4 Project Design

4.1 UML

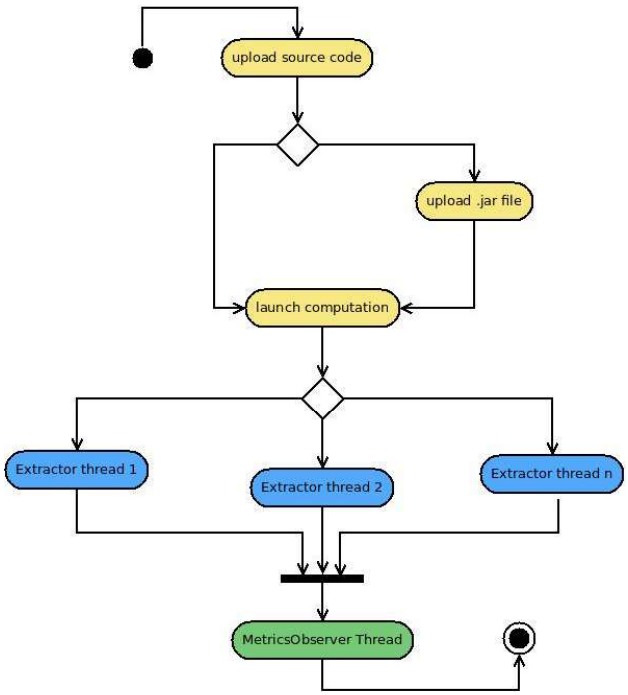

Figure 19: Activity diagram.

The software execution in figure 19 is composed of three different phases:

- Crawling (yellow)

  First of all a directory containing the .java file to be analysed and an optional .jar file containg the related compiled classes are chosen. Only if the directory contains .java files and if the .jar file effectively contains the related compiled classes, the crawling phase is passed.

- Single file analysis (light blue)

  For every .java file found, a MetricsExtractor thread is launched in order to compile code and extract a lot of useful data. All these threads are implemented with the Barrier concurrent pattern.

- General analysis (green)

  When the MetricsExtractors have reached the barrier, one last Thread called MetricsObserver is launched to compute final metrics.

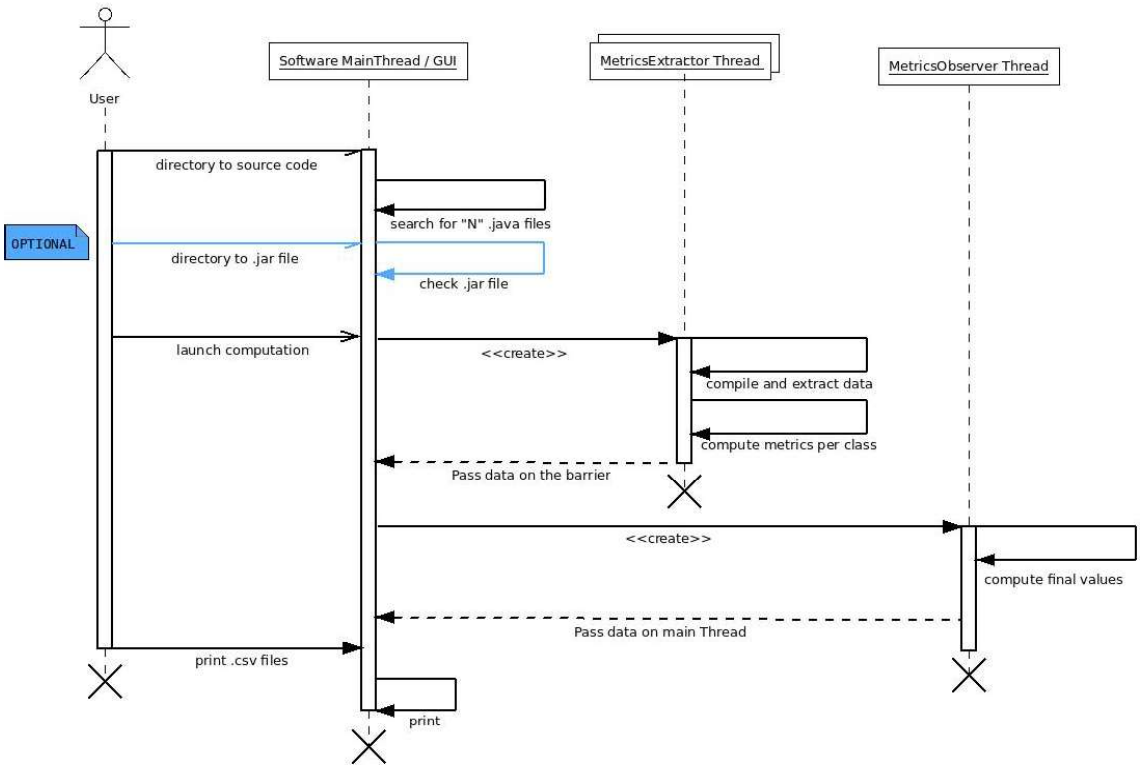

Figure 20: sequence diagram.

The sequence diagram in figure 20 is trying to explain the execution model in details. A user communicates with the main thread through the GUI. When he launches a computation successfully the MainThread creates the needed threads in order to compute the desidered metrics on the uploaded java code. The threads pass the extracted and calculated data to the Mainthread before dying. Once the computation ends the user can ask the MainThread to print the results in three different .csv files which will be explained in chapter 6.

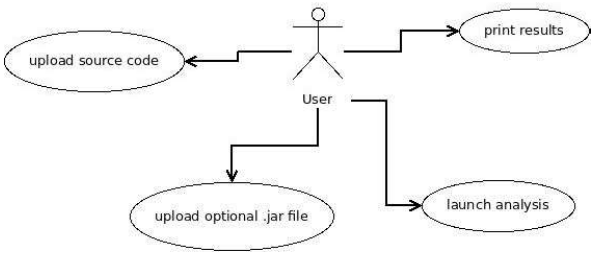

Figure 21: use case diagram.

The use case diagram in figure 21 shows what a user can do with the software from an high level of abstraction. Uploading a .jar file is optional because it is requested only if we want to calculate also structure metrics like "detph of inheritance tree", "number of children" and "coupling beetween objects". The Computation of these metrics is way easier with a compiled code available. For example in order to calculate DIT for a class with just the Java code you have to collect all the classes names and understand which class extends another. After that you need to compare all the collected data and count the depth of a class in the inheritance tree observed before. We also need to consider a plus extension for the Object class that is super class of any class by default. However a java class can extend a class that is located in an imported library. In a situation like that it could be very difficult to trace back the entire inheritance tree of a class. If a compiled code is available instead, calculate the depth of inheritance tree of class is simple as write a recursive method. Indeed a compiled class contains all the needed data. More details about how all the metrics are calculated will be revealed in chapter 5.

In order to create a solid main structure of the software it was implemented using the Model-View-Controller pattern. This choice was made also to simplify the realization of a user friendly GUI with JavaFX, but it will be explained better in the next few pages. In figure 22 we can find a package diagram. The mainPackage contains a class and five different packages:

- generated package

  Containing all the classes generated by ANTLR4.

- data package

  Containing all the classes in which the data extracted during the compilation process are stored.

- logic package

  Containing all the classes that creates the developed execution model, e.g. threads, barrier pattern.

- view package

  Containing all .fxml files of the GUI and the relatives controller classes.

- util package

  Containing all classes with a specific scope. For example the CSVMaganer class is responsible for the results printing in .csv files.

The ComponentsCtrl class represents the main core managing all the GUI scenes, threads and data during the execution. For example, methods in the GUI's controller classes can launch the needed threads or access the extracted data in order to print them through a static reference to an instance of a DataCtrl class (util package).
We will now try to show the software structure in the details benefiting of three different class diagrams, the first for the generated package, the second for the data package and the last one for the logic package.

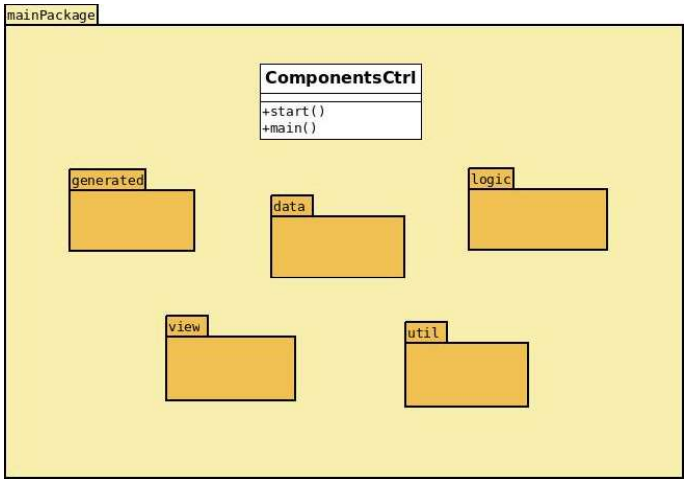

Figure 22: Packages diagram.

The four classes in figure 23 are generated by ANTLR4. The Java8Parser class and the Java8Lexer class are used by every single MetricsExtractor thread in order to compile the code. The Java8Listener interface provides a number of methods equals to the double of different node's type that a java 8 parsing tree can include. Every type of node is related to the possible invocation of two methods during the visit of the parsing tree. One is invoked when the visit is entering the node and the other is invoked when exiting from it.

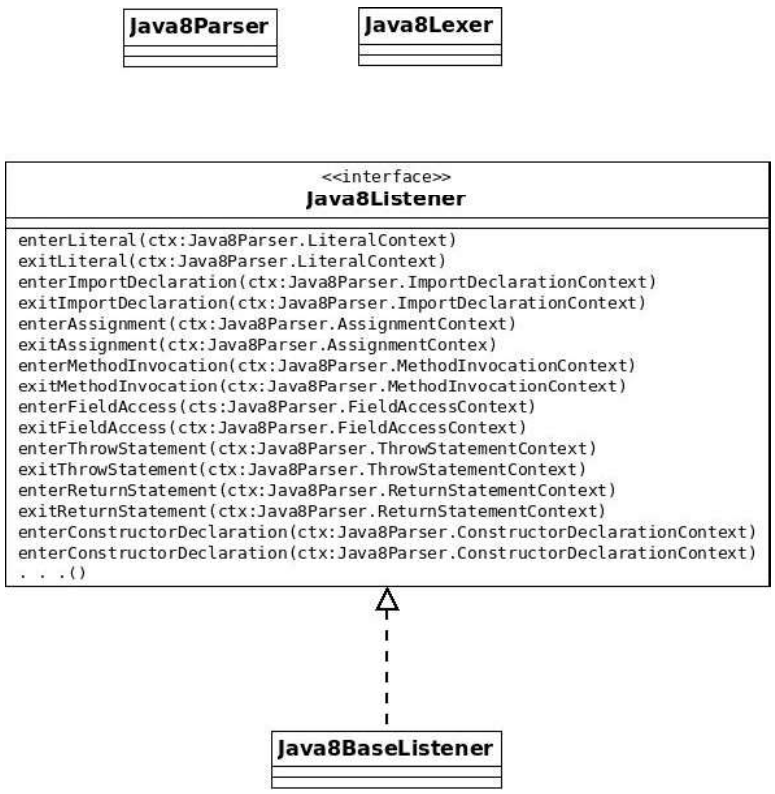

Figure 23: generated package's class diagram.

The class diagram shown in figure 24 is designed to allow the collected data to be passed between the threads in a simple, protected and extensible way. The abstract DataPack class is the idealization of the group of data collected by a MetricsExtractor thread. A number of DataPack instances are aggregated to a GeneralsPack class which represents the group of data related to the MetricsObserver thread. In order to create a more dense analysis of the code also a class corresponding the data of a single function was created. The generalizations for the ExtractorPack class and the GeneralsPack class differentiate the computation with .jar file from the computation without one.

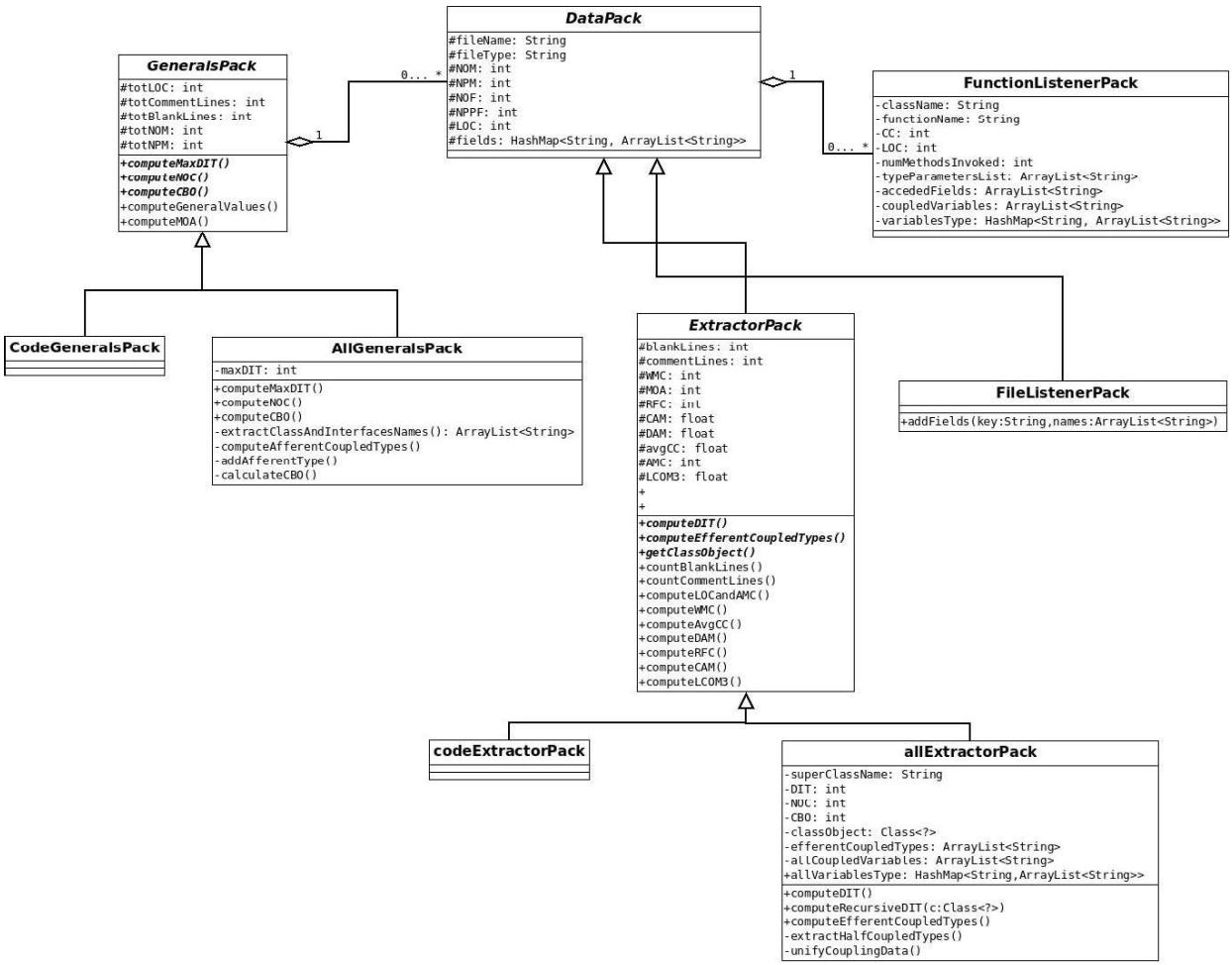

Figure 24: data package's class diagram.

The last class diagram shown in figure 25 includes all the classes which are responsible for the modellation of the lanched threads. MetricsObserver, Barrier and MetricsExtractor classes realized the Barrier concurrent pattern. During the crawling fase the number of the .java files founded in the source code is saved on the Barrier instance. Every single MetricsExtractor thread launched for a .java file is associated to that Barrier instance and when a MetricExtractor computation is done all the relatives ExtractorPacks are passed to it and a counter is incremented. When all the MetricsExtractor have finished their work, the Barrier instance is able to notice it because of the number of .java files saved before compared to the counter. At this point the MetricsObserver thread is launched carrying all the ExtractorPacks collected before. The generalizations of MetricsObserver and MetricsExtractor classes differentiate the computation with .jar file from the computation without one as in the data package. The FileListener and the FucntionListener have an important role because all Java8Listener's overrided methods are implemented in them. These classes are responsible for the direct extraction of data from the compilation process. As anticipated before every class in this package has its own dedicated class in the data package, e.g MetricsObserver has GenerealsPack, MetricsExtractor has ExtractorPack.

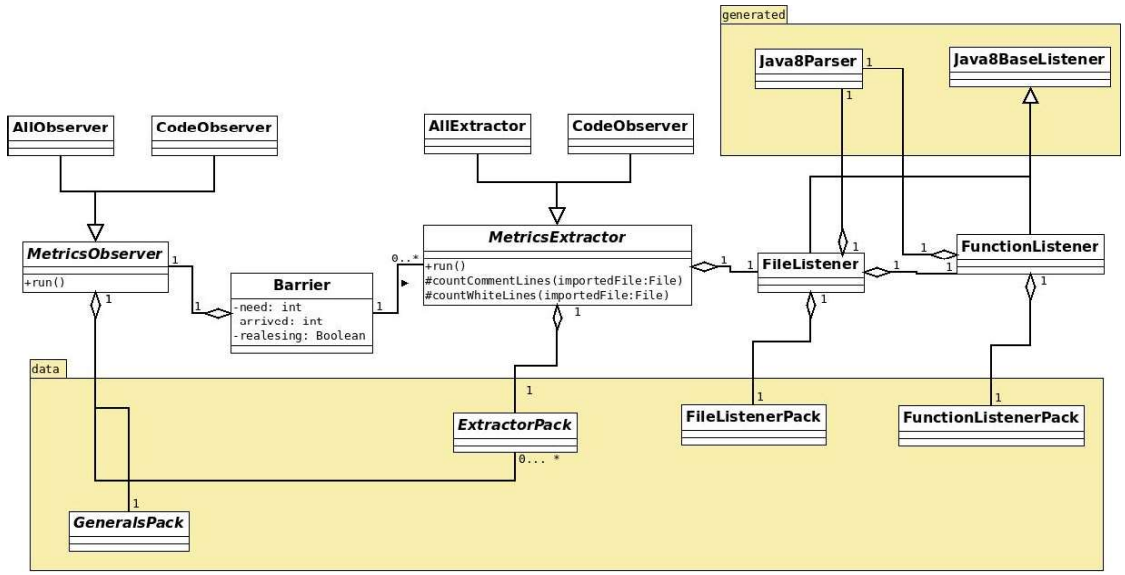

Figure 25: logic package's class diagram.

4.2 GUI

The graphic user interface was obtained thanks to JavaFX and SceneBuilder. The second one is a software to graphically design a scene generating a .fxml file which contain a JavaFX markup language similar to XML. The components in the scene like textfields, charts or links are related to a java object in the corresponding controller class of the scene. For example the InputScene.fxml is controlled by the InputSceneCtrl.java and the connections between these files are made through SceneBuilder. The containers of all scenes (VBox, HBox, Panels, ..) are controlled by ComponentsCtrl class which provides static methods to load and set the .fxml files. An example in figure 26.

```
/**
 * This method set set as the center of the structure layout BorderPane, a VBox which represents
 * the scene for the input request.
 */
public static void showInputSceneLayout(){
    try{
        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(ComponentsCtrl.class.getResource( name: "view/InputScene.fxml"));
        inputLayout = loader.load();

        structureLayout.setCenter(inputLayout);
    } catch (IOException e){
        e.printStackTrace();
    }

}
```

Figure 26: Example of loader method in ComponentsCtrl.

The input scene in figure 27 gives a user the posiibilty to choose a directoriy to the source code and one to jar file. All the calculated metrics are shown too.

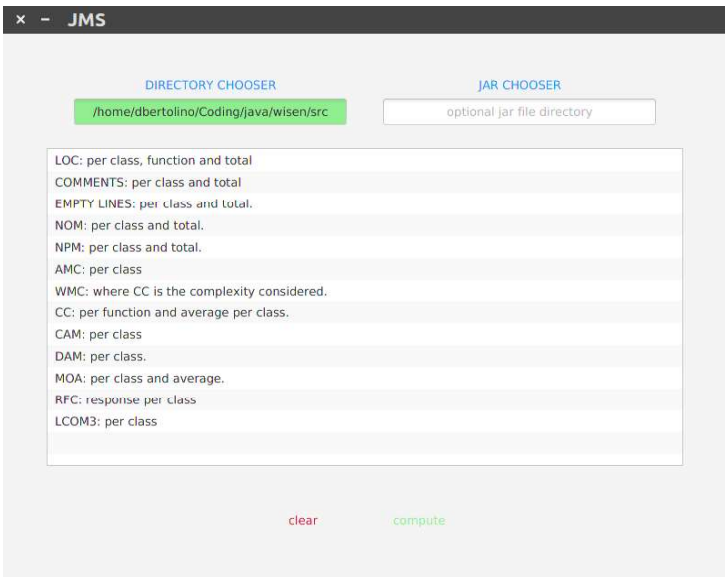

Figure 27: Input scene.

A progress was created to follow the computation and it is shown in figure 28. Every time a MetricsExtractor thread finish his work it notifies the fact to the MainThread which is responsible for the progress bar incrementation. Same discussion for the MetricsObserver thread.

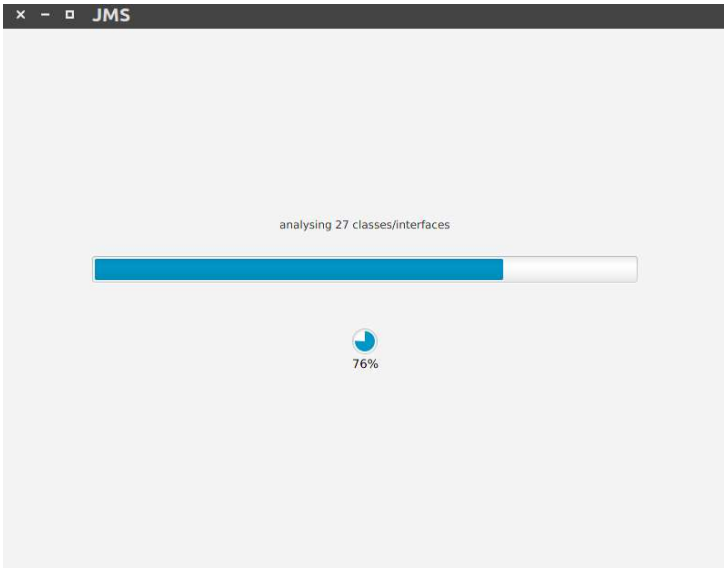

Figure 28: Progress scene.

Another significant scene of the GUI will be shown in chapter 6 to discuss how the results are presented.