# ALDA Web Application

A genomic tool for splicing junctions

Software Applications Course for Genomics, 2019

Dini Alice
Salerno Antonietta
Traversa Daniele
Valeriani Lucrezia

## TABLE OF CONTENTS

# ALDA Web Application
# Project Document

## 1. Introduction

We present our exam project report for the course *Software Applications* at 'Alma Mater Studiorum – University of Bologna'.
For the completion of this course, a distributed real-time system – a software application able to recognise, given a sequence of DNA, the boundaries between exons (the parts of the DNA sequence retained after splicing) and introns (the parts of the DNA sequence that are spliced out) – is specified, designed, built, and simulated.

Different tools have been employed for the realization of our software application, such as *Object-Oriented Programming* (OO) adopted in Python coding language for the implementation of our source code, through the system of formal classes and methods.

Additionally, the backbone for the development and realization of the web application that we have implemented is defined with diagrams built up using the *Unified Modelling Language* (UML), that is a common graphical language for specifying, visualising, building and documenting the artefacts of a software system, such as the source code, documentation, test results, etc.

Finally, we have applied some software engineering strategies to realize a dynamic, maintainable and accurate software product within the release deadline and the available human resources i.e. a team of four people. Among these, a combination of software model processes has been chosen, because the properties of a software directly depend on its development process. In addition, the design of our software is outlined in such a way to follow some GRASP design patterns which would solve recurrent problems that might occur.

## 1.1. System Overview

ALDA web application has been realized in order to attempt the specific purpose of splice junction's recognition, that are "superfluous" DNA sequences at the boundaries of each intron\exon, removed during the process of protein creation in higher organisms. After having recognised them, our software aims at classifying each individual boundary and return their unique categorization as "EI" (exon-intron), "IE" (intron-exon) sites or "N" (neither of them). In the biological community, IE borders are referred to as "acceptors" while EI borders are referred to as "donors" or "none".

In addition, an ideal user who is going to use our software for its own research might also check the number of individuals that exists in the different graphs, provided by the reference dataset exploited by the system or even check the performance of the system in use to verify its efficiency.

## 1.2. Software Process Model

The project is based on the adoption of a software design engineering strategy which aims at the organization of the software development.
Under our judgment, the most appropriate software process model to be employed is the *Rational Unified Process* (RUP): a hybrid model that brings together elements from all generic process models. RUP is usually described under three different perspectives that interact among each other: the *dynamic*, the *static* and the *practice* one.

I. Under the ***dynamic*** point of view, the RUP model is perfect to show the phases of the project development overtime in terms of iterations. In fact, our workflow can be described as a loop made up by four principal well-defined canonical steps, which are iterative too:

   - The **inception** phase consists in identifying all external entities (people and systems) that will interact with the software and defining these interactions. This information is also used to assess how an ideal user interacts with the system by using our software.

   - The **elaboration** phase is characterized by the understanding of the problem domain. The team establishes an architectural framework for the system and develops the project plan. In this phase, therefore, we have realized which are the components that interact in the system and defined such interactions by developing the component diagram.
     The basic requirements have been analysed and modified by us, including the additional components and features introduced in the architectural framework, to satisfy the original requirements that, by consequence, have not been altered but only updated. The aim is to improve the way our software is going to please the needs of the user. At the end of this phase, we achieved a specific model for our system in terms of UML use cases, in order to give the foundation plan for building of the entire software application. In this sense, the team has exploited the plan-driven management skills of the RUP process model, as they have been resulted useful to know in advance when a specific step in the activity workflow should have been done.

- The **construction** phase involves system designing, programming, and testing. This is the step in which we have finished the component diagram and designed the activity and the sequence diagrams.
  Then, we constructed the class diagram in parallel with the modification of the basic source code that has been provided to us, in order to structure it in such a way to follow the connection rules of inheritance, thanks to the adoption of object-oriented programming.
  Actually, the different parts that arrange the system are developed and integrated during this phase at the same time, meaning that the longest and the most repeated iteration is actually the construction step itself. Our understanding of the software has been, in fact, continuously updated, so every time a new version has been set up, each part of the system has been revised as a consequence.
  On completion of this phase, the team has developed a working software system together with its associated documentation.

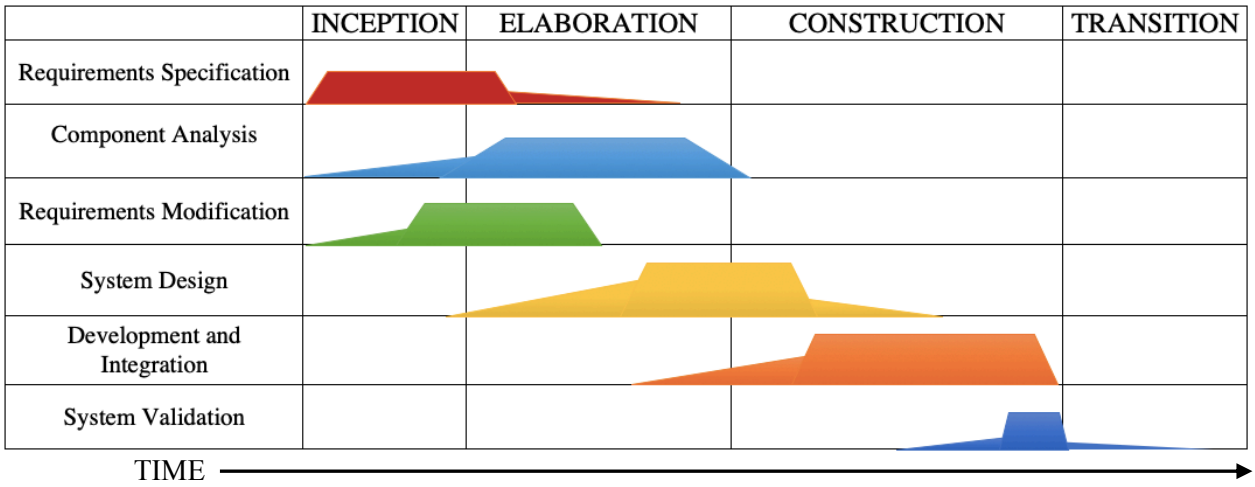
- The **transition** phase consists in the deployment of the final product by moving the system to the user community and making it work in a real environment.
  However, since the project has been developed for an exam, we actually do not have a real user community to validate the correct functioning of the software in the most appreciated way. By consequence, the weight of the transition phase is very low in our software process model.
  On completion of this phase, a documented software system working correctly in its operational environment is obtained.

II. Under the ***static*** point of view, the RUP model provides a focus on the process activities that have been enacted. In our project the activities of the workflow adopted for the development of the entire web application follow a reuse-oriented engineering strategy. This kind of model is centred on the integration of reusable exogenous software components, rather than developing them from scratch, by re-implementing or reinventing functionalities. In this way, the amount of software to be developed is reduced, leading to a faster delivery of the software, reducing costs and risks.

Writing good software, in fact, is a challenging problem and any re-implementation should be avoided: all software contains bugs but well used and maintained ones tend to contain fewer. Computational Biology and Bioinformatics environment is, in addition, an enormous field that requires structured paradigms for accessing data and algorithms written in other languages and systems.

The static steps that constitute our reuse-oriented approach cover all the phases already highlighted in the description of the RUP model dynamic perspective, and the occurrence of each of them is actually spanned throughout the lifetime of more than one of the RUP dynamic phases. The team has created a table to describe the interactions within the dynamic and static points of view under which the RUP process model has been adopted (*Table 1*).

*Table 1: RUP software process model and the interactions within its dynamic (rows) and static (columns) perspectives*




The main aspects of the RUP static perspective under the reuse-oriented approach are:

- **Requirements Specification:** the requirements are read by the team and a first interpretation is given to them, but the understanding of how they should be formalized in the project will evolve in time during all the iterative phases that constitute the development process of our system.

- **Component analysis:** given the requirements specification, a search is made for components to implement that specifications. However, there was no exact match between the basic requirements and the idea our team had regarding the components that should have been used in the most efficient way possible to provide all the functionalities and features required for the realization of the software. Therefore, some new components have been included in the design of our web application: these exploit external resources like stand-alone software systems as the GENO Dataset, web services as SPARQL, and collections of objects that are developed as a package to be integrated with a component framework such as *scikit-learn, flask, panda, matplot library, numpy,* etc.

- **Requirements modification:** the requirements are re-analysed to reflect the new components. A sketch of the use case diagram is designed to formalize the basic requirements specification, to describe the available interactions of the user with the system. Wherever modifications are impossible, the component analysis activity may be re-done to search for alternative solutions, and the requirement specification, together with the use case diagram, are updated another time until a definitive solution is accepted by all the members of the team. In this way, we end this step with a defined idea of the components needed to attempt the requirement specification that we have updated.

- **System design:** the framework of the system is designed de novo by using the UML visualization language. Our team begins this step by taking into account the components which organize the framework. The use case diagram and the component diagram are completed, and the activity and sequence diagrams are realized on the basis of the previous ones. The realization of all the diagrams has been done in parallel in order to grant an iterative process development among the different phases. At the end of this step, the class diagram is sketched.

- **Development and integration:** the components that cannot be externally procured are developed and integrated with the basic source code that has been provided to the team in such a way that it reflects our diagrams. Finally, the class diagram is updated as long as the source code is modified each time. The final outcome of this phase is a document written in Python programming language containing the source code of the implemented software, structured using object-oriented programming.

- **System Validation:** all the methods in the classes are tested through Unit Testing using the *unittest* Python module: the final result of this phase is an additional Python document listing all the tests which have been run to assess the robustness of the code.

III. Under the ***practice*** perspective, the team adopted a set of good practices to have an efficient software application. Usually, the practice perspective of the RUP software process model provides some basic guidelines that can be followed to construct the system. Each of these methodologies are not mandatory.
The team has chosen which are the best ones that can be applied:

- **Develop software iteratively:** this practice suggests the team to build different increments of the system based on the costumer priorities. Since we did not have the user at our disposal, we pretended to be hypothetical users thanks to our knowledge background in the fields of biology and programming. Hence, each time we realized increments of our system, we verified if they reflect the priorities of an ideal user.

- **Use component-based architectures:** this practice is seen both in the employment of a reuse-oriented strategy, and in the choice of Python, which supports object-oriented programming: these two give the basis for a component-based framework.

- **Manage requirements:** since in the early stages of the process, we have identified the basic requirements, and during the development the team updated them, and verified if these changes were suitable to the software that was into building.

- **Visually model software:** in this case we graphically modelled our software using UML diagrams that, during the workflow, were continuously updated.

- **Verify software quality:** we make sure that our software fronts the needs of the user by checking its quality by unit testing.

- **Control changes to software:** by analysing the process of development that we conducted, we were able to manage changes in a very efficient way due to the iterative pipeline we adopted. Therefore, in the moment we modified one aspect of the software, we were able to reduce the possible side effects on the other sections of the system since we developed them mostly in parallel.

After all the phases are accomplished, the team has realized this project document that describes the procedure adopted to realize the software application step-by-step.
In the following part, the requirements of the software application are discussed, along with the user stories that have been provided to us. Then, the architecture design, implementation and testing of the software product are shown. A focus on the maintenance of the system is also faced in a dedicated part.

## 1.3. Teamwork

In this section of the document, we want to give some information about how the members of the team have worked in the development of the ALDA Web Application.
Team work is a fundamental part of software development because a good cooperation between the members usually takes to the development of an efficient product. In fact, in this case, the rigorous rules of software engineering have a secondary role since we focused on the development of good interactions between the components of the staff.

In our case, we organized the work in a way that every member could enter in contact with each part of the system, which means that everybody contributes to the design development of the UML diagrams and on the implementation and testing of the source code.
We worked together also in the drafting of the project document, by sharing ideas and comments. In this way we were able to write a homogeneous document that reflects the coherence of the Web Application in terms of design and code.

Moreover, since we worked always together as a team, during our meetings we took on some strategies inherited from Extreme Programming (XP) approach to make the work highly efficient. Among these, we adopted the *pair programming* (navigator-driver strategy): we always worked in couple where one person was responsible for checking the performance of the other. Every time each couple finished its work, it shared the results to the other members of the team.
Another one was *collective code ownership*, an explicit convention that every team member is not only allowed, but in fact has a positive duty, to make changes to "any" code file as necessary: either to complete a development task, to repair a defect, or even to improve the code's overall structure. All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable and reduced the risk that the absence (or unavailability) of a developer will stall or slow work, increasing the chance that the overall design results from sound technical decisions, rather than from social structure, as in Conway's law.
Finally, the *on-site customer practice* is another XP strategy that resulted useful for the evolution and maintenance of our software. Since we had not the chance to face actual end-users, the customers were represented full-time by the members of the team itself to validate the accomplishment of the requirements.

Team members learned from one another, improving on their current skill set, and advance the overall team performance. Different perspective of the team members can drive creativity and innovation. When working in teams, developers get continual feedback and can improve their processes, hence the overall product quality is boosted; this feedback is quick and necessary changes are made overtime. Code quality standards are also kept high as code review processes go through one or more team members.

# 2. System structure and development

## 2.1 Requirements

The first step that the team had to face for the development of our web application was to identify the requirements of the software and understand which functionalities our system should have in order to allow the user to perform different actions. To do that, we analysed three user stories – brief texts written in natural language – from which we extrapolated the requirements that our software should front.

**USER STORY 1**

The system uses the GENO *linked open* Dataset to gather the knowledge base which our system will interact with. This latter is organized in three different graphs: training graph, test graph and validation graph. The GENO Dataset contains primate splice-junction gene sequences represented as RDF, classified with respect to the classes: exon-intron boundary, intron-exon boundary, and neither (i.e. unknown). GENO can be queried by the system via SPARQL.

Our understanding of this user story is formalized in the elaboration of the following requirements that our system should be able to deal with:

- The system should use the GENO Dataset as knowledge base, which is a stand-alone software system that can be accessed via SPARQL web service, directly from the introduction of a SPARQL query in the source code of our software.
- The system should be provided of a component that is able to query the GENO Dataset from which it extrapolates, as knowledge base, instances of the training graph, test graph and validation graph.
- One component of the system, the GENO Dataset Reader, should be able to convert the RDF format of the instances from the GENO Dataset in a format that can be understood and used by the other components of the system *(.json*).

**USER STORY 2**

The system learns how to classify splice junctions by using the training graph available from the GENO Dataset. The classification algorithm can be one among the inherently multi-class algorithms known in literature. The classes for the classification are *exon-intron* boundary, *intron-exon* boundary, and *neither* (i.e. unknown). Additionally, the performance of the system is assessed by using sequences of splice junctions available from the test graph of GENO. The performance is recorded in terms of confusion matrix with additional metrics based on *precision, recall, and f-measure*. Finally, the system provides a prediction functionality that, given a specific sequence gathered from the validation graph of GENO, returns its classification with respect to the classes IE (intron-exon boundary), EI (exon-intron boundary), and N (neither).

Our understanding of this user story is formalized in the elaboration of the following requirements that our system should be able to deal with:

- The system must be able to recognize splice junctions
- The system must be able to train the classification algorithm (*Gaussian Naïve Bayes* from *scikit-learn*) by using instances form the training graph of the GENO Dataset
- After being trained, the system is able to check and collect the performance of the algorithm by using sequences of splice junctions available from the test graph

- The software should be able to gather the performance of the classification algorithm creating a confusion matrix, built up following additional metrics based on precision, recall and f-measure that have been collected by the system
- The system uses the (trained) classification algorithm to classify DNA sequences of splice junction from the validation. The classification process may produce three possible results: exon-intron (EI), intron-exon (IE), and neither (N)

**USER STORY 3**

The system provides a web-based user interface (UI) running on a web server: it consists of HTML pages that enable the interaction between the system and the user. The UI provides the statistics (i.e. number of individuals) of the training, test, and validation graphs from the GENO Dataset. The statistics might be visualised when the user clicks on a dedicated button. The UI provides the details about the performance of the algorithm computed over the test graph from the GENO Dataset. The UI allows a user to select a specific instance from the validation graph of GENO and return its classification as output.

Our understanding of this user stories is formalized in the elaboration of the following requirements that our system should be able to deal with:

- The system must be able to collect the statistics of the training, test and validation graph. The statistics are represented in terms of number of individuals per graph.
- Since our system has been realized in such a way that a user should be able to interact with it, the system provides him or her a User Interface (UI) made of different HTML pages through which it is possible to perform different actions, by clicking on three different buttons:

    · Statistics: the user can visualize the statistics as number of individuals from the training, test and validation graph;

    · Performance: the user can visualize the performance of the classification algorithm over instances from the test graph;

    · Classification: the user can select a specific instance from the validation graph and optionally decide to classify it by clicking on another dedicated button ‘Classify’ that will result in the opening of another HTML page that shows the classification of the splice junction sequence.

## 2.2. Software architecture design

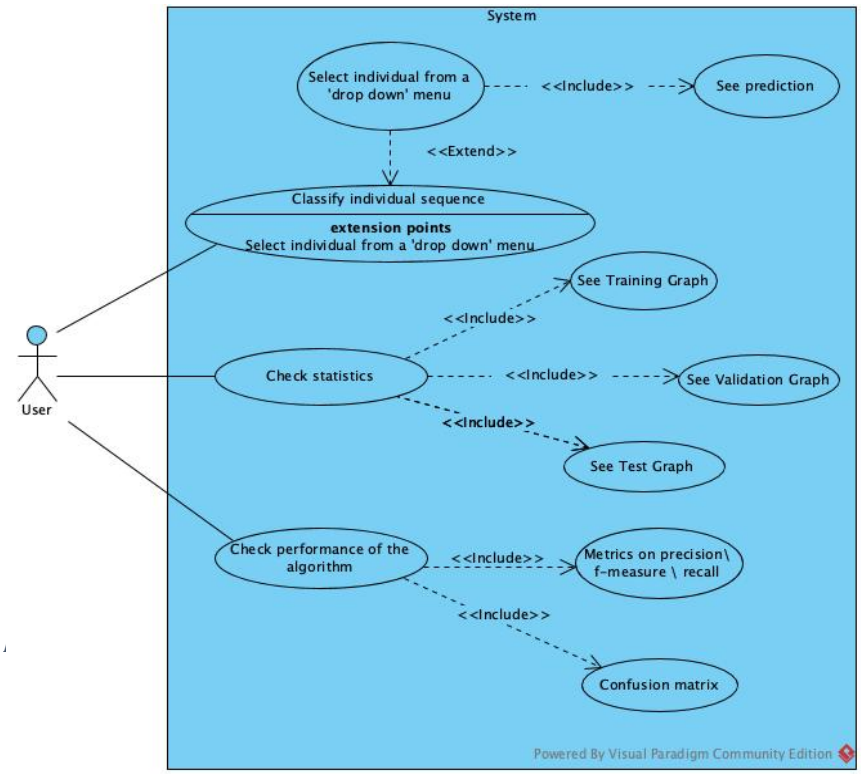
The software architecture design has been developed using graphic diagrams modelled with *Unified Modelling Language* (UML). There are a lot of UML diagrams available to give a structured design but in our project, we specify the following ones:

- Use-case diagram
- Component diagram
- Activity diagram
- Sequence diagram

We have also used UML to develop the class diagram, which concerns the architecture design of our source code and it is based on all the diagrams described in this session. Consequently, we have decided to present it in Section 2.3 rather than in the following one, because it is more related to the implementation and integration of the software. In our working line, in fact, we have studied the code and the class diagram in parallel to reduce any kind of incongruence.

**USE CASE DIAGRAM**

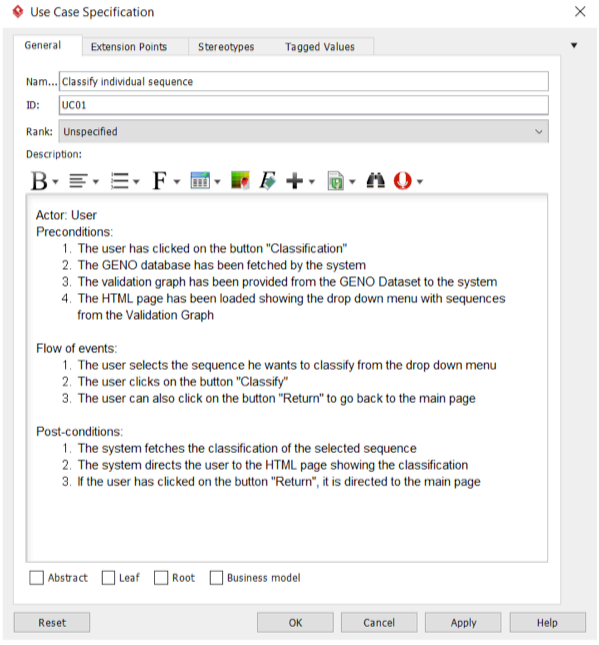
The use case diagram is a type of UML diagram that shows the different ways through which the user interacts with the system. After having identified the requirements, we interpreted that the user is able to communicate with the system by means of a web user interface which allow the user to deal with it in three ways: asking for the classification of an individual from the validation graph; visualizing the performance of the system; checking the statistics of the individuals from the training graph, test graph and validation graph.
Since the actor - that in our system coincides with the user - might perform three different actions, we formalized them in three main different use cases in our diagram.

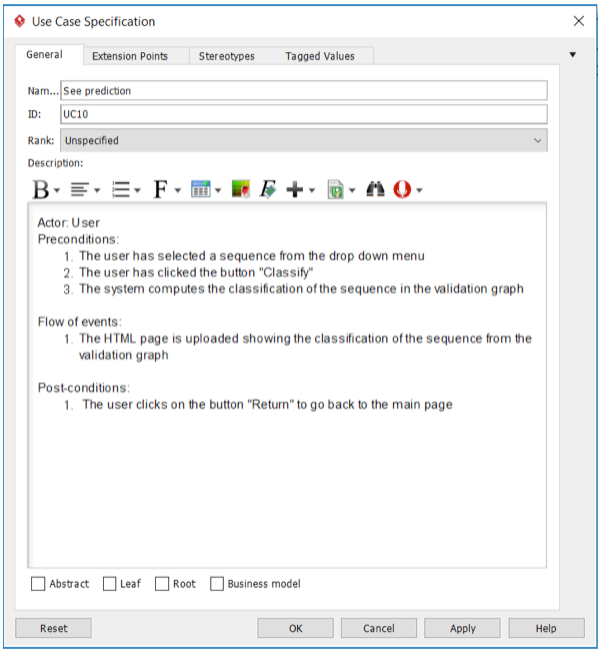


Let's analyse in detail the three principal use cases.

## 1. Classify individual sequence

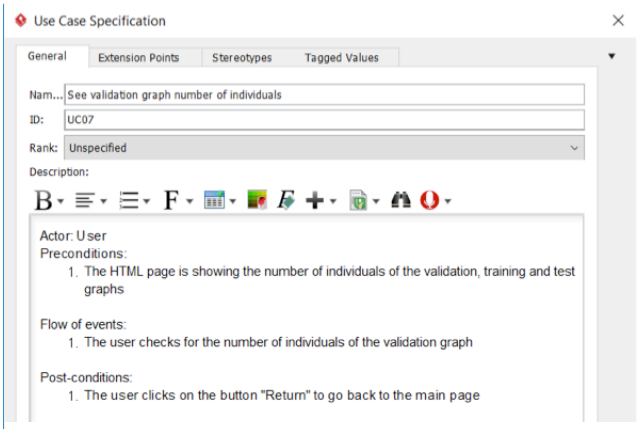
In the moment in which the user starts to use the software, the system will provide an HTML page in which the actor can choose to perform the classification of one of the instances of the validation graph, shown in the form of a "drop down" menu. As a result, if the user decides to see the classification of the selected instance, the system will return the prediction of its classification as exon/intron (EI), intron/exon (IE) or neither (N) in another HTML page of the user interface, provided by the web application.
The reason why the team has thought the Select individual from a 'drop down menu' use case as an *extension* of the main use case Classify individual sequence is, therefore, that the decision of the user to select an individual sequence from the validation graph and ask for its classification prediction is optional. He/she might, in fact, be free to go back to the HTML homepage of the user interface whenever he/she decides to do not ask for the classification anymore, as there is not any constraint that tells to proceed in a particular direction. In this way, we aim at preserving the independence of the actions.
On the other side, in the moment in which the user clicks on a dedicated button in the Select individual from a 'drop down menu' HTML page to ask for the classification prediction of the chosen individual, an *include* association is used to remark the straight relationship between this use case and the See prediction one. The user, in fact, once having decided to classify the selected individual, is directly brought to a secondary HTML page that shows the predicted classification to the user interface.

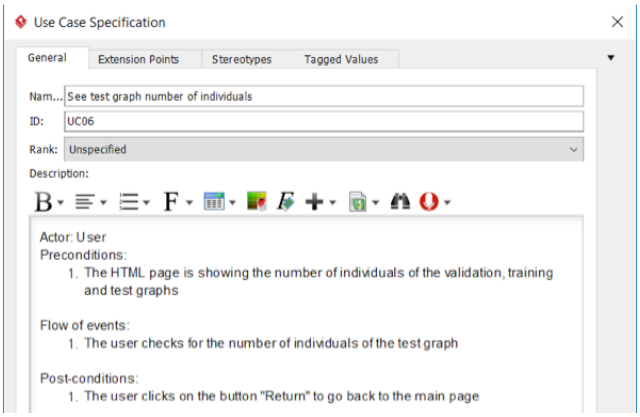


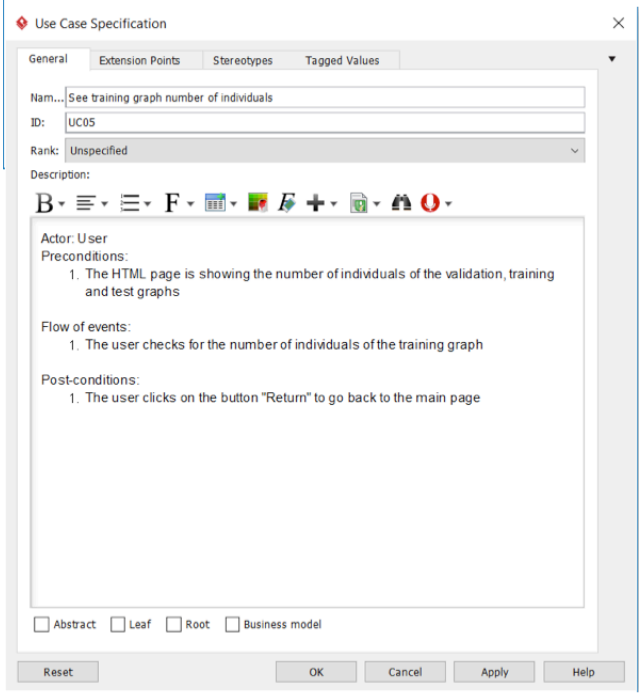



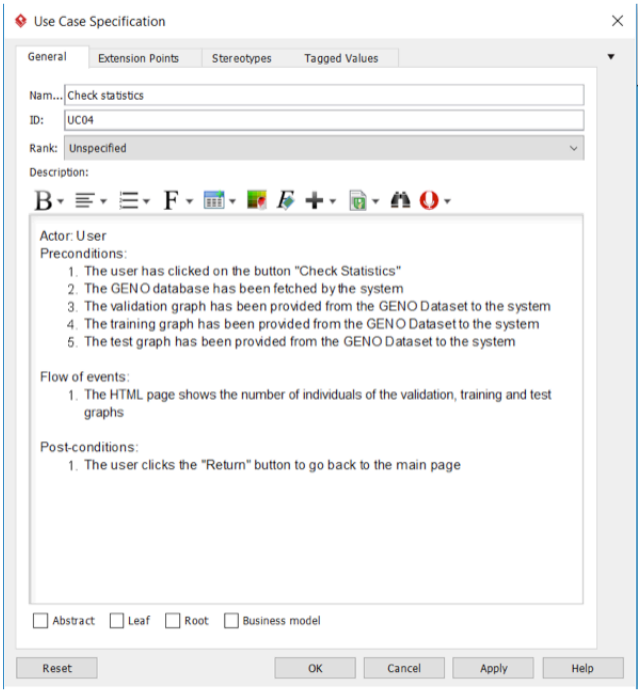
## 2. Check statistics

When the user accesses the HTML homepage to use the software, the system will enable him/her to choose to check the statistics of the individuals of the training, test and validation graphs, visualized as number of individuals per graph, after clicking on a dedicated button which coincides with the use case Check Statistics. As a result, a specific HTML page is shown in the user interface.
In the moment in which the user selects the dedicated button for the statistics on the web interface, the system has to communicate with the training, test and validation graph in a dependent manner. To formalize this concept, the team decided to visualize the association between Check Statistics use case and the different graphs using the *include* association, since the parts of the system responsible for the statistics are obliged to communicate with the individual sections associated to the three graphs.



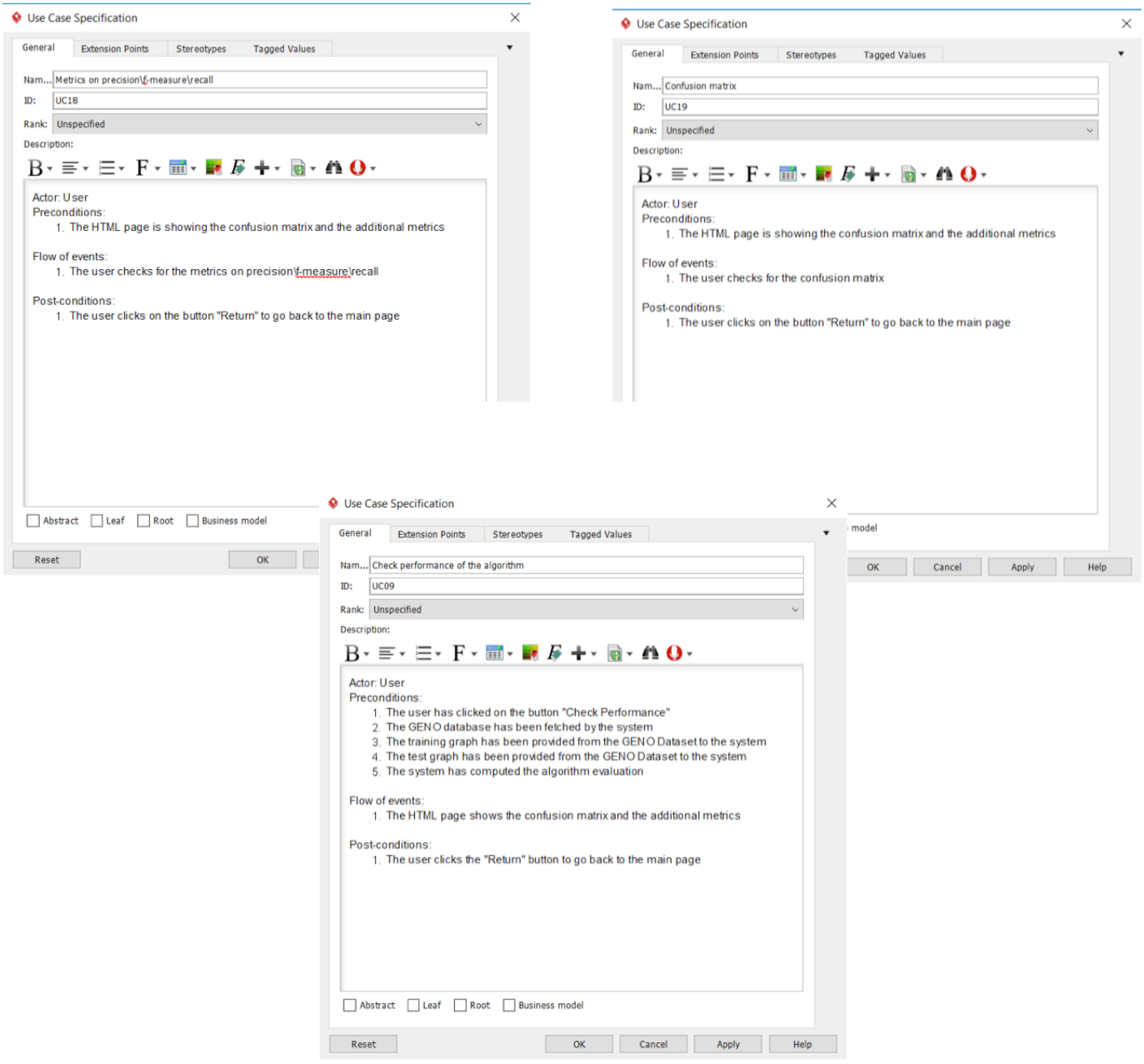



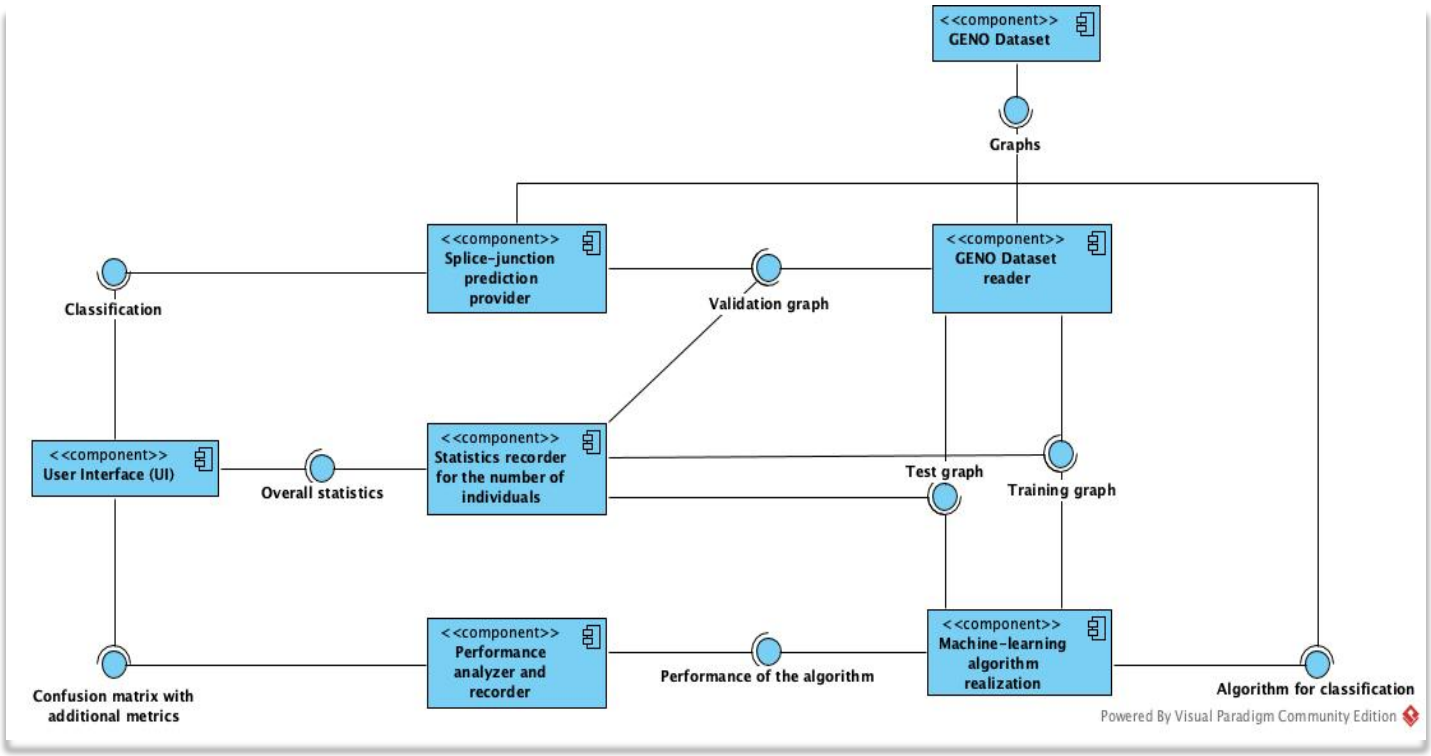



## 3. Check performance of the algorithm

The actor is able to visualize the performance of the classification algorithm as a confusion matrix designed using additional metrics on precision, f-measure and recall through a dedicated HTML page, provided by the User Interface, by clicking on the dedicated button on the HTML homepage. Since the system has not a unique component that allow it to conclude this action, it must interact with the other ones in a dependent manner that guarantees the web application to collect the performance of the algorithm and to show it in a confusion matrix. In our use case diagram, we formalize this concept using the *include* association that connects the main use case Check performance of the algorithm with the secondary use cases Metrics on precision, recall and f-measure and Confusion matrix.



## COMPONENT DIAGRAM




In a reuse-oriented approach, as the one chosen by our team, the component analysis is a very important and powerful step that allows to understand the structure of existing systems and the relationships between them, in order to facilitate the building of new ones. By consequence, the component diagram is an essential visualization tool in this step, because it is very useful to get a general idea of how the architecture of the system should be implemented, by giving a bird's-eye view of its behaviour. A component, in fact, is just a module of classes that represent independent systems or subsystems with the ability to interface with the rest of the system, so it is only the framework around which the class diagram is going to be developed to properly generate the corresponding source code for the software application. It is very common in an object-oriented programming approach that the component diagram allows a developer to group classes together based on common purpose, so that it is possible to look at a software development project at a high level.

First of all, a new component has been inherited from exogenous resources to be included in the design of our web application i.e. the GENO Dataset, a stand-alone software system. This component is able to interact with our software through the mediation of the GENO Dataset reader, by providing the test, validation and training graphs - stored in the GENO dataset - to our application, because they are required as knowledge base to satisfy all requirements. These graphs are directly required by three endogenous components in the system:

- Machine-learning algorithm realization, a component whose main responsibility is providing to the Splice-Junction prediction provider the algorithm that it needs for the classification of individual boundary sequences.
  By definition, a machine-learning algorithm is concerned with the construction and study of systems that can learn from data (i.e. examples and experience) without being explicitly programmed.

For the purposes of our system, the machine-learning component performs a *supervised learning,* which means that it allows a system to build a model that connects a given input to a given output by training a model to generate reasonable predictions for the response to new data. More specifically, it really undergoes a classification problem because the output variable is a discrete value such as IE, EI or N.
In our system, both inputs and outputs of the Machine-learning algorithm realization component are supplied by the GENO Dataset reader, which provides the training graph. The test graph will verify that the classification algorithm works correctly.
On the other side, the component under study also assesses the performance of the algorithm and provides these data to the Performance Analyser, charged by the system to compute the *confusion matrix*: a specific table layout that allows visualization of the performance of an algorithm. This will be finally provided to the User Interface whenever the user will be pleased to look at the performance of the system.

- Splice-junction prediction provider, a component that requires the algorithm for classification and the validation graph interfaces: it enables the system to provide a prediction functionality that, given a specific sequence gathered from the validation graph, returns its classification by using the trained classification algorithm. The output is finally provided to the User Interface whenever the user will be pleased to look at the classification prediction of an individual sequence.

- Statistics recorder for the number of individuals, a component that requires the three graphs - gathered from the GENO Dataset via our system's GENO Dataset reader – to analyse the numbers of individuals found in each of them (i.e., the statistics), and provide them to the User Interface, to allow the user to look at the overall statistics whenever it will be pleased to do it.

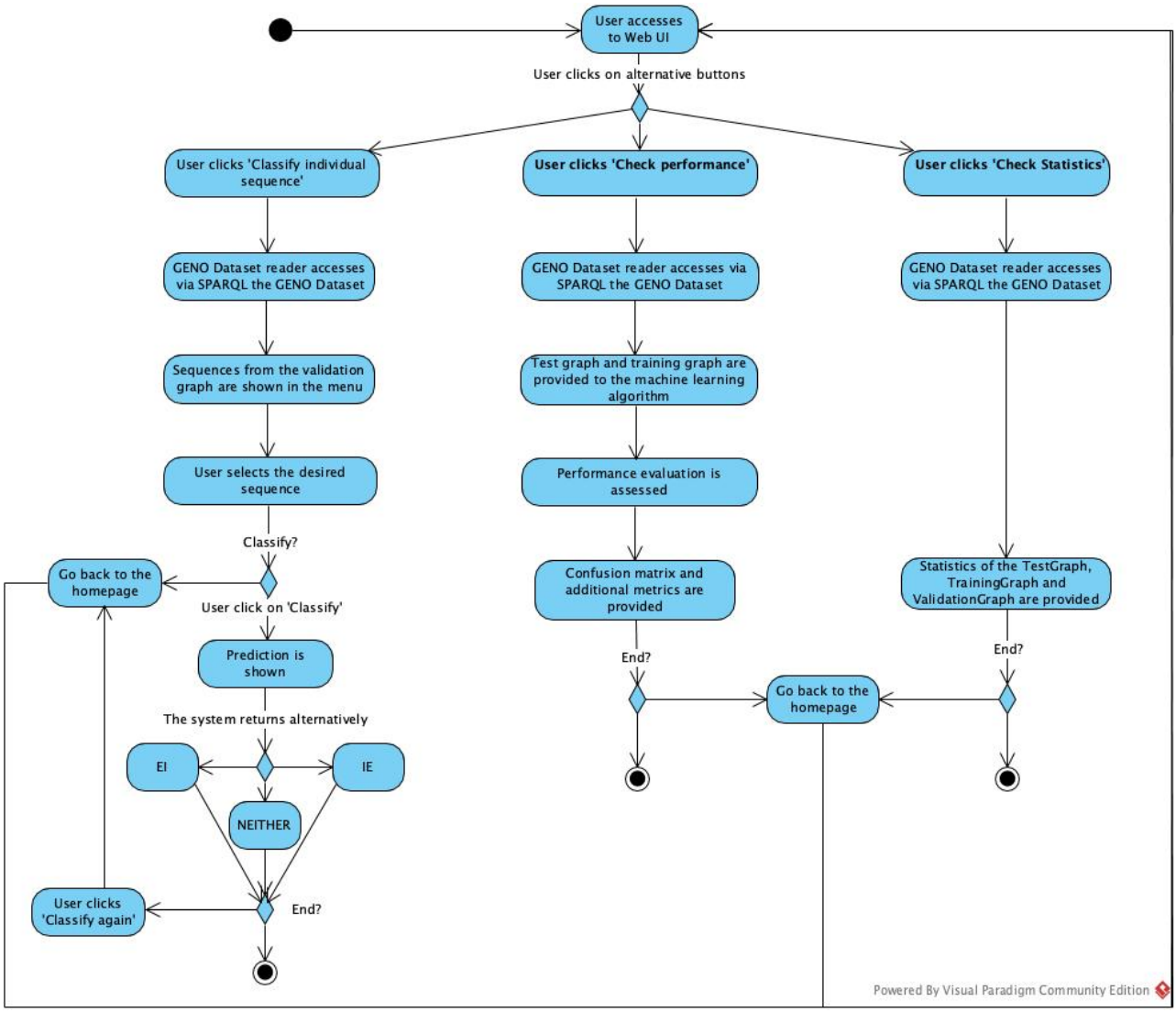
## ACTIVITY DIAGRAM

An activity diagram is used to represent the flow of the activities performed by the system. An activity is an operation of the system, and many ones are usually organized in a consequential manner. Basically, the system uses a set of interconnected activities to please a specific functionality put at user disposal.
As we described the existence of three use cases, i.e. three functionalities provided to the user, we thought that the system should follow three different sets of interconnected activities through which it makes different actions available to the user.
When the software enters in function, it provides to the actor a unique HTML page where the user can click on one among three different buttons, depending on the specific action that the user wants to perform:

1. Classify individual sequence
2. Check performance
3. Check statistics

Secondly, the activities' flow of the system develops in three different directions. Therefore, we can say that the first action performed by the system (User accesses to Web UI) is common to the other three functionalities.

Let's see which activities the system will perform per functionality.




## 1. Classify individual sequence

The actor clicks on the dedicated button in order to ask to the system the classification of a junction sequence. After that, the system queries the GENO Dataset via SPARQL by means of a component called GENO Dataset Reader, and it fetches individuals from the validation graph. At this point, the web application will redirect the user to an HTML page where the actor can select one individual from a “drop down” menu. The system is now able to perform the classification and it will return its prediction to the user in another HTML page. The selected DNA sequence will be categorized in three different ways, exon/intron (EI), intron/exon (IE) or neither (N), depending on the kind of sequence that the user has selected. After the first run of classification, the system offers different options: take back the user to the homepage where he or she is free to perform another action, or it can drive the user back to the classification page with the “drop down” menu, giving him or her the possibility to select another sequence, and repeat the whole process.

### 2. Check Performance

The user clicks on the dedicated button to check the performance of the classification algorithm. The system now is able to query the training graph and the test graph via SPARQL from the GENO Dataset using its GENO Dataset Reader component. The training graph and the test graph are provided to the Machine Learning component that will train the classification algorithm using instances from the training graph. Then, the algorithm will be tested on the instances from the test graph, and the performance is collected by a component of the system. After this process, the performance of the classification algorithm is formalized in a confusion matrix, and the system will provide it to the user, along with additional metrics by means of another HTML page. At the end, the system offers the possibility to go back to the homepage.

### 3. Check statistics

The user clicks on the dedicated button to visualize the statistics of the training, test and validation graph. At this point the system queries the training, test and validation graph from the GENO Dataset using the component GENO Dataset reader. The statistics of each graph are collected by the system as number of individuals. Once collected, the statistics of the three graphs are provided to the user by means of an HTML page. At the end, the system offers the possibility to go back to the homepage.

## SEQUENCE DIAGRAMS

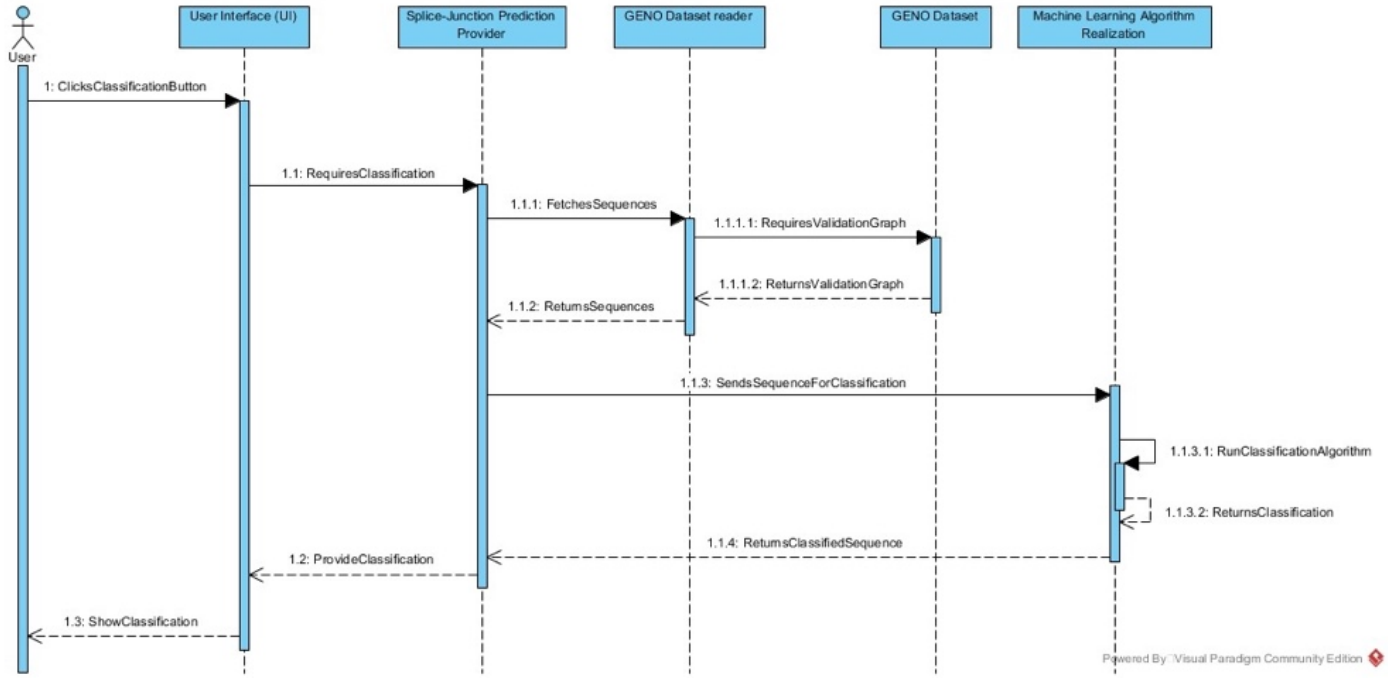
Sequence diagram is one kind of UML diagram which focuses on lifelines and on messages that are exchanged between objects and processes in a system. The purpose of the sequence diagram is to describe how, and in which order, a group of objects like a component, sends messages with other elements to achieve the realization of a specific requirement. In this idealized situation, the team has to develop a sequence diagram per use case identified, so that it can formalize the communication network between the components that are needed for the realization of each use case. Since we identified three different use cases, we developed three sequence diagrams.

Let’s see them in detail.

### 1. Selection of an instance from the validation graph for classification

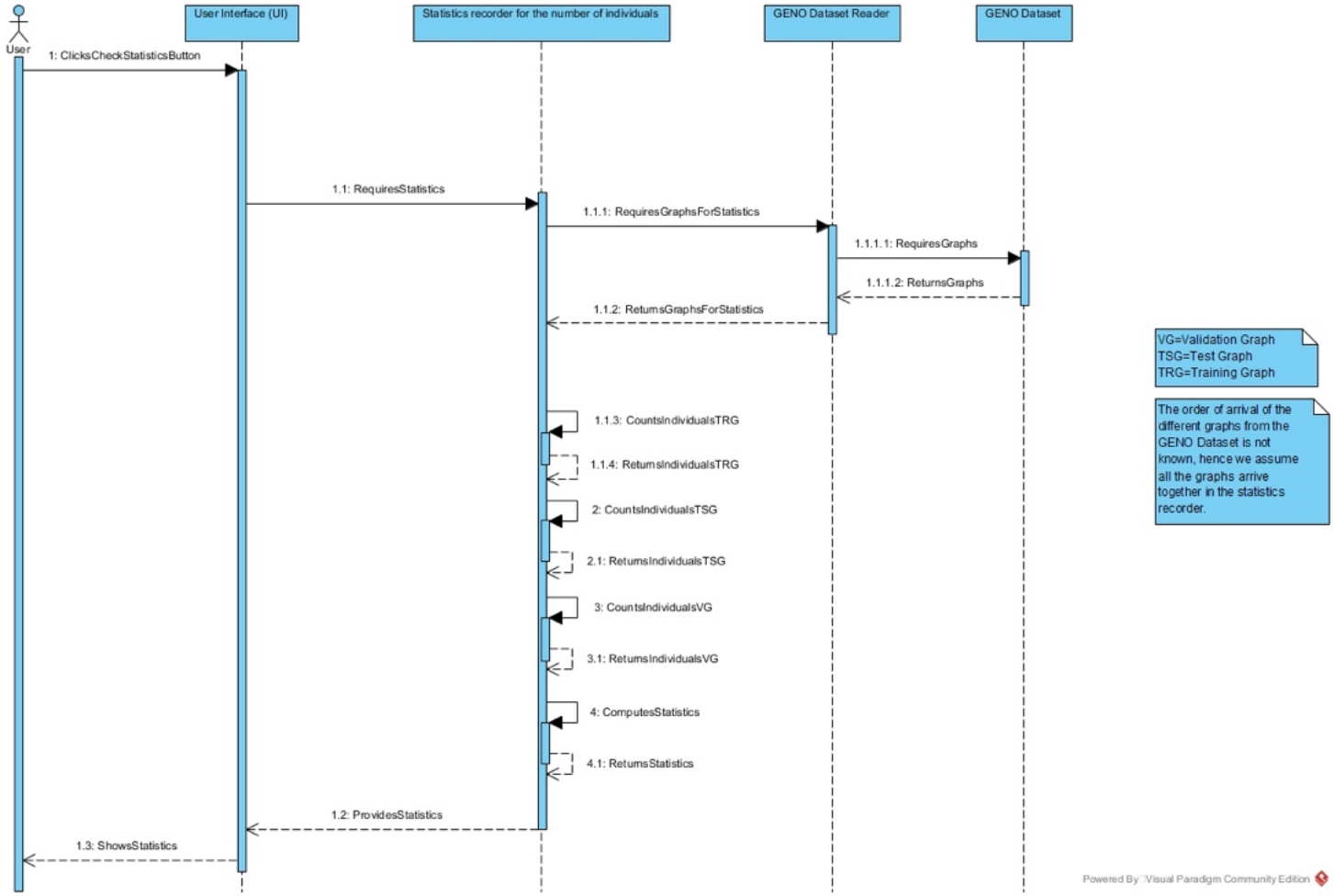
Following the development of this use case overtime, we can firmly confirm that the flow of events starts in the moment in which the user clicks on a dedicated button, that is provided by the user interface component through an HTML page; this makes the User Interface the first interacting component.
The User Interface needs to enter in contact with the Splice-Junction Prediction Provider, since it is the component which has to make the classification available to the user.
Actually, up to now the component does not have any sequence at its disposal, so it needs to import them from the validation graph of the GENO Dataset. To do that, the prediction provider component utilizes the GENO Dataset Reader. The latter will be asked to fetch the sequences from the dataset, and once it receives the message, it will ask the validation graph to the GENO Dataset via SPARQL.
Now the dataset will answer to the request by providing the validation graph to the GENO Dataset reader that will consequentially send the sequence to the Splice-junction prediction provider.
At this point the system is ready for the classification. As we specified in our component diagram, the Splice-Junction Prediction Provider is just responsible of making the classification available to the User Interface, but it does not perform the classification. The collected sequence is sent to the Machine Learning Algorithm Realization: this component contains the necessary knowledge to classify a splice junction DNA sequence such as the trained classification algorithm.

Now, the Machine Learning component runs the classification algorithm, returning the classification. The latter is sent form the Machine Learning component to the Splice-junction Prediction Provider, which is able to provide the classification to the User Interface.
Finally, the user interface will show the classification to the user by means of another HTML page.




## 2. Checks statistics

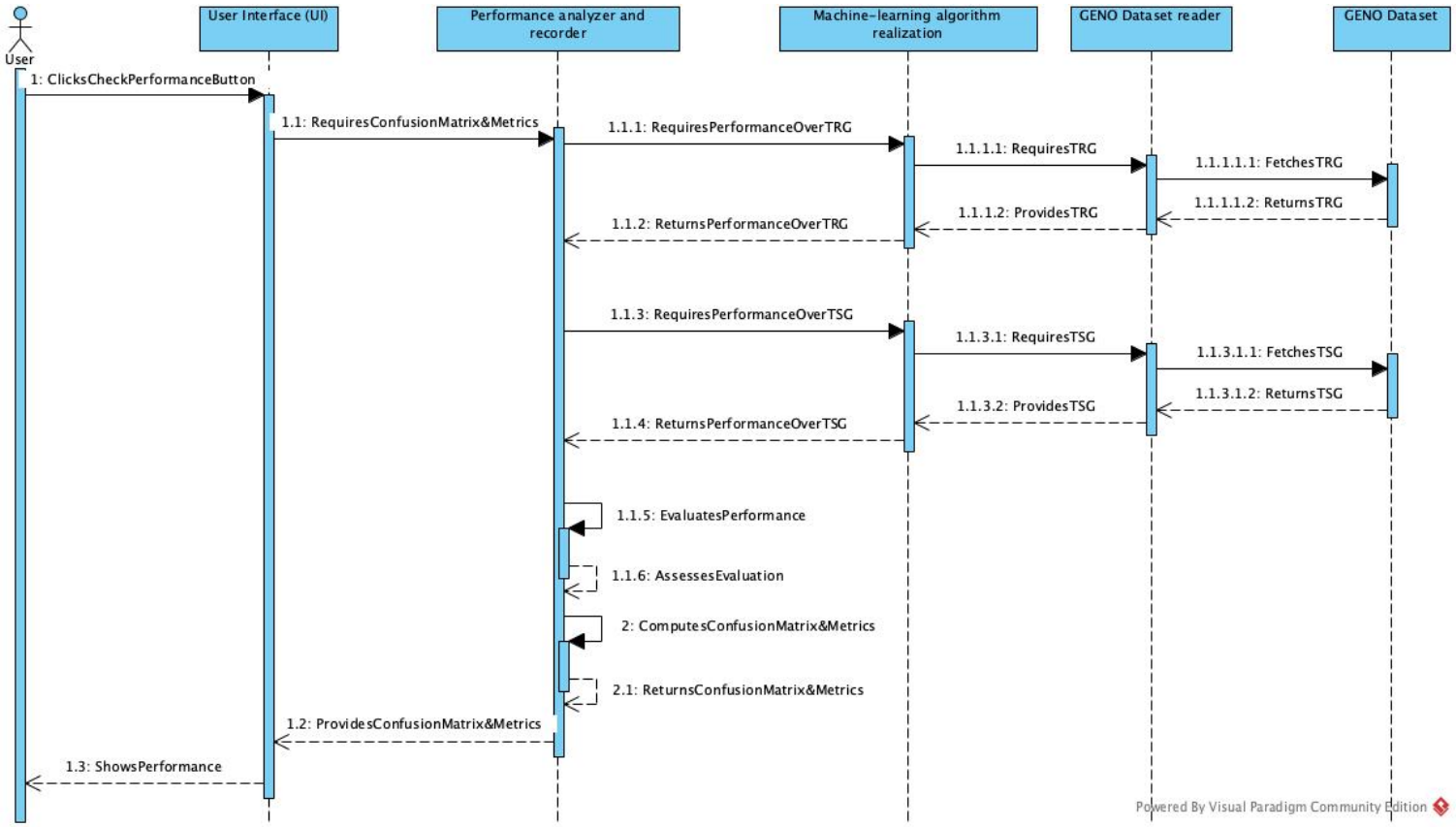
Analysing this flow of interactions, we immediately notice that the functionality formalized by this use case starts in the moment in which the user clicks on the dedicated button for checking the statistics of the training, test and validation graphs. Hence, also in this case, the flow of events starts with the User Interface. The latter enters in contact with a component called Statistics recorder for number of individuals to gather the statistics. In this case, the Statistics recorder does not have the information of the three graphs at its direct disposal, so it requires the knowledge from the GENO Dataset by means of the GENO Dataset Reader, that formalizes the request to the Dataset via SPARQL. Once the Dataset receives the request, it replies by providing the training, test and validation graph for statistics to the statistic recorder. In the reply flow of events, the GENO Dataset Reader plays the role of "intermediate" between the Dataset and the other components of the system. Now the component counts the number of individuals per graph, which are finally provided to the User Interface component that will then show the statistics to the user, by means of another HTML page.




## 3. Check performance of the classification algorithm

In this flow of events we observe that the first component of the system which the user interacts with is, again, the User Interface component, that provides a HTML page to the user where he or she can choose to classify a junction sequence, by clicking on the dedicated button. At this point, the UI asks the Performance analyser and recorder to produce a confusion matrix and to provide additional metrics, representing the performance of the system over the instances from the test graph. So, the Performance analyser and recorder component asks to the Machine Learning Algorithm Realization component for the performance over the training of the classification algorithm. The Machine Learning component asks the GENO Dataset for the training graph by means of the GENO Dataset reader, that fetches the graph via SPARQL. The GENO Dataset answers to the request by providing the training graph to the Machine Learning component again via GENO Dataset reader. Then, this last component returns the performance to the Performance analyser and recorder.
We can observe a similar flow of events regarding the performance of the classification algorithm over instances from the test graph. So, the performance analyser requires the performance of the algorithm over the test graph of the Machine Learning component. The latter fetches the test graph form the GENO Dataset by means of the GENO Dataset reader component, that queries the graph via SPARQL. The dataset answers to the request providing the test graph to the Machine Learning component though the mediation of the GENO Dataset reader.

Now the Machine Learning component returns the data about its performance over the test graph to the Performance analyser.
At this point, the component evaluates the performance information that it has received, and it uses the result of this evaluation to construct the confusion matrix. Finally, the metrics and the confusion matrix are provided to the User Interface and shown to the user by means of another HTML page.



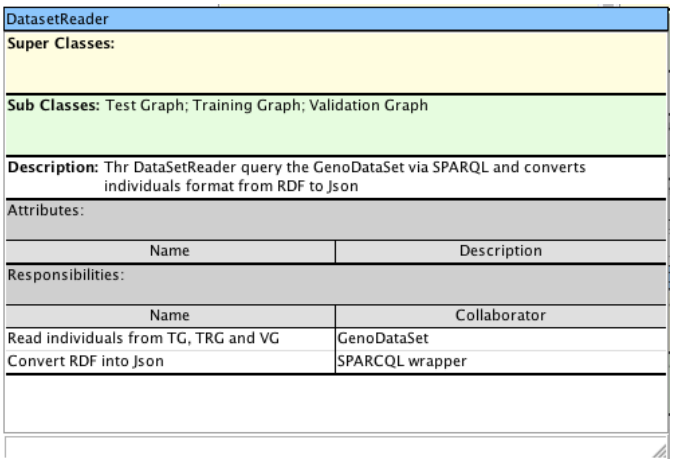
## 2.3 Software implementation

A class diagram shows a set of classes, interfaces, collaborations and their relationships. Class diagrams involve global system description, such as the system architecture, and detail aspects such as the attributes and operations within a class as well. The most common contents of a class diagram are:

- Classes
- Interfaces
- Collaborations
- Dependency, generalization, and association relationships

Our software is deceptively simple: although a user only needs to press a few buttons to receive the classification or check the desired performances and results, there are many layers of computation that a reliable and efficient genomic tool must pass through to prevent leaky data and provide valuable information to researchers. The various classes of our system are illustrated by this easy-to-read diagram – every class has its title, and the methods are listed beneath.

Prior to the modelling of the class diagram, we specified a CRC card for each future class, to provide an overview of their relationships, attributes and operations. In this way we pointed out the dependencies and collaborations with respect to the different responsibilities each class is charged with.



DatasetReader

**Super Classes:**

**Sub Classes:** Test Graph; Training Graph; Validation Graph

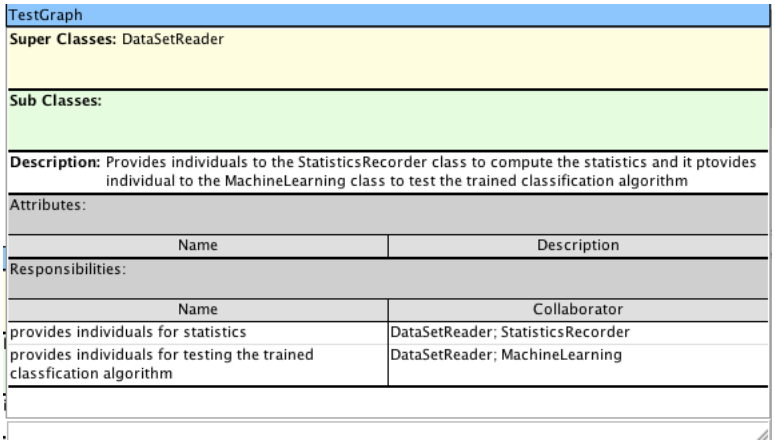
**Description:** Thr DataSetReader query the GenoDataSet via SPARQL and converts individuals format from RDF to Json

Attributes:

| Name | Description |
|---|---|

Responsibilities:

| Name | Collaborator |
|---|---|
| Read individuals from TG, TRG and VG | GenoDataSet |
| Convert RDF into Json | SPARCQL wrapper |



TestGraph

**Super Classes:** DataSetReader

**Sub Classes:**

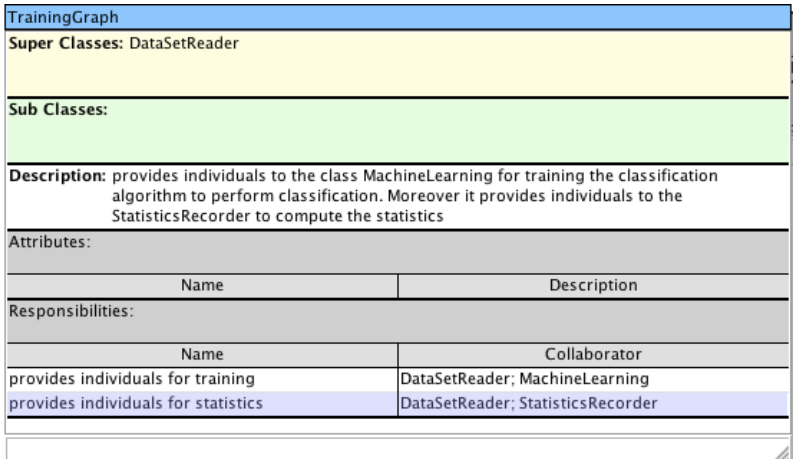
**Description:** Provides individuals to the StatisticsRecorder class to compute the statistics and it ptovides individual to the MachineLearning class to test the trained classification algorithm

Attributes:

| Name | Description |
|---|---|

Responsibilities:

| Name | Collaborator |
|---|---|
| provides individuals for statistics | DataSetReader; StatisticsRecorder |
| provides individuals for testing the trained classfication algorithm | DataSetReader; MachineLearning |



TrainingGraph

**Super Classes:** DataSetReader

**Sub Classes:**

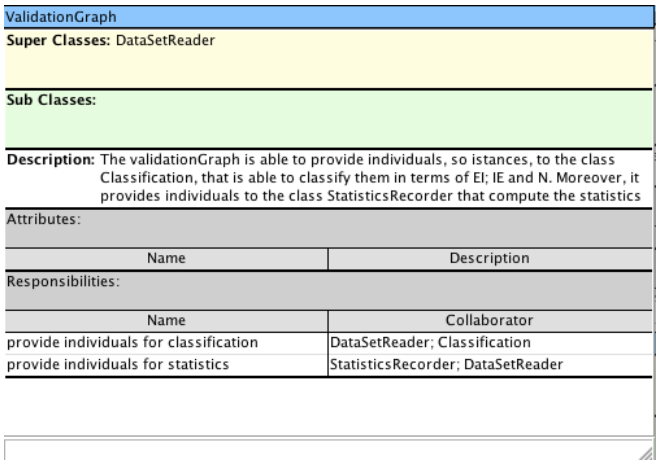
**Description:** provides individuals to the class MachineLearning for training the classification algorithm to perform classification. Moreover it provides individuals to the StatisticsRecorder to compute the statistics

Attributes:

| Name | Description |
|---|---|

Responsibilities:

| Name | Collaborator |
|---|---|
| provides individuals for training | DataSetReader; MachineLearning |
| provides individuals for statistics | DataSetReader; StatisticsRecorder |



ValidationGraph

**Super Classes:** DataSetReader

**Sub Classes:**

**Description:** The validationGraph is able to provide individuals, so istances, to the class Classification, that is able to classify them in terms of EI; IE and N. Moreover, it provides individuals to the class StatisticsRecorder that compute the statistics

Attributes:

| Name | Description |
|---|---|

Responsibilities:

| Name | Collaborator |
|---|---|
| provide individuals for classification | DataSetReader; Classification |
| provide individuals for statistics | StatisticsRecorder; DataSetReader |

**Classification**

**Super Classes:**

**Sub Classes:**

**Description:** performs classification of istances from the validation graph using a GaussianNB() classifier that has been trained by the MachineLearing class

Attributes:

| Name | Description |
|---|---|

Responsibilities:

| Name | Collaborator |
|---|---|
| compute classifications | Validation graph |

**StatisticsRecorder**

**Sub Classes:**

**Description:** computes statistics of the individuals from the training graph, test graph and validation graph

Attributes:

| Name | Description |
|---|---|

Responsibilities:

| Name | Collaborator |
|---|---|
| compute statisitics of test graph individuals | test graph |
| computes statistics of the training graph individuals | training graph |
| computes statistics of the validation graph individuals | validation graph |

**MachineLearning**

**Super Classes:**

**Sub Classes:** PerformanceAnalyzer

**Description:** This class takes the GaussianNB() classifier as parameter and it trains it using istances of from the training graph. Then it tests it using istances of the test graph.

Attributes:

| Name | Description |
|---|---|

Responsibilities:

| Name | Collaborator |
|---|---|
| trains the classification algorithm | TrainingGraph |
| tests the classifier over istances of the test graph | TestGraph |
| show performance | PerformanceAnalyzer |

**PerformanceAnalyzer**

**Super Classes:** MachineLearning

**Sub Classes:**

**Description:** This class collects the performance of the class MachineLearning and provides results in terms of precision, recall, and f1. Then these data are provided to the class responsible for the creation fo the confusion matrix that shows the performance of the classifier

Attributes:

| Name | Description |
|---|---|

Responsibilities:

| Name | Collaborator |
|---|---|
| Analyze performance | MachineLearning |
| contributes to create the confusion matrix | ConfusionMatrixCreator |

**ConfusionMatrixCreator**

**Super Classes:**

**Sub Classes:**

**Description:** This class builds a confusion matrix based on the data collected by the PerformanceAnalyzer

Attributes:

| Name | Description |
|---|---|

Responsibilities:

| Name | Collaborator |
|---|---|
| compute matrix | PerformanceAnalyzer |
| provides matrix with png format | WebPage |

**WebPage**

**Super Classes:**

**Sub Classes:**

**Description:** provides set of web pages through which the user can check for: performance; statistics; classification

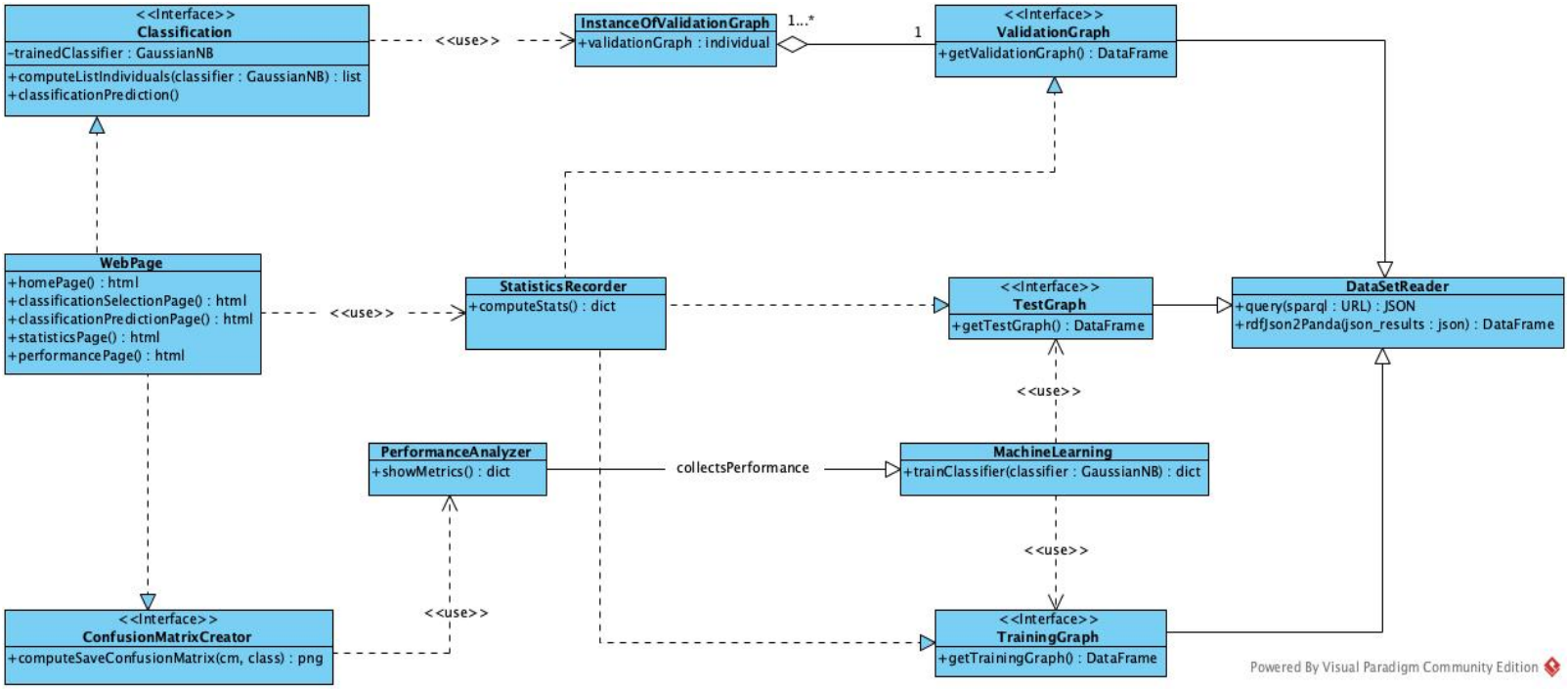
Attributes:

| Name | Description |
|---|---|

Responsibilities:

| Name | Collaborator |
|---|---|
| show statistics to the user | StatisticsRecorder |
| show performance to the user | ConfusionMatrixCreator |
| show classifications to the user | Classification |

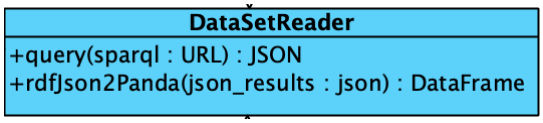
## CLASS DIAGRAM




No attributes were employed in our software because in a utopic framework new graphs are always taken from the GENO Dataset, by the DataSetReader, empowering the dynamicity of the product.
StatisticsRecorder is the class that mostly interacts with the other classes. All the methods are public.
The classification algorithm the team choose, on which the system is built, is indicated in a global variable, *classifier*, storing a *sklearn.Naive_Bayes.GaussianNB* type value, meaning that we choose as algorithm the *GaussianNB()* one, provided by Python's package *sklearn.naive_bayes*, from *scikit-learn*.

### DataSetReader

```
import pandas as pd
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import precision_score, recall_score, f1_score, confusion_matrix
import matplotlib.pyplot as plt
import numpy as np
import itertools, os

classifier = GaussianNB()

class DataSetReader:
    def query(sparql):
        endpoint = SPARQLWrapper("http://wit.istc.cnr.it/geno/sparql")
        endpoint.setQuery(sparql)
        endpoint.setReturnFormat(JSON)
        return endpoint.query().convert()

    def rdfjson2pandas(json_results):
        data = ""
        for result in json_results["results"]["bindings"]:
            data += result["split"]["value"]

            for c in result["sequence"]["value"]:
                value = ord(c)
                data += "," + str(value)
            if "class" in result:
                data += "," + result["class"]["value"] + "\n"
        return pd.read_csv(StringIO(data), sep=",", index_col=0)
```



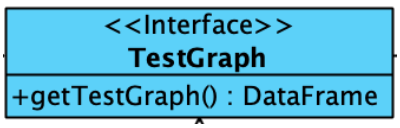

It is the backbone of the system, given that without it, the whole system is not able to work, because it would lack the different graphs on which the computations are performed, to provide the desired outcome. The main operations this class performs are fetching the graphs from the GENO Dataset, and converting them into a format that can be used by our software to work with. These actions are allowed by two methods:

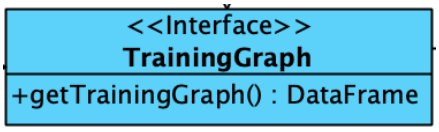
a. query(sparql): requires one argument, which will be a SPARQL endpoint URL code required to fetch the desired graph among the three ones that are available (test graph, training graph, or validation graph). The query(sparql) function will allow to get the data representing each set, explaining how it is crucial for every single class the availability of the different individuals. From the GENO Dataset, the graphs are retrieved in a format which is defined as RDF. This method allows a primary conversion of these obtained data into another format, *json*. Still, to work with them, we need a secondary conversion, provided by the second method of DataSetReader class.

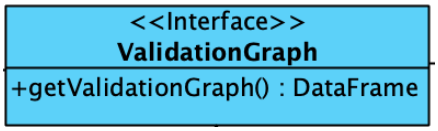
b. rdfjson2pandas(json_results): takes the results coming from query(sparql) as a unique argument and generates a panda data frame from the in-memory CSV, so that it can be used by *scikit-learn*. This function has not been directly tested in our Unit Testing phase because it is implicitly tested when all the other functions which require its use are tested (e.g. the TestGraph, TrainingGraph, and ValidationGraph).

In conclusion: the DataSetReader class queries, via SPARQL HTTP Web service (exploiting *SPARQLWrapper* package), the data contained in the GENO Dataset, where they are stored as RDF files; these latter are finally converted into a readable data frame, which is called *panda*, through an intermediate *json* file type. *scikit-learn* package will then be able to perform the following classification and checking. Finally, this class contains three subclasses: TestGraph, TrainingGraph, and ValidationGraph.

## TestGraph, TrainingGraph and ValidationGraph




```
class TestGraph(DataSetReader):
    def getTestGraph():
        sparql = """
            PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
            SELECT ?split ?sequence ?class ?gene
            FROM <http://wit.istc.cnr.it/geno/test>
            WHERE {
                ?split <http://wit.istc.cnr.it/geno/sequence> ?sequence ;
                    <http://wit.istc.cnr.it/geno/classification> ?class
            }
            """
        results = DataSetReader.query(sparql)

        return DataSetReader.rdfjson2pandas(results)
```

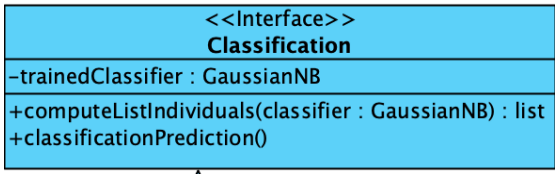
```
class TrainingGraph(DataSetReader):
    def getTrainingGraph():
        sparql = """
            PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
            SELECT ?split ?sequence ?class ?gene
            FROM <http://wit.istc.cnr.it/geno/training>
            WHERE {
                ?split <http://wit.istc.cnr.it/geno/sequence> ?sequence ;
                    <http://wit.istc.cnr.it/geno/classification> ?class
            }
            """
        results=DataSetReader.query(sparql)

        return DataSetReader.rdfjson2pandas(results)
```

```
class ValidationGraph(DataSetReader):
    def getValidationGraph():
        sparql = """
            PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
            SELECT ?split ?sequence ?class ?gene
            FROM <http://wit.istc.cnr.it/geno/validation>
            WHERE {
                ?split <http://wit.istc.cnr.it/geno/sequence> ?sequence
            }
            """
        results = DataSetReader.query(sparql)

        return DataSetReader.rdfjson2pandas(results)
```

These three specular interfaces allow the system to individually fetch the GENO Dataset, by means of the DataSetReader. Each single class contains one specific method: getTestGraph, getTrainingGraph, and getValidationGraph, respectively. The peculiarity of these classes is that they are one independent from the other, allowing high maintenance: if there are some errors in the querying of one graph, the others won't be affected by that. Furthermore, the three classes have one class variable each, named *sparql*, which stores the SPARQL endpoint URL that will be used by DataSetReader.query(sparql). TestGraph, TrainingGraph and ValidationClass collaborate with StatisticsRecorder, MachineLearning and Classification, respectively. In a utopic context, a graph of a certain type (test, training or validation) can be seen as an instance of the class specifically dedicated to each of the three, but we generalized this concept only once, and for the functional purpose of underlying how the classification process depends on the presence of a populated validation graph, where multiple sequences (instances) would be available. Again, the mutability of such process doesn't allow instantiation from these classes.

## Classification

```
class Classification:
    def computeListIndividuals(classifier):
        sparql = """
        PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
        SELECT ?split ?sequence ?class ?gene
        FROM <http://wit.istc.cnr.it/geno/validation>
        WHERE {
            ?split <http://wit.istc.cnr.it/geno/sequence> ?sequence
        }
        """
        results = DataSetReader.query(sparql)
        individuals = []
        for result in results["results"]["bindings"]:

            sequence = ""
            for c in result["sequence"]["value"]:
                value = ord(c)
                if len(sequence) > 0:
                    sequence += ","

                sequence += str(value)

            ind = result["split"]["value"]
            individuals.append(ind)
        return individuals

    def classificationPrediction():
        ind = request.form['individual']
        sparql = """
                PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
                SELECT ?split ?sequence ?class ?gene
                FROM <http://wit.istc.cnr.it/geno/validation>
                WHERE {
                    <""" + ind + """> <http://wit.istc.cnr.it/geno/sequence> ?sequence
                }
                """
        results = DataSetReader.query(sparql)
        sequence = []
        for result in results["results"]["bindings"]:
            for c in result["sequence"]["value"]:
                sequence.append(ord(c))

        predictions = classifier.predict([sequence])
        return ind, predictions
```



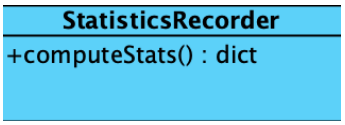

This class relies on the individuals of the validation graph, retrieved by the ValidationGraph class's method, getValidationGraph. Its main methods are:

a. computeListIndividuals(classifier): takes one single argument, *classifier*. It is required to produce the list of individuals that will be visualized in the drop-down menu of the web page, hence all the sequences that are present in the validation graph and are available for a classification. An iteration over the validation graph allows the presentation of the sequences, which are seen by the user as singular HTTP URLs. Clearly, the output of this function is a list of individuals.

b. classificationPrediction(): this method doesn't require arguments because it directly takes the classification for the individual selected by the user, exploiting its HTTP address, allowing the visualization of the answer to user's query, exploiting a for loop which iterates over the list of available items, to fetch its matching junction categorization. Summarizing, we can say that each splice junction sequence is matched to an URL, that is as well associated to a proper classification, either EI, IE, or N.

## StatisticsRecorder:

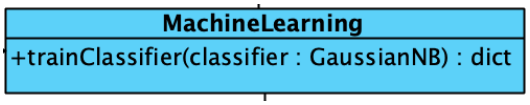
```
class StatisticsRecorder:
    def computeStats():
        sparql = """
            PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
            SELECT (COUNT(DISTINCT ?split) AS ?inds)
            FROM <http://wit.istc.cnr.it/geno/training>
            WHERE {
                ?split <http://wit.istc.cnr.it/geno/sequence> ?sequence
            }
        """
        results = DataSetReader.query(sparql)
        trainingGraphStats = results["results"]["bindings"][0]["inds"]["value"]

        sparql = """
            PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
            SELECT (COUNT(DISTINCT ?split) AS ?inds)
            FROM <http://wit.istc.cnr.it/geno/test>
            WHERE {
                ?split <http://wit.istc.cnr.it/geno/sequence> ?sequence
            }
            """
        results = DataSetReader.query(sparql)
        testGraphStats = results["results"]["bindings"][0]["inds"]["value"]

        sparql = """
            PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
            SELECT (COUNT(DISTINCT ?split) AS ?inds)
            FROM <http://wit.istc.cnr.it/geno/validation>
            WHERE {
                ?split <http://wit.istc.cnr.it/geno/sequence> ?sequence
            }
            """
        results = DataSetReader.query(sparql)
        validationGraphStats = results["results"]["bindings"][0]["inds"]["value"]

        return {"trainingGraphStats": trainingGraphStats,
                "testGraphStats": testGraphStats,
                "validationGraphStats": validationGraphStats}
```



It is the class involved in all the processes that sees the computation of the total amount of individuals present in each of the graphs, explaining why it uses the three classes ValidationGraph, TestGraph and TrainingGraph.
This class comes with a method, computeStats(), which doesn't require arguments. The main purpose of this method is to singularly fetch, one at time, the graphs, by using the respective SPARQL endpoints.
By convention, we choose to use just one variable, and to change it overtime with the SPARQL URL that we required, to avoid using one variable per graph. Indeed, once that each set of individuals has been queried, a string variable containing the number of the graphs' sequences is stored (so three string variables will be created: trainingGraphStats, testGraphStats, and validationGraphStats). Those variables will be the values of a three-keys dictionary, where each key will be labelled with the name of the variable.

## MachineLearning:

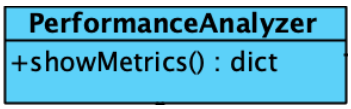
```
class MachineLearning:
    def trainClassifier(classifier):
        training = TrainingGraph.getTrainingGraph()
        test = TestGraph.getTestGraph()
        X_train, y_train = training.iloc[:, :-1], training.iloc[:, -1]
        X_test, y_test = test.iloc[:, :-1], test.iloc[:, -1]
        classifier.fit(X_train, y_train)
        predictions = classifier.predict(X_test)
        precision = precision_score(y_test, predictions, average='weighted')
        recall = recall_score(y_test, predictions, average='weighted')
        f1 = f1_score(y_test, predictions, average='weighted')
        cm = confusion_matrix(y_test, predictions)
        confusion_matrix_image = ConfusionMatrixCreator.computeSaveConfusionMatrix(cm, ["ie", "ei", "n"])

        return          {
                        "precision": precision,
                        "recall": recall,
                        "f1": f1,
                        "confusion_matrix": confusion_matrix_image
                        }
```




This class uses both training graphs and test graphs to both "learn" how to classify the different sequences of junctions, and to provide data available for following analysis regarding the performance of the machine learning algorithm itself, which is supervised.
This class works with a method, trainClassifier, which as the name says takes the *GaussianNB()*, (i.e. the *classifier* global variable, our classification algorithm!), and literally teaches it how to perform the classification. But this is only part of the capabilities that this method has: in fact, by exploiting the function getTestGraph from TestGraph, trainClassifier is able to produce data for a confusion matrix and some additional metrics data analysis, that let the user infer details about the performance of the algorithm itself, which is the foundation of every software based on machine learning, included our system. Moving our attention to the Component Diagram, we can see that the component that implements the MachineLearning class is "Machine-learning algorithm realization", which provides the performance of the algorithm to *the* "Performance analyser and recorder". This may sound a little outstanding with respect to the context of the class diagram, but it is crucial to understand how MachineLearning class by itself is not able to produce consultable data about the algorithm performance; *"*Performance analyser and recorder*"* contains the two independent classes that will allow to do so. Hence, indeed, other collaborations are required to achieve the proper presentation of the data, and at this point other competent classes come to assist MachineLearning one; that's why this latter has also a subclass, PerformanceAnalyzer.

## PerformanceAnalyzer

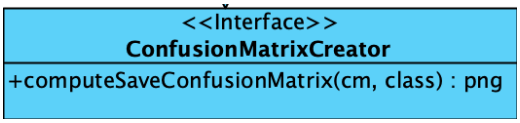
```
class PerformanceAnalyzer(MachineLearning):
    def showMetrics():
        return MachineLearning.trainClassifier(classifier)
```




This subclass of MachineLearning has the main scope of collecting all the data about the additional metrics and the confusion matrix by its superclass, and to present them to the user in the web pages. It has a single method, showMetrics, and it returns as output both the additional metrics and the confusion matrix mentioned above.
This class, which simply "outputs" the data in a representable way, has made the code easier to be managed when dealing with the WebPage class: our desire was to divide the different responsibilities through as much classes as possible, especially with respect to the machine learning training methods, which we want to keep as safe from bugs as possible. In fact, the PerformanceAnalyzer class is still dependent on the MachineLearning class, but the MachineLearning class is not dependent on it.

## ConfusionMatrixCreator

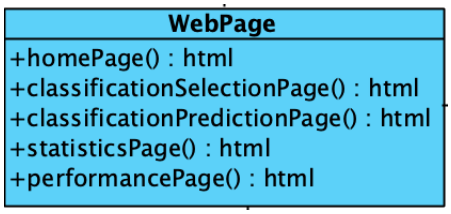
```
class ConfusionMatrixCreator:
    def computeSaveConfusionMatrix(cm, classes):
        cmap=plt.cm.Blues
        plt.imshow(cm, interpolation='nearest', cmap=cmap)
        plt.title('Confusion matrix')
        plt.colorbar()
        tick_marks = np.arange(len(classes))
        plt.xticks(tick_marks, classes, rotation=45)
        plt.yticks(tick_marks, classes)
        fmt = '.2f'
        thresh = cm.max() / 2.
        for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
            plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

        plt.ylabel('True label')
        plt.xlabel('Predicted label')
        plt.tight_layout()

        fileName = "static/confusion-matrix.png"

        plt.savefig(fileName, dpi=None, facecolor='w', edgecolor='w',
                    orientation='portrait', papertype=None, format=None,
                    transparent=False, bbox_inches='tight', pad_inches=0.3,
                    frameon=None)
        return fileName
```




This class contains one method, computeSaveConfusionMatrix, which is used to produce the consultable confusion matrix using the additional metrics on precision, f measure and recall as a *png* file, that will be automatically downloaded into a folder called *"static"*; this matrix will also be shown on the HTML web page to the user, when it accesses the dedicated page.

## WebPage

```
class WebPage:
    app = Flask(__name__)
    @app.route("/")
    def homepage():
        return render_template('homepage.html')

    @app.route('/statistics', methods=['POST'])
    def statisticsPage():
        stats = StatisticsRecorder.computeStats()
        return render_template('statistics.html',stats=stats)

    @app.route('/performance',methods=['POST'])
    def performancePage():
        metrics = PerformanceAnalyzer.showMetrics()
        return render_template('performance.html',metrics=metrics)

    @app.route('/classification', methods=['POST'])
    def classificationSelectionPage():
        individuals = Classification.computeListIndividuals(classifier)
        return render_template('classification.html', validationSet=individuals)

    @app.route('/classify',methods=['POST'])
    def classificationPredictionPage():
        classification,predictions = Classification.classificationPrediction()
        return render_template('prediction.html', individual=classification, predictions=predictions)

    trainingSet = TrainingGraph.getTrainingGraph()
    X_train, y_train = trainingSet.iloc[:, :-1], trainingSet.iloc[:, -1]

    classifier.fit(X_train, y_train)

    if __name__ == '__main__':
        classifier = GaussianNB()

        if not os.path.exists("static"):
            os.mkdir("static")

    if __name__ == '__main__':
        app.run(debug=True)
```




This class implements the web interface that is going to interact with the user, i.e. the user interface. It will make the app available on port 5000 on your Web browser, at http://localhost:5000.
It is based on the *Flask* microframework for Python, used to build web applications: it is classified as a micro web framework because it doesn't require particular tools or libraries. It provides 5 methods, each associated to one HTML page specifically developed. We manually re-elaborated the HTML pages from pre-existent ones, except for the homepage, which we coded from scratch. These methods are:

a. homepage(): simply returns the "homepage.html" as soon as the app is launched.

b. statisticsPage(): calls the function StatisticsRecorder.computeStats() to provide the user the number of individuals that are present in each class. It launches the web page "statistics.html".

c. performancePage(): calls the function PerformanceAnalyzer.showMetrics(), developed to deal specifically with the WebPage, to avoid contacts with the MachineLearning. It launches the web page "performance.html".

d. classificationSelectionPage(): outlines the individuals that are available for the classification, each indicated by an URL, as previously said, through the list produced by Classification.classificationPrediction(). The user will be able, through a specifically coded HTML page ("classification.html", to select the desired sequence from a drop-down menu. To proceed with classification, he will just need to click on "Classify", which will lead to the next HTML page, and to the execution of the following method, classificationPredictionPage().

e. classificationPredictionPage(): opens the HTML page containing the classification for the selected instance ("prediction.html", after having called the function Classification.classificationPrediction().

Each HTML page is rendered through the render_template function, provided by Flask. All the HTML source codes have been provided as documentation along with this paper.

## 2.4 System validation: integration and testing

"Integration" means combining of subroutines, software modules or full programs with other software components in order to develop an application or enhance the functionality of an existing one. Often requiring a lot of source code modification, developers, as well as IT staff, may spend a large amount of their time performing software integration. There are several perks of integrating your software. It makes fault detection a simple process and also removes the need to deal with stand-alone systems, hence reducing time. However, if security is breached in any one of the sub-systems, the entire system could be breached. Hence, it is safe to say that software integration is important, but it could come at a cost. This is why our team paid close attention to the testing phase, especially Unit Testing.
Unit Testing of software applications is done during the development (coding) of an application. The objective of Unit Testing is to isolate a section of code and verify its correctness. The team proceeded by isolating the functions present in the application source code, to test them more rigorously, and to help in revealing unnecessary dependencies between the code being tested and other units or data spaces at the same time.
We used a UnitTest Framework to develop automated test cases: unittest library from Python.
The workflow we followed for the Unit Testing was rather simple: we created a single test case for each class that we'd like to test, containing as much as possible text fixtures, to assure the software's output result is correct. After having created a single test case, we tested it, and if it worked, we would put it inside the main test suite, and run it. Furthermore, when a new test fixture was added to a test case, we immediately tested it as well, to get an incremental development of the test itself, updating it from time to time. It was our choice to use a single Python file for the tests, including both the codes for the test cases, and for the creation of the suite. This allowed us to skip the test loader. The output of the results has been implemented in such a way to follow a user-friendly, tabular, layout, thanks to a reused template from GitHub. However, we had to rework the template's source code, to make it fit our app. If some test fails, the user can see which is the test that has failed in a single test case, by changing the source code of the test, and using a standard output outline.
The test is characterized by 8 test cases, where each one is endowed with several text fixtures (in addition to the setUp and tearDown ones):

- DatasetReaderTestCase is used to mainly test the DatasetReader.query(sparql) method. It is characterized by 8 fixtures which include an evaluation of the content of the output along with its type check, followed by a deep analysis of the rules of content's organization, to ensure the correctness of the primary format conversion underwent by the graphs, fetched from the GENO Dataset.

- StatisticsRecorderTestCase is focused on testing the StatisticsRecorder.computeStats function. The 9 text fixtures chosen for this test case are similar to the ones used in the DatasetReaderTestCase, with an additional one (we checked that a value is greater than another, known to be smaller, in the output data; if not so, the test fails).

- TrainingGraphTestCase, TestGraphTestCase and ValidationGraphTestCase are completely devoted to the interface classes' testing. We decided to divide the three test cases, to provide one case for each interface, but they are identical from the fixtures point of view, except for ValidationGraphTestCase. Excluding this latter one, they have 5 text fixtures which take into account the type of the output as first thing, which is very important, because it will be the readable format required for the system to read and use the graphs from the GENO Dataset. If the type is not the expected one, dreadful effect may come downstream. Following, the other fixtures test the number of elements present in each graph, the actual presence of a conventionally chosen value, known to be actually present, and the non-emptiness of the graphs. ValidationGraphTestCase lacks a testTestGraphList fixture, which tests the actual content of the graph, because it would be too computationally intensive to check, weighting on the performances. These fixtures are based on static graphs, with non-changing values, but outside of an academic environment (that is, outside of our purpose of designing a software as a final exam project), these tests could all be re-implemented to support the dynamicity of the graphs with very little changes, providing maintainability, versatility and compatibility.

- MachineLearningTestCase is prompted to the testing of the MachineLearning class, and it has 8 test fixtures, almost identical to the ones seen in DatasetReaderTestCase. We rely on the conversion of the results into lists to easily manage them (both lists of items and keys, given the outcome is a dictionary, for example).

- ClassificationTestCase is aimed to testing the Classification.computeListIndividuals method in Classification class. It is characterized by 6 test fixtures, which follows the footprint of the previous test cases, but here we required the local fitting of the GaussianNB() variable to test the predictions. Unfortunately, the second method required for the classification of an individual resulted impossible to test, given how much it relies on Flask.

- ConfusionMatrixTestCase is the last test case present. It doesn't require a setUp and tearDown methods because it is just going to check both the existence of the directory in which the confusion matrix is going to be saved, and the existence of the matrix itself, after the app has been run.

Unit testing can't be expected to catch every error in a program. It is not possible to evaluate all execution paths even in the most trivial programs and by its very nature focuses on a unit of code. Hence it can't catch integration errors or broad system level errors.

Our main purpose was to provide a reliable genomic tool, free from any kind of leaky code, given how much the research community can rely on results coming out from the tools they require. This fact prompted the team in developing a test which is rich in fixtures involving precision and correctness of the results, which are crucial whenever the user desire is to drive studies, analysis and pairwise comparison in a biological mindset and working environment.

The system designed as our class project is rather an "abstraction" in which some of the technical corners are cut.

## 2.5 System Evolution

This section of the document explains how we developed our web application in order to support system evolution, which refers to the ability of a system to be updated in time: a software system in use, in fact, will never stop evolving. The adaptation to new requirements and the elimination of conflicts arising through change propagation force a continuous adaptation of the code. More specifically, the environment, in which software is developed, changes more dynamically than the foundations on which a system is built, hence It is more difficult to foresee how requirements will have changed once the software is built. Changes that are impossible to predict cannot be taken into account in the initial design. Software evolution has been greatly impacted by the Internet: the rapid growth of World Wide Web and Internet Resources make it easier for users and engineers to interact, in order to find out possible improvements and modifications that would improve the final product. For the development of our web application, we have no real customer to consult, who would give us new ideas and hints to boost the maintenance of ALDA W.A. Therefore, we focused our attention on system's parallel evolution with the GENO dataset, rather than with new requirements coming from the external stakeholders.

The object-oriented paradigm itself promises to support development and evolution of software: for example, we modelled the three classes that gather the graphs of the GENO Dataset independently from the others and from each other: we can see this ability as an alternative way of supporting change avoidance since we avoid changing of the code of the system in the moment in which the GENO dataset evolves.

Another point that supports evolution is the way through which we produced the tests for our web application: each test case is composed by many test fixtures, and each of them comes with a clear error message, if the test fails. Furthermore, all the test cases are included in a unique source code file, reducing the amount of documentation coming along with the main product code. Hence, a possible programmer that would like to test our software can easily understand, by looking at the output results, in which part of the code the problem that needs to be solved is localized.

ALDA is internally structured in a way that efficiently supports future refactoring, which is another key point of system evolution and maintenance. The code that implements the system is characterized by a low level of code duplication, along with high clearness of the functions, and low amount of dependencies, in terms of inheritance relationships. Furthermore, the implementation itself mirrors the UML diagrams we modelled to give a comprehensible design backbone to the whole system.

*"Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written."*

This quote of Martin Fowler points out the key attributes of refactoring. Refactoring is a coding technique applied mainly in consolidation phases during development. Examples for refactoring are the renaming of a source code entity or the pushing up of methods in a superclass. Periodical refactoring keeps you cleaning up your own code, it allows you to purge unused elements and to keep the structure of source code consistent with a proper design. Clean code will help us to find better implementations during maintenance and evolution.

# 3. Conclusion

In this last section we want to conclude by pointing out which are the positive aspects of ALDA Web Application.

On one side, we provided a detailed and organized UML documentation for the software, to show its design. The UML diagrams the team built in this documentation are: Use Case Diagram, Component Diagram, Activity Diagram, Sequence Diagram and Class Diagram.
Our diagrams are highly coherent among each other: this implies a more accurate comprehension of the system, since the modelling part of software development is a good tool to understand the final product.

On the other side, the web application source code has been developed using object oriented (OO) approach, to have a clear organisation of the code in a real-time development. Furthermore, there is no redundancy among each class that we implemented: almost all of them produce a unique result which is different form the ones of the other classes, still maintaining a high level of cohesion. In this way, we reduce possible misunderstandings between the different sections of the code and therefore the system is able to return clear results to the user.
Still considering the coding part of the system, we developed it in such a manner that it can support possible changes. More precisely, since our software relies completely on data coming from the GENO Dataset, we found out solutions to cope with its evolution overtime. Considering a utopic situation in which that the database changes, our software is able to accommodate possible variation since the way through which the training, test and validation graph are fetched would not be altered and therefore, our software could be still able to work efficiently.