Methods and Tools for the Analysis of Legacy Software Systems

Department of Computers and Information Technology
2021

Ph.D. student: Stana Adelina Diana

Scientific supervisor: prof.dr.ing Cretu Vladimir-Ioan

Contents

Chapter 1

Introduction

This report presents the results obtained so far on the proposed thesis. The goal of the thesis is to develop methods for analyzing legacy software systems, focusing on using historical information describing the evolution of the systems extracted from the versioning systems.

We have developed a tool that extracts and processes the needed information from a software system. The tool workflow and technologies used are presented in section 2.1. The primary information extracted by the tool is described in sections 2.2 and 2.3. The filtering methods selected to be applied to the extracted information are presented in sections 3.2, 3.3, and 3.4.

To perform measurements based on our assumptions, we have selected a set of 27 object-oriented software systems presented in section 3.1. For each listed software system, the tool extracts, filters, and collects the information needed.

Each filtering section (3.2, 3.3, and 3.4) contains the detailed results obtained after analyzing all the software systems and conclusions based on the results.

Section 3.5 focuses on the overlappings between the extracted information from the code and filtered information from the versioning systems.

The conclusions and observations based on the performed measurements are presented in chapter 4.

Chapter 2

Extracting software dependencies

2.1 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 [20]. 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 2-1 represents the activities mentioned above. Each block represents a different activity.

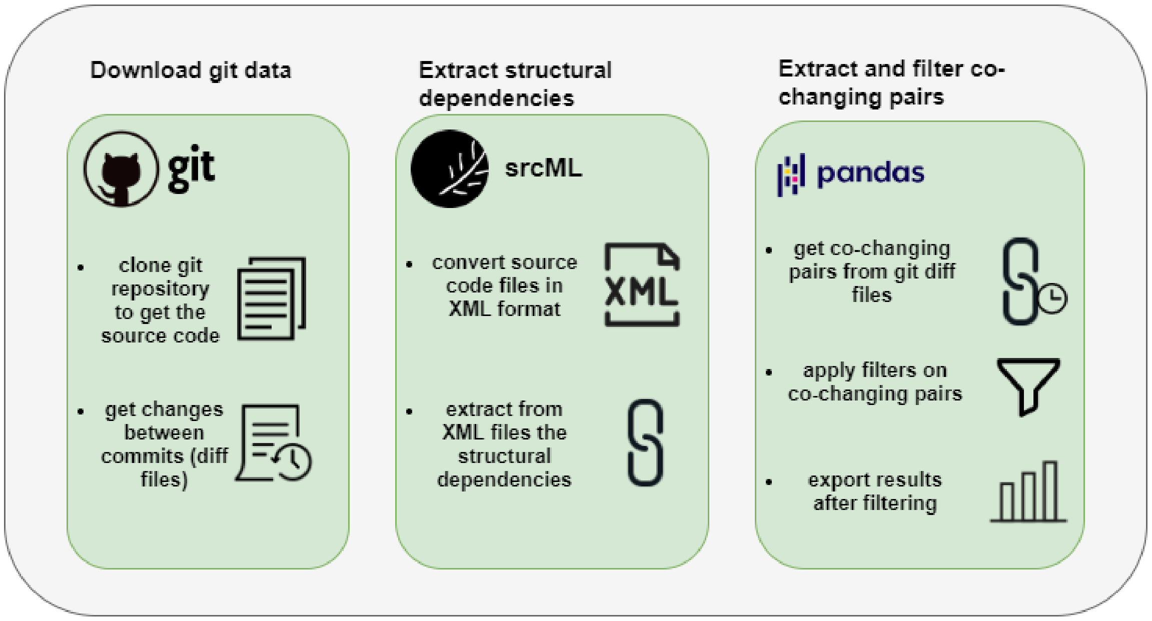
Download git data
git
clone git repository to get the source code
get changes between commits (diff files)
Extract structural dependencies
srcML
convert source code files in XML format
XML
extract from XML files the structural dependencies
Extract and filter co-changing pairs
pandas
get co-changing pairs from git diff files
apply filters on co-changing pairs
export results after filtering

Figure 2-1: 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 2.2 and the change history for co-changing pairs extraction 2.3. 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 2-2, 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.

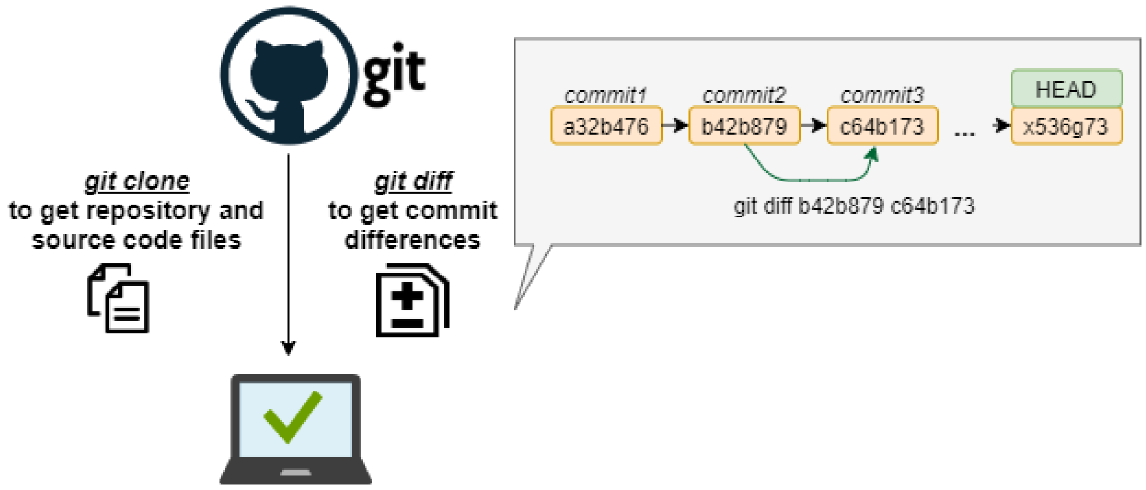

Figure 2-2: Commands used to download the required data from GitHub.

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 [1]. 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 entities mentioned above constitutes a structural dependency.

Extract and filter co-changing pairs. As presented in sections 3.2, 3.3, and

3.4, 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.

2.2 Extracting 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 [7], [10].

Structural dependencies can be found by analyzing the source code [18], [8], [6]. 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 [11], [12] 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#.

2.3 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).

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 [21], [9]. Those changes can be found in the software history maintained by the versioning system. Gall [14], [15] identified as logical coupling between two modules the fact that these modules *repeatedly* change together during the historical evolution of the software system [4].

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 [13]. 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 [2].

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.

Chapter 3

Filtering extracted co-changing pairs

3.1 Data set used

We have analyzed a set of open-source projects found on GitHub[1] [16] 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) 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.

[1]http://github.com/

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 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 [2], [3] only take into consideration commits with less then 10 source code files changed in building the logical dependencies.

The research of Beck et al [5] only takes in consideration transactions with up to 25 files. The research [17] 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 3-1, 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 [20].

As we can see in Figure 3-2 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 of logical dependencies extracted from such a small number of commit transactions

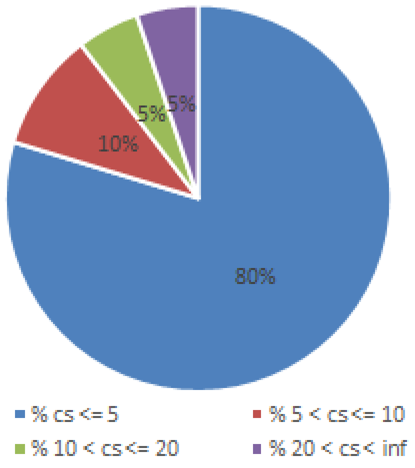

Figure 3-1: Commit transaction size(cs) trend in percentages

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 [20].

3.3 Filtering based on 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

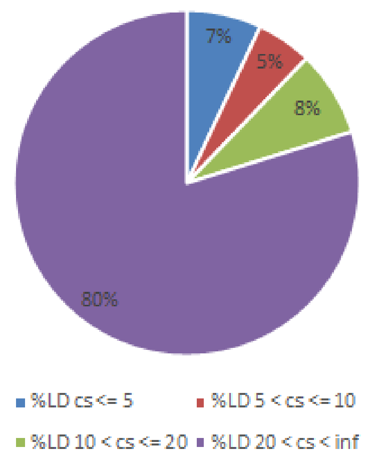

Figure 3-2: Percentages of LD extracted from each commit transaction size(cs) group

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 logical 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 [2], [3]. 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 Filtering based on connection strenght

3.5 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 [2]. 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 [19].

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

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

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 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 [19].

Chapter 4

Conclusions

Bibliography

[1] srcml; www.srcml.org.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[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] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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