Computer Science and Engineering Department
"Politehnica" University of Timisoara

An analysis of the relationship between structural and logical dependencies in software systems

Adelina Diana Stana

Coordinated by
Assoc Prof. Ioana Sora

May 25, 2018

Contents

1 Introduction

1.1 State of the art

Software systems are continuously in change. As long as the software system is used the changing process is never nished. Changes can be triggered by new features , defects, new technologies, system refactoring for maintainability.
Software maintenance can be made during the development process by maintaining the already implemented features, while developing new ones or at the endofthedevelopmentprocesswhenthesystemisno longer open to new features requested by the client and the maintenance is only made for the existing ones. From an architectural point of view,a system is stable when a change in one component of the system does not aect the other components .This rule applies recursively also inside the components.

The ideal situation presumes that changes in one part can be made without changing parts that are in a dependency relation with that part. Those dependencies aect the maintainability of the system and increase the realization eort of any problem that appears during the maintenance time. Studying only the structural dependencies of the system is not enough to get a clear overview of the system dependencies. For more precise results is needed a study that combines structural dependencies and logical dependencies.
We have analysed 19 open-source software systems of dierent sizes to investigate the links between structural dependencies and logical dependencies.

During the development process of a software system new classes and new methods to the existing classes are added in order to fulll new functionalities. All of the adding actions from above have a direct impact on the system structural dependencies. Those are the result of the source code analysis of the system.
The source code is any static, textual, human readable, fully executable description of a computer program that can be compiled automatically into an executable form [1].
On the other hand logical dependencies also can be added during the development process. Logical dependencies refer to those depencencies between entities that are not always visible through source code analysis (Figure 1.1).

Logical dependencies can be easily extracted from the versioning system (e.g. Subversion , Git) revisions.

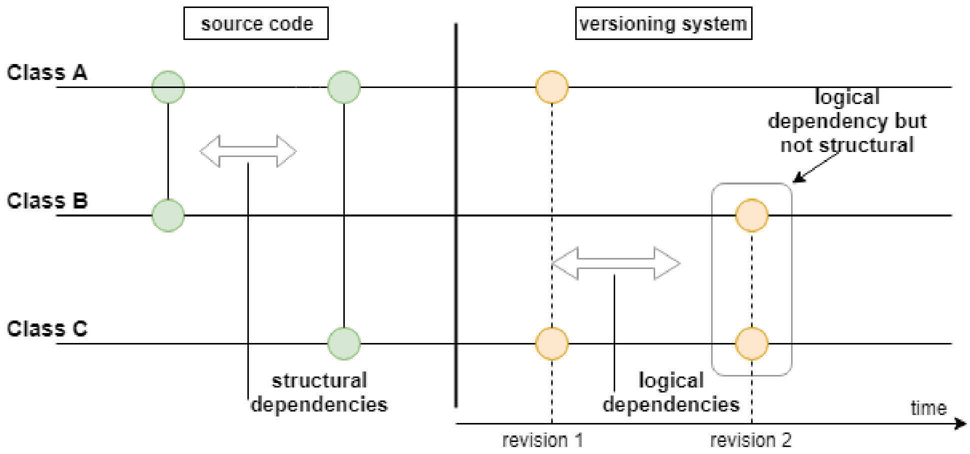

Figure 1.1: Relationships between structural and logical dependencies

1.2 Purpose of the project

In this paper we intend to understand better the intersection between structural dependencies and logical dependencies and their impact over the system (Figure 1.2). To be done ...

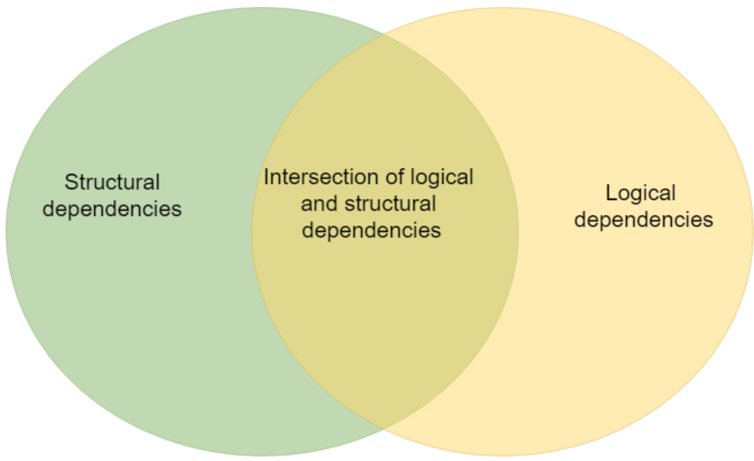

Figure 1.2: Venn Diagram showing the intersection between structural and logical dependencies

2 | Theoretical aspects

2.1 Software 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. In this case, if one of these elements change, there could be an impact to the other [4]. Dependencies are discovered by analysis of source code or from an intermediate representation such as abstract syntax trees [5] . 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 workow can be delimited by three major steps as it follows (Figure 4.1):
Step 1: *Extracting structural dependencies.*
Step 2: *Extracting logical dependencies.*
Step 3: *Processing the information extracted.*

2.2 Structural dependencies

Structural dependencies (a.k.a syntactic dependencies or structural coupling) are the result of source code analysis. Each source code le can contain one or more classes. There are several types of relationships between source code entities, a method can call another class method, a class extends another class, all those create dependencies between classes [6]].

Even though in some of the cases if class A depends on class B , changes in class B can produce changes in class A, but not the other way around [7] . There are other cases in which if class A depends on class B,changes in class B can produce changes in class A and viceversa if we are speaking in the context of a new feature implementation that implies changing return types and adding new methods. So we will consider structural dependencies as bidirectional relationships, "classA depends on class B" and "class B depends on class A".
The choice of building bidirectional relationships is motivated by the fact that we cannot establish for the moment the direction of the logical dependencies of the system. So in order to have a omogeninty between the logical and structural dependencies analysis results, we will take both of the relationships types as bidirectional.

2.3 Logical dependencies

2.3.1 Version control systems

In computer software engineering, revision control is any kind of practice that tracks and provides control over changes to source code. Software developers sometimes use revision control software to maintain documentation and source code.
At a basic level, developers could retain multiple copies of the different versions of the program and label them appropriately. This method can work but it is inefficient as many copies of the program have to be maintained. Because of this, systems to automate some or all of the revision control process have been developed.

Among the keywords used in a versioning system we can mention:
Repository - is a virtual storage of a project. It allows to save versions of the code, which can be access when needed.
Master/Trunk - the main body of development, originating from the start of the project until the present.
Branch - a copy of code derived from a certain point in the master that is used for applying major changes to the code while preserving the integrity of the code in the master. The changes are usually merged back into the master.
Revision - changes are usually identified by a number or letter code, that code is known as "revision".

2.3.2 Definition

The versioning system contains the long-term change history of every file. Each project change made by an individual at certain point of time is contained into a commit [8].
All the commits are stored in the versioning system cronologicaly and each commit has a parent. The parent commit is the baseline from which development began, the only exeption 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 reduntant 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.
Finally after all the differences files are stored , all the files are parsed and logical dependencies are build. In addition , the number of files changed in a commit can influence the logical dependencies. A relatively big number of files changed can indicate a merge of all changes from another branch as a single commit. This can lead to a number of logical dependencies that are redundant since the files are not actualy changing in the same time.The logical dependencies are splitted into three categories :
Category 1: *Dependencies found in commits with less than 5 source code files*

changed.
Category 2: *Dependencies found in commits with more than 5 files changed but less than 20.*
Category 3: *Dependencies found in commits with more than 20 files.*

Also for each category two dependencies analysis will be made: ***A:*** *Considering comments as valid changes.* ***B:*** *Considering comments as redundant changes.* In the second case if class A and class B change together but the only change found is a comment change then between class A and B will not be set logical dependency.

3 Tool usage

Figure 3.1

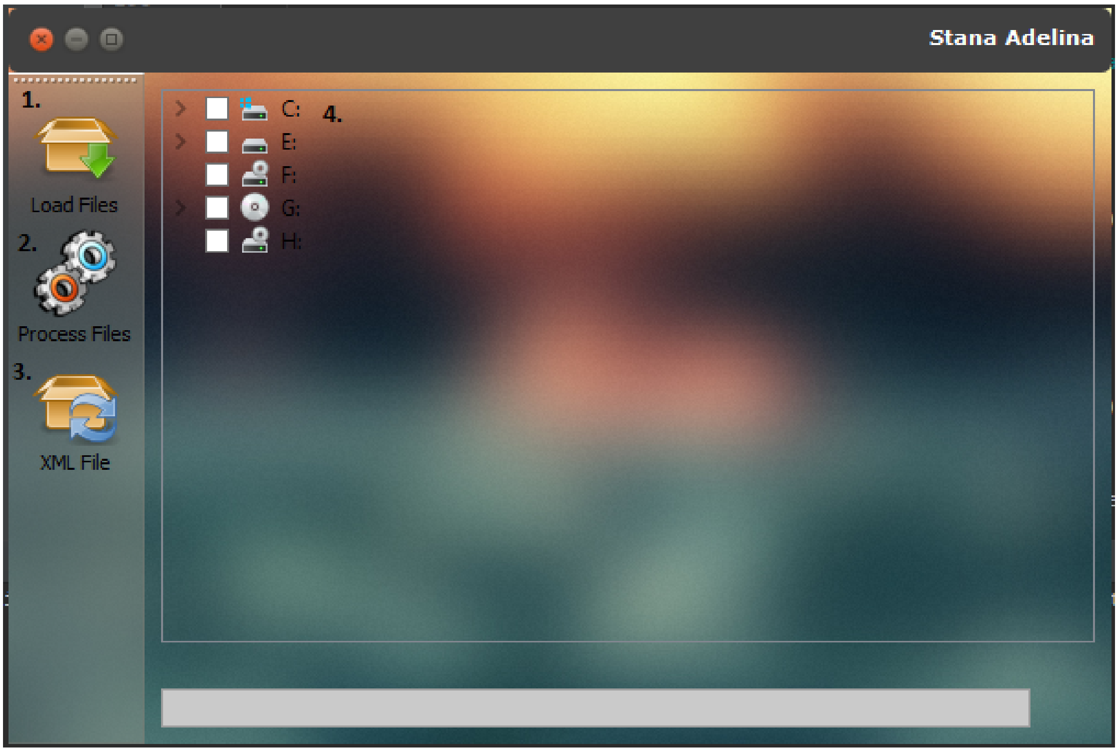

Figure 3.1: Processing phases

4 | Implementation details

4.1 Architecture

here goes the uml

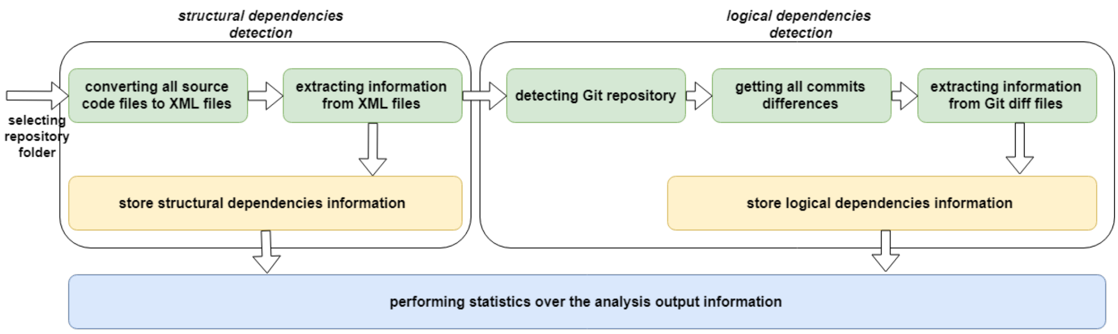

Figure 4.1: Processing phases

4.2 Extracting software dependencies

4.2.1 srcML

SrcML is an open source tool that converts in an XML format C/C#/Cpp/Java source code. [11]
All the original source code files are kept and the corresponding XML files are created. Each source code file is converted in a single XML document.
The srcML toolkit includes **source-to-srcML** and **srcML-to-source** translators:

- source-to-srcML : is responsable for source code to XML conversion.
- srcML-to-source : is responsable for XML to source code conversion, so that the original source code document can be recreated from the srcML XML file.

The srcML XML contains of all text from the source code file and XML tags. The file contains all the syntactic structures from the code (e.g., classes, structures, functions, function call, destructors, methods, if statements, for statements, switch statements, etc.).[12] An example of the XML representation can be found in .

4.2.2 XML parser

4.3 Extracting logical dependencies

tbd

The second step of the analysis is to extract logical dependencies from the versioning system. The versioning system contains the long-term change historyofeveryle.Changes can be made by many individuals over the years.
Changes include the creation and deletion of les as well as edits to their contents. Each change (this can include changes in multiple les) made by an individual at certain point of time is contained into a revision[8]. All the revisions are stored in the versioning system cronologicaly and each revision has a parent revision.
The parent revision is the revision from which development began, the only exeption to this rule is the rst revision which has no parent revision.We will take into consideration only revisions that have a parent revision since the rst revision can include source code les that are already in development (migration from one versioning system to another) and this can introduce reduntant logical links . We also will take into consideration only revisions that contain changes in at least one le with .java extension since our main focus is nding classes that change together that can only be contained in .java and .cpp les [2] .
The tool looks through the repository and gets all the existing revisions, for each revision a dierences le will be made . The dierences le will contain all the changes from all the les merged . Before making the dierences le, the revision shall fulll all the conditions mentioned above. In addition each le will be saved with the number of .java and .cpp les changed in his name. In this way it will be more easy to lter the revisions dierences les by the number of les changed. Finally after all the dierences les are stored , all the les are parsed and logical dependencies are build. The logical dependencies are splitted into three categories : dependencies found in revisions with less than 5 les changed, dependencies found in revisions with more than 5 les changed but less than 20 and dependencies found in revisions with more than 20 les changed. Also dependencies found with only comments change where not taken in to consideration (classA and classB change together but the only change found is in comments).

4.4 Saving and restoring the information processed

tbd

5 Experimental results

In this study, we have made a set of statistical analysis on a set of open-source projects in order to extract the structural and logical dependencies between classes [7], [2] . Table 5.1 illustrates all the systems studied. The 1st column shows the projects IDs; 2nd column shows the project name; 3rd column shows the number of classes extracted; 4th column shows the number of commits from the main branch of each project and the 5th shows the language in which the project was developed.

ID	Project	Nr. of classes	Nr. of commits	Type
1	urSQL	41	89	java
2	JavaCoder	5	11	java
3	jbandwidthlog	14	54	java
4	sjava-logging	18	62	java
5	daedalum	66	29	java
6	prettyfaces	236	207	java
7	jbal	102	113	java
8	guavatools	237	85	java
9	monome-pages	240	280	java
10	kryo	309	743	java
11	bitlyj	21	81	java
12	slema	276	368	java
13	bluecove	435	1679	java
14	gp-net-radius	25	28	java
15	aima-java	833	1181	java
16	powermock	966	1512	java
17	restfb	757	1545	java
18	Tensorflow	1104	2386	cpp
19	mangnum	143	1728	cpp

Table 5.1: Sumary of open source projects studied

Table 5.2, illustrates results for the categories mentioned in subsection 2.2. The 1st column shows the projects IDs; 2nd column shows the number of structural dependencies; 3rd column shows the number logical dependencies found with comments taken into consideration as change; 4th column shows the number of logical dependencies found in col. 3 that are also structural dependencies; 5th column shows logical dependencies found without taking into consideration comments as change; finally the 6th column shows the number of logical dependencies found in col.5 that are also structural dependencies.

Table 5.3 and 5.4 illustrates results in percentage, reported to the structural dependencies, of the analysis for all the systems when logical dependencies where build with/ without comments taken into consideration as change.The 1st column shows the projects IDs;2nd column shows the overlaping procentage between logical and structural dependencies for commits with less then 5 files changed; 3rd shows the overlaping procentage between logical and structural dependencies for commits with more then 5 and less then 20 files changed; 4th column shows the overlaping procentage between logical dependencies and structural dependencies for commits with more then 20 files changed; 5th column shows the overlaping procentage between logical and structural dependencies for all commits regardless of the number of files (this percentage is not always the sum of the other ones since logical dependencies are taken as unique and one logical dependency can be found in many categories);

ID	SD	LD+comments	Overlaps	LD-comments	Overlaps
Commits with less than 5 files changed					
1	48	59	15	49	12
2	3	6	3	6	3
3	5	51	2	51	2
4	6	19	0	19	0
5	51	5	1	5	1
6	210	21	5	19	5
7	104	27	2	27	2
8	116	88	16	83	16
9	193	239	36	217	34
10	521	1576	115	1480	112
11	15	64	5	58	5
12	331	217	33	200	31
13	363	649	53	581	50
14	23	14	4	14	4
15	1198	1062	76	962	64
16	373	1052	53	932	48
17	566	1515	210	1366	204
18	296	866	41	835	37
19	42	94,00	5,00	89,00	3,00
Commits with more than 5 and less than 20 files changed					
1	48	259	23	232	12
2	3	0	0	0	3
3	5	107	2	104	2
4	6	113	5	70	0
5	51	39	0	39	1
6	210	55	0	55	5
7	104	211	18	149	2
8	116	531	41	503	16
9	193	1532	89	1258	34
10	521	3044	142	2803	112
11	15	186	12	177	5
12	331	1617	124	1376	31
13	363	1763	118	1563	50
14	23	43	7	37	4
15	1198	5599	211	5013	64
16	373	4900	76	4006	48
17	566	3105	160	2609	204
18	296	2035	36	2021	32
19	42	336,00	0,00	327,00	0,00
Commits with more than 20 files changed					
1	48	190	17	105	16
2	3	0	0	0	0
3	5	0	0	0	0
4	6	0	0	0	0
5	51	0	0	0	0
6	210	0	0	0	0
7	104	5547	89	5530	89

ID	% less 5	% more 5 less 20	% more 20	% Total
1	31,25	47,92	35,42	77,08
2	100,00	0,00	0,00	100,00
3	40,00	40,00	0,00	40,00
4	0,00	83,33	0,00	83,33
5	1,96	0,00	0,00	1,96
6	2,38	0,00	0,00	2,38
7	1,92	17,31	85,58	85,58
8	13,79	35,34	26,72	71,55
9	18,65	46,11	65,80	72,02
10	22,07	27,26	59,31	65,64
11	33,33	80,00	0,00	86,67
12	9,97	37,46	41,09	64,65
13	14,60	32,51	47,93	63,64
14	17,39	30,43	21,74	47,83
15	6,34	17,61	72,37	75,63
16	14,21	20,38	28,69	54,96
17	37,10	28,27	45,41	79,15
18	13,85	12,16	39,52	39,50
19	11,90	0,00	16,66	28,50
Avg	24,33	36,26	35,33	71,47

Table 5.3: percentage rate of dependencies overlaps, case with comments

ID	% less 5	% more 5 less 20	% more 20	% Total
1	25,00	45,83	33,33	72,92
2	100,00	0,00	0,00	100,00
3	40,00	40,00	0,00	40,00
4	0,00	66,67	0,00	66,67
5	1,96	0,00	0,00	1,96
6	2,38	0,00	0,00	2,38
7	1,92	10,58	85,58	85,58
8	13,79	35,34	26,72	71,55
9	17,62	41,97	63,73	71,50
10	21,50	26,87	57,58	63,92
11	33,33	80,00	0,00	86,67
12	9,37	35,35	29,31	56,50
13	13,77	29,20	47,66	61,71
14	17,39	21,74	21,74	43,48
15	5,34	16,03	71,29	74,29
16	12,87	20,11	27,08	53,08
17	36,04	27,56	42,23	76,86
18	12,50	10,81	33,78	36,82
19	7,14	0,00	16,66	21,42
Avg	23,48	33,14	33,74	68,60

Table 5.4: percentage rate of dependencies overlaps, case without comments

6 | Conclusions

Bibliography

[1] David Binkley, *Source Code Analysis: A Road Map*, Future of Software Engineering, 2007. FOSE '07.

[2] N. Ajienka and A. Capiluppi, *Understanding the interplay between the logical and structural coupling of software classes*, J. Systems Software, vol. 134, pp. 120-137, 2017.

[3] Igor Wiese, Rodrigo Kuroda, Reginaldo Re, Gustavo Oliva, Marco Gerosa, *An Empirical Study of the Relation Between Strong Change Coupling and Defects Using History and Social Metrics in the Apache Aries Project*, IFIP International Conference on Open Source Systems, OSS 2015: Open Source Systems: Adoption and Impact, pp. 3-12.

[4] G. Booch, *Object-Oriented Analysis and Design with Applications*, Third Edition: Addison-Wesley, 2007.

[5] 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 (Volume: 35, Issue: 6, Nov.-Dec. 2009), pp. 864-878.

[6] Beck F., Diehl S., *On the congruence of modularity and code coupling*, In ESEC/FSE'11: European Software Engineering Conference and Symposium on Foundations of Software Engineering (2011), pp. 354-364.

[7] Liguo Yu,*Understanding component co-evolution with a study on Linux*, Empirical Software Engineering, April 2007, Volume 12, Issue 2, pp. 123-141.

[8] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato, *Version Control with Subversion*, http://svnbook.red-bean.com/en/1.7/svn.basic.version-control-basics.html, 2008.

[9] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, Michael L. Collard,*Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code*, 17th Working Conference on IEEE, 2010, pp. 119-128.

[10] Poshyvanyk, Denys, *Using information retrieval based coupling measures for impact analysis*, Empirical Software Engineering 14 ,2008, pp. 5-32.

[11] Collard, M. L., Kagdi, H., and Maletic, J. I., *An XML-Based Lightweight C++ Fact Extractor*, IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 134-143

[12] Collard, M.L., Decker, M., Maletic, J. I., *Lightweight Transformation and Fact Extraction with the srcML Toolkit*, IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'11), 2011, pp. 173-184.