DataMole Developer Manual

Alessandro Zangari

October 3, 2020

Document content

This technical document is organised in two parts:

- Chapter 1 describes the software architecture, as well as the considerations involved in DataMole development;
- Chapter 2 provides a practical guide to extend this software with new features.

Contents

List of Figures

Chapter 1

DataMole architecture

1.1 Technology

This section describes the main technologies that were selected to realise DataMole. DataMole was developed in Python in order to maximise interoperability with other packages and to take advantage of the existing data analysis libraries. The main dependencies are listed below:

- **Pandas 1.0.5**: Pandas dataframe is used to manage datasets and its rich API is used in the program to apply transformations and manipulate the data;
- **Scikit-learn 0.23.1**: this library is used to apply some transformations, because of its excellent compatibility with Pandas;
- **Networkx 2.4**: a library for graph management and analysis used to create and manage the computational graph;
- **Numpy 1.19.1**: a library for scientific computing and number crunching;
- **PySide2 5.15.0**: contains the Python bindings of the Qt Framework version 5.15.0 and is used to create the graphic interface;
- **Prettytable 0.7.2**: a library for printing formatted ASCII tables, used with logging;
- **PyTest 5.4.3**: to write tests;
- **Sphinx 3.2.1**: used to automatically generate the documentation for the project.

1.1.1 Dataset management libraries

Pandas dataframes are used in DataMole to manage datasets, but other libraries offering similar data structures that were taken in consideration and are listed here.

- **Dask**: a project started in 2015 with the goal of creating a distributed computing library with big data support. It offers a dataframe API very similar to Pandas, with the difference of being able to manage huge out-of-memory datasets;

- **Datatable**: a Python porting of the popular `data.table` package for R, developed by the same authors. Only a beta version is currently released, and many features of the R version are missing. Similarly to the R package it can deal with out-of-memory datasets;
- **PySpark**: a big data computing framework written in Scala which supports data stream, map-reduce operations on distributed file systems;
- **Turicreate**: a project currently maintained by Apple with the goal of simplifying the development of custom machine learning models. It supports big data computations and provides the SFrame container, a scalable dataframe similar to a Pandas dataframe;
- **Modin**: a project with the goal of scaling Pandas to big datasets. Internally is uses Dask or Ray to transparently process dataframes and to support out-of-core and parallel computations.

Both `Spark` and `Dask` are designed to work with big data and heavy computations: internally they use a scheduler and add complexity to support distributed file systems. This focus on big data often results in additional computational overhead, which is unjustified for the purpose of this project at its current state.
`Datatable` and `turicreate` are much simpler to use than the previous libraries, since they do not handle distributed computation. `Datatable` main disadvantage is its very limited API with respect to Pandas and the missing support for Windows. `Turicreate` is more limited then Pandas, but still comes with a rich set of features that makes it probably suitable for this project. It has one limitation: it works only with 64-bit machines.
Finally `Modin` was discarded because it is still an experimental project.
Adding big data support is of course desirable, but comes with additional costs and overhead that should be taken into account. With respect to this project objectives, I found the additional complexity not worth it, especially for an initial release, and decided to work with Pandas. This library has also the advantage of being well integrated in the Python ecosystem, and its dataframes are supported by other packages like Scikit-learn. Naturally the other listed libraries may be reconsidered for future extension, if scalability becomes a concern.

1.2 Qt basics

This section briefly describes two important features of Qt that have been widely used to develop DataMole. The purpose of this section is not to provide a comprehensive description of the Qt Framework and its functionalities, but merely to clarify the meaning of some technical terms that will often be used throughout this chapter.

1.2.1 Signals and slots

Signals and slots are used for communication between objects and their combined usage represents the Qt approach to event-driven programming [8].
A *signal* is emitted when a particular event occurs. For example when a button is clicked, the `clicked` signal is emitted. A *slot* is a member function that is called in response to a particular signal. Qt widgets come with many predefined signals and slots, but they can be subclassed to add new customised slots handling specialised signals.

1.2.2 Model-view-delegate

The Qt Framework provides a set of classes that use a model/view architecture to manage the separation between data and the way it is presented to the user [7]. This is achieved with the combined usage of three components:

- *Model*: model classes inherit `QAbstractItemModel` that provides an interface for the other components in the architecture. The model communicates with the data: it can hold the data or it can be a proxy between the data container and the other components;
- *View*: view classes inherit `QAbstractItemView` and their instances obtain references to items of data from the model. With these, a view can retrieve items from the data source and display them to the user;
- *Delegate*: a delegate renders the items of data inside a view and, when items are edited, it communicates with the model to change its state. Every delegate class inherits `QAbstractItemDelegate`.

These components define the *model-view-delegate* pattern, which is extensively used to show information in DataMole.

1.3 Architecture overview

This section describes the architectural choices made while designing the DataMole software architecture. UML class diagrams are presented alongside packages description to model the relation between different components. Moreover, several sequence diagrams are shown in §1.3.4.5 to show the interaction between GUI components that is required to carry out some operations.
The extension mechanisms, some implementation details, as well as a detailed description of various classes and their methods, are included in the next chapter.

1.3.1 Description of the main packages

DataMole is a Python package, composed of many sub-packages and modules. In Python notation a *module* is a single file containing definitions and statements, while a *package* is a collection of modules (e.g. a directory containing Python files and possibly other packages) [2]. The complete directory structure can be seen later, in §2.1.
The main DataMole package, named `dataMole`, is organised in 4 sub-packages:

- *data*: defines the container classes for dataframe objects and everything required for their management. Every Pandas dataframe is wrapped inside `Frame` object. Every dataframe has a `Shape`, which is a description of the columns names and their types. Finally a hierarchy of classes was defined to model the supported data types;
- *flow*: defines the data structure used to contain the flowgraph (`OperationDag`) and the handler to execute it in separate threads (`OperationHandler`);
- *gui*: contains all the widgets and GUI components used within the software. It is further composed of 4 sub-packages:

  - *charts*: contains the widgets used to create and visualise all the charts;
  - *editor*: defines the abstract class `AbsOperationEditor`, the super-type of all editor widgets used for operation configuration. It also contains the definition of the editor factory `OptionsEditorFactory`, defined as a *singleton*, and many utilities to configure the operation and show the operation documentation;
  - *graph*: collection of components which realise the user interface for the *Flow panel*. It relies on the *Qt Graphics View Framework* and part of the implementation was taken from an existing work available at [1];
  - *widgets*: contains the definition of several widgets used in the program.

- *operation*: defines the common super-type of all operations and its subclasses, which model every data transformations;

- *flogging*: the DataMole logging module. Also defines the interface to be implemented in every operation whose execution must be logged to file (`Loggable`).

1.3.2 Model/view classes

Many classes that take part in *model-view-delegate* pattern are defined inside module `gui.mainmodels`. Fig. 1.1 shows the UML class diagram for some of its classes, along with their relationship to the `data` package.
Qt provides specialised models and views for displaying data in many different ways: within DataMole data is often shown in tables and lists. In order to take advantage of the Qt model-view system, model classes are required to subclass `QAbstractItemModel` and implement the relevant methods.
`FrameModel` handles the visualisation of the dataframe inside a `QTableView` (which is a view for tabular data).
`AttributeTableModel` works as a proxy model, keeping a reference to the `FrameModel` and showing only column names and types, hence is used to show the *shape* of a dataset, which is required, for instance, in the *Attribute panel*.
`AttributeProxyModel` further refines the data shown in an `AttributeTableModel` by adding filtering capabilities, like attribute search by name or regular expression and type filtering. This last feature is exploited in the operation editors that only allow the user to select the subset of columns with supported types.
When a dataset with thousands of columns or rows is visualised inside a table, the process of filling every table cell with the data from the model can take time, and the GUI would be unresponsive while this happens. Proxy class `IncrementalRenderFrameModel` solves this problem: it shows the exact same data as the `FrameModel` it proxies, but implements additional methods, specifically `fetchMore` and `canFetchMore`, to make sure the table is filled on demand, only when it is scrolled. When this happens it loads the next batch of 50 columns or 400 rows depending on the scrolling direction.
`SignalTableView` defines the view used to show tabular data in the program. It inherits `QTableView` capabilities and defines a customised *signal* used to correctly handle rows selection and deselection. This class is used inside the `SearchableAttributeTableWidget`, a reusable widget that shows data inside a table complete with a search bar above it.
Finally the module also defines a custom delegate class and a custom header for the table (not shown in Fig. 1.1), to be set in views that need to show a checkbox in one of their columns.

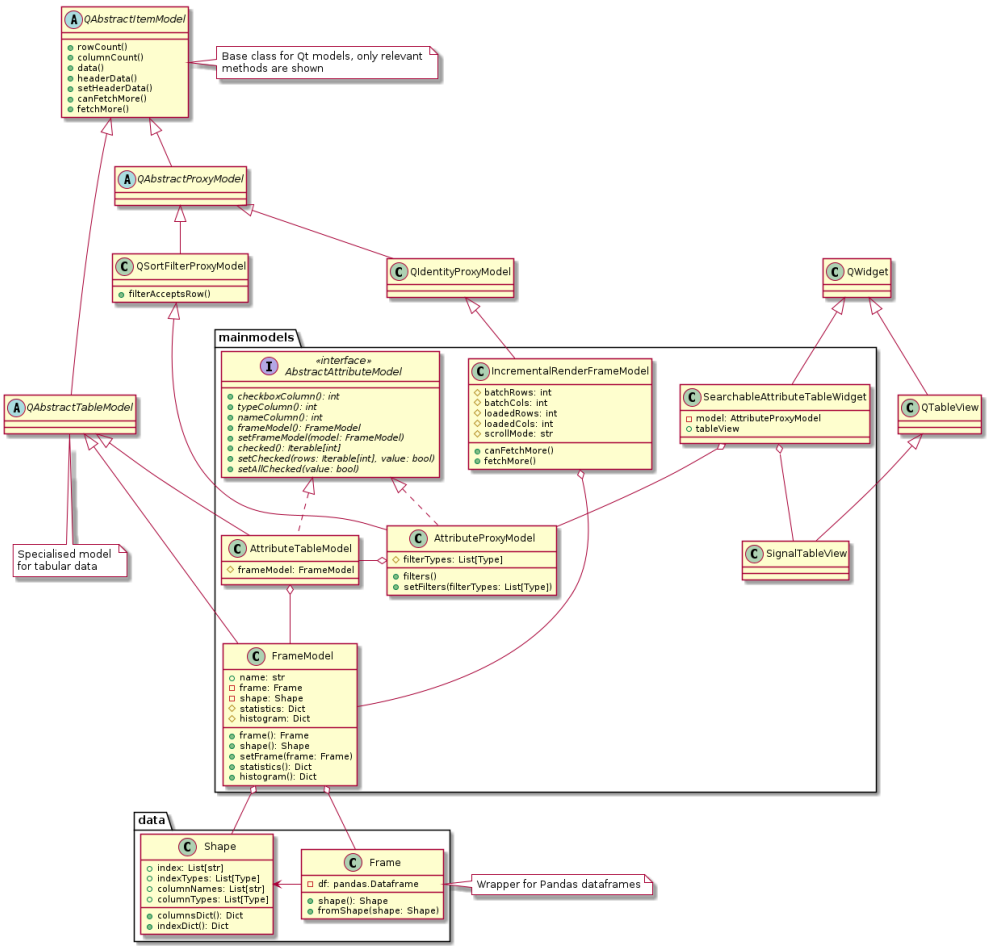

Figure 1.1: Class diagram of the `mainmodels` module

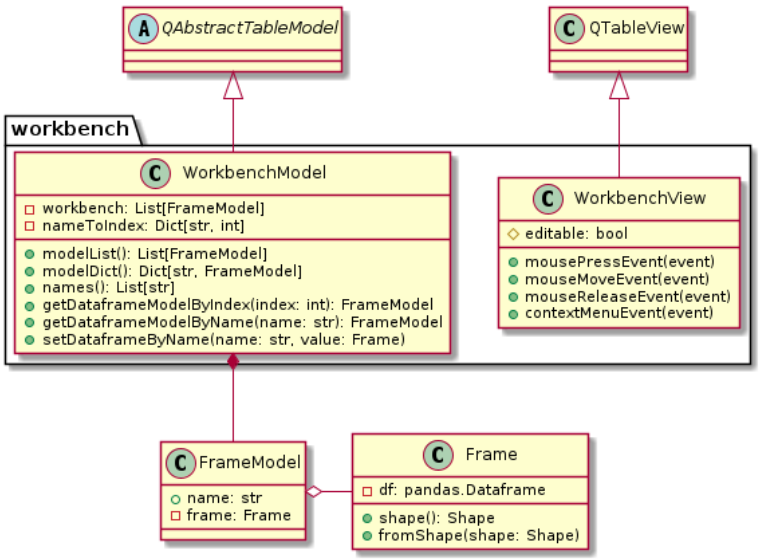

Figure 1.2: Class diagram of the `workbench` module

1.3.2.1 The workbench

As explained in the previous section every loaded dataframe is contained in a `Frame` object. These objects are wrapped in a `FrameModel` for visualisation inside Qt views. All these datasets are kept in the `WorkbenchModel`, the centralised container for dataframes (diagram in Fig. 1.2). It provides methods to access frame models by row number (index) or name, that makes it usable as a list-like or dictionary-like container. Its `setDataframeByName` method is called whenever a new dataset, which was loaded or created by applying operations, must be added to the workbench. The workbench is shown using its specialised view class `WorkbenchView`, which overrides some methods to allow item reordering, selection and the display of a menu on right click.

1.3.3 Representation of a dataset transformation

When defining a pipeline, every operation takes one or more datasets in input and produces a result that must be the input of the next transformation. Sometimes transformations must be parametrised with a variable number of arguments that represent the configuration of the operation. Additionally every transformation should be logged, in order to keep a trace of how a dataset was changed and, depending on the operation, it might be possible to undo it.
With these requirements it seemed convenient to implement a *command pattern*: every *data transformation*, also called *operation* in this document, is encapsulated in a object of type `Operation`, whose contract is described in the next section.

1.3.3.1 The *Operation* abstract class

Every operation defined in DataMole is a concrete subclass of `Operation`. Fig. 1.3 shows the `Operation` abstract class and its subclasses.

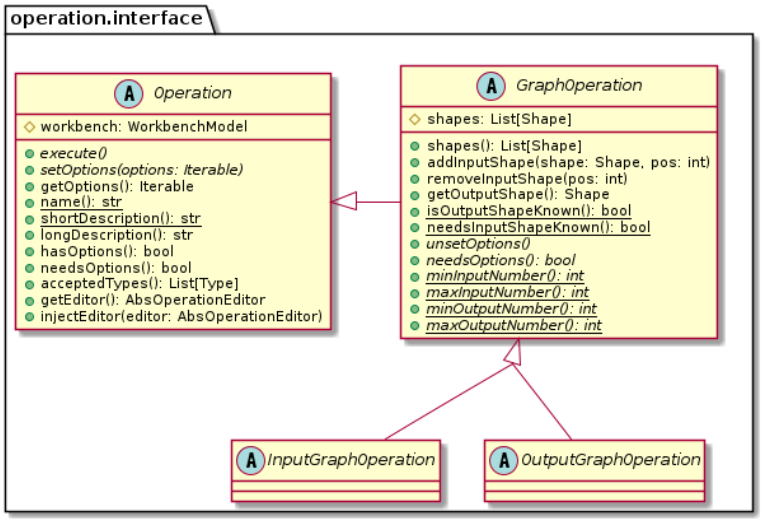

Figure 1.3: Hierarchy of Operation abstract classes

An operation keeps a reference to the workbench, which provides access to any loaded dataset. Every operation has a name, comes with a short and optionally a long description to be shown in the *editor widget*. The long description is shown on user request on a separate help panel, and contains a detailed description of how the operation works and how it should be configured.
The `execute` method is run to apply the operation to its input arguments, which are passed as parameters.
The `getOptions`/`setOptions` methods are reimplemented in every operation to respectively get and set its arguments. If required, `setOptions` can also validate the new options before setting them. If they are not correct it raises an exception of type `OptionValidationError` which is handled by the controller. This mechanism is further described in §1.3.4.5.5.
Methods `hasOptions` and `needsOptions` return a boolean value depending on whether the operation is configured or needs an option editor widget. If the latter is true, the operation also needs `getEditor` to be overridden to return the widget that the operation will use to be configured. In order to avoid the definition of a new widget for every operation, an *editor factory* was provided and can be used to quickly create editors with standard option fields (checkboxes, combo boxes, line edits, etc.).
While `getEditor` only returns a new editor widget, the `injectEditor` method is used to configure the editor.
Finally the `acceptedTypes` method defines which types are accepted by the operation.

1.3.3.2 The editor widget factory

Most operations require options that must be supplied by the user. For example the `KBinsDiscretizer` operation requires the user to select the columns to be discretized, specify the number of bins for every column and select the strategy to be used. To configure an operation, the widget returned by method `getEditor` is used. For most operations this widget is composed of a table with some editable columns and some radio buttons or checkboxes. To avoid code duplication and to speed up the process

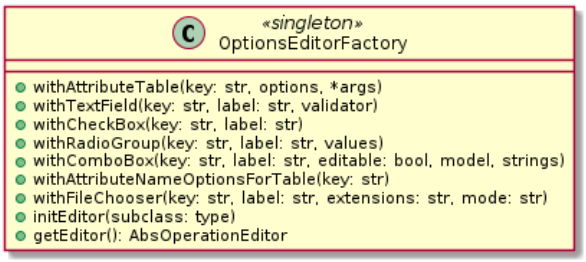

Figure 1.4: Widget factory class specification

of creating widgets, a factory class was defined. Fig. 1.4 gives an overview of the class methods.
Factory methods allow appending widgets to the editor layout stacking them vertically, thus the order of invocation determines their ordering. The factory is a *singleton* object that must be re-initialised every time it is used by invoking the `initEditor` method. This method also accepts an optional `subclass` parameter with the type of the editor to be created. This is particularly useful when an editor needs signals or slots to be part of its class definition. To give an example, Fig. 1.5 shows the editor for the *Fill NaN* operation, created by calling the following factory methods:

- `withAttributeTable`: this method creates widget *(A)*, which is a table showing every attribute name and type, with a configurable number of extra editable columns used to receive options for every attribute. In this example the table requires the values to be filled if the option to fill by value is selected (column *Fill value*);
- `withRadioGroup`: adds a set of exclusive radio buttons and a descriptive title *(B)*: in the example it is used to select the strategy for filling missing values.

Further details about the factory usage and methods configuration are reported in the next chapter, in §2.2.4.2.
Of course not every widget can be created with the factory: complex editors with particular requirements must be defined manually. For instance, this is the case for the `Join` and `ExtractSeries` operations.

1.3.4 The computational graph

As explained in §1.3.1, the package `gui.graph` defines the components of the graphical user interface which allow the user to interactively create a computational graph of operations: new nodes (i.e. operations) can be added, moved, deleted and connections can be created between existing nodes to create chains of operations. Hence, this interactive flowchart can be seen as a *pipeline* of data transformations, internally modelled as a *directed acyclic graph*. The class diagrams of the `gui.graph` and `flow` packages are depicted in Fig. 1.6.

1.3.4.1 Pipeline laziness

Whenever a new operation is added, it is not executed immediately: the pipeline is *lazy*, meaning that it is evaluated completely only when the user requests it. Thus

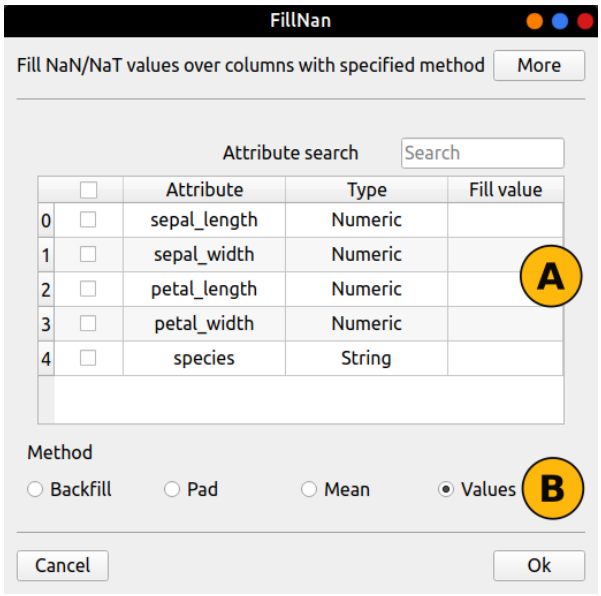

Figure 1.5: Example of an *option editor* whose body widgets (A and B) are created with the widget factory

every node with an ancestor does not know its input until the whole pipeline is executed. However most operations require to be configured with additional arguments through theirs editor widgets, and often this arguments depends on the operation input (for example the user may be asked to select the columns to transform), so in order to do this they *need* some information about their inputs. Additionally, every time the operation is configured, its options are validated, process that often depends on the column and index types. To solve these problems every operation needs to be able to describe some properties of its output before execution, provided a description of its inputs and the required options. Specifically, it should describe the *shape* of its output. The *shape* of a `Frame` object is encapsulated inside the type `Shape`: it includes information about the name and type of every column, including the ones used as dataframe index. How the dataset shape is propagated through the graph is described in the following sections.

1.3.4.2 The *GraphOperation* abstract class

The additional requirements of the operations that need to execute inside the pipeline motivated the definition of the `GraphOperation` class, that specialises `Operation` adding methods only relevant in the pipeline context. Every graph operation needs to define how many inputs it needs and how many output connections it supports. This information is conveyed by the `minInputNumber`, `maxInputNumber` and their respective counterparts for the output number.
Moreover every graph operation reimplements method `getOutputShape`, which returns the `Shape` object describing the names and types of every column in the dataset after the application of the operation. Because this information needs to be propagated through the graph, operations also provide the `addInputShape` and `removeInputShape` methods, to manage the shapes object when some configuration changes or connec-

tions are removed.
However, there are situations when it might not be possible to predict the output shape before the operation is run. If this is the case, method `isOutputShapeKnown` can be redefined to return *False*. This is the case for the `RemoveBijections` operation, which automatically removes every column which is a bijection of the others. Conversely some operations might not require knowing in advance the shape of the input coming from their incoming connections, in which case `needsInputShapeKnown` should be redefined to return *False*. For example `RemoveNanRows`, the operation used to remove every row with a certain ratio of missing values, does not need to know the input dataset shape because its output shape does not depend on it: since the operation just remove rows, the output shape does not change at all.

1.3.4.3 The graph data structure

The operations added to a pipeline are stored inside a `DiGraph`, a data structure that models a direct acyclic graph, provided by `networkx`, a very popular network analysis library. This data structure is wrapped into an `OperationDag` object, that contains some helper functions to manage the graph of operations.
Before choosing to implement a simple computational graph using `networkx`, these Python libraries specific for creation and management of computational graphs were also considered:

- **Luigi**: a package that simplifies the creation of pipelines of batch jobs;
- **GraphKit**: a library used to manage and run graph of computations (DAGs);
- **Dask**: this library offers many tools for scalable computations and its schedulers support the execution of customised task graphs.

Eventually we decided to avoid using such libraries, since computations in DataMole are not particularly heavy and thus would not have benefited from the advanced features provided by these packages. Additionally integrating them in a graphical interface would require much work and time, because their API is not made for this use case, but rather for usage in Python scripts.

1.3.4.4 The GUI for the graph

GUI components used to manage the graph are defined in the `gui.graph` sub-package. This package makes use of the *Qt Graphics View Framework*, the Qt module for efficient rendering and management of graphic items [6]. Many modules of this package were adapted from [1], a project that used Python and Qt to build a generic graph manager, and is licensed under GNU GPL.

1.3.4.4.1 Main graphic components

The *Qt Graphics View Framework* relies on three main classes:

- `QGraphicsItem`: represents a graphic item that can be shown within a graphics scene;
- `QGraphicsScene`: the scene manages all the graphic items and provides functionality to efficiently determine items location, and control zooming and selection;
- `QGraphicsView`: visualises the content of a `QGraphicsScene`.

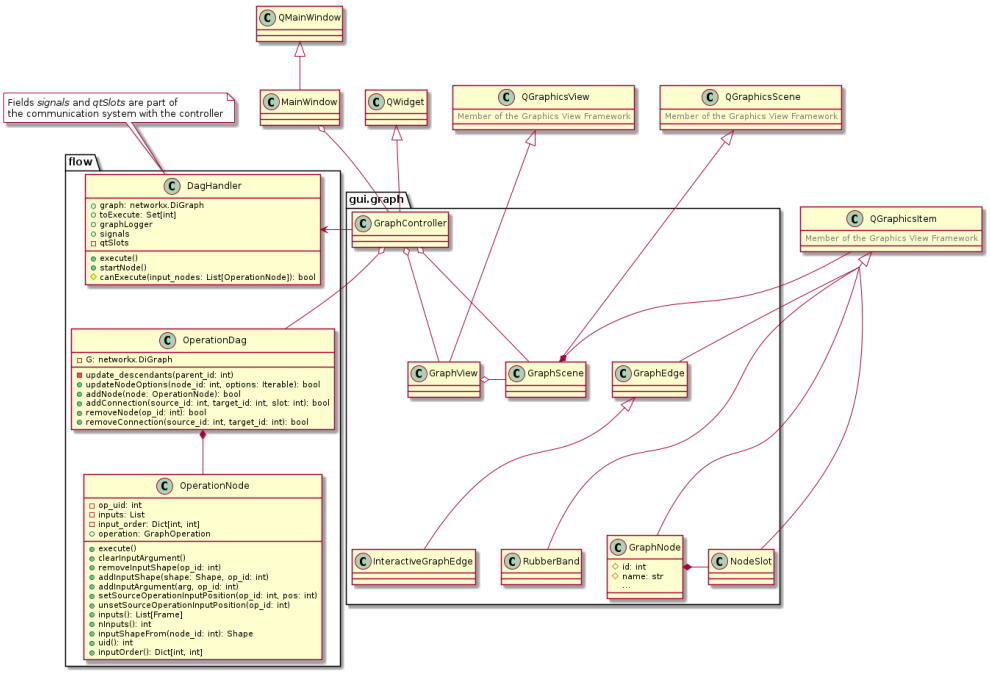

Figure 1.6: Class diagram of the `gui.graph` and `flow` packages

Fig. 1.6 shows the classes defined in the `gui.graph` package. All graphical items are contained in the `GraphScene` a subclass of `QGraphicsScene` that additionally handles drag-and-drop actions and mouse events. Graph *nodes* and *edges* are respectively drawn using the `Node` and `Edge` classes, both subclasses of `QGraphicsItem`. Every `Node` has a unique id that matches the id of the operation it represents. The `NodeSlot` item is used to draw the circular sockets that represent the operation inputs and outputs and are used to create connections. Finally `RubberBand` is used to handle mouse selection.
A *controller*, modelled by class `GraphController`, is used to ensure that the graphic interface is kept synchronized with the underlying data structure containing the graph. Every graph node is an object of type `OperationNode` which wraps a `GraphOperation` and adds some helper methods used to manage pipeline nodes.

1.3.4.5 Pipeline management workflow

The `GraphController` interprets the user actions on the view and keeps updated the graph data structure inside the `OperationDag` object. This section describes how the controller and the other classes interact with each other to manipulate the pipeline.

1.3.4.5.1 Node creation

Nodes are placed on the graphic scene with drag-and-drop, by selecting operations from a list and dropping them on the graphic view. When a new operation is dropped, method `dropEvent` of `GraphScene` is called and the *scene* invokes method `getDropData` to retrieve the type of the operation that was dropped. It then emits a signal to ask the *controller* to manage the creation of the new operation. The

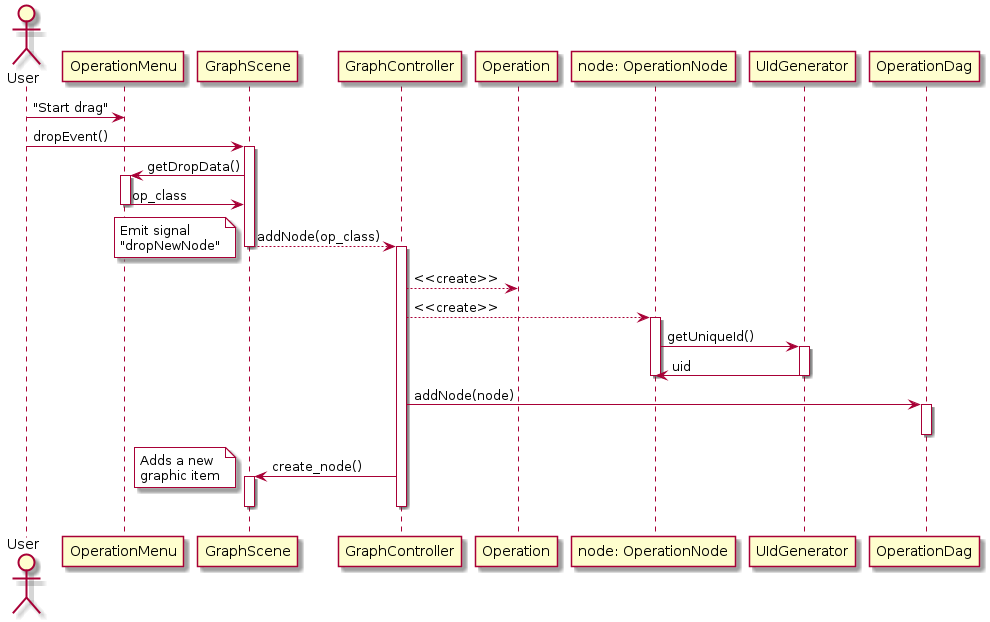

Figure 1.7: Sequence diagram describing the creation of a new pipeline node

controller instantiates the new `Operation` and `OperationNode` objects, updates the graph with the new operation and, if successful, it updates the graphic scene with the new node. This entire process is depicted with the sequence diagram in Fig. 1.7. On the other hand, if something goes wrong and the pipeline cannot be updated, the controller shows an error message and nothing is changed.

1.3.4.5.2 Edge creation

When the user starts to drag a new edge from a `Node`, it emits a signal and causes the `start_interactive_edge` method of the *scene* to be called. Its purpose is to display an interactive draggable edge that exits the source node and follows the mouse pointer until the user drops its head on the target node. When this happens `stop_interactive_edge` is called. This method looks for a free node slot in the target node. If a free slot is found the `addEdge` method of the *controller* is called to update both the acyclic graph and the graphic view accordingly. The corresponding sequence diagram is shown in Fig. 1.8.
If the edge cannot be added, for example because the source operation does not provide an input shape which is needed by the target operation, an error message is shown to the user. In either case the temporary `InteractiveEdge` object created to display the draggable edge is eventually deleted.

1.3.4.5.3 Operation configuration

Double clicks on existing operations cause the editor widget to be shown. Fig. 1.9 describes how the widget is created and configured. First the *controller* retrieves the `Operation` object corresponding to the clicked item using its unique id. Afterwards the editor is created using the `getEditor` method and the existing options are retrieved

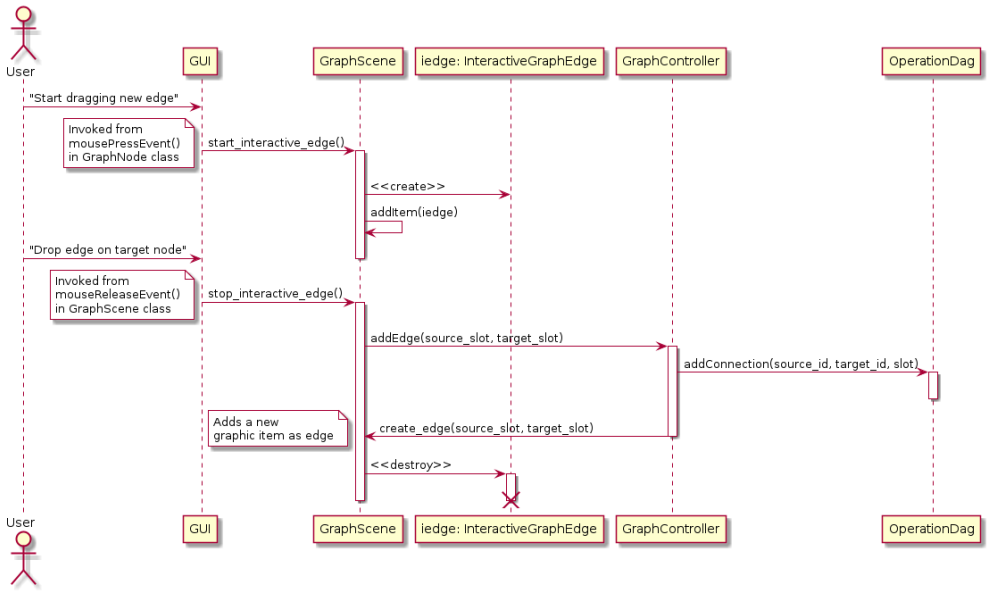

Figure 1.8: Sequence diagram for connecting two existing operations

from the operation and set into the editor widget. Finally the editor is configured with the `injectEditor` invocation and shown to the user.

1.3.4.5.4 Option confirmation

Once the user sets new options in the configuration widget and confirms them, the `accept` signal is emitted by the editor. This signal is handled by the `GraphController` that retrieves the options from the editor and updates the corresponding operation. Finally the `update_descendants` routine is run to propagate the new shape through the graph. The whole process is shown in Fig. 1.10.

1.3.4.5.5 Option confirmation with validation errors

The `setOptions` method defined in every operation can raise an exception of type `OptionValidationError` if one or more options are not correct and need to checked by the user. This exception is parametrised with a list of tuples, where each one represents an error, with a key and an error message for the user. After the user confirms the options in the configuration widget, eventual validation errors are notified inside the editor window. The process leading to this outcome is outlined in Fig. 1.11. Error messages are normally shown on the bottom of the editor widget. If a more advanced error handling is required, it is possible to change this behaviour by defining custom error handling functions. This is detailed in §2.2.4.

1.3.4.6 Executing the pipeline

Execution of the pipeline is handled by the `DagHandler` class. This handler is instantiated by the controller to direct the execution of the whole pipeline. First it checks for runnable operations: an operation is considered *runnable* if all the required options and its input dataframes are set. Starting from the input nodes, runnable operations

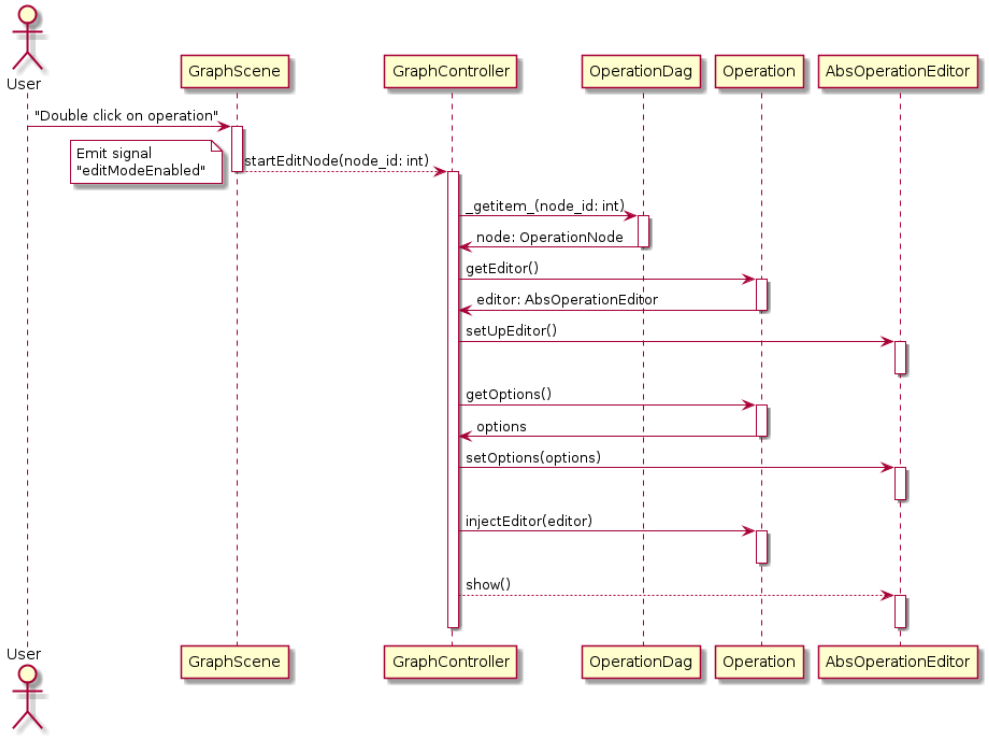

Figure 1.9: Sequence diagram for operation configuration

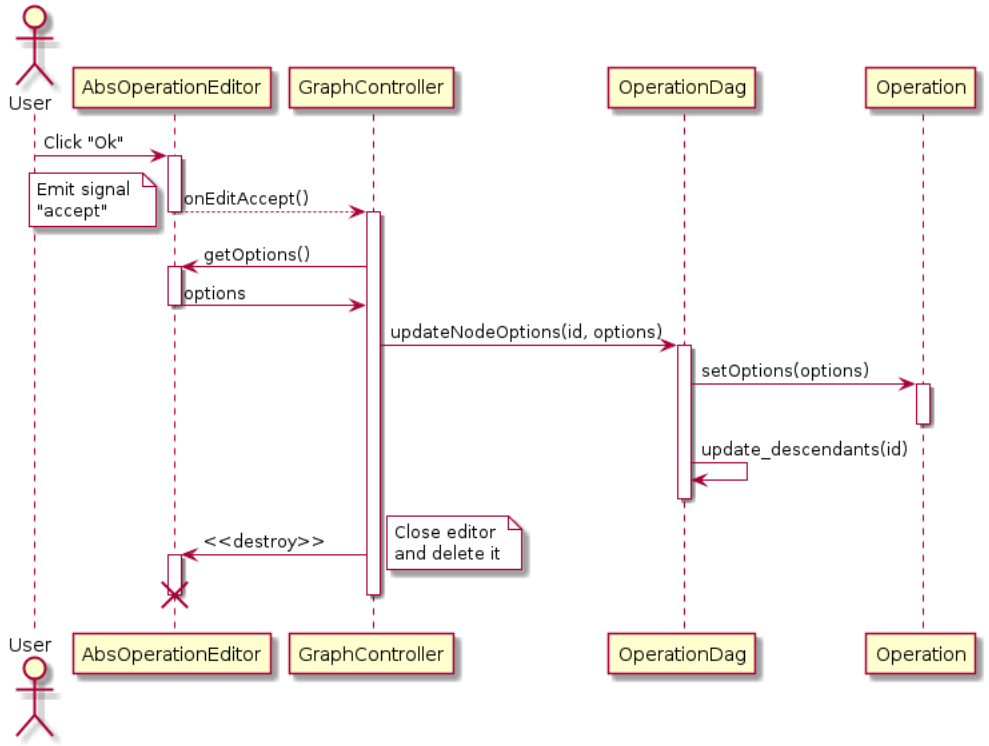

Figure 1.10: Sequence of operations after options confirmation assuming no validation errors occur

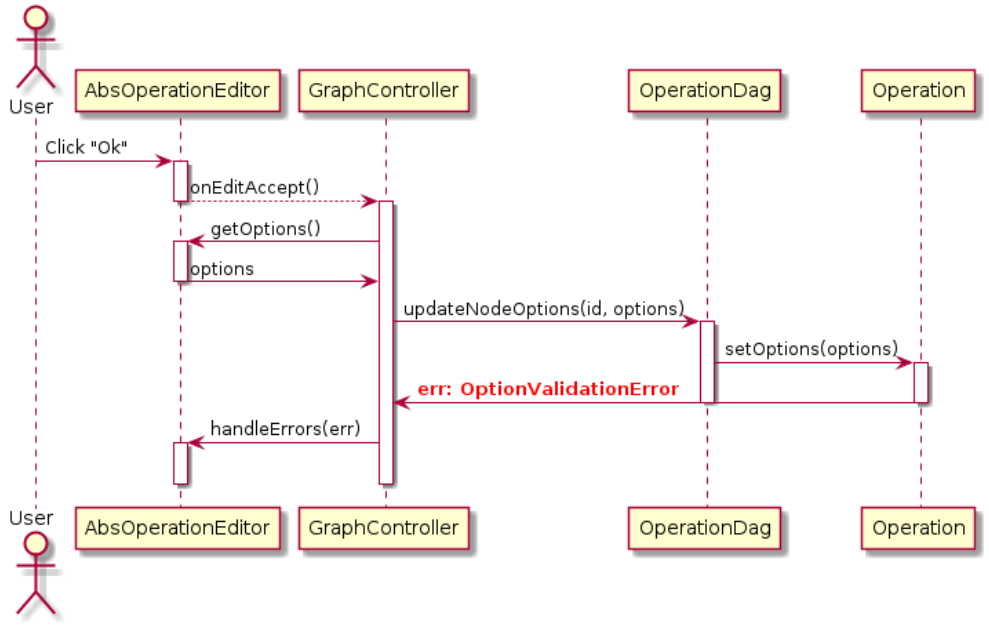

Figure 1.11: Sequence of operations after options validation exception

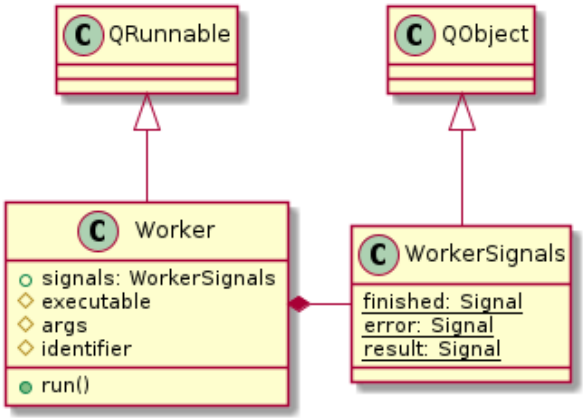

Figure 1.12: Class diagram of the `threads` module

are scheduled for execution on a separate thread. When an operation completes, its output is set as the input of all its successors and all runnable operations are started. Additionally the handler communicates with the controller emitting a specific signal when the status of an active operation changes (i.e. if it is running, has completed with error, etc.). This allows the controller to update the status of each node in the graphic view.

1.3.4.6.1 Multithreaded execution

In a Qt application, all QWidgets run in the *main thread*, also called the *GUI thread* [11]. If a time-expensive computation is run on this same thread the GUI will freeze and stop responding until the operation completes. To avoid this problem every operation to be executed is wrapped inside a `QRunnable` object and is added to a `QThreadPool` for execution. The combined usage of `QRunnable` and `QThreadPool` represents a simple multithreading pattern in Qt, with the advantage that the `QThreadPool` takes care of thread management, including their creation and destruction: it creates a predefined number of threads equal to the number of cores of the machine and reuses them whenever a new operation is added to the pool. Considering that thread creation and destruction can be costly, this pattern provides an higher-level alternative to thread management with the `QThread` class, which would instead create a new thread every time, without reusing them [10].
The helper class `Worker` was defined in the `threads` module, and is shown in Fig. 1.12. `Worker` objects wrap *executable objects*, which are arbitrary Python objects that define an `execute` method (like `Operation` instances). The worker is provided the list of arguments that should be passed to the `execute` method, if any, and an `identifier` to be used when a signal is emitted. A `WorkerSignal` class inheriting `QObject` is defined because `QRunnable` does not inherit `QObject` and thus can not emit signals. Through this class, workers emit three signals, depending on their state:

- *error*: this signal is emitted then the worker was running the operation but a runtime error occurred. The signal arguments are the operation identifier and the error information;
- *result*: emitted then the operation completes successfully. It carries the oper-

ation result (i.e. the return value of the `execute` method) and the operation identifier;

- *finished*: signal emitted immediately before the `run` method of the worker returns, and carries the identifier.

These signals provide a way to monitor the status of every operation and are used by the `GraphController` to update the graphic view.

1.3.4.6.2 Considerations and alternatives

The `QRunnable` and `QThreadPool` pattern is a very simple multithreading pattern, and indeed was chosen for its simplicity. By comparison, the `QThread` class provides much more flexibility, at the cost of explicitly managing threads. The main drawback of the selected approach is the lack of support for stopping a running operation. This comes from the fact that `QThreadPool` does not expose the underlying threads and thus stopping them becomes impossible.
On the other hand, an equivalent approach using `QThread` may involve defining a customised thread pool that keeps track of the running threads, in order to terminate their execution when required. Nonetheless, caution is required when stopping a running thread as it may leave data in an inconsistent state.
Using the Qt Concurrent module would be a simpler alternative: this module, part of the Qt Framework, provides high-level functions to deal with some common parallel computation patterns [4]. Unfortunately this module is not included in the Python bindings for Qt. That is because the Qt Concurrent heavily relies on C++ templates and due to the Python dynamically-typed nature and the impossibility of generating C++ code at runtime there is not way to generate the Python bindings.

1.3.5 The *OperationAction* controller

A controller class named `OperationAction` was designed to manage the application of operations from the *Attribute panel*. This class inherits `QAction` which provides the support for launching operations from menu bars.
An `OperationAction` wraps an `Operation` and when *triggered*, either explicitly or implicitly (e.g. by clicking an action in a menu), it shows the option editor and waits for the user to set them. The whole process is described with a sequence diagram in Fig. 1.13. After confirmation, if validation succeeds the operation is executed: to do this a `Worker` object is created and added to the `QTreadPool` global instance. The sequence diagram for this is shown in Fig. 1.14. If option validation fails, errors are shown as already discussed in §1.3.4.5.5.
Because every `GraphOperation` is also an `Operation`, the `OperationAction` wrapper can also run graph operations as single commands, outside of the graph context. To execute such an operation as a single command there are two additional options that the user must provide: the input dataset and a name for the output. However editor widgets for graph operations do not provide a functionality to ask for these parameters. Consequently, whenever a graph operation is triggered, the `OperationAction` controller is also responsible of adding a combo box and an editable text box to the operation editor, before every other widget. The result can be seen in Fig. 1.15. The combo box allows to choose which dataset to operate on and the text box requires the name of the output dataset, that will be set on the workbench when the operation completes.

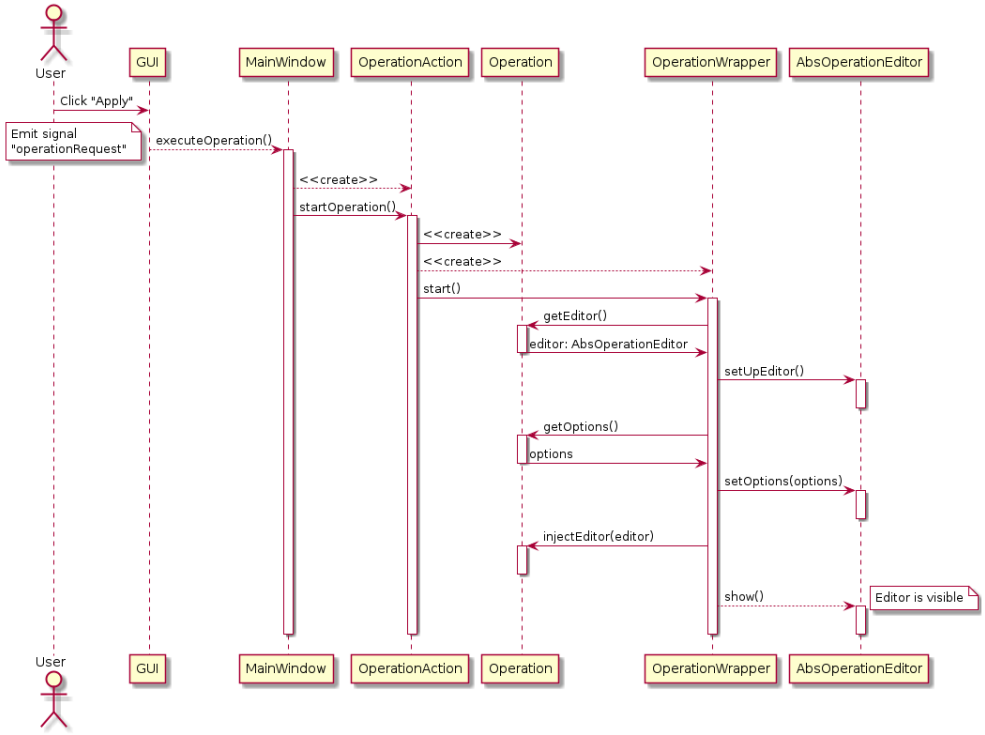

Figure 1.13: Sequence diagram of editor creation and display

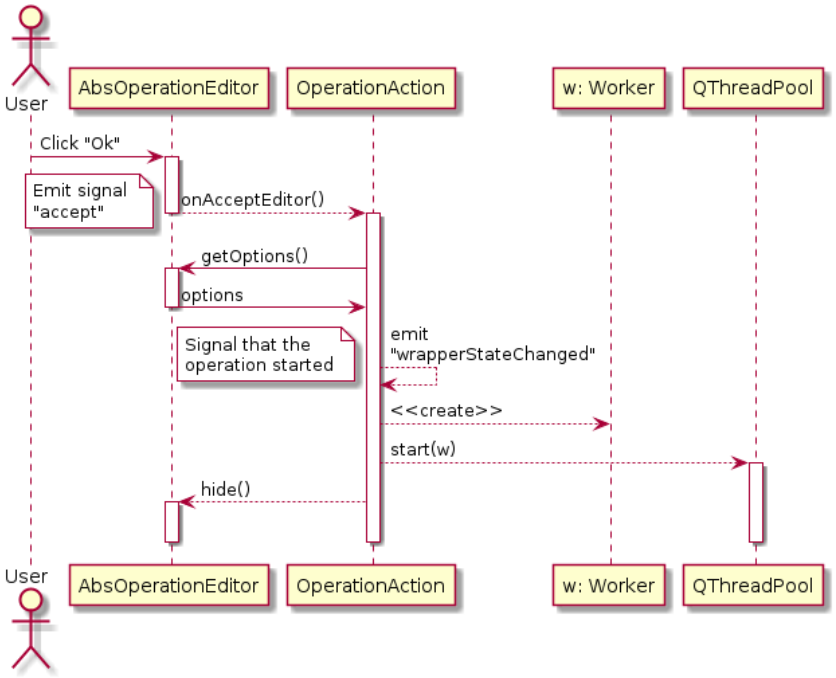

Figure 1.14: Sequence diagram of the option confirmation process

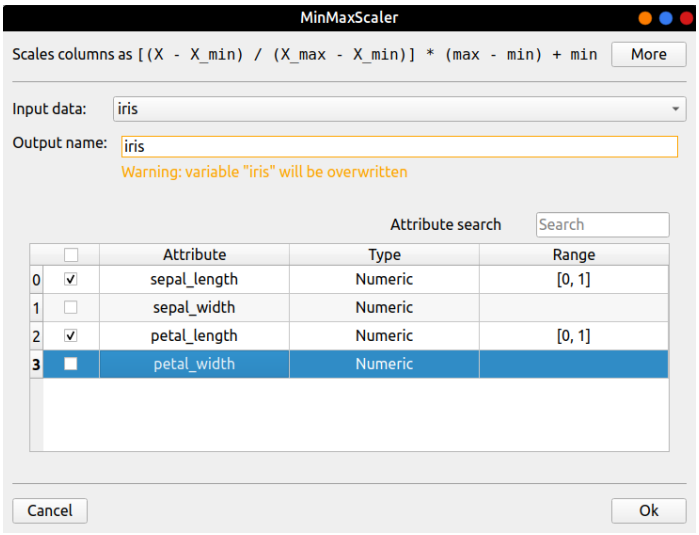

Figure 1.15: The editor widget for min-max scaling. The editor is also warning the user that the output name already exists in the workbench, and thus the output of the operation will overwrite the current dataset.

1.3.6 Charts visualisation

DataMole includes some visualisation features: it can show a frequency histogram to represent value distribution, a scatterplot matrix to visualise bivariate relations and a line chart for temporal series.
Implementing these features required some research about existing plotting libraries that could be embedded in this project, and they are reported in the next section.

1.3.6.1 Technological considerations

All charts in the program are drawn using QtCharts. This is the official module for creating graphs within Qt and is included in the PySide2 Python package, so no additional packages are needed. It builds upon the Graphics View framework and it is quite simple to use. On the other hand, it does not provide many advanced features, and its documentation is a bit lacking, with respect to the rest of the framework.
The following list contains a description of other packages that can be used to plot charts inside a Qt application:

- **PyQtGraph**: a scientific library based on Qt4 and `numpy`. It relies on the Qt GraphicsView framework and is released under MIT license;
- **QCustomPlot**: a Qt-based C++ widget for plotting and data visualisation released under GPL v3 license
- **PyQwt**: another plotting library for Python based on PyQt.

PyQtGraph should be considered for future extension, since it is actively maintained and works with both PyQt and PySide2. It offers many advanced features that would require much work with the official QtCharts. On the other hand, QCustomPlot does not work with PySide2 and PyQwt is not maintained anymore, and only supports older versions of Qt.

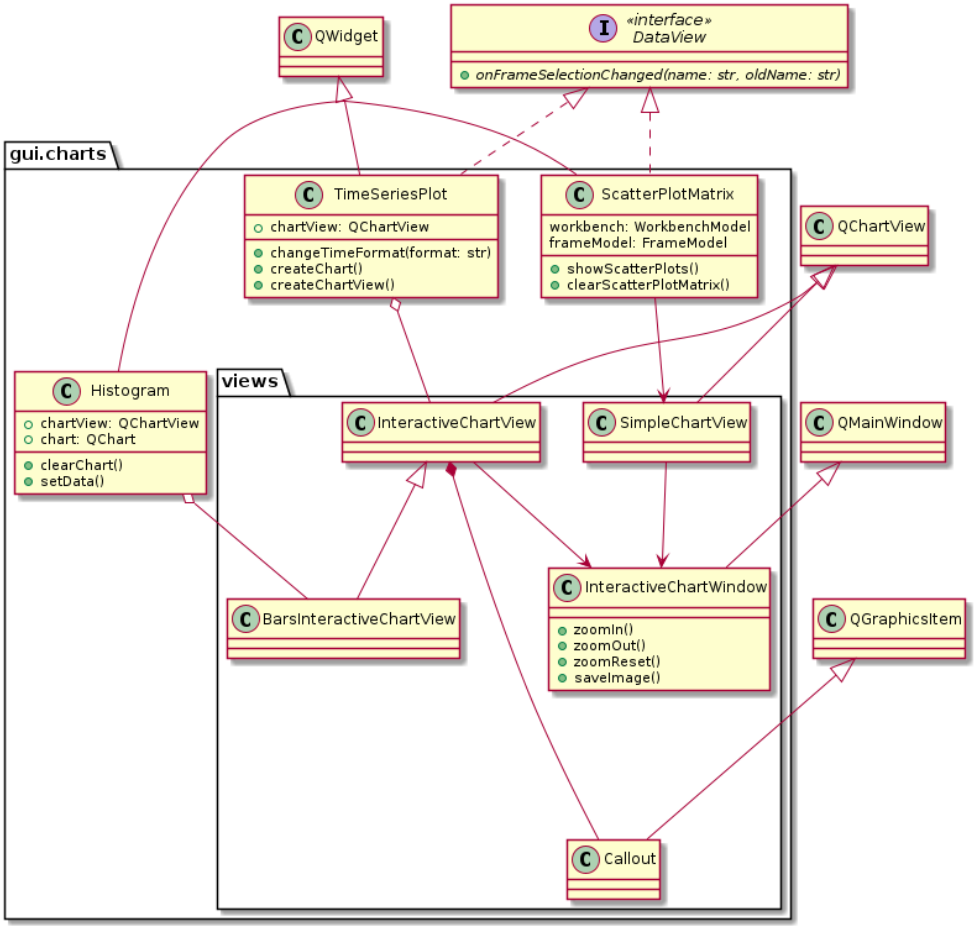

Figure 1.16: Class diagram of the `gui.charts` package

1.3.6.2 The plotting package

Package `gui.charts` defines the widgets for creating and showing plots. Additionally customised chart views are defined inside the `views` module, and some related helper functions are in the `utils` module. Fig. 1.16 shows the package class members.
When a chart view is double clicked the chart is copied on a secondary window, of type `InteractiveChartWindow`. This feature provides a way to move the charts around the screen as independent windows, and adds the ability to save its content as image. This feature relies on the ability to copy the chart data into a new one. Unfortunately this is not possible for the histogram chart, due to a limitation in the QtCharts module, and for this reason the `BarsInteractiveChartView` does not allow to open the chart on a new window. The `InteractiveChartView` also draws a pop-up item showing the data coordinates or label when one of its point is hovered with the cursor. This is implemented in the `Callout` class, which was inspired from the official Qt example available at [5].
`TimeSeriesPlot` and `ScatterPlotMatrix` are the widgets shown in the *View panel*, used to create the line chart for time series and the scatterplot matrix. Every widget

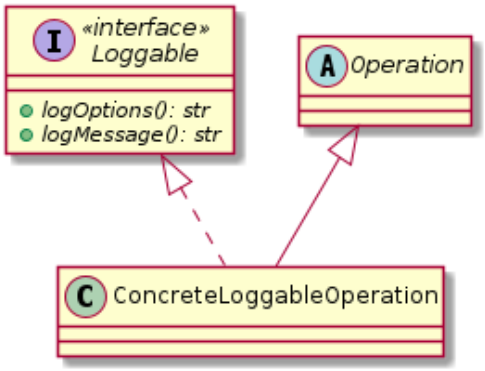

Figure 1.17: Class diagram for a sample operation that can be logged

that adds a visualisation feature to this panel must realise the `DataView` interface. It only requires the definition of one *slot*, which is called whenever the user clicks on a different dataframe in the workbench.

1.3.7 Logging

The `flogging` package provides all the logging functionalities for DataMole and internally uses the Python `logging` module. This package is used to create three different logs:

- **Application log**: contains all the application messages, like warnings, errors and debug messages. These logs are dumped inside the `logs/app` folder and are mainly useful for debugging;
- **Operation log**: logs the operations applied directly from the *Attribute panel*. Every log file is created inside the `logs/operations` folder and contains a summary of every operation run during a program session;
- **Graph log**: logs the execution of a pipeline. A different file is created every time a pipeline is executed inside the `logs/graph` folder.

Additionally the `logging` module also creates a root logger which logs everything passed to any active logger. This logger output is redirected inside the `logs/root` folder.

1.3.7.1 Logging operations

Every operation supports logging. Operations log their options configuration and every parameter set by the user.
Every operation that needs to be logged implements the `Loggable` interface. Diagram in Fig. 1.17 shows its two methods:

- `logOptions`: returns a formatted string with the configuration of the operation (typically user options) or other things that can be logged before the operation is run;
- `logMessage`: returns a string with everything that should be logged after the operation completes, possibly including execution details.

Every time a pipeline is run the `OperationHandler` creates a new log file and logs the operations while they are executed. Operations launched from the *Attribute panel* are instead logged by the `OperationAction` controller. In either case, only operations realising the `Loggable` interface are logged.

1.4 Testing

Every operation defined in DataMole has been tested to ensure that input data are always treated correctly and produce a correct output. The `Pytest` package has been used to create test suites since it is very easy to use and require no boilerplate code. Every test suite is defined in a separate file inside the `tests` folder. Every operation is tested to ensure that every accepted data type with any possible combination of options is handled as expected and that exceptions are thrown when invalid options are set.
Module `mocks` contains the *mock class* definition for the `WorkbenchModel` and the `FrameModel` class. This is necessary because they are both subclasses of the Qt model classes, and thus they cannot be instantiated outside of a Qt Application, which is not initialised for testing purposes.
These unit-tests have been essential to define operations that correctly operate with Pandas and Scikit-learn. In addition, the `OperationDag` methods are tested to ensure that the graph data structure is updated correctly.
GUI components were not tested. To do this it may be worth using `pytest-qt` [3], a Pytest plug-in that allows to simulate user interaction to test widgets.

Chapter 2

Extending DataMole

Part of the development was dedicated to making DataMole easily extensible. Extensibility is meant with respect to the DataMole core features, which are the usage of operations and the visualisation features, like charts. Thus the following topics are discussed in this chapter:

- Definition of *new operations*, to be applied from the *Flow panel* or singularly (§2.2);
- Extension of the *View panel* with new widgets (§2.3).

Additionally the last sections describe how to use the notification system (§2.4), to visualise pop-up with custom messages, the DataMole logger (§2.5) and finally §2.6 explains how non-code files can be added to the resource system.
This chapter only describes classes, methods and everything relevant in order to extend DataMole. A complete description of the program API, with all classes and methods, will be released in the software repository along with the code.

2.1 Package organisation

DataMole is organised as a standard Python package, and its complete structure is shown in the tree below, where only the `tests/` and `docs/` folders have not been expanded.
An overview of the content of the main sub-packages, marked in green in the below tree, was given in §1.3.1.

```
dataMole/
├── main.py
├── requirements.txt
├── makefile
├── docs/ ..... Automatic documentation
│   └── ...
├── tests/
│   └── ...
└── dataMole
    ├── threads.py
    ├── status.py
    ├── resources.qrc
    ├── exceptions.py
    ├── utils.py
    ├── flow/ ..... Pipeline management
    │   ├── dag.py
    │   └── handler.py
    ├── operation/ ..... Every opeation is defined here
    │   ├── actionwrapper.py ..... Handles operation execution
    │   ├── cleaner.py
    │   ├── dateoperations.py
    │   ├── discretize.py
    │   ├── dropcols.py
    │   ├── duplicate.py
    │   ├── extractseries.py
    │   ├── fill.py
    │   ├── index.py
    │   ├── input.py
    │   ├── join.py
    │   ├── onehotencoder.py
    │   ├── output.py
    │   ├── removenan.py
    │   ├── rename.py
    │   ├── replacevalues.py
    │   ├── scaling.py
    │   ├── typeconversions.py
    │   ├── utils.py
    │   ├── readwrite/ ..... I/O operations
    │   │   ├── csv.py
    │   │   └── pickle.py
    │   ├── computations/ ..... Worker operations
    │   │   └── statistics.py
    │   └── interface/
    │       ├── graph.py
    │       └── operation.py
    ├── data/ ..... Dataframe utilities
    │   ├── Shape.py
    │   ├── Frame.py
    │   └── types.py
    └── gui/ ..... Qt GUI classes
```

```
    ├── mainmodels.py
    ├── workbench.py
    ├── window.py
    ├── utils.py
    ├── charts/
    │   ├── views.py
    │   ├── scatterplot.py
    │   ├── timeseriesplot.py
    │   ├── histogram.py
    │   └── utils.py
    ├── panels/
    │   ├── framepanel.py
    │   ├── diffpanel.py
    │   ├── viewpanel.py
    │   ├── attributepanel.py
    │   └── dataview.py
    ├── graph/
    │   ├── node.py
    │   ├── controller.py
    │   ├── constant.py
    │   ├── edge.py
    │   ├── rubberband.py
    │   ├── scene.py
    │   ├── view.py
    │   └── polygons.py
    ├── widgets/
    │   ├── waitingspinnerwidget.py
    │   ├── statusbar.py
    │   ├── operationmenu.py
    │   └── notifications.py
    └── editor/
        ├── configuration.py
        ├── interface.py
        ├── OptionsEditorFactory.py
        └── infoballoon.py
├── flogging/ ..... Logging package
│   ├── loggable.py
│   ├── utils.py
│   └── operationlogger.py
├── resources/ ..... Static resources
│   ├── style.css
│   ├── descriptions.html
│   └── icons/
└── config/ ..... Configuration files for extension
    ├── operations.json
    └── dataviews.json
```

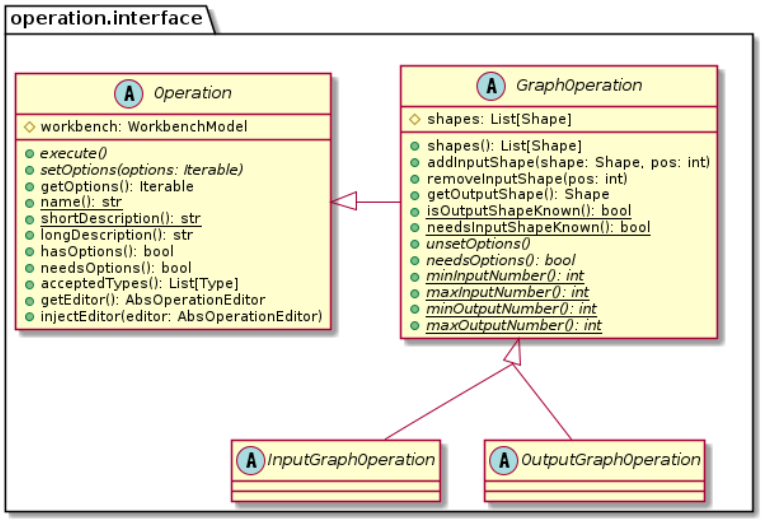

Figure 2.1: Abstract classes derived from `Operation`

2.2 Definition of a new operation

Adding a new operation is simple. Three steps are required:

1. Choose the `Operation` abstract class to subclass;
2. Define the new operation inside a module;
3. Edit a configuration file to tell DataMole where to look for new operations.

When defining a new operation it is probably necessary to define an editor widget to support setting user options. This step is described later, in §2.2.4.

2.2.1 Choosing the abstract class

All data transformations in DataMole are represented with subclasses of the `Operation` base class. Package `operation.interface` contains the definitions of 4 abstract classes shown in Fig. 2.1. The choice of the class to inherit depends on the operation that needs to be defined:

- `Operation`: subclass this to define an operation as a generic task. The only abstract methods are `execute` and `setOptions`. Such an operation can not be used with the pipeline in the *Flow panel*, but can only be applied from the *Attribute panel*. Even background workers can implement this interface to take advantage of the multithreading module, as is later described in §2.2.5;
- `GraphOperation`: its subclasses can be used in the *Flow panel* as well as in the *Attribute panel*, provided it has only one input. This constraint is set to reduce complexity, since graph operations must be adapted to be used from the *Attribute panel* and having to deal with multiple inputs was not easy.
  Graph operations must not do side effects on their inputs (or on the workbench);

- `InputGraphOperation`: this is defined for convenience. It gives a default implementation to some methods that are not relevant when defining an input operation for the pipeline graph. These operations has no input nodes;
- `OutputGraphOperation`: again this is subclassed only once by the graph operation `ToVariable` which has no output and exactly one input. This interface may be used to define an operation with no outgoing edges.

Every operation can hold a reference to the workbench object, but this is optional. In general `GraphOperation` subclasses should avoid any side effect: the pipeline consists of a chain of *functional* transformations where the output of the parent operation is set as the input of the child node, and every operation must not change its input while producing its output. `InputGraphOperation` and `OutputGraphOperation` can instead do side-effect on the workbench: in fact they are mainly used to retrieve the input dataframe from the workbench when the pipeline starts and to save the pipeline output when it completes.

2.2.2 Implementing the operation

New operations can be defined in new modules inside the `operation` package, where all DataMole operations are defined. Additionally related transformations are usually grouped in the same file. For example the module `type.py` contains the definition of all the operations used for type conversion. However it is not important where the new operations are defined, so they may be stored in different packages if desired.
The `operation` package also contains an `utils` module with many helper functions for parsing editor options and some validators used by the editor widgets to validate inputs.

2.2.2.1 *Operation* methods

This section describes the methods that every `Operation` subclass inherits and override when required.

- `execute(*args, **kwargs)`: this method is run to execute the operation. It takes any number of arguments (0 as well) and returns whatever the operation produces. It may also return nothing, if the operation does side effects;
- `setOptions(*args, **kwargs)`: the method used to configure an operation with its options. This method is called with the arguments provided by the `getOptions` method of the editor widget, so its arguments depends on the way the widget provides options. For example if the editor widget returns a tuple of 3 integers, this method should expect to receive 3 integer arguments. If the operation does not require options this method can be set to a no-op. This method can also perform fields validation, see §2.2.2.2;
- `getOptions()`: this method returns the options currently set in the operation, in the same format required by `setOptions`. If you are defining an operation to use from the *Attribute* panel this method is probably never used and its reimplementation can be skipped. The default implementation returns an empty dictionary;
- `name()`: this static method returns a string with the name of the operation to be shown to the user;
- `shortDescription()`: a static method with a reasonably short description to be visualised inside the header of the editor widget;
- `longDescription()`: returns the text to be shown when the user clicks the "More" button in the editor widget. The description can be long and can include HTML formatted text. To avoid cluttering code with long formatted strings, they can be placed in the `resources/descriptions.html` file, under a `section` tag with the operation class name. By default this method searches the description in that file. More details are in §2.6;
- `hasOptions()`: returns a boolean value saying whether all the required options are set. This is required because every operation can have its own option fields. Typically it returns *True* if all option fields are not set to *None*;
- `needsOptions()`: returns a boolean value that tells whether the operation needs options, and consequently an editor widget. If this method returns *False* the `getEditor` method can be no-op;
- `acceptedTypes()`: returns the list of types that the operation supports, to choose between Numeric, Nominal, Ordinal, String and Datetime. These types are defined in the `data.types` module. By default it supports all types;
- `getEditor()`: builds the editor widget of type `AbsOperationEditor` which should be used to configure the operation. This is described in §2.2.4;
- `injectEditor(AbsOperationEditor)`: if some editor components require additional configuration that cannot be provided during the editor creation, this method can be reimplemented. Typically it is used to fix column size on tables created with the editor factory.

2.2.2.2 Options validation

Operations can refuse to accept parameters if they are not set correctly: options validation may be performed inside the `setOptions` method of every operation. This method should first check if the provided options are acceptable and save them only if they are. Otherwise it should raise an exception of type `OptionValidationError`. This exception can be parametrised with a list of errors that occurred while validating the options, thus it allows to notify more than one error at once. Every item of this list is a pair made up of a string error code and an error message. The error code allows to customise error handling and will be discussed in §2.2.4.1.
To give a practical example, part of the definition of the `BinsDiscretizer` class is reported here and commented:

```
class BinsDiscretizer(GraphOperation, Loggable):
def __init__(self, *args, **kwargs):
super().__init__(*args, **kwargs)
# Initialisation of the options fields
# Strategy to use for discretizing
self.__strategy: BinStrategy = BinStrategy.Uniform
# The column indices to be discretized
self.__attributes: Dict[int, int] = dict()
# The suffix for the column to create (if the
    transformation is not done in-place)
self.__attributeSuffix: Optional[str] = '_discretized'

def setOptions(self,
attributes: Dict[int, Dict[str, str]],
strategy: BinStrategy,
suffix: Tuple[bool, Optional[str]]) -> None:
# 'attributes' is a dictionary like {row: {column\_key:
    value} }
# Thus it maps every row to the values of any column in
    the table

# Validate options
errors = list()
if not attributes:
# Error: the user did not select any attribute
errors.append(('e1', 'Error: At least one attribute
    should be selected'))
for r, options in attributes.items():
bins = options.get('bins', None)
if not bins:
# Error: column 'bins' is not set for this row
errors.append(
('e2', 'Error: Number of bins must be set at row {:d}'.
    format(r))
)
elif not isPositiveInteger(bins):
# Error: column 'bins' is not a valid number
```

```
        errors.append(('e3', 'Error: Number of bins must be > 1
            at row {:d}'.format(r)))
        if strategy is None:
        # Error: no strategy is selected from the radio buttons
36      errors.append(('e4', 'Error: Strategy must be set'))
        if suffix[0] and not suffix[1]:
        # Error: suffix is not set
        errors.append(('e5', 'Error: suffix for new attribute
            must be specified'))
        if errors:
41      # If any validation error occurred stop
        raise OptionValidationError(errors)

        # No error occurred, then set options
        # Clear previously set attributes
46      self.__attributes = dict()
        # Set options
        for r, options in attributes.items():
        k = int(options['bins'])
        self.__attributes[r] = k
51      self.__strategy = strategy
        self.__attributeSuffix = suffix[1] if suffix[0] else None
```

In the above example the `setOptions` method checks for options correctness in lines 20-39. Then if one or more validation errors were detected it raises an exception (line 42), otherwise it sets the new options (lines 44-52). The error message will be shown to the user as in Fig. 2.2.

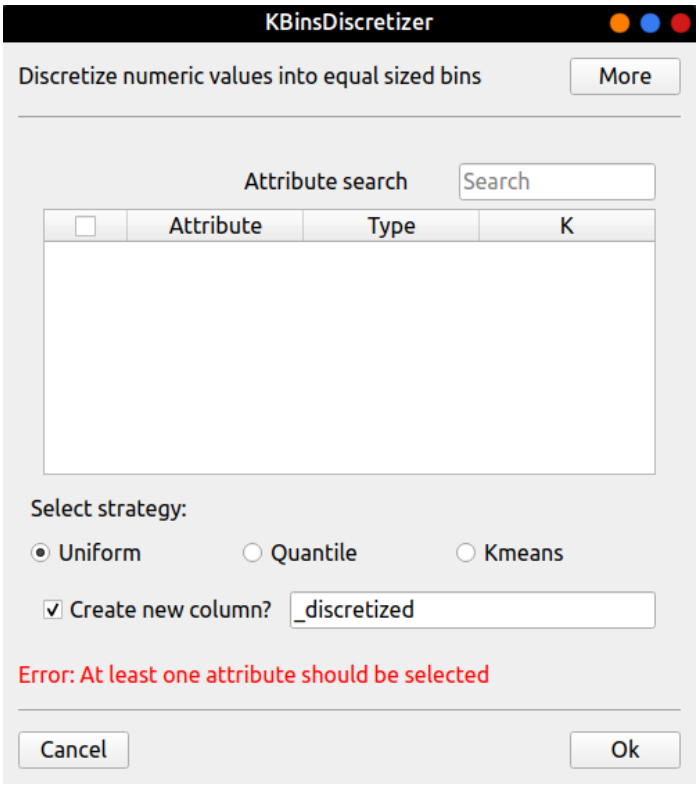

Figure 2.2: Error message in the *BinsDiscretizer* operation

2.2.2.3 *GraphOperation* methods

The following section describes methods defined in the `GraphOperation` class. All methods inherited from `Operation` are not discussed here, unless they require further explanation.

2.2.2.3.1 Utility methods

The following three methods provide functionalities required to manage every operation inside the pipeline. They are already implemented and should not be redefined in most cases.

- `shapes()`: getter method for the protected `shapes` field, which contains the shapes of the inputs of the operation;
- `addInputShape(Shape, int)`: this method is used to set the shape at the specified position in the `shapes` list;
- `removeInputShape(int)`: remove the `Shape` object at the specified position in the `shapes` list and sets it to `None`.

2.2.2.3.2 Methods to be reimplemented

The following methods should be overridden in every subclass to customise the operation behaviour.

- `execute(*Frame)`: the execute method has a different signature from the one defined in the `Operation` class. As already explained graph operations should be defined in a functional way, with no side effects. Hence this method is passed the input dataframes and must apply the transformation and return the output dataframe (of type `Frame`) without affecting its input;
- `getOutputShape()`: this method should be overridden to return a `Shape` object with the column names and types of the output of this operation. If the input shapes are not set (for example because the node is not yet connected to a predecessor) or any relevant option is not configured it must return `None`;
- `isOutputShapeKnown()`: this static method must return `True` if `getOutputShape` is able to infer the output shape, given the operation options and the shapes of its input. Otherwise it must return `False`. There are situations in which this method could *always* return `False`. For instance, this is the case with the operation to remove all columns with more than a threshold of NaN values: there is no way of knowing in advance which columns will satisfy this condition, hence the operation does not know its output shape;
- `needsInputShapeKnown()`: another static method that returns `True` for operations that require their input shapes to be used. Operations that do not need the input shape are the only operations that can be placed after operations that do not know their output shape (i.e. their `isOutputShapeKnown` method always returns `False`);
- `unsetOptions()`: this method resets every option that depends on the input shapes of the operation. It is called by the `DagHandler` whenever new input shapes are propagated through the pipeline;
- `minInputNumber()`: a static method which returns the minimum number of input connections that the operation supports;
- `maxInputNumber()`: a static method returning the maximum number of input connections that the operation supports or -1 if there is no maximum;
- `minOutputNumber()`: a static method returning the minimum number of output connections that the operation supports;
- `maxOutputNumber()`: a static method returning the maximum number of output connections that the operation supports or -1 if there is no maximum;

2.2.3 Export the operation

Once operations have been defined it is necessary to make them visible to DataMole. Every module must define a global variable `export` pointing to the new operation class or, if more than one operations are defined in the same module, to a list (or tuple) of classes. For instance if two new operations were defined in the same module with classes `TransformData1` and `TransformData2`, the export variable should be set as in this snippet:

```
class TransformData1(Operation, Loggable):
def execute(self, *args, **kwargs):
pass
...

class TransformData2(Operation, Loggable):
def execute(self, *args, **kwargs):
pass
...

class OtherStuff:
# Class that is not an operation
...

export = TransformData1, TransformData2
# or export = [TransformData1, TransformData2]
```

Additionally the name of every module to be searched for operations must be appended to the list defined in file `config/operations.json`. The fully qualified name of the module should be used, like in the following example:

```
{
  "modules": [
  "dataMole.operation.fill",
  "dataMole.operation.discretize",
  ...
  "dataMole.operation.myNewModule"
  ]
}
```

2.2.4 Definition of editor widgets

Every operation requiring user options is responsible for defining its specialised *editor widget* to support options configuration. An example is in Fig. 2.3.
Every editor widget must subclass the `AbsOperationEditor` abstract class, defined in the `gui.editor` package. Package structure is shown in Fig. 2.4.
Custom editors are usually defined in the file containing the operation definition, since they are used only by a single operation. However, they can be placed on a different file if they are reused many times.
An *editor widget* should reimplement the following methods:

- `editorBody()`: this abstract method should return the `QWidget` to place in the editor, between the header and the footer;
- `getOptions()`: this abstract method must return the options set in the editor. It should return them inside an iterable sequence that can be unpacked when passed to the `setOptions` method of the operation. Its return type should therefore be compatible with the argument type of the operation `setOptions` method;
- `setOptions(*args, **kwargs)`: sets the options inside the editor. The default implementation does nothing, which is ok in case the operation does not require a configurable editor;

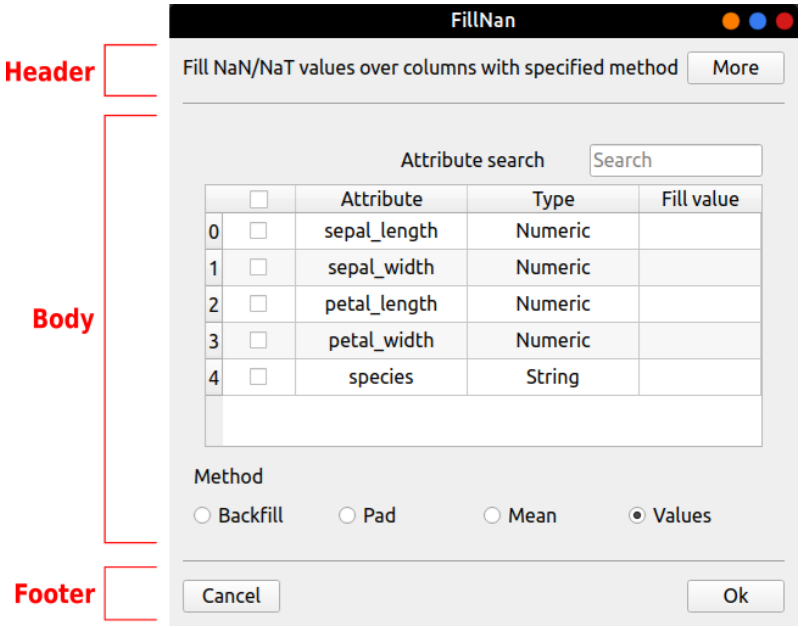

Figure 2.3: The three parts that compose every *editor widget*. The *header* shows the short description and button to access the long description of the operation. The *footer* has a button to quit the editor and one to confirm the options set.

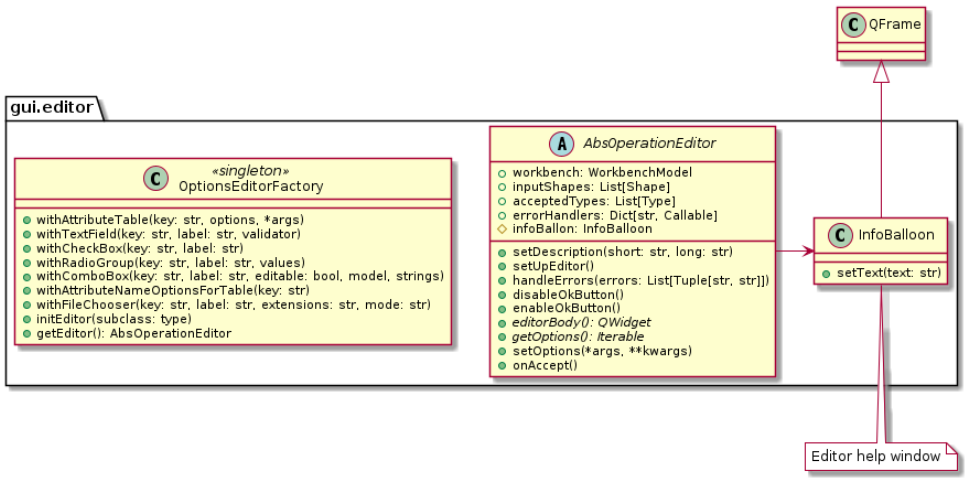

Figure 2.4: Classes defined in the `gui.editor` package

- `onAccept()`: this hook method is called after the options are confirmed (i.e. the *"Ok"* button is clicked) and before setting them in the operation. It can be reimplemented to perform additional actions. Does nothing by default.

2.2.4.1 Customised validation error handling

As previously described, when the user configures an editor widget, the operation can raise an error with type `OptionValidationError` if supplied options are not correct. In this situation, every editor widget shows the error messages in red font immediately above the footer, like in Fig. 2.2. This is generally good enough, but complex editors may behave differently, for example by applying a red border around a `QLineEdit` or by showing pop-up immediately above the wrong fields. In fact, editor widgets derived from `AbsOperationEditor` support custom error handling behaviours. This abstract class defines a `errorHandlers` public field, of type `Dict[str, Callable]`, that contains arbitrary functions (`Callable`s) used to handle specific validation errors. Recall from §2.2.2.2 that every pair passed to the `OptionValidationError` constructor contains an *error code*, as well as an error message. When handling validation error, the editor first checks if a custom error handler is provided: to do this it searches the `errorHandlers` dictionary for a pair with the specified error code as key. If a matching key is found, the corresponding method is invoked, otherwise the default error handling strategy is used.
Hence, once a customised method to handle errors has been defined, it may be added to this dictionary with an error code that matches the one of the error it should handle.

2.2.4.2 The editor factory

Since many operation editors required very similar components a *factory class* has been defined to quickly build standard editors with a variety of fields. To do this, the factory class `OperationEditorFactory` can be instantiated inside method `Operation.getEditor` and used to configure the editor. Since the factory is a *singleton*, method `initEditor` must be always called to initialise a new editor.
When the factory is used, widgets options are passed around between the editor and the operations as Python dictionaries. This is the reason why methods `getOptions` and `setOptions` of classes `AbsOperationEditor` and `Operation` also accept key-value pairs with the `**kwargs` argument. The key used for every option can be specified with the `key` argument when using the factory methods described below.

- `withAttributeTable(key, options, checkbox, nameEditable, showTypes, types, *args)`: adds a table to show dataframe columns, with optional column of checkboxes for selection and any number of additional columns. For example the widget in Fig. 2.3 had a table with 1 additional column, named *Fill value*, to specify the value that should be used to substitute NaNs. The `options` argument is a dictionary to specify the additional columns to show in the table: every entry consists of a column identifier and a tuple with the column name, a delegate for that column and a default value to show when nothing is set. `checkbox`, `nameEditable`, `showTypes` are boolean parameters to control whether to show a checkbox column, whether the name column should be editable and the type column should be showed. Argument `types` allows to filter only certain types, in case the operation does not support every type. For example the *Fill NaN* editor table is configured with this method call:

```
factory.withAttributeTable(
```

```
    key='selected',
    checkbox=True,
    nameEditable=False,
    showTypes=True,
    types=self.acceptedTypes(),
    options={
      'fill': (
      'Fill value',
      OptionValidatorDelegate(SingleStringValidator()),
      None
      )
    })
```

The *delegate* is Qt component that controls how column items are rendered inside the view. By defining a custom delegate, it is possible to change items appearance in any way: the checkbox in the first column of Fig. 2.3, for instance, is created by defining a custom delegate for boolean values. The `OptionValidatorDelegate` was defined for convenience and only provides a customised validation the input, through the `QValidator` passed as its argument. If no delegate and no default value is needed, `None` can be used in their place. Definition of custom delegates will not be discussed here, since it is part of the Qt Framework and is explained in its official documentation;

- `withTextField(key, label, validator)`: adds a `QLineEdit` setting a label above it and with an optional `QValidator` for its input. Some validators commonly used are defined in the `operation.utils` module. Widget marked with *(T)* in Fig. 2.5 was created using this method;

- `withCheckBox(key, label)`: adds a `QCheckBox` with a label;

- `withRadioGroup(key, label, values)` inserts a group of `QRadioButtons` with the specified label above it. the `values` argument is a list of pairs, that maps the label to show (as a string) with the combo box value (of any type). The radio buttons in Fig. 2.3 is created with the following options:

```
    factory.withRadioGroup(
    key='fillMode',
    label='Method',
    values=[
    ('Backfill', 'bfill'),
    ('Pad', 'ffill'),
    ('Mean', 'mean'),
    ('Values', 'value')
    ])
```

- `withComboBox(key, label, editable, model, strings)`: adds a `QComboBox` with a label. If `editable` is *True* the combo box allows to enter arbitrary values, otherwise it only allows to choose between one of the predefined values. Arguments `model` and `strings` allow to set the values to show when the combo box is used. A Qt model class can be used or alternatively a list of strings can be provided;

- `withAttributeNameOptionsForTable(key)`: using this method allows to add a widget like Fig. 2.6. It adds an option to avoid overwriting attributes when transformations are applied by defining new ones with the specified suffix;
- `withFileChooser(key, label, extensions, mode, **kwargs)`: used to create a window that allow to choose an existing file using a `QFileDialog`. Showed files can be filtered by their extension using the `extensions` argument. The `mode` string argument must be set to `"save"` or `"load"`, depending on whether the file dialog should allow to select non existing files (*save* mode) or not (*load* mode). An example of such widget is shown in Fig. 2.5, marked with *(F)*. Additional arguments can be passed to the `QFileDialog` by setting them as `**kwargs`;
- `initEditor(subclass)`: this method must be called before any factory method, and is used to initialise a new editor widget. The factory is a singleton, thus a call to this method resets its internal state and clean the parameters of widgets previously created. The `subclass` parameter accepts a `type` of a class that should be used as base class for the new widget. Of course it must be a subclass of `AbsOperationEditor`;
- `getEditor()`: assembles the new editor widget and returns it.

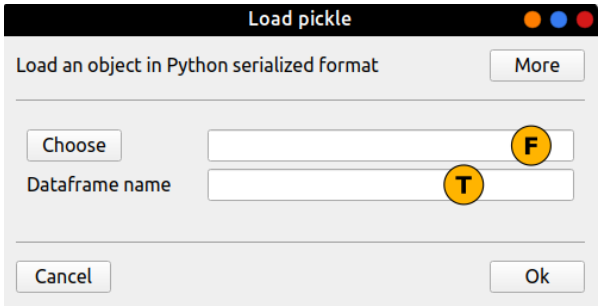

Figure 2.5: Widget to import *pickle* dataframes created with factory methods

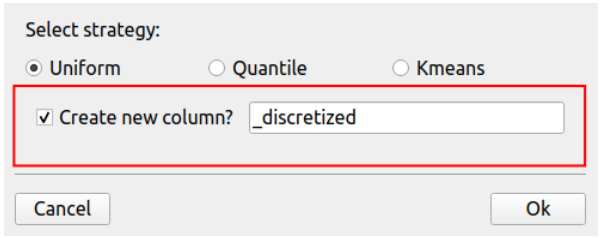

Figure 2.6: Widget created with factory method `withAttributeNameOptionsForTable`

Using the factory, a typical `getEditor` implementation for an operation may look like this:

```
1   def getEditor(self):
    factory = OptionsEditorFactory()
```

```
    factory.initEditor()
    ...
    # Call factory methods
    ...
    return factory.getEditor()
```

Finally it must be said that editor widgets defined using the factory do not support the customised error handling mechanism described in §2.2.4.1.

2.2.5 Creating worker operations

Sometimes there is the necessity to do some background computation without freezing the user interface. Operations can also be defined for completing tasks that can be computed in background. For example the `operation.computations.statistics` module contains two operations used respectively to compute data for the *statistics panel* and the *histogram* when an attribute is clicked in the *Attribute* tab.
The `threads` module can then be exploited to run the operation in another thread. This module defines a `Worker` class (deriving from `QRunnable`) that can be scheduled for execution in a `QThreadPool`.
After defining a worker operation by implementing the `Operation` interface, a background computation can be set up using this general pattern:

```
# Define a worker operation
class BackgroundComputation(Operation):
def __init__(*args):
# Initialise
...

def execute(arg1, arg2, *args):
...
# Computation
...
return result

# Initialise a new operation
comp = BackgroundComputation(*my_args)
# Set eventual options and define arguments for the
    execute() method
myArg1 = ...
myArg2 = ...
# Create a worker for the operation and set the execute()
     args
worker = Worker(comp, args=(myArg1, myArg2), identifier='
    comp1')
# Connect worker signals to appropriate handler slots
worker.signals.result.connect(self.onResult)
worker.signals.error.connect(self.onError)
worker.signals.finished.connect(self.onFinish)
# Start computation on the thread pool
QThreadPool.globalInstance().start(statWorker)
```

The `worker` needs an identifier that will be passed back as the first parameter of the emitted signals. This identifier can be of any type, and should be used to recognise which worker emitted a particular signal, but can be omitted if it is not relevant. The `args` parameter can be omitted as well if the operation `execute` method does not require arguments. In the above example `onResult`, `onError` and `onFinish` are methods marked as Qt *slots* (which allow to connect them to signals) and are invoked when the worker status changes, through the following signals:

- `result(id: object, result: object)`: this signal is emitted when the worker completes successfully (i.e. without runtime errors) and carries two arguments,

the identifier passed to the worker constructor and the value returned by the `execute` method, which can be `None` if the operation does side effects with the result;

- `error(id: object, err: tuple)`: it is emitted when the `execute` method fails with a runtime error. The second parameter is a tuple with the type of the exception, the exception object itself and the stack-trace as a string. This data can be used to create a log entry and to notify the user;
- `finished(id: object)`: signal emitted after the worker stops executing, either because it failed (and the `error` signal was emitted) or because it completed successfully (and the `result` signal was fired).

2.3 Extension of the *View panel*

Currently the *View panel* supports the creation of two type of charts, the line chart for time series and the scatterplot matrix, but it is possible to add customised visualisation features to this panel. The active widget can be switched by using the combo box shown in Fig. 2.7. These widgets are dynamically discovered and loaded every time DataMole is started by looking at the configuration in file `config/dataviews.json`. This operation is done in the `__init__` file of the `gui.panels` package. The *json* file contains the following lines:

```
1  {
2    "config": {
3      "default": "Scatterplot",
4      "description": "Select a data view:"
5    },
6    "classes": {
7      "Scatterplot": "dataMole.gui.charts.scatterplot.
         ScatterPlotMatrix",
8      "Time series": "dataMole.gui.charts.timeseriesplot.
         TimeSeriesPlot"
9    }
10 }
```

The `classes` dictionary contains the fully qualified name of the widget class to show in the panel, with the label to show in the combo box as keys. The `config` dictionary contains the label to set as default one in the combo box, and the label to place before the combo box.
Hence, in order to add widgets to this panel, one should:

1. Define the new widget implementing the `DataView` interface; it requires the definition of a single *slot*, namely `onFrameSelectionChanged`,in order to react properly when the user changes the active dataframe;
2. Add the new class name to the `classes` dictionary in the `config/dataviews.json` file.

2.4 Using the notification system

Small pop-ups are used to notify the user whenever an error occurs or some operation is completed. An example is shown in Fig. 2.9. Notification messages are stacked vertically and can be closed by clicking on the small right button.
These pop-ups are defined in the `gui.widgets.notifications` module, which contains 3 classes, outlined in diagram Fig. 2.8. An instance of the `Notifier` class is always available in the *gui.notifier* global variable. Thus in order to add notifications, this global variable should be imported from the `gui` package. The `addMessage` method can be used to add messages, or they can be cleared invoking `clearMessages`. The `gui` package also exposes the `gui.statusBar` variable, which is the global access point to the DataMole status bar, placed at the bottom of the main window, shown in Fig. 2.9. It inherits `QStatusBar`, so its methods can be used to show messages.

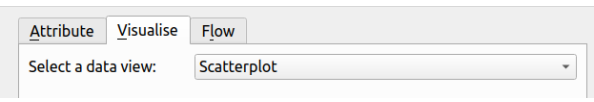

Figure 2.7: Combo box used to switch active widget in the *View panel*

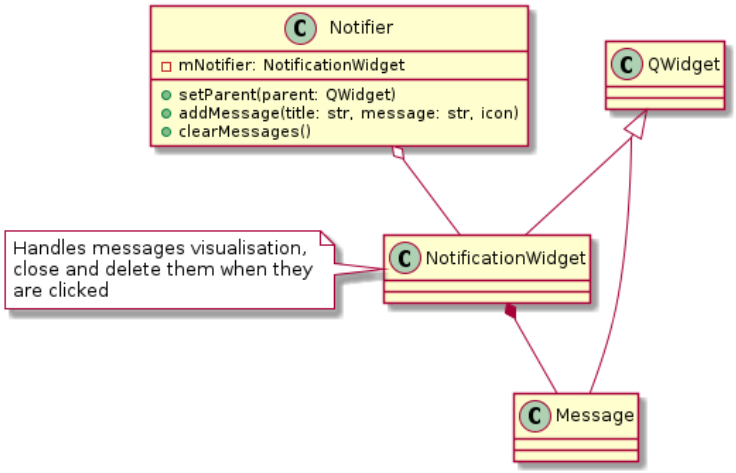

Figure 2.8: Class diagram of the `notifications` module

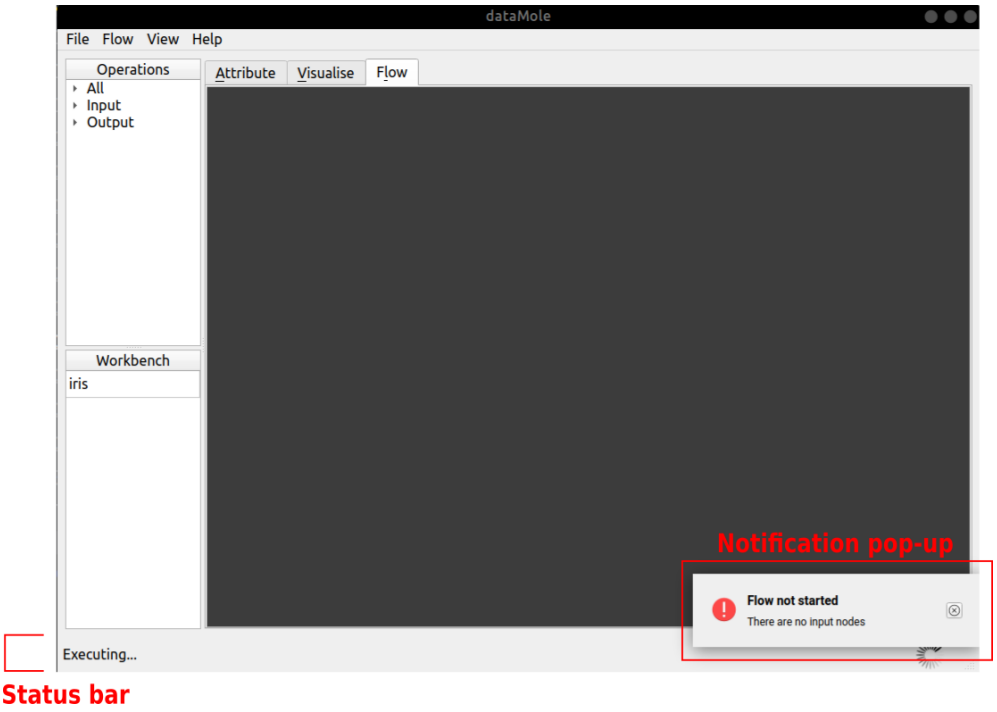

Figure 2.9: The main window, with a message on the status bar and a pop-up used for notifications. In this example the user tried to execute a pipeline with no input nodes.

2.5 The logging package

DataMole defines a collections of logging utilities in the `flogging` package, which makes used of the Python `logging` module. Three loggers are defined:

- `appLogger`: the *application logger*, where debug messages are printed, and the standard error stream (`stderr`) is redirected. These log files are created in the `logs/app` folder;
- `graphLogger`: when a pipeline is executed the *DagHandler* creates a new log file in the `logs/graph` folder and uses this logger to log every operation. In order to be logged operations need to have implemented the `Loggable` interface, defined in the `flogging.loggable` module;
- `opsLogger`: every operation applied singularly (i.e. outside of the pipeline) is logged through this object inside the `logs/operations` folder. Also in this case the operations need to implement the two methods of the `Loggable` interface.

Loggers are objects of type `logging.Logger` and can be imported directly from the `flogging` package. A simple example is the following:

```
from dataMole import flogging

# Log to the application logger
flogging.appLogger.warning('A warning')

# Log to the operation logger
flogging.opsLogger.error('An operation failed')
```

2.5.1 Implementing the *Loggable* interface

The two methods defined in `Loggable` interface, that must be implemented in order to log an operation, are the following:

- `logOptions`: must return a formatted string with the configuration of the operation, like its options. This method is invoked before the operation is executed;
- `logMessage`: returns a formatted string with additional info. Differently from the previous method, this one is invoked after the operation completes, but it is not called if the operation fails with an error. It may include details on what happened during execution. For instance, in the `BinsDiscretizer` class it is used to inform the user of which intervals were used for discretization.

2.6 The resource system

The Qt resource system is a platform-independent mechanism for storing binary files in the application's executable. This is useful for applications that need to access a certain set of files like icons, translation files, etc. [9]. The `resource` directory groups every such file used in DataMole. It has the following structure:

```
dataMole/
```

```
├── resources.qrc ..... Qt resource file
└── resources/
    ├── icons/ ..... PNG icons
    ├── descriptions.html ..... Operations descriptions
    └── style.css ..... Stylesheet
```

2.6.1 The operation description file

File `descriptions.html` contains the formatted text that is returned by method `longDescription` of an operation. Since these descriptions can be quite long, and need to be formatted properly using HTML syntax, I decided to put them all in single file, instead of having them scattered inside the operation classes. It is possible to add new descriptions or edit the existing ones directly in this file. Every new description must be placed in a new section, with a `name` attribute set to the class name of the related operation, like in this example:

```
<section name="MyOperationClass">
<h2>Operation name</h2>
Very short description <br/>
<h3>Options description</h3>
...
Explain how the operation can be configured
...
</section>
```

There are no rules on how to write the long description, but it should include everything that is needed to understand the purpose of the operation and how it should be configured, if some options are required.
The description file is read during initialisation of the `operation` package, inside the `__init__` file.

2.6.2 Adding new resources

Adding a new resource can be done by adding its path to the `resources.qrc` file, as explained in the official Qt documentation. After that, the resources must be converted to bytecode by using the Qt Resource Compiler, which, in Qt for Python, can be invoked with the `pyside2-rcc` command.
For convenience DataMole comes with a makefile that include the command needed to do this: it is sufficient to run `"make resources"` from the main folder. This command will generate a file named `qt_resources.py` with the bytecode for every resource.

Bibliography

[1] *dsideb/nodegraph-pyqt*. URL: https://github.com/dsideb/nodegraph-pyqt (visited on 09/09/2020) (cit. on pp. 7, 13).

[2] *Modules - Python 3.8.5 documentation*. URL: https://docs.python.org/3/tutorial/modules.html (visited on 09/09/2020) (cit. on p. 6).

[3] *pytest-qt*. URL: https://pypi.org/project/pytest-qt (visited on 09/09/2020) (cit. on p. 25).

[4] *Qt Concurrent*. URL: https://doc.qt.io/qt-5/qtconcurrent-index.html (visited on 09/09/2020) (cit. on p. 20).

[5] *Qt Documentation: Callout Example*. URL: https://doc.qt.io/qt-5/qtcharts-callout-example.html (visited on 09/09/2020) (cit. on p. 23).

[6] *Qt Documentation: Graphics View Framework*. URL: https://doc.qt.io/qt-5/graphicsview.html (visited on 09/09/2020) (cit. on p. 13).

[7] *Qt Documentation: Model/View Programming*. URL: https://doc.qt.io/qt-5/model-view-programming.html (visited on 09/09/2020) (cit. on p. 6).

[8] *Qt Documentation: Signals & Slots*. URL: https://doc.qt.io/qt-5/signalsandslots.html (visited on 09/09/2020) (cit. on p. 5).

[9] *Qt Documentation: The Qt Resource System*. URL: https://doc.qt.io/qt-5/resources.html (visited on 09/09/2020) (cit. on p. 46).

[10] *QThreadPool and QRunnable: Reusing Threads*. URL: https://doc.qt.io/qt-5/threads-technologies.html#qthreadpool-and-qrunnable-reusing-threads (visited on 09/09/2020) (cit. on p. 19).

[11] *Threading Basics: GUI Thread and Worker Thread*. URL: https://doc.qt.io/qt-5/thread-basics.html#gui-thread-and-worker-thread (visited on 09/09/2020) (cit. on p. 19).