NaturalApi

Developer manual

Version	1.0.0
Approval	Edoardo Tinto
Drafting	Matteo Infantino
	Simone Innocente
	Andrea Polo
Check	Matteo Infantino
	Edoardo Tinto
State	Approved
Use	Esterno
Addressed to	FourCats
	TealBlue
	Prof. Tullio Vardanega
	Prof. Riccardo Cardin
Email	fourcats.unipd@gmail.com

Description

Developer manual inherent to the NaturalAPI product of the FourCats group, useful for the maintenance and extension of the product

Version	Date	Name	Role	Description	Checker	Outcome
1.0.0	11-05-2020	Edoardo Tinto	Programmer	Updated § 3 and 5.2	Andrea Polo	Positive
	05-05-2020	Andrea Polo	Programmer	Updated § 4 and 5.1	Edoardo Tinto	Positive
	03-05-2020	Simone Innocente	Programmer	General check	Francesco Battistella	Positive
0.9.0	10-04-2020	Matteo Infantino	Programmer	Updated § 5 and 6	Edoardo Tinto	Positive
	09-04-2020	Andrea Polo	Programmer	Drafting § 2, 3 and 4	Matteo Infantino	Positive
	07-04-2020	Simone Innocente	Programmer	Drafting § 6	Matteo Infantino	Positive
	07-04-2020	Andrea Polo	Programmer	Drafting § 5	Matteo Infantino	Positive
	06-04-2020	Simone Innocente	Programmer	Drafting § 2,3 and 4	Matteo Infantino	Positive

Indice

1 Introduction

1.1 Preamble

The current document refers to version 1.0.0 of the product *NaturalAPI*.

1.2 Document purpose

The purpose of this document is to define in detail the components of the software product *NaturalAPI* to allow its maintenance and/or extension.
The product is composed of three distinct modules for which the adopted architecture is the same. This will be illustrated and then proceed with an in-depth analysis of the components of each module. First of all, the installation methods of the components, frameworks and libraries necessary for the correct functioning of the product will be discussed.

1.3 Product purpuse

The purpose of the product is the creation of a toolkit, *NaturalAPI*, able to automatically generate application programming interfaces (APIs) and related unit tests for a given programming language, starting from features and scenarios in *Gherkin* format and from text documents related to the domain of interest.

1.4 Glossary

At the end of this document there is the appendix A where you can find definitions of terms' definition that may be ambiguous or new for *NaturalAPI* users. These terms are identified by the use of italics.

2 Technologies

The technologies used by this product are listed below. The technologies are divided into two groups:

- Development tools, including programming languages, IDE and other Items configurations[1];
- External libraries, including frameworks and libraries used in the toolkit.

2.1 Development tools

2.1.1 Java

The programming language chosen to develop NaturalAPI is Java.
This programming language, thanks to the Java Virtual Machine (JVM) is independent of the hardware execution platform. The portability of this programming language led us to its choice.

2.1.2 IDE

The IDE used for product development is IntelliJ IDEA (version 2019.3 or higher), in this document we will refer exclusively to this editor referring to IDE.
IntelliJ IDEA allows the addition of plugins. We decided to use the CodeMR plugin as a static code analysis tool. Its configuration follows the standard procedure of adding a plugin in the IDE.

2.1.3 Git and GitFlow

We use Git and GitFlow (distributed VCS) software to manage the versioning. See the Installation section of this document for local repository cloning and configuration procedures.

2.1.4 Maven

Build automation software used to manage the build process and dependencies. The version in use is 4.0.

2.2 External libraries

The management of dependencies to external libraries is managed through the Maven software.

2.2.1 Stanford Parser

Library containing natural language programming functions developed by Stanford University. This library is open source.

2.2.2 Jackson

Library that provides json file management functionality in Java. The version in use is 2.10.0.

2.2.3 Swing

Official library for the creation of graphical user interfaces in Java.

[1]See Norme di progett v1.0.0, section 3.2: Configuration Management.

3 Prerequisites

3.1 Software Requirements

- Java Development Kit (jdk) 1.8 or higher;
- IntelliJ IDEA 2019.3 or higher
- Git (GitFlow optional);
- Maven.

3.2 System Requirements

- Windows: 10/8/7/Vista/2003/XP — MacOS X: 10.5 — Linux: GNOME or KDE desktop;
- 2 GB RAM minimum, 4 GB RAM recommended;
- 700 MB hard disk space;
- Minimum screen resolution of 1024x768;
- Internet connection to download the required application and software;
- Browser web.

3.2.1 Notes on the development environment

This software is developed under Windows and MacOS, in particular the product has been tested under Windows 10 Home and MacOS Catalina operating systems. We would like to specify that the toolkit is cross-platform and therefore no specific technologies have been and will not be used for a specific operating system.

4 Installation

4.1 Git and GitFlow

4.1.1 Git installation

First thing to do is install Git. Go to `https://git-scm.com/` and download the latest version available for your operating system.
Second, proceed with the installation. If you need help, please consult the documentation at the following link: `https://git-scm.com/docs`.

4.1.2 Local repository creation

To clone the repository on your machine open the terminal and type the following command:

- *git clone https://github.com/fourcatsteam/NaturalAPI.git*

4.1.3 GitFlow installation

To install GitFlow follow the directions at the following link github.com/nvie/gitflow. Once the installation is complete follow the information proposed in the next section.

4.1.4 GitFlow configuration

Access the local repository via terminal. Type the command *gitflow init*.
After the execution of the program it will be proposed to rename the branch prefix, to adhere to the team's rules it is recommended to use the same standard prefixes.
For example, feature branches should follow the standard notation feature/<feature name>[2].

4.2 IntelliJ IDEA

Download the latest version available at the following link www.jetbrains.com/idea. Proceed with the installation. Once the installation is complete, open the previously imported project with Git. To do this, go to File− >Open and select the cloned repository.

4.3 Maven

4.3.1 Maven installation

Download the desired version at the following link maven.apache.org/download. Once the download is complete, continue with the installation, possibly following the guide on the above page. Check the correct installation of the software by searching for the terminal version. In Linux and MacOS environment, type *mvn -v*.

4.3.2 Maven configuration

Inside the repository there is a pom.xml file containing Maven configuration information. The version distributed with this product release is updated and tested, so no further changes are required to perform the planned lifecycle build. If you want to add configurations, refer to the official software documentation at this link maven.apache.org/guides.

[2]Report to note a guide to using GitFlow commands: git-flow-cheatsheet

5 Architecture

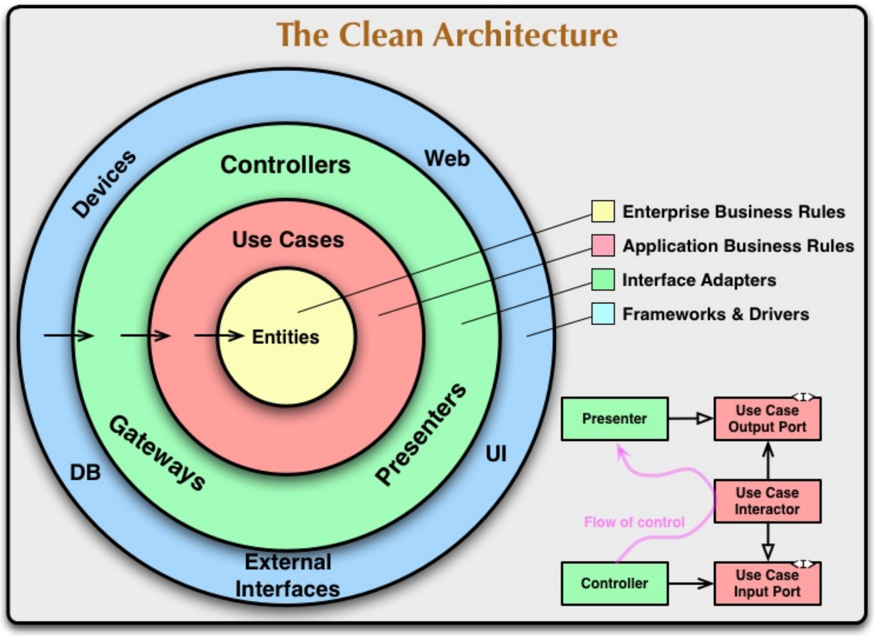
5.1 Preamble

In order to better understand how to extend the functionality of the product, it is necessary to understand the fundamentals and principles of the architecture used for the development of NaturalAPI. To achieve this goal, we invite the maintainer to read carefully the following section.

5.2 Clean architecture

Designed by Robert C. Martin, Clean Architecture is a particular layered architecture, where each layer performs a specific task.

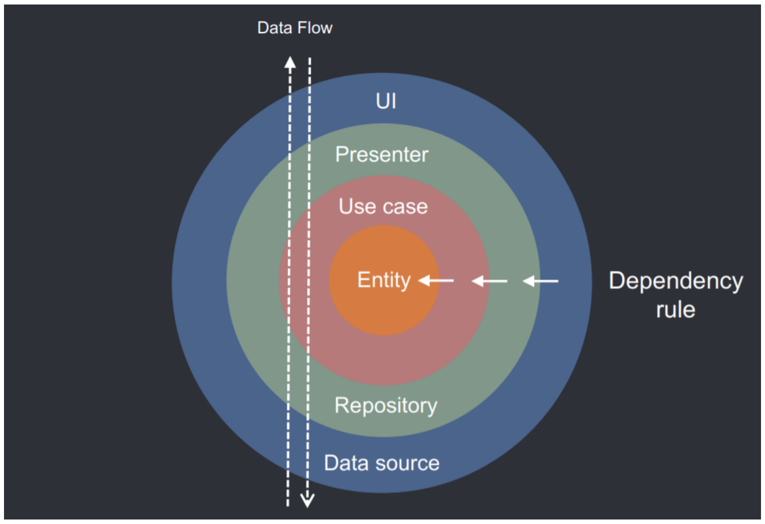
- **Entities** : the inner circle contains the "Enterprise Business Rules", i.e. the general and high-level rules.(They are the most stable component and less subject to change);
- **Use case**: the use case layer indicates the "Application Business Rules" and implements all use cases of the system. It represents the procedures that we want to automate;
- **Interface Adapter**: where there are "adapters" that convert data from the most convenient format for use cases and entities, to the most convenient format for other external sources (e.g. Database/Web);
- **Framework and Driver**: frameworks and tools, such as the database: it contains the code that acts as "glue" to communicate with the inner circle. This layer represents implementation details that can be replaced.



5.2.1 Dependency rule

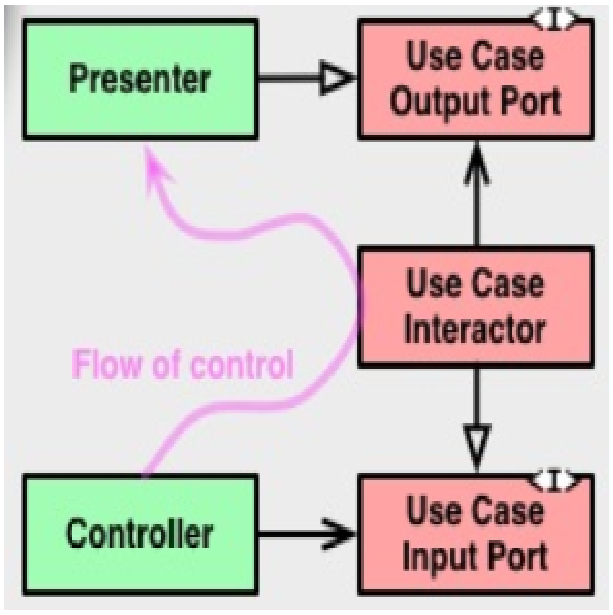
The clean architecture is structured around the principle of the dependency rule:

- Dependencies must go from the outer circle to the inner circle and never vice versa;
- The inner circles know nothing about the outer ones. In particular, the inner circles must never invoke methods of the outer circles;
- The data format used by the outer circles should not be used by the inner circles.



5.2.2 Dependency inversion principle

The program control flow causes dependency rule violations: internal circles call methods of external ones. To avoid this problem, the dependency inversion principle is applied where interfaces are defined that are implemented by the external ones, so the called methods belong to the same circle.



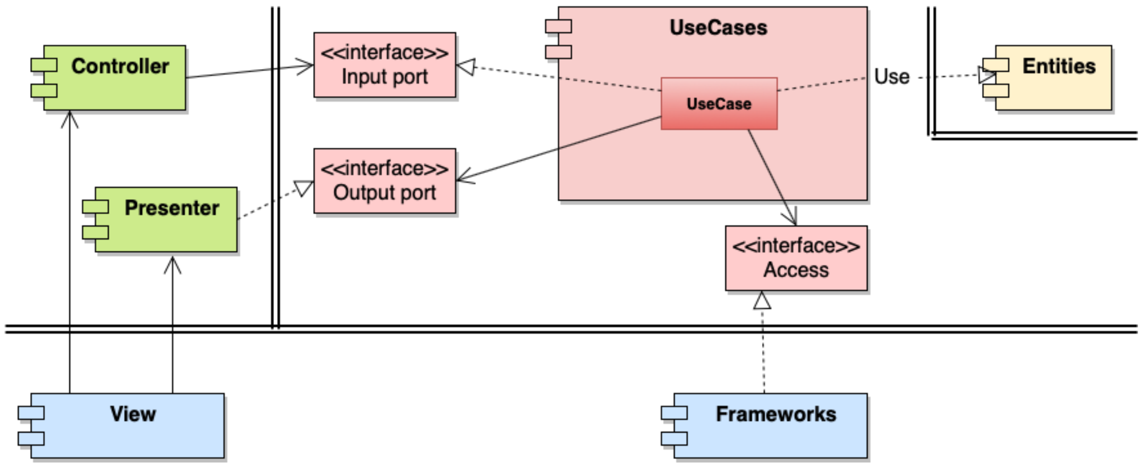
5.3 NaturalAPI

5.3.1 Composition and general architecture

NaturalAPI consists of three modules that must be able to operate independently:

- NaturalAPI Discover;
- NaturalAPI Design;
- NaturalAPI Develop;

The architectural scheme followed for their design and coding is the same and is based on the *Clean Architecture*. It can be summarized in the following diagram:



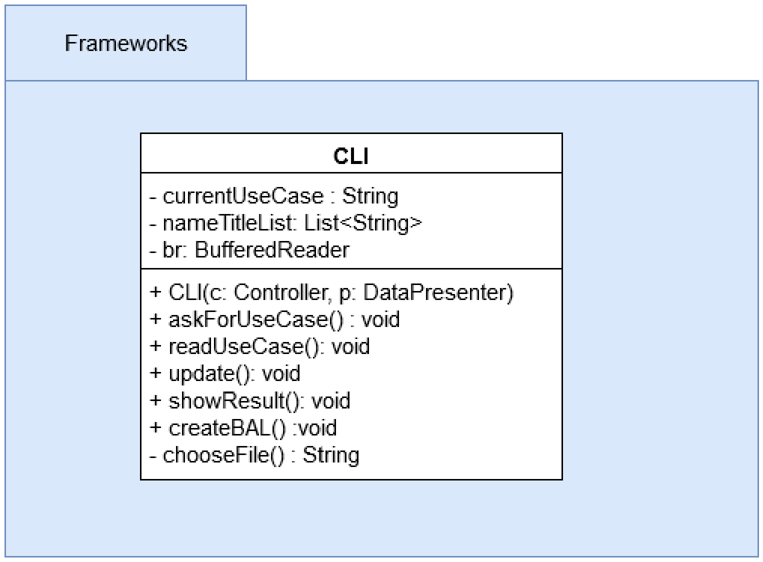
The diagram shows that:

- Entities doesn't have any kind of dependence. The use cases depends on the methods and fields given by the entities;
- Frameworks implement the output port interfaces that are used by external circles to invoke the use case;
- Frameworks depends on input port interfaces to interact with external circles;
- Controllers and presenters depend on the input and output ports that contain the signatures of the methods used to interact with the use cases;
- The outer layer contains tools such as databases and GUI/CLI. The code of this module is used to communicate with the inner layers. We want to keep these components outside so as not to pollute the software logic.

To avoid unnecessarily repeating simple concepts shared between different modules, the next sub-sections will include the components that are common to all three NaturalAPI modules.

5.3.2 View

In all three modules, a Command Line Interface has been implemented through a "CLI" class and a particular GUI for each module that will be better described later.

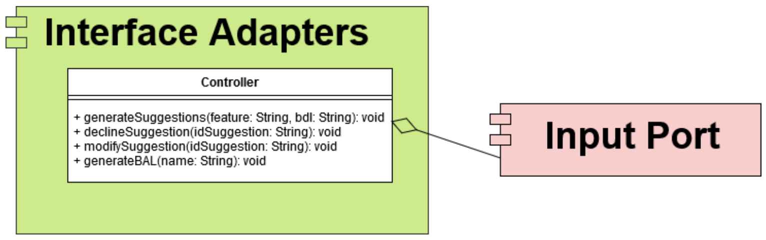


This class, very similar for all modules, has three data fields, a string indicating the use case chosen by the user; a list of strings indicating the names of the documents the user has chosen and the BufferedReader used to read the input.

5.4 Controller and DataPresenter

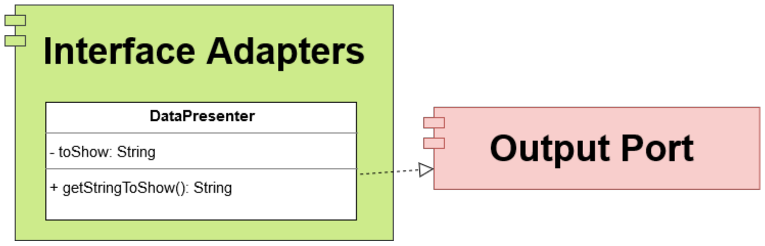
5.4.1 Controller

The controller in the clean architecture calls up the appropriate use cases by providing them with user input via the input ports.
In fact, the controller will have as data field the input ports, which are interfaces, and will use their methods to pass the data to the use case.
The controller's data fields will be instantiated with the use cases as they are a concrete class that implements the input ports.
So the controller calls the interface method, but it will actually be called the use case method.


Interface Adapters
Controller
+ generateSuggestions(feature: String, bdl: String): void
+ declineSuggestion(idSuggestion: String): void
+ modifySuggestion(idSuggestion: String): void
+ generateBAL(name: String): void
Input Port

5.4.2 DataPresenter

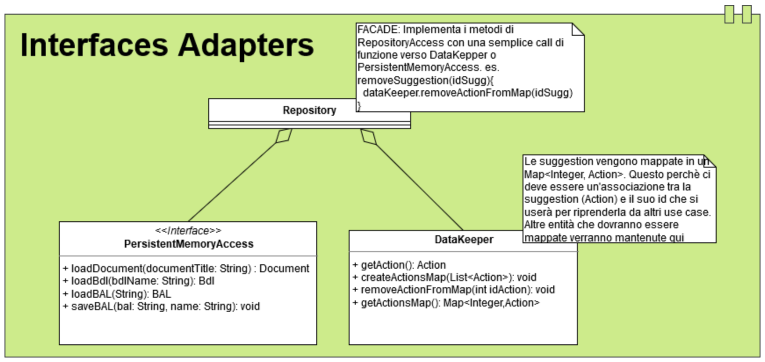
The task of the DataPresenter is instead to take the data from the use cases and format them so that the view can read them.
To do this, it relies on output ports, which are interfaces that are implemented by the DataPresenter.
These interfaces will be data fields of the use cases that will be instantiated with the DataPresenter.
The principle that is used is the same as the controller with the input ports, but this time it is the use case that has a reference to the DataPresenter through the use of the output ports, so the methods that are called are those of the presenter.
In our case, the DataPresenter receives data, formats it and asks the CLI to take it.



Both controller and DataPresenter, for each module are very similar, they change the various ports that the DataPresenter implements and that the controller has as a private field.

5.5 DataKeeper

The DataKeeper, as the name suggests, is a container of object references. Its use is necessary to allow use cases to work on the same entity instances. Each of these is associated with an id so that it can be subsequently identified: each reading from the DataKeeper requires the id of the object you want to retrieve. The DataKeeper has been implemented only in the Design and Develop modules, because for Discover there is no need to share object references between use cases.


Interfaces Adapters
FACADE: Implementa i metodi di RepositoryAccess con una semplice call di funzione verso DataKepper o PersistentMemoryAccess. es. removeSuggestion(idSugg){ dataKeeper.removeActionFromMap(idSugg) }
Repository
Le suggestion vengono mappate in un Map<Integer, Action>. Questo perchè ci deve essere un'associazione tra la suggestion (Action) e il suo id che si userà per riprenderla da altri use case. Altre entità che dovranno essere mappate verranno mantenute qui
<<Interface>>
PersistentMemoryAccess
+ loadDocument(documentTitle: String) : Document
+ loadBdl(bdlName: String): Bdl
+ loadBAL(String): BAL
+ saveBAL(bal: String, name: String): void
DataKeeper
+ getAction(): Action
+ createActionsMap(List<Action>): void
+ removeActionFromMap(int idAction): void
+ getActionsMap(): Map<Integer,Action>

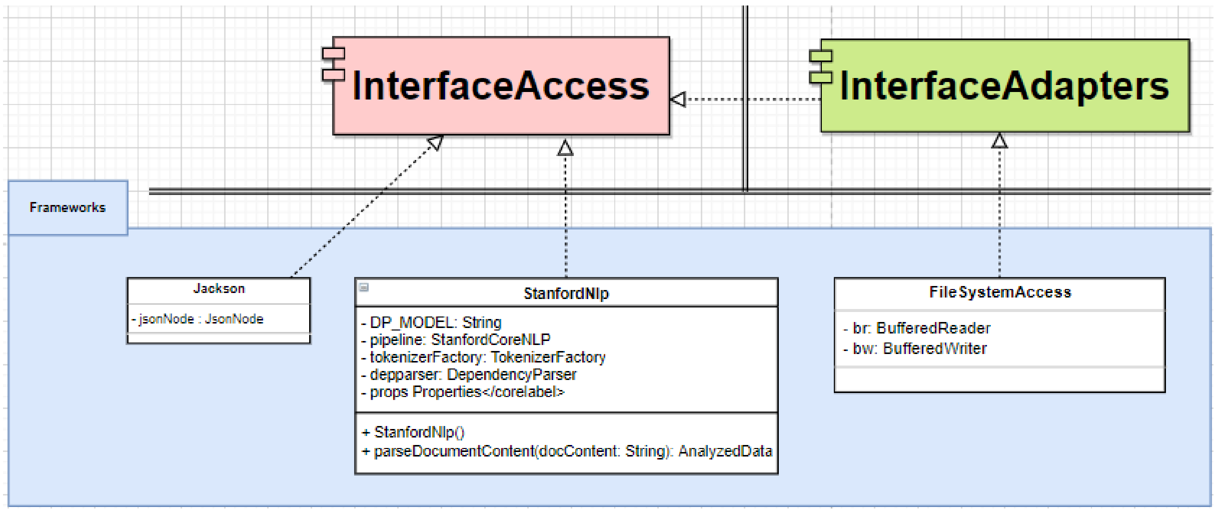
Repository is a class that implements the methods of the interface that allows the connection with the inner layer, and also acts as a *facade* to the DataKeeper and another interface called PersisentMemoryAccess.

5.5.1 Framework and libraries

The main frameworks/libraries NaturalAPI currently supports are two:

- **Stanford Parser** for the documents processing;
- **Jackson** for the translation of entities into json format and vice versa.

NaturalAPI does not use a database but implements functions that make use of writing and reading from files using Java BufferedReader and BufferedWriter. The FileSystemAccess class is the one that fulfills this purpose.


InterfaceAccess
InterfaceAdapters
Frameworks
Jackson
- jsonNode : JsonNode
StanfordNlp
- DP_MODEL: String
- pipeline: StanfordCoreNLP
- tokenizerFactory: TokenizerFactory
- depparser: DependencyParser
- props Properties</corelabel>
+ StanfordNlp()
+ parseDocumentContent(docContent: String): AnalyzedData
FileSystemAccess
- br: BufferedReader
- bw: BufferedWriter

5.6 NaturalAPI Discover

5.6.1 Purpuse

This module deals with finding the most used terms in a certain business and the way they relate. The first step is to collect business related documents. The result is a Business Domain Language (BDL) that contains the most used terms.

5.6.2 Entities

The Discover module uses two entities called "Document" and "BDL". The first one simply contains two private string type data fields that indicate the file name and its content respectively.
The second contains a string type data field that indicates the name of the BDL and three lists of WordCount, object that allows you to associate a word to its relative frequency.

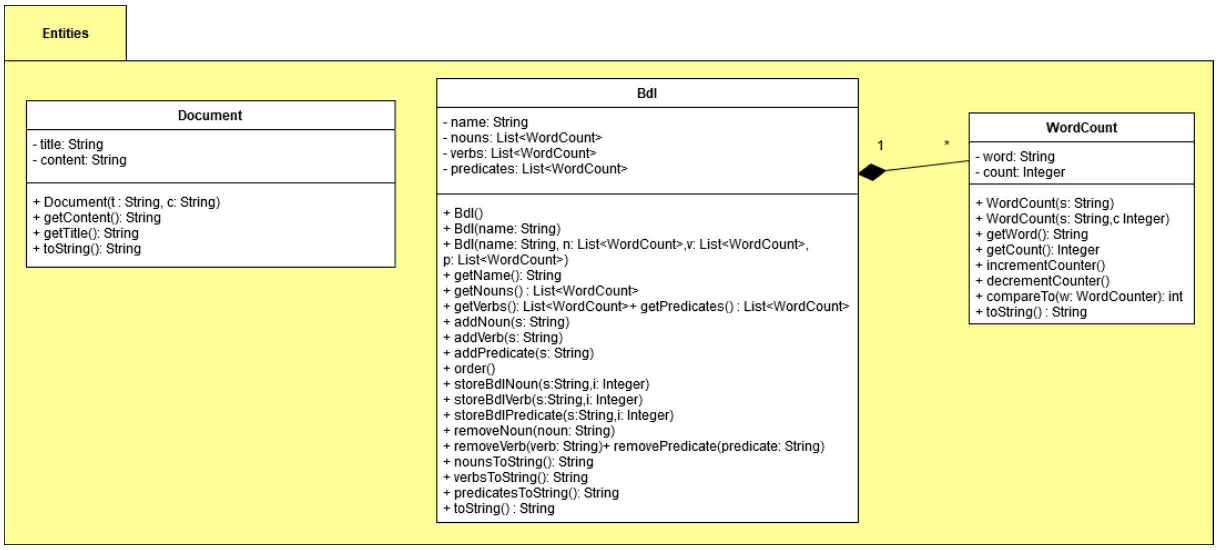

Entities
Document
- title: String
- content: String
+ Document(t : String, c: String)
+ getContent(): String
+ getTitle(): String
+ toString(): String
Bdl
- name: String
- nouns: List<WordCount>
- verbs: List<WordCount>
- predicates: List<WordCount>
+ Bdl()
+ Bdl(name: String)
+ Bdl(name: String, n: List<WordCount>,v: List<WordCount>, p: List<WordCount>)
+ getName(): String
+ getNouns() : List<WordCount>
+ getVerbs(): List<WordCount>+ getPredicates() : List<WordCount>
+ addNoun(s: String)
+ addVerb(s: String)
+ addPredicate(s: String)
+ order()
+ storeBdlNoun(s:String,i: Integer)
+ storeBdlVerb(s:String,i: Integer)
+ storeBdlPredicate(s:String,i: Integer)
+ removeNoun(noun: String)
+ removeVerb(verb: String)+ removePredicate(predicate: String)
+ nounsToString(): String
+ verbsToString(): String
+ predicatesToString(): String
+ toString() : String
1
*
WordCount
- word: String
- count: Integer
+ WordCount(s: String)
+ WordCount(s: String,c Integer)
+ getWord(): String
+ getCount(): Integer
+ incrementCounter()
+ decrementCounter()
+ compareTo(w: WordCounter): int
+ toString() : String

Figura 1: Entities NaturalAPI Discover

5.6.3 Use case

The use cases of NaturalAPI Discover are the follows:

- **CreateBdl**: Allows the creation of a BDL by giving input a series of text files;
- **AddDocuments**: You can integrate a BDL by adding new documents;
- **RemoveDocuments**: You can remove documents from an already existing BDL;
- **ViewBdl**: Use case that gives you the possibility to view a Bdl.

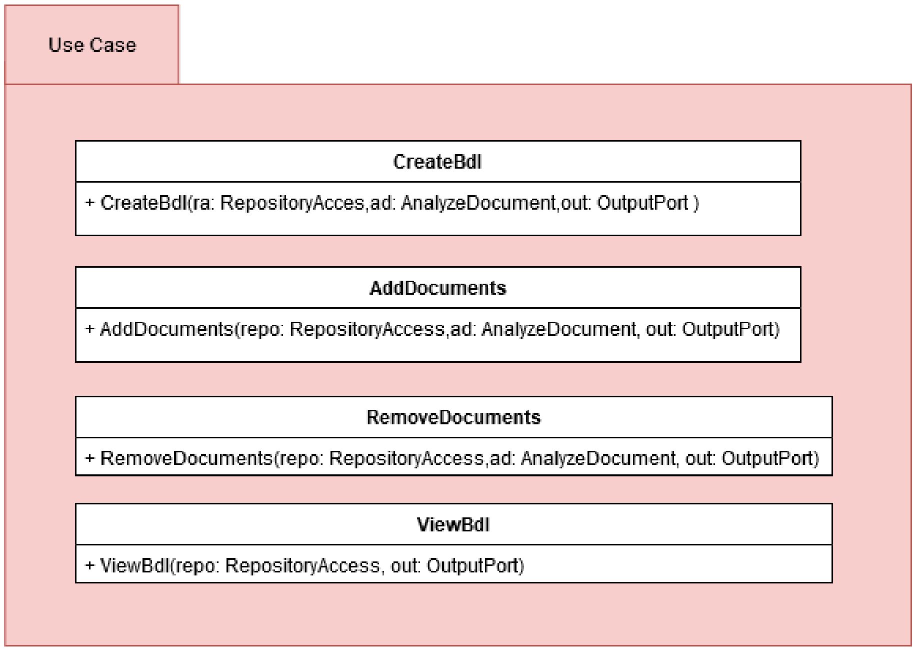
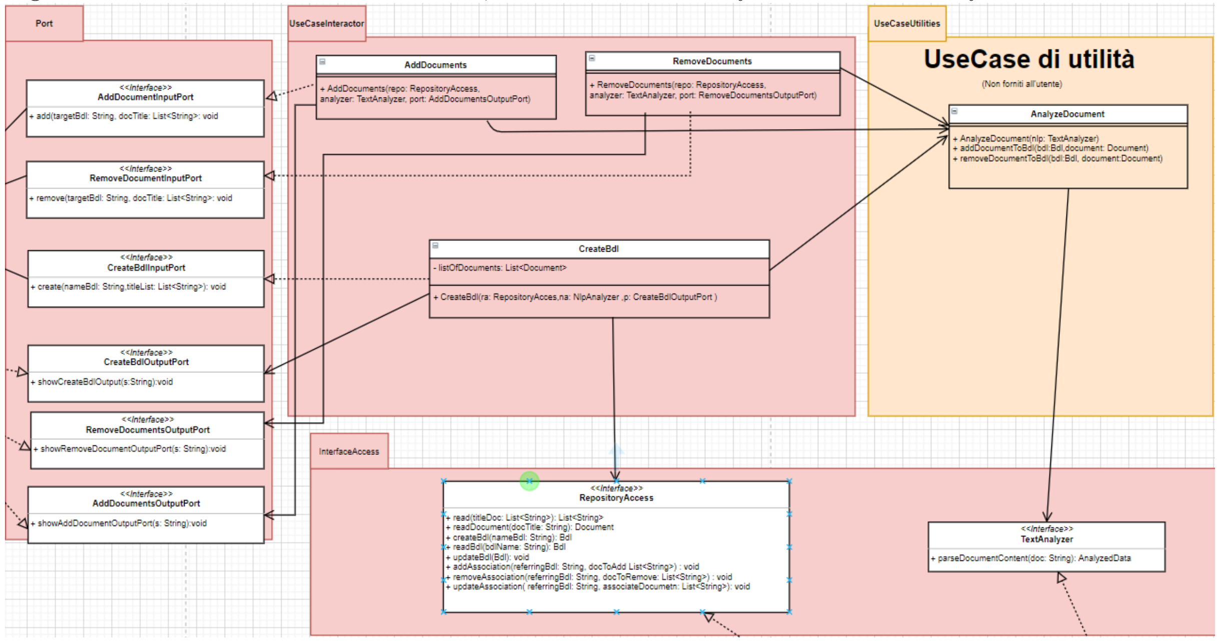

Use Case
CreateBdl
+ CreateBdl(ra: RepositoryAcces,ad: AnalyzeDocument,out: OutputPort)
AddDocuments
+ AddDocuments(repo: RepositoryAccess,ad: AnalyzeDocument, out: OutputPort)
RemoveDocuments
+ RemoveDocuments(repo: RepositoryAccess,ad: AnalyzeDocument, out: OutputPort)
ViewBdl
+ ViewBdl(repo: RepositoryAccess, out: OutputPort)

Figura 2: Use case NaturalAPI Discover

This module use "AnalyzeDocument", an utility use case, whose function is to support document analysis and management for use case interactors. In fact, it is not linked to any direct functionality offered to the user.


Port
<<Interface>>
AddDocumentInputPort
+ add(targetBdl: String, docTitle: List<String>: void
<<Interface>>
RemoveDocumentInputPort
+ remove(targetBdl: String, docTitle: List<String>: void
<<Interface>>
CreateBdlInputPort
+ create(nameBdl: String,titleList: List<String>): void
<<Interface>>
CreateBdlOutputPort
+ showCreateBdlOutput(s:String):void
<<Interface>>
RemoveDocumentsOutputPort
+ showRemoveDocumentOutputPort(s: String):void
<<Interface>>
AddDocumentsOutputPort
+ showAddDocumentOutputPort(s: String):void
UseCaseInteractor
AddDocuments
+ AddDocuments(repo: RepositoryAccess, analyzer: TextAnalyzer, port: AddDocumentsOutputPort)
RemoveDocuments
+ RemoveDocuments(repo: RepositoryAccess, analyzer: TextAnalyzer, port: RemoveDocumentsOutputPort)
CreateBdl
- listOfDocuments: List<Document>
+ CreateBdl(ra: RepositoryAcces,na: NlpAnalyzer ,p: CreateBdlOutputPort)
UseCaseUtilities
UseCase di utilità
(Non forniti all'utente)
AnalyzeDocument
+ AnalyzeDocument(nlp: TextAnalyzer)
+ addDocumentToBdl(bdl:Bdl,document: Document)
+ removeDocumentToBdl(bdl:Bdl, document:Document)
InterfaceAccess
<<Interface>>
RepositoryAccess
+ read(titleDoc: List<String>): List<String>
+ readDocument(docTitle: String): Document
+ createBdl(nameBdl: String): Bdl
+ readBdl(bdlName: String): Bdl
+ updateBdl(Bdl): void
+ addAssociation(referringBdl: String, docToAdd List<String>) : void
+ removeAssociation(referringBdl: String, docToRemove: List<String>) : void
+ updateAssociation(referringBdl: String, associateDocumetn: List<String>): void
<<Interface>>
TextAnalyzer
+ parseDocumentContent(doc: String): AnalyzedData

All use cases use the Repository interface to access the outer layer.
They also have to use the Stanford Parser to perform their operations, so some of them are connected to Analyze-Document which will have an interface to communicate.

5.6.4 Graphic User Interface Discover

The graphic user interface of all modules has been created using the Swing library provided by Java. As far as Discover is concerned, the GUI is formed by a single class called GUI_Discover.

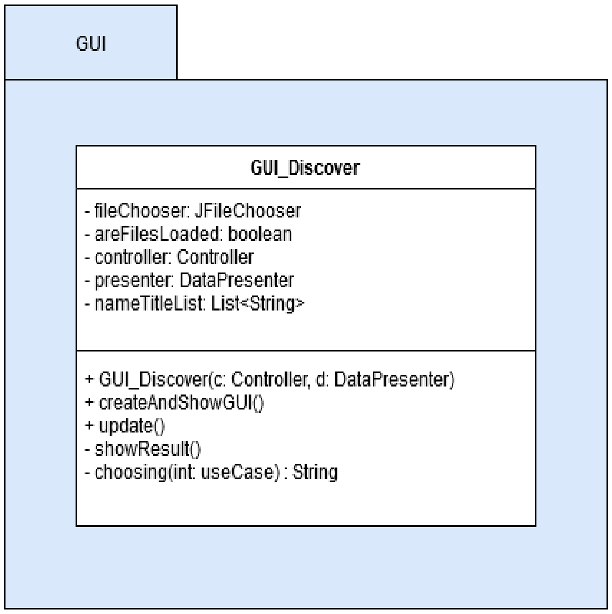
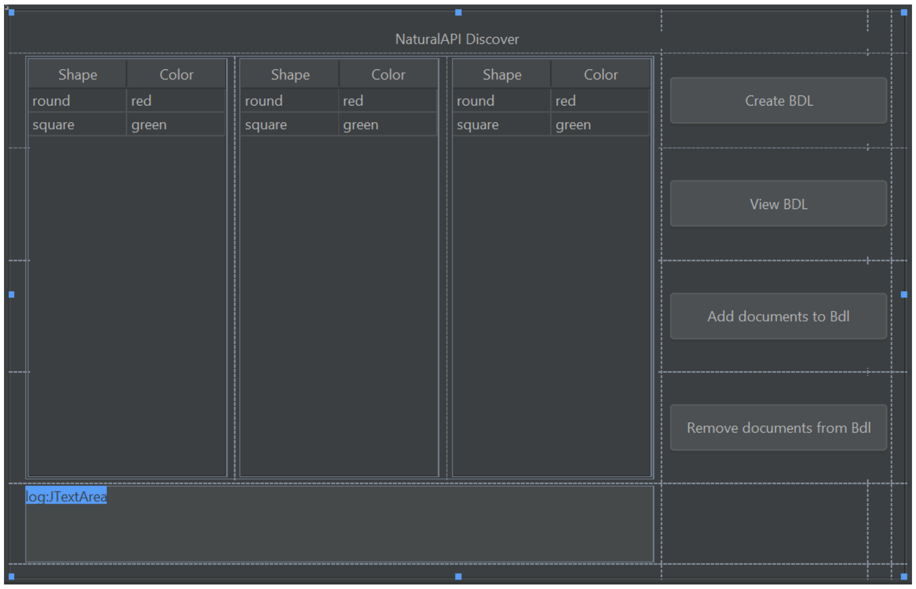

GUI
GUI_Discover
- fileChooser: JFileChooser
- areFilesLoaded: boolean
- controller: Controller
- presenter: DataPresenter
- nameTitleList: List<String>
+ GUI_Discover(c: Controller, d: DataPresenter)
+ createAndShowGUI()
+ update()
- showResult()
- choosing(int: useCase) : String

Figura 3: Diagram of GUI_Discover class

For ease of viewing, several data fields related to the GUI_Discover have been excluded. In general the GUI is composed of:

- **JPanel**: for the division of the various Widgets into panels;
- **JScrollPane**: for the division of the various Widgets into sliding panels;
- **JButton**: simple buttons;
- **JTextPane**: panel containing a text area;
- **JTextArea**: panel containing a text area;
- **JFileChooser**: for the file choosing;
- **JTable**: to visualize data in a table.



Shape	Color
round	red
square	green

Shape	Color
round	red
square	green

Shape	Color
round	red
square	green

As you can see from the figure above, the GUI of NaturalAPI Discover consists of several buttons to which are associated events that will trigger the use case chosen by the user.
The text areas instead are used to show the various data.

Events and Listener

For some of the present elements, mainly the JButtons, the listener has been defined.
When the user performs a certain action, such as pressing a button, the listener will see that the button has been pressed and a certain part of the code is executed.
In particular, the controller method, which triggers the chosen use case, will be called up.
The various component listener and actions are defined in the class constructor.

5.7 NaturalAPI Design

5.7.1 Purpuse

This module has the task of extrapolating from features file containing the scenarios in *Gherkin* format of the operations that will form the *BAL*, a high-level language that represents API.

5.7.2 Prerequisites for correct operation

In order for *NaturalAPI Design* to automatically obtain the actor of each file.feature, the user must enter the keyword "As a" in the second line of the file followed by the name of the actor concerned, preceded by the keyword "Feature:". In case the keyword is not found, NaturalAPI Design will assume that that feature refers to all the actors of the system you want to model. In this case, the actor will be called "All". This behavior is defined in the GenerateBALSuggestion use case (see Use Case section).

5.7.3 Entities

The main entity is the *BAL*. It is composed of *Actor* because you want to group the actions extrapolated from a scenario (the *Action*) per actor.

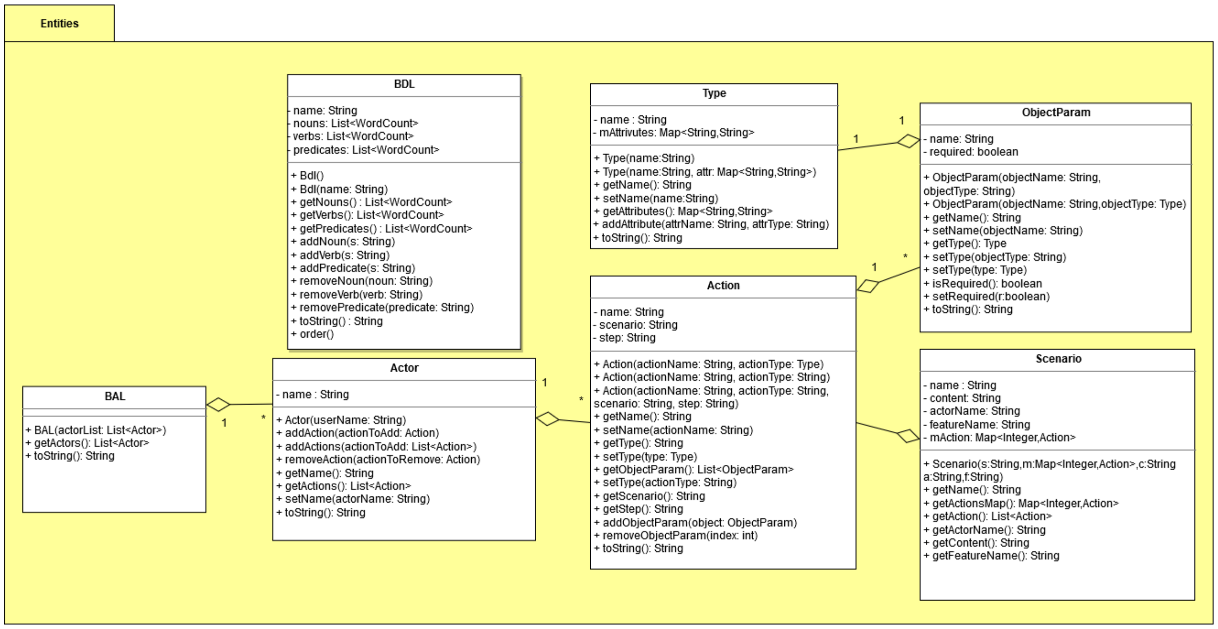

Entities
BDL
- name: String
- nouns: List<WordCount>
- verbs: List<WordCount>
- predicates: List<WordCount>
+ Bdl()
+ Bdl(name: String)
+ getNouns() : List<WordCount>
+ getVerbs(): List<WordCount>
+ getPredicates() : List<WordCount>
+ addNoun(s: String)
+ addVerb(s: String)
+ addPredicate(s: String)
+ removeNoun(noun: String)
+ removeVerb(verb: String)
+ removePredicate(predicate: String)
+ toString() : String
+ order()
Type
- name : String
- mAttrivutes: Map<String,String>
+ Type(name:String)
+ Type(name:String, attr: Map<String,String>)
+ getName(): String
+ setName(name:String)
+ getAttributes(): Map<String,String>
+ addAttribute(attrName: String, attrType: String)
+ toString(): String
1
1
ObjectParam
- name: String
- required: boolean
+ ObjectParam(objectName: String, objectType: String)
+ ObjectParam(objectName: String,objectType: Type)
+ getName(): String
+ setName(objectName: String)
+ getType(): Type
+ setType(objectType: String)
+ setType(type: Type)
+ isRequired(): boolean
+ setRequired(r:boolean)
+ toString(): String
1
*
Action
- name: String
- scenario: String
- step: String
+ Action(actionName: String, actionType: Type)
+ Action(actionName: String, actionType: String)
+ Action(actionName: String, actionType: String, scenario: String, step: String)
+ getName(): String
+ setName(actionName: String)
+ getType(): String
+ setType(type: Type)
+ getObjectParam(): List<ObjectParam>
+ setType(actionType: String)
+ getScenario(): String
+ getStep(): String
+ addObjectParam(object: ObjectParam)
+ removeObjectParam(index: int)
+ toString(): String
BAL
+ BAL(actorList: List<Actor>)
+ getActors(): List<Actor>
+ toString(): String
1
*
Actor
- name : String
+ Actor(userName: String)
+ addAction(actionToAdd: Action)
+ addActions(actionToAdd: List<Action>)
+ removeAction(actionToRemove: Action)
+ getName(): String
+ getActions(): List<Action>
+ setName(actorName: String)
+ toString(): String
1
*
Scenario
- name : String
- content: String
- actorName: String
- featureName: String
- mAction: Map<Integer,Action>
+ Scenario(s:String,m:Map<Integer,Action>,c:String a:String,f:String)
+ getName(): String
+ getActionsMap(): Map<Integer,Action>
+ getAction(): List<Action>
+ getActorName(): String
+ getContent(): String
+ getFeatureName(): String

Figura 4: Entities NaturalAPI Design

The scenario entity takes care, in addition to keeping the content of the scenario imported by the user, of "mapping" the suggestions for a given scenario. It also keeps track of the name of the actor responsible for the actions.

5.7.4 Use case

The use cases in this module are:

- **GenerateBALSuggestions**: generates suggestions for BAL. A suggestion is seen as an *Action* candidate for BAL. The private method extractActorName() takes care of getting the name of the actor from the scenario as described above in the prerequisites section;
- **DeclineBALSuggestion**: delete a candidate suggestion for the BAL through its id and the id of the scenario it belongs to;

- **AddBALSuggestion**: allows you to add suggestions to a scenario;
- **ModifyBALSuggestion**: allows the modification of some fields of a suggestion;
- **CreateCustomType**: gives the possibility to create custom type;
- **GenerateBAL**: collects the suggestions generated for each scenario and encloses them inside a BAL object. This is finally saved as a string in permanent memory;
- **LoadBDL**: for loading the BDL into the program;
- **RemoveBDL**: for the removal of BDL from the program;
- **ShowTypes**: for displaying types.

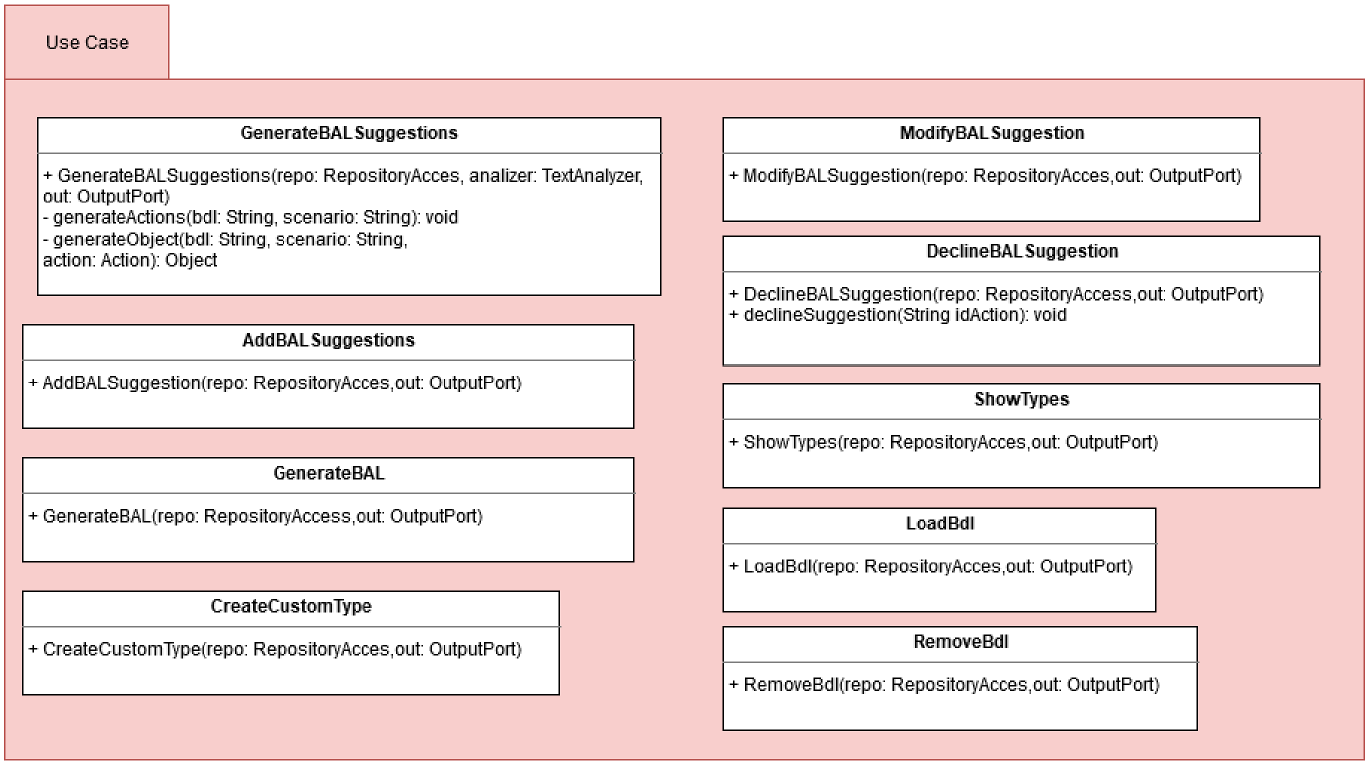
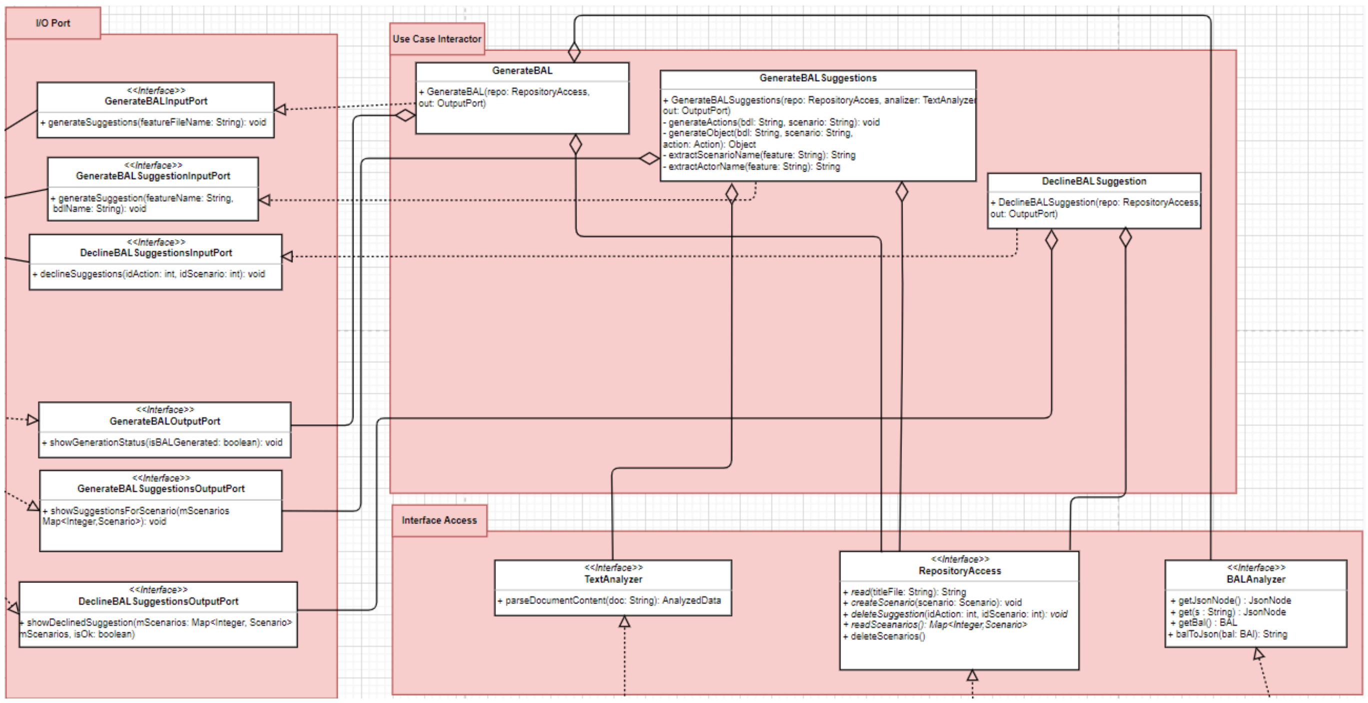

Use Case
GenerateBALSuggestions
+ GenerateBALSuggestions(repo: RepositoryAcces, analizer: TextAnalyzer, out: OutputPort)
- generateActions(bdl: String, scenario: String): void
- generateObject(bdl: String, scenario: String, action: Action): Object
AddBALSuggestions
+ AddBALSuggestion(repo: RepositoryAcces,out: OutputPort)
GenerateBAL
+ GenerateBAL(repo: RepositoryAccess,out: OutputPort)
CreateCustomType
+ CreateCustomType(repo: RepositoryAcces,out: OutputPort)
ModifyBALSuggestion
+ ModifyBALSuggestion(repo: RepositoryAcces,out: OutputPort)
DeclineBALSuggestion
+ DeclineBALSuggestion(repo: RepositoryAccess,out: OutputPort)
+ declineSuggestion(String idAction): void
ShowTypes
+ ShowTypes(repo: RepositoryAcces,out: OutputPort)
LoadBdl
+ LoadBdl(repo: RepositoryAcces,out: OutputPort)
RemoveBdl
+ RemoveBdl(repo: RepositoryAcces,out: OutputPort)

Figura 5: Use case NaturalAPI Design

Use Cases implement the related input and output ports as illustrated by the following diagram:
N.B. For ease of viewing, several use cases will be excluded in the image below


I/O Port
Use Case Interactor
Interface Access
<<Interface>> GenerateBALInputPort
+ generateSuggestions(featureFileName: String): void
<<Interface>> GenerateBALSuggestionInputPort
+ generateSuggestion(featureName: String, bdlName: String): void
<<Interface>> DeclineBALSuggestionsInputPort
+ declineSuggestions(idAction: int, idScenario: int): void
<<Interface>> GenerateBALOutputPort
+ showGenerationStatus(isBALGenerated: boolean): void
<<Interface>> GenerateBALSuggestionsOutputPort
+ showSuggestionsForScenario(mScenarios Map<Integer,Scenario>): void
<<Interface>> DeclineBALSuggestionsOutputPort
+ showDeclinedSuggestion(mScenarios: Map<Integer, Scenario> mScenarios, isOk: boolean)
GenerateBAL
+ GenerateBAL(repo: RepositoryAccess, out: OutputPort)
GenerateBALSuggestions
+ GenerateBALSuggestions(repo: RepositoryAcces, analizer: TextAnalyzer, out: OutputPort)
- generateActions(bdl: String, scenario: String): void
- generateObject(bdl: String, scenario: String, action: Action): Object
- extractScenarioName(feature: String): String
- extractActorName(feature: String): String
DeclineBALSuggestion
+ DeclineBALSuggestion(repo: RepositoryAccess, out: OutputPort)
<<Interface>> TextAnalyzer
+ parseDocumentContent(doc: String): AnalyzedData
<<Interface>> RepositoryAccess
+ read(titleFile: String): String
+ createScenario(scenario: Scenario): void
+ deleteSuggestion(idAction: int, idScenario: int): void
+ readSceanarios(): Map<Integer,Scenario>
+ deleteScenarios()
<<Interface>> BALAnalyzer
+ getJsonNode() : JsonNode
+ get(s : String) : JsonNode
+ getBal() : BAL
+ balToJson(bal: BAl): String

They also use interfaces to communicate with the outer layers. In particular:

- **GenerateBALSuggestions**: needs a TextAnalyzer to analyze the scenarios provided in input;
- **GenerateBAL**: needs a BALAnalyzer to convert the BAL object to the most convenient string format (e.g. json).

All the use cases require access to a Repository to read, update, delete, or temporarily or permanently save the entities of interest to them.

5.7.5 Suggestion Frequency

For calculating the frequency of suggestions in Design using BDL, it was decided to add a class *SuggestionFrequency*. This class was not treated as a use case since it is not something the user can call or act on.
Also, integrating it into existing use cases would have meant fully modifying the already tested use cases and adding other dependencies to them.
To avoid these problems it was decided to delegate to the DataPresenter the task of taking the frequency of a term through the algorithm.

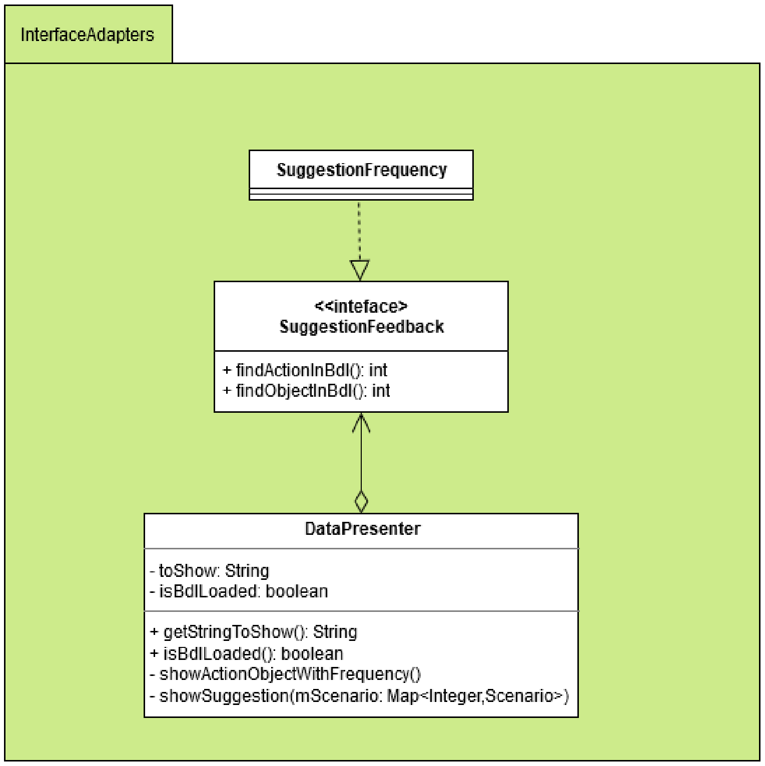


Figura 6: SuggestionFrequency

It was decided to implement the SuggestionFrequency class with a strategy called SuggestionFeedback.

5.7.6 Graphic User Interface Design

The graphic user interface of all modules has been created using the Swing library provided by Java.
This module consists of several classes.

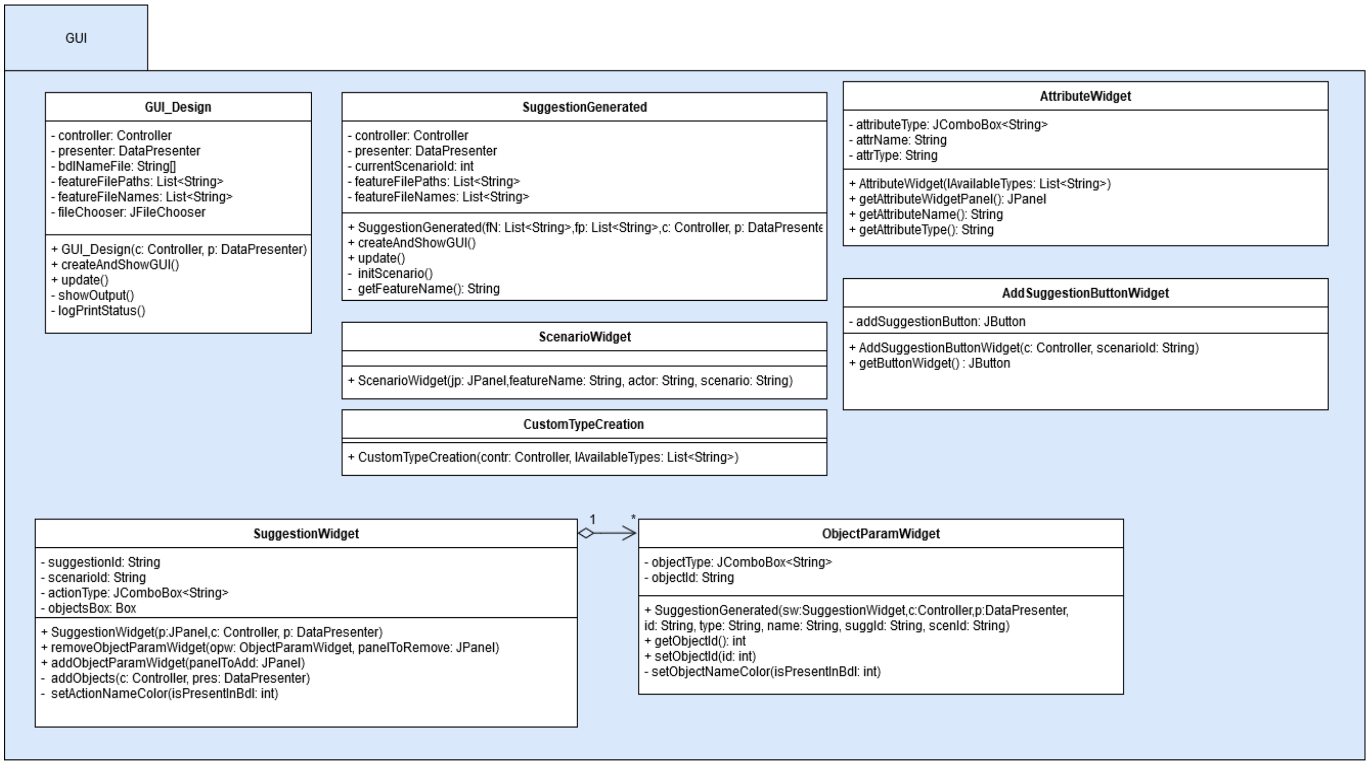

GUI
GUI_Design
- controller: Controller
- presenter: DataPresenter
- bdlNameFile: String[]
- featureFilePaths: List<String>
- featureFileNames: List<String>
- fileChooser: JFileChooser
+ GUI_Design(c: Controller, p: DataPresenter)
+ createAndShowGUI()
+ update()
- showOutput()
- logPrintStatus()
SuggestionGenerated
- controller: Controller
- presenter: DataPresenter
- currentScenarioId: int
- featureFilePaths: List<String>
- featureFileNames: List<String>
+ SuggestionGenerated(fN: List<String>,fp: List<String>,c: Controller, p: DataPresente
+ createAndShowGUI()
+ update()
- initScenario()
- getFeatureName(): String
AttributeWidget
- attributeType: JComboBox<String>
- attrName: String
- attrType: String
+ AttributeWidget(lAvailableTypes: List<String>)
+ getAttributeWidgetPanel(): JPanel
+ getAttributeName(): String
+ getAttributeType(): String
AddSuggestionButtonWidget
- addSuggestionButton: JButton
+ AddSuggestionButtonWidget(c: Controller, scenarioId: String)
+ getButtonWidget() : JButton
ScenarioWidget
+ ScenarioWidget(jp: JPanel,featureName: String, actor: String, scenario: String)
CustomTypeCreation
+ CustomTypeCreation(contr: Controller, lAvailableTypes: List<String>)
SuggestionWidget
- suggestionId: String
- scenarioId: String
- actionType: JComboBox<String>
- objectsBox: Box
+ SuggestionWidget(p:JPanel,c: Controller, p: DataPresenter)
+ removeObjectParamWidget(opw: ObjectParamWidget, panelToRemove: JPanel)
+ addObjectParamWidget(panelToAdd: JPanel)
- addObjects(c: Controller, pres: DataPresenter)
- setActionNameColor(isPresentInBdl: int)
1
*
ObjectParamWidget
- objectType: JComboBox<String>
- objectId: String
+ SuggestionGenerated(sw:SuggestionWidget,c:Controller,p:DataPresenter,
id: String, type: String, name: String, suggId: String, scenId: String)
+ getObjectId(): int
+ setObjectId(id: int)
- setObjectNameColor(isPresentInBdl: int)

Figura 7: Diagram of GUI_Discover class

For ease of viewing, several data fields related to the GUI_Design have been excluded.
In general the GUI is composed of:

- **JPanel**: for the division of the various Widgets into panels;
- **JScrollPane**: for the division of the various Widgets into sliding panels;
- **JButton**: simple buttons;
- **JTextPane**: panel containing a text area;
- **JTextArea**: panel containing a text area;
- **JLabel**: simple text;
- **JComboBox**: to allow multiple options;
- **GridBagLayout**: a particular layout applicable to panels;
- **GridBagConstraints**: some constraints to be applied to a GridBagLayout.

The *GUI_Design* class is the main class that is started in the main. All other classes are generated after a particular event created by the user. For example, when the user decides to create suggestions by pressing the corresponding button, an object of type *SuggestionGenerated*. will be instantiated.
It has been decided to further subdivide the widgets to be displayed, so that the code is as readable as possible. In fact, the *SuggestionGenerated* object will create other objects, in particular *ScenarioWidget* which contains the scenario entered by the user, and *SuggestionWidget* which contains the suggestions that have been generated.

Events and Listener

For some of the present elements, mainly the JButtons, the listener has been defined.
When the user performs a certain action, such as pressing a button, the listener will see that the button has been pressed and a certain part of the code is executed.
In particular, the controller method, which triggers the chosen use case, will be called up.
The various component listener and actions are defined in the class constructor.

5.8 NaturalAPI Develop

5.8.1 Purpuse

This module uses *BAL* to produce the API in a specific programming language. To perform the translation *NaturalAPI Develop* needs a *PLA* for the target language. API suggestions are provided and must be evaluated and accepted by the user before generating the API file. *NaturalAPI Develop* also allows you to modify or add methods to the API if the user deems it necessary.

5.8.2 Prerequisites for correct operation

For *NaturalApi Develop* to work correctly, you need to have a BAL in Json format, output of the Design, and a PLA in .txt format, in which the first line shows the extension of the API that will be created, and then you must write an API template with keywords instead of the class name, method name, type and parameter name. These keywords are "group_action", "action_type", "action_name", "parameter_type","parameter_name","custom_class","attribute_name" and "attribute_type" respectively.

5.8.3 Entities

The entities in this module are partially taken from the *NaturalApi Design* module, due to the use of *BAL*. Then there are the API and *PLA* entities: the first is responsible for mapping the API keeping track of the name of the file that will be created and the actual API; the second contains four string data fields that contain the extension of the API that will be created and the text of the *PLA*.

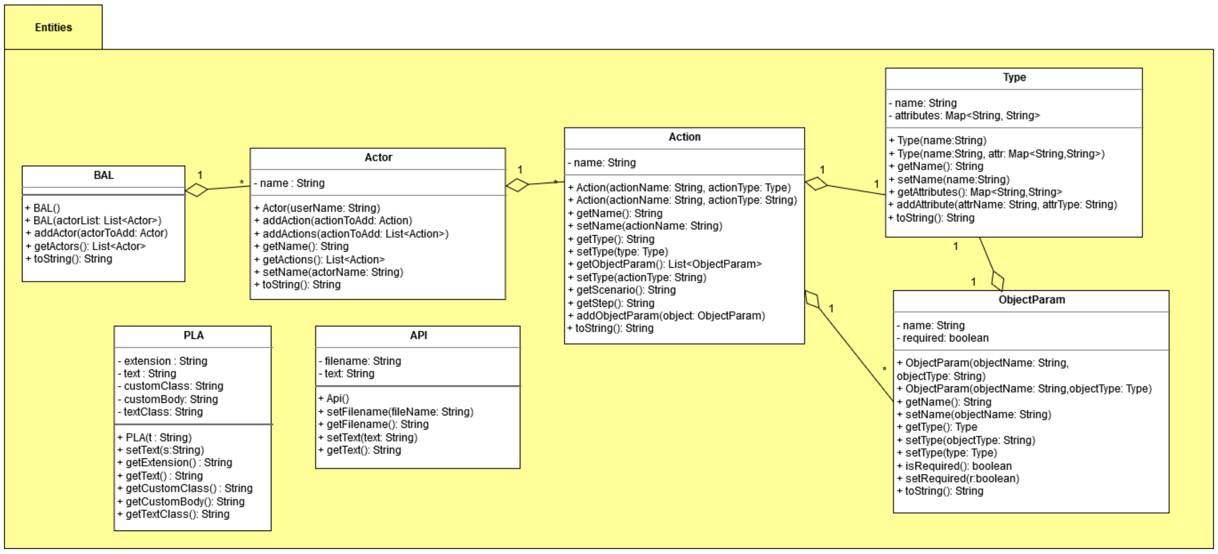

Entities
BAL
+ BAL()
+ BAL(actorList: List<Actor>)
+ addActor(actorToAdd: Actor)
+ getActors(): List<Actor>
+ toString(): String
Actor
- name : String
+ Actor(userName: String)
+ addAction(actionToAdd: Action)
+ addActions(actionToAdd: List<Action>)
+ getName(): String
+ getActions(): List<Action>
+ setName(actorName: String)
+ toString(): String
Action
- name: String
+ Action(actionName: String, actionType: Type)
+ Action(actionName: String, actionType: String)
+ getName(): String
+ setName(actionName: String)
+ getType(): String
+ setType(type: Type)
+ getObjectParam(): List<ObjectParam>
+ setType(actionType: String)
+ getScenario(): String
+ getStep(): String
+ addObjectParam(object: ObjectParam)
+ toString(): String
Type
- name: String
- attributes: Map<String, String>
+ Type(name:String)
+ Type(name:String, attr: Map<String,String>)
+ getName(): String
+ setName(name:String)
+ getAttributes(): Map<String,String>
+ addAttribute(attrName: String, attrType: String)
+ toString(): String
ObjectParam
- name: String
- required: boolean
+ ObjectParam(objectName: String, objectType: String)
+ ObjectParam(objectName: String,objectType: Type)
+ getName(): String
+ setName(objectName: String)
+ getType(): Type
+ setType(objectType: String)
+ setType(type: Type)
+ isRequired(): boolean
+ setRequired(r:boolean)
+ toString(): String
PLA
- extension : String
- text : String
- customClass: String
- customBody: String
- textClass: String
+ PLA(t : String)
+ setText(s:String)
+ getExtension() : String
+ getText() : String
+ getCustomClass() : String
+ getCustomBody(): String
+ getTextClass(): String
API
- filename: String
- text: String
+ Api()
+ setFilename(fileName: String)
+ getFilename(): String
+ setText(text: String)
+ getText(): String
1
*
1
*
1
1
1
1
1
*

Figura 8: Entities NaturalAPI Develop

5.8.4 Use case

The use cases of this module are:

- **SuggestApi**: it creates suggestin for API, starting from a *BAL* and a *PLA*;
- **GenerateApi**: it generates the API created by the SuggestApi use case;
- **ModifyApi**: allows you to modify the selected APIs before generating them;
- **CreatePLA**: allows THE creation of an *PLA* for a programming language;
- **ModifyPLA**: gives you the possibility to modify a certain *PLA*.

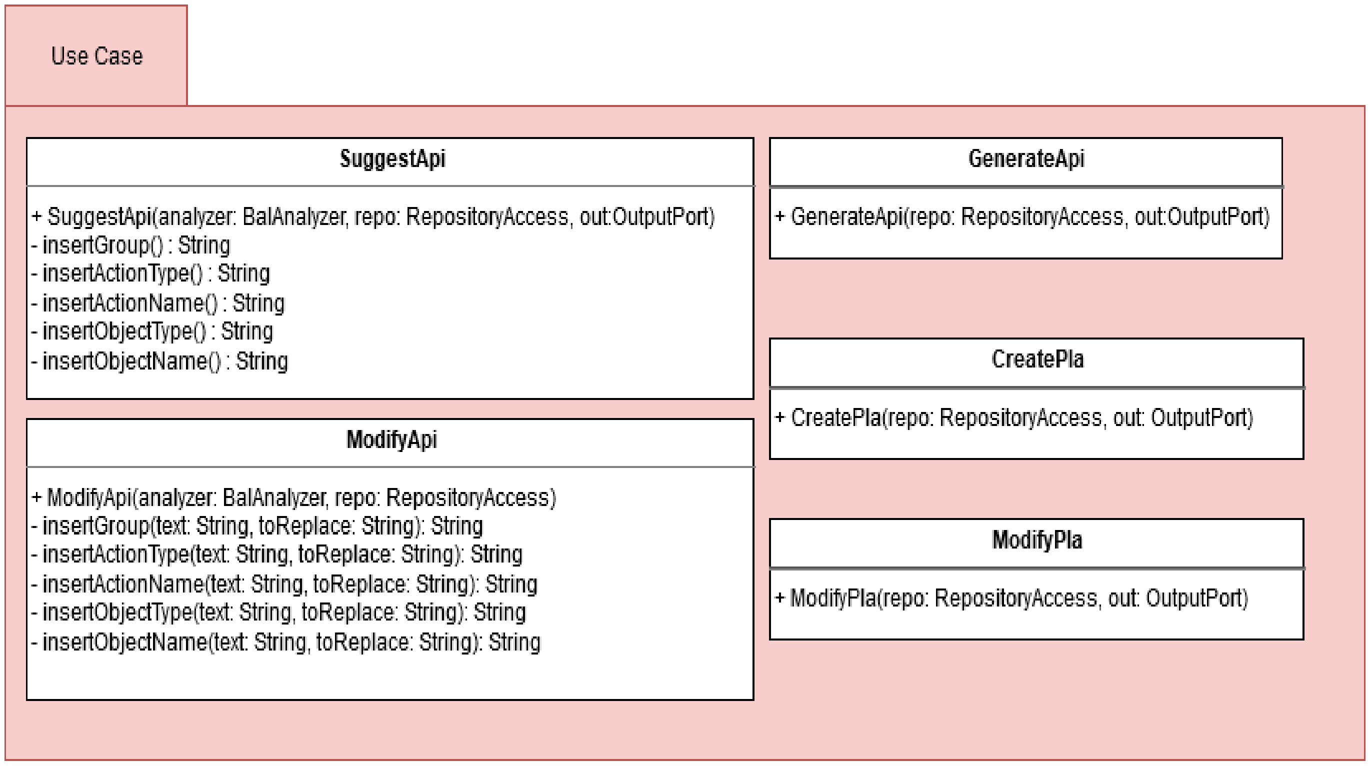
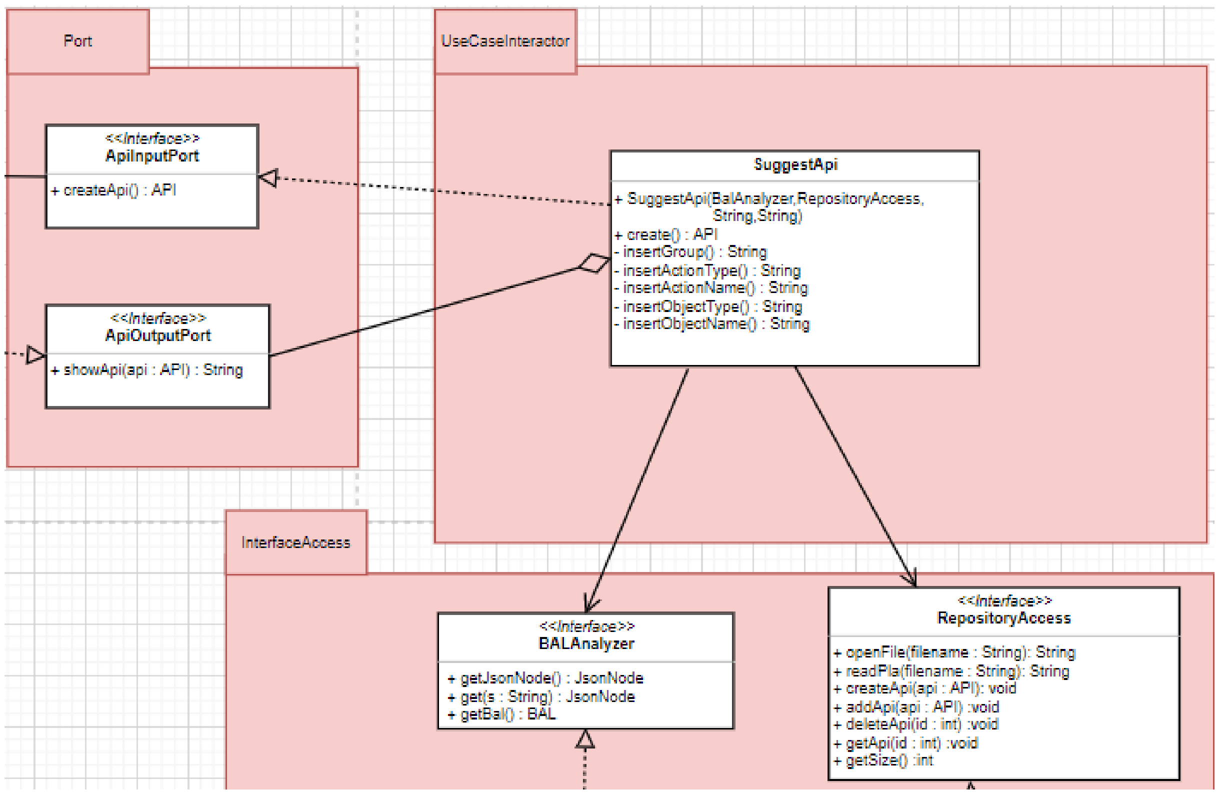

Use Case
SuggestApi
+ SuggestApi(analyzer: BalAnalyzer, repo: RepositoryAccess, out:OutputPort)
- insertGroup() : String
- insertActionType() : String
- insertActionName() : String
- insertObjectType() : String
- insertObjectName() : String
ModifyApi
+ ModifyApi(analyzer: BalAnalyzer, repo: RepositoryAccess)
- insertGroup(text: String, toReplace: String): String
- insertActionType(text: String, toReplace: String): String
- insertActionName(text: String, toReplace: String): String
- insertObjectType(text: String, toReplace: String): String
- insertObjectName(text: String, toReplace: String): String
GenerateApi
+ GenerateApi(repo: RepositoryAccess, out:OutputPort)
CreatePla
+ CreatePla(repo: RepositoryAccess, out: OutputPort)
ModifyPla
+ ModifyPla(repo: RepositoryAccess, out: OutputPort)

Figura 9: Use Case NaturalAPI Develop

The following diagrams show the interaction between use case and input/output port. You can also see that the use cases use interfaces to communicate with the frameworks of the outer layer. In fact, all and the use cases need access to a Repository to read, update, delete or save temporarily or permanently the entities of their interest.
In addition, SuggestApi and ModifyApi use BALAnalyzer to create the *BAL* object from the Json file.
N.B. For ease of viewing, only one use case will be shown.


Port
<<Interface>>
ApiInputPort
+ createApi() : API
<<Interface>>
ApiOutputPort
+ showApi(api : API) : String
UseCaseInteractor
SuggestApi
+ SuggestApi(BalAnalyzer,RepositoryAccess, String,String)
+ create() : API
- insertGroup() : String
- insertActionType() : String
- insertActionName() : String
- insertObjectType() : String
- insertObjectName() : String
InterfaceAccess
<<Interface>>
BALAnalyzer
+ getJsonNode() : JsonNode
+ get(s : String) : JsonNode
+ getBal() : BAL
<<Interface>>
RepositoryAccess
+ openFile(filename : String): String
+ readPla(filename : String): String
+ createApi(api : API): void
+ addApi(api : API) :void
+ deleteApi(id : int) :void
+ getApi(id : int) :void
+ getSize() :int

5.8.5 Graphic User Interface Develop

The graphic user interface of all modules has been created using the Swing library provided by Java. This module consists of three different classes.

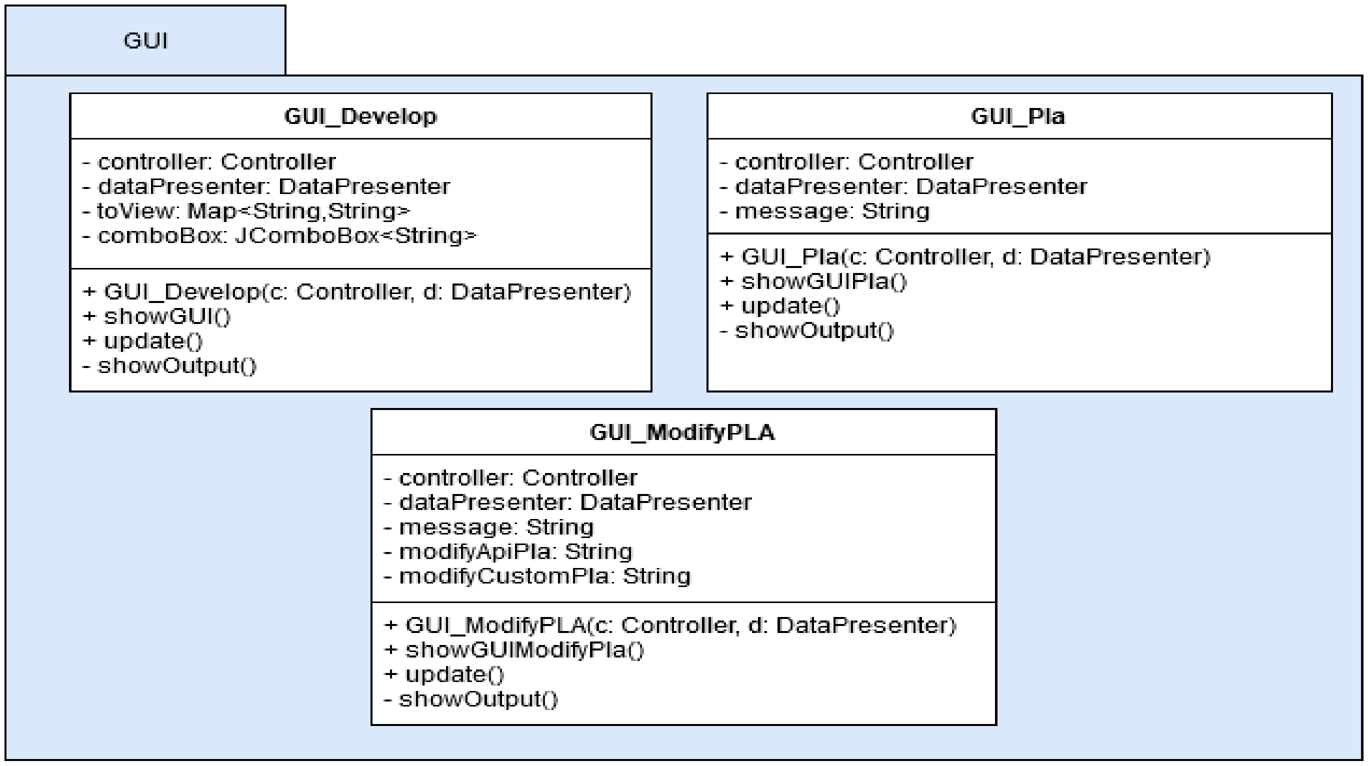

GUI
GUI_Develop
- controller: Controller
- dataPresenter: DataPresenter
- toView: Map<String,String>
- comboBox: JComboBox<String>
+ GUI_Develop(c: Controller, d: DataPresenter)
+ showGUI()
+ update()
- showOutput()
GUI_Pla
- controller: Controller
- dataPresenter: DataPresenter
- message: String
+ GUI_Pla(c: Controller, d: DataPresenter)
+ showGUIPla()
+ update()
- showOutput()
GUI_ModifyPLA
- controller: Controller
- dataPresenter: DataPresenter
- message: String
- modifyApiPla: String
- modifyCustomPla: String
+ GUI_ModifyPLA(c: Controller, d: DataPresenter)
+ showGUIModifyPla()
+ update()
- showOutput()

Figura 10: Diagram of GUI_Develop class

For ease of viewing, several data fields related to the GUI_Design have been excluded.
In general the GUI is composed of:

- **JPanel**: for the division of the various Widgets into panels;
- **JButton**: simple buttons;
- **JTextPane**: panel containing a text area;
- **JTextArea**: panel containing a text area;
- **JTextField**: area where you can insert text;
- **JLabel**: simple text;
- **JComboBox**: to allow multiple options.

The class *GUI_Develop* is the main one that is started in the main. All other classes are generated after a particular event created by the user. *GUI_Pla* is instantiated when the user decides to create a PLA, and *GUI_ModifyPla* when the user decides to modify a PLA.

Events and Listener

For some of the present elements, mainly the JButtons, the listener has been defined.
When the user performs a certain action, such as pressing a button, the listener will see that the button has been pressed and a certain part of the code is executed.
In particular, the controller method, which triggers the chosen use case, will be called up.
The various component listener and actions are defined in the class constructor.

6 Extension of functionality

6.1 Preamble

The following section is intended to provide general guidelines to achieve a software modification with as few side effects as possible. The positioning of the classes as indicated is intended to respect the current structure of the product.

6.2 Modality

The addition of functionalities must be done without compromising those already present in any way. To this end, it is good practice to add new use cases to achieve your goal, without getting involved with existing ones. If it is necessary to extend an existing functionality, it is essential to pay particular attention during the test phase, making sure that the basic functionality is not affected in any way.

6.3 Adding a new entity

New entities can be added within the ”entities” package. It is a good idea to define the set and get methods to access their data fields.

6.4 Adding a new use case

New use cases can be added to the ”usecaseinteractor” package. The use case must display one input and one output interface. These must be placed inside the ”port” package. The input interface methods must be implemented by the use case.

6.5 Adding a new framework

Use cases may need to perform operations with the help of external libraries or frameworks. The use case should not communicate directly with them but only through interfaces. These must be placed inside the ”interfaceaccess” package. It will be the task of the class that integrates the external library to implement its methods. This class must be added inside the ”frameworks” package.

6.6 JAR executable generation

After you have made all the changes, you will need to generate a new jar file. To achieve this, the following plugin must be present in the pom.xml file:

```
<plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
                <archive>
                        <manifest>
                                <mainClass>fully.qualified.MainClass</mainClass>
                        </manifest>
                </archive>
                <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
        </configuration>
        <executions>
                <execution>
                <id>make-assembly</id>
                <phase>package</phase>
        <goals>
```

```
                <goal>single</goal>
        </goals>
                </execution>
        </executions>
</plugin>
```

Next, running the Maven command ”mvn clean compile assembly:single” will generate a jar executable inside the ”target” folder in the project root.

A Glossary

A.1 A

A.1.1 Action

An operation contained in a BAL suggestion. It can be translated, for example, as a function or as a method of a class.

A.1.2 API

Application Programming Interface - A set of procedures and functions that perform a given task and whose specifications form an interface.

A.2 B

A.2.1 BAL

See Business Application Language.

A.2.2 Business Application Language

Pseudolanguage that interposes itself between the natural language and the programming language. It allows the understanding of actions and objects necessary for APIs even for non-projectors.

A.2.3 BD

See Business Domain.

A.2.4 Business Domain

Aspects of the real world in which the software operates.

A.2.5 BDL

See Business Domain Language.

A.2.6 Business Domain Language

A set of domain-specific terms, consisting of nouns, verbs and predicates and their frequency of use.

A.3 F

A.3.1 Facade

Design pattern that defines an object that allows, through a simpler interface, access to subsystems that display complex and very different interfaces.

A.3.2 Feature

Desired functionality of a product that often includes various behaviors that the software assumes. Behaviors are defined through scenarios.

A.4 G

A.4.1 Gherkin

Standard and widely used format for writing use case scenarios.

A.5 J

A.5.1 Json

Acronym for JavaScript Object Notation, it is a format suitable for exchanging data between client/server applications. It is based on the standard JavaScript language but is independent of it.

A.6 N

A.6.1 NaturalAPI Design

NaturalAPI module that transforms features and scenarios into a BAL.

A.6.2 NaturalAPI Develop

NaturalAPI module that translates BAL into API in the chosen programming language.

A.6.3 NaturalAPI Discover

NaturalAPI module that extracts the BDL from documents related to the domain.

A.7 P

A.7.1 PLA

See Programming Language Adapter.

A.7.2 Programming Language Adapter

Set of rules that allow the Developer to generate APIs in a specific programming language.

A.8 S

A.8.1 Scenario

Specifies actions and steps that define the behavior of a software using a formal language.