QDE — A visual animation system.

MTE-7103: Master-Thesis

Sven Osterwalder*

February 20, 2017

*sven.osterwalder@students.bfh.ch

Contents

1 TODO Introduction

[Introduction here].

2 TODO Administrative aspects

Some administrative aspects of this thesis are covered, while they are not required for the understanding of the result.

The whole documentation uses the male form, whereby both genera are equally meant.

2.1 Involved persons

Author	Sven Osterwalder[1]	
Supervisor	Prof. Claude Fuhrer[2]	*Supervises the student doing the thesis*
Expert	Dr. Eric Dubuis[3]	*Provides expertise concerning the thesis' subject, monitors and grades the thesis*

2.2 Structure of the documentation

This thesis is structured as follows:

- Introduction
- Objectives and limitations
- Procedure
- Implementation
- Conclusion

2.3 Deliverable results

- Report
- Implementation

[1] sven.osterwalder@students.bfh.ch
[2] claude.fuhrer@bfh.ch
[3] eric.dubuis@comet.ch

3 TODO Scope

3.1 Motivation

[Motivation.]

3.2 Objectives and limitations

[Objectives and limitations.]

3.3 Preliminary activities

[Preliminary activities.]

3.4 New learning contents

[New learning contents.]

4 TODO Procedure

4.1 Organization of work

4.1.1 Meetings

Various meetings with the supervising professor, Mr. Claude Fuhrer, helped reaching the defined goals and preventing erroneous directions of the thesis. The supervisor supported the author of this thesis by providing suggestions throughout the held meetings. The minutes of the meetings may be found under .

4.1.2 Phases of the project and milestones

Phase	Description	Week / 2017
Start of the project		8
Definition of objectives and limitations		8-9
Documentation and development		8-30
Corrections		30-31
Preparation of the thesis' defense		31-32

Milestone	Description	End of week / 2017
Project structure is set up		8
Mandatory project goals are reached		30
Hand-in of the thesis		31
Defense of the thesis		32

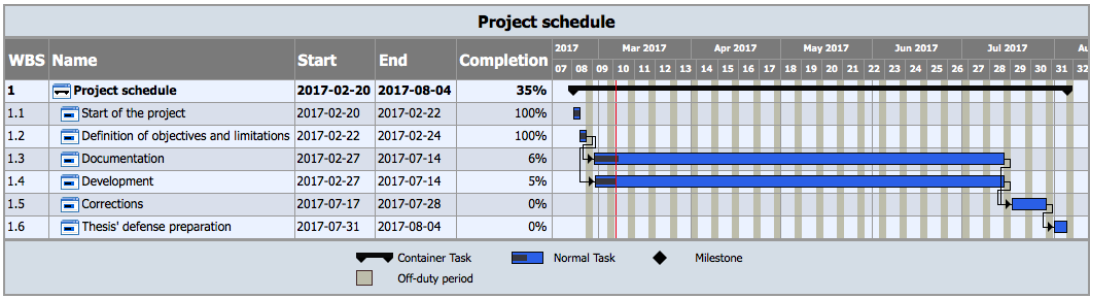

Figure 4.1: The project's schedule.

Figure 4.1 shows the project's schedule.

4.1.3 Literate programming

This thesis' implementation is done by a procedure named "literate programming", invented by Donald Knuth. What this means, is that the documentation as well as the code for the resulting program reside in the same file. The documentation is then *weaved* into a separate document, which may be any by the editor support format. The code of the program is *tangled* into a run-able computer program.

TODO Provide more information about literate programming.

Citations, explain fragments, explain referencing fragments, code structure does not have to be "normal"

Originally it was planned to develop this thesis' application test driven, providing (unit-) test-cases first and implementing the functionality afterwards. Initial trails showed quickly that this method, in company with literate programming, would exaggerate the effort needed. Therefore conventional testing is used. Test are developed after implementing functionality and run separately. A coverage as high as possible is intended. Test cases are *tangled* too, and may be found in the appendix.

TODO Insert reference/link to test cases here.

4.2 Standards and principles

4.2.1 Code

TODO Principles

- Classes use camel case.
- Folders / name-spaces use only small letters.
- Methods are all small caps and use underscores as spaces.
- Signals: do_something
- Slots: on_something
- Importing: `from Foo import Bar`
  As the naming of the PyQt5 modules prefixes them by *Qt*, it is very unlikely to have naming conflicts with other modules. Therefore the import format `from PyQt5 import [QtModuleName]` is used. This still provides a (relatively) unique naming most probably without any conflicts but reduces the effort when writing a bit. The import of system modules is therefore as follows.

Layering

Concerning the architecture, a layered architecture is foreseen, as stated in [1, p. 38 ff.]. A relaxed layered architecture leads to low coupling, reduces dependencies and enhances cohesion as well as clarity.

As the architecture's core components are all graphical, a graphical user interface for those components is developed. As the their data shall be exportable, it would be relatively tedious if the graphical user interface would hold and control that data. Instead models and model-view separation are used. Additionally controllers are introduced which act as workflow objects of the `application` layer and interfere between the model and its view.

1. Model-View-Controller

   While models may be instantiated anywhere directly, this would although not contribute to having clean code and sane data structures. Instead controllers, lying within the `application` layer, will manage instances of models. The instantiating may either be induced by the graphical user interface or by the player when loading and playing exported animations.

   A view may never contain model-data (coming from the `domain` layer) directly, instead view models are used [2].

   The behavior described above corresponds to the well-known model-view-controller pattern expanded by view models.

As Qt is used as the core for the editor, it may be quite obvious to use Qt's model/view programming practices, as described by [1]. However, Qt combines the controller and the view, meaning the view acts also as a controller while still separating the storage of data. The editor application does not actually store data (in a conventional way, e.g. using a database) but solely exports it. Due to this circumstance the model-view-controller pattern is explicitly used, as also stated in [1, p. 38].

TODO Describe the exact process of communication between

ViewModel, Controller and Model.

To avoid coupling and therefore dependencies, signals and slots[2] are used in terms of the observer pattern to allow inter-object and inter-layer communication.

Framework for implementation

To stay consistent when implementing classes, it make sense to define a rough framework for implementation, which is as follows:

1. Define necessary signals.
2. Within the constructor,
   - Set up the user interface when it is a class concerning the graphical user interface.
   - Set up class-specific aspects, such as the name, the tile or an icon.
   - Set up other components, used by that class.
   - Initialize the connections, meaning hooking up the defined signals with corresponding methods.
3. Implement the remaining functionality in terms of methods and slots.

4.2.2 Diagrams

[Diagrams.]

4.2.3 Project structure

[Project structure.]

[1] `http://doc.qt.io/qt-5/model-view-programming.html`
[2] `http://doc.qt.io/qt-5/signalsandslots.html`

5 TODO Implementation

5.1 Requirements

5.1.1 Requirements

This chapter describes the requirements to extract the source code out of this documentation using *tangling*.

At the current point of time, the requirements are the following:

- A Unix derivative as operating system (Linux, macOS).
- Python version 3.5.x or up[1].
- Pyenv[2].
- Pyenv-virtualenv[3].

The first step is to install a matching version of python for the usage within the virtual environment. The available Python versions may be listed as follows.

```
1 pyenv install --list
```

Listing 1: Listing all available versions of Python for use in Pyenv.

The desired version may be installed as follows. This example shows the installation of version 3.6.0.

```
1 install 3.6.0
```

Listing 2: Installation of Python version 3.6.0 for the usage with Pyenv.

It is highly recommended to create and use a project-specific virtual Python environment. All packages, that are required for this project are installed within this virtual environment protecting the operating systems' Python packages. First the desired version of Python has to be specified, then the desired name of the virtual environment.

```
1 pyenv virtualenv 3.6.0 qde
```

Listing 3: Creation of the virtual environment `qde` for Python using version 3.6.0 of Python.

All required dependencies for the project may now safely be installed. Those are listed in the file `python_requirements.txt` and are installed using `pip`.

```
1 pip install -r python_requirements.txt
```

Listing 4: Installation of the projects' required dependencies.

All requirements and dependencies are now met and the actual implementation of the project may begin now.

[1] `https://www.python.org`
[2] `https://github.com/yyuu/pyenv`
[3] `https://github.com/yyuu/pyenv-virtualenv`

5.2 Name-spaces and project structure

This chapter describes the planned directory structure as well as how the usage of name-spaces is intended.

The whole source code shall be placed in the `src` directory underneath the main directory. The creation of the single directories is not explicitly shown respectively done, instead the `:mkdirp` option provided by the source code block structure is used[4]. The option has the same effect as would have `mkdir -p [directory/subdirectory]`: It creates all needed (sub-) directories, even when tangling a file. This prevents the tedious and non-interesting creation of directories within this document.

When dealing with directories and files, Python uses the term *package* for a (sub-) directories and *module* for files within directories, that is modules.[5]

To prevent having multiple modules having the same name, name-spaces are used[6]. The main name-space shall be analogous to the projects' name: `qde`. Underneath the source code folder `src`, each sub-folder represents a package and acts therefore also as a name-space.

To actually allow a whole package and its modules being imported *as modules*, it needs to have at least a file inside called `__init__.py`. Those files may be empty or they may contain regular source code such as classes or methods.

The first stage of the project shows the creation of the *editor* component, as it provides the possibility of creating and editing real-time animations which may then be played back by the *player* component[1, p. 29].

5.3 Editor

This chapter describes the creation of the *editor* component.

The *editor* component shall be placed within the `editor` directory beneath the `src/qde` directory tree. As stated in the prior chapter this requires as well an `__init__.py` file to let Python recognize the `editor` directory as a importable module. This fact and the creation of it is mentioned here for the sake of completeness. Later on it will be assumed as given and only the source code blocks for the creation of the `__init__.py` files are provided.

5.3.1 DONE Main application

The main class of a Qt application using a graphical user interface (GUI) is provided by the class `QApplication`. According to [7] the class may be initialized and used directly without sub-classing it. It may however be useful to sub-class it nevertheless as this provides higher flexibility. Therefore the class `Application` is introduced, which sub-classes the `QApplication` class.

At this point it is necessary to think about the functionality of the class `Application` itself. Very roughly sketched, such a type of application initializes resources, enters a main loop where it stays until told to shut down. At the end it frees resources again.

Due to the usage of `QApplication` as super class it is not necessary to implement a main (event-) loop, as such is provided by Qt itself [8].

Concerning the initialization of resources[9], the application has to act as central node between the various layers of the architecture, initializing them and connecting them using signals.[1, S. 37 bis 38]

Before going into too much details about the actual `Application` class, let us first have a look at the structure of a Python module. Each (proper) Python module contains an (optional) file encoding,

[4] `http://orgmode.org/manual/mkdirp.html#mkdirp`
[5] `https://docs.python.org/3/reference/import.html#packages`
[6] `https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces`
[7] `http://doc.qt.io/Qt-5/qapplication.html`
[8] `http://doc.qt.io/Qt-5/qapplication.html#exec`
[9] `https://www.python.org/dev/peps/pep-0263/`

a docstring[10], imports of other modules and either loose methods or a class definition with methods underneath.

The main module `application` containing also the `Application` class, looks therefore as follows.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

"""Main application module for QDE."""

<<app-application-imports>>

<<app-application-class-definition>>
```

Listing 5: Main application module holding the `Application` class.

Imports

As you can see, the imports of the module are defined by `«app-application-imports»`. For achieving better readability, the imports are split up into system imports, meaning modules provided by the Python library itself or external modules, and project imports, modules created within the project. The imports are therefore split up as follows.

```
# System imports
<<app-application-system-imports>>

# Project imports
<<app-application-project-imports>>
```

Listing 6: `«app-application-imports»`, definition of the application modules' imports.

As the actual imports are not known yet, let us first look at the applications' structure, defined by `«app-class-definition»`. The class is defined by its name, its super class (the parent class) and a class body. As stated at the beginning, the class will inherit from the Qt class `QApplication`, which provides the basics for a Qt GUI application.

```
class Application(QtWidgets.QApplication):
    """Main application for QDE."""

    <<app-application-class-body>>
```

Listing 7: `«app-application-class-definition»`, definition of the `Application` class.

As stated before and as clearly can be seen the class inherits from `QApplication`. This base class is not yet defined however which would produce an error when executing the main class. It is therefore necessary to make that base class available by importing it. As `QApplication` is an external class, not defined by this project, its import is added to the system imports.

Python offers multiple possibilities concerning imports:

- `from foo import bar` or
  `import foo.bar`

  Imports the module `bar` from the package `foo`. All classes, methods and variables within `bar` are then accessible.

- `from foo import bar as baz` or
  `import foo.bar as baz`

  The importing is the same as above, `bar` is masked as `baz` although. This can be convenient when multiple modules have the same name.

[10] `https://www.python.org/dev/peps/pep-0257/#what-is-a-docstring`

- `from bar import SomeClass` or
  `import bar.SomeClass` or
  `import bar.SomeClass as SomeClass`

  Imports the class `SomeClass` from the module `bar`.

- `from foo.bar import some_method` or
  `import foo.bar.some_method` or
  `import foo.bar.some_method as some_method`

  Imports the method `some_method` from the module `bar`.

- `from foo import *` or
  `import * from foo`

  Imports *all* sub-packages and sub-modules from the package `foo`. However, explicit importing is better than implicit and therefore this option should not be used.[11]

- `from bar import *` or `import * from bar`

  Imports *all* classes and methods from the module `bar`. As stated above, explicit importing is better than implicit and therefore this option should also not be used.

As the naming of the PyQt5 modules prefixes them by *Qt*, it is very unlikely to have naming conflicts with other modules. Therefore the import format `from PyQt5 import [QtModuleName]` is used. This still provides a (relatively) unique naming most probably without any conflicts but reduces the effort when writing a bit. The import of system modules is therefore as follows.

```
1 from PyQt5 import QtGui
2 from PyQt5 import QtWidgets
```

Listing 8: «app-application-system-imports», import of system imports.

At this point of time it is rather unclear what the classes body consists of. What surely must be done, is initializing the class's parent, `QApplication`. Additionally it would be nice to having a matching title for the window set as well as maybe an icon for the application. The class's body therefore solely consists its constructor, as follows.

```
1 <<app-application-constructor>>
2
3 def setup_components(self):
4     <<app-application-methods-setup-components>>
5
6 def setup_connections(self):
7     <<app-application-methods-setup-connections>>
8
9 <<app-application-methods>>
```

Listing 9: «app-application-class-body», body of the class `Application`, containing only the constructor at the moment.

When looking at the constructor of the `QApplication` class[12] (as the documentation of PyQt does not provide a proper description and points to the C++ documentation), one can see that it needs the argument count `argc` as well as a vector `argv` containing the arguments. The argument count states how many arguments are being held by the argument vector `argv`. In the PyQt implementation however, only one argument is necessary: a list containing the arguments. `argc` may easily be derived by e.g. `len(arguments)`. Therefore it is necessary for to constructor to take in `arguments` as a required parameter. As described in section 4.2.1, a method for setting up the connections, which may be defined later on, is added to the constructor. The application's constructor looks hence as follows.

[11] https://www.python.org/dev/peps/pep-0020/
[12] https://doc.qt.io/qt-5/qapplication.html#QApplication

```
def __init__(self, arguments):
    """Constructor.

    :param arguments: a (variable) list of arguments, that are
                      passed when calling this class.
    :type  argv:      list
    """

    super(Application, self).__init__(arguments)
    self.setWindowIcon(QtGui.QIcon("assets/icons/im.png"))
    self.setApplicationName("QDE")
    self.setApplicationDisplayName("QDE")

    self.setup_components()
    self.setup_connections()
```

Listing 10: «app-application-constructor», constructor of the `Application` class.

5.3.2 DONE Main entry point

If you run the application at this point nothing happens. Python is able to resolve all dependencies but as there is no `main` function there is nothing else to do, so nothing happens. The execution of the main loop is started when calling the `exec` function of a `QApplication`. So, for actually being able to start the application, a `main` function is needed, which creates an instance of the `Application` class and then runs its `exec` function.

The main function could easily be added to the `Application` class, but for somebody who is not familiar with this applications' structure, this might be rather confusing. Instead a `editor.py` file at the root of the source directory `src` is much more intuitive.

All that the main file shall do, is creating an instance of the main application, execute it and exit at the end of its life cycle.

As stated in , the constructor of `QApplication` requires the argument `arguments` to be passed in (yes, the naming may be a bit confusing here). The `arguments` argument is a list of arguments passed when calling the main entry point of the editor application. For example when calling `python editor.py foo bar baz`, the variable `arguments` would be the list `[foo, bar, baz]` with `len(arguments)` being 3. To obtain the passed-in arguments, the `argv` attribute of the `sys` module may be used, as this holds exactly the list of the passed-in arguments when calling a Python script.

The main entry script of the editor `editor.py` is therefore defined as follows.

```
    #!/usr/bin/python
    # -*- coding: utf-8 -*-

    """ Main entry point for the QDE editor application. """

    # System imports
    import sys

    # Project imports
    from qde.editor.application import application


    if __name__ == "__main__":
        app = application.Application(sys.argv)
        status = app.exec()
        sys.exit(status)
```

Listing 11: «main», the main entry point for the whole editor application.

If you run the application now, a bit more happens. Python is able to resolve all dependencies and to find a `main` function which is then called. The `main` function creates an instance of the `Application` class and executes it by calling `exec`. This in turn enters the Qt main loop which keeps the application

running unless explicitly told to shut down. But at this point there is nothing who could receive the request to shut down, so the only possibility to shut down the application is to quit or kill the spawned Python process itself — not very nice.

5.3.3 DONE Components

Instead it would be nice to have at least a window shown when starting the application, which allows a normal, deterministic and convenient shut down of the application, either by a keyboard shortcut or by selecting an appropriate option in the applications' menu.

But having only a plain window is not that interesting, so this might be a good time to look at the components of the editor, which are defined by [1, p. 29 ff.] and are the following:

- A scene graph, allowing the creation and deletion of scenes. The scene graph has at least a root scene.
- A node-based graph structure, allowing the composition of scenes using nodes and connections between the nodes.
- A parameter window, showing parameters of the currently selected graph node.
- A rendering window, rendering the currently selected node or scene.
- A sequencer, allowing a time-based scheduling of defined scenes.

What [1] does not explicitly mention, is the main window, which holds all those components and allows a proper shut down of the application.

As a starting point, we shall implement the class `MainWindow` representing the main window.

5.3.4 TODO Main window

TODO Main window

Before implementing the features of the main window, its features will be described. The main window is the central aspect of the graphical user interface and is hence part of the `gui` package.

Its main functionality is to set up the actual user interface, containing all the components, described by 5.3.3, as widgets. Qt offers the class `QMainWindow` from which `MainWindow` may inherit. The thoughts about the implementation follow section 4.2.1.

The first step is setting up the necessary signals. They may not all be known at this point and may therefore be expanded later on. As described in section 5.3.3, it would be nice if `MainWindow` would react to a request for closing it, either by a keyboard shortcut or a menu command. However, `MainWindow` is not able to force the `Application` to quit by itself. It would be possible to pass `MainWindow` a reference to `Application` but that would lead to a somewhat tight coupling and is therefore not considered as an option. Signals and slots allow exactly such cross-layer communication without coupling components tightly.

First, the outline of `MainWindow` is defined.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

""" Module holding the main application window. """

# System imports
<<main-window-system-imports>>

# Project imports
<<main-window-project-imports>>


class MainWindow(QtWidgets.QMainWindow):
    """The main window class.
    Acts as an entry point for the QDE editor application.
    """

    <<main-window-signals>>

    def __init__(self):
        """Constructor."""

        <<main-window-constructor>>

    <<main-window-methods>>

    <<main-window-slots>>
```

Listing 12: Module holding the main application window class, `MainWindow`.

A fitting name for the signal, when the window and therefore the application, shall be closed might be `window_closing`. The signal is introduced as follows.

```
# Signals
window_closing = QtCore.pyqtSignal()
```

Listing 13: Definition of signals for the main application window class, `MainWindow`.

Now, that the signal for closing the window and the application is defined, two additional things need to be considered: The emission of the signal by `MainWindow` itself as well as the consumption of the signal by a slots of other classes.

First, the emission of the signal is implemented. The signal shall be emitted when the escape key on the keyboard is pressed or when the corresponding menu item was selected. For the first case, the keyboard event, Qt provides luckily events which may be used. Their outline is already provided by the parent class `QMainWindow` and therefore the event(s) simply need to be implemented. The event which listens to keyboard keys being pressed is called `keyPressEvent` and provides an event-object of type `QEvent`. All there is to do, is to retrieve the event's key by calling its `key` method and check if that key is actually the escape key by comparing it to `Key_Escape`, provided by Qt. If this comparison is true, the signal shall be emitted.

```
def keyPressEvent(self, event):
    """Gets triggered when a key press event is raised.

    :param event: holds the triggered event.
    :type  event: QKeyEvent
    """

    if event.key() == QtCore.Qt.Key_Escape:
        self.window_closing.emit()
    else:
        super(MainWindow, self).keyPressEvent(event)
```

Listing 14: Implementation of the `keyPressEvent` method on the `MainWindow` class.

Additionally the signal shall be emitted when selecting a corresponding menu item. But currently there is no such menu item defined. Qt handles interactions with menu items by using actions (`QAction`). They provide themselves a couple of signals, one being `triggered`, which gets emitted as soon as the action was triggered by a clicking on a menu item. As it is not possible to connect a signal with another signal, a slot, which receives the signal, needs to be defined. A slot is an annotated method.

```
@QtCore.pyqtSlot()
def on_quit(self):
    """Slot which emits the :any:`window_closing` signal.
    This slot gets triggered upon the selection of the menu item to close the
    QDE application.
    """

    self.window_closing.emit()
```

Listing 15: The `on_quit` method, which acts as a slot when the menu item for quitting the application was triggered.

Now the main window is able to emit the signal it is shutting down (or rather it would like to shut down), but so far no one is listening to that signal, so nothing happens when that signal is being emitted.

This leads to an expansion of the main application's construction: The main application has to create a main window an listen to its `window_closing` signal. Luckily `Application` provides already a `quit` slot through `QApplication`.

```
self.main_window = qde_main_window.MainWindow()
self.main_window.window_closing.connect(self.quit)
```

Listing 16: Expansion of setting up the main application's components by the initialization of `MainWindow` and its signals.

So far none of the additional modules have been defined as there are no additional modules imported yet. The missing modules need to be added to the main application as well as the main window.

```
from qde.editor.gui import main_window as qde_main_window
```

Listing 17: Expansion of `«app-application-project-imports»` by the missing imports.

```
from PyQt5 import QtCore
from PyQt5 import QtWidgets
```

Listing 18: Expansion of `«main-window-system-imports»` by the missing imports.

Yet the constructor for the main window is still missing, so running the application would still do nothing. Therefore the constructor for the main window is now implemented. At the current point its solely purpose is to call the its parent's constructor.

```
super(MainWindow, self).__init__()
```

Listing 19: Constructor for the main window class `MainWindow`.

Although a Python process is spawned when starting the application, the main window is still not shown. The problem is, that the main window has no central widget set[13]. Setting a central widget and setting a layout for it solves this problem.

The above described task matches perfectly the second point described in section 4.2.1. The described task will therefore be put in a method named `setup_ui` and the constructor will be expanded correspondingly.

[13] `http://doc.qt.io/qt-5/qmainwindow.html#creating-main-window-components`

The method `setup_ui` seems also a very good place for setting things like the size of the window, setting its (object-) name and its title as well as moving it to a position on the user's screen. To ensure that the window is not hidden behind other windows, the method `activateWindow` coming from `QWidget` is called.

As it is not sure at this point, if the main window will receive additional methods, it may be wise to split `«main-window-methods»` up, by inserting the yet known methods (only `setup_ui` so far) explicitly. This provides the advantage, that new methods can easily be appended and the implemented methods may be expanded easily as well.

```
<<main-window-keypressevent>>

<<main-window-setupui>>
```

Listing 20: The placeholder `«main-window-methods»` declared explicitly.

```
def setup_ui(self):
    """Sets up the user interface specific components."""

    self.setObjectName('MainWindow')
    self.setWindowTitle('QDE')
    self.resize(1024, 768)
    self.move(100, 100)
    self.activateWindow()

    central_widget = QtWidgets.QWidget(self)
    central_widget.setObjectName('central_widget')
    grid_layout = QtWidgets.QGridLayout(central_widget)
    central_widget.setLayout(grid_layout)
    self.setCentralWidget(central_widget)
    self.statusBar().showMessage('Ready.')
```

Listing 21: The method `setup_ui`, which was added to `«main-window-methods» before, for setting up user interface specific tasks within the main window class =MainWindow`.

Now the `setup_ui` method simply needs to be added to the constructor of the class `MainWindow`.

```
self.setup_ui()
```

Listing 22: The method `setup_ui` is added to the constructor of main window class `MainWindow`.

Finally the main window has to be shown by calling its `show` method at the end of the main application's construction.

```
    self.main_window.show()
```

Listing 23: The main window is being shown at the end of the main application's construction.

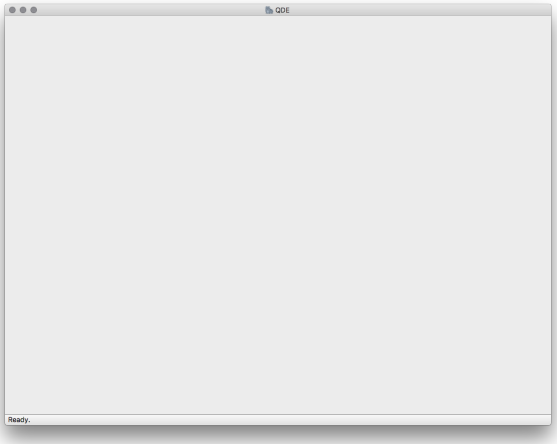

Figure 5.1: The QDE editor application in a very early stage, containing only a grid layout.

When starting the application a plain window containing a grid layout is shown, as can be seen in figure 5.1. As written in 5.3.3 and shown in [1, p. 29 ff.], the main window will contain all the components. To ensure, that those components are shown as defined, a simple grid layout may not provide enough possibilities.

A possible solution to reach the desired layout is to use the horizontal box layout `QHBoxLayout` in combination with splitters. The horizontal box layout lines up widgets horizontally where as the splitters allow splitting either horizontally or vertically. Recalling the components from 5.3.3, the following are needed:

- A scene graph, on the left of the window, covering the whole height
- A node graph on the right of the scene graph, covering as much height as possible
- A view for showing the properties (and therefore parameters) of the selected node on the right of the node graph, covering as much height as possible
- A display for rendering the selected node, on the right of the properties view, covering as much height as possible
- A sequencer at the right of the scene graph and below the other components at the bottom of the window, covering as much width as possible

To sum up, a horizontally box layout and a vertical splitter allow splitting the main window in two halves: The left side will be used for the scene graph where as the other side will hold the remaining components. As the sequencer is located below the other components of the right side, a horizontal splitter is needed for proper separation. The components above the sequencer could simply be added to the right side of the split as a horizontal box layout builds the layout's basis, for convenience however, additional splitters will be used. This allows the user to re-arrange the layout to his taste. To achieve the described layout, the following tasks are necessary:

- Create a widget for the horizontal box layout
- Create the horizontal box layout
- Add the scene graph to the horizontal box layout
- Instantiate the components of the split's right side
  - The node graph
  - The parameter view
  - The rendering view
- Create a horizontal splitter

  - Add the rendering view to it
  - Add the parameter view to it
- Create a vertical splitter
  - Add the horizontally splitter to it
  - Add the scene graph to it
- Add the vertical splitter to the horizontal box layout

The implementation of the explained layout is done in the `setup_ui` method and is as follows. For the not yet existing widgets placeholders are used.

```
    horizontal_layout_widget = QtWidgets.QWidget(central_widget)
    horizontal_layout_widget.setObjectName('horizontal_layout_widget')
    horizontal_layout_widget.setGeometry(QtCore.QRect(12, 12, 781, 541))
    horizontal_layout_widget.setSizePolicy(QtWidgets.QSizePolicy.MinimumExpanding,
                                           QtWidgets.QSizePolicy.MinimumExpanding)
    grid_layout.addWidget(horizontal_layout_widget, 0, 0)

    horizontal_layout = QtWidgets.QHBoxLayout(horizontal_layout_widget)
    horizontal_layout.setObjectName('horizontal_layout')
    horizontal_layout.setContentsMargins(0, 0, 0, 0)

    <<main-window-setupui-scenegraph>>
    <<main-window-setupui-nodegraph>>
    <<main-window-setupui-parameterview>>
    <<main-window-setupui-renderview>>

    horizontal_splitter = QtWidgets.QSplitter()
    <<main-window-setupui-add-renderview-to-horizontal-splitter>>
    <<main-window-setupui-add-parameterview-to-horizontal-splitter>>

    vertical_splitter = QtWidgets.QSplitter()
    vertical_splitter.setOrientation(QtCore.Qt.Vertical)
    vertical_splitter.addWidget(horizontal_splitter)
    <<main-window-setupui-add-nodegraph-to-vertical-splitter>>

    horizontal_layout.addWidget(vertical_splitter)
```

Listing 24: Lay-outing of the main window by expanding the `setup_ui` method.

All the above taken actions to lay out the main window change nothing in the window's yet plain appearance. This is quite obvious, as none of the actual components are implemented yet.

The most straight-forward component to implement may be scene graph, so this is a good starting point for the implementation of the remaining components.

5.3.5 DONE Scene graph

The scene graph component does, as also the other components do, have two aspects to consider: A graphical aspect as well as its data structure. As written in section 4.2.1, each component has a view — residing in the *gui* package —, a model — residing in the *domain* package — and a controller acting as workflow object — residing in the *application* package.

The `SceneGraphController` class will manage instances of scene models whereas the `SceneGraphView` will display a tree of scenes, starting with a root scene of type `SceneModel`.

The least tedious of those aspects may be the scene model, `SceneModel`, so the scene model is implemented first.

As at this point its functionality is not known, its implementation is rather dull. It is composed of solely an empty constructor.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

""" Module holding scene related aspects concerning the domain layer. """

# System imports
<<domain-scene-system-imports>>

# Project imports
<<domain-scene-project-imports>>


class SceneModel(object):
    """The scene model.
    It is used as a base class for scene instances within the scene graph.
    """

    <<domain-scene-signals>>

    <<domain-scene-constructor>>

    <<domain-scene-methods>>

    <<domain-scene-slots>>
```

Listing 25: Scene module inside the `domain` package, holding the `SceneModel` class.

```
def __init__(self):
    pass
```

Listing 26: Constructor of the scene model class, `SceneModel`.

Scenes may now be instantiated, it is however important to do the management of scenes in a controlled manner. This is where the specific controllers within the `application` layer come in, as described in more detail in section 4.2.1. Therefore the class `SceneGraphController` will now be implemented, for being able to manage scenes.

As the scene graph shall be built as a tree structure, an appropriate data structure is needed. Qt provides the `QTreeWidget` class, but that class is in this case not suitable, as it does not separate the data from its representation, as stated by Qt: "Developers who do not need the flexibility of the Model/View framework can use this class to create simple hierarchical lists very easily. A more flexible approach involves combining a QTreeView with a standard item model. This allows the storage of data to be separated from its representation."[14]

Therefore the class `QAbstractItemModel`[15] is chosen for implementation. Before implementing the actual methods, it is important to think about the attributes, that the scene graph controller will have. According to the class's documentation, some methods must be implemented at very least: "When subclassing QAbstractItemModel, at the very least you must implement index(), parent(), rowCount(), columnCount(), and data(). These functions are used in all read-only models, and form the basis of editable models."

For being able edit the nodes of the scene graph and to have a custom header displayed, further methods have to be implemented: "To enable editing in your model, you must also implement setData(), and reimplement flags() to ensure that ItemIsEditable is returned. You can also reimplement headerData() and setHeaderData() to control the way the headers for your model are presented."

From the remarks above the attributes may be defined. As the scene graph is implemented as a tree structure, it must have a **root node**, which is of type `SceneGraphViewModel` (coming from the `gui_domain` layer). Whenever a scene is added as a node, the item model needs to be informed for updating the display. This happens by emitting the `rowsInserted` signal, which is already given by the `QAbstractItemModel` class. This signal needs the current model index as well as the first and last position as parameters. The

[14] `http://doc.qt.io/qt-5/qtreewidget.html#details`
[15] `http://doc.qt.io/qt-5/qabstractitemmodel.html`

current model index represents the parent of the item to add, whereas the item will be inserted between the two given positions, first and last. Concerning the model index the Qt documentation states: "An invalid model index can be constructed with the QModelIndex constructor. Invalid indexes are often used as parent indexes when referring to top-level items in a model." Therefore for creating the initial node of the scene graph, the root node, the constructor of `QModelIndex` will be used. As **header data** the name of the scenes as well as the number of nodes a scene contains shall be displayed.

Speaking of signals, brings up the definition of signals for the scene graph controller. To prevent coupling, two signals are added: `scene_added` and `scene_removed`. The first will be emitted whenever a new node is inserted into the scene graph by `insertRows` being called. The latter is emitted whenever an existing node is removed from the scene graph by calling the `removeRows` method.

But what currently is missing for being able to implement a first draft of the scene graph, is the view model `SceneGraphViewModel`. View models are used to visually represent something within the graphical user interface and they provide an interface to the `domain` layer. To this point, a simple reference in terms of an attribute is used, which may be changed later on. Concerning the user interface, a view model must fulfill the requirements posed by the user interface's corresponding component. In terms of the scene graph the view model must provide at least a name and a row. Additionally, as already mentioned, a reference to the domain object is being added. The class inherits from `QObject` as this base class already provides a tree structure, which fits the structure of the scene graph perfectly.

```
1   #!/usr/bin/python
2   # -*- coding: utf-8 -*-
3
4   """ Module holding scene related aspects concerning the gui_domain layer. """
5
6   # System imports
7   from PyQt5 import Qt
8   from PyQt5 import QtCore
9   <<guidomain-scene-system-imports>>
10
11  # Project imports
12  <<guidomain-scene-project-imports>>
13
14  <<guidomain-scene-body>>
```

Listing 27: Scene module inside the `gui_domain` package.

```
<guidomain-scene-body>=
        class SceneGraphViewModel(Qt.QObject):
            """View model representing scene graph items.

            The SceneGraphViewModel corresponds to an entry within the scene graph. It
            is used by the QAbstractItemModel class and must therefore at least provide
            a name and a row.
            """



                # .. py:function::
                def __init__(
                        self,
                        row,
                        domain_object,
                        name=QtCore.QCoreApplication.translate('SceneGraphViewModel', 'New scene'),
                        parent=None
                ):
                    """Constructor.

                    :param row:           The row the view model is in.
                    :type  row:           int
                    :param domain_object: Reference to a scene model.
                    :type  domain_object: qde.editor.domain.scene.SceneModel
                    :param name:          The name of the view model, which will be displayed in
                                          the scene graph.
                    :type  name:          str
                    :param parent:        The parent of the current view model within the scene
                                          graph.
                    :type parent:         qde.editor.gui_domain.scene.SceneGraphViewModel
                    """

                    super(SceneGraphViewModel, self).__init__(parent)
                    self.row  = row
                    self.domain_object = domain_object
                    self.name = name
                        self.scene_view_model = SceneViewModel()
```

Listing 28: Definition of the body of the `scene` module, which is in the `gui_domain` layer.

```
# .. py:function::
def __init__(
        self,
        row,
        domain_object,
        name=QtCore.QCoreApplication.translate('SceneGraphViewModel', 'New scene'),
        parent=None
):
    """Constructor.

    :param row:           The row the view model is in.
    :type  row:           int
    :param domain_object: Reference to a scene model.
    :type  domain_object: qde.editor.domain.scene.SceneModel
    :param name:          The name of the view model, which will be displayed in
                          the scene graph.
    :type  name:          str
    :param parent:        The parent of the current view model within the scene
                          graph.
    :type parent:         qde.editor.gui_domain.scene.SceneGraphViewModel
    """

    super(SceneGraphViewModel, self).__init__(parent)
    self.row  = row
    self.domain_object = domain_object
    self.name = name
```

Listing 29: Constructor for the scene graph view model, `SceneGraphViewModel`.

Now, with the scene graph view model being available, the scene graph controller may finally be implemented.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

""" Module holding scene graph related aspects concerning the application layer.
"""

# System imports
from PyQt5 import QtCore
<<app-scenegraph-system-imports>>

# Project imports
from qde.editor.domain     import scene as domain_scene
from qde.editor.gui_domain import scene as guidomain_scene
<<app-scenegraph-project-imports>>


class SceneGraphController(QtCore.QAbstractItemModel):
    """The scene graph controller.
    A controller for managing the scene graph by adding, editing and removing
    scenes.
    """

    scene_added = QtCore.pyqtSignal(domain_scene.SceneModel)
    scene_removed = QtCore.pyqtSignal(domain_scene.SceneModel)
    <<app-scenegraph-controller-signals>>

    def __init__(self, root_node_domain_object, parent=None):
        """Constructor.

        :param root_node_domain_object: The domain object of the root node of
                                        the scene graph view model.
        :type root_node_domain_object:  qde.editor.domain.scene.SceneModel
        :param parent: The parent of the current view model within the scene
                       graph.
        :type parent:  qde.editor.gui_domain.scene.SceneGraphViewModel
        """

        super(SceneGraphController, self).__init__(parent)
        self.header_data = [
            QtCore.QCoreApplication.translate(__class__.__name__, 'Name'),
            QtCore.QCoreApplication.translate(__class__.__name__, '# Nodes')
        ]
        self.root_node = guidomain_scene.SceneGraphViewModel(
            row=0,
            domain_object=root_node_domain_object,
            name=QtCore.QCoreApplication.translate(__class__.__name__, 'Root scene')
        )
        self.rowsInserted.emit(QtCore.QModelIndex(), 0, 1)
        <<app-scenegraph-controller-constructor>>

    <<app-scenegraph-controller-methods>>

    <<app-scenegraph-controller-slots>>
```

Listing 30: The outline of the `SceneGraphController` class, inside the `application` package.

At this point data structures in terms of a (data-) model, which holds the actual, for the scene graph relevant data of a scene, and a view model, which holds the data relevant for the user interface, are implemented. Further a controller for handling the flow of the data for both models is implemented. What is still missing, is the actual representation of the scene graph in terms of a view.

Qt offers a plethora of widgets for implementing views. One such widget is `QTreeView`, which "implements a tree representation of items from a model. This class is used to provide standard hierarchical lists that were previously provided by the QListView class, but using the more flexible approach provided by Qt's

model/view architecture."[16]

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

""" Module holding scene related aspects concerning the graphical user interface layer.
"""

# System imports
from PyQt5 import QtWidgets
<<gui-scene-system-imports>>

# Project imports
<<gui-scene-project-imports>>


<<gui-scene-graph-class-decorators>>
class SceneGraphView(QtWidgets.QTreeView):
    """The scene graph view widget.
    A widget for displaying and managing the scene graph.
    """

    # Signals
    <<gui-scene-graph-signals>>

    def __init__(self, parent=None):
        """Constructor.

        :param parent:        The parent of the current view widget.
        :type parent:         QtCore.QObject
        """

        super(SceneGraphView, self).__init__(parent)
        <<gui-scene-graph-constructor>>

    <<gui-scene-graph-methods>>

    # Slots
    <<gui-scene-graph-slots>>
```

Listing 31: The outline of the `SceneGraphView` class, within the `scene` module of the `gui` package.

Having the scene graph view implemented as a widget, it is now necessary to add the widget to the main window and initializing it. As described in section TODO, the widget is added to the horizontal layout, using the earlier defined `main-window-setupui-scenegraph` placeholder. For being able to instantiate a scene graph widget, its module must be imported as well. The maximum width of the widget is limited by using the `setMaximumWidth` method.

```
from qde.editor.gui import scene as guiscene
```

Listing 32: Import of the `scene` module from the `gui` layer.

```
self.scene_graph_widget = guiscene.SceneGraphView()
self.scene_graph_widget.setObjectName('scene_graph')
self.scene_graph_widget.setMaximumWidth(300)
horizontal_layout.addWidget(self.scene_graph_widget)
```

Listing 33: The scene graph widget is being initialized and added to the horizontal layout.

When starting the editor application now, after implementing and adding the scene graph widget, the widget appears on the left side of the main window. It does not provide any functionality yet.

[16] `http://doc.qt.io/qt-5/qtreeview.html#details`

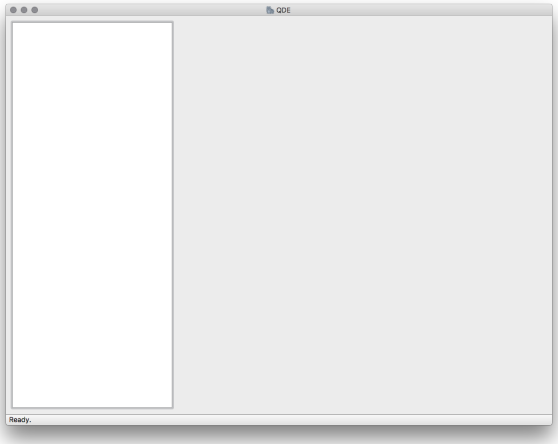

Figure 5.2: The QDE editor application having the scene graph widget added, which is visible as a blank, white rectangle on the left of the window.

For finally being able to manage scenes within the scene graph, a few aspects are still missing, which will be tackled now.

First of all, the scene graph appears to hold no data at all. This is not surprising, as no scene nodes were added by now, which might be a good point to start with. Actually this is not the entire truth, as the root node (view model) was already added within the scene graph controller. The controller emits the signal, that a row was inserted, but no other component is receiving this signal. Obviously this could be achieved by connecting the scene graph controller and the scene graph view, but as Qt's model/view approach is at least partially used, simply setting the view's model leads to the same result while providing greater functionality.

```
1    <app-application-methods-setup-connections>=
2        <<app-application-methods-setup-connections-01>>
```

Listing 34: The method `setup_connections` being defined by setting the scene graph widget's model.

The component that ties the layers together, is, as previously described, the main application. This means, that the main application has to provide all the necessary data structures and controllers. Regarding the scene graph this means setting up a root scene (as a domain-/data-model) and setting up the scene graph controller. As the main application's layer, the `application` layer, is directly below the layer of the view models, `gui_domain` this opposes no problem.

Therefore the root scene as well as the scene graph controller will be implemented in the main application's `setup_components` method, whereas setting the scene graph widget's model will be implemented in the `setup_connections` method.

```
1    <app-application-methods-setup-components>+=
2        <<app-application-methods-setup-components-01>>
```

Listing 35: The method `setup_components` being expanded by the creation of the root scene as well as the scene graph controller.

The necessary imports are still missing however, so those are added to the main application's imports.

```
1    <app-application-project-imports>+=
2        from qde.editor.gui import main_window as qde_main_window
3            from qde.editor.domain import scene
4            from qde.editor.application import scene_graph
```

Listing 36: Expansion of the main application's imports by the necessary packages.

The application is still not showing the desired result: The display of the scene graph in form of a tree containing the root node. When looking at the outputs of the application, the messages as seen in listing 37 can be observed.

```
    NotImplementedError: QAbstractItemModel.columnCount() is abstract and must be overridden
    NotImplementedError: QAbstractItemModel.rowCount() is abstract and must be overridden
```

Listing 37: Output (erroneous) when running the editor application.

The messages from listing 37 state, that not all of the necessary methods from the sub-classed `QAbstractItemModel` are implemented yet. Currently the methods `columnCount` and `rowCount` are missing. Those methods return "the number of columns for the children of the given parent"[17] and "the number of rows under the given parent"[18] respectively. The implementation of those missing methods are as follows in listing 38. The method `columnCount` is trivial, as there will always be only two columns (as defined by the header in listing 30): The name of the scene and the number of nodes it contains. The method `rowCount` shall return `1` if the parent is invalid, otherwise it shall return the parent's children.

```
    <app-scenegraph-controller-methods>=
            def columnCount(self, parent):
                """Return the number of columns for the children of the given parent.

                :param parent: The index of the item in the scene graph, which the
                               column count shall be returned for.
                :type  parent: QtCore.QModelIndex

                :return: the number of columns for the children of the given parent.
                :rtype:  int
                """

                return len(self.header_data)

            def rowCount(self, parent):
                """Return the number of rows for the children of the given parent.

                :param parent: The index of the item in the scene graph, which the
                               row count shall be returned for.
                :type  parent: QtCore.QModelIndex

                :return: the number of rows for the children of the given parent.
                :rtype:  int
                """

                if not parent.isValid():
                    return 1

                # Get the actual object stored by the parent. In this case it is a
                # SceneGraphViewModel.
                node = parent.internalPointer()

                return len(node.children())
```

Listing 38: The code block «app-scenegraph-controller-methods», defining the methods `columnCount` and `rowCount` within the scene controller.

When running the application now, there is still an error message, although a new one as can be seen in listing 39.

```
    NotImplementedError: QAbstractItemModel.index() is abstract and must be overridden
```

Listing 39: Output (erroneous) when running the editor application.

[17] http://doc.qt.io/qt-5/qabstractitemmodel.html#columnCount
[18] http://doc.qt.io/qt-5/qabstractitemmodel.html#rowCount

This time the `index` method is missing in the scene controller. According the documentation, the method "returns the index of the item in the model specified by the given row, column and parent index."[19] Furthermore, "when reimplementing this function in a subclass, call createIndex() to generate model indexes that other components can use to refer to items in your model."[20]

The implementation of the missing method `index` is as follows in listing 40. The method needs to return the index of the given row and column for the given parent. There are two cases however: either the parent is valid or it is not. In the former case, the scene graph view model of the parent is extracted and an index based on the row, the column and the child node at the given row as parent is being created. In the latter case, when the given parent is not valid, an index based on the scene graph's root node is created.

```
    <app-scenegraph-controller-methods>+=
            def index(self, row, column, parent=QtCore.QModelIndex()):
                """Return the index of the item in the model specified by the given row,
                column and parent index.

                :param row: The row for which the index shall be returned.
                :type  row: int
                :param column: The column for which the index shall be returned.
                :type column: int
                :param parent: The parent index of the item in the model. An invalid model
                               index is given as the default parameter.
                :type parent: QtQore.QModelIndex

                :return: the model index based on the given row, column and the parent
                         index.
                :rtype: QtCore.QModelIndex
                """

                # If the given parent (index) is not valid, create a new index based on the
                # currently set root node
                if not parent.isValid():
                    return self.createIndex(row, column, self.root_node)

                # The internal pointer of the the parent (index) returns a scene graph view
                # model
                parent_node = parent.internalPointer()
                child_nodes = parent_node.children()

                return self.createIndex(row, column, child_nodes[row])
```

Listing 40: The code block «app-scenegraph-controller-methods», is expanded by the `index` method within the scene controller.

Although the scene graph is showing now two columns when running the editor application, there are still error messages, as shown in listing 41.

```
    NotImplementedError: QAbstractItemModel.parent() is abstract and must be overridden
    NotImplementedError: QAbstractItemModel.data() is abstract and must be overridden
```

Listing 41: Output (erroneous) when running the editor application.

The methods `parent` and `data` are missing from the implementation. The Qt documentation states about `parent`: "Returns the parent of the model item with the given index. If the item has no parent, an invalid QModelIndex is returned.

A common convention used in models that expose tree data structures is that only items in the first column have children. For that case, when reimplementing this function in a subclass the column of the returned QModelIndex would be 0.

[19] http://doc.qt.io/qt-5/qabstractitemmodel.html#index
[20] http://doc.qt.io/qt-5/qabstractitemmodel.html#index

When reimplementing this function in a subclass, be careful to avoid calling QModelIndex member functions, such as QModelIndex::parent(), since indexes belonging to your model will simply call your implementation, leading to infinite recursion."[21]

Those remarks lead to the implementation, that can be seen in listing 42.

```
<app-scenegraph-controller-methods>+=
        def parent(self, model_index):
            """Return the parent of the model item with the given index. If the item has
            no parent, an invalid QModelIndex is returned.

            :param model_index: The model index which the parent model index shall be
                                derived for.
            :type model_index: int

            :return: the model index of the parent model item for the given model index.
            :rtype: QtCore.QModelIndex
            """

            if not model_index.isValid():
                return QtCore.QModelIndex()

            # The internal pointer of the the model index returns a scene graph view
            # model.
            node = model_index.internalPointer()
            if node.parent() is None:
                return QtCore.QModelIndex()
            else:
                return self.createIndex(node.parent().row, 0, node.parent())
```

Listing 42: The code block «app-scenegraph-controller-methods», is expanded by the parent method within the scene controller.

About the data method, the Qt documentation says the following:

"Returns the data stored under the given role for the item referred to by the index.

Note: If you do not have a value to return, return an invalid QVariant instead of returning 0."[22]

The scene graph stores two different kinds of data: the name of the scene and its nodes. Which of the two gets returned depends on the column. The first column, column 0, returns the name, where as the second column, column 1, returns the number of nodes the scene contains. It is not yet possible to implement the second case, as scenes itself do not exist (as view models) and are not yet provided as a reference within the scene graph view model.

For still being able to follow the current stream of thought, only a minimalist realization of the scene view model class SceneViewModel is provided by now, as can be seen in listing 43.

```
<guidomain-scene-body>+=
        class SceneViewModel(Qt.QObject):
            """View model representing a scene.

            The SceneViewModel corresponds to an SceneGraphViewModel entry within the
            scene graph.
            """

            pass
```

Listing 43: Expansion of the scene module, which is within the gui_domain layer, by the SceneViewModel class. Note, that the implementation of the class provides no functionality at all at the moment.

Having the scene view model class defined, it may now be used by the scene graph view model. This reference will then be used by the scene graph controller for getting the number of nodes a scene contains.

[21] http://doc.qt.io/qt-5/qabstractitemmodel.html#parent
[22] http://doc.qt.io/qt-5/qabstractitemmodel.html#data

```
<guidomain-scene-scenegraphviewmodel-constructor>+=
            self.scene_view_model = SceneViewModel()
```

Listing 44: Expansion of the constructor of the `SceneGraphViewModel` class by a reference to a scene view model.

All prerequisites for implementing the `data` method of the scene graph controller are now met and the method may therefore now be implemented. The method has two parameters: the model index and the role. The model index holds the position of the item within the data model. The role indicates what type of data is provided. Currently the only role considered is the display of models (further information may be found at[23]). Depending on the column of the model index, either the name of the scene graph node or the number of nodes its scene holds is returned.

```
<app-scenegraph-controller-methods>+=
        def data(self, model_index, role=QtCore.Qt.DisplayRole):
            """Return the data stored unter the given role for the item referred by the
            index.

            :param model_index: The (data-) model index of the item.
            :type model_index: int
            :param role: The role which shall be used for representing the data. The
                         default (and currently only supported) is displaying the data.
            :type role:  QtCore.Qt.DisplayRole

            :return: the data stored under the given role for the item referred by the
                     given index.
            :rtype:  str
            """

            if not model_index.isValid():
                return None

            # The internal pointer of the model index returns a scene graph view
            # model.
            node = model_index.internalPointer()

            if role == QtCore.Qt.DisplayRole:
                # Return either the name of the scene or its number of nodes.
                column = model_index.column()

                if column == 0:
                    return node.name
                elif column == 1:
                    return node.scene_view_model.graph_node_count
```

Listing 45: The code block «app-scenegraph-controller-methods» is expanded by the `data` method within the scene controller.

The editor application would at this point still produce an error when being run. The `data` method accesses a property of the scene view model when getting the second column, the number of nodes a scene contains: `graph_node_count`. As the scene view model is only a placeholder at the moment, it is necessary to implement that property first. As the name says, the property `graph_node_count` returns the number of graph nodes a scene view model contains. Therefore the scene view model needs to hold graph nodes as a list which leads to the definition of its constructor before implementing the `graph_node_count` method.

```
<guidomain-scene-sceneviewmodel-constructor>=
        def __init__(self):
            """Constructor."""

            self.graph_nodes = []
```

Listing 46: Definition of the constructor of the `SceneViewModel` class.

[23] `http://doc.qt.io/qt-5/qt.html#ItemDataRole-enum`

The method `graph_node_count` then simply returns the length of the graph node list, as can be seen in listing 47.

```
<guidomain-scene-sceneviewmodel-methods>+=
        @property
        def graph_node_count(self):
            """Return the number of graph nodes, that this scene contains."""

            return len(self.graph_nodes)
```

Listing 47: Expansion of the scene view model's methods by adding the `graph_node_count` property.

When launching the editor application now, the scene graph is shown containing the root node, as intended. One small detail is still left although. The header data was defined in the scene graph controller, but it is not shown correctly. Only the numbers 1 and 2 are shown as header. To get the header display the column names correctly, the `headerData` method has to be implemented.

The Qt documentation states: "Returns the data for the given role and section in the header with the specified orientation.

For horizontal headers, the section number corresponds to the column number. Similarly, for vertical headers, the section number corresponds to the row number."[24]

At the moment only the displaying-role and a horizontal orientation shall be supported. The sections are given by the two columns 0 and 1, which correspond to the header data. The implementation of the `headerData` is shown in listing 48.

```
<app-scenegraph-controller-methods>+=
        def headerData(self, section, orientation=QtCore.Qt.Horizontal,
                       role=QtCore.Qt.DisplayRole):
            """Return the data for the given role and section in the header with the
            specified orientation.

            Currently vertical is the only supported orientation. The only supported
            role is DisplayRole. As the sections correspond to the header, there are
            only two supported sections: 0 and 1. If one of those parameters is not
            within the described values, None is returned.

            :param section: the section in the header. Currently only 0 and 1 are
                            supported.
            :type  section: int
            :param orientation: the orientation of the display. Currently only
                                Horizontal is supported.
            :type orientation:  QtCore.Qt.Orientation
            :param role: The role which shall be used for representing the data. The
                         default (and currently only supported) is displaying the data.
            :type role:  QtCore.Qt.DisplayRole

            :return: the header data for the given section using the given role and orientation.
            :rtype:  str
            """

            if (
                    orientation == QtCore.Qt.Horizontal  and
                    role        == QtCore.Qt.DisplayRole and
                    section     in [0, 1]
            ):
                return self.header_data[section]
```

Listing 48: Expansion of the scene graph controller's methods by adding the `headerData` method which overwrites the method inherited by `QAbstractItemModel`.

[24] `http://doc.qt.io/qt-5/qabstractitemmodel.html#headerData`

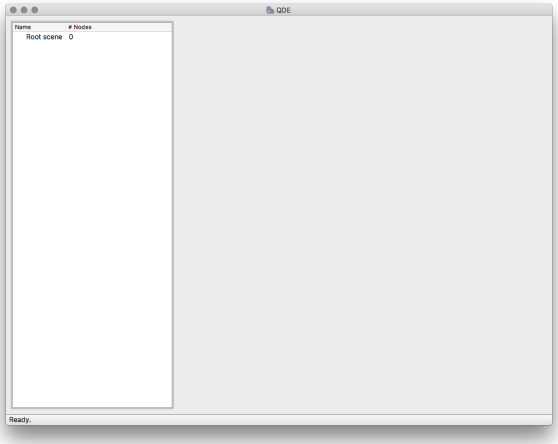

Figure 5.3: The QDE editor application showing the scene graph widget, containing the root node of the scene graph.

So far the application creates an instance of a scene model through the main application, then managed by the scene graph controller. But for having only a single (root-) scene, the whole scene graph architecture would be a massive overkill. Instead it shall be possible to have multiple and nested scenes, what allows the creation of diversified animations. Therefore the scene graph view needs to provide at least the creation of new nodes, the deletion of existing nodes and the selection of a existing nodes. First the selection of existing nodes is implemented.

To detect if a node was selected within the scene tree of the scene graph view, the selection model provides the `selectionChanged` signal. The selection model is inherent in the data model of the `QTreeView`. For being able to use the signal, the `setModel` method of the tree view must be overridden. It is however very important to call the very same method on the parent first. When setting the model, the root item of the model is set to be selected. For more flexibility, the slot `on_tree_item_selected` will be triggered upon a selection of a tree item. The implementation of those aspects can be seen in listings 49, 50, 51, 52 and 53.

```
1  <gui-scene-system-imports>=
2          from PyQt5 import Qt
3          from PyQt5 import QtCore
```

Listing 49: Definition of the necessary system imports for selecting tree items within the view's scene package.

```
1  <gui-scene-project-imports>=
2          from qde.editor.gui_domain import scene
```

Listing 50: Definition of the necessary imports for selecting tree items within the view's scene package.

```
1  <gui-scene-graph-signals>=
2          tree_item_selected = QtCore.pyqtSignal(scene.SceneViewModel)
```

Listing 51: Definition of the signal in case tree items are selected.

```
<gui-scene-graph-methods>+=
        def setModel(self, model):
            """Set the model for the view to present.

            This method is only used for being able to use the selection model's
            selectionChanged method and setting the current selection to the root node.

            :param model: The item model which the view shall present.
            :type  model: QtCore.QAbstractItemModel
            """

            super(SceneGraphView, self).setModel(model)

            selection_model = self.selectionModel()
            selection_model.selectionChanged.connect(
                self.on_tree_item_selected
            )

            self.setCurrentIndex(model.index(0, 0))
```

Listing 52: The overridden `setModel` method coming from `QTreeView` being added to the methods of the scene graph view class.

```
<gui-scene-graph-slots>+=
        @QtCore.pyqtSlot(QtCore.QItemSelection, QtCore.QItemSelection)
        def on_tree_item_selected(self, selected, deselected):
            """Slot which is called when the selection within the scene graph view is
            changed.

            The previous selection (which may be empty) is specified by the deselected
            parameter, the new selection is specified by the selected paramater.

            This method emits the selected scene graph item as scene graph view model.

            :param selected: The new selection of scenes.
            :type  selected: QtCore.QModelIndex
            :param deselected: The previous selected scenes.
            :type  deselected: QtCore.QModelIndex
            """

            selected_item = selected.first()
            selected_index = selected_item.indexes()[0]
            selected_scene_graph_view_model = selected_index.internalPointer()
            self.tree_item_selected.emit(selected_scene_graph_view_model)
```

Listing 53: Definition of the slot which gets called in case tree items are selected.

In the same manner the adding and removal of scenes is implemented. However, the tree widgets does not provide direct signals for those cases as it is the case when selecting a tree item, instead actions have to be used. Two actions are implemented: one for adding a new item and one for removing an existing item.

An action gets triggered, typically by hovering over some item (in terms of a context menu for example) or by pressing a defined shortcut. For the adding and the removal the latter will be used. As the particular shortcut shall only be valid for the widget, the `WidgetShortcut` shortcut context is used. Adding of a scene item shall happen when pressing the `a` key on the keyboard, removal of a selected node upon the press of the `delete` key on the keyboard.

The actions have to be added to the scene graph view and their `triggered` signal is connected with the slot `on_new_tree_item` and `on_tree_item_removed` respectively.

The implementation of the addition and removal of tree items within the scene graph view is shown in listings 57 and 57.

Taking a step back at this point, the (main-) functionality of the editor application is as follows. When starting, an instance of the `Application` class is spawned. As stated before, this class is a central aspect

of the application as it connects the various layers of the architecture. The `Application` class spawns the main window, creates the root scene of the application and spawns the scene graph controller by providing it with the root scene.

```
<gui-scene-graph-signals>+=
        tree_item_added = QtCore.pyqtSignal(QtCore.QModelIndex)
```

Listing 54: The signal in case a tree item is added gets appended to the scene graph widget's signals.

```
<gui-scene-graph-signals>+=
        tree_item_removed = QtCore.pyqtSignal(QtCore.QModelIndex)
```

Listing 55: The signal in case a tree item is removed gets appended to the scene graph widget's signals.

```
<gui-scene-graph-constructor>+=
        new_action_label = QtCore.QCoreApplication.translate(
            __class__.__name__, 'New scene'
        )
        new_action = QtWidgets.QAction(new_action_label, self)
        new_action.setShortcut(Qt.QKeySequence('a'))
        new_action.setShortcutContext(QtCore.Qt.WidgetShortcut)
        new_action.triggered.connect(self.on_new_tree_item)
        self.addAction(new_action)

        remove_action_label = QtCore.QCoreApplication.translate(
            __class__.__name__, 'Remove selected scene(s)'
        )
        remove_action = QtWidgets.QAction(remove_action_label, self)
        remove_action.setShortcut(Qt.QKeySequence('Delete'))
        remove_action.setShortcutContext(QtCore.Qt.WidgetShortcut)
        remove_action.triggered.connect(self.on_tree_item_removed)
        self.addAction(remove_action)
```

Listing 56: The actions to add a new scene and to remove existing scenes are added to the constructor of the scene graph view.

```
<gui-scene-graph-slots>+=
        @QtCore.pyqtSlot()
        def on_new_tree_item(self):
            """Slot which is called when a new tree item was added by the scene graph
            view.

            This method emits the selected scene graph item as new tree item in form of
            a scene graph view model.
            """

            selected_indexes = self.selectedIndexes()

            # Sanity check: is actually an item selected?
            if len(selected_indexes) > 0:
                selected_item = selected_indexes[0]
                self.tree_item_added.emit(selected_item)
                    self.logger.debug("A new scene graph item was added.")
```

Listing 57: The `on_new_tree_item` slot is added to the scene graph view's slots.

```
<gui-scene-graph-slots>+=
        @QtCore.pyqtSlot()
        def on_tree_item_removed(self):
            """Slot which is called when a one or multiple tree items were removed by
            the scene graph view.

            This method emits the removed scene graph item in form of scene graph view
            models.
            """

            selected_indexes = self.selectedIndexes()

            # Sanity check: is actually an item selected? And has that item a parent?
            # We only allow removal of items with a valid parent, as we do not want to
            # have the root item removed.
            if len(selected_indexes) > 0:
                selected_item = selected_indexes[0]
                if selected_item.parent().isValid():
                    self.tree_item_removed.emit(selected_item)
                    self.logger.debug((
                        "The scene graph item at row {row} "
                        "and column {column} was removed."
                    ).format(
                        row=selected_item.row(),
                        column=selected_item.column()
                    ))
```

Listing 58: The `on_tree_item_removed` slot is added to the scene graph view's slots.

When launching the editor application now, the root scene is selected within the scene graph. When pressing the `a` or the `delete` key on the keyboard nothing happens. But why does nothing happen? Let us reconsider.

Both times, when one of the two keys is pressed, the corresponding slot is hopefully called. When the scene graph has a selection, one of the two signals, `tree_item_added` or `tree_item_removed`, is emitted. The problem seems to be, that currently no other component is paying attention to those signals. So let us connect the two signals with a corresponding slot. As stated before, the `Application` class acts as a connection between layers and therefore inter-layer connections have to happen there. The implementation of the connections is shown in listing 59.

```
<app-application-methods-setup-connections>+=
        self.main_window.scene_graph_widget.tree_item_added.connect(
            self.scene_graph_controller.on_tree_item_added
        )
        self.main_window.scene_graph_widget.tree_item_removed.connect(
            self.scene_graph_controller.on_tree_item_removed
        )
```

Listing 59: Connections between the scene graph view and the scene graph controller are added to the `setup_connections` method of the main application.

Setting up the connections as shown in listing 59 connects the scene graph view with the controller. But currently the controller does not know what to do in the case an scene graph item is added or removed as the needed slots are missing.

But what shall actually happen upon those events? In the case a scene graph item is added, a new scene graph entry (a row) has to be added to the data model. In the case an existing scene graph item is being removed, the item has to be removed from the data model.

As the scene graph controller inherits from `QAbstractItemModel` the corresponding methods, `insertRows` and `removeRows`, have to be implemented. First, let us implement the slots as they are very easy to implement. Implementing `on_tree_item_added` is straightforward: the `insertRows` method is called by providing the row, the count and the parent of the new item. Note, that the row is currently always zero. The implementation can be seen in listing 60.

```
<app-scenegraph-controller-slots>+=
        @QtCore.pyqtSlot(QtCore.QModelIndex)
        def on_tree_item_added(self, parent_index):
            """Add a new row under the given parent.

            :param parent_index: The index of the parent item.
            :type  parent_index: QtCore.QModelIndex
            """

            if parent_index.isValid():
                self.insertRows(0, 1, parent_index)
            else:
                # TODO: Log warning or error
                pass
```

Listing 60: The slot `on_tree_item_added` is being added to the scene graph controller's slots.

The implementation of `on_tree_item_added` is analogous: the `removeRows` method is called by providing the row, the count and the parent of the new item. The implementation can be seen in listing 61.

```
<app-scenegraph-controller-slots>+=
        @QtCore.pyqtSlot(QtCore.QModelIndex)
        def on_tree_item_removed(self, selected_index):
            """Remove the currently selected item from the scene graph.

            :param selected_index: The index of the current selection.
            :type  selected_index: QtCore.QModelIndex
            """

            if selected_index.isValid():
                row    = selected_index.row()
                parent = selected_index.parent()
                self.removeRows(row, 1, parent)
            else:
                # TODO: Log warning or error
                pass
```

Listing 61: The slot `on_tree_item_removed` is being added to the scene graph controller's slots.

Having the slots for adding and removing scene graph items implemented, the actual methods for these actions are still missing. So, let us implement these now.

When inserting a row, the first thing to do is calling `beginInsertRows` by providing the index of the parent item, the current row and the last row of insertion (which is the current row plus the count minus one). Then a scene model, representing the actual data structure of a scene, as well as a scene graph view model is being created, representing the very same scene model within the graphical user interface. The transaction is then being ended by calling `endInsertRows`. Finally the view widget is being told to redraw itself by emitting the `layoutChanged` signal and the `scene_added` signal is emitting the newly created domain model to inform other components (subscribers) about the creation. This can be seen in listing 62.

```
    <app-scenegraph-controller-methods>+=
            def insertRows(self, row, count, parent_index=QtCore.QModelIndex()):
                """ Insert the given number of rows into the scene graph below the given
                parent.

                :param row: The row after which the new rows shall be inserted.
                :type  row: int
                :param count: The number of rows to insert.
                :type  count: int
                :param parent_index: The index of the parent item, under which the rows will
                                     be inserted.
                :type parent_index:  QtCore.QModelIndex

                :return: a boolean value. True when the insertion was successful, False otherwise.
                :rtype:  bool
                """

                if parent_index.isValid():
                    self.beginInsertRows(parent_index, row, row + count - 1)

                    # The internal pointer of the parent index returns a scene graph view
                    # model.
                    parent_node = parent_index.internalPointer()

                    domain_scene_model = domain_scene.SceneModel()
                    guidomain_scene.SceneGraphViewModel(
                        row=row,
                        domain_object=domain_scene_model,
                        parent=parent_node
                    )

                    self.endInsertRows()

                    self.layoutChanged.emit()
                    self.scene_added.emit(domain_scene_model)
                else:
                    return False
```

Listing 62: The method `insertRows` is being added to the scene graph controller's methods.

Removing a row is very similar. Analogous, the first thing to do is calling `beginRemoveRows` by providing the index of the parent item, the current row and the last row of insertion (which is the current row plus the count minus one). The actual removal of the node is then done by getting that node from its parent by using the provided row and the parent's column. The node is then removed by setting its parent to `None`. Qt's data model will therefore then remove the node. The transaction is then being ended by calling `endRemoveRows`. Finally, again the view widget is being told to redraw itself by emitting the `layoutChanged` signal and the `scene_removed` signal is emitting the linked domain model to inform other components (subscribers) about the removal. This can be seen in listing 63.

```
<app-scenegraph-controller-methods>+=
        def removeRows(self, row, count, parent_index=QtCore.QModelIndex()):
            """Remove the given number of rows, lying below the given parent, from the
            scene graph.

            :param row: The row which marks the beginning of the removal.
            :type  row: int
            :param count: The number of rows to remove starting from the given row.
            :type  count: int
            :param parent_index: The index of the parent item, under which the rows will
                                 be removed.
            :type parent_index:  QtCore.QModelIndex

            :return: a boolean value. True when the removal was successful, False otherwise.
            :rtype:  bool
            """

            if parent_index.isValid():
                self.beginRemoveRows(parent_index, row, row + count - 1)

                node_index = parent_index.child(row, parent_index.column())
                # The internal pointer of the node index returns a scene graph view
                # model.
                node = node_index.internalPointer()
                node.setParent(None)

                self.endRemoveRows()

                self.layoutChanged.emit()
                self.scene_removed.emit(node.domain_object)
            else:
                return False
```

Listing 63: The method `removeRows` is being added to the scene graph controller's methods.

The newly added functionality now allows the adding and removal of scenes within the scene graph, which can be seen in figure 5.5. The scenes are added to (or removed from respectively) the graphical user interface as well as the data structure.

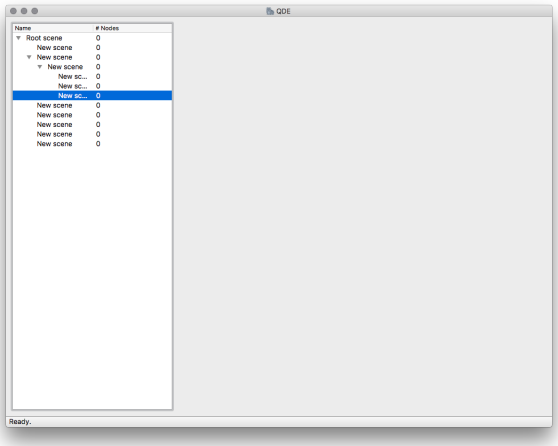

Figure 5.4: The QDE editor application showing the scene graph widget, containing multiple scenes.

So far the application (or rather the scene graph) seems to be working as intended. But how do we ensure, that it really does? Without a doubt, unit and integration tests are one of the best instruments to ensure functionality of code. As stated before, in section 4.1.3, it was an intention of this project to develop the application test driven. Due to required amount of work for developing test driven, it was abstained from this intention and regular unit tests are written instead, which can be found in appendix 8.1.

But nevertheless, it would be very handy to have at least some idea what the code is doing at certain places and at certain times. One of the simplest approaches to achieve this, is a verbose output at various places of the application, which may be as simple as using Python's `print` function. Using the `print` function may allow printing something immediately, but it lacks of flexibility and demands each time a bit of effort to format the output accordingly (e.g. adding the class and the function name and so on). Python's logging facility provides much more functionality while being able to keep things simple as well — if needed. The usage of the logging facility to log messages throughout the application may later even be used to implement a widget which outputs those messages. So logging using Python's logging facility will be implemented and applied for being able to have feedback when needed.

5.3.6 DONE Logging

As logging is a very central and basic functionality, the module is placed in the `foundation` layer.

Logging shall be provided on a class-basis, meaning that each class (which wants to log something) needs to instantiate a logger and use a corresponding handler.

Python's logging module uses the basic configuration by default, calling the `basicConfig` whenever something is logged for the first time. This creates a stream handler with a basic formatter. However, the logging facility may extensively be configured[25].

For example, the logging may be configured by using the "Configuration API", which offers configuring the logging facility by using a dictionary. A dictionary may very easily be created by using a JSON file.

As mentioned before, logging is a central aspect of the application. Therefore it is the task of the main application to set up the logging facility which may then be used by other classes through a decorator.

The main application shall therefore set up the logging facility as follows:

- Use either an external logging configuration or the default logging configuration.
- When using an external logging configuration
  - The location of the external logging configuration may be set by the environment variable `QDE_LOG_CFG`.
  - Is no such environment variable set, the configuration file is assumed to be named `logging.json` and to reside in the application's main directory.
- When using no external logging configuration, the default logging configuration defined by `basicConfig` is used.
  - Always set a level when using no external logging configuration, the default being `INFO`.

This leads to two parameters when setting up the logging configuration: The default path of the external logging configuration and the default logging level when using no external logging configuration. The implementation of setting up the logging facility can be seen in listing 64.

[25] `https://docs.python.org/3/library/logging.html`

```
<app-application-methods>+=
       def setup_logging(self,
                         default_path='logging.json',
                         default_level=logging.INFO):
           """Setup logging configuration"""

           env_key  = 'QDE_LOG_CFG'
           env_path = os.getenv(env_key, None)
           path     = env_path or default_path

           if os.path.exists(path):
               with open(path, 'rt') as f:
                   config = json.load(f)
                   logging.config.dictConfig(config)
           else:
               logging.basicConfig(level=default_level)
```

Listing 64: The `setup_logging` method is being added to the main application class `Application`.

```
<app-application-constructor>+=
           self.setup_logging()
```

Listing 65: The call of the `setup_logging` method is being added to the main application's constructor.

```
<app-application-system-imports>+=
        import logging
        import logging.config
        import os
        import json
```

Listing 66: The `logging` module is added to the application module's system imports.

For not having only basic logging available, a logging configuration is defined and provided by listing 67. The logging configuration provides three handlers: a console handler, which logs debug messages to STDOUT, a info file handler, which logs informational messages to a file named `info.log`, and a error file handler, which logs errors to a file named `error.log`. The default level is set to debug and all handlers are used.

```
{
    "version": 1,
    "disable_existing_loggers": false,
    "formatters": {
        "simple": {
            "format": "%(asctime)s - %(levelname)-7s - %(name)s.%(funcName)s::%(lineno)s: %(message)s"
        }
    },

    "handlers": {
        "console": {
            "class": "logging.StreamHandler",
            "level": "DEBUG",
            "formatter": "simple",
            "stream": "ext://sys.stdout"
        },

        "info_file_handler": {
            "class": "logging.handlers.RotatingFileHandler",
            "level": "INFO",
            "formatter": "simple",
            "filename": "info.log",
            "maxBytes": 10485760,
            "backupCount": 20,
            "encoding": "utf8"
        },

        "error_file_handler": {
            "class": "logging.handlers.RotatingFileHandler",
            "level": "ERROR",
            "formatter": "simple",
            "filename": "errors.log",
            "maxBytes": 10485760,
            "backupCount": 20,
            "encoding": "utf8"
        }
    },

    "root": {
        "level": "DEBUG",
        "handlers": ["console", "info_file_handler", "error_file_handler"],
        "propagate": "no"
    }
}
```

Listing 67: The configuration of the logging facility in JSON format.

The logging configuration as shown in listing 67 allows to get an arbitrarily named logger which uses that configuration.

As stated before, logging shall be provided on a class basis. This has the consequence, that each class has to instantiate a logging instance. To prevent the repetition of the same code fragment over and over, Python's decorator pattern is used[26].

The decorator will be implemented as a method named `with_logger` in the `common` module. All, that this method does is to set the logger name to the name of the module it is in combined with his own name. It then attaches a property named `logger` to the class that calls the method. The method has therefore the following functionality.

- Provide a name based on the current module and class.

```
logger_name = "{module_name}.{class_name}".format(
    module_name=cls.__module__,
    class_name=cls.__name__
)
```

Listing 68: Setting of the name based on the current module and class name.

[26] `https://www.python.org/dev/peps/pep-0318/`

- Provide an easy to use interface for logging.

```
    cls.logger = logging.getLogger(logger_name)

    return cls
```

Listing 69: The logger is being attached to the class itself.

This definition of the functionality allows the actual implementation of the logging facility which follows in listing 70.

```
    # -*- coding: utf-8 -*-

    """Module holding common helper methods."""

    # System imports
    import logging


    # Project imports



    def with_logger(cls):
        """Add a logger instance (using a stream handler) to the given class.

        :param cls: the class which the logger shall be added to.
        :type  cls: a class of type cls.

        :return: the class with the logger instance added.
        :rtype:  a class of type cls.
        """

            logger_name = "{module_name}.{class_name}".format(
                module_name=cls.__module__,
                class_name=cls.__name__
            )
            cls.logger = logging.getLogger(logger_name)

            return cls
```

Listing 70: Implementation of the logging facility as a method inside the `common` module.

The implementation of the `with_logger` method allows the usage of the logging facility as a decorator as shown in the example in listing 71.

```
    # Project imports
    from qde.editor.foundation import common

    @common.with_logger
    def SomeClass(object):
        """This class provides literally nothing and is used only to demonstrate the
        usage of the logging decorator."""

        def some_method():
            """This method does literally nothing and is used only to demonstrate the
            usage of the logging decorator."""

            self.logger.debug(("I am some logging entry used for"
                               "demonstration purposes only."))
```

Listing 71: The class `SomeClass` gets annotated by the `with_logger` decorator from the `common` module. The whole class is then able to use the `logger` property as can be seen in method `some_method`.

This brings us back to original intention: log whenever a scene is added or removed in the scene graph view. To implement the logging three steps are necessary. First, the `common` module needs to be imported.

```
<gui-scene-project-imports>+=
        from qde.editor.foundation import common
```

Listing 72: The `common` module is added to the project imports of the `scene` module residing in the `gui` layer.

Second, the class needs the `with_logger` decorator.

```
<gui-scene-graph-class-decorators>+=
        @common.with_logger
```

Listing 73: The `with_logger` decorator is added to the scene graph view class's decorators, <31>.

And third, the actual logging needs to be added to the corresponding methods.

```
<gui-scene-graph-slots-on-tree-item-added>+=
        self.logger.debug("A new scene graph item was added.")
```

Listing 74: A debug message is being logged, whenever a new scene is added to the scene graph within the scene graph view.

```
<gui-scene-graph-slots-on-tree-item-removed>+=
        self.logger.debug((
            "The scene graph item at row {row} "
            "and column {column} was removed."
        ).format(
            row=selected_item.row(),
            column=selected_item.column()
        ))
```

Listing 75: A debug message is being logged, whenever an existing scene is removed from the scene graph within the scene graph view.

Whenever the `a` or the `delete` key is being pressed now, when the scene graph view is focused, the corresponding log messages appear in the standard output, hence the console. This behavior can be seen in figure 5.5.

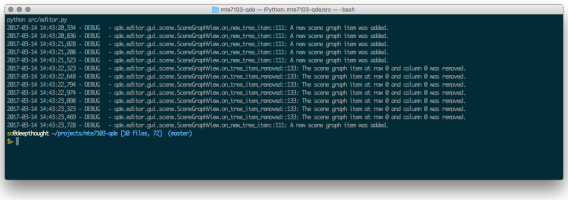

Figure 5.5: The console output when the `a` and the `delete` keys are pressed multiple times with the scene graph view being the active widget.

Now, having the scene graph component as well as an interface to log messages throughout the application implemented, the next component may be approached. A very interesting aspect to face would be the rendering. But for being able to render something, there actually needs to exist something to render: nodes. Nodes are being represented within the node graph. So this is a good point to begin with the implementation of the node graph.

5.3.7 TODO Node graph

The functionality of the node graph is, as its name states, to represent a data structure composed of nodes and edges. Each scene from the scene graph is represented within the node graph as such a data structure.

The nodes are the building blocks of a real time animation. They represent different aspects, such as scenes themselves, time line clips, models, cameras, lights, materials, generic operators and effects. These aspects are only examples (coming from [1, p. 30 and 31]) as the node structure must be expandable for allowing the addition of new nodes.

The implementation of the scene graph component was relatively straightforward partly due to its structure and partly due to the used data model and representation. The node graph component however, seems to be a bit more complex.

To get a first overview and to manage its complexity, it might be good to identify its sub components first before implementing them. When thinking about the implementation of the node graph, one may identify the following sub components:

- Nodes

  Domain model Holds data of a node, like its definition, its inputs and so on.

  Definitions Represents a domain model as JSON data structure.

  Controller Handles the loading of node definitions as well as the creation of node instances.

  View model Represents a node within the graphical user interface.

- Scenes

  Domain model Holds the data of a scene, e.g. its nodes.

  Controller Handles scene related actions, like when a node is added to a scene, when the scene was changed or when a node within a scene was selected.

  View model Defines the graphical representation of scene which can be represented by the corresponding view. Basically the scene view model is a canvas consisting of nodes.

  View Represents scenes in terms of scene view models within the graphical user interface.

Nodes

One of the most basic sub components are the definitions of nodes. But what are those definitions actually? What do they actually define? There is not only one answer to this question, it is simply a matter of how the implementation is being done and therefore a set of decisions.

The whole (rendering) system shall not be bound to only one representation of nodes, e.g. triangle based meshes. Instead it shall let the user decide, what representation is the most fitting for the goal he wants to achieve.

At this point the system shall be able to theoretically support multiple kinds of node representations: Images, triangle based meshes and solid modeling through function modeling (using signed distance functions for modeling implicit surfaces). Whereas triangle based meshes may either be loaded from externally defined files (e.g. in the Filmbox (FBX), the Alembic (ABC) or the Object file format (OBJ)) or directly be generated using procedural mesh generation.

When implementing the nodes, keep in mind that the nodes are part of a graph, hence the name node graph, and are therefore typically connected by edges. This means that the graph gets evaluated recursively by its nodes. However, the goal is to have OpenGL shading language (GLSL) code at the end, independent of the node types.

From this point of view it is theoretically possible to let the user define shader code directly within a node (definition) and to simply evaluate this code, which adds a lot of (creative) freedom. The problem with this approach is though, that image and triangle based mesh nodes are not fully implementable by

using shader code only. Instead they have specific requirements, which are only perform-able on the CPU instead of the GPU (e.g. allocating buffer objects).

When thinking of nodes used for solid modeling however, it may appear, that they may be evaluated directly, without the need for pre-processing, as they are fully implementable using shader code only. This is kind of misleading however as each node has its own definition which has to be added to shader and this definition is then used in a mapping function to compose the scene. This would mean to add a definition of a node over and over again, when spawning multiple instances of the same node type, which results in overhead bloating the shader. It is therefore necessary to pre-process solid modeling nodes too, exactly as triangle mesh based and image nodes, for being able to use multiple instances of the same node type within a scene while having the definition added only once.

All of these thoughts sum up in one central question for the implementation: Shall objects be predefined within the code (and therefore only nodes accepted whose type and sub type match those of predefined nodes) or shall all objects be defined externally using files?

This is a question which is not that easy to answer. Both methods have their advantages and disadvantages. Pre-defining nodes within the code minimizes unexpected behavior of the application. Only known and well-defined nodes are processed.

But what if someone would like to have a new node type which is not yet defined? The node type has to be implemented first. As Python is used for the editor application, this is not really a problem as the code is interpreted each time and is therefore not being compiled. Nevertheless such changes follow a certain process, such as making the actual changes, reviewing and checking-in the code and so on, which the user normally does not want to be bothered with. Furthermore, when thinking about the player application, the problem of the necessity to recompile the code is definitively given. The player will be implemented in C, as there is the need for performance, which Python may not fulfill satisfactorily.

Considering these aspects, the external definition of nodes is chosen. This may result in nodes which cannot be evaluated or which have unwanted effects. As it is (most likely) in the users best interest to create (for his taste) appealing real time animations, it can be assumed, that the user will try avoiding to create such nodes or quickly correct faulty nodes or simply does not use such nodes.

Now, having chosen how to implement nodes, let us define what a node actually is. A node is essentially composed by the following elements:

ID A global unique identifier (UUID [27])

Name The name of the node, e.g. "Cube".

Inputs A list of the nodes inputs. The inputs may either be parameters (which are atomic types such as float values or text input) or references to other nodes.

Outputs A list of the nodes outputs. The outputs may also either be parameters or references to other nodes.

Parts Defines parts that may be evaluated when evaluating the node. Contains code which can be interpreted directly.

Nodes The children a node has (child nodes). These entries are references to other nodes only.

Connections A list of connections between the input and output of the node. Each input and output is composed by two parts: A reference to another node and a reference to that nodes output or input respectively. Input: Reference to a node X + Reference to *output* A of node X. Output: Reference to a node Y + Reference to *input* B of node Y.

As can be derived from the above thoughts, each of the mentioned node representations need some effort in terms of allocating buffers or render targets before they may be used for rendering a frame. They may as well want to free or release some of their made allocations when not being used anymore.

In other words, every node will be pre-processed before being processed and post-processed after being processed.

[27] `https://docs.python.org/3/library/uuid.html`

To keep the learning curve at a decent level when using the editor application, it is important to provide the user with predefined nodes to choose from — independent from their type — otherwise users could get easily frustrated at the beginning. The following nodes are envisaged:

- Solid modeling objects
  - Sphere
  - Cube
  - Plane
  - ...
- Solid modeling operations
  - Transformation
  - Scaling
  - Rotation
  - Union
  - Differentiation
  - ...
- Post-processing effects
  - Blur
  - Glare
  - ...
- Images

To get the node graph implementation started, a sample node definition is implemented as well as its defining class.

Node definitions

Node definitions are implemented in the JSON[28] format and are placed in the `data/nodes/` sub-directory, seen from the main directory.

ParameterID defines a globally unique ID and is used for inputs and outputs. It may however come from the ID of an actual node, then the input is another node. If it does not come from another node, the input is a basic type.

An input defines a parameter and may be referenced as output as well.

Every input and output receives a ParameterInstanceID, which is globally unique as well. Such an ID is used for making connections between nodes and/or parts of nodes. Parts = Code?

A node may contain other nodes as well, those are references however.

A few node types are pre-defined however. Those can be selected as input and as output as well. The pre-defined types are:

- Generic
- Float (value)
- Text
- Scene
- Image

[28] `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`

- Dynamic (value)
- Mesh

Connections are actually parts. We need a context. Nodes get processed using this (global) context.

6 Work log

<2017-02-20 Mon> Set up and structure the document initially.

<2017-02-21 Tue> Re-structure the document, add first contents of the implementation. Add first tries to tangle the code.

<2017-02-22 Wed> Provide further content concerning the implementation: Introduce name-spaces/initializers, first steps for a logging facility.

<2017-02-23 Thu> Extend logging facility, provide (unit-) tests. Restructure the documentation.

<2017-02-24 Fri> Adapt document to output LaTeX code as desired, change styling. Begin development of the applications' main routine.

<2017-02-27 Mon> Remove (unit-) tests from main document and put them into appendix instead. Begin explaining literate programming.

<2017-02-28 Tue> Provide a first draft for objectives and limitations. Re-structure the document. Correct LaTeX output.

<2017-03-01 Wed> Remove split files, re-add everything to index, add objectives.

<2017-03-02 Thu> Set up project schedule. Tangle everything instead of doing things manually. Begin changing language to English instead of German. Re-add make targets for cleaning and building the source code.

<2017-03-03 Fri> Keep work log up to date. Revise and finish chapter about name-spaces and the project structure for now.

<2017-03-04 Sat> Finish translating all already written texts from German to English. Describe the main entry point of the application as well as the main application itself.

<2017-03-05 Sun> Finish chapter about the main entry point and the main application for now, start describing the main window and implement its functionality. Keep the work log up to date. Fiddle with references and LaTeX export. Find a bug: main_window needs to be attached to a class, by using the `self` keyword, otherwise the window does not get shown. Introduce new make targets: one to clean Python cache files (*.pyc) and one to run the editor application directly.

<2017-03-06 Mon> Update the work log. Add an image of the editor as well as the project schedule. Add the implementation of the main window's layout. Implement the scene domain model. Move keyPressEvent to its own source block instead of expanding the methods of the main window directly. Add a section about (the architecture's) layers to the principles section. Add Dr. Eric Dubuis as an expert to the involved persons. Introduce the 'verb' macro for having nicer verbatim blocks. Use the given image-width for inline images in org-mode when available.

<2017-03-07 Tue> Expand the layering principles by adding a section about the model-view-controller pattern and introduce view models. Explain and implement the data- and the view model for scene graph items.

<2017-03-08 Wed> Implement the controller for handling the scene graph. Allow the semi-automatic creation of an API documentation by introducing Sphinx. Introduce new make targets for creating the API documentation as RST and as HTML.

<2017-03-10 Fri> Implement the scene graph view as widget and integrate it into the application. Update the work log. Fix typing errors. Start to implement missing methods in the scene graph controller for being able to use the scene graph widget.

<2017-03-13 Mon> Implement the scene view model. Initialize such a model within the scene graph view model. Implement the `headerData` as well as the `data` methods of the scene graph controller. Update the work log. Add an image of the editor's current state. Continue implementation of the scene graph view model.

<2017-03-14 Tue> Continue the implementation of the scene graph view model. Implement logging. Implement logging. Implement logging. Implement logging functionality. Log whenever a node is added or removed from the scene graph view.

<2017-03-15 Wed> Move logging further down in structure. Add connections between scene graph view and controller. Finish implementing the adding and removal of scene graph items. Update the work log.

Next steps: (Re-) Introduce logging. Begin implementing the node graph.

<2017-03-16 Thu> Run sphinx apidoc when creating the HTML documentation. Add an illustration about the state of the editor after finishing the implementation of the scene graph. Change width of the images to be 50% of the text width. Name slots of the scene graph view explicitly to maintain sanity. Re-add logging chapter with a corresponding introduction. Fix display of code listings. Keep work log up to date. Add missing TODO annotations to headings.

Next steps: Continue implementing the node graph.

<2017-03-17 Fri> Change verbatim output to be less intrusive, update to do tags, begin adding references do code fragment definitions, begin implement the node graph. Move chapters into separate org files.

<2017-03-20 Mon> Re-think how to implement node definitions and revise therefore the chapter about the node graph component, fix various typographic errors, expand and change the Makefile, keep the work log up to date.

<2017-03-21 Tue> Re-think how to implement node definitions.

<2017-03-22 Wed> Re-think how to implement node definitions and nodes. Begin adding notes about how to implement nodes.

<2017-03-23 Thu> Expand notes about the node implementation, begin writing the actual node implementation down, keep the work log up to date.

7 TODO Bibliography

Bibliography

[1] S. Osterwalder, *QDE - a visual animation system. Software-Architektur.* Bern University of Applied Sciences, Aug. 5, 2016.

[2] Martin Fowler. (Jul. 19, 2004). Presentation model, martinfowler.com, [Online]. Available: `https://martinfowler.com/eaaDev/PresentationModel.html` (visited on 03/07/2017).

8 TODO Appendix

8.1 Test cases

8.1.1 Test cases

Zunächst wird jedoch der entsprechende Unit-Test definiert. Dieser instanziert die Klasse und stellt sicher, dass sie ordnungsgemäss gestartet werden kann.

Als erster Schritt wird der Header des Test-Modules definiert.

```
# -*- coding: utf-8 -*-

"""Module for testing QDE class."""
```

Listing 76: Header des Test-Modules, «test-app-header».

Dann werden die benötigen Module importiert. Es sind dies das System-Modul *sys* und das Modul *application*, bei welchem es sich um die Applikation selbst handelt. Das System-Modul *sys* wird benötigt um der Applikation ggf. Start-Argumente mitzugeben, also zum Beispiel:

```
python main.py argument1 argument2
```

Listing 77: Aufruf des Main-Modules mit zwei Argumenten, `argument1` und `argument2`.

Der Einfachheit halber werden die Importe in zwei Kategorien unterteilt: Importe von Pyhton-eigenen Modulen und Importe von selbst verfassten Modulen.

```
# System imports
<<test-app-system-imports>>

# Project imports
<<test-app-project-imports>>
```

Listing 78: Definition der Importe für das Modul zum Testen der Applikation.

```
# System imports
import sys
```

Listing 79: Importe von Python-eigenen Modulen im Modul zum Testen der Applikation.

```
# Project imports
from qde.editor.application import application
```

Listing 80: Importe von selbst verfassten Modulen im Modul zum Testen der Applikation.

Somit kann schliesslich getestet werden, ob die Applikation startet, indem diese instanziert wird und die gesetzten Namen geprüft werden.

```
def test_constructor():
    """Test if the QDE application is starting up properly."""
    app = application.QDE(sys.argv)
    assert app.applicationName() == "QDE"
    assert app.applicationDisplayName() == "QDE"
```

Listing 81: Methode zum Testen des Konstruktors der Applikation.

Finally, one can merge the above defined elements to an executable test-module, containing the header, the imports and the test cases (which is in this case only a test case for testing the constructor).

```
<<test-app-header>>

<<test-app-imports>>

<<test-app-test-constructor>>
```

Listing 82: Modul zum Testen der Applikation.

Führt man die Testfälle nun aus, schlagen diese erwartungsgemäss fehl, da die Klasse, und somit die Applikation, als solche noch nicht existiert. Zum jetzigen Zeitpunkt kann noch nicht einmal das Modul importiert werden, da diese noch nicht existiert.

```
python -m pytest qde/editor/application/test_application.py
```

Listing 83: Aufruf zum Testen des Applkations-Modules.

Um sicherzustellen, dass die Protokollierung wie gewünscht funktioniert, wird diese durch die entsprechenden Testfälle abgedeckt.

Der einfachste Testfall ist die Standardkonfiguration, also ein Aufruf ohne Parameter.

```
def test_setup_logging_without_arguments():
    """Test logging of QDE application without arguments."""
    app = application.QDE(sys.argv)
    root_logger = logging.root
    handlers = root_logger.handlers
    assert len(handlers) == 1
    handler = handlers[0]
```

Listing 84: Testfall 1 der Protkollierung der Hauptapplikation: Aufruf ohne Argumente.

Da obige Testfälle das *logging*-Module benötigen, muss das Importieren der Module entsprechend erweitert werden.

```
import logging
```

Listing 85: Erweiterung des Importes von System-Modulen im Modul zum Testen der Applikation.

Und der Testfall muss den Testfällen hinzugefügt werden.

```
<<test-app-test-logging-default>>
```

Listing 86: Hinzufügen des Testfalles 1 zu den bestehenden Testfällen im Modul zum Testen der Applikation.

Auch hierfür werden wiederum zuerst die Testfälle verfasst.

```python
# -*- coding: utf-8 -*-

"""Module for testing common methods class."""

# System imports
import logging

# Project imports
from qde.editor.foundation import common


@common.with_logger
class FooClass(object):
    """Dummy class for testing the logging decorator."""

    def __init__(self):
        """Constructor."""
        pass

def test_with_logger():
    """Test if the @with_logger decorator works correctly."""

    foo_instance = FooClass()
    logger = foo_instance.logger
    name = "qde.editor.foundation.test_common.FooClass"
    assert logger is not None
    assert len(logger.handlers) == 1
    handler = logger.handlers[0]
    assert type(handler) == logging.StreamHandler
    assert logger.propagate == False
    assert logger.name == name
```

Listing 87: Testfälle der Hilfsmethode zur Protokollierung.

```
python -m pytest qde/editor/foundation/test_common.py
```

8.2 Meeting minutes

8.2.1 Meeting minutes

Meeting mintutes 2017-02-23

No.:	01
Date:	2017-02-23 13:00 - 13:30
Place:	Cafeteria, Main building, Berne University of applied sciences, Biel
Involved persons:	Prof. Claude Fuhrer (CF)
	Sven Osterwalder (SO)

Kick-off meeting for the thesis.

1. Presentation and discussion of the current state of work
    - Presentation of the workflow. Emacs and Org-Mode is used to write the documentation as well as the actual code. (SO)
        - This is a very interesting approach. The question remains if the effort of this method does not prevail the method of developing the application and the documentation in parallel. It is important to reach a certain state of the application. Also the report should not exceed around 80 pages. (CF)
            * A decision about the used method is made until the end of this week. (SO)
    - The code will unit-tested using py.test and / or hypothesis. (SO)
    - Presentation of the structure of the documentation. It follows the schematics of the preceding documentations. (SO)

2. Further steps / proceedings
    - The expert of the thesis, Mr. Dubuis, puts mainly emphasis on the documentation. The code of the thesis is respected too, but is clearly not the main aspect. (CF)
    - Mr. Dubuis also puts emphasis on code metrics. Therefore the code needs to be (automatically) tested and a coverage of at least 60 to 70 percent must be reached. (CF)
    - A meeting with Mr. Dubuis shall be scheduled at the end of March or beginning of April 2017. (CF)
    - The administrative aspects as well as the scope should be written until end of March 2017 for being able to present them to Mr. Dubuis. (CF)
    - Mr. Dubuis should be asked if the publicly available access to the whole thesis is enough or if he wishes to receive the particular status right before the meetings. (CF)
    - Regularly meetings will be held, but the frequency is to be defined yet. Further information follows per e-mail. (CF)
    - At the beginning of the studies, a workplace at the Berne University of applied sciences in Biel was offered. Is this possibility still available? (SO)
        - Yes, that possibility is still available and details will be clarified and follow per e-mail. (CF)
3. To do for the next meeting
    a) **DONE** Create GitHub repository for the thesis. (SO)
        i. **DONE** Inform Mr. Fuhrer about the creation of the repository. (SO)
    b) **DONE** Ask Dr. Dubuis by mail how he wants to receive the documentation. (SO)
        i. **TODO** Await answer of Mr. Dubuis (ED)
    c) **DONE** Set up appointments with Dr. Dubuis (CF)
        i. **TODO** Await answer of Mr. Dubuis (ED)
    d) **DONE** Clarify possibility of a workplace at Berne University of applied sciences in Biel. (CF)
        i. A workplace was found at the RISIS laboratory and may be used instantly. (CF)
    e) **DONE** Decide about the method used for developing this thesis. (SO)
        i. After discussions with a colleague the method of literate programming is kept. The documentation containing the literate program will although be attached as appendix as it most likely will exceed 80 pages. Instead the method will be introduced in the report and the report will be endowed with examples from the literate program.
    f) **TODO** Describe procedure and set up a time schedule including milestones. (SO)
4. Scheduling of the next meeting
    - To be defined

9 TODO Glossary

animation An animation is a composition of scenes. Each animation is defined by a time span, meaning it has a defined start- and end-time. As the name indicates, an animation contains animated elements, being properties of nodes (e.g. the position of a node and so on).