No	35
11/30/18	12/30/18	Stage 2	No	30
1/1/19	3/1/19	Stage 3	Yes: ENASE, SACI	60
3/1/19	9/1/19	Stage 4	No	180
9/1/19	2/1/20	Stage 5	Yes: ICSME	150
2/1/20	7/1/20	Stage 6	Yes: ENASE	150
6/1/20	1/1/21	Stage 7	Yes: ICSE	210
12/1/20	5/1/21	Stage 8	No	150



4.3 Proposed contents of the thesis

Chapter 1. Introduction - This chapter will contain the motivation of the research topic and the motivation for choosing it.

Chapter 2. Dependencies in software systems - *State of the art* - This chapter will present existing types of dependencies in software systems and their nature.

Chapter 3. Filtering co-changes into logical dependencies - *Original contribution* - This chapter will present methods and experiments in filtering co-changes in order to obtain logical dependencies.

Chapter 4. Logical dependencies in practice - *Validation* - This chapter will present a series of applications of logical dependencies extracted from software systems.

Chapter 5. Conclusions - This chapter will provide the final conclusions and will summarize the contributions realized through this paper.

Bibliography

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

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

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

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

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

[13] 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.

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

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

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

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

[29] 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.

[30] 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.

[31] 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.

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

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

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

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

[49] 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.

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

[51] 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.

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

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

[54] 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.

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

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

[57] 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.

[58] 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.

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

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

[61] 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.

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

[63] 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.

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

[65] 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.