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	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

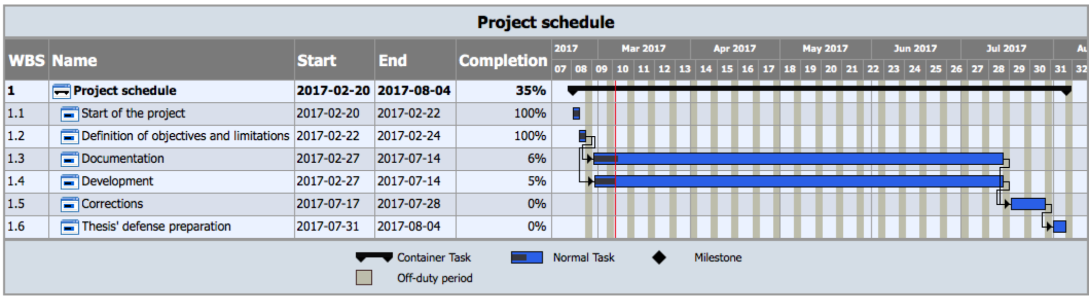

Figure 4.1: The project's schedule.

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.
   - 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

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.

```
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.

```
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.

```
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`.

```
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 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-imports>>

<<app-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-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-system-imports>>

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

Listing 6: «`app-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-class-body>>
```

Listing 7: «`app-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.

```
from PyQt5 import QtGui
from PyQt5 import QtWidgets
```

Listing 8: «`app-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.

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

Listing 9: «`app-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. The applications' constructor looks hence as follows.

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

```python
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")
```

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

5.3.2 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.

```python
#!/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 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.

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.

```python
#!/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>>

    <<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.

```python
# 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.

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

Listing 16: Expansion of the main application's constructor, `«app-constructor»`, 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 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-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.

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

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

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. 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.

```python
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`.

```python
    self.setup_ui()
```

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

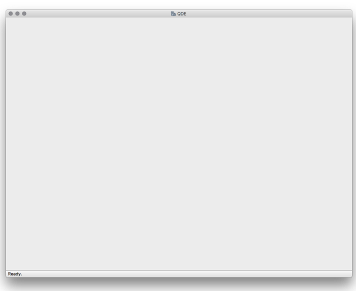

Figure 5.1: The QDE editor application in a very early stage, containing only a grid layout.

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.

```python
    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 23: 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.

TODO 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 24: Scene module inside the `domain` package, holding the `SceneModel` class.

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

Listing 25: 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` [fn:e3eb4d58d8c947d:http://doc.qt.io/qt-5/qabstractitemmodel.html] 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

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

the `QAbstractItemModel` class. This signal needs the current model index as well as the first and last position as parameters. The 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.

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

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

# System imports
from PyQt5 import Qt
from PyQt5 import QtCore
<<guidomain-scene-system-imports>>

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


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.
    """

    <<guidomain-scene-viewmodel-signals>>

    <<guidomain-scene-viewmodel-constructor>>

    <<guidomain-scene-viewmodel-methods>>

    <<guidomain-scene-viewmodel-slots>>
```

Listing 26: Scene module inside the `gui_domain` package, holding currently only the `SceneGraphViewModel` class.

```python
# .. 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 27: 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.

```python
#!/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, parent=None):
        """Constructor.

        :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,
            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 28: 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.”[15]

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

5.3.4 Logging

Da es sehr nützlich ist, den Zustand einer Applikation jederzeit in Form von gezielten Ausgaben nachvollziehen zu können, bietet es sich an als ersten Schritt ein Modul zur Protokollierung zu implementieren.
Protokollierung ist ein sehr zentrales Element, daher wird das Modul im Namespace =foundation= erstellt.

Die (Datei-) Struktur zur Erstellung und Benennung der Module erfolgt ab diesem Zeitpunkt nach dem Schichten-Modell gemäss \cite[S. 40]{osterwalder_qde_2016}.

```
#+ATTR_LaTeX: :options fontsize=\footnotesize,linenos,bgcolor=bashcodebg
#+CAPTION:    Erstellung und Initialisierung des =foundation=-Namespaces.
#+NAME:       fig:editor-foundation-namespace
#+BEGIN_SRC python :tangle ../src/qde/editor/foundation/__init__.py :noweb tangle :comments link :mkd
#+END_SRC
```

Die Protokollierung auf Klassen-Basis stattfinden. Vorerst sollen Protokollierungen als Stream ausgegeben werden. Pro Klasse muss also eine =logging=-Instanz instanziert und mit dem entsprechenden Handler ausgestattet werden. Um den Programmcode nicht unnötig wiederholen zu müssen, bietet sich hierfür das Decorator-Pattern von Python an[fn:9:https://www.python.org/dev/peps/pep-0318/].

Die Klasse zur Protokollierung soll also Folgendes tun:

```
- Einen Logger-Namen auf Basis des aktuellen Moduls und der aktuellen Klasse setzen
  #+NAME: logger-name
  #+ATTR_LaTeX: :options fontsize=\footnotesize,linenos,bgcolor=bashcodebg
  #+CAPTION:    Setzen des Logger-Names auf Basis des aktuellen Modules und Klasse.
  #+BEGIN_SRC python
  logger_name = "%s.%s" % (cls.__module__, cls.__name__)
  #+END_SRC

  #+RESULTS: logger-name

- Einen Stream-Handler nutzen
  #+NAME: logger-stream-handler
  #+ATTR_LaTeX: :options fontsize=\footnotesize,linenos,bgcolor=bashcodebg
  #+CAPTION:    Initialisieren eines Stream-Handlers.
  #+BEGIN_SRC python
    stream_handler = logging.StreamHandler()
  #+END_SRC

    #+RESULTS: logger-stream-handler

- Die Stufe der Protokollierung abhängig von der aktuellen Konfiguration setzen
  #+NAME: logger-set-level
  #+ATTR_LaTeX: :options fontsize=\footnotesize,linenos,bgcolor=bashcodebg
  #+CAPTION:    Setzen des =DEBUG= Log-Levels.
  #+BEGIN_SRC python
    # TODO: Do this according to config.
    stream_handler.setLevel(logging.DEBUG)
  #+END_SRC

    #+RESULTS: logger-set-level
```

```
- Protokoll-Einträge ansprechend formatieren
  #+NAME: logger-set-formatter
  #+ATTR_LaTeX: :options fontsize=\footnotesize,linenos,bgcolor=bashcodebg
  #+CAPTION:    Anpassung der Ausgabe von Protokoll-Meldungen.
  #+BEGIN_SRC python
    # TODO: Set up formatter in debug mode only
    formatter = logging.Formatter("%(asctime)s - %(levelname)-7s - %(name)s.%(funcName)s::%(lineno)s:
    stream_handler.setFormatter(formatter)
  #+END_SRC

    #+RESULTS: logger-set-formatter

- Eine einfache Schnittstelle zur Protokollierung bieten
  #+NAME: logger-return-logger
  #+ATTR_LaTeX: :options fontsize=\footnotesize,linenos,bgcolor=bashcodebg
  #+CAPTION:    Nutzung des erstellten Stream-Handlers und Rückgabe der Klasse.
  #+BEGIN_SRC python
    cls.logger = logging.getLogger(logger_name)
    cls.logger.propagate = False
    cls.logger.addHandler(stream_handler)

    return cls
  #+END_SRC

    #+RESULTS: logger-return-logger

Nun kann die eigentliche Funktionalität implementiert werden.

#+ATTR_LaTeX: :options fontsize=\footnotesize,linenos,bgcolor=bashcodebg
#+CAPTION:    Das =common=-Modul und eine Methode zur Protokollierung in Klassen.
#+NAME:       fig:editor-common-logging
#+BEGIN_SRC python :tangle ../src/qde/editor/foundation/common.py :noweb tangle :comments link :mkdir
# -*- coding: utf-8 -*-

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

# System imports
import logging


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

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

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

    <<logger-name>>
    <<logger-stream-handler>>
    <<logger-set-level>>
    <<logger-set-formatter>>
    <<logger-return-logger>>
#+END_SRC
```

#+RESULTS: fig:editor-common-logging

Der Decorator kann nun direkt auf die Klasse der QDE-Applikation angewendet werden.

Damit die Protokollierung jedoch nicht nur via STDOUT in der Konsole statt findet, muss diese entsprechend konfiguriert werden. Das /logging/-Modul von Python bietet hierzu vielfältige Möglichkeiten.[fn:10:https://docs.python.org/3/library/logging.html] So kann die Protokollierung mittels der ‘‘Configuration API’’ konfiguriert werden. Hier bietet sich die Konfiguration via Dictionary an. Ein Dictionary kann zum Beispiel sehr einfach aus einer JSON-Datei generiert werden.

Die Haupt-Applikation soll die Protokollierung folgendermassen vornehmen:
- Die Konfiguration erfolgt entweder via externer JSON-Datei oder verwendet die Standardkonfiguration, welche von Python mittels =basicConfig= vorgegeben wird.
- Als Name für die JSON-Datei wird =logging.json= angenommen.
- Ist in den Umgebungsvariablen des Betriebssystems die Variable /LOG_CFG/ gesetzt, wird diese als Pfad für die JSON-Datei angenommen. Ansonsten wird angenommen, dass sich die Datei =logging.json= im Hauptverzeichnis befindet.
- Existiert die JSON-Konfigurationsdatei nicht, wird auf die Standardkonfiguration zurückgegeriffen.
- Die Protokollierung verwendet immer eine Protokollierungsstufe (Log-Level) zum Filtern der verschiedenen Protokollnachrichten.

Die Haupt-Applikation nimmt also die Parameter =Pfad=, =Protokollierungsstufe= sowie =Umgebungsvariable= entgegen.

Nun kann die eigentliche Umsetzung zur Konfiguration der Protokollierung umgesetzt und der Klasse hinzugefügt werden.

#+NAME: app-setup-logging
#+ATTR_LaTeX: :options fontsize=\footnotesize,linenos,bgcolor=bashcodebg
#+CAPTION: Methode zum Initialisieren der Protokollierung der Applikation.
#+BEGIN_SRC python

```python
def setup_logging(self,
    default_path='logging.json',
    default_level=logging.INFO,
    env_key='LOG_CFG'
):
    """Setup logging configuration"""

    path = default_path
    value = os.getenv(env_key, None)

    if value:
        path = value

    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)
#+END_SRC
```

6 Working 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 Mr. Eric Dubuis as an expert to the involved persons. Introduce the command 'myverb' for having nicer verbatim blocks. Use the given image-width for inline images in org-mode when available.

7 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 Appendix

8.1 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 29: 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 30: 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 31: Definition der Importe für das Modul zum Testen der Applikation.

```
# System imports
import sys
```

Listing 32: Importe von Python-eigenen Modulen im Modul zum Testen der Applikation.

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

Listing 33: 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 34: 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 35: 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 36: 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 37: 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 38: 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 39: 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 40: Testfälle der Hilfsmethode zur Protokollierung.

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

8.2 Meeting minutes

8.2.1 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.

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)

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)

To do for the next meeting

1. **DONE** Create GitHub repository for the thesis. (SO)
   a) **DONE** Inform Mr. Fuhrer about the creation of the repository. (SO)
2. **DONE** Ask Mr. Dubuis by mail how he wants to receive the documentation. (SO)
   a) **TODO** Await answer of Mr. Dubuis (ED)
3. **DONE** Set up appointments with Mr. Dubuis (CF)
   a) **TODO** Await answer of Mr. Dubuis (ED)
4. **DONE** Clarify possibility of a workplace at Berne University of applied sciences in Biel. (CF)
   a) A workplace was found at the RISIS laboratory and may be used instantly. (CF)
5. **DONE** Decide about the method used for developing this thesis. (SO)
   a) 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.
6. **TODO** Describe procedure and set up a time schedule including milestones. (SO)

Scheduling of the next meeting

- To be defined

9 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).