

CADOCS

Conversational Agent for the Detection of Community Smells

csDetector - Modifications Report

<https://github.com/gianwario/csDetector>

Team Members

Gianmario Voria
g.voria6@studenti.unisa.it

Antonio Della Porta
a.dellaporta26@studenti.unisa.it

Reviewer

Stefano Lambiase
slambiase@unisa.it

18th July 2022

Contents

1 Context of the Project	3
2 State of the art: CSDETECTOR	3
3 Goals of the Project	3
4 CADOCS: a Conversational Agent for the Detection of Community Smells	4
5 Analysis of csDetector through reverse engineering	4
5.1 Dependency Extraction	6
5.2 Requirements extraction	7
6 CSDETECTOR explanation	7
7 Proposed changes & contributions	9
7.1 CR_1: Extracting the core functionalities and isolating the I/O process	9
7.1.1 Methodology	9
7.1.2 Expected Results	10
7.2 CR_2: Wrapping csDetector in a web service	11
7.2.1 Methodology	11
7.2.2 Expected Results	11
7.3 CR_3: Creating a conversational agent that will execute the tool	12
7.3.1 Methodology	12
7.3.2 Expected Results	12
7.4 CR_4: Improving software's robustness	13
7.4.1 Methodology	13
7.4.2 Expected Results	13
7.5 CR_5: Source code refactoring for readability concerns	14
7.5.1 Methodology	14
7.5.2 Expected Results	14
8 Impact Analysis	14
8.1 CR_1: Extracting the core functionalities and isolating the I/O process	15
8.2 CR_2: Wrapping csDetector in a web service	15
8.3 CR_3: Creating a conversational agent that will execute the tool	15
8.4 CR_4: Improving software's robustness	16
8.5 CR_5: Source code refactoring for readability concerns	17
9 Implementation of the changes	17
9.1 Analysis of the existing project	18
9.2 CR_1: Extracting the core functionalities and isolating the I/O process	18
9.2.1 CR_1.1: Changes in the output's handling	19
9.2.2 CR_1.2: Decommissioning of <i>devNetwork.py</i> by creating a <i>CsDetector</i> class	19
9.2.3 CR_1.3: Creation of an adapter to execute the tool in different ways	20
9.2.4 Results	20
9.3 CR_2: Wrapping csDetector in a web service	21
9.3.1 Results	21
9.4 CR_3: Creating a conversational agent that will execute the tool	22
9.4.1 Results	22
9.5 CR_4 Improving software's robustness	24
9.6 CR_5: Source code refactoring for readability concerns	25

1 Context of the Project

In recent Software Engineering studies, the community started to worry about the impact of human aspects in software development. Existing works analyzed how developers and sub-communities interact, with the aim of identifying communication and collaboration patterns, leading to the concept of **Community Smells**. Community Smells reflect sub-optimal organizational and socio-technical patterns in the structure of the software community. As far as we know, there is no practical way for Project Managers to detect these problems automatically. Furthermore, given the recent introduction of these concepts, a lot of managers are not aware of their existence and the importance of their detection.

2 State of the art: CSDETECTOR

CSDETECTOR is a tool that detects Community Smells automatically with a machine learning approach. Starting from the analysis of public repositories on GITHUB, it makes use of *natural language processing* to understand the intent behind developers' commit messages, pull requests, and issues. With this knowledge, the tool calculates socio-technical metrics, that have been used to train a machine learning model able to predict community smells on given repositories.

Despite the work of Almarimi et al., the documentation of the tool—CSDETECTOR—lacks essential pieces of information that would help practitioners using it. On the same line, the tool has been created for research purposes, causing it to be (1) not well designed in terms of exception and error management and (2) complex to be installed and used.

In particular, we identified CSDETECTOR's limitations in the context of the following **dependency** requirements:

- **Reliability:** The tool is hard to install and use, and for this reason the engagement of practitioners could be low. Also, exceptions and problems with the execution are not handled correctly, decreasing the trustworthiness of the system.
- **Maintainability:** We identified problems in the readability and structure of the tool that could have degraded the tool's market value. The tool's implementation does not follow any of the best-practices of an engineered approach.
- **Availability:** Since it is a command-line tool, it is not really available by practitioners, and it needs to be installed and executed by new comers through long and tedious steps.

3 Goals of the Project

To provide managers with novel knowledge about community smells, we want to give practitioners a tool able to detect them in a really fast and easy way. Such a tool will be available in environments which most of the practitioners are already familiar with, keeping the develop-

ment of an intuitive and user-friendly software our main focus. In our opinion, this will help project managers take more informed decisions during all the phases of their work.

4 CADOCS: a Conversational Agent for the Detection of Community Smells

To achieve our goal, we plan to create a **Conversational Agent** that can be used to automatically detect Community Smells. Therefore, we will make use of the above-mentioned tool, CSDETECTOR, which will be integrated as our smell detection module.

To address the tools' limitations, which we mentioned in the Section 2, and with the final goal of creating a conversational agent, we will have to perform both **maintenance** and **evolution** tasks on the tool.

We also plan to add a **novelty** in the smell detection context: while showing the user the community smells, the tool will also suggest **refactoring strategies** that could be used to fix them.

The conversational agent—named **CADOCS**—will be invoked using natural language. From a requirements point of view, CADOCS will have the following 5 *intents*:

1. Execute the CSDETECTOR tool on a given repository;
2. Execute the CSDETECTOR tool on a given repository with a specified starting date (which means it will only analyze commits made after that date);
3. Show the user a report (or send it on a given email) of the last execution, stored in the agent's context;
4. Show information about the community smells the tool can detect;
5. Show refactoring strategies for the detected smells.

CADOCS will be made available for the most used communication platform in the practitioner's field, e.g., SLACK.

5 Analysis of csDetector through reverse engineering

Before performing contribution to CSDETECTOR, we had to get a deeper knowledge of the tool's implementation. For this reason, we performed reverse engineering on the source code. The first thing we did was to extract a class diagram of the system, which can be seen in the Figure 1, even though it does not really follow a classic object oriented approach. The next step was to understand the responsibilities of each module. The execution of this process will be divided in two activities, namely (1) the extraction of dependencies among the various modules and (2) the extraction of the functional requirements that in our opinion led the designer to create the tool.

As a consequence of the absence of documentation, we only had the source code and installation guide to work with. For this reason, we will not have to worry about traceability between artifacts while performing changes.

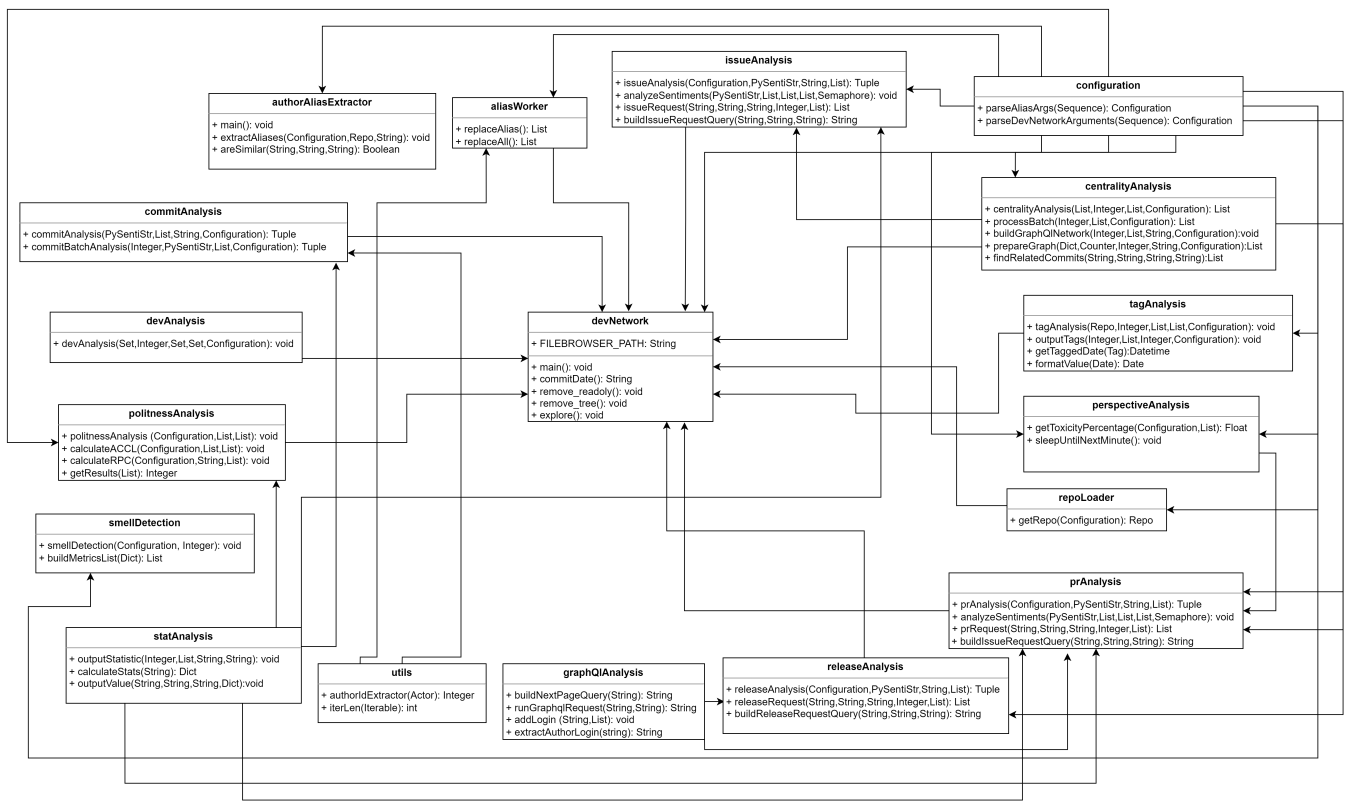


Figure 1: csDetector Class Diagram.

5.1 Dependency Extraction

The first step we had to do was the extraction of the dependencies between the modules of the system. We analyzed the source code module by module, in order to get a deep knowledge of the system's implementation. We managed to create a matrix that shows which module uses a functionality of another by adding the number of times this happens in the corresponding cell, e.g., *aliasWorker.py* uses *configuration.py* once.

Table 1 Dependency Matrix.

Module	Id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
aliasWorker	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
authorAliasExtractor	1	0	0	0	0	2	0	0	0	0	1	0	0	0	2	0	0	0	0
centralityAnalysis	2	0	0	0	0	1	0	0	0	0	0	0	5	0	3	0	0	0	0
commitAnalysis	3	0	0	0	0	1	0	0	0	0	0	0	7	0	1	0	0	0	0
configuration	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
devAnalysis	5	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
devNetwork	6	1	0	2	1	1	1	0	0	1	1	1	0	1	0	0	1	1	1
perspectiveAnalysis	7	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
politenessAnalysis	8	0	0	0	0	1	0	0	0	0	0	0	2	0	0	0	0	0	0
repoLoader	9	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
smellDetection	10	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
statsAnalysis	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
tagAnalysis	12	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0
utils	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
graphqlAnalysisHelper	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
issueAnalysis	15	0	0	1	0	1	0	0	1	0	0	0	7	0	0	3	0	0	0
prAnalysis	16	0	0	1	0	1	0	0	1	0	0	0	8	0	0	3	0	0	0
releaseAnalysis	17	0	0	0	0	2	0	0	0	0	0	0	2	0	0	2	0	0	0

After a first look at the dependency matrix, shown in the Table 1, it is clear that there are two modules among the others that have a lot of dependencies, which have been highlighted: (1) *devNetwork.py* has its row—reported in the dependency matrix with id 6—almost full of nonzero values, which means that uses a lot of other modules; (2) *configuration.py* has its column—reported in the dependency matrix with id 4—similar to the previous-mentioned module, which means that is used by almost each other part of the system.

With this process we found out that (1) the whole execution of the tool is managed by the module *devNetwork.py* and (2) *configuration.py* is responsible of keeping the state of the program during its execution (and as matter of fact, it is the only module implemented as a proper CLASS) . In particular, after getting the parameters in input, each of the logic steps performed by CSDETECTOR is executed by *devNetwork*, much like a procedural programs, using the parsed information stored in a *Configuration* object.

5.2 Requirements extraction

By looking at the source code and analyzing the execution of the tool, we were able to identify the *functional requirements* of the original tool. Even though the purpose of the system is actually one, i.e. the detection of community smells, there are actually a lot of steps involved, which may be considered as requirements.

As already stated in the first sections, CSDetector is able to detect community smells through artificial intelligence techniques. So, the first requirement would be the actual **detection** of community smells.

Starting from this, we considered the steps needed to get the prediction out of the model: gathering the input data. We studied which are the dependent variables of the model, and we identified them as various **metrics** of the project of different kind, e.g. process, developers or structural metrics. Of course, to gather them, the tool needs two things: (1) the software **repository** of the project and (2) a way to compute them. This leads us to the definition of two more requirements, which of course are (1) the management of the GitHub repository and (2) analysis of source code and sentiment behind developers messages to gather metrics.

One last requirement we identified is the management of the **persistence** of metrics: CSDetector saves these metrics as **.csv** files and **.pdf** graphs, in order to use them later for the prediction.

After this analysis, we found ourselves with the **functional requirements** shown in the Table 2

Table 2 CSDetector Functional Requirements

FR_1: Community Smell Detection	The system must be able to detect and show community smells on a given GitHub repository.
FR_2: GitHub Repository Management	The system must be able to download a specified GitHub repository and access its content.
FR_3: Project's Metrics Computation	The system must be able to gather metrics of the given software project by analyzing both the software and the sentiment of the messages of commits, issues and pull requests.
FR_4: Persistence of the metrics	The system must be able to locally write and read metrics of the project in account.

6 CSDetector explanation

To give a better understanding of the whole situation, we divided the activities of the tool in four logic steps, and for each one of these steps we extracted the responsible modules, creating the model shown in Figure 2.

Please notice that, both in this section and in the rest of the document, we will only expand the case of the regular community smell detection. The reason is that, even though CSDetector is also able to compute smells starting from a specified date, the flow is exactly the so any change that works with the normal execution also works with the starting date as input.

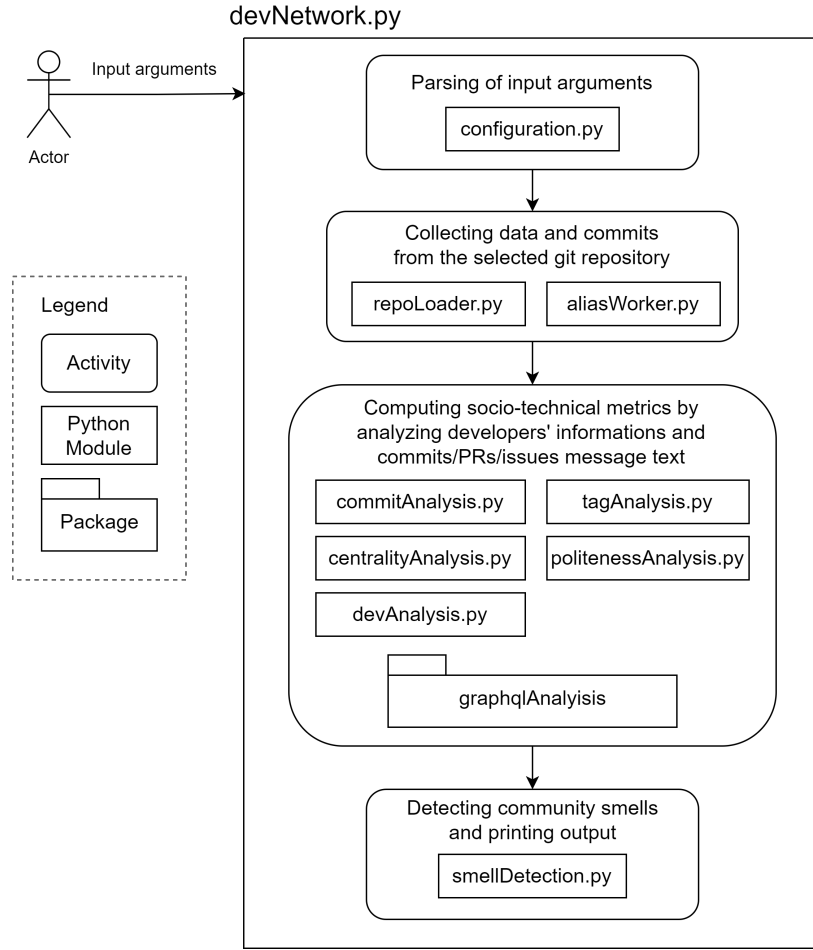


Figure 2: Summary of tool's execution handled by *devNetwork.py*.

These modules have the following responsibilities, which actually can be paired with the functional requirements identified in the Table 2:

- **Macro-module *devNetwork.py***: this may be seen, as already stated, as a wrapper for the whole execution of the tool;
- **Parsing of input arguments**: at this stage of the process, a *Configuration* object is created and initialized with the input passed as arguments by the user. It may be seen as a part of the FR_1, i.e. the way the tool gets the repository URL by the user;
- **Collecting data and commits from the selected git repository**: at this stage, the *Configuration* information about the git repository is consumed and the repository is downloaded locally, collecting everything needed. In this process the requirements FR_2 gets fulfilled;
- **Computing socio-technical metrics by analyzing developers' information and commits/PRs/issues message text**: this step actually involves two functional requirements—FR_3 and FR_4—and it is the core module which gathers data needed to perform a prediction on the pre-trained modules to detect community smells;
- **Detecting community smells and printing output**: this is the last stage of the execution, in which the system finally executes the FR_1 and detects community smells on the given repository.

7 Proposed changes & contributions

In this section, we will explain how we plan to modify and improve the CSDETECTOR tool in order to reach our goal. The CRs will be listed and performed in order of priority. The ratio behind the priorities is the need of some changes to be implemented as soon as possible, namely CR_1 (table 3) and CR_2 (table 4), in order to start working on the Conversational Agents itself, which will be the CR_3 (table 5). The remaining CRs will have less direct consequences on the agent's implementation, but since it will make use of CSDETECTOR, we need it to be as reliable and robust as possible. Each of the change requests has been thought and structured so that it will be useful for both the community and the development of CADOCS. For instance, CR_1 will be about changing the input and the output of the tool, which is a necessary step for the creation the wrapper for CSDETECTOR (CR_2). These changes will then be used by the conversational agent, that will call the web service, and by any practitioner in their private works.

7.1 CR_1: Extracting the core functionalities and isolating the I/O process

Table 3 CR_1: Extracting the core functionalities and isolating the I/O process.

Description			
Extraction of the core modules of the tool, in particular the ones dedicated to the detection of community smells, to modify (1) how parameters are taken as input and (2) how the result of the execution is shown as output, with the goal of making them usable by different clients.			
Motivation			
In the actual state, CSDETECTOR's execution relies on command line inputs, and its output is split among various files, making the understanding of valuable information difficult. It's required to make the user able to use the tool more efficiently and to get the output of an execution in a different and improved way.			
Priority			
High[X]	Medium[]	Low[]	
Effort			
High[X]	Medium[]	Low[]	
Consequences if not accepted			
The tool will continue to be difficult to interact with, and reading the output of an execution won't be an easy task.			
Has to be done after			
NA			

7.1.1 Methodology

By performing reverse engineering on the source code, we will found out which of the modules are responsible for input and output. For which concerns the input part, we plan to use the Adapter Pattern to create a class that will be able to make the tool executable in different ways. By doing so, we will be able to run CSDETECTOR not only through command line. For the output part, we are going to extract the functionality that prints detected community smells in

console, in order to (1) change what gets printed with more details (2) save information about the execution in a spreadsheet.

7.1.2 Expected Results

If the change request get accepted, there will be several improvements in the tool's usability:

- If anyone wants to use CSDETECTOR in private projects, the existence of an adapter to handle input will make the task a lot easier;
- After the execution, detected community smells will be displayed in a more comprehensive manner, making it easier to understand results;
- After an huge number of executions, the spreadsheet containing data about them can be used as a dataset to perform any kind of new research.

7.2 CR_2: Wrapping csDetector in a web service

Table 4 CR_2: Wrapping csDetector in a web service.

Description			
Creation of a wrapper of the CSDETECTOR tool, in particular a web service that will be available as an API on the web, for matters of usability.			
Motivation			
In the current state, the tool requires the user to manually install or update the needed dependencies in order to be used. Thanks to the introduction of the wrapper, any user will be able to run CSDETECTOR without any of the above-mentioned steps, by simply calling a web service. This will improve the usability and the performance of the tool.			
Priority			
High[X]	Medium[]	Low[]	
Effort			
High[X]	Medium[]	Low[]	
Consequences if not accepted			
The tool will continue to require the manual installation of the dependencies, putting a wall between a potential user (e.g. a Project Manager) and the tool itself.			
Has to be done after			
CR_1			

7.2.1 Methodology

Since we will execute the current CR after CR_1, we will have the tool ready to use by calling an adapter. This adapter will expose an interface requiring simple strings representing parameters needed to execute CSDETECTOR. We will create a web service, which can be seen as a wrapper for the original tool, that will be available through APIs built on the adapter. By calling an API with the right parameters, the tool will be executed and results will be sent back as a JSON response. This web service will be hosted online, and so its APIs will be accessible for anyone both to call them and to use them in new applications. Also, to give more insight about the tool's predictions, we will also give the possibility of getting some of the socio-technical metrics graphs computed by CSDETECTOR by using a parameter in the query.

7.2.2 Expected Results

If the change request get accepted, there will be several improvements in the tool's availability:

- If anyone wants to integrate CSDETECTOR on a web application a simple API call to our web service will do the job;
- Since an HTTP GET request can be executed in a lot of different ways, the tool will be accessible even without applications.

7.3 CR_3: Creating a conversational agent that will execute the tool

Table 5 CR_3: Creating a conversational agent that will execute the tool.

Description		
Implementation of a conversational agent, based on natural language, that will then be available on the most famous platforms.		
Motivation		
Even with the introduction of a wrapper, proposed in the CR_2, the usage of the tool will remain enclosed in a small circle of practitioners. As explained by the 1 st Lehman's law, <i>continuous change</i> is required to keep satisfying users' needs. For this reason, following technology and literature trends, a conversational agent will be developed and published. Thanks to a natural language processing approach, that will make it easier to use, it will gather new users and increase the tool's value.		
Priority		
High[X]	Medium[]	Low[]
Effort		
High[X]	Medium[]	Low[]
Consequences if not accepted		
The tool will slowly become less satisfying for practitioners until it will become unused.		
Has to be done after		
CR_2		

7.3.1 Methodology

Having to be executed after the CR_2, we will have access to the endpoints that exposes the CSDETECTOR functionalities. So we want to create a Conversational Agent that can receive messages from the user, understand the intent of the request using an NLP model, call the endpoint that implements that functionality and display the results of the request to the user on the chat. For this purpose we are going to use the well known messaging app Slack and create a Slack App to interact with the user.

7.3.2 Expected Results

If this change request will be accepted, there will be the following improvement to the tool usability:

- Any practitioner will be able with little to no effort to use detect community smells by querying a conversational agent through natural language.

7.4 CR_4: Improving software's robustness

Table 6 CR_4: Improving software's robustness.

Description		
For quality management purposes, it is necessary the refactoring of some methods to correctly manage the unhandled exceptions raised, and it is required to fix some dependencies which are not working in their current version.		
Motivation		
After a first analysis of the source code, we found out that in some parts of the software some exceptions were raised but non managed correctly. The result of this behavior is that the software does not detect some possible fatal defects. Also, some of the libraries required to make it work, which are listed in the tool's documentation, are wrong and may cause crashes.		
Priority		
High[X]	Medium[]	Low[]
Effort		
High[]	Medium[X]	Low[]
Consequences if not accepted		
The tool will continue the execution in some parts, making it difficult for the user to know why and where the software has encountered a failure.		
Has to be done after		
NA		

7.4.1 Methodology

In this CR the aim is to make the software more robust. In this regards, we will perform a deep analysis of the code and analyze all the exception that are raised and not correctly managed. Secondly we will analyze the correctness of the listed dependency in the regard of the system

7.4.2 Expected Results

If this change request will be accepted, there will be the following improvement to the tool robustness:

- User's will have direct feedback on where an error has occurred;
- There will be less crashes for unknown reasons.

7.5 CR_5: Source code refactoring for readability concerns

Table 7 CR_5: Source code refactoring for readability concerns

Description		
Refactoring of the code with the goal of improving its readability and maintainability.		
Motivation		
Reading and consequentially maintaining the code is very challenging at the moment, given the existence of some design issues and code smells. As it has been built for academic purposes, CSDETECTOR is used only to prove a concept rather than effectively being applied in a real-world scenario, in which the software will need to evolve and be maintained to satisfy business changes. It's so required a refactor to improve this aspect of the code, in order to make future change requests easier.		
Priority		
High[]	Medium[X]	Low[]
Effort		
High[]	Medium[X]	Low[]
Consequences if not accepted		
The tool will be difficult to maintain, resulting in a faster decline in its value.		
Has to be done after		
NA		

7.5.1 Methodology

In this CR the aim is to increase the software readability and maintainability. To perform this task we will use some tools that detect the standard of a clean code and refactor where it is necessary.

7.5.2 Expected Results

If this change request will be accepted, there will be the following improvement to the tool:

- The standard PEP8 of the tool will be followed, improving the readability of the source code and the possibility of future changes

8 Impact Analysis

Since we had to study if our changes could have a side effect on the rest of the system, we analyzed the impact of each alteration by identifying the set of modules that could be affected. We started by identifying which functional requirement was involved in the change, so that we could make a mapping between the functionality and the module used. Also, we gathered additional information by using (1) the dependency matrix and (2) the source code. With such a knowledge, we are going to be able to perform maintenance with no worries about introducing faults.

8.1 CR_1: Extracting the core functionalities and isolating the I/O process

For which concerns the first change request, we excluded from the beginning the impact on FR_2, FR_3 and FR_4, since it has nothing to do with the actual computation of metrics and the detection process. So, we were able to identify the following **Starting Impact Set** by checking the modules that were actually affected by the FR_1, keeping as a reference the pairing among activities and requirements shown in the Section 6:

- *devNetwork.py*, since it is the main module invoked when the user runs the tool;
- *configuration.py*, since it is the module invoked to parse input arguments;
- *smellDetection.py*, since it is both responsible of detecting community smells and printing them in console.

In order to identify the **Candidate Impact Set**, we analyzed the dependency matrix to look for possible direct or indirect effects caused by the module involved in the SIS. Since *devNetwork.py* only calls the various modules one after another, but the changes only refer to the FR_1, none of the dependent modules are actually considered. For which concerns *configuration.py*, we considered the actual implementation plans we had for it. The changes may be isolated locally in the module since they only affect how data are *gathered* and not how they are *stored*. This means that any other modules that depends on *configuration.py* is not affected.

Given these considerations, the CIS contains the same modules of the SIS.

The results of the impact analysis for the CR_1, in terms of discovered SIS and CIS are shown in the Table 8, while the *Actual Impact Set (AIS)* will be defined after the implementation.

Table 8 Impact sets for CR_1.

Module	Starting Impact Set	Candidate Impact Set	Actual Impact Set
devNetwork	X	X	
configuration	X	X	
smellDetection	X	X	

8.2 CR_2: Wrapping csDetector in a web service

After the implementation of CR_1, the tool can be used without the necessity of the command line execution. For the concerns of the CR_2 instead, the focus is on improving the usability of the tool's specific functionalities with the usage of a modern Web Services approach. Our purpose will be achieved by introducing a new module responsible of serving an API to execute the tool. In particular, the new function will execute the tool by using the Adapter introduced in the previous CR. For this reason, the introduction of the web service itself will have no impact on the previously-existing modules.

8.3 CR_3: Creating a conversational agent that will execute the tool

As a consequence of the realization of CR_2, CSDETECTOR will make available the feature also with the web services, making it easier for developers to integrate them within their projects. In order to further improve the system, we will use the abstraction layer of the web services

to integrate the CSDETECTOR features into a *Conversational Agent*—named **CADOCS**—without impacting any of the modules of the system.

8.4 CR_4: Improving software's robustness

For this change request we have searched in the source code the usage of external resources to find some potential flaws. We found that one of the functionalities at the core of the tool, the download of a git repository and are also used some GraphQL query for the retrieve of repository information. These two resources can be not accessible for some not user-dependent fault (e.g. the server hosting the repository is not online or the user has reached the maximum amount of GraphQL query¹). These fault are not correctly managed by the software and the user is not warned correctly. Following this, with the first run of the tool we discovered that two of the tool's dependency, *Spacy 3.2* and *Convokit 2.4.4* were incompatible and made the tool unusable in the current state so it is necessary to overcome this issue acting on the versions. At the end we looked at the way the tool logs information and find out that it is done everywhere with the *print* statement instead of using a proper logger. It is necessary to implement a much sophisticated logging method to help the developer to debug the software to lower the effort needed of finding fault. By using the well known **logging** library inside Python this activity will not add any external dependency.

Table 9 Impact sets for CR_4.

Module	Starting Impact Set	Candidate Impact Set	Actual Impact Set
devNetwork	X	X	
repoLoader	X	X	
aliasWorker	X	X	
authorAliasExtractor	X	X	
centralityAnalysis	X	X	
commitAnalysis	X	X	
devAnalysis	X	X	
perspectiveAnalysis	X	X	
politenessAnalysis	X	X	
smellDetection	X	X	
statsAnalysis	X	X	
tagAnalysis	X	X	
graphqlAnalysisHelper	X	X	
issueAnalysis	X	X	
prAnalysis	X	X	
releaseAnalysis	X	X	

¹For references on the maximum GraphQL calls see the documentation at <https://docs.github.com/en/graphql/overview/resource-limitations>

8.5 CR_5: Source code refactoring for readability concerns

In the regard of this change request, we put a major focus on maintaining the existing tool on an high-quality status. The Starting Impact Set is composed of all the modules of the system because yet again we have to potentially operate on the entire system. As it has been built for proving a concept, rather than being an actual product, CSDETECTOR has some design issues to overcome in order to be commercially viable. So we began inspecting the code to search any possible improvement on readability.

- There is a commented code block inside the *perspectiveAnalysis.py* module that needs to be removed because it is very similar to the actual module code.
- Inside *devNetwork.py* there is a piece of code that isn't concerned in the detection of community smells but it is only about validating pre-requisites of the installation.
- The naming convention of all the function in the code is not compliant with the Python standard. In the code it is used the Camel Case convention but, standing to the PEP8 Convention², Python requires the Snake Case convention.
- The spacing in the functions is not compliant to the PEP8 Convention

Table 10 Impact sets for CR_5.

Module	Starting Impact Set	Candidate Impact Set	Actual Impact Set
devNetwork	X	X	
repoLoader	X	X	
aliasWorker	X	X	
authorAliasExtractor	X	X	
centralityAnalysis	X	X	
commitAnalysis	X	X	
devAnalysis	X	X	
perspectiveAnalysis	X	X	
politenessAnalysis	X	X	
smellDetection	X	X	
statsAnalysis	X	X	
tagAnalysis	X	X	
graphqlAnalysisHelper	X	X	
issueAnalysis	X	X	
prAnalysis	X	X	
releaseAnalysis	X	X	

9 Implementation of the changes

After analysing the possible impacts of the changes on the system, we forked the original repository on GITHUB in order to perform our contributions. In this section we will give a quick summary of the technologies used in the project and how they work, since the understanding

²More information here: <https://peps.python.org/pep-0008/>

of code patterns and programming language has played an huge role in the process of maintenance and evolution, and then we will describe in detail the implementation of changes and their result.

9.1 Analysis of the existing project

The original project—`csDetector`—has been developed with an high level programming language named PYTHON. Despite it's object-oriented capabilities, it is mostly known and used for scripting and MACHINE LEARNING activities.

In our case, the whole project is based on a set of independent scripts, and each of them makes use of the *Configuration.py* class, which is the only class implemented by the creators of the tool. This class has the responsibility of parsing input arguments through a package named *argparse*, that is able to define required command line input arguments and parse them into strings. The parsed arguments will then be stored in the object's state and be accessible—passing it through function arguments—in each module during the whole execution. In the case of our tool, the parsed arguments were:

- -p: GitHub PAT—Personal Access Token—used to query the repository;
- -r: Repository URL;
- -s: Directory containing *Sentistrength*, a library used for sentiment analysis;
- -o: Directory that will contain the output;
- -sd(Optional): Starting date from which commit have to be analyzed.

For which concerns the analysis of socio-technical metrics to detect community smells, the tool runs the following macro-steps, in order:

- Mining of data about developers in the repository, saving them as *.csv* files in the output's directory;
- Iterative computation of different metrics by reading information previously saved in files locally, saving them as well;
- Detection of community smells based on previously computed metrics saved in files locally.

Since we plan to make a well engineered process, we will adopt an object-oriented methodology. In Python, the `__init__` method is the constructor of the class, and the arguments of the function will then be used as attributes. The inheritance is handled by adding the parent's class in parenthesis after the class definition.

Also, we will make sure that the standard Python convention for code style is used, for instance we will follow the PEP 8 – Style Guide for Python Code set of convention written by Python developers.

9.2 CR_1: Extracting the core functionalities and isolating the I/O process

Since this change request is made of more than one logically independent parts, we decided to split them into subsections, just like we did with the actual implementation.

9.2.1 CR_1.1: Changes in the output's handling

The first step made in the development of the first change request has been the refactoring of the output. After detecting community smells, the *smellDetection.py* module printed the output in console as a list of acronyms, without returning anything to the main file, *devNetwork.py*. We actually made so that, after detecting the smells, the responsible function would return its results back to the one who called it. In this way we were able to have full control on the output, and as a consequence we managed to change how it is printed in console, having the following results:

```
# previous output
['2021-11-22', 'SV', 'SD', 'UI']

# new output
{'Index': 0, 'StartingDate': '12/03/2018', 'Smell1': ['SV', 'Sharing Villainy'],
 'Smell2': ['SD', 'Solution Defiance'], 'Smell3': ['UI', 'Unhealthy Interaction']}
```

In addition to this, we also saved the execution's results in a spreadsheet that could be used in the future as a dataset.

9.2.2 CR_1.2: Decommissioning of *devNetwork.py* by creating a *CsDetector* class

Of course the main part of this CR was about introducing a new way of executing the tool. The first thing we realized was a new class, named *CsDetector*, which exposes a public method named *executeTool*. The method takes as input the command line arguments used in the normal executions and passes these data to *devNetwork.py*, so that it can continue its normal execution. The main goal we achieved with this addition has been the fact that we are no longer calling *devNetwork* as a common python function, but instead we are passing through an object with a public method.

In this process, we also had to modify *configuration.py*. We found out that the creator of the tool made a small mistake during the parsing of the arguments: since he was passing a variable named *args* to the method in *Configuration* that had the responsibility of reading arguments, we thought that we only needed to simulate that string to make the tool executable. Instead, he was parsing arguments from an environment variable that stored system arguments, and so he didn't really make any use of *args*. To fix this, we only had to add the above-mentioned variable—containing our arguments to parse—in the actual parser function.

From the user's perspective, this tiny change is only visible when running the tool through command line. In particular, the name of the module containing the name is now "*csDetector.py*", which in our opinion adds some clarity to the process.

```
# previous command line execution
> .\devNetwork.py -p "github_PAT" -r "https://github.com/repourl"
-s "./sentistrength" -o "./output"

# new command line execution
> .\csDetector.py -p "github_PAT" -r "https://github.com/repourl"
```

```
-s "./sentistrength" -o "./output"
```

The real impact of this change is from developers' point of view: since there is now a class responsible of the execution, we no longer have to call a simple script, and instead from our main class we can run the tool through an object, as shown below. In this way, anyone in the future could integrate CSDETECTOR by passing arguments to the method of the new class, making the whole usage of the tool more comprehensive.

```
# command line arguments not parsed yet
inputData = sys.argv[1:]
tool = CsDetector()
tool.executeTool(inputData)
```

In this process no already existing modules were modified.

9.2.3 CR_1.3: Creation of an adapter to execute the tool in different ways

After the previously-mentioned changes, we had all the necessary instruments to create an Adapter for the execution. Since we already had the class CsDetector, we were able to introduce it by simply creating a new class—CsDetectorAdapter—that inherits the original class and overrides the method *executeTool*. The new method takes as input each single argument used in the original function, and calls the tool by simulating command line arguments when executing the parent's original method. In this way, we no longer need to pass arguments through command line, and so we can execute the tool in a new and improved manner:

```
# command line execution
> .\csDetector.py -p "github_PAT" -r "https://github.com/repourl"
  -s "./sentistrength" -o "./output"

# adapter execution (with a main file that simulates arguments)
> .\csDetectorAdapter.py
```

These changes will be crucial for the implementation of the following change requests since they will use the tool through the Adapter.

9.2.4 Results

The implementation of this first change request has been a success since everything works as expected. We now find ourselves with all the necessary tools to execute future change request, and in the process we also evolved CSDETECTOR in terms of usability and understanding.

For which concerns the impact on the existing modules, we found that modules discovered during the impact analysis were correct.

The updated impact sets are shown in the Table 11.

Table 11 Actual impact set for CR_1 after implementation.

Module	Candidate Impact Set	Discovered Impact Set	Actual Impact Set
devNetwork	X		X
configuration	X		X
smellDetection	X		X

The following metrics about the impact analysis were extracted:

$$\mathbf{Recall} = \frac{|CIS \cap AIS|}{|AIS|} = \frac{3}{3} = 100\%$$

$$\mathbf{Precision} = \frac{|CIS \cap AIS|}{|CIS|} = \frac{3}{3} = 100\%$$

9.3 CR_2: Wrapping csDetector in a web service

In the context of this CR, we planned to create a web service able to make the tool executable through an HTTP call. In the end, we managed to do so pretty easily.

We implemented a new module, named *csDetectorWebService.py*. By using Flask, we created a route through which an user can query the tool.

Whenever this endpoint gets called—with the GET method—the Adapter of the tool implemented in the CR_1 is created and executed with the GET query parameters, which should be the same as the normal execution of the tool.

If a parameter *graphs* is specified and it is *True*, the tool also answers with an array of paths. These paths may be used to query another endpoint which is able to send back the files stored in the paths specified, which are the socio-technical graphs computed by CSDetector.

Some examples of GET calls are:

```
# simple csDetector execution
http://URL:PORT/getSmells?repo=REPOSITORY_URL&pat=GIT_PAT

# csDetector execution with date and graphs
http://URL:PORT/getSmells?repo=REPOSITORY_URL&pat=GIT_PAT&
    date=STARTING_DATE&graphs=True

# retrieval of graphs given as output by the previous call
http://URL:PORT/uploads/GRAPH_FILENAME
```

9.3.1 Results

The web service works as expected and it is, by himself, a valid solution to use CSDetector without having to install it.

In the process of implementing this web service, no module of the original tool were affected, and since the impact sets shown in the previous sections were empty, no metrics will be computed.

9.4 CR_3: Creating a conversational agent that will execute the tool

In the end, we managed to create a conversational agent—named **CADOCS**—which has been made available on the SLACK platform.

As shown in Figure 3, we divided the tool into three separable and modular modules described below.

Tools Wrapper In its initial version, CADOCS was designed to wrap CSDETECTOR. This means that our tool currently makes available the functionalities provided by CSDETECTOR, both in terms of socio-technical metrics measurements and community smell detection. With this implementation, we addressed some limitations of CSDETECTOR, such as installation process and usability.. It is worth pointing out that our implementation was made on the basis of a *Strategy* design pattern that enables the integration of additional community smell detectors and the implementation of strategies able to handle more sophisticated combinations of these detectors.

Machine Learning System The machine learner behind the tool had the goal of easing the interaction between users and system's logic. Such an interaction was thought in terms of users' intents, the goals the user has in mind when questioning the bot, and were implemented by means of natural language processing (NLP) and natural language understanding (NLU). In particular, the machine learner was trained through a survey study that we performed to understand how stakeholders of the tool—researchers, practitioners, and students—would use to ask the bot about community smells and their refactoring. We distributed the survey to 9,207 stakeholders identified on SURVEYCIRCLE. In addition, we provided the module with an *active learning* mechanism to ensure the bot's increasing ability to recognize users' intentions and improve its usability over time. Specifically, if the bot's confidence—the degree of certainty about the user's intent—stands below a threshold of 0.7, the tool asks the user to disambiguate the intent, hence leading to request an update of the training data. This process has a double advantage: (1) the user is more engaged and (2) we gather new training data.

Conversational Agent This is the core part of the bot, containing the logic to interact with the user. It interacts with the *Slack workspaces* and has been implemented using the *Slack Events API*. The module also contains the logic to suggest refactoring strategies. It works by having an API end-point able to catch events happening in the *Slack workspace* authenticated through personal tokens. Whenever CADOCS detects a new message, written in any of the channels in which it has been added, the whole process starts. After having computed the right message, basing on the intent detected by the ML model, it posts the answer in the channel from which the message was received.

An example of the execution of CADOCS is shown in the Figure 4

9.4.1 Results

In the process of implementing the conversational agent, no module of the original tool were affected, and since the impact sets shown in the previous sections were empty, no metrics will be computed.

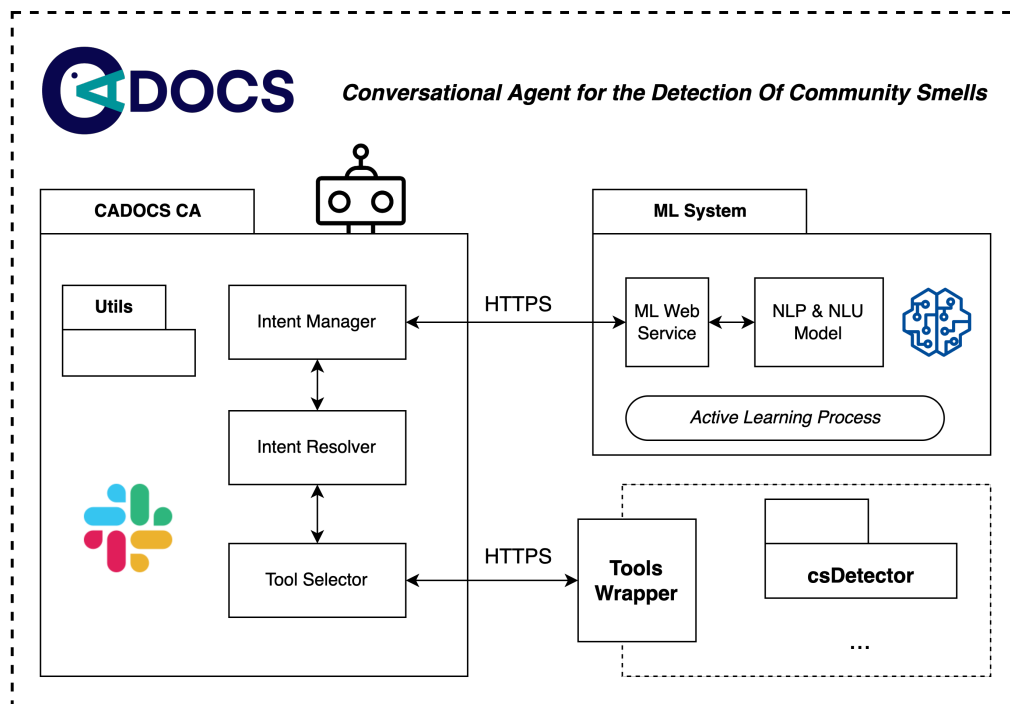




Figure 3: CADOCS's architecture.

 **GIANMARIO VORIA** 16:40
hi cadocs, can you give me community smells in <https://github.com/microsoft/QuantumKatas>

 **CADOCS** APP 16:41
Hi GIANMARIO 🙌

This is the community smells we were able to detect in the repository
<https://github.com/microsoft/QuantumKatas> :

OSE Organizational Silo Effect 🧑🧑🧑
Siloed areas of the community that do not communicate, except through one or two of their respective members.

BCE Black-cloud Effect 🌑🌑🌑
Information overload due to lack of structured communications or cooperation governance.

Some possible mitigation strategies are:

- Create communication plan ★★★★★
- Restructure the community ★★★
- Introduce a Social sanctioning mechanism ★★★

Figure 4: Example of CADOCS's execution.

9.5 CR_4 Improving software's robustness

In the context of this CR, the main goal was the improvement of the overall robustness of the tool. As we already know, CSDETECTOR has the final goal of analyzing a GitHub repository and output the Community Smells found on it. This elaboration is achieved by downloading the repository using the GraphQL API that requires a Github PAT, Personal Access Token, that is connected to the user's GitHub account. The number of repository that we can download using a single PAT is not endless, in fact when the limit of access to GitHub is reached, the API call returns the HTTP code *401* and the tool crashes exposing a generic error to the user. To enhance the bot robustness we decided to catch the exception raised in the case of reaching the PAT limit and give to the user an hint of what could have happened by linking the documentation of GitHub on this aspect.

Other than this, the tool has others exceptions not correctly managed that, when raised, results in a deep stacktrace of the error that could lead in confusion the user, so we decided to catch the exceptions and give to the user only valuable informations that could explain the root of the error.

In the actual state of the tool, the user is informed of the tool's progress using some *print* statements. The problem here is that in a generic execution of the tool, we do not always want to visualize all that information or viceversa, we want some more informations on the tool's progress and state. So we decided to improve the explainability of the tool replacing the previous *print* statement with *log* statement. So we introduced a custom logging class, called *cadocsLogger* that uses the *logging* module integrated in python. We decided to output to the user the information in this format:

```
%(asctime)s - %(name)s - %(levelname)s - %(message)s
```

where *asctime* refers to the time expressed in the STD format, *name* refers to the name of the module in which we are logging, *levelname* the level of severity of the message and *message* the actual message.

Using this approach, the user will be able to select the level of severity of the log and so reduce or increase the information he receive, and also be informed on which modules are being executed in a specified time frame.

The updated impact sets are shown in Table 12

Table 12 Impact sets for CR_4.

Module	Candidate Impact Set	Discovered Impact Set	Actual Impact Set
devNetwork	X		X
repoLoader	X		X
aliasWorker	X		X
authorAliasExtractor	X		X
centralityAnalysis	X		X
commitAnalysis	X		X
devAnalysis	X		X
perspectiveAnalysis	X		X
politenessAnalysis	X		X
smellDetection	X		X
statsAnalysis	X		X
tagAnalysis	X		X
graphqlAnalysisHelper	X		X
issueAnalysis	X		X
prAnalysis	X		X
releaseAnalysis	X		X

The following metrics about the impact analysis were extracted:

$$\mathbf{Recall} = \frac{|CIS \cap AIS|}{|AIS|} = \frac{16}{16} = 100\%$$

$$\mathbf{Precision} = \frac{|CIS \cap AIS|}{|CIS|} = \frac{16}{16} = 100\%$$

9.6 CR_5: Source code refactoring for readability concerns

As the name suggest, this CR was mainly focused on a general improvement of readability of the source code. As it has been built for academic purposes, CSDetector is used only to prove a concept rather than effectively being applied in a real-world scenario, in which the software will need to evolve and be maintained to satisfy business changes. So we began removing all the code pieces that were commented and not used anymore, to polish the code. Then, looking at the devNetwork.py module, we noticed that the initial part was not relative to the detection of the community smells, but rather the validation of the pre-requisite of the tool installation. So we decided to extract that part in a separate validate method, only responsible for that specific action that in case of errors in the tool installation raises exception and so terminate the tool execution, otherwise if the check were all passed the execution continues normally. In the end we decided to make the source code compliant to the PEP8 Convention adopted by python users, in order to make the code more readable. For this task, we've used a module called *autopep8* that refactored all of our classes correcting all the wrong tabs and spaces.

We decided to go further on the readability task and decided to refactor the code in order to follow another convention of the Python users that states that every variable and function

name should be written in a *snake_case* type rather than the *camelCase* used in csDetector.

We have done this changes because we strongly believe that a much readable and comprehensible code will facilitate any future enhancement on the tool.

The updated impact sets are shown in Table 13

Table 13 Impact sets for CR_5.

Module	Candidate Impact Set	Discovered Impact Set	Actual Impact Set
devNetwork	X		X
repoLoader	X		X
aliasWorker	X		X
authorAliasExtractor	X		X
centralityAnalysis	X		X
commitAnalysis	X		X
devAnalysis	X		X
perspectiveAnalysis	X		X
politenessAnalysis	X		X
smellDetection	X		X
statsAnalysis	X		X
tagAnalysis	X		X
graphqlAnalysisHelper	X		X
issueAnalysis	X		X
prAnalysis	X		X
releaseAnalysis	X		X

The following metrics about the impact analysis were extracted:

$$\mathbf{Recall} = \frac{|CIS \cap AIS|}{|AIS|} = \frac{16}{16} = 100\%$$

$$\mathbf{Precision} = \frac{|CIS \cap AIS|}{|CIS|} = \frac{16}{16} = 100\%$$

10 Conclusions

To summarize, we thought that CADOCS could be a first step in the diffusion—also in the practitioners’ world—of community smells like a standard to characterize and measure social patterns in software development communities in order to improve the software development lifecycle. We managed to do it by evolving CSDETECTOR.

We preliminarily assessed the capabilities of the tool, yet we are aware that more empirical experimentations would be required. We plan to conduct those experimentations as part of our future research agenda. We also plan to keep evolving the tool’s features, supporting more refactoring recommendations and integrating more community smell detection tools.

For which concerns the CSDETECTOR tool, we made so that whoever wants to re-implement its smell detection in a new project can easily interact with our Adapter. We improved the system both in its business value, implementing new ways of interacting with it, and in its technical implementation, addressing some of the limitations of the original version.