Analyzing information from versioning systems to detect logical dependencies in software systems

Adelina Diana Stana
Department of Computer and Information Technology
Politehnica University of Timisoara
Timisoara, Romania

Ioana Şora
Department of Computer and Information Technology
Politehnica University of Timisoara
Timisoara, Romania

***Abstract*—Emerging software engineering approaches support the idea that general methods and tools for dependency management should take into account not only structural dependencies but also logical dependencies. In this work we try to identify a set of factors that can be used to filter the co-changes such that true logical dependencies are identified. We present preliminary results obtained through an experimental study on a set of open source software projects with their historical evolution, and outline additional factors which need to be investigated in future work.**

***Index Terms*—software evolution, logical dependencies, structural dependencies**

I. Introduction

Methods and techniques for the analysis of software systems operate on models of the software system. A type of frequently used models are dependency models which make dependencies due to relationships between program parts or system components explicit [1], [2].

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 [3], [4]. In the case of object oriented software systems, dependency models are usually class dependency models where elements are entities such as classes and interfaces [2]. There are several types of relationships between these source code entities, for example a method of a class can call a method of another class, a class extends another class, all those create *structural dependencies* between classes (a.k.a syntactic dependencies or structural coupling). These dependencies can be found by the analysis of the source code.

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 [5]. Those changes can be found in the software history maintained by the versioning system. Gall [6] identified as logical coupling between two modules the fact that these modules *repeatedly* change together during the historical evolution of the software system. *Logical dependencies* (a.k.a logical coupling) ca be found by software history analysis and can reveal relationships that are not always present in the source code (structural dependencies).

Figure 1 presents an example of relationships between structural and logical dependencies. As we can see in the figure, from the source code of the system we extract structural dependencies between classes A - B and A - C. On the other hand, in the versioning system classes A - C and B - C change together which means that those pairs can be logical dependencies. It is expected for a structural dependency to also be a logical dependency since changes in one element of the dependency could trigger also changes in the other one and this makes them change together. Classes B and C also change together even thought they have no structural dependency. This could mean that classes B and C have a logical dependency which leads to their concomitant change.

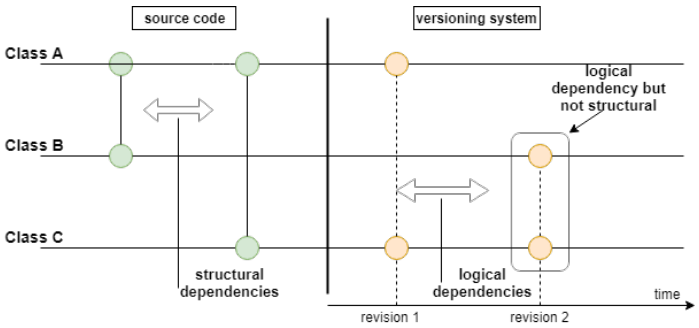

Fig. 1. Relationships between structural and logical dependencies

Changes made to two modules in the same commit do not necessarily indicate the co-evolution of the two. These changes could be completely unrelated. The study [7] acknowledges the fact that evolutionary coupling could also be determined accidentally by two modules changing in the same commit (independent evolution, as it is called) and this will bring noise to the measurement of logical coupling.

In our work we investigate which are the patterns of repeated co-change that produce true logical dependencies. This paper is organized as follows: Section II discusses the state of the art regarding the detection and usage of logical dependencies. In Section III we describe our tool used to analyze software systems in order to extract structural and logical dependencies. Section IV presents our experimental results on a set of open source systems. Future work is outlined in Section V, while Section VI summarizes our concluding remarks.

II. State of the art

The current software engineering trends recommend that general dependency management methods and tools should also include logical dependencies besides the structural dependencies that are extracted by analyzing the source code [8], [9], [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 [10], for identifying the potential ripple effects caused by software changes during software maintenance and evolution [11], [8], [12], [13] or for their link to deffects [14], [15]. But, in order to add logical dependencies besides structural dependencies as inputs for methods and tools for dependency management and analysis, co-changes must be filtered until they remain only a reduced but relevant set of true logical dependencies.

There are researches that investigated quantitative aspects of logical dependencies and their interplay with structural dependencies. Oliva and Gerosa [8], [11] 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.

Zimmermann et al [15] 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.

In order to add logical dependencies besides structural dependencies as inputs for methods and tools for dependency management and analysis, class co-changes must be filtered until they remain only a reduced but relevant set of true logical dependencies.

III. 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. The workflow can be delimited by three major steps as it follows (Figure 2):

Step 1: *Extracting structural dependencies.*
Step 2: *Extracting logical dependencies.*
Step 3: *Processing the information extracted.*

A. 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 [16], [17] 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 easyer the parsing of source code written in different programming languages such as Java, C++ and C#.

B. 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 [18]. 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 [9].

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.

IV. Filtering mechanisms for logical dependencies and why we need them

We have analyzed a set of open-source projects found on GitHub[1] [19] in order to extract the structural and logical dependencies between classes. Table I 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/

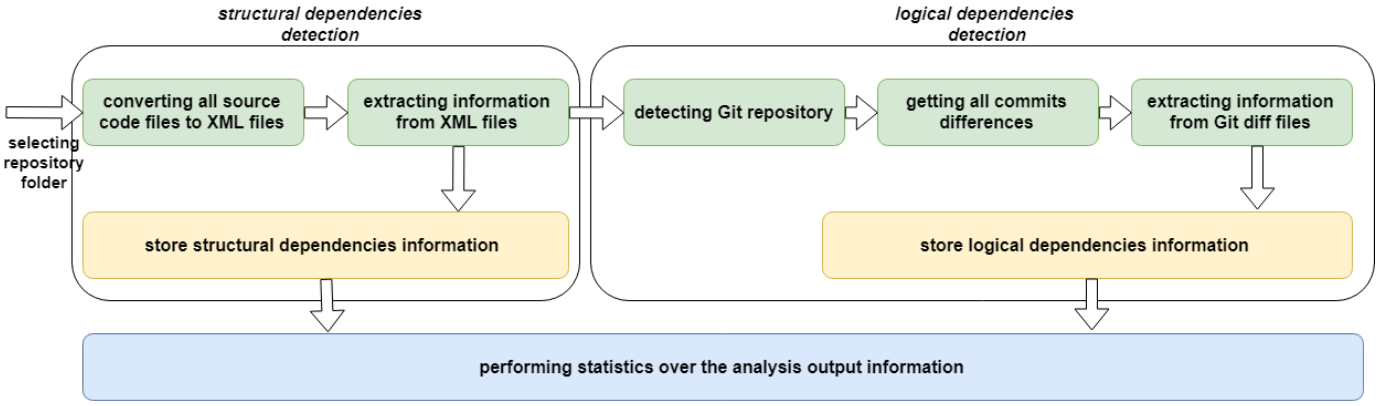

Fig. 2. Processing phases

TABLE I
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

A. Estimating the number of co-changing classes vs the number of structural dependencies

In a preliminary experiment, we consider every pair of classes that change together in any transaction as a logical dependency. By extracting all the logical dependencies(LD) of systems from table I and integrating them as they are with the structural dependencies(SD) we will obtain a huge amount of dependencies. To get an overview of how huge is that amount, we summed the extracted logical and structural dependencies from each system then calculated the percentages of each type of dependency from that amount. In average, structural dependencies represent only 0,23% of the total amount of dependencies, the rest of 99,77% are logical dependencies.

It is clear that we cannot use the extracted logical dependencies(LD) as they are since they number is to high and we have to find filtering mechanisms in order to reduce their percentage until it is smaller than the percentage of structural dependencies(SD). In the following subsections we will define two filtering mechanisms for logical dependencies.

B. Filtering based on commit size transaction

We define the size of a commit transaction as the total number of source code files that have changed. Big commit transactions can be due to merges with other branches or folders renamings. 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 ignore logical dependencies produced by big commit transactions, by choosing fixed threshold values for the maximum number of files accepted in a commit transaction: [9], [20], [21].

Before we decided to also put a fixed threshold over the size of commit transactions we analyzed the overall transaction size trend of our studied the systems. In order to do this we have chosen the following threshold values for the number of files in a commit transaction: 5, 10, 20 and no threshold (infinity) and we counted the number of commit transactions that fall into each category. Based on the results presented in figure 3 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.

So, how will this threshold will help in filtering the huge amount of extracted logical dependencies from the systems? As we can see in figure 4 even though only 5% of the commit transactions have more than 20 files changed ($ts < inf$) 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 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 size transaction 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 as our fixed threshold for the maximum size of a commit transaction.

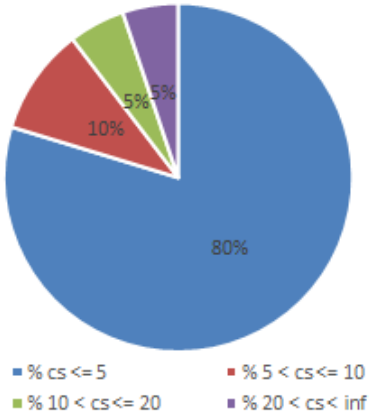

Fig. 3. Transaction size(ts) trend in percentages

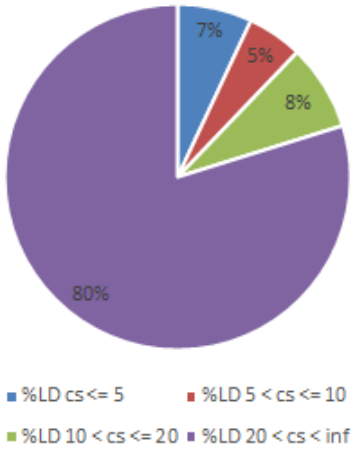

Fig. 4. Percentages of LD extracted from each transaction size(ts) group

C. Filtering based on the number of occurrences

Filtering only based on the commit size transaction is not enought to filter logical dependencies since the remaining amount still exceeds the amount of structural dependencies. One occurrence of a logical dependency between two entities cannot be a valid logical dependency since that can be a coincidence. If two entities have multiple occurrences together then the probability of a coincidence is smaller. Based on this assumption we have performed a series of analysis on the systems with a incremented number of occurrences 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 were presented in table II as percentages of logical dependencies reported to the total number of dependencies in the system. The total number of dependencies from the system is the sum of logical and structural dependencies.

We stopped the minimum occurrences number to 4 because we observed that for systems with ID 2, 6 and 10 the percent of LD is lower than 50% which means that the number of SD is higher than the number of LD. On the other hand for systems with ID 19, 24, 27 the threshold of 4 for minimum number of occurrences has not changed almost at all 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.

V. Future work

As we can see in section IV-C filtering logical dependencies based on occurrences with a fixed value is not the best solution. The threshold for occurrences may also be related to the number of commits of the system. Logical dependencies from systems with 5000 commits will have more chances to appear together than logical dependencies from systems with 1000 commits. Also the number of entities from the systems can influence the number of occurrences. A system with a small number of entities and a big number of commits may have bigger numbers of occurrences since there are not so many combinations of entities that can appear together. We will try to establish all the factors that influence the number of occurrences and dynamically calculate the threshold value for the minimum number of occurrences for each system.

Also, we consider that in the future, the validation of extracted logical dependencies will occur by using them to enhance dependency graphs for applications such as architectural reconstruction [22], [23] through clustering [24] or finding of key classes [25], and evaluating the positive impact on their results.

VI. Conclusion

In this work we experimentally define methods to filter out the most relevant logical dependencies from co-changing classes.

Our experiments showed that the most important factors studied until now, which affect the quality of logical dependencies are: the maximum number of files allowed in a commit to

TABLE II
PERCENTAGE OF LD RAPORTED TO THE TOTAL NUMBER OF DEPENDENCIES IN THE SISTEM

ID	$occ \geq 1$	$occ \geq 2$	$occ \geq 3$	$occ \geq 4$
1	94,22	88,72	83,58	55,57
2	66,95	47,03	32,05	24,20
3	94,51	87,27	74,77	60,58
4	99,19	98,97	98,87	98,81
5	92,87	89,91	74,04	70,06
6	77,26	61,79	52,07	45,05
7	92,72	88,08	76,75	70,78
8	90,68	79,70	66,59	57,20
9	92,75	88,41	82,38	77,22
10	75,20	68,56	51,74	44,34
11	96,26	93,68	90,81	87,50
12	89,61	83,07	77,56	71,92
13	97,10	91,51	81,16	78,47
14	88,64	80,35	75,73	72,30
15	89,70	78,35	69,91	57,46
16	80,51	63,74	49,90	41,58
17	84,82	75,70	61,57	53,13
18	88,06	81,87	70,79	61,36
19	93,00	89,04	81,50	79,27
20	80,93	68,30	58,95	46,89
21	93,84	84,06	75,83	70,34
22	88,68	81,67	70,49	64,44
23	90,27	77,51	68,61	55,92
24	90,80	83,13	74,19	66,94
25	97,71	96,90	96,18	95,77
26	91,75	85,81	81,20	77,17
27	93,34	88,93	84,73	80,63
M	89,31	81,55	72,66	65,36

be counted as logical dependency, and the minimum number of repeated occurrences for a co-change to be counted as logical dependency.

REFERENCES

[1] T. B. Callo Arias, P. van der Spek, and P. Avgeriou, "A practice-driven systematic review of dependency analysis solutions," *Empirical Software Engineering*, vol. 16, no. 5, pp. 544–586, Oct 2011. [Online]. Available: https://doi.org/10.1007/s10664-011-9158-8

[2] N. Sangal, E. Jordan, V. Sinha, and D. 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*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 167–176. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094824

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

[4] 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*, vol. 35, pp. 864–878, 2009.

[5] L. Yu, "Understanding component co-evolution with a study on linux," *Empirical Softw. Engg.*, vol. 12, no. 2, pp. 123–141, Apr. 2007. [Online]. Available: http://dx.doi.org/10.1007/s10664-006-9000-x

[6] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 190–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850947.853338

[7] L. Yu, "Understanding component co-evolution with a study on linux," *Empirical Software Engineering*, vol. 12, no. 2, pp. 123–141, Apr 2007. [Online]. Available: https://doi.org/10.1007/s10664-006-9000-x

[8] G. A. Oliva and M. A. Gerosa, "On the interplay between structural and logical dependencies in open-source software," in *Proceedings of the 2011 25th Brazilian Symposium on Software Engineering*, ser. SBES '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 144–153. [Online]. Available: https://doi.org/10.1109/SBES.2011.39

[9] N. Ajienka and A. Capiluppi, "Understanding the interplay between the logical and structural coupling of software classes," *Journal of Systems and Software*, vol. 134, pp. 120–137, 2017. [Online]. Available: https://doi.org/10.1016/j.jss.2017.08.042

[10] X. 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.*, May 2005, pp. 664–665.

[11] G. A. Oliva and M. A. 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*, 2015, pp. 250–260. [Online]. Available: https://doi.org/10.1109/ISSRE.2015.7381818

[12] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical Software Engineering*, vol. 14, no. 1, pp. 5–32, Feb 2009. [Online]. Available: https://doi.org/10.1007/s10664-008-9088-2

[13] 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*, Oct 2010, pp. 119–128.

[14] I. S. Wiese, R. T. Kuroda, R. Re, G. A. Oliva, and M. A. Gerosa, "An empirical study of the relation between strong change coupling and defects using history and social metrics in the apache aries project," in *Open Source Systems: Adoption and Impact*, E. Damiani, F. Frati, D. Riehle, and A. I. Wasserman, Eds. Cham: Springer International Publishing, 2015, pp. 3–12.

[15] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 563–572. [Online]. Available: http://dl.acm.org/citation.cfm?id=998675.999460

[16] 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*, ser. IWPC '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 134–. [Online]. Available: http://dl.acm.org/citation.cfm?id=851042.857028

[17] M. L. Collard, M. J. Decker, and J. 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*, ser. SCAM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 173–184. [Online]. Available: https://doi.org/10.1109/SCAM.2011.19

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

[19] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining github," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, Oct 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9393-5

[20] N. Ajienka, A. Capiluppi, and S. Counsell, "An empirical study on the interplay between semantic coupling and co-change of software classes," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1791–1825, 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9569-2

[21] F. Beck and S. 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*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 354–364. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025162

[22] M. Shtern and V. Tzerpos, "Clustering methodologies for software engineering," *Adv. Soft. Eng.*, vol. 2012, pp. 1:1–1:1, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1155/2012/792024

[23] S. Ducasse and D. Pollet, "Software architecture reconstruction: A

process-oriented taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, July 2009.

[24] I. Şora, G. Glodean, and M. 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*, May 2010, pp. 259–264.

[25] I. 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*. Springer International Publishing, 2015, pp. 122–140.