Computer Science and Engineering Department
"Politehnica" University of Timisoara
Master of Information Technology

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

Adelina Diana Stana

Coordinated by
Assoc Prof. Ioana Sora

June 5, 2018

Contents

1 Introduction

1.1 Domain overview

Software systems are continuously in change. As long as the software system is used the changing process is never finished. 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 end of the development process when the system is no longer open to new features requested by the client and the maintenance is only made for the existing ones.

During the development process of a software system new classes and new methods to the existing classes are added in order to fulfill 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. Logical dependencies can be easily extracted from the versioning system (e.g. Subversion , Git) commits. (Figure 1.1).

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 affect the maintainability of the system and increase the realization effort 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.

1.2 Thesis objectives

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

In earlier studies, the relationships between logical and structural de-

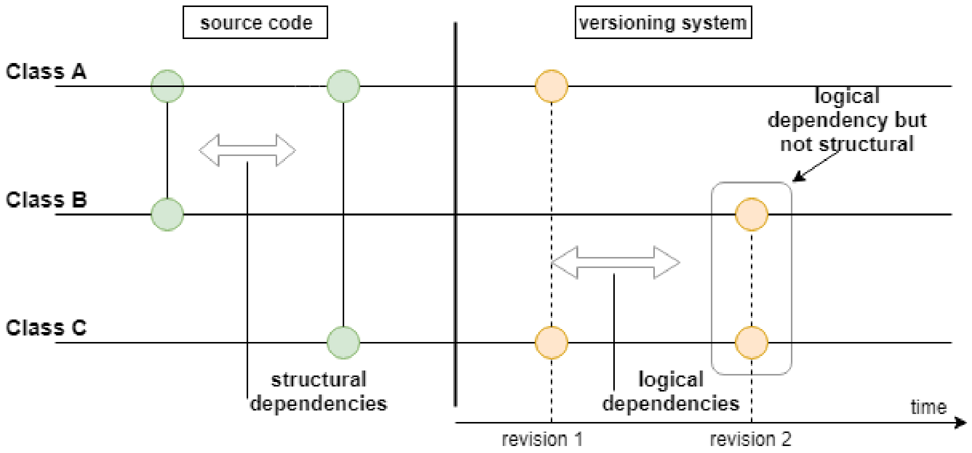


Figure 1.1: Relationships between structural and logical dependencies

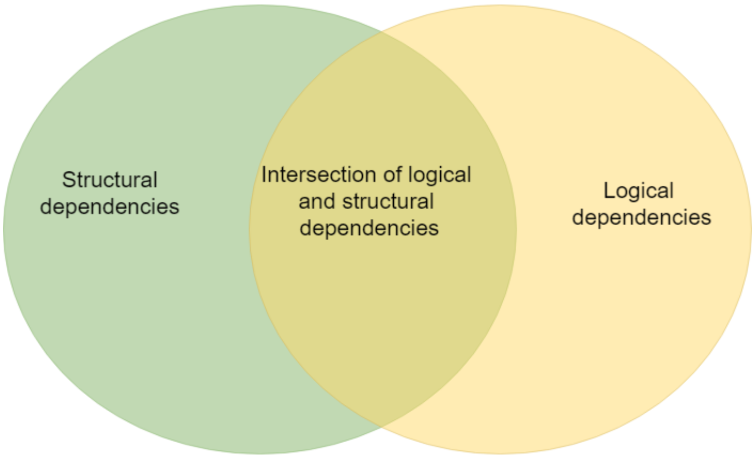


Figure 1.2: Venn Diagram showing the intersection between structural and logical dependencies

pendencies have been analysed (Oliva and Gerosa, 2011 [5] ; Yu, 2007 [9]; Oliva and Gerosa, 2015 [4], Cappiluppi and Ajienka, 2017 [2], Cappiluppi, Ajienka and Counsell, 2018 [3]). All of these studies showed that most of the structural dependencies do not co-evolve and a great part of logical dependencies are not doubled by structural dependencies.

In this work we build logical dependencies based on three questions with related motivations presented in the following :

Question 1 . How the number of source files changed in a commit can influence the logical dependencies of the system and the overlapping rates with the structural dependencies.
Motivation : 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 logical dependencies can be introduced since not all the files are updated in the same time for a development reason. Because of this,

Cappiluppi and Ajienka, in their works [3], [2] only take into consideration commits with less then 10 source code files changed in building the logical dependencies. But, on the other hand, sometimes large commits are unavoidable, e.g. a major change in API, so we will also study large commits but we will split the logical dependencies extracted into categories based on the number of files changed so that we can compare the impact of this sectioning on the final results.

Question 2. Considering comments can lead to additional logical dependencies ? How many logical dependencies are introduced by considering comment changes as valid changes and in what percentage this can influence the final result ?
Motivation : Not all the commits that have source code files changed include code changes , some of them can be only comments changes. Regarding this aspect, we can consider that there is no logical dependency between two classes that change in the same time only by comments changes . Some studies have not taken this aspect into consideration, so we will analyse the impact of not considering/ considering comments as valid changes on the results.

Question 3. One occurrence of a logical dependency is enough to consider it as valid ? If we consider only logical dependencies with more then one occurrence as valid, the results are influenced in a significant way ?
Motivation : One occurrence of a logical dependency between two classes can be a valid logical dependency, but can also be a coincidence. Taking into consideration only logical dependencies with multiple occurances as valid dependencies can lead to more accurate logical dependencies and more accurate results.
But if the project studied has a relatively small amount of commits, the probablilty to find multiple updates of the same classes in the same time can be small, so filtering after the number of occurences 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 analyse also the impact of this filtering on different projects.

In order to answer these research questions, we have built a tool that extracts structural and logical dependencies on different scenarios, the design and implementation of the tool is presented in chapters 3 and 4.

We have analyzed 19 open-source software systems of different sizes within the tool developed, the experimental results obtained being presented in chapter 5 . The final chapter, chapter 6, discusses the conclusions drawn from our experimental research.

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 [6]. Dependencies are discovered by analysis of source code or from an intermediate representation such as abstract syntax trees [7] .

2.2 Structural dependencies

Structural dependencies (a.k.a syntactic dependencies or structural coupling) are the result of source code analysis. Each source code file can contain one or more classes [8].

In object-oriented programming, a class is an template for creating objects that provides initial values for member variables and implementations of functions or methods. [17]

In the following we will present the class components and how those components can introduce software dependencies with other classes.

Class components to discuss :

- Class name. Inheritance.
- Comments.
- Attributes.
- Constructors.
- Methods.

Class name. Inheritance

The user-defined objects are created using the class keyword. The class is a blueprint that defines a nature of a future object. The name of the class is the name located after the keyword class. During the structural analysis of the projects the classes names are extracted in order to identify the structural dependencies elements names .

Listing 2.1: Class declaration.

```
public class A {
}
public class B extends A {
}
```

A structural dependency between a class and other class can be given by inheritance. Inheritance can be defined as the process where one class acquires the properties of another class [17] .
The class which inherits the properties of other class is known as subclass and the class whose properties are inherited is known as superclass (a.k.a base class, parent class). The name of the superclass is the name located after the keyword extends .

Comments

Comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information about a variable, method or any other statement. There are multiple types of comments :

/* text */ - ignores characters from begining of /* to */.
/** documentation */ - indicates a documentation comment, just like comments that use /* and */.
// *text* - ignores everything from // to the end of the line.

Comments never introduce any kind of structural dependencies. They are used just for the source code intelligibility.

Attributes

Attributes represent the state of the object because they store the information about the object. An attribute is another term for a field.

private B b;
The keyword private signifies that a method or variable can be accessed only within the declaring object. The keyword public allows access also from outside of the object [18], [13] .

The attributes or members of a class can lead to structural dependencies between the class and the class types of the attributes.

Constructors

A constructor of a class is a member function that is executed whenever we create a new object of that class. A parameterized constructor introduces new structural dependencies by its parameters.

Methods

Some of these attributes and methods are publicly visible from outside the object: the interface. Other attributes and methods are for private use of the object itself : the implementation. Either ways when studying structural dependencies the publicity of an attribute or of a method is irrelevant. [17] The relevant part is the type of the attribute or the return type and the parameters of the method. A method can introduce structural dependencies in three ways : by its return type , by its call parameters or by its local variables .

On the other hand, 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 [9] . 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

In software engineering, we use co-evolution to represent the phenomenon in software evolution that change in one component C2 in response to a change in another component C1 [9], [4], [2]. There are differences between co-evolution and evolutionary coupling.

Co-evolution is the result of causeeffect changes between two components: changes to one component require changes to another component. On the other hand, evolutionary coupling is based on the evolution history of two components and is a measurement of the observation that two components are changed at the same time.

Modifications made to two components at the same commit do not necessarily indicate the co-evolution of the two. These modications could be completely unrelated.
So, evolutionary coupling could also be determend accidentally by two components changing in the same commit and it will bring noise to the measurement of evolutionary coupling. [9]
We will try to filter that noise by setting different thresholds in the process of analysing the systems.

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 [10].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 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 or a folder renaming. 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, files that changed only by comments change can lead to redundant

logical dependencies. If class A and class B change together but the only change was a change in the comments then there may not be any logical dependency between them. For each category mentioned above, 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 a logical dependency.

Another filtering method that can lead to a more accurate results is to count the occurrences of the logical dependencies. If class A and class B change together but only once during all the commits from the active branch the class A and B may or may not represent a logical dependency.

3 | Implementation details

The programming language used for developing the tool is ***Python*** . Python is a general-purpose interpreted, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. [16] In addition ***PyQt*** is used as UI layer. PyQt is a Python binding for the Qt cross-platform C++ framework.

3.1 Processing phases

In order to build structural and logical dependencies the 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 3.1):

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

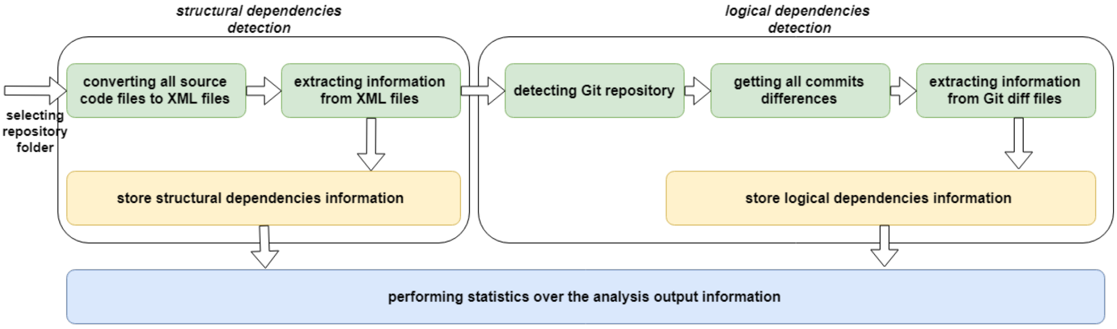

structural dependencies detection
logical dependencies detection
selecting repository folder
converting all source code files to XML files
extracting information from XML files
detecting Git repository
getting all commits differences
extracting information from Git diff files
store structural dependencies information
store logical dependencies information
performing statistics over the analysis output information

Figure 3.1: Processing phases

3.2 Architecture

MainDialog is responsible for the User Interface creation . This class also handles the events from the user (process button pressed , load files button pressed, close window button pressed). MainDialog uses the TitleBar class for the title bar creation and CheckableDirModel for file system tree view.

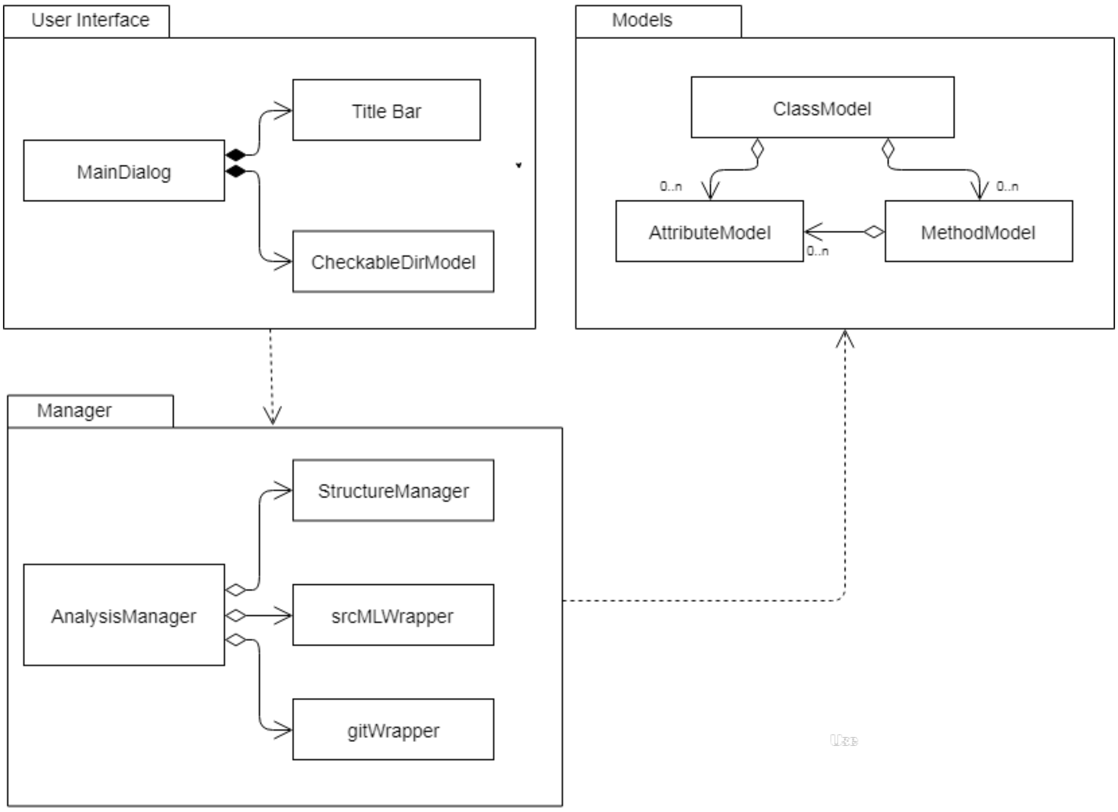


Figure 3.2: UML diagram of the tool

The ***CheckableDirModel*** inherits QDirModel class which provides a data model for the local filesystem.In addition CheckableDirModel adds checkboxes to each filesystem item for a more easyer selection and returns a list of all files selected.

TitleBar inherits QMenuBar class which provides a horizontal menu bar. TitleBar class is used to create a new title bar, different from the one provided by Windows system . The class handles events like minimize , maximize and close window which are then emited to MainDialog class .

A ***ClassModel*** object contains all the informations extracted from the structural analysis about a class. ClassModel class contains a member for the name of the class , one for the parent name of the class and one for the xml file path in which the class was found. It also contains a list of MethodModels, a list of AttributeModels and 3 lists for git links, one for links found in commits with below 5 files changed , one for above 5 files changed and below 20 and one for links found in commits with above 20 files changed.The git links lists are lists of string objects. The class provides setters and getters for all the members mentioned above for retrieving and updating their values.

A ***MethodModel*** object contains all the information extracted from the structural analysis about a method of a class. MethodModel contains a member for the name of the method and two lists of AttributeModels . One is used for the call parameters of the method and one is used for the local variables found inside the method.

AttributeModel contains two members , type and name. ”Type” is used to identify the object type and ”Name” is used to identify the object name. The class provides setters and getters for the members mentioned above for retrieving and updating their values.

AnalysisManager is created and called by *MainDialog*. MainDialog creates a AnalysisManager object and passes to it a list of source code files paths to be analysed and the parent folder path. The path to the parent folder is used to identify the Git repository. AnalysisManager is called each time the user triggers events in the User Interface. AnalysisManager calls the srcMLWrapper for XML conversion and XML files parsing and data extraction.
All the data extracted are passed to StructureManager. It also calls GitWrapper which interogates the project repository and saves all the commit differences in a temporary folder. The class is also responsable for parsing all the diff files and extracting the git links. The git links are passed to StructureManager which splits them into categories and setts them to the coresponding ClassModel object (figure 3.3). Finally is responsable for plotting the results of different queries (Example : all the overlappings between structural dependencies and logical dependencies extracted from commits with less then 5 files changed) .

StructureManager contains all the ClassModels extracted and is used by the AnalysisManager to save all datas extracted to an XML file and to load the datas extracted from an XML file (each project has an coresponding XML file that is created automaticaly after the analysis is done). Also is responsable to add the git links to the coresponding classes .

srcMLWrapper has methods for converting source code files into XML files and to parse the resulting XML files in order to extract informations about classes , methods , members.

GitWrapper identifies the git repository of the path given as a parameter and creates a temporary folder in which all the differences files are stored.

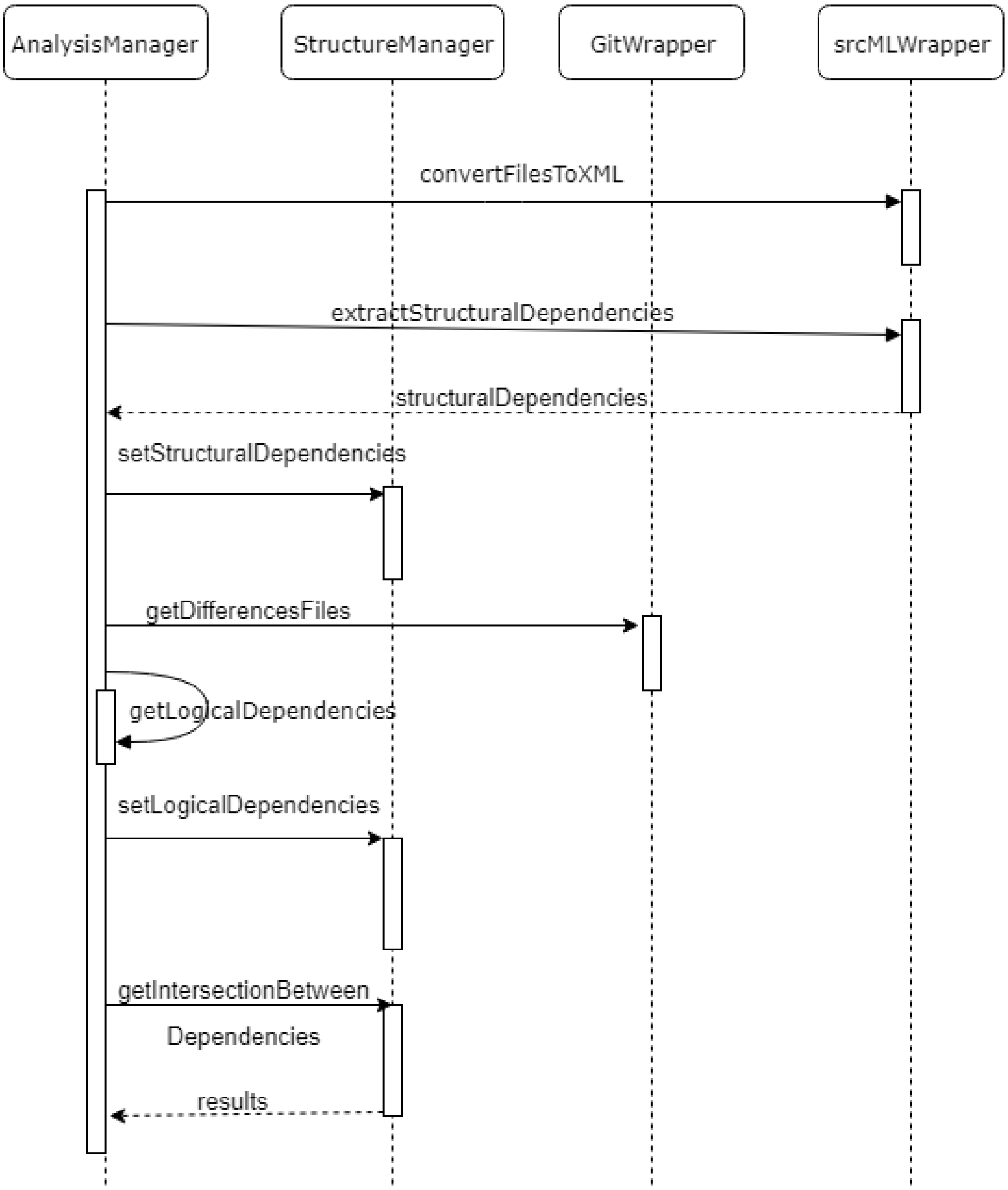


Figure 3.3: Sequence diagram

3.3 User Interface

Creating MainDialog components

MainDialog class inherits QMainWindow from PyQt which provides a main application window. The window is set as frameless window since a custom TitleBar will be attached to it.

Listing 3.1: Create the title bar for the main window.

```
self.setWindowFlags(QtCore.Qt.FramelessWindowHint)
self.titleBar = TitleBar(self)
self.setMenuBar(self.titleBar)
```

In addition other widgets will be created : progress line , progress bar , tree view of the file system . All the above will be added after creation to the main layout of the window. There are two types of layouts in PyQt : *QVBoxLayout* and *QHBoxLayout* . The first layout lines up widgets vertically and the second horizontally.

Listing 3.2: Add widgets to the main window layout.

```
self.boxLayout = QVBoxLayout(self)

self.boxLayout.addWidget(self.tree)
self.boxLayout.addWidget(self.progressLine)
self.boxLayout.addWidget(self.progressBar)
```

Creating Menus

Qt implements menus in QMenu and QMainWindow keeps them in a QMenuBar. QAction is an abstraction for actions performed with a menubar and can be created with an icon and a name. After the creation, the action is connected to a method, so when we select a particular action, a triggered signal is emitted and the specified method executes.

Listing 3.3: A menu creation with two actions.

```
processAction = QAction(QIcon('resources/run.png'), 'Process Files', self)
xmlLoadAction = QAction(QIcon('resources/load.png'), 'XML File', self)

processAction.triggered.connect(self.processFilesClicked)
xmlLoadAction.triggered.connect(self.loadXmlClicked)

self.toolbar.addAction(processAction)
self.toolbar.addAction(xmlLoadAction)
```

3.4 Extracting software dependencies

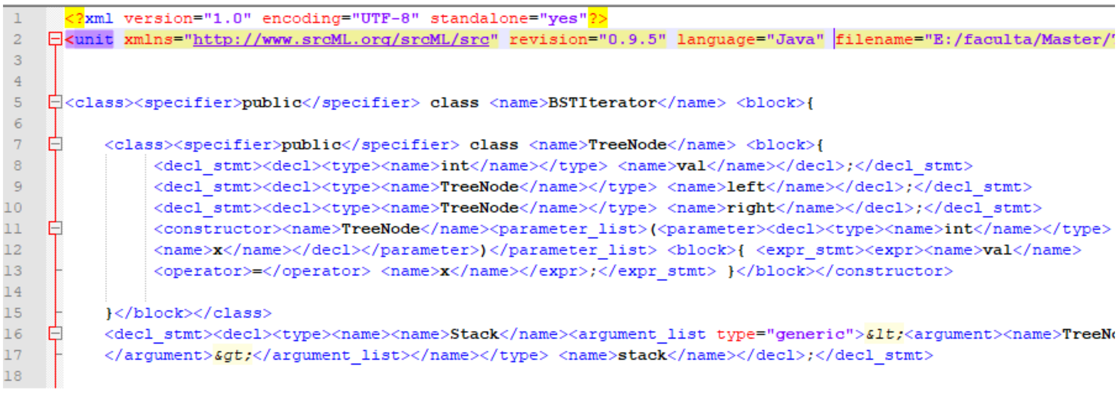
3.4.1 srcML

SrcML is an open source tool that converts in an XML format C/C#/Cpp/Java source code. [14]

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.).[15] An example of the XML representation can be found in Figure 3.4 .



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.srcML.org/srcML/src" revision="0.9.5" language="Java" filename="E:/faculta/Master/


<class><specifier>public</specifier> class <name>BSTIterator</name> <block>{

    <class><specifier>public</specifier> class <name>TreeNode</name> <block>{
        <decl_stmt><decl><type><name>int</name></type> <name>val</name></decl>;</decl_stmt>
        <decl_stmt><decl><type><name>TreeNode</name></type> <name>left</name></decl>;</decl_stmt>
        <decl_stmt><decl><type><name>TreeNode</name></type> <name>right</name></decl>;</decl_stmt>
        <constructor><name>TreeNode</name><parameter_list>(<parameter><decl><type><name>int</name></type>
        <name>x</name></decl></parameter>)</parameter_list> <block>{ <expr_stmt><expr><name>val</name>
        <operator>=</operator> <name>x</name></expr>;</expr_stmt> }</block></constructor>

    }</block></class>
    <decl_stmt><decl><type><name><name>Stack</name><argument_list type="generic">&lt;<argument><name>TreeN
    </argument>&gt;</argument_list></name></type> <name>stack</name></decl>;</decl_stmt>

```

Figure 3.4: Example of XML representation of source code.

3.4.2 Convert files to XML using srcML

As mentioned above, for conversion from source code to XML we will use srcML. Through UI the user can choose the main folder of the to-be analysed project. The tool will walk trough all the subfolders and files of the folder and will get only the files that are source code files (files with extension such as .cpp ,.c++, .h , .java).

Listing 3.4: Walk through the given directory and extract only source code files

```python
for path, dirs, files in os.walk(self.filePath(index)):
    for filename in files:
        if QtCore.QFileInfo(filename).completeSuffix() in acceptedSuffix:
            selection.add(os.path.join(path, filename))
```

The next step is to convert each file to XML. The conversion will be made through calls to srcML using the **subprocess** module . The subprocess module allows to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. Arguments to pass to srcML :

- the path to the file to-be-converted
- ”-o” option , stands for outputting to a file
- the output file path

Listing 3.5: Convert source code to XML using subprocess module.

```
def convertFiles(self, file):
    file_name = os.path.basename(file)
    file_xml = self.workingDir + "/"+file_name + ".xml"
     cmd = "srcml \""+file+"\" -o \""+file_xml+"\""
     rez = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE).stdout.read()
```

3.4.3 Extracting structural dependencies from XML

The ElementTree library

For XML parsing we used **The ElementTree** library.It includes tools for parsing XML using event-based and document-based APIs, searching parsed documents with XPath expressions, and creating or modifying existing documents. Parsing an entire document with parse() returns an ElementTree instance. The tree knows about all of the data in the input document, and the nodes of the tree can be searched or manipulated.

Listing 3.6: Import and create ElementTree instance

```
import xml.etree.ElementTree as ET
tree = ET.parse(file)
```

Class extraction

Element.findall(”name”) finds only elements with the tag ”name” which are direct children of the current element. The items returned by findall() are Element objects, each representing a node in the XML tree. Each Element has attributes for accessing the data of the XML.

Listing 3.7: Get name of Element item.

```
def getName(self, item):
       if item is not None:
           text = item.text
           if text is None:
               return self.getText(item.find("name"))
           return text
```

One xml file can contain one or more class types. Also one class can contain other class definitions so the parsing will be made recursively. Each time a new class is found a ClassModel object is created and added to a list.

Listing 3.8: Recursively find all the classes from XML file.

```
def getClassModelJava(self, file, root):
        classList = []

        for item in root.findall("{http://www.srcML.org/srcML/src}class"):
            className = self.getName(item)
            insideClassList = self.getClassModelJava(file, item)

            if insideClassList:
                classList.extend(insideClassList)

            classModel = ClassModel()
            classModel.setFile(file)
            classModel.setName(className)
        ....
```

Method extraction

A ClassModel has a list of attributes and a list of methods. The methods are relevant because their call parameters and attributes can create structural dependencies. In order to get the parameters names we will look for **parameter_list** tag inside the tree. For attributes we will look for **decl_stmt** tag .

Listing 3.9: Get the parameter list of a method.

```
method = MethodModel()
method.setType(element_type)
method.setName(element_name)

for param in self.getAttributes(self.getItem(decl, "parameter_list"), "parameter"):
    method.addArgs(param)
....
classModel.addMethod(method)
```

Attribute extraction

Both ClassModel and MethodModel have a list of attributes. Those attributes give the structural dependencies of the class . The call parameters of a function are also considered as attributes . We will refer to the AttributeModel class as a class that contain the information required to identify one side of the structural depencency link. The other side is the ClassModel containing the AttributeModel .

Listing 3.10: Get the attributes list of a method or class.

```
for a in self.getAllItems(decl, "decl"):
   element_type = self.getType(a)
   element_name = self.getName(a)

   attribute = AttributeModel()
   attribute.setType(element_type)
   attribute.setName(element_name)

   attributes.append(attribute)
 return attributes
```

3.5 Extracting logical dependencies

The second step of the analysis is to extract logical dependencies from the versioning system. Changes can be made by many individuals over the years. Changes include the creation and deletion of files as well as edits to their contents. Each change (this can include changes in multiple files) made by an individual at certain point of time is contained into a commit[10].
The tool looks through the repository and gets all the existing commits, for each commit a differences file will be made .After all the differences files are stored , all the files are parsed and logical dependencies are build.

3.5.1 Git repository configuration

To get informations about a git repository we use ***GitPython*** library that provides object model access to the git repository. The first step is to create a git.Repo object to represent the repository, with the local repository path as argument. A repo object provides high-level access to the repository data and allows you to create and delete heads and access the configuration of the repository. After the Repo object creation, the repository needs to be checked if it's bare or not .
A bare Git repository is typically used as a remote repository that is sharing a repository among several different people. The developers don't do work inside the remote repository so there's no Working Tree (the project files that you edit), just bare repository data. If the repository is not bare then other configurations can be checked : repository description, active branch , number of commits.

Listing 3.11: Get informations about a git repository.

```
repo = Repo(self.repo_path)
if not repo.bare:
        print('Repo description: '.format(repo.description))
        print('Repo active branch is {}'.format(repo.active_branch))
        branches = repo.remotes.origin.refs
        print('Number of branches : {}'.format(len(branches)))
        for branch in branches:
            commits = list(repo.iter_commits(branch))
      print('Branch named {} - commits number: {}'.format(branch, len(commits)))
```

3.5.2 Get diff files of active branch

The tool iterates through all the commits from the active branch and creates the diff files. The differences file will contain all the changes from all the files merged. Each file will be saved with ”_FilesChanged_” followed by the number of source code files changed in his name. In this way it will be more easy to extract the number of files changed and to filter the dependencies found by the number of files changed.
If a commit has 0 source code files changed then the commit will be ignored and no diff file will be created. The diff file will be created through a system call to git

since GitPython does not have this functionality. All the diff files will be saved in a temporary folder.

Listing 3.12: Creating the diff files of the active branch commits.

```python
commits = list(repo.iter_commits('master'))
index = 0
for commit in commits:
    parent = commit.parents[0] if commit.parents else EMPTY_TREE_SHA
    nrOfFilesChanged = self.getNrOfChangedFiles(commit, parent)
    if nrOfFilesChanged >= 1:
        os.system("git diff "+parent.hexsha+" "+commit.hexsha+" > "+self.repo_path+
"\~diffs\diff"+str(index)+"_FilesChanged_"+str(nrOfFilesChanged)+".txt")
        index += 1
```

3.5.3 Get logical dependencies from diff files

All the actions mentioned in subsections 3.5.1 and 3.5.2 are made by GitWrapper. The process of logical dependencies extraction from diff files is made by the AnalysisManager.
The first step in getting the logical dependencies from a diff file is to get the number of changed files.

Listing 3.13: Get number of changed files from file name.

```python
file = file.replace('.txt', '')
nrOfCommitsStr = file.split('FilesChanged\_')[1]
nrOfCommits = int(nrOfCommitsStr)
```

The diff file can contain comments, in order to build logical dependencies without comments we need to call a function that removes all the comments from the diff file . The removing is made with the regular expressions library , re.

Listing 3.14: Remove comments from file.

```python
def removeComments(self, filepath):
        string = open(filepath).read()
        string = re.sub(re.compile("/\*.*?\*/", re.DOTALL), "",string)
        string = re.sub(re.compile("//.*?\n"), "", string)
        return string
```

Next we need to identify the classes change. We will iterate through all the lines of the diff file and search for key words like "class" or "private class". If a line contains the keywords the the line will be split into words and the name of the class will be extracted.

Listing 3.15: Get classes changed from the diff file.

```python
if re.search('.*class .*\{', line) or re.search('.* public class.*', line)
or re.search('.* private class .*', line):
    words = line.split(' ')
    for i in range(0, len(words)):
         word = words[i].strip()
         if word == 'class' :
                    gitlist.append(words[i + 1].strip())
```

Finally, after the entire file is parsed and all the git links are gathered, we will add all the git links to the corresponding classes by calling *setGitLinksToClass* method of the structure manager. For each class name found in the diff file all the other class names will be set as git links.

Listing 3.16: StructureManager call to set the git links.

```
if len(gitlist) > 1:
 for className in gitlist:
    self.structureManager.setGitDependenciesToClass(className, gitlist, nrOfCommits)
```

3.5.4 Splitting git dependencies on categories

The following step after extracting all the git dependencies found in a diff file is to set the dependencies to the corresponding git classes .
For example, if in one diff file we found that classes A , B and C are updating together, then class A has logical dependencies with class B and C, Class B has logical dependencies with class A and B and Class C has logical dependencies with class A and B.

One case often encountered is to found between the git dependencies list passed to the StructureManager some classes that are not known (their name is not among the classes held in StructureManager). Since git dependencies are extracted from all the commits over time it can be posible that on class has been deleted or renamed meanwhile.
We will not take into consideration such cases since we only study the stability of the current state of the system . Structural dependencies are only identified from the source code of the last commit so we will only study the evolution of the classes found in it. So the classes that are not known will be ignored.

Listing 3.17: Set git dependencies to the coresponding classes.

```
def setGitDependenciesToClass(self, className, listOfDep, nrOfCommits):
   flag = False
     for classStruct in self.classlist:
        if classStruct.getName() == className:
             flag = True
             classStruct.setGitDependencies(listOfDep, nrOfCommits)
     if not flag:
             print("Class: " + className+" not found!")
```

Each ClassModel has three lists for git dependencies (mentioned in this document also as links) . One for links found in diff files with less then 5 files changed, one for dependencies found in diff files with more then 5 and less then 20 files changed and one for dependencies found in diff files with more then 20 files changed .

As mentioned in the previous section, ClassModel is responsible for dividing the dependencies received from StructureManager, so as for each class in the received list all the other classes are set as logical dependencies.

Listing 3.18: Split dependencies in categories.

```
def setGitDependencies(self, dependencies, nrOfCommits):
   for dep in dependencies:
              if dep != self.name:
                   if nrOfCommits <= 5:
                       self.git_dep_below5.append(dep)
                   if 5 < nrOfCommits <= 20:
                       self.git_dep_below20.append(dep)
                   if nrOfCommits > 20:
                       self.git_dep_more20.append(dep)
```

Because the entire list of git dependencies received from the AnalysisManager is passed to each ClassModel without taking out from the list the called class for which the other classes are set as logical dependencies, we will need to parse inside ClassModel the entire list and to make sure that the class is not added as logical dependencie to itself.

3.6 Overlappings between logical and structural dependencies

Create unified list of structural dependencies for each class

Each ClassModel has a list of members and a list of methods . Each method contains also a lists of local variables and a list of call arguments . In order to build the lists with all the structural dependencies found in each class an additional list will be created in each ClassModel. The list will contain all the classes found in the members list of the class, call arguments, local variables of the methods and the superclass if exists.

Listing 3.19: Create unified list of structural dependencies.

```
self.relation_list = []
for attrib in self.attributes:
    self.relation_list.append(attrib.getType())

for method in self.methods:
   for arg in method.getArgs():
      self.relation_list.append(arg.getType())

for local in method.getLocals():
   self.relation_list.append(local.getType())

if self.superclass != "None":
   self.relation_list.append(self.superclass)

self.relation_list = [x for x in self.relation_list if x not in self.relation_list]
```

The obtained list is filtered for duplicates. So the resulting list contains unique structural dependencies between the class and other classes.

Optain overlapings with logical dependencies

Once the structural dependencies list is build, we can find overlapping between the list and the structural dependencies lists.

Listing 3.20: Get overlapping between structural dependencies and logical dependencies found in diff files with less then 5 files changed.

```
def getMatch5(self):
        return set(self.relation_list).intersection(self.git_dep_below5)
```

3.7 Saving and restoring the information processed

The process of saving all the informations about classes and structural and logical dependencies is made by StructureManager. The informations are saved in a single XML file, later the file can be loaded in the tool and by this a lot of time will be saved since it will be no need build the logical and structural dependencies from scratch.
For each member of the ClassModel , MethodModel and AttributeModel classes a xml tag is created. The XML file creation is made using ElementTree library. The Element class knows how to generate a serialized form of its contents, which can then be written to a file.
There are two functions used for creating a hierarchy of Element nodes. Element() creates a standard node and SubElement() attaches a new node to a parent.

Listing 3.21: Save informations to XML file using ElementTree.

```
def saveToXml(self):
    data = ET.Element('data')
    for classItem in self.classlist:
         classElement = ET.SubElement(data, 'class')
         className = ET.SubElement(classElement, 'name')
         className.text = classItem.getName()

         attribElement = ET.SubElement(classElement, 'attributes')
                for attribItem in classItem.getAttributes():
                     attrib = ET.SubElement(attribElement, 'attribute')
                     attribName = ET.SubElement(attrib, 'name')
                     attribName.text = attribItem.getName()

         gitLinksElement = ET.SubElement(classElement, 'gitlinksmore5below20')
         gitList = ",".join(gitLinksList)
         gitLinksElement.text = gitList

     filedata = ET.tostring(data)
     file = open(path, "w+")
     file.write(mydata)
```

For the XML loading part , ElementTree is used again, the tool looks for tags like class , attribute, gitlinks and rebuilds all the structure with the collected informations.

Listing 3.22: Load informations from XML file using ElementTree.

```
tree = ET.parse(file)
root = tree.getroot()
for item in root.findall("class"):
   classModel = ClassModel()
   classModel.setFile(self.getItemTextByName(item, "file"))
   classModel.setName(self.getItemTextByName(item, "name"))
```

3.8 Plots creation

After the logical and structural dependencies are extracted, a series of plots will be created by the tool . The plots are displayed by the tool but also saved as images in a results folder together with the generated XML file.
For plots creation **Matplotlib** and **NetworkX** is used. Matplotlib is a Python 2D plotting library and it provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, PyQt.

Listing 3.23: Create plot with networkx.

```
 import matplotlib.pyplot as plt
plt.figure(1)
g = nx.Graph()
for classItem in self.structureManager.getClassList():
    g.add_node(classItem.name)
    related_list = classItem.getMatch()
    for related in related_list:
        g.add_edge(classItem.name, related)

plt.title("Code+ Git Links. Count: " + str(g.number_of_edges()))
```

A Graph is a collection of nodes and pairs of nodes (called edges or links). In NetworkX, nodes can be any hashable object : a string, an image or an XML object. The graph creation is made by calling the Graph method of networkx. First, the class list held by StructureManager is parsed and nodes with the classes names are created. For each class a list of edges is build in function of the purpose of the graph .
For example if we build a graph with the overlappings between structural and all logical dependencies, we need to aquire for each class the list of coresponding logical dependencies list . The list will be parsed and edges will be created between the class and all the items of the list.

4 | Tool usage

The tool has multiple usage options that will be presented in the following sections.

4.1 Loading a project

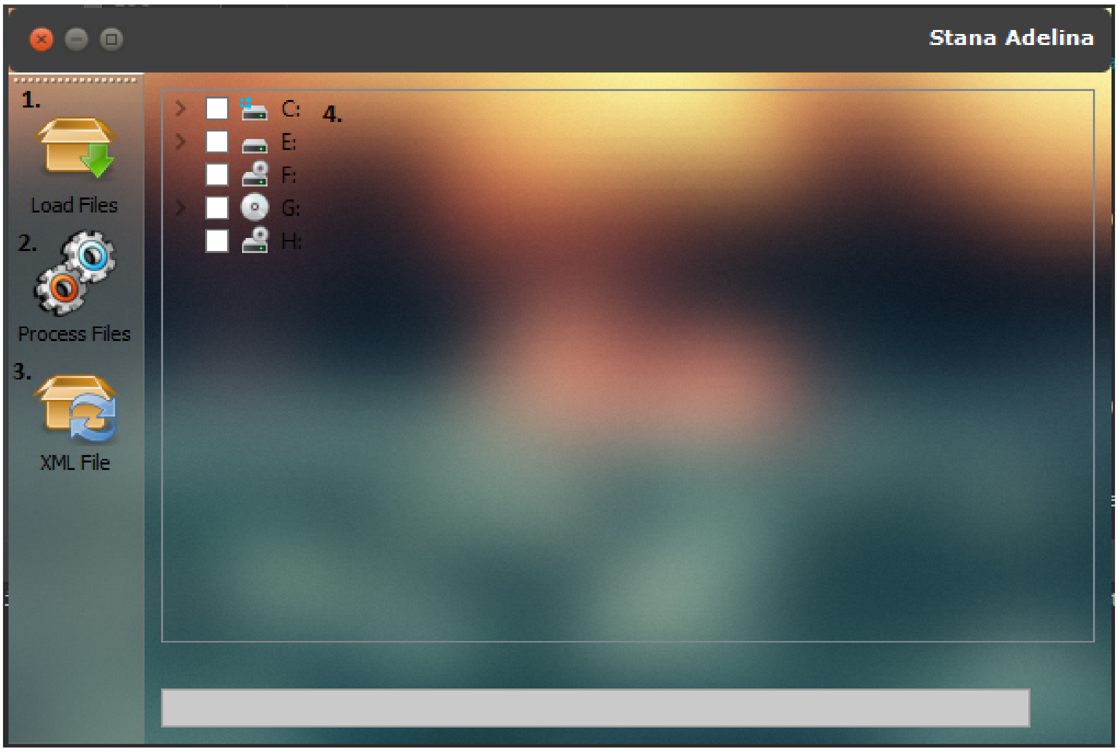
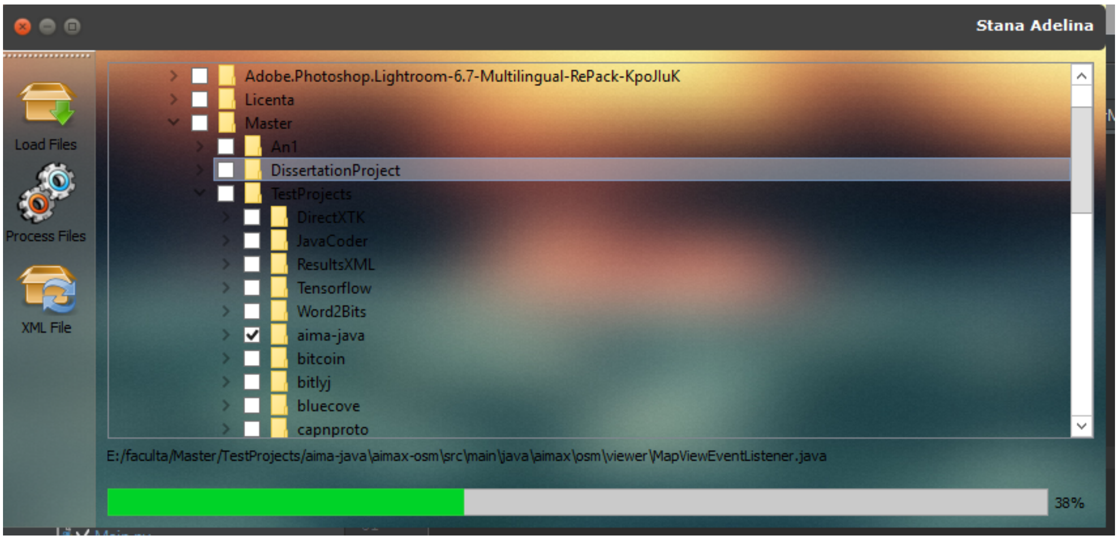


Figure 4.1: User interface

For building the logical and structural dependencies the project root folder is needed. The tool offers a tree view of all the file system as shown in 4. of figure 4.1 .
The user does not have to select manualy each file, the selection of a folder implies the automatic selection of all subfolders and files. Also there is no need to select only the source code files, the tool processes all the files selected and only the ones with the accepted extensions are taken into consideration. To load all the files selected in the tool the button *Load Files* needs to be pressed shown in 1. of figure 4.1 .

4.2 Processing files

The next and final step to obtain the results is to press *Process Files* button shown in 2. in figure 4.1 . The tool has a process bar wich shows to the user the percent of files processed as shown in figure 4.2 .



E:/faculta/Master/TestProjects/aima-java\aimax-osm\src\main\java\aimax\osm\viewer\MapViewEventListener.java

Figure 4.2: Loading files process bar

This step implies the creation of XML files corresponding to the source code files , identifying the Git repository of the project, optaining all the structural and logical dependencies and their overlapings graphs .

4.3 Load XML of previous run

Once the logical and structural dependencies are extracted the tool builds and saves a XML file with all the informations collected. So, if a project needs to be reanalysed, only the XML file needs to be uploaded. In this way a lot of time is saved since is no need to recolect data from source code files and commits. To perform this action the Load XML button (number 3. in figure 4.1) needs to be pressed and a file browse dialog will appear so the user can choose the file as shown in figure 4.3.

The xml file is saved in the *results* folder created by the tool. The tool creates for each project analysed 3 folders , 2 temporary folders which are deleted after the process run and one results folder which is not deleted after the process run. The first temporary folder is the one that contains all the source code coresponing XML files , the second contains all the differences files extracted from the project commits. The last one is the *results* folder which contains all the graphics generated and the XML file with all the data extracted.

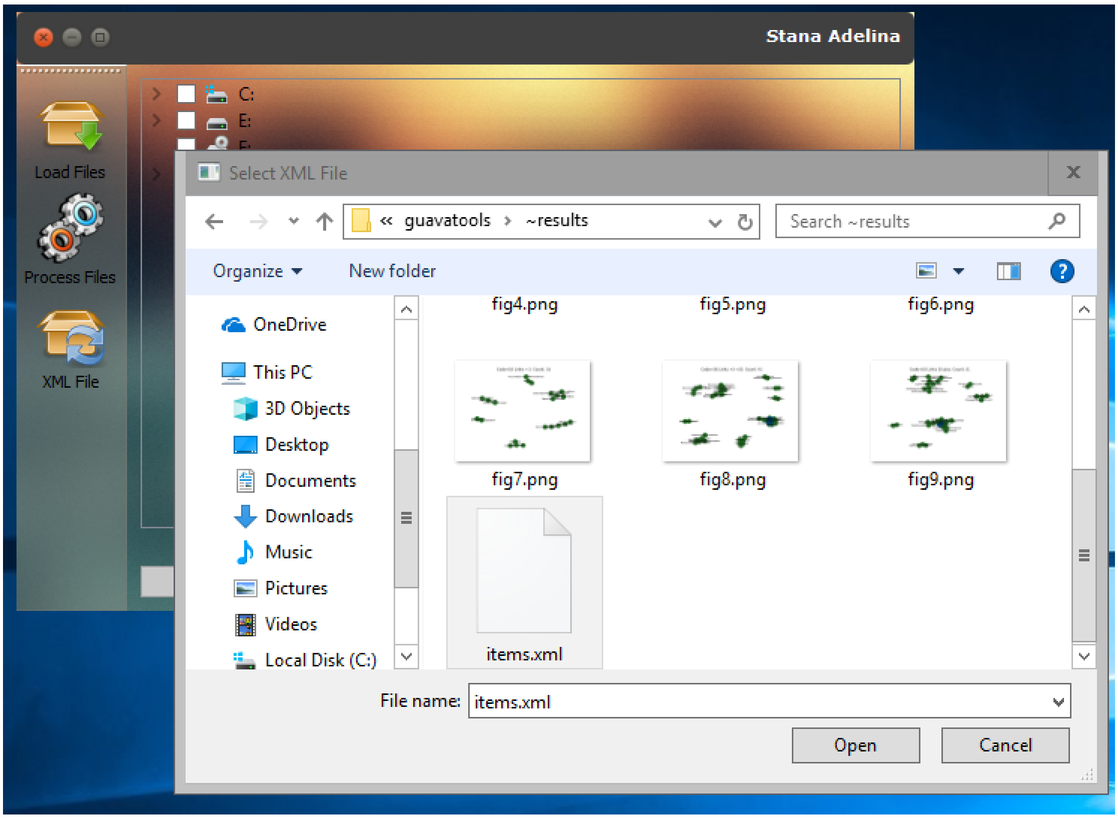


Figure 4.3: Loading a XML of a previous run.

4.4 Output graphics

The output of the analysis made by the tool is represented as a series of graphics. Each graphic has in it's title the semnification of the displayed values and their counter (Figure 4.4). For example for structural and logical dependencies found in files with less then 5 files changed the tool will display a graphic with the name "Overlapping structural and logical for more the 5 files changed . Counter : numberOfOverlaps" . The tool will create graphics for the following casess:

- Structural dependencies found.
- Logical dependencies found in commits with less then 5 files changed.
- Logical dependencies found in commits with more then 5 files changed and less the 20.
- Logical dependencies found in commits with more then 20 files changed.
- The intersection of all the logical dependencies from all the 3 categories.
- The intersection between structural and logical dependencies found in commits with less then 5 files changed.

- The intersection between structural and logical dependencies found in commits with more then 5 files changed and less the 20.
- The intersection between structural and logical dependencies found in commits with more then 20 files changed .
- The intersection between structural dependencies and the intersection of all the logical dependencies from all the 3 categories .

Figure 4.4

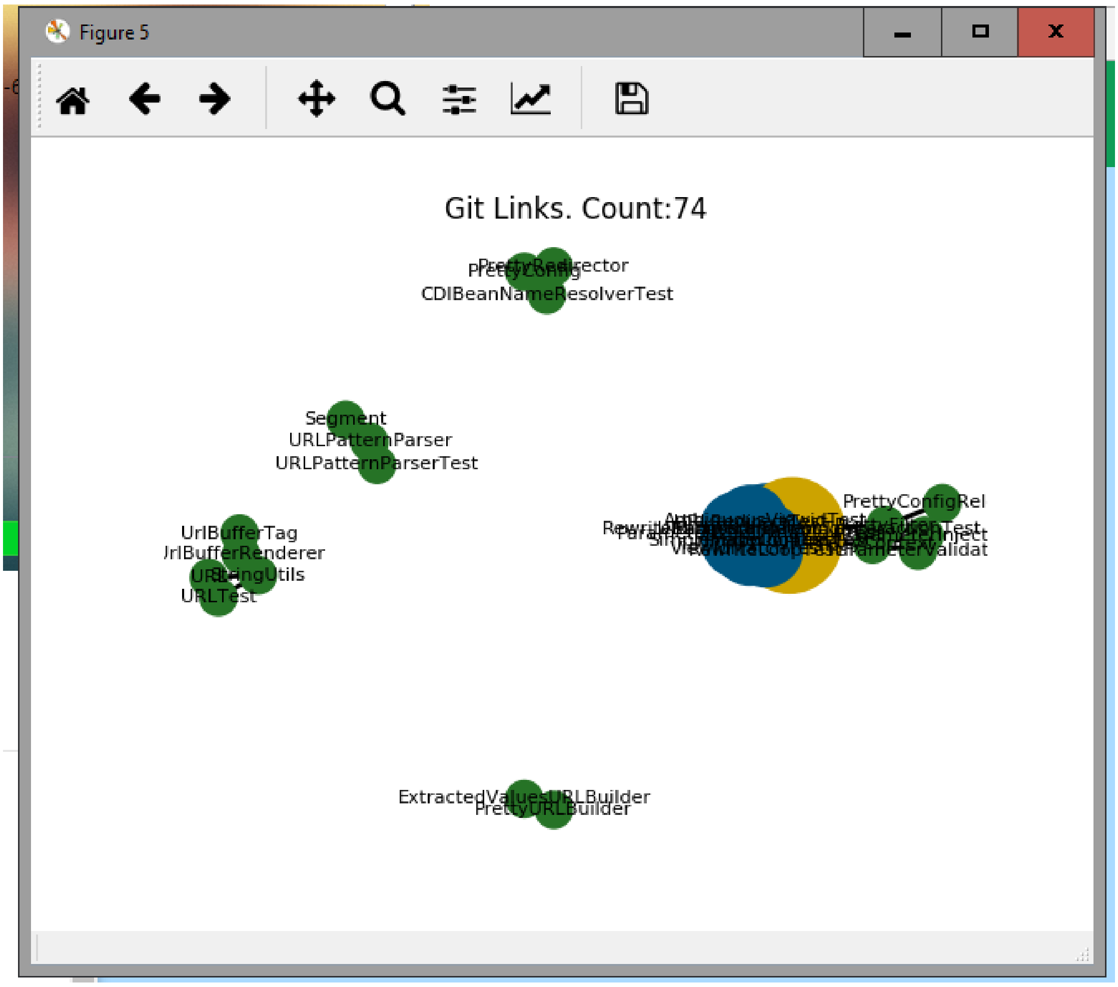


Figure 4.4: Output graph from the analysis

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 [9], [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 active 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	39	89	
2	JavaCoder	4	11	
3	jbandwidthlog	14	54	
4	sjava-logging	18	62	
5	daedalum	62	29	
6	prettyfaces	257	207	
7	jbal	102	113	
8	guavatools	209	85	
9	monome-pages	196	280	
10	kryo	289	743	
11	bitlyj	21	81	
12	slema	267	368	
13	bluecove	386	1679	
14	gp-net-radius	25	28	
15	aima-java	818	1181	
16	powermock	803	1512	
17	restfb	713	1545	
18	Tensorflow	1104	2386	
19	magnum	143	1728	

Table 5.1: Summary of open source projects studied.

ID	SD	LD+comments	Overlaps	LD-comments	Overlaps
			Fasting Value		
1	52	59	15	49	12
2	5	6	3	6	3
3	8	51	5	51	5
4	8	20	0	19	0
5	66	5	2	5	2
6	264	21	5	19	5
7	106	27	2	27	2
8	138	89	19	84	19
9	250	239	40	217	38
10	566	1576	129	1505	126
11	16	64	5	58	5
12	358	217	37	200	34
13	447	649	61	581	58
14	26	14	4	14	4
15	1463	1062	101	973	86
16	466	1052	73	938	68
17	832	1515	297	1371	286
18	293	867	45	861	39
19	46	94	9	90	8

Table 5.2: Results for commits with less than 5 files changed

Table 5.2, illustrates results for commits with less than 5 files changed. 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.

ID	SD	LD+comm.	Overlaps	LD-comm.	Overlaps
1	52	259	25	232	23
2	5	0	0	0	0
3	8	107	5	104	5
4	8	113	7	72	6
5	66	39	0	39	0
6	264	55	0	55	0
7	106	211	18	149	11
8	138	521	49	494	48
9	250	1532	112	1258	103
10	566	3042	162	2801	160
11	16	186	13	185	13
12	358	1617	147	1376	139
13	447	1763	135	1563	123
14	26	43	7	37	5
15	1463	5599	293	5082	268
16	466	4901	116	4015	110
17	832	3105	258	2609	248
18	293	2040	39	1913	31
19	46	336	0	327	0

Table 5.3: Results for commits with more than 5 and less than 20 files changed

Table 5.3, illustrates results for commits with more than 5 and less than 20 files changed. 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.

ID	SD	LD+comm.	Overlaps	LD-comm.	Overlaps
1	52	190	18	105	17
2	5	0	0	0	0
3	8	0	0	0	0
4	8	0	0	0	0
5	66	0	0	0	0
6	264	0	0	0	0
7	106	5561	91	5544	91
8	138	474	36	474	36
9	250	4213	159	3581	152
10	566	21214	330	19360	321
11	16	38	0	38	0
12	358	5802	152	3635	100
13	447	31266	224	30688	222
14	26	119	6	119	6
15	1463	154757	1044	148333	1028
16	466	38736	130	32767	123
17	832	29956	447	26498	418
18	293	1256784	134	1255963	124
19	46	941	7	522	3

Table 5.4: Results for commits with more than 20 files changed

Table 5.4, illustrates results for commits with more than 20 files changed. 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.

ID	% less 5	% more 5 less 20	% more 20	% Total
1	28,85	48,08	34,62	76,92
2	60,00	0,00	0,00	60,00
3	62,50	62,50	0,00	62,50
4	0,00	87,50	0,00	87,50
5	3,03	0,00	0,00	3,03
6	1,89	0,00	0,00	1,89
7	1,89	16,98	85,85	85,85
8	13,77	35,51	26,09	70,29
9	16,00	44,80	63,60	69,60
10	22,79	28,62	58,30	65,55
11	31,25	81,25	0,00	87,50
12	10,34	41,06	42,46	66,48
13	13,65	30,20	50,11	64,88
14	15,38	26,92	23,08	46,15
15	6,90	20,03	71,36	75,32
16	15,67	24,89	27,90	57,73
17	35,70	31,01	53,73	81,49
18	15,36	13,31	45,73	45,73
19	19,57	0,00	15,22	32,61
Avg	19,7	31,19	31,47	60,4

Table 5.5: Percentage rates of dependencies overlaps, reported to the structural dependencies, case with comments

Table 5.5 and 5.6 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	% less 5	% more 5 less 20	% more 20	% Total
1	23,08	44,23	32,69	71,15
2	60,00	0,00	0,00	60,00
3	62,50	62,50	0,00	62,50
4	0,00	75,00	0,00	75,00
5	3,03	0,00	0,00	3,03
6	1,89	0,00	0,00	1,89
7	1,89	10,38	85,85	85,85
8	13,77	34,78	26,09	69,57
9	15,20	41,20	60,80	68,40
10	22,26	28,27	56,71	63,96
11	31,25	81,25	0,00	87,50
12	9,50	38,83	27,93	58,38
13	12,98	27,52	49,66	63,31
14	15,38	19,23	23,08	42,31
15	5,88	18,32	70,27	74,16
16	14,59	23,61	26,39	55,58
17	34,38	29,81	50,24	78,97
18	13,31	10,58	42,32	43,00
19	19,05	0,00	7,14	23,81
Avg	18,9	28,7	29,43	57,28

Table 5.6: Percentage rate of dependencies overlaps, reported to the structural dependencies , case without comments

For the following results we have filtered the logical dependencies after the number of occurrences in the categories mentioned . Only logical dependencies that have appeared more than once will be taken into consideration. Once again the logical dependencies are splited in three categories : logical dependencies with multiple occurences found in commits with less then 5 files changed, logical dependencies with multiple occurences found in commits with more the 5 files changed and less then 20 and logical dependencies with multiple occurences found in commits with more then 20 files changed.

Table 5.7, illustrates results *for multiple occurences of logical dependencies taken into consideration as valid dependencies* for commits with less than 5 files changed. 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.

ID	SD	LD+comments	Overlaps	LD-comments	Overlaps
			Fasting Value		
1	52	21	8	17	5
2	5	1	1	1	1
3	8	32	4	32	4
4	8	3	0	3	0
5	66	0	0	0	0
6	264	4	2	4	2
7	106	3	1	3	1
8	138	13	3	13	3
9	250	70	20	61	19
10	566	714	71	699	70
11	16	15	0	15	0
12	358	33	13	27	11
13	447	124	26	100	23
14	26	2	0	2	0
15	1463	217	26	206	24
16	466	145	12	133	12
17	832	817	221	737	215
18	293	291	28	283	26
19	46	5	6	2	6

Table 5.7: Results for commits with less than 5 files changed with filtered logical dependencies

ID	SD	LD+comm.	Overlaps	LD-comm.	Overlaps
1	52	50	11	37	7
2	5	0	0	0	0
3	8	73	5	73	5
4	8	32	3	7	1
5	66	0	0	0	0
6	264	0	0	0	0
7	106	76	5	67	2
8	138	156	7	154	7
9	250	716	81	565	70
10	566	1255	84	1181	84
11	16	63	7	48	7
12	358	423	76	304	62
13	447	452	61	368	49
14	26	7	0	6	0
15	1463	1098	103	913	91
16	466	988	42	756	38
17	832	1546	188	1402	187
18	293	665	18	641	16
19	46	6	0	6	0

Table 5.8: Results for commits with more than 5 and less than 20 files changed with filtered logical dependencies

Table 5.8, illustrates results for *multiple occurences of logical dependencies taken into consideration as valid dependencies* for commits with more than 5 and less than 20 files changed. 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.

ID	SD	LD+comm.	Overlaps	LD-comm.	Overlaps
1	52	0	0	0	0
2	5	0	0	0	0
3	8	0	0	0	0
4	8	0	0	0	0
5	66	0	0	0	0
6	264	0	0	0	0
7	106	4571	89	4550	89
8	138	132	10	132	10
9	250	2001	115	1616	89
10	566	3806	134	3650	133
11	16	33	0	33	0
12	358	509	46	201	14
13	447	6814	117	6624	105
14	26	0	0	0	0
15	1463	87249	770	84242	743
16	466	18138	86	16879	83
17	832	14702	302	11773	248
18	293	922737	118	922315	97
19	46	23	5	14	2

Table 5.9: Results for commits with more than 20 files changed with filtered logical dependencies

Table 5.9, illustrates results *for multiple occurences of logical dependencies taken into consideration as valid dependencies* for commits with more than 20 files changed. 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.

ID	% less 5	% more 5 less 20	% more 20	% Total
1	15,38	21,15	0,00	46,15
2	20,00	0,00	0,00	20,00
3	50,00	62,50	0,00	62,50
4	0,00	37,50	0,00	37,50
5	0,00	0,00	0,00	0,00
6	0,76	0,00	0,00	0,76
7	0,94	4,72	83,96	84,91
8	2,17	5,07	7,25	15,22
9	8,00	32,40	46,00	58,40
10	12,54	14,84	23,67	42,05
11	0,00	43,75	0,00	62,50
12	3,63	21,23	12,85	32,68
13	5,82	13,65	26,17	43,40
14	0,00	0,00	0,00	19,23
15	1,78	7,04	52,63	58,58
16	2,58	9,01	18,45	30,69
17	26,56	22,60	36,30	64,66
18	9,56	6,14	40,27	40,27
19	13,04	0,00	10,87	23,91
Avg	9,09	15,87	18,8	39,1

Table 5.10: Percentage rates of dependencies overlaps reported to the structural dependencies with filtered logical dependencies, case with comments

Table 5.10 and 5.11 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 and multiple occurences of logical dependencies taken into consideration as valid dependencies.The 1st column shows the projects IDs; 2nd column shows the overlaping procentage between logical and structural dependencies, reported to the structural dependencies, for commits with less then 5 files changed; 3rd shows the overlaping procentage between logical and structural dependencies, reported to the 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, reported to the structural dependencies, for commits with more then 20 files changed; 5th column shows the overlaping procentage between logical and structural dependencies, reported to the 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	% less 5	% more 5 less 20	% more 20	% Total
1	9,62	13,46	0,00	34,62
2	20,00	0,00	0,00	20,00
3	50,00	62,50	0,00	62,50
4	0,00	12,50	0,00	12,50
5	0,00	0,00	0,00	0,00
6	0,76	0,00	0,00	0,76
7	0,94	1,89	83,96	83,96
8	2,17	5,07	7,25	15,22
9	7,60	28,00	35,60	52,00
10	12,37	14,84	23,50	42,05
11	0,00	43,75	0,00	62,50
12	3,07	17,32	3,91	24,86
13	5,15	10,96	23,49	41,16
14	0,00	0,00	0,00	15,38
15	1,64	6,22	50,79	56,73
16	2,58	8,15	17,81	29,18
17	25,84	22,48	29,81	60,70
18	8,87	5,46	33,11	34,47
19	14,29	0,00	4,76	21,43
Avg	8,67	13,29	16,53	35,38

Table 5.11: Percentage rates of dependencies overlaps reported to the structural dependencies with filtered logical dependencies, case without comments

ID	% less 5	% more 5 less 20	% more 20	% Total
1	25,42	9,65	9,47	9,47
2	50,00	0,00	0,00	0,00
3	9,80	4,67	0,00	0,00
4	0,00	6,19	0,00	0,00
5	40,00	0,00	0,00	0,00
6	23,81	0,00	0,00	0,00
7	7,41	8,53	1,64	1,64
8	21,35	9,40	7,59	7,59
9	16,74	7,31	3,77	3,77
10	8,19	5,33	1,56	1,56
11	7,81	6,99	0,00	0,00
12	17,05	9,09	2,62	2,62
13	9,40	7,66	0,72	0,72
14	28,57	16,28	5,04	5,04
15	9,51	5,23	0,67	0,67
16	6,94	2,37	0,34	0,34
17	19,60	8,31	1,49	1,49
18	5,19	1,91	0,01	0,01
19	9,57	0,00	0,74	0,74
Avg	16,65	5,73	1,87	1,87

Table 5.12: Percentage rates of dependencies overlaps reported to the logical dependencies, case without comments

Table 5.12 and 5.13 illustrates results in percentage, *reported to the logical dependencies*, of the analysis for all the systems when logical dependencies where build with/ without comments taken into consideration as change without filtering the number of occurrences of the logical dependencies found. The 1st column shows the projects IDs; 2nd column shows the overlaping procentage between logical and structural dependencies reported to the logical dependencies found for commits with less then 5 files changed; 3rd shows the overlaping procentage between logical and structural dependencies reported to the logical dependencies found for commits with more then 5 and less then 20 files changed; 4th column shows the overlaping procentage between logical dependencies and structural dependencies reported to the logical dependencies found for commits with more then 20 files changed; 5th column shows the overlaping procentage between logical and structural dependencies reported to the logical dependencies found 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	% less 5	% more 5 less 20	% more 20	% Total
1	24,49	9,91	16,19	11,60
2	50,00	0,00	0,00	50,00
3	9,80	4,81	0,00	4,76
4	0,00	8,33	0,00	7,32
5	40,00	0,00	0,00	4,65
6	26,32	0,00	0,00	6,76
7	7,41	7,38	1,64	1,64
8	22,62	9,72	7,59	9,78
9	17,51	8,19	4,24	4,27
10	8,37	5,71	1,66	1,76
11	8,62	7,03	0,00	5,76
12	17,00	10,10	2,75	4,41
13	9,98	7,87	0,72	0,89
14	28,57	13,51	5,04	7,75
15	8,84	5,27	0,69	0,72
16	7,25	2,74	0,38	0,72
17	20,86	9,51	1,58	2,30
18	4,53	1,62	0,01	0,01
19	8,89	0,00	0,57	1,05
Avg	16,89	5,87	2,26	6,63

Table 5.13: Percentage rates of dependencies overlaps reported to the logical dependencies, case with comments

6 | Discussion and conclusions

This chapter presents the results of the three analyses, performed on the selected projects (5.1). The purpose is to answer the three research questions outlined in Chapter 1.2.

Question 1 . How the number of source files changed in a commit can influence the logical dependencies of the system and the overlapping rates with the structural dependencies.

Based on the results presented in tables 5.5 and 5.6, from chapter Experimental Results, the number of changed files taken into consideration has an important influence over the overall rates of the logical and structural dependencies overlaps. If no threshold is set for the number of files then we can affirm that a significant number of structural dependencies are also logical. In table 5.5 we have obtained an overlap of structural and logical dependencies of 60,4% which is with 40,7% more than if consider only commits with less then 5 source code files changed per commit (Figure 6.1) and with 29,2% more if consider only commits with more then 5 and less then 20 source code files changed per commit.

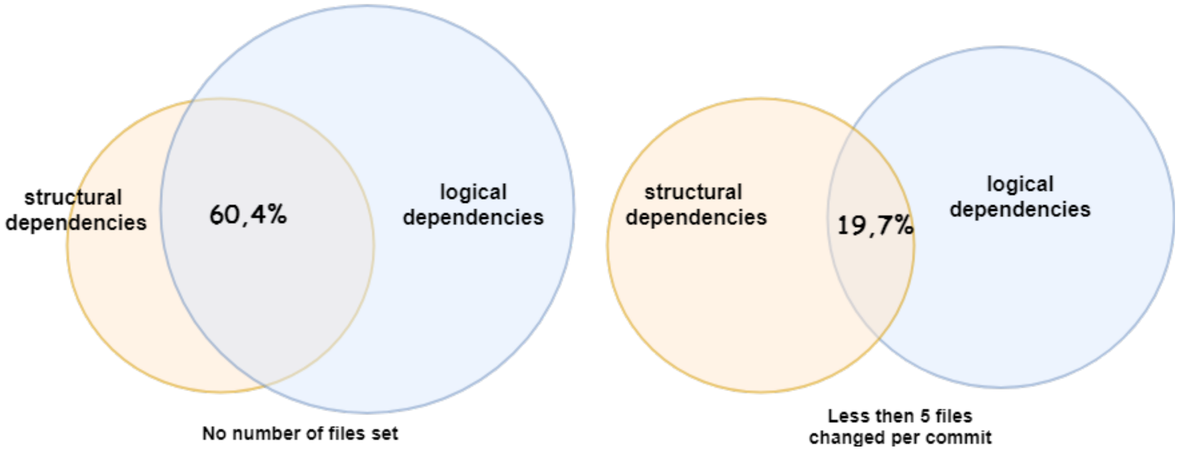


Figure 6.1: Venn diagrams of the overlapping rates with comments taken into consideration as change

Question 2. Considering comments can lead to additional logical dependencies ? How many logical dependencies are introduced by considering comment changes as valid changes and in what percentage this can influence the final result ?

Table 6.1 illustrates the percentages rates extracted from tables 5.5 and 5.6 from chapter Experimental Results. As it was specified in the tables description

Category	With comments	Without comments
less 5	19,7%	18,9%
more 5 less 20	31,19%	28,7%
more 20	31,47%	29,43%
total	60,4%	57,28%

Table 6.1: Average percentages rates with and without considering comments as valid changes.

, the percentages rates are the number of structural and logical dependencies overlaps, reported to the total number of structural dependencies. How it can be seen, the overlapping rates are also influenced by the comments filtering but in a small percentage. The rates are with aprox 3% lower if comments are not taken into consideration as a change. (Figure 6.2)

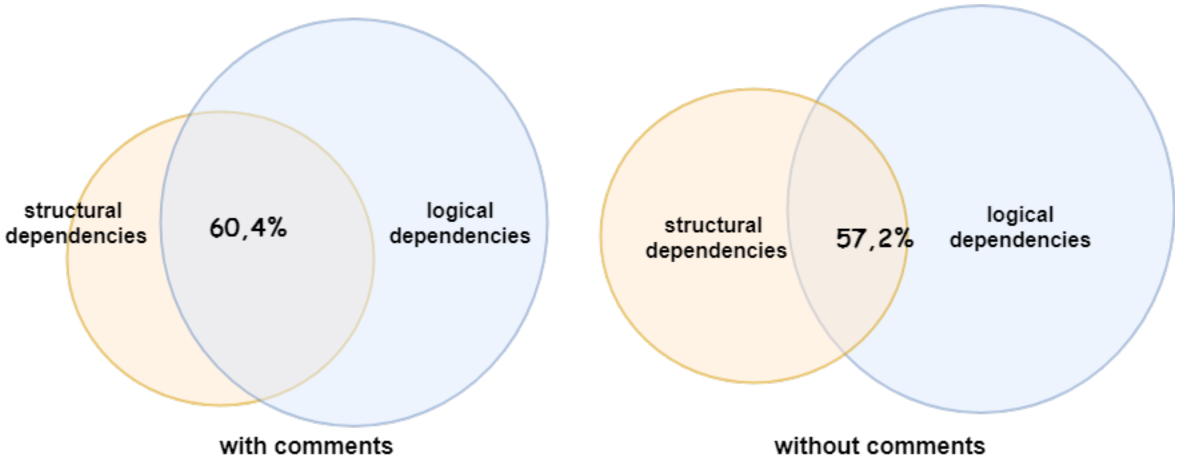


Figure 6.2: Venn diagrams of the overlapping rates with comments and without comments taken into consideration.

Question 3. One occurrence of a logical dependency is enough to consider it as valid ? If we consider only logical dependencies with more then one occurrence as valid, the results are influenced in a significant way ?

Table 5.10 and 5.11 from chapter Experimental Results, illustrates results in percentages, 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 and multiple occurences of logical dependencies taken into consideration as valid dependencies. Based on the experimental results averages 6.2 , we can affirm that the results are with approx 50% lower after filtering 6.1 . This indicates that a lot of logical dependencies are the result of a single commit in which the two elements of the dependency where changed together.
If we look at the average rates of projects 13 , 15 , 16 , 18 we can observe that the difference is much smaller (15-20%), the only thing that those projects have in

Category	With comments	Without comments
less 5	9,09%	8,67%
more 5 less 20	15,87%	13,29%
more 20	18,8%	16,52%
total	39,1%	35,38%

Table 6.2: Overall percentages rates by filtering the logical dependencies occurrences.

common is the size and the number of commits compared to the other ones from the list, the size and the number of commits of the projects mentioned above are relatively big.
This can lead to the conclusion that, if the project studied has a relatively small amount of commits (Project ID 8.), the probablilty to find multiple updates of the same classes in the same time can be small, so filtering after the number of occurences can lead to filtering all the logical dependencies extracted. (Figure 6.3)

As a conclusion, it results that large number of structural dependencies are doubled by logical or not, this number is particularly influenced by the number of files that participate in a commit that taken into consideration. It also results that taken or not comments as change, the final results are not influenced in a big percentage.

In this research work we have tried to identify methods to acquire the most relevant logical dependencies from the system so that can be used in the future for architectural reconstruction, that is currently based only on the information provided by structural dependencies. For future work, we will investigate the cause for the large number of logical dependencies which are not overlapping with structural dependencies. As we can see in tables 5.12 and 5.13, where the percentages are reported to the logical dependencies, only a small amount (aprox 10%) of logical dependencies are also structural .
In this work we have been extracted structural dependencies from the last revision of the system but logical dependencies from all the revisions of the system. We will study also structural dependencies from all the revisions of the system since some logical dependencies may have been also structural on previos 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 will probably increase.

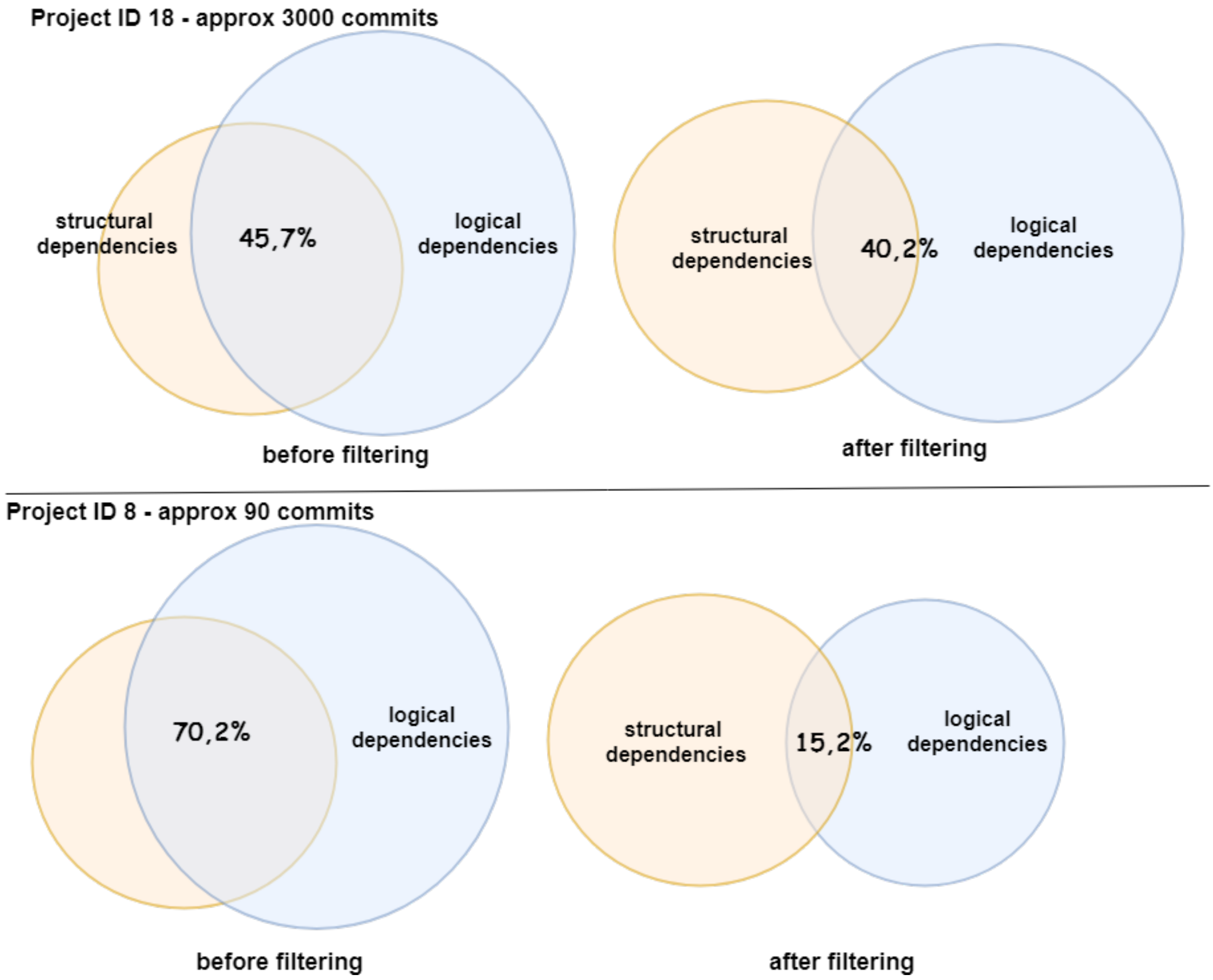


Figure 6.3: Impact of logical dependencies occurrences filtering on projects with different sizes.

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] Ajienka, N., Capiluppi, A., Counsell., *An empirical study on the interplay between semantic coupling and co-change of software classes*, S. Empir Software Eng (2018) 23: 1791.

[4] 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.

[5] Oliva, G.A., Gerosa, M.A., *On the interplay between structural and logical dependencies in open-source software.*, Software Engineering (SBES), 2011 25th Brazilian Symposium on. IEEE, pp. 144153

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

[13] Arisholm E, Briand LC, Foyen A *Dynamic coupling measurement for object-oriented software.*, 2004, IEEE Trans Softw Eng 30(8):491506.

[14] 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

[15] 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.

[16] Kuhlman Dave, *A Python Book: Beginning Python, Advanced Python, and Python Exercises.*, 2009, pp. 12-13.

[17] K. B. Bruce, *Foundations of Object-Oriented Languages.* The MIT Press, Cambridge, MA, 2002, pp. 18-20.

[18] Gilbert, Stephen, Bill McCarty. *Object-Oriented Design in Java.* The Waite Group, 1998, pp. 34-46.