BFH Logo

LOGO

QDE — A visual animation system.
MTE7103

Master-Thesis

Major:	Computer science
Author:	Sven Osterwalder[1]
Advisor:	Prof. Claude Fuhrer[2]
Expert:	Dr. Eric Dubuis[3]
Date:	2017-05-05 14:34:42
Version:	ed0b7ea90c914b515e083f339633ee3b2ab3207d

---

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

Versions

Revision	Date	Author(s)	Description
ed0b7ea90c914b515e083f339633ee3b2ab3207d	2017-05-05 14:34:42	SO	foo

Todo list

Abstract

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Contents

1 Introduction

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

1.1 Purpose and situation

1.1.1 Motivation

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

1.1.2 Objectives and limitations

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

1.1.3 Preliminary activities

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

1.2 Related works

Preliminary to this thesis two project works were done: "Volume ray casting — basics & principles" [1], which describes the basics and principles of sphere tracing, a special form of ray tracing, and "QDE — a visual animation system, architecture" [2], which established the ideas and notions of an editor and a player component as well as the basis for a possible software architecture for these components. The latter project work is presented in detail in the chapter about the procedure, the former project work is presented in the chapter about the implementation.

1.3 Document structure

This document is divided into N chapters, the first being this introduction. The second chapter on *administrative aspects* shows the planning of the project, including the involved persons, deliverables and the phases and milestones.

The administrative aspects are followed by a chapter on the *procedure*. The purpose of that chapter is to show the procedure concerning the execution of this thesis. It introduces a concept called literate programming, which builds the foundation for this thesis. Furthermore it establishes a framework for the actual implementation, which is heavily based on the previous project work, "QDE — a visual animation system, architecture" [2] and also includes standards and principles.

The following chapter on the *implementation* shows how the implementation of the editor and the player component as well as how the rendering is done using a special form of ray tracing as described in "Volume ray casting — basics & principles" [1]. As the editor component defines the whole data structure it builds the basis of the thesis and can be seen as main part of the thesis. The player component re-uses concepts established within the editor.

Given that literate programming is very complete and elaborated, as components being developed using this procedure are completely derived from the documentation, the actual implementation is found in the appendix as otherwise this thesis would be simply too extensive.

The last chapter is *discussion and conclusion* and discusses the procedure as well as the implementation. Some further work on the editor and the player components is proposed as well.

After the regular content follows the *appendix*, containing the requirements for building the before mentioned components, the actual source code in form of literal programming as well as test cases for the components.

2 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]	
Advisor	Prof. Claude Fuhrer[2]	*Supervises the student doing the thesis*
Expert	Dr. Eric Dubuis[3]	*Provides expertise concerning the thesis's subject, monitors and grades the thesis*

Table 2.1: List of the involved persons.

2.2 Deliverables

- **Report**
  Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

- **Implementation**
  Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

2.3 Organization of work

2.3.1 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 «Meeting minutes».

---

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

2.3.2 Phases of the project and milestones

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

Table 2.2: Phases of the project.

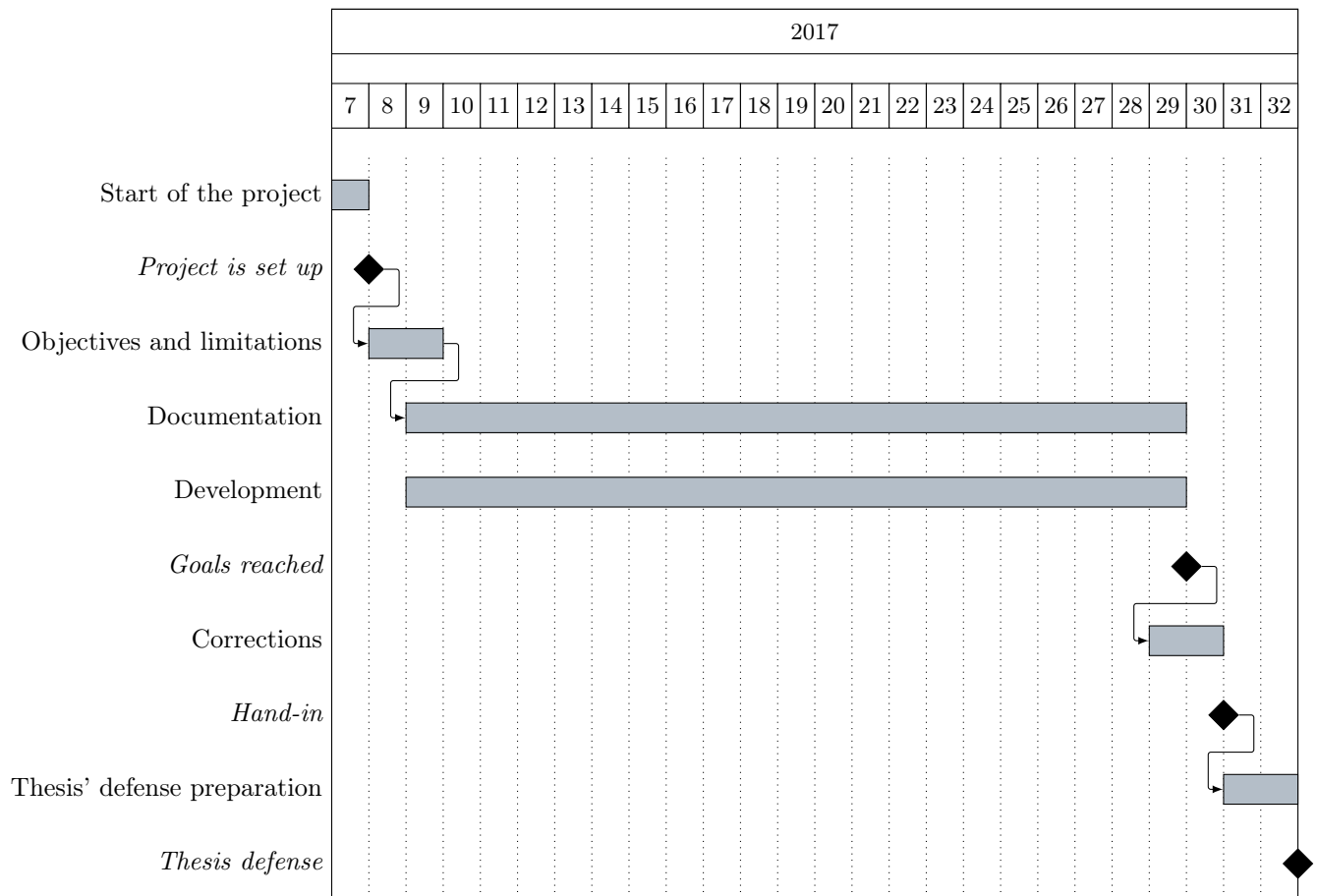| Phase	Description	End of week / 2017
Project structure is set up		8
Mandatory project goals are reached		30
Hand-in of the thesis		31
Defense of the thesis		32

Table 2.3: Milestones of the project.

Figure 2.1: Schedule of the project by calendar weeks, including milestones.

3 Procedure

3.0.1 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.

> 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.

> Insert reference/link to test cases here.

3.1 Standards and principles

3.1.1 Requirements

The requirements are defined by the preceding project work, "QDE — a visual animation system, software architecture" [2, p. 8 ff.], and are still valid.

For the editor application however, Python is used as a programming language. This decision is made as the author of the thesis has several years of experience concerning Python and as the performance of the editor is not a critical factor. By performance all aspects are concerned, e.g. the evaluation of the node graph or rendering itself.

As Python provides no direct bindings to Qt, an additional library is needed, which provides those bindings. Currently there exist two Python bindings for Qt: PySide and PyQt. As Qt version 5 is used, the bindings need to provide access to version 5 too. Currently this is only achieved by PyQt5 in a stable and complete way. PySide2 supports Qt version 5 too, is although under heavy development and far from being complete and stable.

Therefore PyQt5 is an additional requirement.

3.1.2 Code

- 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: verb(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 verb(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 [2, 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 ⎡Link to com
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.

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 verb(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 verb(domain) layer) directly, instead view models are used [3].

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 [fn:20:http://doc.qt.io/qt-5/model-view-programming.html]. 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 [2, p. 38].

> Describe the exact process of communication between ViewModel, Controller and Model.

To avoid coupling and therefore dependencies, signals and slots[fn:16:http://doc.qt.io/qt-5/signalsandslots.html] are used in terms of the observer pattern to allow inter-object and inter-layer communication.

4 Implementation

4.1 Editor

4.2 Player

4.3 Rendering

Bibliography

[1] S. Osterwalder, *Volume ray casting - basics & principles.* Bern University of Applied Sciences, Feb. 14, 2016.

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

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

List of Figures

List of Tables

Listings

5 Appendix

5.1 Implementation

To start the implementation of a project, it is necessary to first think about the goal that one wants to reach and about some basic structures and guidelines which lead to the fulfillment of that goal.

The main goal is to have a visual animation system, which allows the creation and rendering of visually appealing scenes, using a graphical user interface for creation, and a ray tracing based algorithm for rendering.

The thoughts to reach this goal were already developed in chapter 3, "Procedure", and will therefore not be repeated again.

As stated in chapter 3, literate programming is used to implement the components. To maintain readability only relevant code fragments are shown in place. The whole code fragments, which are needed for tangling, are found at section 5.3.

First, the implementation of the editor component is described, as it is the basis for the whole project and also contains many concepts, that are re-used by the player component. Before starting with the implementation it is necessary to define requirements and some kind of framework for the implementation.

5.1.1 Requirements

At the current point of time, the requirements for running the components are the following:

- A Unix derivative as operating system (Linux, macOS).
- Python [1] version 3.5.x or above
- PyQt5 [2] version 5.7 or above

> Add more requirements? E.g. OpenGL?

5.1.2 Name spaces and project structure

To give the whole project a structure and for being able to stick to the thoughts established in chapter 3, it may be wise to structure the project in analogous way as defined in chapter 3.

Therefore 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, it is done by parts of this documentation which are tangled but not exported.

When dealing with directories and files, Python uses the term *package* for (sub-) directories and *module* for files within directories.[3]

---

[1] http://www.python.org
[2] https://riverbankcomputing.com/software/pyqt/intro
[3] https://docs.python.org/3/reference/import.html#packages

To prevent having multiple modules having the same name, name spaces are used.[4] The main name space shall be analogous to the project's 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.

5.1.3 Coding style

To stay consistent throughout the implementation of components, a coding style is applied which is defined as follows.

- Classes use camel case, e.g. `class SomeClassName`.

- Folders respectively name spaces use only small letters, e.g. *foo.bar.baz*.

- Methods are all small caps and use underscores as spaces, e.g. `some_method_name`.

- Signals are methods, which are prefixed by the word "do", e.g. `do_something`.

- Slots are methods, which are prefixed by the word "on", e.g. `on_something`.

- Importing is done by the `from Foo import Bar` syntax, whereas `Foo` is a module and `Bar` is either a module, a class or a method.

Importing of modules

As mentioned at subsection 5.1.1, Python is used. Python has "batteries included", which means that it offers a lot of functionality through various modules, which have to be imported first before using them. The same applies of course for self written modules.

Python offers multiple possibilities concerning imports, for details see `https://docs.python.org/3/tutorial/modules.html`.

Is direct url reference ok or does this need to be citation?

However, PEP number 8 recommends to either import modules directly or to import the needed functionality directly. [5] As defined by the coding style, subsection 5.1.3, imports are done by the `from Foo import Bar` syntax.

The imported modules are always split up: first the system modules are imported, modules which are provided by Python itself or by external libraries, then project-related modules are imported.

Framework for implementation

For also staying consistent when implementing classes and methods, it make sense to define a rough framework for implementation, which is as follows:

- Define necessary signals.

- 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.

---

[4]https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces
[5]`https://www.python.org/dev/peps/pep-0020/`

- Implement the remaining functionality in terms of methods and slots.

Now, having defined the requirements, a project structure, a coding style and a framework for the actual implementation, the implementation of the editor may begin.

5.1.4 Editor

Before diving right into the implementation of the editor, it may be good to reconsider what shall actually be implemented, therefore what the main functionality of the editor is and what its components are.

The quintessence of the editor application is to output a structure, be it in the JSON format or even in bytecode, which defines an animation.

An animation is simply a composition of scenes which run in a sequential order within a time span. A scene is then a composition of nodes, which are at the end of their evaluation nothing else as shader specific code which gets executed on the GPU.

As this definition is rather abstract, it may be easier to define what shall be achieved in terms of content and then work towards this definition.

A very basic definition of what shall be achieved is the following.

It shall be possible to create an animated scene using the editor application. The scene shall be composed of two objects, a sphere and a cube. Additionally it shall have a camera as well as a point light.

The camera shall be placed 5 units in height and 10 units in front of the center of the scene. The cube shall be placed in the middle of the scene, the sphere shall have an offset of 5 units to the right and 2 units in depth. The point light shall be placed 10 units above the center.

Both objects shall have different materials: the cube shall have a dull surface of any color whereas the sphere shall have a glossy surface of any color.

There shall be an animation of ten seconds duration. During this animation the sphere shall move towards the cube and they shall merge into a blob-like object. The camera shall move 5 units towards the two objects during this time.

> Scene: Composition of nodes. Define scene already here.

To achieve this overall goal while providing the user a user-friendly experience, several components are needed. These are the following, being defined in *QDE - a visual animation system. Software-Architektur.* pp. 29 ff.

- 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. There exists at least a root node at the root scene of the scene graph.
- 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.

However, the above list is not complete. It is somehow intuitively clear, that there needs to be some main component, which holds all the mentioned components and allows a proper handling of the application (like managing resources, shutting down properly and so on).

As the whole architecture uses layers and the MVC principle (see section 3.1.2 and section 3.1.2), the main component is composed of a view and a controller. A model is (at least at this point) not necessary. The view component shall be called *main window* and its controller shall be called *main application*.

> Fix references to subsection (they are displaying section atm)

Main entry point

Before implementing any of these components, the editor application needs an entry point, that is a point where the application starts when being called.

Python does this by evaluating a special variable within a module, called `__name__`. Its value is set to `'__main__'` if the module is "read from standard input, a script, or from an interactive prompt." [6]

All that the entry point needs to do in case of the editor application, is spawning the editor application, execute it and exit again, as can be seen below.

⟨*Main entry point* 14⟩ ≡

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

Fragment referenced in 88a.

But where to place this entry point? A very direct approach would be to implement that main entry point within the main application controller. But when running the editor application by calling it from the command line, calling a controller directly may rather be confusing. Instead it is more intuitive to have only a minimal entry point which is clearly visible as such. Therefore the main entry point will be put in a file called *editor.py* which is at the top level of the *src* directory.

Main application

Although a main entry point is defined by now, the editor application cannot be started as there is no such thing as an editor application yet. Therefore a main application needs to be implemented.

As stated in the requirements, see subsection 3.1.1, Qt version 5 is used through the PyQt5 wrapper. Therefore all functionality of Qt 5 may be used. Qt already offers a main application class, which can be used as a controller. The class is called `QApplication`.

But what does such a main application class actually do? What is its functionality? Very roughly sketched, such a type of application initializes resources, enters a main loop, where it stays until told to shut down, and at the end it frees the allocated 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 [7].

As the main application initializes resources, it act as central node between the various layers of the architecture, initializing them and connecting them using signals.[2, pp. 37 — 38]

Therefore it needs to do at least three things: initialize itself, set up components and connect components. This all happens when the main application is being initialized.

---

[6] `https://docs.python.org/3/library/__main__.html`
[7] http://doc.qt.io/Qt-5/qapplication.html#exec

⟨*Main application declarations* 15a⟩ ≡

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

    ⟨ Main application constructor 15b ⟩
    ⟨ Main application methods 38a ⟩
    ◇
```
Fragment referenced in 88b.

⟨*Main application constructor* 15b⟩ ≡

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

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

⟨ Set up internals for main application 15c, ... ⟩
⟨ Set up components for main application 17b ⟩
⟨ Add root node for main application 23c ⟩
⟨ Set model for scene graph view 30b ⟩
    self.main_window.show()
◇
```
Fragment referenced in 15b.

Setting up the internals is straight forward: Passing any given arguments directly to `QApplication`, setting an application icon, a name as well as a display name.

⟨*Set up internals for main application* 15c⟩ ≡

```
super(Application, self).__init__(arguments)
self.setWindowIcon(QtGui.QIcon("assets/icons/im.png"))
self.setApplicationName("QDE")
self.setApplicationDisplayName("QDE")◇
```
Fragment defined by 15c, 38b.
Fragment referenced in 15b.

The other two steps, setting up the components and connecting them can however not be done at this point, as there simply are no components available. A component to start with is the view component of the main application, the main window.

Main window

Having a very basic implementation of the main application, its view component, the main window, can now be implemented and then be set up by the main application.

The main functionality of the main window is to set up the actual user interface, containing all the views of the components. Qt offers the class `QMainWindow` from which `MainWindow` may inherit.

⟨*Main window declarations* 16a⟩ ≡

```
    common.with_logger
    class MainWindow(QtWidgets.QMainWindow):
        """The main window class.
        Acts as main view for the QDE editor application.
        """
```

⟨ *Main window signals* 16b ⟩

⟨ *Main window methods* 17a, . . . ⟩
◇

Fragment referenced in 89a.

For being able to shut down the main application and therefore the main window, they need to react to a request for shutting down, either by a keyboard shortcut or a menu command. However, the main window is not able to force the main application to quit by itself. It would be possible to pass the main window a reference to the application, but that would lead to tight coupling and is therefore not considered as an option. Signals and slots allow exactly such cross-layer communication without coupling components tightly.

To avoid tight coupling a signal within the main window is introduced, which tells the main application to shut down. A fitting name for the signal might be `do_close`.

⟨*Main window signals* 16b⟩ ≡

```
    do_close = QtCore.pyqtSignal()
```
◇

Fragment referenced in 16a.

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 the main window itself as well as the consumption of the signal by a slot of other classes.

The signal shall be emitted when the escape key on the keyboard is pressed or when the corresponding menu item was selected. As there is no menu at the moment, only the key pressed event is implemented by now.

⟨*Main window methods* 17a⟩ ≡

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

        super(MainWindow, self).__init__(parent)
        self.setup_ui()

    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.do_close.emit()
        else:
            super(MainWindow, self).keyPressEvent(event)
```
◇

Fragment defined by 17a, 19.
Fragment referenced in 16a.

The main window can now be set up by the main application controller, which also listens to the `do_close`
signal through the inherited `quit` slot.

⟨*Set up components for main application* 17b⟩ ≡

 ⟨ *Set up controllers for main application* 29b, . . . ⟩
 ⟨ *Connect controllers for main application* 52a, . . . ⟩
 ⟨ *Set up main window for main application* 17c ⟩◇

Fragment referenced in 15b.

⟨*Set up main window for main application* 17c⟩ ≡

 `self`.main_window = qde_main_window.MainWindow()
 ⟨ *Connect main window components* 17d, . . . ⟩◇

Fragment referenced in 17b.

⟨*Connect main window components* 17d⟩ ≡

 `self`.main_window.do_close.connect(`self`.quit)◇

Fragment defined by 17d, 37a, 56b.
Fragment referenced in 17c.

The used view component for the main window, `QMainWindow`, needs at least a central widget with a
layout for being rendered. [8]

As the main window will set up and hold the whole layout for the application through multiple view
components, a method `setup_ui` is introduced, which sets up the whole layout. The method creates a
central widget containing a grid layout.

---

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

As the main window holds all other view components and a look as proposed in *QDE - a visual animation system. Software-Architektur.* p. 9 is targeted, a simple grid layout does not provide enough possibilities. Instead a horizontal box layout in combination with splitters is used.

Recalling the components, the following layout is approached:

- 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

Provide a picture of the layout here.

⟨*Main window methods* 19⟩ ≡

```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)
        # Ensure that the window is not hidden behind other windows
        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.')

        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)

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

⟨*Set up scene view in main window* 51b⟩
⟨*Set up parameter view in main window* ?⟩
⟨*Set up render view in main window* ?⟩

```python
        horizontal_splitter = QtWidgets.QSplitter()
```
⟨*Add render view to horizontal splitter in main window* ?⟩
⟨*Add parameter view to horizontal splitter in main window* ?⟩

```python
        vertical_splitter = QtWidgets.QSplitter()
        vertical_splitter.setOrientation(QtCore.Qt.Vertical)
        vertical_splitter.addWidget(horizontal_splitter)
```
⟨*Add scene view to vertical splitter in main window* 51c⟩

```python
        horizontal_layout.addWidget(vertical_splitter)
```
◇

Fragment defined by 17a, 19.
Fragment referenced in 16a.

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.

Scene graph

As mentioned in subsection 5.1.4, the scene graph has also two aspects to consider: a graphical aspect as well as its data structure.

```
Define what a scene is by prose and code.
```

As described in chapter 3, two kinds of models are used. A domain model, containing the actual data and a view model, which holds a reference to its corresponding domain model.

As the domain model builds the basis for the whole (data-) structure, it is implemented first.

⟨ *Scene model declarations* 20a ⟩ ≡

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

⟨ *Scene model signals* ? ⟩

⟨ *Scene model methods* 20b ⟩

⟨ *Scene model slots* ? ⟩
◇

Fragment referenced in 89b.

At this point the only known fact is, that a scene is a composition of nodes, and therefore it holds its nodes as a list. Additionally it holds a reference to its parent.

⟨ *Scene model methods* 20b ⟩ ≡

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

    :param parent: the parent scene of this scene. The parent is None if the
                   current scene is the root scene.
    :type parent: SceneModel
    """

    self.id_ = uuid.uuid4()
    self.nodes = []
    self.parent = parent
```
◇

Fragment referenced in 20a.

The counter part of the domain model is the view model. 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 as interface, 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 this case, this are actually two components: the scene graph view as well as the scene view.

It would therefore make sense the use one view model for both components, but this is not possible as the view model of the scene view, `QGraphicsScene`, uses its own data model.

Therefore `QObject` will be used for the scene graph view model and `QGraphicsScene` will be used for the scene view model.

⟨*Scene graph view model declarations* 21a⟩ ≡

```python
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.
    """
```

⟨ *Scene graph view model signals ?* ⟩

⟨ *Scene graph view model constructor* 21b, . . . ⟩

⟨ *Scene graph view model methods* 29a, . . . ⟩

⟨ *Scene graph view model slots ?* ⟩
◇

Fragment referenced in 89c.

In terms of the scene graph, the view model must provide at least a name and a row. In addition, as written above, it holds a reference to the domain model.

⟨*Scene graph view model constructor* 21b⟩ ≡

```python
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.id_ = domain_object.id_
    self.row = row
    self.domain_object = domain_object
    self.name = name
```
◇

Fragment defined by 21b, 28b.
Fragment referenced in 21a.

Scenes may now be instantiated, it is although necessary to manage scenes in a controlled manner. 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."[9]

Such a standard item model is `QAbstractItemModel`[10], which is used as a base class for the scene graph controller.

⟨*Scene graph controller declarations* 22a⟩ ≡

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

⟨ *Scene graph controller signals* 37b ⟩

⟨ *Scene graph controller constructor* 22b, . . . ⟩
⟨ *Scene graph controller methods* 23b, . . . ⟩

⟨ *Scene graph controller slots* 35 ⟩
◇

Fragment referenced in 90c.

As at this point the functionality of the scene graph controller is not fully known, the constructor simply initializes its parent class and an empty list of scenes.

⟨*Scene graph controller constructor* 22b⟩ ≡

```
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)
```
◇

Fragment defined by 22b, 23d.
Fragment referenced in 22a.

As the scene graph controller holds and manages the data, it needs to have at least a root node. As the controller manages both, domain models and the view models, it needs to create both models.

Due to the dependencies of other components this cannot be done within the constructor, as components dependening on the scene graph controller may not be listening to its signals at this point. Therefore this is done in a separate method called `add_root_node`.

---

[9]http://doc.qt.io/qt-5/qtreewidget.html#details
[10] http://doc.qt.io/qt-5/qabstractitemmodel.html

⟨*Scene graph controller add root node* 23a⟩ ≡

```python
    def add_root_node(self):
        """Add a root node to the data structure.
        """

        if self.root_node is None:
            root_node = domain_scene.SceneModel()
            self.view_root_node = guidomain_scene.SceneGraphViewModel(
                row=0,
                domain_object=root_node,
                name=QtCore.QCoreApplication.translate(__class__.__name__, 'Root scene')
            )
            self.do_add_scene.emit(root_node)
            self.layoutChanged.emit()
            self.logger.debug("Added root node")
        else:
            self.logger.warn("Not (re-) adding root node, already present!")
```
◇

Fragment referenced in 23b.

⟨*Scene graph controller methods* 23b⟩ ≡

⟨ *Scene graph controller add root node* 23a ⟩
◇

Fragment defined by 23b, 24, 25ab, 26, 27, 28a, 36ab.
Fragment referenced in 22a.

The root scene can then be added by the main application, when all components are set up properly.

⟨*Add root node for main application* 23c⟩ ≡

```python
    self.scene_graph_controller.add_root_node()
```
◇
Fragment referenced in 15b.

The scene graph controller must also provide the header data, which is used to display the header within the view (due to the usage of the Qt view model). As header data the name of the scenes as well as the [Add referenc]
number of nodes a scene contains shall be displayed.

⟨*Scene graph controller constructor* 23d⟩ ≡

```python
        self.header_data = [
            QtCore.QCoreApplication.translate(__class__.__name__, 'Name'),
            QtCore.QCoreApplication.translate(__class__.__name__, '# Nodes')
        ]
        self.root_node = None
        self.view_root_node = None
```
◇
Fragment defined by 22b, 23d.
Fragment referenced in 22a.

As `QAbstractItemModel` is used as a basis for the scene graph controller, 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."[10]

The method `index` returns the position of an item in the (data-) model for a given row and column below a parent item.

⟨*Scene graph controller methods* 24⟩ ≡

```python
    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 not parent.isValid():
            self.logger.debug((
                "Getting index for row {0}, col {1}, root node"
            ).format(row, column))
            return self.createIndex(row, column, self.view_root_node)

        parent_node = parent.internalPointer()
        self.logger.debug((
            "Getting index for row {0}, col {1}, parent {2}. Children: {3}"
        ).format(row, column, parent_node, len(parent_node.children())))
        child_nodes = parent_node.children()

        # It may happen, that the index is called at the same time as a node is
        # being deleted respectively was deleted. In this case an invalid index is
        # returned.
        try:
            child_node = child_nodes[row]
            return self.createIndex(row, column, child_node)

        except IndexError:
            return QtCore.QModelIndex()◇
```

Fragment defined by 23b, 24, 25ab, 26, 27, 28a, 36ab.
Fragment referenced in 22a.

The method `parent` returns the parent item of an item identified by a provided index. If that index is invalid, an invalid index is returned as well.

⟨*Scene graph controller methods* 25a⟩ ≡

```python
    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
        """

        # self.logger.debug("Getting parent")

        if not model_index.isValid():
            # self.logger.debug("No valid index for parent")
            return QtCore.QModelIndex()

        # The internal pointer of the the model index returns a scene graph view
        # model.
        node = model_index.internalPointer()
        if node and node.parent() is not None:
            # self.logger.debug("Index for parent")
            return self.createIndex(node.parent().row, 0, node.parent())
        else:
            # self.logger.debug("Index for root")
            return QtCore.QModelIndex()
```

◇

Fragment defined by 23b, 24, 25ab, 26, 27, 28a, 36ab.
Fragment referenced in 22a.

Implementing the `columnCount` and `rowCount` methods is straight forward. The former returns simply the number of columns, in this case the number of headers, therefore 2.

⟨*Scene graph controller methods* 25b⟩ ≡

```python
    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
        """

        column_count = len(self.header_data) - 1
        self.logger.debug("Getting column count: %s", column_count)

        return column_count
```

◇

Fragment defined by 23b, 24, 25ab, 26, 27, 28a, 36ab.
Fragment referenced in 22a.

The method `rowCount` returns the number of nodes for a given parent item (identified by its index within the data model).

⟨*Scene graph controller methods* 26⟩ ≡
    in

```python
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():
        self.logger.debug("Parent is not valid")
        row_count = 1
    else:
        # Get the actual object stored by the parent. In this case it is a
        # SceneGraphViewModel.
        node = parent.internalPointer()

        if node is None:
            self.logger.debug("Parent (node) is not valid")
            row_count = 1
        else:
            row_count = len(node.children())

    self.logger.debug("Getting row count: %s", row_count)
    return row_count
```
    ◇

Fragment defined by 23b, 24, 25ab, 26, 27, 28a, 36ab.
Fragment referenced in 22a.

The last method, that has to be implemented due to the usage of `QAbstractItemModel`, is the `data` method. It returns the data for an item identified by the given index for the given role.

A role indicates what type of data is provided. Currently the only role considered is the display of models (further information may be found at `http://doc.qt.io/qt-5/qt.html#ItemDataRole-enum`).

Depending on the column of the model index, the method returns either the name of the scene graph node or the number of nodes a scene contains.

⟨Scene graph controller methods 27⟩ ≡

```python
def data(self, model_index, role=QtCore.Qt.DisplayRole):
    """Return the data stored under 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():
        self.logger.debug("Model index is not valid")
        return None

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

    if node is None:
        self.logger.debug("Node is not valid")
        return None

    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.node_count
```

◇

Fragment defined by 23b, 24, 25ab, 26, 27, 28a, 36ab.
Fragment referenced in 22a.

In addition to the above mentioned methods, the `QAbstractItemModel` offers the method `headerData`, which "returns the data for the given role and section in the header with the specified orientation."[11]

---

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

⟨*Scene graph controller methods* 28a⟩ ≡

```python
    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]
```
◇

Fragment defined by 23b, 24, 25ab, 26, 27, 28a, 36ab.
Fragment referenced in 22a.

One thing, that may stand out, is, that the above defined **data** method returns the number of graph nodes within a scene by accessing the **node_count** property of the *scene graph view model*.

The *scene graph view model* does therefore need to keep track of the nodes it contains, in form of a list, analogous to the domain model.

It does not make sense however to use the list of nodes from the domain model, as the view model will hold references to graphical objects where as the domain model holds only pure data objects. Therefore it is necessary, that the scene view model keeps track of its nodes separately.

⟨*Scene graph view model constructor* 28b⟩ ≡

```python
        self.nodes = []
```
◇

Fragment defined by 21b, 28b.
Fragment referenced in 21a.

The method **node_count** then simply returns the length of the node list.

⟨*Scene graph view model methods* 29a⟩ ≡

```python
    property
    def node_count(self):
        """Return the number of nodes that this scene contains."""

        return len(self.nodes)
```
◇

Fragment defined by 29a, 90a.
Fragment referenced in 21a.

The scene graph controller can now be set up by the main application controller.

⟨*Set up controllers for main application* 29b⟩ ≡

```python
    self.scene_graph_controller = scene.SceneGraphController(self)
```
◇

Fragment defined by 29b, 52d, 84b.
Fragment referenced in 17b.

At this point data structures in terms of a (data-) model and a view model concerning the scene graph 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." [12] Therefore `QTreeView` is used as basis for the scene graph view.

⟨*Scene graph view declarations* 29c⟩ ≡

```python
    ⟨ Scene graph view decorators 93b ⟩
    class SceneGraphView(QtWidgets.QTreeView):
        """The scene graph view widget.
        A widget for displaying and managing the scene graph.
        """

    ⟨ Scene graph view signals 32a, ... ⟩

    ⟨ Scene graph view constructor 30a, ... ⟩

    ⟨ Scene graph view methods 31b ⟩

    ⟨ Scene graph view slots 31a, ... ⟩
```
◇

Fragment referenced in 91.

As at this point the functionality of the scene graph view is not fully known, the constructor simply initializes its parent class.

---

[12]fn:f377826acb87691:http://doc.qt.io/qt-5/qtreeview.html#details

⟨*Scene graph view constructor* 30a⟩ ≡

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

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

    super(SceneGraphView, self).__init__(parent)
```
◇

Fragment defined by 30a, 32c, 33.
Fragment referenced in 29c.

For being able to display anything, the scene graph view needs a controller to work with. In terms of Qt, the controller is called a model, as due its model/view architecture. This model may although not be set too early, as otherwise problems arise. It may only then be added, when the depending components are properly initialized, e.g. when the root node has been added.

⟨*Set model for scene graph view* 30b⟩ ≡

```python
self.main_window.scene_graph_view.setModel(
    self.scene_graph_controller
)
```
◇

Fragment referenced in 15b.

But scenes shall not only be displayed, instead it shall be possible to work with them. What shall be achieved, are three things: Adding and removing scenes, renaming scenes and switching scenes.

To switch between scenes it is necessary to emit what scene was selected. This is needed to tell the other components, such as the node graph for example, that the scene has changed.

Through the `selectionChanged` signal the scene graph view already provides a possibility to detect if another scene was selected. This signal emits an item selection in terms of model indices although.

As this is very view- and model-specific, it would be easier for other components if the selected scene is emitted directly. To emit the selected index of the currently selected scene directly, the slot `on_tree_item_selected` is introduced.

⟨*Scene graph view slots* 31a⟩ ≡

```
     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]
         self.do_select_item.emit(selected_index)
         self.logger.debug("Tree item was selected: %s"% selected_index)◇
```
Fragment defined by 31a, 34.
Fragment referenced in 29c.

The `on_tree_item_selected` slot needs to be triggered as soon as the selection is changed. This is done by connecting the slot with the `selectionChanged` signal. The `selectionChanged` signal is however not directly accessible, it is only accessible through the selection model of the scene graph view (which is given by the usage of `QTreeView`). The selection model can although only be accessed when setting the data model of the view, which needs therefore to be expanded.

⟨*Scene graph view methods* 31b⟩ ≡

```
     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)

         # Use a slot to emit the selected scene graph view model upon the selection of a
         # tree item
         selection_model = self.selectionModel()
         selection_model.selectionChanged.connect(
             self.on_tree_item_selected
         )

         # Set the index to the first node of the model
         self.setCurrentIndex(model.index(0, 0))
         self.logger.debug("Root node selected")◇
```
Fragment referenced in 29c.

As stated in the above code fragment, `on_tree_item_selected` emits another signal containing a reference to the currently selected scene.

⟨*Scene graph view signals* 32a⟩ ≡

```
do_select_item = QtCore.pyqtSignal(QtCore.QModelIndex)
```
    ◇
Fragment defined by 32ab.
Fragment referenced in 29c.

In the same manner as the selection of an item was implemented, the adding and removal of a scene are implemented. However, the tree widget does not provide direct signals for those cases as it is the case when selecting a tree item, instead own signals, slots and actions have to be used.

⟨*Scene graph view signals* 32b⟩ ≡

```
do_add_item = QtCore.pyqtSignal(QtCore.QModelIndex)
do_remove_item = QtCore.pyqtSignal(QtCore.QModelIndex)
```
    ◇
Fragment defined by 32ab.
Fragment referenced in 29c.

An action gets triggered, typically by hovering over some item (in terms of a context menu for example) or by pressing a defined keyboard shortcut. For the adding and the removal, a keyboard shortcut will be used.

Adding of a scene item shall happen when pressing the `a` key on the keyboard.

⟨*Scene graph view constructor* 32c⟩ ≡

```
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)
```
    ◇
Fragment defined by 30a, 32c, 33.
Fragment referenced in 29c.

Removal of a selected node shall be triggered upon the press of the `delete` and the `backspace` key on the keyboard.

⟨Scene graph view constructor 33⟩ ≡

```
        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.setShortcut(Qt.QKeySequence('Backspace'))
        remove_action.setShortcutContext(QtCore.Qt.WidgetShortcut)
        remove_action.triggered.connect(self.on_tree_item_removed)
        self.addAction(remove_action)
```
    ◇
Fragment defined by 30a, 32c, 33.
Fragment referenced in 29c.

As can be seen in the two above listings, the `triggered` signals are connected with a corresponding slot. All these slots do is emitting another signal, but this time it contains a scene graph view model, which may be used by other components, instead of a model index.

```python
    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.do_add_item.emit(selected_item)
```
⟨ *Scene graph view log tree item added* 39d ⟩

```python
    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.do_remove_item.emit(selected_item)
```
⟨ *Scene graph view log tree item removed* 40 ⟩

```python
            else:
                self.logger.warn("Root scene cannot be deleted")
        else:
            self.logger.warn('No item selected for removal')
```
◇

Fragment defined by 31a, 34.
Fragment referenced in 29c.

One of the mentioned other components is the scene graph controller. He needs to be informed whenever a scene was added, removed or selected, so that he is able to manage his data model correspondingly.

⟨*Scene graph controller slots* 35⟩ ≡

```python
QtCore.pyqtSlot(QtCore.QModelIndex)
def on_tree_item_added(self, selected_item):
    # TODO: Document method.

    self.insertRows(0, 1, selected_item)
    self.logger.debug("Added new scene")

QtCore.pyqtSlot(QtCore.QModelIndex)
def on_tree_item_removed(self, selected_item):
    # TODO: Document method.

    if not selected_item.isValid():
        self.logger.warn("Selected scene is not valid, note removing")
        return False

    row = selected_item.row()
    parent = selected_item.parent()
    self.removeRows(row, 1, parent)

QtCore.pyqtSlot(QtCore.QModelIndex)
def on_tree_item_selected(self, selected_item):
    # TODO: Document method.

    if not selected_item.isValid():
        self.logger.warn("Selected scene is not valid")
        return False

    selected_scene_view_model = selected_item.internalPointer()
    selected_scene_domain_model = selected_scene_view_model.domain_object
    self.do_select_scene.emit(selected_scene_domain_model)◇
```

Fragment referenced in 22a.

Having the slots for adding, removing and selecting scene graph items implemented, the actual methods for adding and removing scenes, `on_tree_item_added` and `on_tree_item_removed`, are still missing.

When inserting a new scene graph item, actually a row must be inserted, as the data model (Qt's) is using rows to represent the data. At the same time the controller has to keep track of the domain model.

As can be seen in the implementation below, it is not necessary to add the created model instances to a list of nodes, the usage of `QAbstractItemModel` keeps already track of this.

⟨ₘScene graph controller methods 36a⟩ ≡

```python
    def insertRows(self, row, count, parent=QtCore.QModelIndex()):
        # TODO: Document method.

        if not parent.isValid():
            return False

        parent_node = parent.internalPointer()
        self.beginInsertRows(parent, row, row + count - 1)
        domain_model = domain_scene.SceneModel(parent_node.domain_object)
        view_model = guidomain_scene.SceneGraphViewModel(
            row=row,
            domain_object=domain_model,
            parent=parent_node
        )
        self.endInsertRows()

        self.layoutChanged.emit()
        self.do_add_scene.emit(domain_model)

        return True
```
◇

Fragment defined by 23b, 24, 25ab, 26, 27, 28a, 36ab.
Fragment referenced in 22a.

The same logic applies when removing a scene.

⟨ₘScene graph controller methods 36b⟩ ≡

```python
    def removeRows(self, row, count, parent=QtCore.QModelIndex()):
        # TODO: Document method.

        if not parent.isValid():
            self.logger.warn("Cannot remove rows, parent is invalid")
            return False

        self.beginRemoveRows(parent, row, row + count - 1)
        parent_node = parent.internalPointer()
        node_index = parent.child(row, parent.column())
        node       = node_index.internalPointer()
        node.setParent(None)
        # TODO: parent_node.child_nodes.remove(node)
        self.endRemoveRows()
        self.logger.debug(
            "Removed {0} rows starting from {1} for parent {2}. Children: {3}".format(
                count, row, parent_node, len(parent_node.children())
            )
        )

        self.layoutChanged.emit()
        self.do_remove_scene.emit(node.domain_object)

        return True
```
◇

Fragment defined by 23b, 24, 25ab, 26, 27, 28a, 36ab.
Fragment referenced in 22a.

As before, the main application needs connect the components, in this case the scene graph view with the scene graph controller.

⟨*Connect main window components* 37a⟩ ≡

```
self.main_window.scene_graph_view.do_add_item.connect(
    self.scene_graph_controller.on_tree_item_added
)
self.main_window.scene_graph_view.do_remove_item.connect(
    self.scene_graph_controller.on_tree_item_removed
)
self.main_window.scene_graph_view.do_select_item.connect(
    self.scene_graph_controller.on_tree_item_selected
)◇
```
Fragment defined by 17d, 37a, 56b.
Fragment referenced in 17c.

To inform other components, such as the node graph for example, the scene graph controller emits signals when a scene is being added, removed or selected respectively.

⟨*Scene graph controller signals* 37b⟩ ≡

```
do_add_scene   = QtCore.pyqtSignal(domain_scene.SceneModel)
do_remove_scene = QtCore.pyqtSignal(domain_scene.SceneModel)
do_select_scene = QtCore.pyqtSignal(domain_scene.SceneModel)
    ◇
```
Fragment referenced in 22a.

At this point it is possible to manage scenes in terms of adding and removing them. The scenes are added to (or removed from respectively) the graphical user interface as well as the data structure.

So far the application (or rather the scene graph) seems to be working as intended. But how does one 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 subsection 3.0.1, 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, section 5.4.

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.

Logging

It is always very useful to have a facility which allows tracing of errors or even just the flow of an application. Logging does allow such aspects by outputting text messages to a defined output, such as STDERR, STDOUT, streams or files.

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.

As logging is a very central aspect of the application, 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`.

⟨*Main application methods* 38a⟩ ≡

```python
    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)◇
```
Fragment referenced in 15a.

For not having only basic logging available, a logging configuration is defined. 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.

This configuration allows to get an arbitrarily named logger which uses that configuration.

⟨*Set up internals for main application* 38b⟩ ≡

```python
        self.setup_logging()◇
```
Fragment defined by 15c, 38b.
Fragment referenced in 15b.

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 [13].

The decorator will be available as a method named `with_logger`. The method has the following functionality.

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

⟨*Set logger name* 39a⟩ ≡

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

Fragment referenced in 93a.

- Provide an easy to use interface for logging.

⟨*Logger interface* 39b⟩ ≡

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

return cls◇
```

Fragment referenced in 93a.

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

⟨*With logger example* 39c⟩ ≡

```python
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."))
```

◇

Fragment never referenced.

The logging facility may now be used wherever it is useful to log something. Such a place is for example the adding and removal of scenes in the scene graph view.

⟨*Scene graph view log tree item added* 39d⟩ ≡

```python
self.logger.debug("A new scene graph item was added.")
```

◇

Fragment referenced in 34.

---

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

⟨*Scene graph view log tree item removed* 40 ⟩ ≡

```
    self.logger.debug((
        "The scene graph item at row {row} "
        "and column {column} was removed."
    ).format(
        row=selected_item.row(),
        column=selected_item.column()
    ))
    ◇
```

Fragment referenced in 34.

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.

Now, having the scene graph component as well as an interface to log messages throughout the application implemented, the next component may be approached.

Scenes build the basis for the scene graph and the node graph as well. This is a good point to begin with the implementation of the node graph.

5.1.5 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 *QDE - a visual animation system. Software-Architektur.* p. 30 and 31) as the node structure will 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 Building blocks of a real time animation.

>**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 A composition of nodes, connected by edges.

>**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

As mentioned before, nodes are the building blocks of a real time animation. 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.

Therefore the system shall be able to 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.

The nodes are always 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, starting with the root node within the root scene. 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 would make sense 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 (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 within the code, 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, it is important to define what a node actually is. As a node may be referenced by other nodes, it must be uniquely identifiable and must therefore have a globally unique identifier. Concerning the visual representation, a node shall have a name as well as a description.

Each node can have multiple inputs and at least one output. The inputs may be either be atomic types (which have to be defined) or references to other nodes. The same applies to the outputs.

A node consists also of a definition. In terms of implicit surfaces this section contains the actual definition of a node in terms of the implicit function. In terms of triangle based meshes this is the part where the mesh and all its prerequisites as vertex array buffers and vertex array objects are set up or used from a given context.

In addition to a definition, a node contains an invocation part, which is the call of its defining function (coming from the definition mentioned just before) while respecing the parameters.

A node shall be able to have one or more parts. A part typically contains the "body" of the node in terms of code and represents therefore the code-wise implementation of the node. A part can be processed when evaluating the node. This part of the node is mainly about evaluating inputs and passing them on to a shader.

Furthermore a node may contain children, child-nodes, which are actually references to other nodes combined with properties such as a name, states and so on.

Each node can have multiple connections. A connection is composed of an input and an output plus a reference to a part. The input respectively the output may be zero, what means that the part of the input or output is internal.

Or, a bit more formal:

⟨*Connections between nodes in EBNF notation* 42⟩ ≡

```
input = internal input | external input
internal input = zero reference, part reference
external input = node reference, part reference
zero reference = "0"
node reference = "uuid4"
part reference = "uuid4"
◇
```
Fragment never referenced.

Recapitulating the above made thoughts, a node is essentially composed by the following elements:

The inputs and outputs may be parameters of an atomic type, as stated above. This seems like a good point to define the atomic types the system will have:

- Generic
- Float
- Text
- Scene
- Image
- Dynamic
- Mesh

As these atomic types are the foundation of all other nodes, the system must ensure, that they are initialized before all other nodes. Before being able to create instances of atomic types, there must be classes defining them.

For identification of the atomic types, an enumerator is used. Python provides the `enum` module, which provides a convenient interface for using enumerations[15].

---

[15]https://docs.python.org/3/library/enum.html

Component	Description
ID	A global unique identifier (UUID [14])
Name	The name of the node, e.g. "Cube".
Description	A description of the node's purpose.
Inputs	A list of the node's 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 node's outputs. The outputs may also either be parameters or references to other nodes.
Definitions	A list of the node's definitions. This may be an actual definition by a (shader-) function in terms of an implicit surface or prerequisites as vertex array buffers in terms of a triangle based mesh.
Invocation	A list of the node's invocations or calls respectively.
Parts	Defines parts that may be processed 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 of the node's inputs and outputs. Each connection is composed by two parts: A reference to another node and a reference to an input or an output of that node. Is the reference not set, that is, its value is zero, this means that the connection is internal.

⟨*Node type declarations* 43⟩ ≡

```
class NodeType(enum.Enum):
    """Atomic types which a parameter may be made of."""

    GENERIC = 0
    FLOAT   = 1
    TEXT    = 2
    SCENE   = 3
    IMAGE   = 4
    DYNAMIC = 5
    MESH    = 6
    IMPLICIT = 7
```
◊

Fragment referenced in 93c.

Now, having identifiers for the atomic types available, the atomic types themselves can be implemented. The atomic types will be used for defining various properties of a node and are therefore its parameters.

Each node may contain one or more parameters as inputs and at least one parameter as output. Each parameter will lead back to its atomic type by referencing the unique identifier of the atomic type. For being able to distinguish multiple parameters using the same atomic type, it is necessary that each instance of an atomic type has its own identifier in form of an instance identifier (instance ID).

⟨*Parameter declarations* 44⟩ ≡

```python
class AtomicType(object):
    """Represents an atomic type and is the basis for each node."""

    def __init__(self, id_, type_):
        """Constructor.

        :param id_: the globally unique identifier of the atomic type.
        :type id_: uuid.uuid4
        :param type_: the type of the atomic type, e.g. "float".
        :type type_: types.NodeType
        """

        self.id_   = id_
        self.type_ = type_
```

◇

Fragment defined by 44, 45a, 95.
Fragment referenced in 94.

As the word atomic indicates, these types are atomic, meaning there only exists one explicit instance per type, which is therefore static. As can be seen in the code fragment below, the atomic types are parts of node definitions themselves. Only the creation of the generict atomic type is shown, the rest is omitted and can be found at ██ Add referenc

⟨*Parameter declarations* 45a⟩ ≡

```python
class AtomicTypes(object):
    """Creates and holds all atomic types of the system."""

    staticmethod
    def create_node_definition_part(id_, type_):
        """Creates a node definition part based on the given identifier and
        type.

        :param id_: the identifiert to use for the part.
        :type id_: uuid.uuid4
        :param type_: the type of the part.
        :tpye type_: qde.editor.domain.parameter.AtomicType

        :return: a node definition part.
        :rtype: qde.editor.domain.node.NodeDefinitionPart
        """

        def create_func(id_, default_function, name, type_):
            node_part = node.NodePart(id_, default_function)
            node_part.type_ = type_
            node_part.name = name
            return node_part

        node_definition_part = node.NodeDefinitionPart(id_)
        node_definition_part.type_ = type_
        node_definition_part.creator_function = create_func

        return node_definition_part

    Generic = create_node_definition_part.__func__(
        id_="54b20acc-5867-4535-861e-f461bdbf3bf3",
        type_=types.NodeType.GENERIC
    )
```

◇

Fragment defined by 44, 45a, 95.
Fragment referenced in 94.

Having the atomic types defined, nodes may now be defined.

⟨*Node domain model declarations* 45b⟩ ≡

```python
class Node(object):
    """Represents a node."""

    # Signals
```
⟨ *Node domain model signals* ? ⟩

⟨ *Node domain model constructor* 46a, … ⟩

⟨ *Node domain model methods* 48a ⟩
◇

Fragment referenced in 99a.

⟨*Node domain model constructor* 46a⟩ ≡

```python
    def __init__(self, id_, name="New node"):
        """Constructor.

        :param id_: the globally unique identifier of the node.
        :type  id_: uuid.uuid4
        :param name: the name of the node.
        :type  name: str
        """

        self.id_  = id_
        self.name = name

        self.definition = None
        self.description = ""
        self.parent = None
        self.inupts = []
        self.outputs = []
        self.parts = []
        self.nodes = []
        self.connections = []
```
◇

Fragment defined by 46a, 49b.
Fragment referenced in 45b.

While the details of a node are rather unclear at the moment, it is clear that a node needs to have a view model, which renders a node within a scene of the node graph.

As Qt does not offer a graph view by default, it is necessary to implement such a graph view.

The most obvious choice for this implementation is the `QGraphicsView` component, which displays the contents of a `QGraphicsScene`, whereas `QGraphicsScene` manages `QGraphicsObject` components.

It is therefore obvious to use the `QGraphicsObject` component for representing graph nodes through a view model.

⟨*Node view model declarations* 46b⟩ ≡

```python
    class NodeViewModel(Qt.QGraphicsObject):
        """Class representing a single node within GUI."""

        # Constants
        WIDTH = 20
        HEIGHT = 17

        # Signals
```
⟨ *Node view model signals* ? ⟩

⟨ *Node view model constructor* 47a, ... ⟩

⟨ *Node view model methods* 47b, ... ⟩
◇

Fragment referenced in 99b.

⟨*Node view model constructor* 47a⟩ ≡

```python
def __init__(self, id_, domain_object, parent=None):
    """Constructor.

    :param id_: the globally unique identifier of the atomic type.
    :type  id_: uuid.uuid4
    :param domain_object: Reference to a scene model.
    :type  domain_object: qde.editor.domain.scene.SceneModel
    :param parent: The parent of the current view widget.
    :type  parent: QtCore.QObject
    """

    super(NodeViewModel, self).__init__(parent)
    self.id_ = id_
    self.domain_object = domain_object

    self.position = QPoint(0, 0)
    self.width = 4
```
◇

Fragment defined by 47a, 48c, 99c.
Fragment referenced in 46b.

To distinguish nodes, the name and the type of a node is used. It makes sense to access both attributes directly via the domain model instead of duplicating them.

⟨*Node view model methods* 47b⟩ ≡

```python
property
def type_(self):
    """Return the type of the node, determined by its domain model.

    :return: the type of the node.
    :rtype: types.NodeType
    """

    return self.domain_model.type_
```
◇

Fragment defined by 47bc, 48b, 49ac, 101.
Fragment referenced in 46b.

⟨*Node view model methods* 47c⟩ ≡

```python
property
def name(self):
    """Return the name of the node, determined by its domain model.

    :return: the name of the node.
    :rtype: str
    """

    return self.domain_model.name
```
◇

Fragment defined by 47bc, 48b, 49ac, 101.
Fragment referenced in 46b.

However, the domain model does not provide access to its type at the moment. The type is directly derived from the primary output of a node. If a node has no outputs at all, its type is assumed to be generic.

⟨*Node domain model methods* 48a⟩ ≡

```
property
    def type_(self):
        """Return the type of the node, determined by its primary output.
        If no primary output is given, it is assumed that the node is of
        generic type."""

        type_ = types.NodeType.GENERIC

        if len(self.outputs) > 0:
            type_ = self.outputs[0].type_

        return type_
```
◇

Fragment referenced in 45b.

Concerning the drawing of nodes (or painting, as Qt calls it) , each node type may be used multiple times. But instead of re-creating the same image representation over and over again, it makes sense to create it only once per node type. Qt provides `QtPixmap` and `QtPixmapCache` for this use case.

⟨*Node view model methods* 48b⟩ ≡

```
def paint(self, painter, option, widget):
    """Paint the node.

    First a pixmap is loaded from cache if available, otherwise
    a new pixmap gets created. If the current node is selected a
    rectangle gets additionally drawn on it. Finally the name, the type
    as well as the subtype gets written on the node.
    """
```

⟨ *Node view model methods paint* 50a, ... ⟩
◇

Fragment defined by 47bc, 48b, 49ac, 101.
Fragment referenced in 46b.

Each node has a cache key assigned, which is used to identify that node.

⟨*Node view model constructor* 48c⟩ ≡

```
        self.cache_key = None
```
◇

Fragment defined by 47a, 48c, 99c.
Fragment referenced in 46b.

The cache key is composed of the type of the node, its status and whether it is selected or not.

⟨*Node view model methods* 49a⟩ ≡

```python
    def create_cache_key(self):
        """Create an attribute based cache key for finding and creating
        pixmaps."""

        return "{type_name}{status}{selected}".format(
            type_name=self.type_,
            status=self.status,
            selected=self.isSelected(),
        )
```
◇

Fragment defined by 47bc, 48b, 49ac, 101.
Fragment referenced in 46b.

As can be seen in the above code fragment, the status property of the node is used to create a cache key, but currently nodes do not have a status.

It may make sense although to provide a status for each node, which allows to output eventual problems like a node not having required connections and so on.

This status is added to the constructor of the domain model of a node.

⟨*Node domain model constructor* 49b⟩ ≡

```python
        self.status = flag.NodeStatus.OK
```
◇

Fragment defined by 46a, 49b.
Fragment referenced in 45b.

Concerning the view model, again the status of the domain model is used as otherwise different states between user interface and domain model would be possible in the worst case.

⟨*Node view model methods* 49c⟩ ≡

```python
    property
    def status(self):
        """Return the current status of the node.

        :return: the current status of the node.
        :rtype: flag.NodeStatus
        """

        return self.domain_object.status
```
◇

Fragment defined by 47bc, 48b, 49ac, 101.
Fragment referenced in 46b.

Therefore it can now be checked, whether a node has a cache key or not. If it has no cache key, a new cache key is created.

⟨*Node view model methods paint* 50a⟩ ≡

```
        if self.cache_key is None:
            self.cache_key = self.create_cache_key()
    ◇
```
Fragment defined by 50abc.
Fragment referenced in 48b.

The cache key itself is then used to find a corresponding pixmap.

⟨*Node view model methods paint* 50b⟩ ≡

```
        pixmap = Qt.QPixMapCache.find(self.cache_key)
    ◇
```
Fragment defined by 50abc.
Fragment referenced in 48b.

If no pixmap with the given cache key exists, a new pixmap is being created and added to the cache using the cache key created before.

⟨*Node view model methods paint* 50c⟩ ≡

```
        if pixmap is None:
            pixmap = self.create_pixmap()
            Qt.QPixmapCache.insert(self.cache_key, pixmap)
    ◇
```
Fragment defined by 50abc.
Fragment referenced in 48b.

For actually displaying the nodes, another component is necessary: the scene view which is a graph consisting the nodes and edges.

⟨*Scene view declarations* 50d⟩ ≡

```
    common.with_logger
    class SceneView(Qt.QGraphicsView):
        """Scene view widget.
        A widget for displaying and managing scenes including their nodes and
        connections between nodes."""

        # Signals
⟨ Scene view signals ? ⟩

⟨ Scene view constructor 51a, … ⟩
⟨ Scene view methods ? ⟩
⟨ Scene view slots 56c ⟩
    ◇
```
Fragment referenced in 91.

⟨*Scene view constructor* 51a⟩ ≡

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

    :param parent: the parent of this scene view.
    :type parent: Qt.QObject
    """

    super(SceneView, self).__init__(parent)
```
◇

Fragment defined by 51a, 85b.
Fragment referenced in 50d.

The scene view can now be set up by the main window and is then added to its vertical splitter.

⟨*Set up scene view in main window* 51b⟩ ≡

```python
self.scene_view = guiscene.SceneView()
self.scene_view.setObjectName('scene_view')
size_policy = QtWidgets.QSizePolicy(
    QtWidgets.QSizePolicy.Expanding,
    QtWidgets.QSizePolicy.Expanding
)
size_policy.setHorizontalStretch(2)
size_policy.setVerticalStretch(0)
size_policy.setHeightForWidth(self.scene_view.sizePolicy().hasHeightForWidth())
self.scene_view.setSizePolicy(size_policy)
self.scene_view.setMinimumSize(Qt.QSize(0, 0))
self.scene_view.setAutoFillBackground(False)
self.scene_view.setFrameShape(QtWidgets.QFrame.StyledPanel)
self.scene_view.setFrameShadow(QtWidgets.QFrame.Sunken)
self.scene_view.setLineWidth(1)
self.scene_view.setVerticalScrollBarPolicy(QtCore.Qt.ScrollBarAsNeeded)
self.scene_view.setHorizontalScrollBarPolicy(QtCore.Qt.ScrollBarAsNeeded)
brush = QtGui.QBrush(Qt.QColor(0, 0, 0, 255))
brush.setStyle(QtCore.Qt.NoBrush)
self.scene_view.setBackgroundBrush(brush)
self.scene_view.setAlignment(QtCore.Qt.AlignLeading|QtCore.Qt.AlignLeft|QtCore.Qt.AlignTop)
self.scene_view.setDragMode(QtWidgets.QGraphicsView.RubberBandDrag)
self.scene_view.setTransformationAnchor(QtWidgets.QGraphicsView.AnchorUnderMouse)
self.scene_view.setOptimizationFlags(QtWidgets.QGraphicsView.DontAdjustForAntialiasing)
```
◇

Fragment referenced in 19.

⟨*Add scene view to vertical splitter in main window* 51c⟩ ≡

```python
vertical_splitter.addWidget(self.scene_view)
```
◇

Fragment referenced in 19.

At this point the scene view does not react whenever the scene is changed by the scene graph view. As before, the main application needs connect the components.

As the scene graph view and the scene view have use both different view models, it would not make much sense to connect them directly. Instead it makes sense to connect the `do_select_scene` signal of the scene graph controller with the `on_scene_changed` slot of the scene controller as they both use the domain model of the scene.

⟨*Connect controllers for main application* 52a⟩ ≡

```
    self.scene_graph_controller.do_select_scene.connect(
        self.scene_controller.on_scene_changed
    )◇
```

Fragment defined by 52ab.
Fragment referenced in 17b.

The scene controller does not manage scene models directly, as the scene graph controller does. Instead it reacts on signals sent by the latter and manages its own scene view models.

⟨*Connect controllers for main application* 52b⟩ ≡

```
    self.scene_graph_controller.do_add_scene.connect(
        self.scene_controller.on_scene_added
    )
    self.scene_graph_controller.do_remove_scene.connect(
        self.scene_controller.on_scene_removed
    )◇
```

Fragment defined by 52ab.
Fragment referenced in 17b.

⟨*Scene controller declarations* 52c⟩ ≡

```
    common.with_logger
    class SceneController(Qt.QObject):
        """The scene controller.

        A controller for switching scenes and managing the nodes of a scene by
        adding, editing and removing nodes to / from a scene.
        """

        # Signals
⟨ Scene controller signals 55c ⟩

⟨ Scene controller constructor 54b ⟩
⟨ Scene controller methods ? ⟩

⟨ Scene controller slots 55a, … ⟩
        ◇
```

Fragment referenced in 90c.

⟨*Set up controllers for main application* 52d⟩ ≡

```
    self.scene_controller = scene.SceneController(self)◇
```

Fragment defined by 29b, 52d, 84b.
Fragment referenced in 17b.

The scene view models are of type `QGraphicsScene` and are used to manage nodes. They represent a certain scene of the scene graph and hold the nodes of that scene.

⟨*Scene view model declarations* 53a⟩ ≡

```
common.with_logger
class SceneViewModel(Qt.QGraphicsScene):
    """Scene view model.
    Represents a certain scene from the scene graph and is used to manage the
    nodes of that scene."""

    # Constants
    WIDTH = 15
    HEIGHT = 15

    # Signals
```
⟨ *Scene view model signals* ? ⟩

⟨ *Scene view model constructor* 53b ⟩
⟨ *Scene view model methods* 54a, … ⟩
◇

Fragment referenced in 89c.

⟨*Scene view model constructor* 53b⟩ ≡

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

    :param domain_object: Reference to a scene model.
    :type  domain_object: qde.editor.domain.scene.SceneModel
    :param parent:        The parent of the current view model.
    :type  parent:        qde.editor.gui_domain.scene.SceneViewModel
    """

    super(SceneViewModel, self).__init__(parent)

    self.id_ = domain_object.id_
    self.nodes = []

    self.width = SceneViewModel.WIDTH * 20
    self.height = SceneViewModel.HEIGHT * 17

    self.setSceneRect(0, 0, self.width, self.height)
    self.setItemIndexMethod(self.NoIndex)
```
◇

Fragment referenced in 53a.

For being able to distinguish different scenes, their identifier will be drawn at the top left position.

⟨*Scene view model methods* 54a⟩ ≡

```python
def drawBackground(self, painter, rect):
    # io = Qt.QGraphicsTextItem()
    # io.setPos(0, 0)
    # io.setDefaultTextColor(Qt.QColor(102, 102, 102))
    # io.setPlainText(
    #     "Scene: {0}".format(str(self))
    # )
    # self.addItem(io)

    scene_rect = self.sceneRect()
    text_rect = QtCore.QRectF(scene_rect.left() + 4,
                              scene_rect.top()   + 4,
                              scene_rect.width() - 4,
                              scene_rect.height() - 4)
    message = str(self)
    font = painter.font()
    font.setBold(True)
    font.setPointSize(14)
    painter.setFont(font)
    painter.setPen(QtCore.Qt.lightGray)
    painter.drawText(text_rect.translated(2, 2), message)
    painter.setPen(QtCore.Qt.black)
    painter.drawText(text_rect, message)◇
```

Fragment defined by 54a, 90b.
Fragment referenced in 53a.

As the scene controller does not directly manages scenes, it has to react upon the signals sent by the scene graph controller.

Additionally it needs to keep track of the currently selected scene, by holding a reference to that. The common identifier is the identifier of the domain model.

⟨*Scene controller constructor* 54b⟩ ≡

```python
def __init__(self, parent):
    """Constructor.

    :param parent: the parent of this scene controller.
    :type  parent: Qt.QObject
    """

    super(SceneController, self).__init__(parent)

    self.scenes = {}
    self.current_scene = None
```
◇
Fragment referenced in 52c.

Whenever a new scene is created, the scene controller needs to create a scene of type `QGraphicsScene` and needs to keep track of that scene.

⟨ₘScene controller slots 55a⟩ ≡

```python
    QtCore.pyqtSlot(domain_scene.SceneModel)
    def on_scene_added(self, scene_domain_model):
        """React when a scene was added.

        :param scene_domain_model: the scene that was added.
        :type  scene_domain_model: qde.domain.scene.SceneModel
        """

        if scene_domain_model.id_ not in self.scenes:
            scene_view_model = guidomain_scene.SceneViewModel(
                domain_object=scene_domain_model
            )
            self.scenes[scene_domain_model.id_] = scene_view_model
            self.logger.debug("Scene '%s' was added"% scene_view_model)
        else:
            self.logger.debug("Scene '%s' already known"% scene)
```
◇

Fragment defined by 55ab, 56a.
Fragment referenced in 52c.

Whenever a scene is deleted, it needs to delete the scene from its known scenes as well.

⟨ₘScene controller slots 55b⟩ ≡

```python
    QtCore.pyqtSlot(domain_scene.SceneModel)
    def on_scene_removed(self, scene_domain_model):
        """React when a scene was removed/deleted.

        :param scene_domain_model: the scene that was removed.
        :type  scene_domain_model: qde.domain.scene.SceneModel
        """

        if scene_domain_model.id_ in self.scenes:
            del(self.scenes[scene_domain_model.id_])
            self.logger.debug("Scene '%s' was removed"% scene_domain_model)
        else:
            self.logger.warn((
                "Scene '%s' should be removed, "
                "but is not known"
            ) % scene_domain_model)
```
◇

Fragment defined by 55ab, 56a.
Fragment referenced in 52c.

To actually change the scene, the scene controller needs to react whenever the scene was changed. This happens by reacting to the `do_select_scene` signal sent by the scene graph controller.

⟨ₘScene controller signals 55c⟩ ≡

```python
    do_change_scene = QtCore.pyqtSignal(guidomain_scene.SceneViewModel)
```
◇

Fragment referenced in 52c.

⟨*Scene controller slots* 56a⟩ ≡

```
    QtCore.pyqtSlot(domain_scene.SceneModel)
    def on_scene_changed(self, scene_domain_model):
        """Gets triggered when the scene was changed by the view.

        :param scene_domain_model: The currently selected scene.
        :type  scene_domain_model: qde.editor.domain.scene.SceneModel
        """

        if scene_domain_model.id_ in self.scenes:
            self.current_scene = self.scenes[scene_domain_model.id_]
            self.do_change_scene.emit(self.current_scene)
            self.logger.debug("Scene changed: %s", self.current_scene)
        else:
            self.logger.warn((
                "Should change to scene '%s', "
                "but that scene is not known"
            ) % scene_domain_model)
```
◇

Fragment defined by 55ab, 56a.
Fragment referenced in 52c.

As can be seen in the fragment above, the scene controller actually emits another signal, `do_change_scene`, which provides the view model of the currently set scene. The `do_change_scene` signal is then in turn consumed by the `on_scene_changed` slot of the scene view for actually changing the displayed scene.

⟨*Connect main window components* 56b⟩ ≡

```
    self.scene_controller.do_change_scene.connect(
        self.main_window.scene_view.on_scene_changed
    )
```
◇

Fragment defined by 17d, 37a, 56b.
Fragment referenced in 17c.

⟨*Scene view slots* 56c⟩ ≡

```
    QtCore.pyqtSlot(scene.SceneViewModel)
    def on_scene_changed(self, scene_view_model):
        # TODO: Document method

        self.setScene(scene_view_model)
        # TODO: self.scrollTo(scene_view_model.view_position)
        self.scene().invalidate()
        self.logger.debug("Scene has changed: %s", scene_view_model)
```
◇

Fragment referenced in 50d.

At this point scenes can be managed and displayed but they still cannot be rendered as nodes cannot be added yet. First of all as there are no nodes yet and second as there exists no possibility to add nodes.

Thinking of the definition of what shall be achieved, as defined at the beginning of this chapter, a node defining a sphere is implemented.

⟨*Implicit sphere node* 57a⟩ ≡

```
{
    "name": "Implicit sphere",
    "id_": "16d90b34-a728-4caa-b07d-a3244ecc87e3",
    "description": "Definition of a sphere by using implicit surfaces",
    "inputs": [
⟨ Implicit sphere node inputs 57b ⟩
    ],
    "outputs": [
⟨ Implicit sphere node outputs 57c ⟩
    ],
    "definitions": [
⟨ Implicit sphere node definitions 58a ⟩
    ],
    "invocations": [
⟨ Implicit sphere node invocations 58b ⟩
    ],
    "parts": [
⟨ Implicit sphere node parts 59a ⟩
    ],
    "nodes": [
⟨ Implicit sphere node nodes ? ⟩
    ],
    "connections": [
⟨ Implicit sphere node connections 59b ⟩
    ]
}◇
```

Fragment referenced in 102a.

At the current point the sphere node will only have one input: the radius of the sphere. The positition of the sphere will be at the center (meaning the X-, the Y- and the Z-position are all 0). For being able to change the positition, another node will be introduced.

⟨*Implicit sphere node inputs* 57b⟩ ≡

```
{
    "name": "radius",
    "atomic_id": "468aea9e-0a03-4e63-b6b4-8a7a76775a1a",
    "default_value": {
        "type_": "float",
        "value": "1"
    },
    "id_": "f5c6a538-1dbc-4add-a15d-ddc4a5e553da",
    "description": "The radius of the sphere",
    "min_value": "-1000",
    "max_value": "1000"
}◇
```

Fragment referenced in 57a.

The output of the sphere node is of type implicit as the node represents an implicit surface.

⟨*Implicit sphere node outputs* 57c⟩ ≡

```
{
    "name": "output",
    "id_": "a3ac68e5-5afe-4779-9e9f-5b619e041ae6",
    "atomic_id": "c019271c-35b6-425c-9ff2-a1d893111adb"
}◇
```

Fragment referenced in 57a.

The definition of the node is the actual implementation of a sphere as a implicit surface.

⟨*Implicit sphere node definitions* 58a⟩ ≡

```
{
    "id_": "99d20a26-f233-4310-adb2-5e540726d079",
    "script": [
        "// Returns the signed distance to a sphere with given radius for the",
        "// given position.",
        "float sphere(vec3 position, float radius)",
        "{",
        "    return length(position) - radius;",
        "}"
    ]
}◇
```
Fragment referenced in 57a.

The invocation of the node is simply calling the above definition using the parameters of the node, which is in this case the radius.

The parameters are in case of implicit surfaces uniform variables of the type of the parameter, as implicit surfaces are rendered by the fragment shader. The uniform variables are defined by a type and an identifier, whereas in the case of paramaters their identifier is used.

The position of the node is an indirect parameter, which is not defined by the node's inputs. It will be setup by the node's parts.

⟨*Implicit sphere node invocations* 58b⟩ ≡

```
{
    "id_": "4cd369d2-c245-49d8-9388-6b9387af8376",
    "type": "implicit",
    "script": [
        "float s = sphere(",
        "    16d90b34-a728-4caa-b07d-a3244ecc87e3-position,",
        "    5c6a538-1dbc-4add-a15d-ddc4a5e553da",
        ");"
    ]
}◇
```
Fragment referenced in 57a.

The parts of the node, in this case it is only one part, contain the body of the node. The body is about evaluating the inputs and passing them on to a shader.

⟨*Implicit sphere node parts* 59a⟩ ≡

```
{
    "id_": "74b73ce7-8c9d-4202-a533-c77aba9035a6",
    "name": "Implicit sphere node function",
    "type_": "implicit",
    "script": [
        "# -*- coding: utf-8 -*-",
        "",
        "from PyQt5 import QtGui",
        "",
        "",
        "class Class_ImplicitSphere(object):",
        "    def __init__(self):",
        "        self.position = QtGui.QVector3D()",
        "",
        "    def process(self, context, inputs):",
        "        shader = context.current_shader.program",
        "        ",
        "        radius = inputs[0].process(context).value",
        "        shader_radius_location = shader.uniformLocation(\"f5c6a538-1dbc-4add-a15d-ddc4a5e553da\")",
        "        shader.setUniformValue(shader_radius_location, radius)",
        "        ",
        "        position = self.position",
        "        shader_position_location = shader.uniformLocation(",
        "            \"16d90b34-a728-4caa-b07d-a3244ecc87e3-position\"",
        "        )",
        "        shader.setUniformValue(shader_position_location, position)",
        "        ",
        "        return context"
    ]
}◇
```

Fragment referenced in 57a.

Connections are composed of an input and an output plus a reference to a part, as stated in . In this ⌐Add reference
case there is exactly one input, the radius, and one output, an object defined by implicit functions.

The radius is being defined by an input, which is therefore being referenced as source. There is although
no external node being referenced, as the radius is of the atomic type float. Therefore the source node is
0, meaning it is an internal reference. The input itself is used as part for the input.

The very same applies for the output of that connection. The radius is being consumed by the first part
of the node's part (which has only this part). As this definition is within the same node, the target node
is also 0. The part is then being referenced by its identifier.

⟨*Implicit sphere node connections* 59b⟩ ≡

```
{
    "source_node": "00000000-0000-0000-0000-000000000000",
    "source_part": "f5c6a538-1dbc-4add-a15d-ddc4a5e553da",
    "target_node": "00000000-0000-0000-0000-000000000000",
    "target_part": "74b73ce7-8c9d-4202-a533-c77aba9035a6"
}◇
```

Fragment referenced in 57a.

Now a very basic node is avaialble, but the node does not get recognized by the application yet. As nodes
are defined by external files, they need to be searched, loaded and registered to make them available to
the application.

Therefore the node controller is introduced, which will manage the node definitions.

⟨*Node controller declarations* 60a⟩ ≡

```
    common.with_logger
    class NodeController(object):
        """The node controller.

        A controller managing nodes.
        """

        # Constants
        NODES_PATH = "nodes"
        NODES_EXTENSION = "node"
        ROOT_NODE_ID = uuid.UUID("026c04d0-36d2-49d5-ad15-f4fb87fe8eeb")
        ROOT_NODE_OUTPUT_ID = uuid.UUID("a8fadcfc-4e19-4862-90cf-a262eef2219b")

        # Signals
```
⟨ *Node controller signals ?* ⟩

⟨ *Node controller constructor* 60b, . . . ⟩
⟨ *Node controller methods* 60c, . . . ⟩

⟨ *Node controller slots ?* ⟩
    ◇

Fragment referenced in 102b.

The node controller assumes, that all node definitions are placed within the `nodes` subdirectory of the
application's working directory. Further it assumes, that node definition files use the `node` extension.

⟨*Node controller constructor* 60b⟩ ≡

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

        self.nodes_path = "{current_dir}{sep}{nodes_path}".format(
            current_dir=os.getcwd(),
            sep=os.sep,
            nodes_path=NodeController.NODES_PATH
        )
        self.nodes_extension = NodeController.NODES_EXTENSION◇
```
Fragment defined by 60b, 61c, 68b, 82.
Fragment referenced in 60a.

The node controller will then scan that directory containing the node definitions and load each one.

⟨*Node controller methods* 60c⟩ ≡

```
    def load_nodes(self):
        """Loads all files with the ending NodeController.NODES_EXTENSION
        within the NodeController.NODES_PATH directory, relative to the current
        working directory.
        """
```
⟨ *Node controller load nodes method* 67a, . . . ⟩◇

Fragment defined by 60c, 70, 73, 81.
Fragment referenced in 60a.

Node definitons will contain parts. The parts within node definition are used to create corresponding parts within instances of themselves. The parts are able to create values based on the atomic types through functions.

⟨*Node definition part domain model declarations* 61a⟩ ≡

```python
class NodeDefinitionPart(object):
    """Represents a part of the definition of a node."""

    # Signals
⟨ Node definition part domain model signals ? ⟩

⟨ Node definition part domain model constructor 61b ⟩
⟨ Node definition part domain model methods ? ⟩◇
```
Fragment referenced in 99a.

The part of a node definition holds an identifier as well as an expression to create a function for creating and handling values which will be used when evaluating a node. Further it provides a function which allows to instantiate itself as part of a node (instance).

⟨*Node definition part domain model constructor* 61b⟩ ≡

```python
def __init__(self, id_):
    """Constructor.

    :param id_: the globally unique identifier of the part of the node
                definition.
    :type  id_: uuid.uuid4
    """

    self.id_   = id_
    self.type_ = None
    self.name  = None
    self.parent = None

    # This property is used when evaluating node instances using this node
    # definition
    self.function_creator = lambda: create_value_function(
        parameter.FloatValue(0)
    )

    # This property will be used to create/instantiate a part of a node
    # instance
    self.creator_function = None
◇
```
Fragment referenced in 61a.

The node controller needs to keep track of node defintion parts, as they are a central aspect and may be reused.

⟨*Node controller constructor* 61c⟩ ≡

```python
    self.node_definition_parts = {}
◇
```
Fragment defined by 60b, 61c, 68b, 82.
Fragment referenced in 60a.

The code snippet defining the constructor of a node definition part uses a function called `create_value_function` of the `functions` module.

⟨*Node domain module methods* 62⟩ ≡

```python
def create_value_function(value):
    """Creates a new value function using the provided value.

    :param value: the value which the function shall have.
    :type  value: qde.editor.domain.parameter.Value
    """

    value_function = NodePart.ValueFunction()
    value_function.value = value.clone()

    return value_function
```

◇

Fragment defined by 62, 78a.
Fragment referenced in 99a.

That brings up the concept of value functions. Value functions are one of the building blocks of a node. They are used to evaluate a node value-wise through its inputs.

⟨*Node part domain model value function declarations* 63 ⟩ ≡

```python
class ValueFunction(Function):
    """Class representing a value function for nodes."""

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

        super(NodePart.ValueFunction, self).__init__()
        self.value = None

    def clone(self):
        """Clones the currently set value function.

        :return: a clone of the currently set value function.
        :rtype: qde.editor.domain.node.NodePart.Function
        """

        new_function = create_value_function(self.value)
        new_function.node_part = self.node_part

        return new_function

    def process(self, context, inputs, output_index):
        """Processes the value function for the given context, the given inputs
        and the given index of the output.

        :param context: the context of the processing
        :type  context: qde.editor.domain.node.NodePartContext
        :param inputs: a list of inputs to process
        :type  inputs: list
        :param output_index: the index of the output which shall be used
        :type  output_index: int

        :return: the context
        :rtype: qde.editor.domain.node.NodePartContext
        """

        if not self.value.is_cachable or self.has_changed:
            if len(inputs) > 0:
                inputs[0].process(context, self.processing_index)
                value.set_value_from_context(context)
            else:
                self.value.set_value_in_context(context)

            self.has_changed = False
        else:
            self.value.set_value_in_context(context)

        # TODO: Handle events

        return context◇
```
Fragment referenced in 65a.

During the initialization of a node, its value function may not be clear or it is simply subject to change. Therefore it makes sense to provide a default value function which gets used by default.

⟨*Node part domain model default value function declarations* 64⟩ ≡

```python
class DefaultValueFunction(ValueFunction):
    """The default value function of a node part."""

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

        super(NodePart.DefaultValueFunction, self).__init__()

    def clone(self):
        """Returns itself as a default value function may not be cloned.

        :return: a self-reference.
        :rtype: DefaultValueFunction
        """

        return self

    def process(self, context, inputs, output_index):
        """Processes the default value function for the given context, the given inputs
        and the given index of the output.

        :param context: the context of the processing
        :type  context: qde.editor.domain.node.NodePartContext
        :param inputs: a list of inputs to process
        :type inputs: list
        :param output_index: the index of the output which shall be used
        :type output_index: int

        :return: the context
        :rtype: qde.editor.domain.node.NodePartContext
        """

        self.value.set_value_in_context(context)
        self.has_changed = False

        return context◇
```

Fragment referenced in 65a.

The value function relies strongly on the conecpt of node parts, which is not defined yet. A part of a node is actually an instance of an atomic type (which is usually an input) within an instance of a node definition.

⟨*Node part domain model declarations* 65a⟩ ≡

```python
class NodePart(object):
    """Represents a part of a node."""
```

⟨ *Node part domain model function declarations* 66a ⟩
⟨ *Node part domain model value function declarations* 63 ⟩
⟨ *Node part domain model default value function declarations* 64 ⟩

```python
    # Signals
```
⟨ *Node part domain model signals* ? ⟩

⟨ *Node part domain model constructor* 65b ⟩
⟨ *Node part domain model methods* ? ⟩
◇

Fragment referenced in 99a.

⟨*Node part domain model constructor* 65b⟩ ≡

```python
def __init__(self, id_, default_function):
    """Constructor.

    :param id_: the identifier of the node part.
    :type  id_: uuid.uuid4
    :param default_function: the default function of the part
    :type  default_function: Function
    """

    self.id_             = id_
    self.function_       = default_function
    self.default_function = default_function
    self.type_           = types.NodeType.GENERIC
```
◇

Fragment referenced in 65a.

A part of a node has a function, which gets called whenever a part of a node is being processed.

⟨*Node part domain model function declarations* 66a⟩ ≡

```python
class Function(object):
    """Represents the function of a part of a node."""

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

        self.has_changed = True
        self.evaluation_index = 0
        self.changed_state = types.StateChange.VALUE.value | types.StateChange.SUBTREE.value

    def clone(self):
        """Clones the currently set function."""

        message = QtCore.QCoreApplication.translate(
            __class__.__name__,
            "This method must be implemented in a child class"
        )
        raise NotImplementedError(message)

    def process(self, context, inputs, output_index):
        """Processes the value function for the given context, the given
        inputs."""

        message = QtCore.QCoreApplication.translate(
            __class__.__name__,
            "This method must be implemented in a child class"
        )
        raise NotImplementedError(message)
```
◇

Fragment referenced in 65a.

When a part of a node is being processed, also its inputs are processed. Whenever an input (value) changes, the node part needs to handle the changes. There are three possible types of changes: nothing has changed, the value (of the function) has changed or the subtree (inputs) has changed.

⟨*Node part state changed declarations* 66b⟩ ≡

```python
class StateChange(enum.Enum):
    """Possible changes of state."""

    NOTHING = 0
    VALUE   = 1
    SUBTREE = 2
```◇
Fragment referenced in 93c.

At the end, all (end-) nodes will be composed of parts of atomic type. When building the node definition from the JSON input, the (atomic) part of the node definition is fetched from the node controller. Therefore it is necessary to provide parts for the atomic types before loading all the node definitions.

⟨*Node controller load nodes method* 67a⟩ ≡

```
    for atomic_type in parameter.AtomicTypes.atomic_types:
        if atomic_type.id_ not in self.node_definition_parts:
            self.node_definition_parts[atomic_type.id_] = atomic_type
            self.logger.info(
                "Added atomic type %s: %s",
                atomic_type.type_, atomic_type.id_
            )
        else:
            self.logger.warn((
                "Already knowing node part for atomic type %s. This should not"
                "happen"
            ), atomic_type.type_)◇
```
Fragment defined by 67a, 69.
Fragment referenced in 60c.

Having the atomic types avaialble as parts, the node definitions themselves may be loaded. There is only one problem to that: there is nothing to hold the node defintions. Therefore the node definition domain model is introduced.

⟨*Node definition domain model declarations* 67b⟩ ≡

```
    class NodeDefinition(object):
        """Represents the definition of a node."""

        # Signals
⟨ Node definition domain model signals ? ⟩

⟨ Node definition domain model constructor 68a ⟩
⟨ Node definition domain model methods 83 ⟩◇
```
Fragment referenced in 99a.

The definition of a node is quite similar to a node itself. As the definiton of a node may be changed, the flag `was_changed` is added. Further a node definition holds all instances of itself, meaning nodes.

⟨*Node definition domain model constructor* 68a⟩ ≡

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

        :param id_: the globally unique identifier of the node.
        :type id_: uuid.uuid4
        """

        self.id_         = id_

        self.name        = ""
        self.description = ""
        self.parent      = None
        self.inupts      = []
        self.outputs     = []
        self.definitions = []
        self.invocations = []
        self.parts       = []
        self.nodes       = []
        self.connections = []
        self.instances   = []
        self.was_changed = False◇
```
Fragment referenced in 67b.

Now the controller is able to instantiate nodes definitions and keep them in a list.

⟨*Node controller constructor* 68b⟩ ≡

```
        self.node_definitions = {}◇
```
Fragment defined by 60b, 61c, 68b, 82.
Fragment referenced in 60a.

The controller scans the `node` subdirectory, containing the node definitions, for files ending in `node`.

⟨*Node controller load nodes method* 69⟩ ≡

```python
if os.path.exists(self.nodes_path):
    node_definition_files = glob.glob("{path}{sep}*.{ext}".format(
        path=self.nodes_path,
        sep=os.sep,
        ext=self.nodes_extension
    ))
    num_node_definitions = len(node_definition_files)
    if num_node_definitions > 0:
        self.logger.info(
            "Found %d node definition(s), loading.",
            num_node_definitions
        )
        t0 = time.perf_counter()
        for file_name in node_definition_files:
            self.logger.debug(
                "Found node definition %s, trying to load",
                file_name
            )
            node_definition = self.load_node_definition_from_file_name(file_name)
            if node_definition is not None:
                self.node_definitions[node_definition.id_] = node_definition
        t1 = time.perf_counter()
        self.logger.info(
            "Loading node definitions took %.10f seconds",
            (t1 - t0)
        )
    else:
        message = QtCore.QCoreApplication.translate(
            __class__.__name__, "No node definitions found."
        )
        self.logger.warn(message)
else:
    message = QtCore.QCoreApplication.translate(
        __class__.__name__, "No node definitions found."
    )
    self.logger.warn(message)
```
◇

Fragment defined by 67a, 69.
Fragment referenced in 60c.

If such a file is found, its identifier is extracted from the file name. If the node definition is not known yet, it gets loaded and added to the list of known node definitions.

```python
def load_node_definition_from_file_name(self, file_name):
    """Loads a node definition from the given file name.
    If no such file exists, an FileNotFoundError is raised.

    :param file_name: the file name to load.
    :type  file_name: str

    :return: the loaded node definition and its identifier or None
    :rtype: qde.editor.domain.node.NodeDefinition or None
    """

    if not os.path.exists(file_name):
        self.logger.warn((
            "Tried to load node defintion from file %s, "
            "but the file does not exist"
        ), file_name)
        return None

    # Extract the defintion identifier from the file name, which is
    # "uuid4.node".
    definition_id = os.path.splitext(os.path.basename(file_name))[0]

    if definition_id in self.node_definitions:
        self.logger.warn(
            "Should load node definition from file %s, but is already loaded",
            file_name
        )
        return self.node_definitions[definition_id]

    try:
        with open(file_name) as definition_fh:
            node_definition = json.Json.load_node_definition(
                self, definition_fh
            )
            self.logger.debug(
                "Loaded node definition %s from file %s",
                definition_id, file_name
            )
            # TODO: Trigger (loading) callback
            return node_definition
    except json.json.decoder.JSONDecodeError as exc:
        self.logger.warn(
            "There was an error loading the node definition %s: %s",
            definition_id, exc
        )
        return None◇
```

Fragment defined by 60c, 70, 73, 81.
Fragment referenced in 60a.

The loading of the node definiton is simply about parsing the various sections and handling them correspondingly. To prevent the node controller from being bloated, the parsing is done in a separate module responsible for JSON specific tasks.

```python
    classmethod
    def load_node_definition(cls, node_controller, json_file_handle):
        """Loads a node definition from given JSON input.

        :param node_controller: reference to the node controller
        :type  node_controller: qde.editor.application.node.NodeController
        :param json_file_handle: an open file handle containing JSON data
        :type  json_file_handle: file

        :return: a node definition
        :rtype: qde.editor.domain.node.NodeDefinition
        """

        o = json.load(json_file_handle)

        name        = str(o['name'])
        id_         = uuid.UUID(o['id_'])
        description = str(o['description'])

        inputs = []
        for input in o['inputs']:
            node_definition_input = cls.build_node_definition_input(
                node_controller, input
            )
            inputs.append(node_definition_input)

        outputs = []
        for output in o['outputs']:
            node_definition_output = cls.build_node_definition_output(
                node_controller, output
            )
            outputs.append(node_definition_output)

        node_definitions = {}
        for node_def in o['nodes']:
            definition_id, node_definition = cls.build_node_definition(node_def)
            node_definitions[definition_id] = node_definition

        connections = []
        for conn in o['connections']:
            connection = cls.build_node_definition_connection(conn)
            connections.append(connection)

        definitions = []
        for d in o['definitions']:
            definition = cls.build_node_definition_definition(d)
            definitions.append(definition)

        invocations = []
        for i in o['invocations']:
            invocation = cls.build_node_definition_invocation(i)
            invocations.append(invocation)

        node_definition             = node.NodeDefinition(id_)
        node_definition.name        = name
        node_definition.description = description
        node_definition.inputs      = inputs
        node_definition.outputs     = outputs
        node_definition.nodes       = node_definitions
        node_definition.connections = connections
        node_definition.definitions = definitions
        node_definition.invocations = invocations

        # TODO: Check if this part can be abve the def. instance
        parts = []
        for p in o['parts']:
            part = cls.build_node_definition_part(node_controller, node_definition, p)
            parts.append(part)
```

As can be seen in the above listing, there are parts, that are not yet defined: inputs, outputs, other node definitions, connections, definitions, invocations and parts.

First the building of the node definition inputs is defined.

⟨JSON methods 72⟩ ≡

```python
classmethod
def build_node_definition_input(cls, node_controller, json_input):
    """Builds and returns a node definition input from the given JSON input
    data.

    :param node_controller: a reference to the node controller
    :type  node_controller: qde.editor.application.node.NodeController
    :param json_input: the input in JSON format
    :type  json_input: dict

    :return: a node definition input
    :rtype: qde.editor.domain.node.NodeDefinitionInput
    """

    input_id            = uuid.UUID(json_input['id_'])
    name                = str(json_input['name'])
    atomic_id           = uuid.UUID(json_input['atomic_id'])
    description         = str(json_input['description'])
    node_definition_part = node_controller.get_node_definition_part(atomic_id)

    default_value_str = ""
    default_value_entry = json_input['default_value']
    default_value = parameter.create_value(
        default_value_entry['type_'],
        default_value_entry['value']
    )

    min_value = float(json_input['min_value'])
    max_value = float(json_input['max_value'])

    node_definition_input = node.NodeDefinitionInput(
        input_id,
        name,
        node_definition_part,
        default_value
    )
    node_definition_input.description = description
    node_definition_input.min_value = min_value
    node_definition_input.max_value = max_value

    cls.logger.debug(
        "Built node definition input for node definition %s",
        atomic_id
    )
    return node_definition_input
```
◇

Fragment defined by 71, 72, 78b, 80, 84a, 104a, 105a, 106a.
Fragment referenced in 103a.

However, there are a few things missing, which are used in the above code fragments. The possibility to create values from given parameters, the actual node definition input as domain model and getting the node definition part identified by the given atomic identifier.

⟨*Node controller methods* 73⟩ ≡

```python
    def get_node_definition_part(self, id_):
        """Returns the node definition part identified by the given identifier.

        If no such part is available, a generic part with that identifier is being
        created.

        :param id_: the identifier of the part of the node definition to get.
        :type  id_: uuid.uuid4

        :return: the node definition part identified by the given identifier.
        :rtype: qde.editor.domain.node.NodeDefinitionPart
        """

        if str(id_) not in self.node_definition_parts:
            self.logger.warn((
                "Part %s of the node definition was not found. Creating a"
                "generic one."
            ), id_)

            type_ = types.NodeType.GENERIC
            def create_func(id_, default_function, name, type_):
                node_part = node.NodePart(id_, None)
                node_part.type_ = type_
                node_part.name = name
                return node_part
            node_definition_part = node.NodeDefinitionPart(id_)
            node_definition_part.type_ = type_
            node_definition_part.creator_function = create_func
            self.node_definition_parts[id_] = node_definition_part
            return node_definition_part
        else:
            return self.node_definition_parts[str(id_)]
```
◇

Fragment defined by 60c, 70, 73, 81.
Fragment referenced in 60a.

The creation of values from given parameters is done within the parameter module, as this is something very parameter specific. Therefore a static method is defined, which returns an instance of an atomic type, e.g. a float value or a scene. ⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯ instance of a

⟨Parameter domain module methods 74⟩ ≡

```python
def create_value(type_, value_string):
    """Creates an object of the given type with the given value.

    :param type_: the type of the value to create.
    :type  type_: str
    :param value_string: the value that the value shall have.
    "type value_string: str

    :return: a value-type of the given type with the given value.
    :rtype: qde.editor.domain.parameter.Value
    """

    if type_.lower() == "float":
        float_value = float(value_string)
        return FloatValue(float_value)
    elif type_.lower() == "text":
        return TextValue(value_string)
    elif type_.lower() == "image":
        return ImageValue()
    elif type_.lower() == "scene":
        return SceneValue()
    elif type_.lower() == "generic":
        return GenericValue()
    elif type_.lower() == "dynamic":
        return DynamicValue()
    elif type_.lower() == "mesh":
        return MeshValue()
    elif type_.lower() == "implicit":
        return ImplicitValue()
    else:
        message = QtCore.QCoreApplication.translate(
            __module__.__name__, "Unknown type for value provided"
        )
        raise Exception(message)◇
```

Fragment referenced in 94.

Further the instanciable classes of the atomic types are defined at the very same place.

First, a generic value interface is defined. This interface holds a refernce to the atomic type of the value and defines what type the function of a value is.

⟨*Paramater domain model value generic interface* 75a⟩ ≡

```python
class ValueInterface(object):
    """Generic value interface."""

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

        self.function_type = None

    def clone(self):
        """Clones the currently set value.

        :return: a clone of the currently set value
        :rtype: qde.editor.domain.parameter.ValueInterface
        """

        message = QtCore.QCoreApplication.translate(
            __module__.__name__,
            "This method must be implemented in a child class"
        )
        raise NotImplementedError(message)◇
```
Fragment referenced in 94.

Then an interface for setting and getting values is defined.

⟨*Paramater domain model value interface* 75b⟩ ≡

```python
class Value(ValueInterface):
    """Value interface for setting and getting values."""

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

        :param value: the value that shall be held
        :type  value: object
        """

        super(Value, self).__init__()
        self.value = value◇
```
Fragment referenced in 94.

Then the specific value types are implemented, based either on the generic or the concrete value interface, depending on the type. Here just two implementations are given as an example. The other implementations can be found at . link to fragm

⟨*Paramater domain model float value* 76a⟩ ≡

```python
# Python
class FloatValue(Value):
    """A class holding float values."""

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

        :param float_value: the float value that shall be held
        :type  float_value: float
        """

        super(FloatValue, self).__init__(float_value)
        self.function_type = types.NodeType.FLOAT

    def clone(self):
        """Clones the currently set value.

        :return: a clone of the currently set value
        :rtype: qde.editor.domain.parameter.ValueInterface
        """

        return FloatValue(self.value)◇
```

Fragment referenced in 94.

⟨*Paramater domain model scene value* 76b⟩ ≡

```python
class SceneValue(ValueInterface):
    """A class holding scene values."""

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

        super(SceneValue, self).__init__()
        self.function_type = types.NodeType.SCENE

    def clone(self):
        """Clones the currently set value.

        :return: a clone of the currently set value
        :rtype: qde.editor.domain.parameter.ValueInterface
        """

        return SceneValue()◇
```

Fragment referenced in 94.

What now still is missing, is the definition of the node definition input domain model.

⟨*Node definition input domain model declarations* 77a⟩ ≡

```python
class NodeDefinitionInput(object):
    """Represents an input of a definition of a node."""

    # Signals
```
⟨ *Node definition input domain model signals ?* ⟩

⟨ *Node definition input domain model constructor* 77b ⟩
⟨ *Node definition input domain model methods ?* ⟩◇

Fragment referenced in 99a.

⟨*Node definition input domain model constructor* 77b⟩ ≡

```python
def __init__(self, id_, name, node_definition_part, default_value):
    """Constructor.

    :param id_: the identifier of the definition
    :type  id_: uuid.uuid4
    :param name: the name of the definition
    :type  name: str
    :param node_definition_part: the atomic part of the node definition
    :type node_definition_part: TODO
    :param default_value: the default value of the input
    :type default_value: qde.editor.domain.parameter.Value
    """

    self.id_               = id_
    self.name              = name
    self.node_definition_part = node_definition_part
    self.description       = ""
    self.min_value         = -100000
    self.max_value         = 100000

    self.default_function = create_default_value_function(
        default_value
    )◇
```

Fragment referenced in 77a.

The code snippet defining the constructor of a node definition input uses a function called `create_default_value_function`
of the `functions` module. This function creates a default value function based on the given default
value.

```python
    def create_default_value_function(value):
        """Creates a new default value function using the provided value.

        :param value: the value which the function shall have.
        :type  value: qde.editor.domain.parameter.Value
        """

        value_function = NodePart.DefaultValueFunction()
        value_function.value = value.clone()

        return value_function◇
```

With this implementation all the parts needed for creating and handling node definition inputs are defined, which leads to the next implementation. The outputs of a node definition. The outputs are in the same way implemented as the inputs of a node definition.

```python
    classmethod
    def build_node_definition_output(cls, node_controller, json_input):
        """Builds and returns a node definition output from the given JSON input
        data.

        :param node_controller: a reference to the node controller
        :type  node_controller: qde.editor.application.node.NodeController
        :param json_input: the input in JSON format
        :type  json_input: dict

        :return: a node definition output
        :rtype: qde.editor.domain.node.NodeDefinitionOutput
        """

        output_id            = uuid.UUID(json_input['id_'])
        name                 = str(json_input['name'])
        atomic_id            = uuid.UUID(json_input['atomic_id'])
        node_definition_part = node_controller.get_node_definition_part(atomic_id)

        node_definition_output = node.NodeDefinitionOutput(
            output_id,
            name,
            node_definition_part
        )

        cls.logger.debug(
            "Built node definition output for node definition %s",
            atomic_id
        )
        return node_definition_output
    ◇
```

The domain model of the node definition output is very similar to the input, has less attributes although.

⟨*Node definition output domain model declarations* 79a⟩ ≡

```python
class NodeDefinitionOutput(object):
    """Represents an output of a definition of a node."""

    # Signals
```
⟨ *Node definition output domain model signals* ? ⟩

⟨ *Node definition output domain model constructor* 79b ⟩
⟨ *Node definition output domain model methods* ? ⟩◇

Fragment referenced in 99a.

⟨*Node definition output domain model constructor* 79b⟩ ≡

```python
def __init__(self, id_, name, node_definition_part):
    """Constructor.

    :param id_: the identifier of the definition
    :type  id_: uuid.uuid4
    :param name: the name of the definition
    :type  name: str
    :param node_definition_part: the atomic part of the node definition
    :type  node_definition_part: qde.editor.domain.node.NodeDefinitionPart
    """

    self.id_                = id_
    self.name               = name
    self.node_definition_part = node_definition_part◇
```
Fragment referenced in 79a.

As a node definition may contain references to other node defintions, it has to parse them. The parsing is similar to that of the inputs and outputs.

⟨*JSON methods* 80⟩ ≡

```
    classmethod
    def build_node_definition(cls, node_controller, json_input):
        """Builds and returns a node definition from the given JSON input data.

        :param node_controller: a reference to the node controller
        :type  node_controller: qde.editor.application.node.NodeController
        :param json_input: the input in JSON format
        :type  json_input: dict

        :return: a dictionary containg the node definition at the index of the
                 definition identifier.
        :rtype: dict
        """

        definition_id = uuid.UUID(json_input['id_'])
        atomic_id     = uuid.UUID(json_input['atomic_id'])
        node_definition = node_controller.get_node_definition(atomic_id)

        cls.logger.debug(
            "Built node definition for node definition %s",
            atomic_id
        )
        return (definition_id, node_definition)
```
    ◇
Fragment defined by 71, 72, 78b, 80, 84a, 104a, 105a, 106a.
Fragment referenced in 103a.

As can be seen in the above code fragment, the node definition is returned by the node controller. This is very similar to getting the node definition part from the node controller.

⟨*Node controller methods* 81⟩ ≡

```python
    def get_node_definition(self, id_):
        """Returns the node definition identified by the given identifier.

        If no such definition is available, it will be tried to load the
        definition. If this is not possible as well, None will be returned.

        :param id_: the identifier of the node definition to get.
        :type id_: uuid.uuid4

        :return: the node definition identified by the given identifier or None.
        :rtype: qde.editor.domain.node.NodeDefinition or None
        """

        self.logger.debug(
            "Getting node definition %s",
            id_
        )

        if str(id_) in self.node_definitions:
            return self.node_definitions[str(id_)]
        elif self.root_node is not None and id_ == self.root_node.id_:
            return self.root_node
        else:
            # The node definition was not found, try to load it from node
            # definition files.
            file_name = os.path.join(
                self.nodes_path,
                id_,
                self.nodes_extension
            )
            node_definition = self.load_node_definition_from_file_name(
                file_name
            )
            if node_definition is not None:
                self.node_definitions[node_definition.id_] = node_definition
                return node_definition
            else:
                return None◇
```

Fragment defined by 60c, 70, 73, 81.
Fragment referenced in 60a.

As can be seen in the above code snippet, the node controller holds the root node, which is placed within the root scene.

⟨*Node controller constructor* 82⟩ ≡

```python
        # TODO: Load from coonfiguration?
        self.root_node = node.NodeDefinition(NodeController.ROOT_NODE_ID)
        self.root_node.name = QtCore.QCoreApplication.translate(
            __class__.__name__,
            'Root'
        )
        root_node_output = node.NodeDefinitionOutput(
            NodeController.ROOT_NODE_OUTPUT_ID,
            QtCore.QCoreApplication.translate(
                __class__.__name__,
                'Output'
            ),
            parameter.AtomicTypes.Generic
        )
        self.root_node.add_output(root_node_output)
        self.logger.debug("Created root node %s", NodeController.ROOT_NODE_ID)◇
```

Fragment defined by 60b, 61c, 68b, 82.
Fragment referenced in 60a.

Currently there is no possiblity to add outputs to a node definition. Adding an ouptut simply adds that output to the list of outputs the node definition has. Furthermore that output needs to added for each instance of that node definition as well. ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ Add inputs a

⟨*Node definition domain model methods* 83⟩ ≡

```python
    def add_output(self, node_definition_output):
        """Adds the given output to the beginning of the list of outputs and
        also to all instances of this node definition.

        :param node_definition_output: the output to add.
        :type  node_definition_output: qde.editor.domain.node.NodeDefinitionOutput
        """

        self.add_output_at(len(self.outputs), node_definition_output)

    def add_output_at(self, index, node_definition_output):
        """Adds the given output to the list of outputs at the given index
        position and also to all instances of this node definition.

        :param index: the position in the list of outputs where the new output
                      shall be added at.
        :type  index: int
        :param node_definition_output: the output to add.
        :type  node_definition_output: qde.editor.domain.node.NodeDefinitionOutput

        :raise: an index error when the given index is not valid.
        :raises: IndexError
        """

        if index < 0 or index > len(self.outputs):
            raise IndexError()

        self.outputs.insert(index, node_definition_output)

        for instance in self.instances:
            instance.add_output_at(
                index,
                node_definition_output.create_instance()
            )

        # TODO: Insert connection if output is atomic

        self.was_changed = True◇
```
Fragment referenced in 67b.

Having the reading and parsing of inputs, outputs and other node definition implemented, the reading and parsing of connections, definitions, invocations and parts still remains.

The reading and parsing of connections, definitions and invocation is very straightforward and very similar to the one of the node definitions. Therefore it will not be shown in detail. Details are found at

Add reference to code fragments here

.

The last part when loading a node definition, is reading and parsing the code part of the node.

```python
    classmethod
    def build_node_definition_part(cls, node_controller, parent, json_input):
        """Builds and returns a node definition part from the given JSON input data.

        :param node_controller: a reference to the node controller
        :type  node_controller: qde.editor.application.node.NodeController
        :param parent: the parent of the node definition part
        :type  parent: qde.editor.domain.node.NodeDefinition
        :param json_input: the input in JSON format
        :type  json_input: dict

        :return:
        :rtype:
        """

        part_id        = uuid.UUID(json_input['id_'])
        name           = str(json_input['name'])

        script_lines = []
        for script_line in json_input['script']:
            script_lines.append(str(script_line))
        script = "\n".join(script_lines)

        type_string = json_input['type_']
        type_ = types.NodeType[type_string.upper()]

        node_definition_part = node.NodeDefinitionPart(part_id)
        node_definition_part.name = name
        node_definition_part.type_ = type_
        node_definition_part.parent = parent

        node_controller.node_definition_parts[part_id] = node_definition_part

        cls.logger.debug(
            "Built part for node definition %s",
            part_id
        )
        return node_definition_part
```
   ◇

Fragment defined by 71, 72, 78b, 80, 84a, 104a, 105a, 106a.
Fragment referenced in 103a.

Finally the node controller needs to be instantiated by the main application and the loading of the node definitions needs to be triggered.

⟨ₘSet up controllers for main application 84b⟩ ≡

```python
    self.node_controller = node.NodeController()
    self.node_controller.load_nodes()◇
```
Fragment defined by 29b, 52d, 84b.
Fragment referenced in 17b.

Now node definitions are being loaded and parsed. Although there is no possiblity to select and instantiate the node definitons yet.

To allow the instantiation of nodes, a (user interface) component is necessary. It will access all the loaded nodes and provide an interface for selecting a node definition which then will be instantiated.

⟨*Add node dialog declarations* 85a⟩ ≡

```
common.with_logger
class AddNodeDialog(QtWidgets.QDialog):
    """Class for adding nodes to a scene view."""

    class Column(object):
        """Class representing a column within the add node dialog."""

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

            self.frame       = None
            self.sub_frames  = []
            self.label       = None
            self.v_box_layout = None

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

        :param parent: the parent of this dialog.
        :type  parent: QtGui.QWidget
        """

        super(AddNodeDialog, self).__init__(parent)
        self.columns = {}
        self.chosen_node_definition = None

        self.setWindowTitle("Add node")

        column_layout = QtWidgets.QHBoxLayout(self)
        column_layout.setContentsMargins(0, 0, 0, 0)
        column_layout.setSizeConstraint(Qt.QLayout.SetFixedSize)
        self.setLayout(column_layout)◇
```

Fragment referenced in 106b.

The dialog for adding a node instance from a node defintion shall only be shown from within a scene, that is from within the scene view. Therefore the add node dialog is added to the scene view.

⟨*Scene view constructor* 85b⟩ ≡

```
        self.add_node_dialog = node.AddNodeDialog(self)
    ◇
```

Fragment defined by 51a, 85b.
Fragment referenced in 50d.

The dialog for adding a node shall be shown as soon as the scene view is focussed and the tabulator key is being pressed.

5.2 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.
he document initially.

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   M
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   T
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   V
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

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

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

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

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

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 50the 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

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

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

Re-think how to implement node definitions.

2017-03-22

Re-think how to implement node definitions and nodes. Begin adding notes about how to implement nodes.

2017-03-23

Expand notes about the node implementation, begin writing the actual node implementation down, keep the work log up to date.

2017-03-24

Attend a meeting with Prof. Fuhrer, change and expand the chapter about node implementation according to the before made thoughts, begin implementing the node graph structure, keep the work log up to date.

5.3 Code fragments

`"../src/editor.py"` 88a≡

```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
```

⟨ *Main entry point* 14 ⟩
◇

`"../src/qde/editor/application/application.py"` 88b≡

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

"""Main application module for the QDE editor."""

# System imports
import logging
import logging.config
import os
import json
from PyQt5 import Qt
from PyQt5 import QtCore
from PyQt5 import QtGui
from PyQt5 import QtWidgets

# Project imports
from qde.editor.foundation import common
from qde.editor.application import node
from qde.editor.application import scene
from qde.editor.gui import main_window as qde_main_window
```

⟨ *Main application declarations* 15a ⟩
◇

```
"../src/qde/editor/gui/main_window.py" 89a≡
    #!/usr/bin/python
    # -*- coding: utf-8 -*-

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

    # System imports
    from PyQt5 import Qt
    from PyQt5 import QtCore
    from PyQt5 import QtGui
    from PyQt5 import QtWidgets

    # Project imports
    from qde.editor.foundation import common
    from qde.editor.gui import scene as guiscene


    ⟨ Main window declarations 16a ⟩
    ◇
```

```
"../src/qde/editor/domain/scene.py" 89b≡
    #!/usr/bin/python
    # -*- coding: utf-8 -*-

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

    # System imports
    import uuid
    from PyQt5 import Qt
    from PyQt5 import QtCore

    # Project imports


    ⟨ Scene model declarations 20a ⟩
    ◇
```

```
"../src/qde/editor/gui_domain/scene.py" 89c≡

    #!/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

    # Project imports
    from qde.editor.foundation import common

    ⟨ Scene graph view model declarations 21a ⟩
    ⟨ Scene view model declarations 53a ⟩
    ◇
```

⟨*Scene graph view model methods* 90a⟩ ≡

```python
    def __str__(self):
        """Return the string representation of the current object."""

        return str(self.id_)[0:8]
```
◇

Fragment defined by 29a, 90a.
Fragment referenced in 21a.

⟨*Scene view model methods* 90b⟩ ≡

```python
    def __str__(self):
        """Return the string representation of the current object."""

        return str(self.id_)[0:8]
```
◇

Fragment defined by 54a, 90b.
Fragment referenced in 53a.

"../src/qde/editor/application/scene.py" 90c≡

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

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

# System imports
from PyQt5 import Qt
from PyQt5 import QtCore

# Project imports
from qde.editor.foundation import common
from qde.editor.domain    import scene as domain_scene
from qde.editor.gui_domain import scene as guidomain_scene
```

⟨ *Scene graph controller declarations* 22a ⟩
⟨ *Scene controller declarations* 52c ⟩
◇

"../src/qde/editor/gui/scene.py" 91≡
```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

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

# System imports
from PyQt5 import Qt
from PyQt5 import QtCore
from PyQt5 import QtWidgets

# Project imports
from qde.editor.foundation import common
from qde.editor.gui_domain import scene
from qde.editor.gui import node
```
⟨ *Scene graph view declarations* 29c ⟩
⟨ *Scene view declarations* 50d ⟩
◇

```
"../logging.json" 92≡
    {
        "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"
        }
    }◇
```

```
"../src/qde/editor/foundation/common.py" 93a≡
    #!/usr/bin/python
    # -*- coding: utf-8 -*-

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

    # System imports
    import logging
    from PyQt5 import Qt
    from PyQt5 import QtCore
    from PyQt5 import QtWidgets

    # 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.
        """
```

⟨ *Set logger name* 39a ⟩
⟨ *Logger interface* 39b ⟩
◇

⟨ₘ*Scene graph view decorators* 93b ⟩ ≡

```
    common.with_logger
```
◇

Fragment referenced in 29c.

```
"../src/qde/editor/foundation/type.py" 93c≡
    # -*- coding: utf-8 -*-
    """Module for type-specific aspects."""

    # System imports
    import enum

    # Project imports

```

⟨ *Node type declarations* 43 ⟩
⟨ *Node part state changed declarations* 66b ⟩
◇

`"../src/qde/editor/domain/parameter.py" 94≡`

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

"""Module for parameter-specific aspects."""

# System imports

# Project imports
from qde.editor.foundation import type as types
from qde.editor.domain import node
```

⟨ *Parameter declarations* 44, . . . ⟩
⟨ *Paramater domain model value generic interface* 75a ⟩
⟨ *Paramater domain model value interface* 75b ⟩
⟨ *Paramater domain model float value* 76a ⟩
⟨ *Paramater domain model text value* 96a ⟩
⟨ *Paramater domain model scene value* 76b ⟩
⟨ *Paramater domain module methods* 74 ⟩
◇

$\langle$ *Parameter declarations* 95 $\rangle \equiv$

```
FloatValue = create_node_definition_part.__func__(
    id_="468aea9e-0a03-4e63-b6b4-8a7a76775a1a",
    type_=types.NodeType.FLOAT
)
Text = create_node_definition_part.__func__(
    id_="e43bdd1b-a895-4bd8-8d5a-b401a63f7a6f",
    type_=types.NodeType.TEXT
)
Scene = create_node_definition_part.__func__(
    id_="bfb47e7text7-1b05-4864-8397-de30bf005ff8",
    type_=types.NodeType.SCENE
)
Image = create_node_definition_part.__func__(
    id_="21fd1960-1307-4b53-b7bf-d08f02757335",
    type_=types.NodeType.IMAGE
)
DynamicValue = create_node_definition_part.__func__(
    id_="68720ae3-8068-43ce-94d8-8705dc3b8bfe",
    type_=types.NodeType.DYNAMIC
)
Mesh = create_node_definition_part.__func__(
    id_="9791d341-b92c-43dd-954a-9d83b9020e43",
    type_=types.NodeType.MESH
)
Implicit = create_node_definition_part.__func__(
    id_="c019271c-35b6-425c-9ff2-a1d893111adb",
    type_=types.NodeType.IMPLICIT
)

atomic_types = [
    FloatValue,
    Text,
    Scene,
    Image,
    DynamicValue,
    Mesh,
    Implicit,
]
```

$\diamond$

Fragment defined by 44, 45a, 95.
Fragment referenced in 94.

⟨*Paramater domain model text value* 96a⟩ ≡

```python
class TextValue(Value):
    """A class holding values for text/string nodes."""

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

        :param string_value: the string value that shall be held
        :type  string_value: str
        """

        super(TextValue, self).__init__(string_value)
        self.function_type = types.NodeType.TEXT

    def clone(self):
        """Clones the currently set value.

        :return: a clone of the currently set value
        :rtype: qde.editor.domain.parameter.ValueInterface
        """

        return TextValue(self.value)◇
```
Fragment referenced in 94.

⟨*Paramater domain model image value* 96b⟩ ≡

```python
class ImageValue(ValueInterface):
    """A class holding values for image nodes."""

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

        super(ImageValue, self).__init__()
        self.function_type = types.NodeType.IMAGE

    def clone(self):
        """Clones the currently set value.

        :return: a clone of the currently set value
        :rtype: qde.editor.domain.parameter.ValueInterface
        """

        return ImageValue()◇
```
Fragment never referenced.

⟨*Paramater domain model generic value* 97a⟩ ≡

```python
class GenericValue(ValueInterface):
    """A class holding values for generic nodes."""

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

        super(GenericValue, self).__init__()
        self.function_type = types.NodeType.GENERIC

    def clone(self):
        """Clones the currently set value.

        :return: a clone of the currently set value
        :rtype: qde.editor.domain.parameter.ValueInterface
        """

        return GenericValue()◇
```
Fragment never referenced.

⟨*Paramater domain model dynamic value* 97b⟩ ≡

```python
class DynamicValue(ValueInterface):
    """A class holding values for dynamic nodes."""

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

        super(DynamicValue, self).__init__()
        self.function_type = types.NodeType.DYNAMIC

    def clone(self):
        """Clones the currently set value.

        :return: a clone of the currently set value
        :rtype: qde.editor.domain.parameter.ValueInterface
        """

        return DynamicValue()◇
```
Fragment never referenced.

⟨*Paramater domain model mesh value* 98a⟩ ≡

```python
class MeshValue(ValueInterface):
    """A class holding values for mesh nodes."""

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

        super(MeshValue, self).__init__()
        self.function_type = types.NodeType.MESH

    def clone(self):
        """Clones the currently set value.

        :return: a clone of the currently set value
        :rtype: qde.editor.domain.parameter.ValueInterface
        """

        return MeshValue()◇
```
Fragment never referenced.

⟨*Paramater domain model implicit value* 98b⟩ ≡

```python
class ImplicitValue(ValueInterface):
    """A class holding values for implicit surface nodes."""

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

        super(ImplicitValue, self).__init__()
        self.function_type = types.NodeType.IMPLICIT

    def clone(self):
        """Clones the currently set value.

        :return: a clone of the currently set value
        :rtype: qde.editor.domain.parameter.ValueInterface
        """

        return ImplicitValue()◇
```
Fragment never referenced.

```
"../src/qde/editor/domain/node.py" 99a≡
    # -*- coding: utf-8 -*-

    """Module for node-specific aspects."""

    # System imports

    # Project imports
    from qde.editor.foundation import type as types
    from qde.editor.foundation import flag

    ⟨ Node domain model declarations 45b ⟩
    ⟨ Node part domain model declarations 65a ⟩
    ⟨ Node definition domain model declarations 67b ⟩
    ⟨ Node definition part domain model declarations 61a ⟩
    ⟨ Node definition input domain model declarations 77a ⟩
    ⟨ Node definition output domain model declarations 79a ⟩
    ⟨ Node definition connection domain model declarations 103b ⟩
    ⟨ Node definition definition domain model declarations 104b ⟩
    ⟨ Node definition invocation domain model declarations 105b ⟩
    ⟨ Node domain module methods 62, . . . ⟩
    ◇
```

```
"../src/qde/editor/gui_domain/node.py" 99b≡
    # -*- coding: utf-8 -*-

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

    # System imports
    from PyQt5 import Qt

    # Project imports
    from qde.editor.foundation import common

    common.with_logger
    ⟨ Node view model declarations 46b ⟩
    ◇
```

⟨ Node view model constructor 99c ⟩ ≡

```
        self.setPos(self.position)
        self.setAcceptHoverEvents(True)
        self.setFlag(QGraphicsObject.ItemIsFocusable)
        self.setFlag(QGraphicsObject.ItemIsMovable)
        self.setFlag(QGraphicsObject.ItemIsSelectable)
        self.setFlag(QGraphicsObject.ItemClipsToShape)
    ◇
```
Fragment defined by 47a, 48c, 99c.
Fragment referenced in 46b.

```
"../src/qde/editor/foundation/flag.py" 100≡
    # -*- coding: utf-8 -*-

    """Module for flag-specific aspects."""

    # System imports
    import enum

    # Project imports


    class NodeStatus(enum.Enum):
        """Statues which a node can have."""

        OK             = 0
        NO_INPUTS      = 1
        WRONG_INPUT    = 2
        INPUT_ERRONEOUS = 3
        INPUT_CYCLIC   = 4
        LINK_MISSING   = 5
        TOO_MANY_INPUTS = 6
    ◇
```

⟨*Node view model methods* 101⟩ ≡

```python
    def boundingRect(self):
        """Return the bounding rectangle of the node.

        :return: the bounding rectangle of the node.
        :rtype: Qt.QRectF
        """

        return Qt.QRectF(
            0, 0, self.width * NodeViewModel.WIDTH, NodeViewModel.HEIGHT
        )

    def create_pixmap(self):
        """Creation of the pixmap (=bitmap, the actual 'image')"""

        image = QImage(self.boundingRect().size().toSize(),
                       QImage.Format_ARGB32_Premultiplied)
        pixmap = QPixmap.fromImage(image)
        pixmap.fill(Qt.transparent)

        rect = self.boundingRect()

        painter = QPainter()
        painter.begin(pixmap)
        painter.setRenderHint(QPainter.Antialiasing)

        # Shape
        path = QPainterPath()
        path.addRect(rect)
        # path.addRoundedRect(rect, 5, 5)
        painter.drawPath(path)

        # Color / gradient
        color = QColor(255, 0, 0, 128)
        color.setHsv(color.hsvHue(), 160, 255)
        color_desaturated = color
        color_desaturated.setHsv(color.hsvHue(), 40, 255)
        top_color = QColor(60, 70, 80)
        if self.status is not flag.NodeStatus.OK:
            top_color = QColor(255, 0, 0)
        gradient_top_color = cmn.multiply_colors(
            top_color, color_desaturated
        )
        gradient_bottom_color = cmn.multiply_colors(
            QColor(110, 120, 130), color_desaturated
        )
        rect_gradient = QLinearGradient(
            QPoint(0.0, 0.0), QPoint(0.0, rect.height())
        )
        rect_gradient.setColorAt(0.0, gradient_top_color)
        rect_gradient.setColorAt(1.0, gradient_bottom_color)

        brush = QBrush(rect_gradient)

        painter.fillPath(path, brush)
        painter.end()

        return pixmap
```

◇

Fragment defined by 47bc, 48b, 49ac, 101.
Fragment referenced in 46b.

`"../nodes/16d90b34-a728-4caa-b07d-a3244ecc87e3.node"` 102a≡
⟨ *Implicit sphere node* 57a ⟩◇

`"../src/qde/editor/application/node.py"` 102b≡

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

""" Module holding node related aspects concerning the application layer.
"""

# System imports
import glob
import os
import time
import uuid
from PyQt5 import Qt
from PyQt5 import QtCore

# Project imports
from qde.editor.foundation import common
from qde.editor.foundation import type as types
from qde.editor.technical import json
from qde.editor.domain import parameter
from qde.editor.domain import node
```

⟨ *Node controller declarations* 60a ⟩
◇

`"../src/qde/editor/technical/json.py"` 102c≡

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

""" Module holding JSON related aspects.
"""

# System imports
import json
import uuid

# Project imports
from qde.editor.foundation import common
from qde.editor.foundation import type as types
from qde.editor.domain import node
from qde.editor.domain import parameter
```

⟨ *JSON module declarations* 103a ⟩
◇

⟨*JSON module declarations* 103a⟩ ≡

```
    common.with_logger
    class Json(object):
        """Class handling JSON relevant tasks.
        """


    ⟨ JSON methods 71, . . . ⟩
    ◇
```
Fragment referenced in 102c.

⟨*Node definition connection domain model declarations* 103b⟩ ≡

```
    class NodeDefinitionConnection(object):
        """Represents a connection of a definition of a node."""


        # Signals
```
⟨ *Node definition connection domain model signals* ? ⟩

```
        def __init__(self,
                     source_node_id, source_part_id,
                     target_node_id, target_part_id):
            """Constructor.

            :param source_node_id: the identifier of the source node.
            :type  source_node_id: uuid.uuid4
            :param source_part_id: the identifier of the part of the source node.
            :type  source_part_id: uuid.uuid4
            :param target_node_id: the identifier of the target node.
            :type  target_node_id: uuid.uuid4
            :param target_part_id: the identifier of the part of the target node.
            :type  target_part_id: uuid.uuid4
            """


            self.source_node_id = source_node_id
            self.source_part_id = source_part_id
            self.target_node_id = target_node_id
            self.target_part_id = target_part_id◇
```
Fragment referenced in 99a.

⟨*JSON methods* 104a⟩ ≡

```python
    classmethod
    def build_node_definition_connection(cls, json_input):
        """Builds and returns a connection for a node definition from the given
        JSON input data.

        :param json_input: the input in JSON format
        :type  json_input: dict

        :return: the connection of a node definition.
        :rtype: qde.editor.domain.node.NodeDefinitionConnection
        """

        source_node_id = uuid.UUID(json_input['source_node'])
        source_part_id = uuid.UUID(json_input['source_part'])
        target_node_id = uuid.UUID(json_input['target_node'])
        target_part_id = uuid.UUID(json_input['target_part'])

        node_definition_connection = node.NodeDefinitionConnection(
            source_node_id,
            source_part_id,
            target_node_id,
            target_part_id
        )

        cls.logger.debug("Built node definition connection")
        return node_definition_connection
```
◇

Fragment defined by 71, 72, 78b, 80, 84a, 104a, 105a, 106a.
Fragment referenced in 103a.

⟨*Node definition definition domain model declarations* 104b⟩ ≡

```python
    class NodeDefinitionDefinition(object):
        """Represents a definition part of a definition of a node."""

        def __init__(self, id_, script):
            """Constructor.

            :param id_: the globally unique identifier of the definition.
            :type  id_: uuid.uuid4
            :param script: the script part of the definition.
            :param script: str
            """

            self.id_ = id_
            self.script = script
```
◇

Fragment referenced in 99a.

⟨*JSON methods* 105a⟩ ≡

```
    classmethod
    def build_node_definition_definition(cls, json_input):
        """Builds and returns a definition for a node definition from the given
        JSON input data.

        :param json_input: the input in JSON format
        :type  json_input: dict

        :return: the definition of a node definition.
        :rtype: qde.editor.domain.node.NodeDefinitionDefinition
        """

        definition_id = uuid.UUID(json_input['id_'])
        script        = str(json_input['script'])

        node_definition_definition = node.NodeDefinitionDefinition(
            definition_id,
            script
        )

        cls.logger.debug("Built node definition definition")
        return node_definition_definition
```
◇

Fragment defined by 71, 72, 78b, 80, 84a, 104a, 105a, 106a.
Fragment referenced in 103a.

⟨*Node definition invocation domain model declarations* 105b⟩ ≡

```
    class NodeDefinitionInvocation(object):
        """Represents an invocation of a definition of a node."""

        def __init__(self, id_, script):
            """Constructor.

            :param id_: the globally unique identifier of the definition.
            :type  id_: uuid.uuid4
            :param script: the script part of the invocation.
            :param script: str
            """

            self.id_ = id_
            self.script = script◇
```

Fragment referenced in 99a.

⟨*JSON methods* 106a⟩ ≡

```python
    classmethod
    def build_node_definition_invocation(cls, json_input):
        """Builds and returns a invocation for a node definition from the given
        JSON input data.

        :param json_input: the input in JSON format
        :type json_input: dict

        :return: the invocation of a node definition.
        :rtype: qde.editor.domain.node.NodeDefinitionInvocation
        """

        invocation_id = uuid.UUID(json_input['id_'])
        script        = str(json_input['script'])

        node_definition_invocation = node.NodeDefinitionInvocation(
            invocation_id,
            script
        )

        cls.logger.debug("Built node definition invocation")
        return node_definition_invocation
```
◇

Fragment defined by 71, 72, 78b, 80, 84a, 104a, 105a, 106a.
Fragment referenced in 103a.

"../src/qde/editor/gui/node.py" 106b≡

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

    """Module holding node related aspects."""

    # System imports
    from PyQt5 import Qt
    from PyQt5 import QtWidgets

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

⟨ *Add node dialog declarations* 85a ⟩
◇

5.4 Test cases

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.5 Requirements

5.6 Directory structure and name-spaces

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