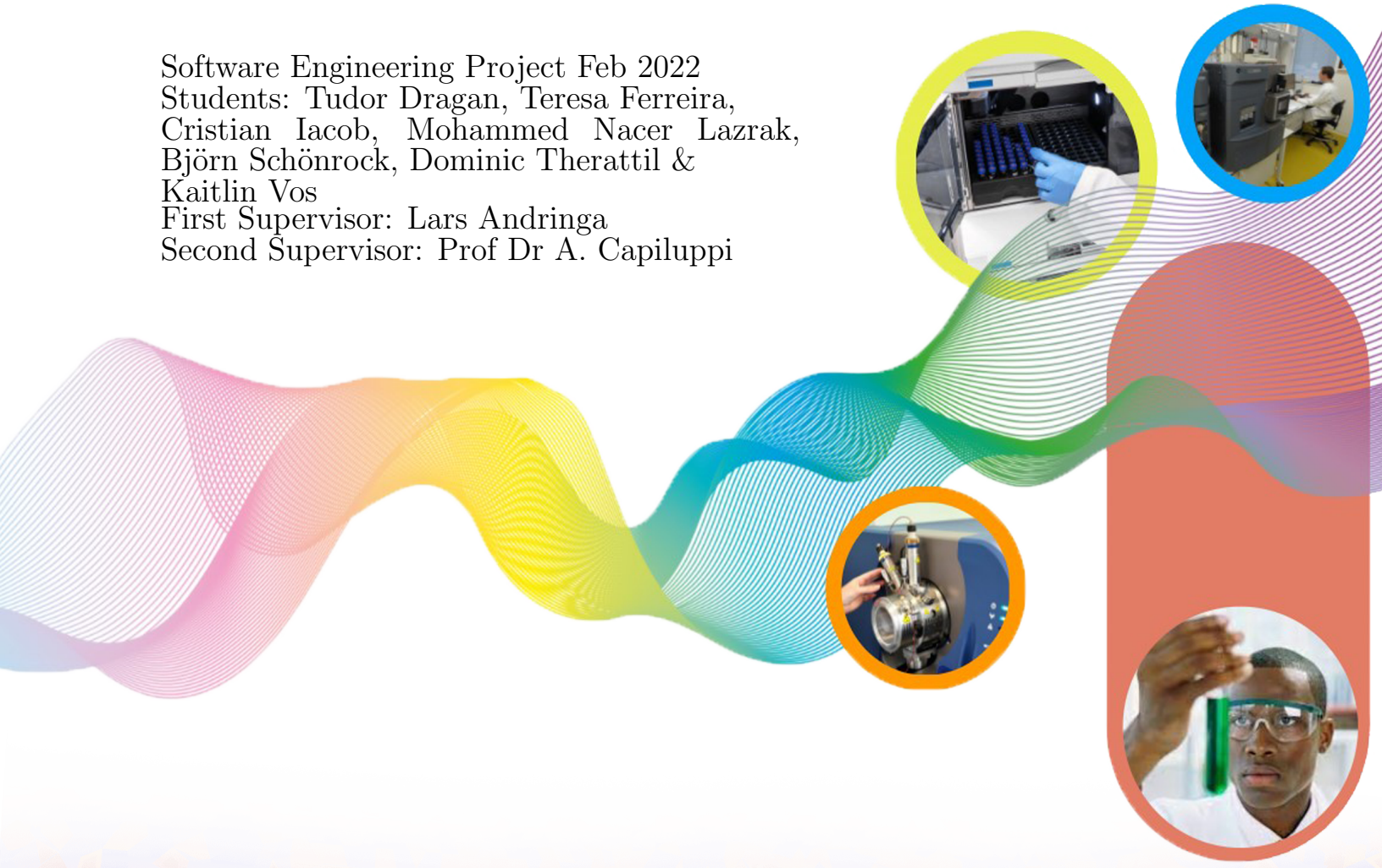
university of groningen // PASTAQ

Architecture Document

Software Engineering Project Feb 2022
Students: Tudor Dragan, Teresa Ferreira, Cristian Iacob, Mohammed Nacer Lazrak, Björn Schönrock, Dominic Therattil & Kaitlin Vos
First Supervisor: Lars Andringa
Second Supervisor: Prof Dr A. Capiluppi



Contents

1 Introduction

The `PASTAQ` pipeline is being developed to quantitatively pre-process, explore and annotate LC-MS/MS data, with a special focus on full data traceability and the quantification of low-intensity signals. This project concerns itself with the maintenance and improvement of an existing graphical user interface (GUI) for the `PASTAQ` pipeline. The system as a whole covers the functionalities needed to convert `.raw` data to `.csv` data with accompanying quality control plots as `.png` files. The process can be observed in Figure 1.

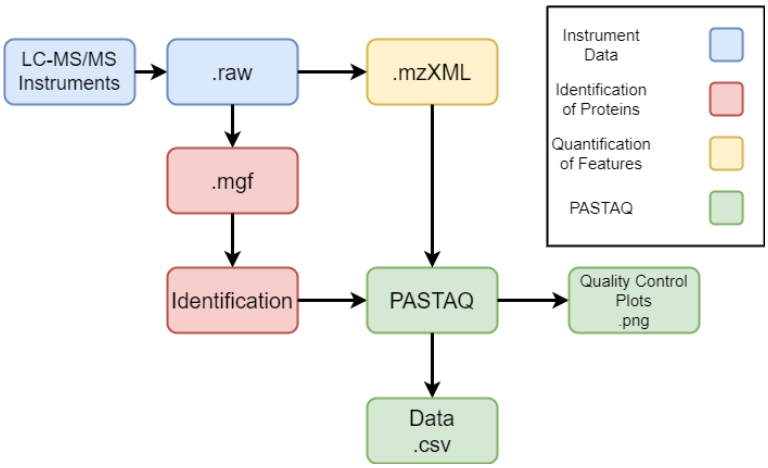


Figure 1: Overview of the ecosystem where PASTAQ is used

Our objective is to enhance the GUI into a more stable and usable version through the additions of tooltips, refactored code smells, support for multiple operating systems, and added quality-of-life features. These features include shortcuts, categorized parameters, a drag and drop feature for inputting files, and an accessible guide for the GUI. We also include automatic file processing from identification files of .mgf format to .mzID format, given that the user has MSFragger and idconvert installed.

Throughout this document we will be discussing the measures taken to ensure a high quality for the final product, along with how our changes were implemented and the way they affected the final product.

2 Technology Stack

2.0.1 Languages

- Python: the language in which our `PASTAQ-GUI` is written in

2.0.2 Technologies

Tools: Currently, there will only be one option to process the identification files into the correct format, which includes MSFragger and idconvert. For different identification processes, different tools can be used.

- MSFragger: an ultrafast database search tool for peptide identification in mass spectrometry-based proteomics provided by `Nesvilab` [1]
- idconvert: a command line tool for converting between various file formats provided by `ProteoWizard` [2]
- PASTAQ: takes the quantification and identification files and outputs `.csv` files and `.png` files for quality control

Libraries:

- PyQt5: a comprehensive set of Python bindings for `Qt5`
- Qt5: Qt is a cross-platform software for creating graphical user interfaces

[1] `https://msfragger.nesvilab.org/`
[2] `https://proteowizard.sourceforge.io/index.html`

3 Architectural Overview

The architecture of our system can be represented throughout the architecture overview found below in Figure 2. The class diagram of our project can be found in the Appendix, Figure 18.

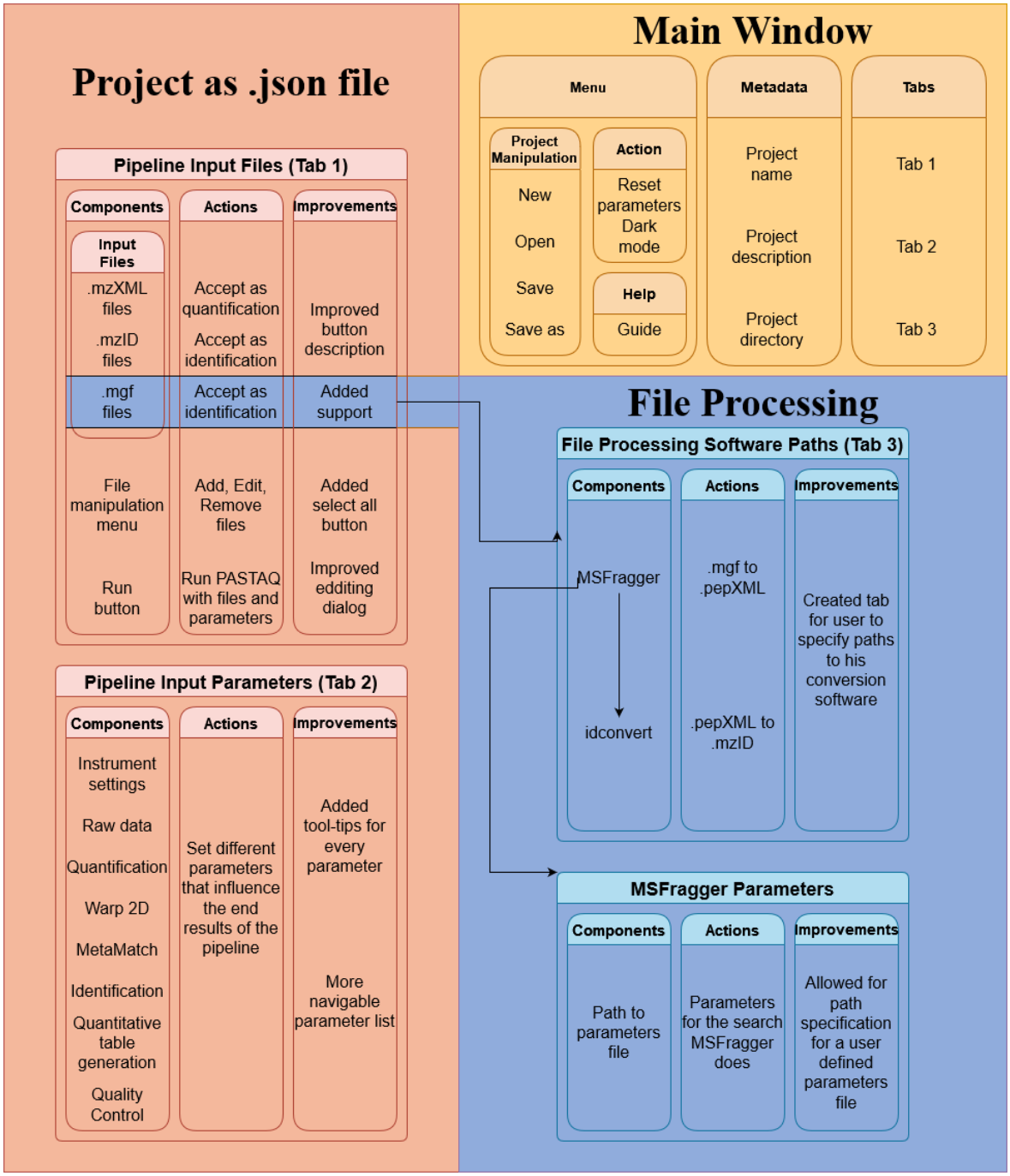


Figure 2: Overview of the Architecture

In regards to the GUI's architecture, the GUI roughly follows the Model–View–Controller (MVC) pattern. The MVC framework has become the standard in modern software development, with the model layer, display layer, and controller layer making it easier and faster. [1] The basic description of the MVC structure can be observed below in Figure 3.

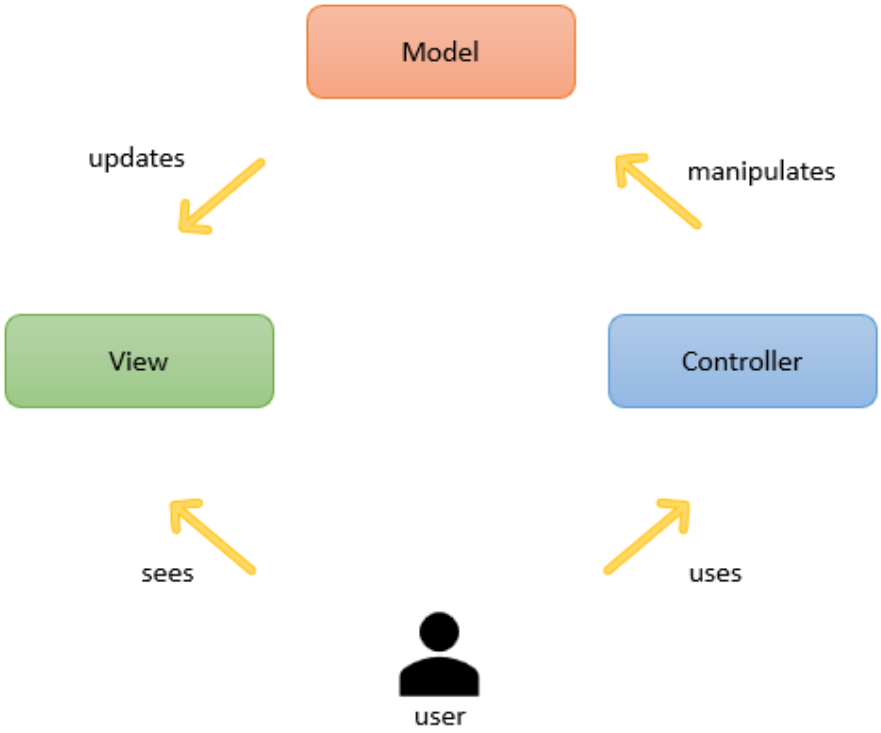


Figure 3: General MVC Structure

4 Model

4.1 Introduction

The model is the basic application that implements the domain logic. It is considered the `backend`. Its main purpose is to manage the data.

4.2 Data

Our data structure can be viewed in the following Figure 4:

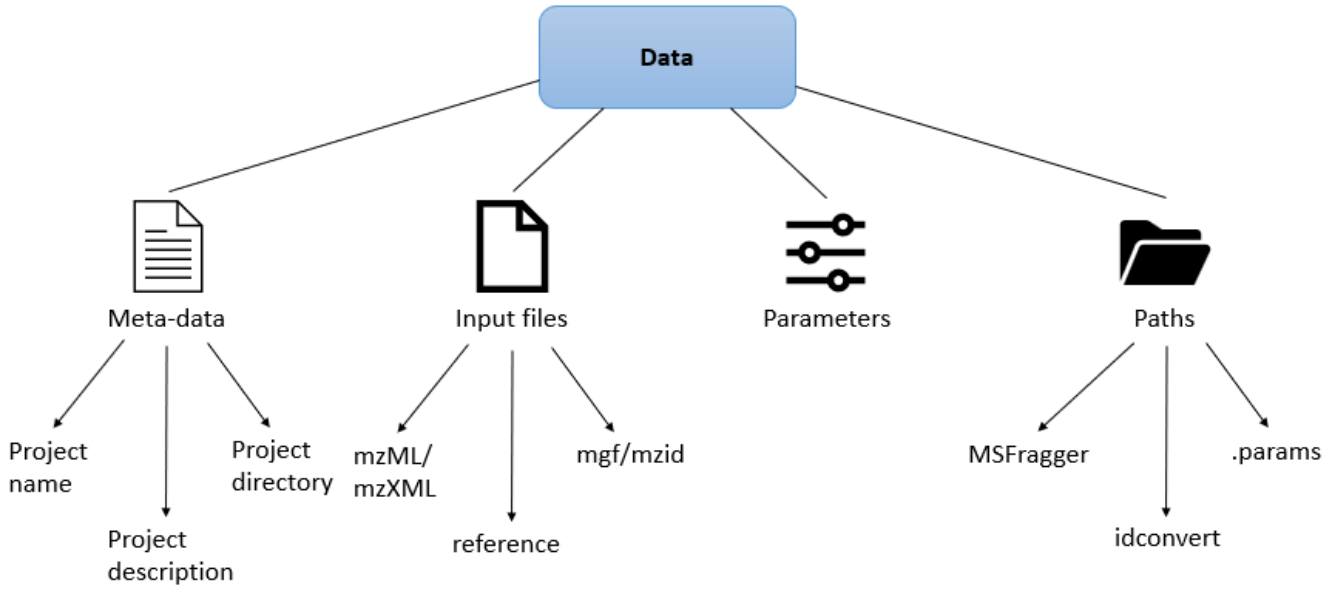


Figure 4: Data structure

4.3 Logic

4.3.1 Project

The project `meta-data`, `input files`, `parameters` and the `path` to the .params file are stored in a `.json` file, since each of these are unique for each project. This file is then stored in the `project directory`.

4.3.2 Configuration settings

The paths to the MSFragger `.jar` file and the idconvert `.exe` file are stored in a configuration settings file of `.ini` format. It is stored in the `PASTAQ-GUI` directory, which allows for all projects to have access to the paths of these applications.
The configuration settings file will be loaded into the `GUI` when the user creates a new project or opens a project. When an error occurs while reading the configuration file, the user will be informed with a message box.
When the configuration settings file does not exist, it will be created and written to upon saving. If the configuration settings file does exist, the correctness of the paths will be evaluated and changed accordingly upon saving.

4.3.3 File Processing

For the pipeline to execute, all identification files need to be .mzID files. Since the GUI also takes .mgf files as input, they need to be processed into the correct file format.
For MSFragger to successfully run, the path to the `.jar` file needs to be provided and the path to the .params file as well. The .params file contains the path to the protein database in .FASTA format and the parameters for MSFragger.
For idconvert to successfully run, the path to the `.exe` needs to be provided.
The file processing is summarized in Figure 5.

4.3.4 Pipeline

The `Run` button opens the process and can be viewed in Figure 5. Before the execution of the pipeline, the individual file processing of each .mgf identification file takes place. MSFragger allows for processing of multiple files at the same time. To better inform the user of the progress, the files are processed individually. This does not affect the efficiency nor the latency.

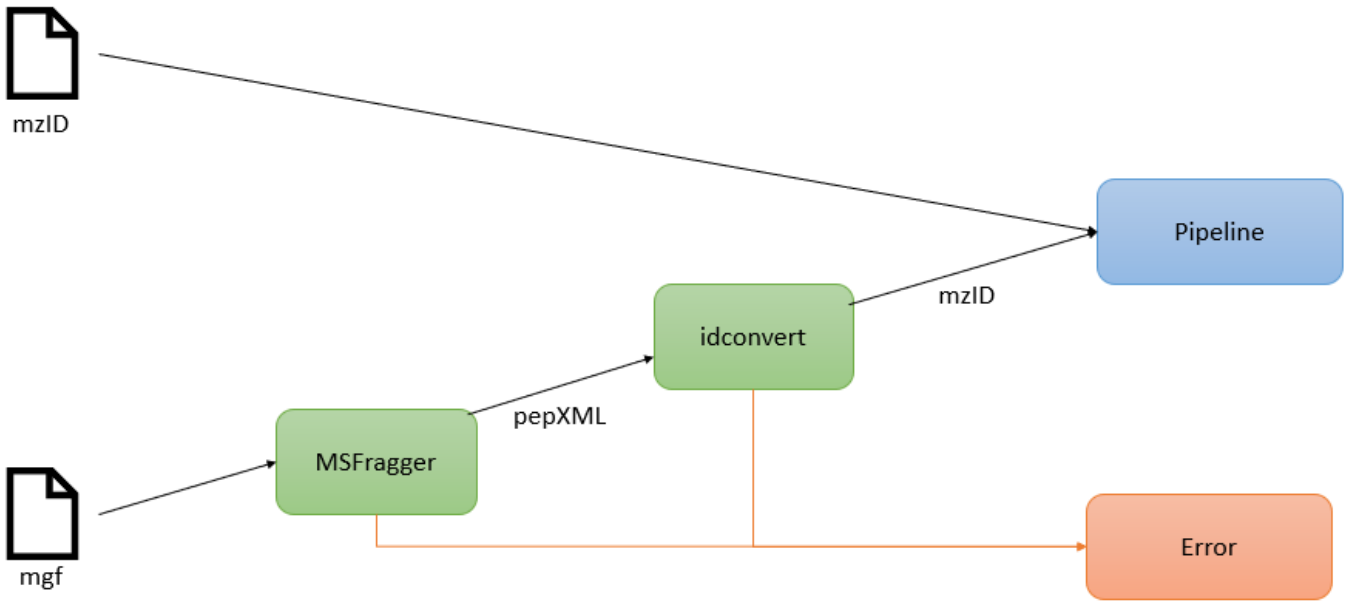


Figure 5: File Processing & Pipeline

5 View

5.1 Introduction

The view, in the context of MVC, represents the visual interface components of the GUI application which interact with the user. The interaction is of an event-driven nature where actions are initiated via keyboard and mouse. Before the main window of the GUI appears, a splash screen is shown, to notify the users that the GUI is loading.

5.2 Splash screen

The splash screen serves as an introduction to the GUI. The QSplashScreen provided by `PyQt5` has been utilized for this purpose. A logo has been created with the objective of identification. The logo includes the actual pasta food, spaghetti in specific, which discretely displays the letter `Q` as well, as can been seen in Figure 6.
This directly corresponds to the name of the GUI PASTAQ. We have designed a splash screen with the name PASTAQ in bold letters with a minimal yellow shadowing, inspired by the logo's color scheme. Since PASTAQ is an abbreviation, what it actually stands for is also included in a smaller font size, to make the title stand out more. The background is greyed out, for a more relaxing touch, and mimics chemistry related scribbles.

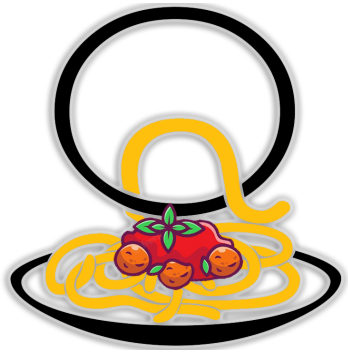


Figure 6: PASTAQ Logo

At the bottom of the splash screen, a thin bar was added with the identifying colors, red and yellow, of PASTAQ, to accentuate the grey background, as can be seen in Figure 7. The splash screen appears before the GUI to mask the loading time.

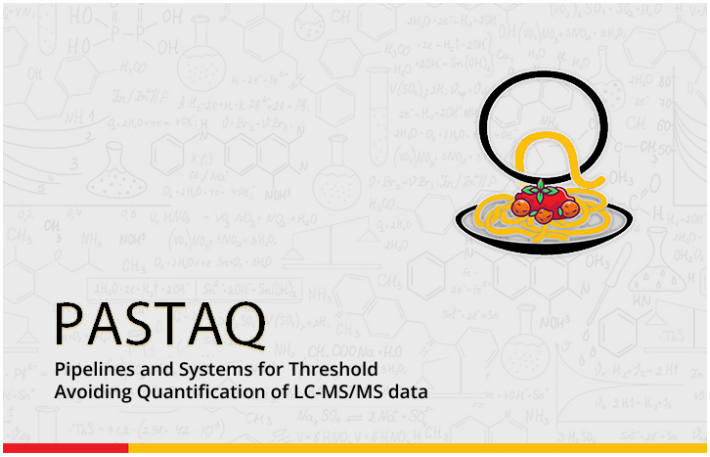


Figure 7: PASTAQ Splash Screen

5.3 Main Window

The structure of the main window can be seen in Figure 8.
The main window is disabled until the user opens a project or creates a new project.

When the user closes the main window and they have not saved the state of the project, they are met with a QMessageBox, which provides the user with three options. The user can choose to save the

project and exit the application, discard the changes and exit the application or cancel the action. This draws the user's attention to the unsaved changes that have been made and prevents data from being lost due to accidental exit actions.

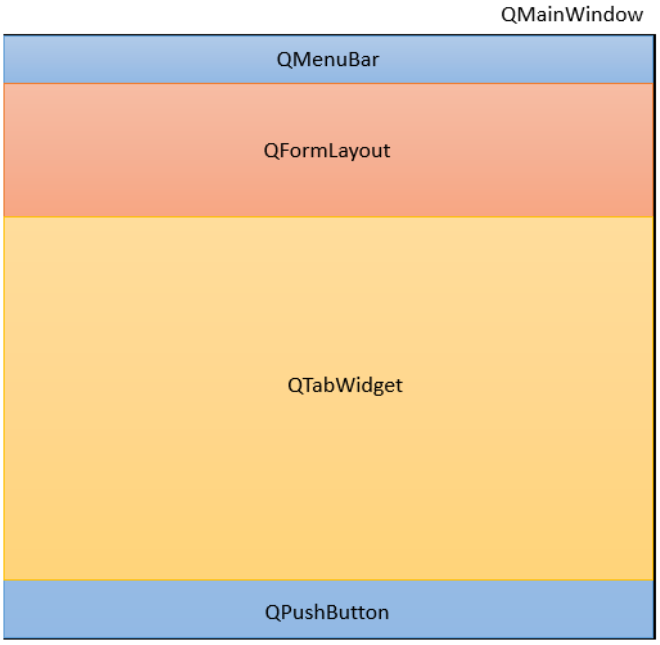


Figure 8: Main Window

5.4 Menu Bar

For ease of use, there is a QMenuBar with 3 drop-down navigation menus: **File**, **Action**, **Help**. Their structures and options are described in Figure 9. These can be considered default actions.

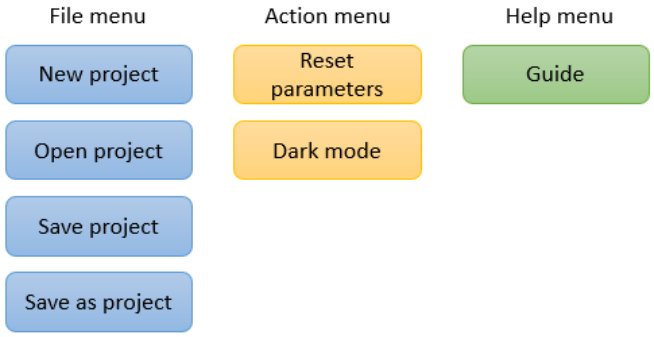


Figure 9: Menu Bar

File:
All of the options in the file menu are done through QActions, and are also tied to shortcuts through QKeySequences.

If the user wishes not to interact with the menu bar, they can make use of these shortcuts. To create a new project, the user may hit `Ctrl+n`, to open an existing project, the user may hit `Ctrl+o`, to save

their project, the user may hit `Ctrl+s`.

Action:
The options this menu offers are done through QActions.
Resetting the parameters simply reverts all the fields back to the default values, and toggling dark mode on and off changes the QPalette to a darker color scheme.

Help:
With the `Guide` QAction the user is able to open a guide for the GUI. It simply opens a link in the user's browser, which is a link to the updated `PASTAQ-GUI` tutorial documentation[3].

5.5 Tabs

5.5.1 Input files

This tab is structured like the following Figure 10.

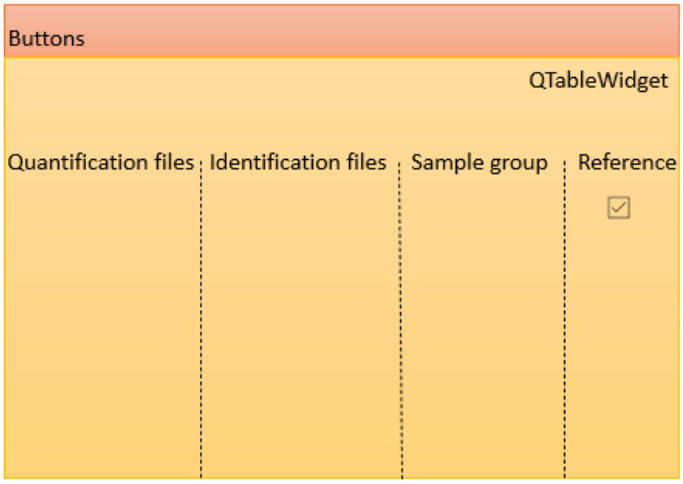


Figure 10: Input Tab

This tab has a row of four QPushButtons at the top, `Add mzXML/mzML`, `Add/Edit mgf/mzID and group`, `Remove` and `Select All`.

The `Add mzXML/mzML` button, upon clicking, opens a QFileDialog for the user to select the identification files.

The `Add/Edit mgf/mzID and group` button, upon clicking, displays a QDialog containing the possibility to enter the QWhatsThis mode in the title bar, represented by the question mark `?`. The QWhatsThis provides the user with more information about the course of action. Then the QDialog window has the following structure that can be seen in Figure 11, supported by a QVBoxLayout.

[3] `https://pastaq.horvatovichlab.com/gui-tutorial/index.html`

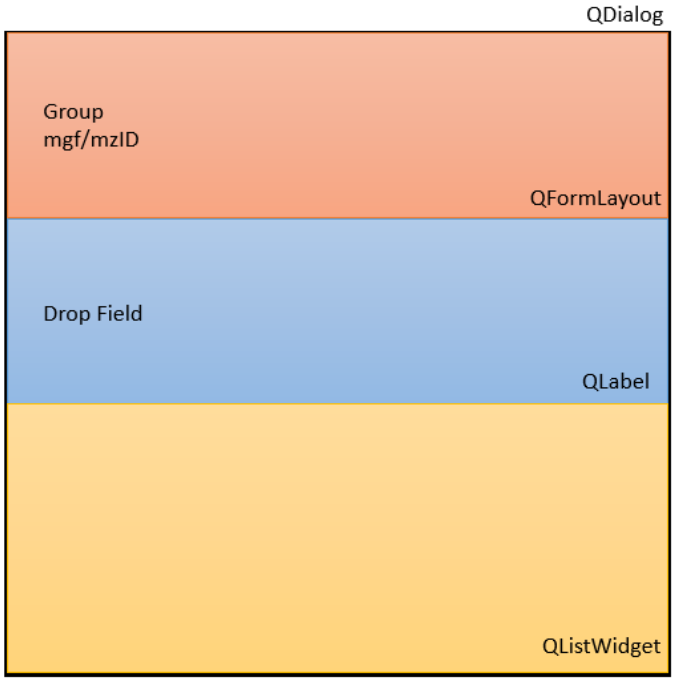


Figure 11: Parameters Tab

The **Group** contains a tooltip and the group can be entered through a QLineEdit.

The **mgf/mzId** also contains a tooltip and a QPushButton, which opens a QFileDialog to browse for the files.

The **drop field** is a custom-made QLabel that indicates where the user may drop their files onto. Upon a drop event, the border of the drop field changes color to indicate that the event has been registered and is being processed.

The QListWidget displays the file paths of the files that have been either dropped or selected through browsing.

Upon pressing the `OK` button of the QDialog, the files and their group categorization will be loaded into the GUI.

The `Remove` button allows the user to delete rows in the QTableWidget. This removal action can also be utilized by pressing the `del` key which is the shortcut we've added.

The `Select All` button allows the user to select all the rows in the QTableWidget. This action is also supported by the shortcut `Ctrl+a`.

Additionally, the tab contains a QTableWidget which can be seen in Figure 10. The table lists the files that have been loaded and the group that has been assigned to certain subsets. The **Reference** column consists of QCheckBoxes and its states play a role in the pipeline execution.

A summary of the sequence of events concerning the identification files and group can be seen in Figure 12 below.

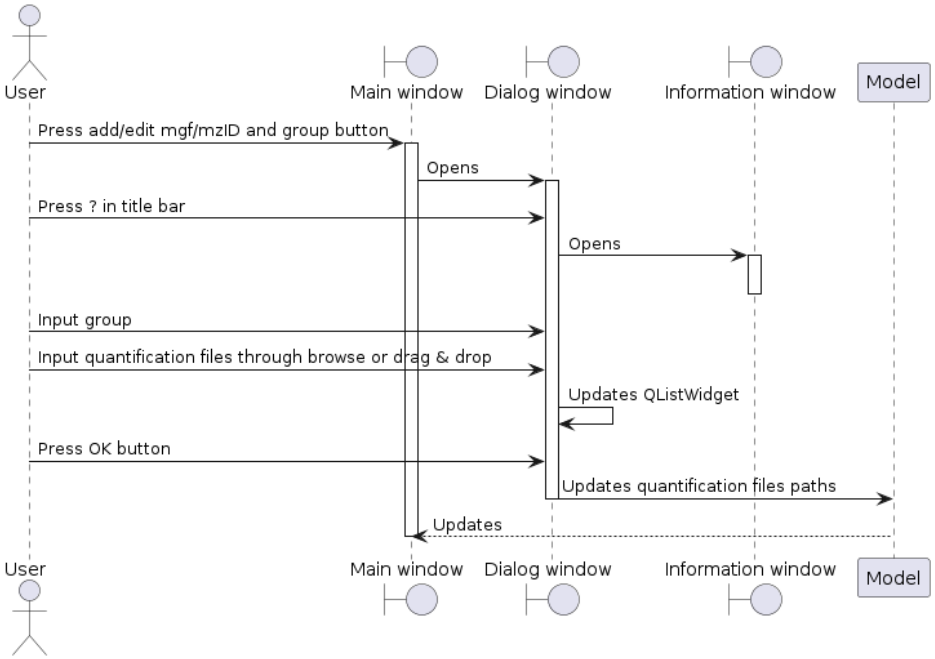


Figure 12: Add/Edit mgf/mzID and group

5.5.2 Parameters

The structure of the parameters tab can be seen in Figure 13.

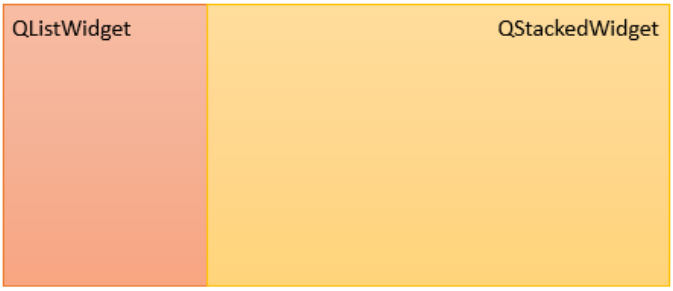


Figure 13: Parameters Tab

The QListWidget lists all the parameter categorizations and the QStackedWidget has been filled with the corresponding QGroupBoxes. Upon clicking on a parameter category in the QListWidget, the correct QGroupBox will be displayed. The separation of all the parameters from just one QScrollArea containing all parameters to separate categories, displayed one at a time, benefits the user. A clear organization has been introduced and the user will not be overwhelmed by the plethora of parameters. Additionally, each QGroupBox contains a description of the parameter category at the very top to provide the user with information. All parameters also contain tooltips explaining the meaning of what the parameter stands for.

5.5.3 Paths

The structure of the `Paths` tab can be viewed in Figure 14.

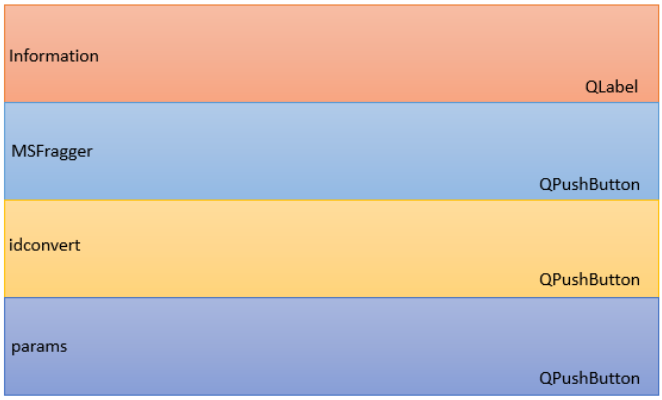


Figure 14: Paths Tab

The information contained in the tab informs the user that the tab is for the automatic file processing of .mgf to .mzID only. The paths of MSFragger, idconvert and .params file can be selected through browsing only, which the QPushButton allows for.

5.6 Execution

At the bottom of the QMainWindow, there is a QPushButton, labeled with `Run`. When the user is ready to execute the pipeline after completing all the inputs, the pipeline can be started by pressing this button. The button is disabled when there are no input files.

6 Controller

6.1 Introduction

The controller is the heart of the control logic and joins the Model with the View by associating user-generated events with data actions. It controls the data flow into a model object and updates the view whenever data changes.

6.2 Controller Logic

In the `PASTAQ-GUI`, the controller does not have advanced embedded logic. It is only responsible for reacting to user interaction, regardless of what it might concern, including inputting files, changing parameters or inputting the paths. The controller can be summarized in the following Figure 15, when the user changes a parameter.

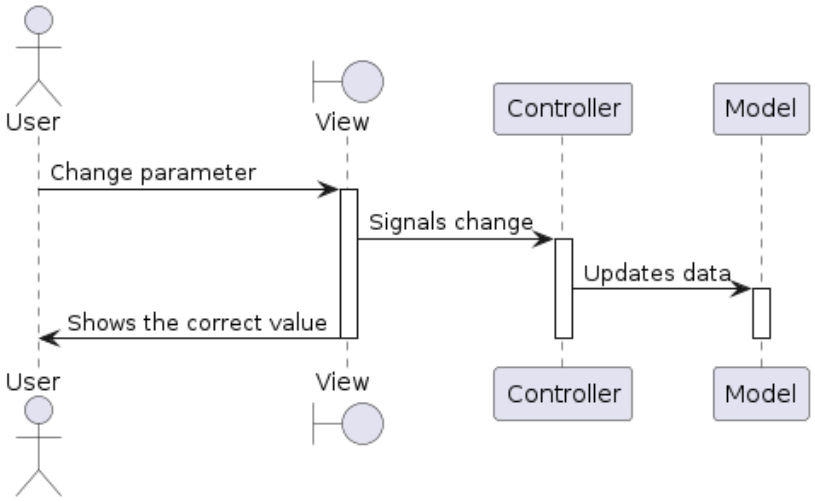


Figure 15: Controller

7 Continuous Integration

Through Github Continuous Integration, automatic processes can be run whenever changes to the code are being made. When a commit is made, Github CI will automatically invoke several actions that can be used for i.e. testing or compiling the code. Github CI is used for the automation of several processes.

7.1 Testing

Github CI is used to perform several tests. The actual tests consist of several Python files found in the `tests` folder. The Python code uses assert statements to perform different tests on the program. When an assertion evaluates to true, it passes the test, and when an assertion evaluates to false, Python raises an `AssertionError`.

Github CI will run all configured test files whenever a new commit is pushed to Github. When a test fails, an `AssertionError` is raised by the Python code. The CI job will then fail, indicating that one of the tests did not run successfully, and the raised error can then be found in the console logs.

7.2 Executables

Github CI is used to generate executables. When a commit is tagged with a version number, Github will automatically create a release. It will then run jobs on Windows and MacOS to create executable files for all of these operating systems. These files will then be uploaded as release assets. The generation process of the executables is summarized in Figure 16. This way, the user does not have to perform complex installation steps and compile the code, but can instead simply launch a single file to use the application. They can be found under the following link [4], which is the repository for our project.

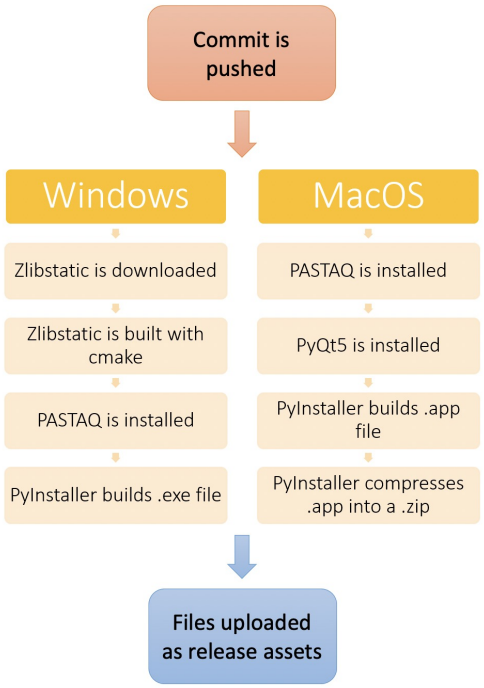


Figure 16: Github CI executable generation

[4] https://github.com/tudor-dragan/PASTAQ-GUI/releases

7.2.1 Windows

On Windows, `PyInstaller` is used to generate the executable. The installation of `PASTAQ` on Windows is more complex than on other platforms. Firstly, `zlibstatic` has to be installed by downloading and building it from the source using `cmake`. Then, PASTAQ is installed using pip and additional dependencies are installed. After that, `PyInstaller` can be used to package the application and its dependencies into an `.exe` file, that is then uploaded to Github.

7.2.2 MacOS

On MacOS, `PyInstaller` is used to generate the executable. First of all, the dependencies like `PASTAQ` and `PyQt5` are installed through pip. Then, `PyInstaller` is used to package the application and all its dependencies into an `.app` file. This file is then uploaded to Github in a `.zip` file.

8 Quality Assurance

As we work on our objectives, we are going to follow the developmental structure outlined below in Figure 17 in order to ensure high quality for our final program. This structure relies on manual testing done by developers and automatic testing through continuous integration using Github Actions

In order to follow the diagram, certain requirements must be met as the code must have comments and be properly stuctured so as to allow development and as the program must be usable by users and developers on all of the major operating systems.

Figure 17: QA Diagram

Aside from the aforementioned means of testing and quality assurance, the program is also verified through unit testing and acceptance testing. Certain requirements, however, were not tested as they have not been implemented due to changes in development philosophy or due to them being beyond the scope of the project.

8.1 Unit Tests

Unit testing is accomplished through the use of `pytest`. `pytest` was chosen as our testing framework since it facilitates testing, and is a great alternative to other frameworks such as `Unittest/PyUnit`. Tests in `pytest` can be written in one file in the form of functions as opposed to the approach of `Unittest/PyUnit` where the creation of a new file per test is required.

We split the tests into four files depending on which section of the program the tests are targeting: the file processing, the app, the parameters and the pipeline. The files have one class per file unless the use of mocking is required. Mocking is used to simulate certain conditions and external applications. Mocking can be predominantly observed within our pipeline and file processing test files, `test_pipeline.py` and `test_files.py`.

The overall structure and results of our unit tests are positive and can be further observed within the traceability matrix and the SonarQube report.

8.2 Acceptance Tests

After the implementation of unit tests, we considered certain requirements to be insufficiently covered. Since our project concerns the GUI and its usability as well, we opted into performing acceptance testing so as to more extensively verify the compliance of our current system to our requirements and the users.

Our method of choice for acceptance testing was user acceptance testing. This was done since most of the requirements that needed more testing were non-functional GUI requirements. Since the GUI is to be used by the user in the future, we believed it was only natural that user feedback would be most valuable.

The overall feedback of our user acceptance testing is largely positive and can be further observed within the user acceptance table.

9 Team Organization

Table 1: Team Organization

Team Member	Main Focus
Björn Schönrock	• CI executable Windows and MacOS • File management of code base • GUI features – Icon for main window

Kaitlin Vos	• GUI features
	  – Tooltips
	  – Splash screen
	  – Drag and drop
	  – Shortcuts
	  – Parameters tab
	  – Guide menu
	  – Paths tab
	• Testing
	  – SonarQube
	  – Unit testing `pipeline.py`, `files.py` and `app.py`
	  – Acceptance testing
	• File processing
	• Removing all code smells
	• Configuration file saving and loading
	• Documentation

Tudor Dragan	• Code documentation • Unit testing of `params.py` • Documentation

Continued on next page

Table 1: Team Organization (Continued)

Team Member	Main Focus

Mohammed Nacer Lazrak	• GUI features
	  – Drag and drop
	  – `Select All` button
	  – Default parameters
	• Unit testing `app.py`

Dominic Therattil	• GUI features
	  – Logo
	  – Dark mode
	  – Resizeable window & full-screen
	  – File menu
	  – Default parameters
	• Use cases
	• Update `PASTAQ` Tutorial

Teresa Ferreira	• GUI features – Removed the cancel button • Use cases
Cristian Iacob	• Documentation • Traceability matrix

A Class Diagram

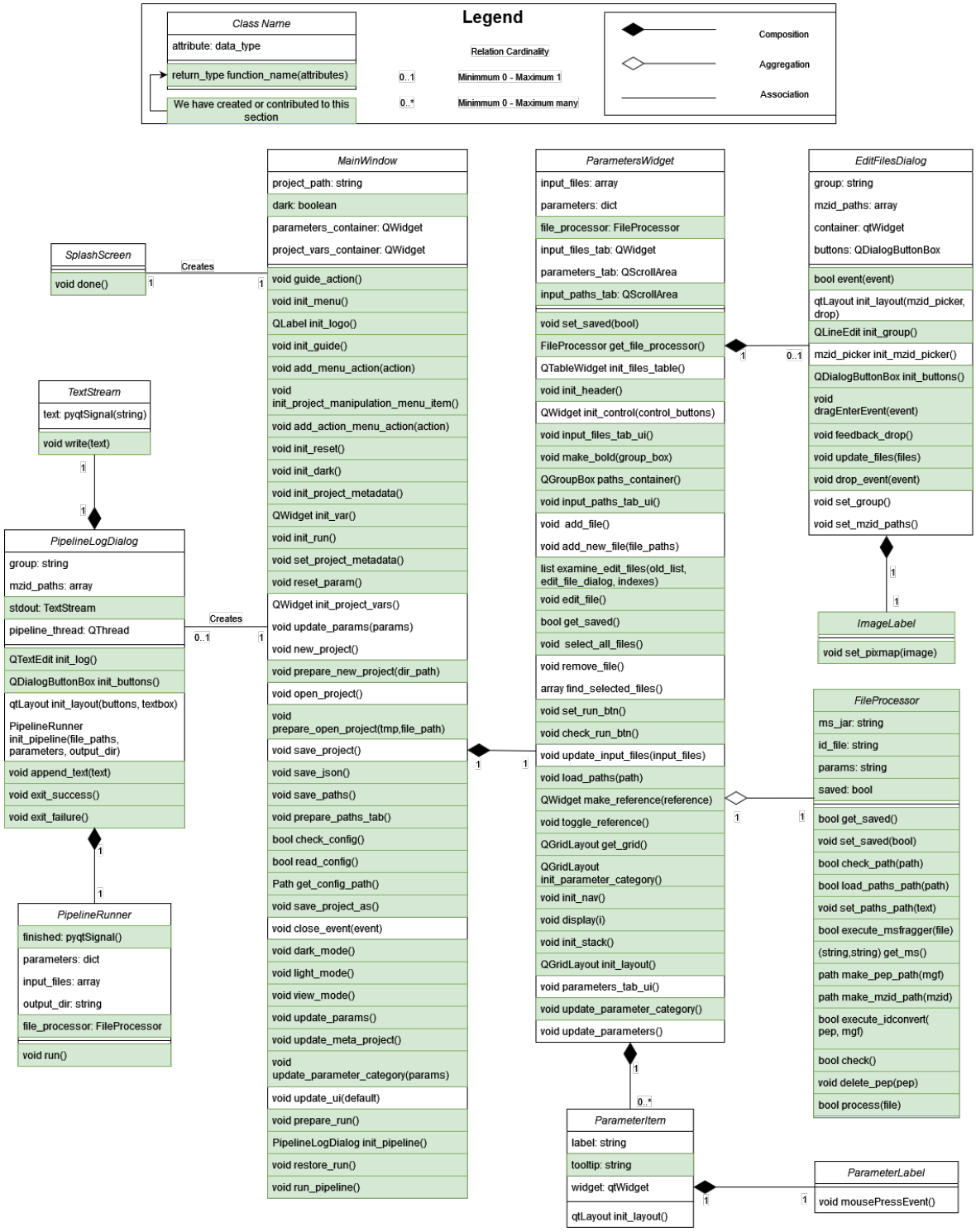


Figure 18: Architecture Class Diagram

B Terminology

Glossary

.FASTA A text-based format for representing either nucleotide sequences or amino acid (protein) sequences, in which nucleotides or amino acids are represented using single-letter codes. 7

.mgf Mascot generic format files are the standard format to store protein data. 2, 7, 13

.mzID The .mzID format stores peptide and protein identifications based on mass spectrometry and captures metadata about methods, parameters, and quality metrics. 2, 7, 13

.params The .params file is needed to execute MSFragger. It contains the path to the FASTA database and the search parameters for MSFragger. 6, 7, 13

Github Actions Automate, customize, and execute the software development workflows right in the repository with GitHub Actions. 16

idconvert A command line tool for converting between various file formats. 2, 3, 6, 7, 13

MSFragger Database search tool for peptide identification in mass spectrometry. 2, 3, 6, 7, 13

PASTAQ Pipelines and Systems for Threshold-Avoiding Quantification; PASTAQ provides a set of tools for high-performance pre-processing of LC-MS/MS data. 8

QAction This class provides an abstract user interface action that can be inserted into widgets. 9, 10

QCheckBoxes This widget provides a checkbox with a text label. 11

QDialog This class is the base class of dialog windows. 10, 11

QFileDialog This class provides a dialog that allow users to select files or directories. 10, 11

QGroupBox This widget provides a group box frame with a title. 12

QKeySequence This class encapsulates a key sequence as used by shortcuts. 9

QLabel This widget provides a text or image display. 11

QLineEdit This widget is a one-line text editor. 11

QListWidget This class provides an item-based list widget. 11, 12

QMainWindow This class provides a main application window. 13

QMenuBar This class provides a horizontal menu bar. 9

QMessageBox This class provides a modal dialog for informing the user or for asking the user a question and receiving an answer. 8

QPalette This class contains color groups for each widget state. 10

QPushButton This widget provides a command button. 10, 11, 13

QScrollArea This class provides a scrolling view onto another widget. 12

QSplashScreen This widget provides a splash screen that can be shown during application startup. 8

QStackedWidget This class provides a stack of widgets where only one widget is visible at a time. 12

QTableWidget This class provides an item-based table view with a default model. 11

QVBoxLayout This class lines up widgets vertically. 10

QWhatsThis class provides a simple description of any widget, i.e. answering the question "What's This?". 10

References

[1] Mohammad Robihul Mufid et al. "Design an MVC Model using Python for Flask Framework Development". In: *2019 International Electronics Symposium (IES)*. 2019, pp. 214–219. DOI: 10.1109/ELECSYM.2019.8901656.

C Change log

Table 2: Change Log

Date	Time	Team Member	Changes	Version
12.2.2022	13.58	Kaitlin Vos	Inserted change log table	1.0
14.03.2022	13:33	Kaitlin Vos	Added structure and team organization	1.1
15.03.2022	23:50	Cristian Iacob	Added title page, QA diagram & descriptions for technology stack	1.2
16.03.2022	00:47	Kaitlin Vos	Detailed structure & added technology stack	1.3
21.03.2022	17:22	Cristian Iacob	Wrote introduction & moved QA diagram to introduction	1.4
21.03.2022	22:57	Kaitlin Vos	Added tooltip section & organized the document	1.5
26.03.2022	15:00	Björn Schönrock	Started CI section	1.6
28.03.2022	02:06	Cristian Iacob	Fixed the introduction and quality assurance sections & moved sections so they follow more logically	1.7
29.03.2022	19:01	Kaitlin Vos	Completed the tooltip and file procedure sections	1.8
30.03.2022	15:00	Björn Schönrock	Worked on the CI section & fixed document structure	1.9
31.03.2022	12:39	Kaitlin Vos	Made file procedure align with presentation	1.10

Continued on next page

Table 2: Change Log (Continued)

Date	Time	Team Member	Changes	Version
01.04.2022	12:05	Cristian Iacob	Fixed document formatting, added stability section and rewrote supported OS subsection	1.11
01.04.2022	13:15	Cristian Iacob	Moved supported OS subsection to QA & changed stability section to major errors	1.12
01.04.2022	13:30	Cristian Iacob	Reformatted sections based on priority	1.13
01.04.2022	16:10	Cristian Iacob	Wrote GUI rework beginnings	1.14
01.04.2022	16:30	Cristian Iacob	Wrote abstract section	1.15
02.04.2022	14:27	Teresa Ferreira	Added architecture overview diagram	1.16
02.04.2022	22:27	Tudor Dragan	Added a new architecture overview diagram	1.17
03.04.2022	23:46	Kaitlin Vos	Worked on the GUI rework section and made a diagram for file procedure	1.18
04.04.2022	11:13	Kaitlin Vos	Proof-read entire document, fixed typos, syntax, logic	1.18.1
29.04.2022	13:50	Cristian Iacob	Made the change log consistent	1.18.2
01.05.2022	15:00	Björn Schönrock	Updated CI section	1.18.3
08.05.2022	16:00	Björn Schönrock	Updated CI section with testing	1.19
08.05.2022	17:00	Tudor Dragan	Created and wrote threading section	1.19.1
09.05.2022	10:00	Tudor Dragan	Added architecture class diagram	1.19.2
09.05.2022	10:10	Cristian Iacob	Changed document formatting, wrote file processing section, added to input parameters section	1.19.3
09.05.2022	10:25	Cristian Iacob	Linked requirements with QA section	1.19.4
26.05.2022	14:55	Kaitlin Vos	Restructured document into MVC sections, wrote the sections model, view and controller	1.19.5
29.05.2022	12:25	Cristian Iacob	Added to QA section & added CI illustration	1.19.6

Continued on next page

Table 2: Change Log (Continued)

Date	Time	Team Member	Changes	Version
31.05.2022	23:25	Kaitlin Vos	Rewrote unit test section	1.19.7
02.06.2022	18:28	Kaitlin Vos	Introduction & added diagram for MVC, data and model logic	1.19.7
09.06.2022	02:19	Kaitlin Vos	Replaced most diagrams, adjusted MVC text	1.19.8
09.06.2022	15:19	Kaitlin Vos	Small changes, made glossary, specified team organization	1.19.9
13.06.2022	18:28	Cristian Iacob	Updated class diagram and architecture overview diagram	1.19.10
14.06.2022	15:14	Kaitlin Vos	Added sequence diagram, redid executable diagram, added references	1.20
15.06.2022	19:26	Tudor Dragan	Updated Architecture Overview Diagram	1.20.1
16.06.2022	13:12	Cristian Iacob	Updated Class Diagram	1.20.2
16.06.2022	23:14	Kaitlin Vos	Updated Architecture Overview Diagram for final deliverable, added sequence diagram for 5.5.1, updated glossary, enabled hyperlinks, moved class diagram to appendix	1.20.2