QDE — A visual animation system.

MTE-7103: Master-Thesis

Sven Osterwalder*

February 20, 2017

*sven.osterwalder@students.bfh.ch

Contents

1 TODO Introduction

[Introduction here].

2 TODO Administrative aspects

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

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

2.1 Involved persons

Author	Sven Osterwalder[1]	
Supervisor	Prof. Claude Fuhrer[2]	*Supervises the student doing the thesis*

2.2 Structure of the documentation

This thesis is structured as follows:

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

2.3 Deliverable results

- Report
- Implementation

[1] sven.osterwalder@students.bfh.ch
[2] claude.fuhrer@bfh.ch

3 TODO Scope

3.1 Motivation

[Motivation.]

3.2 Objectives and limitations

[Objectives and limitations.]

3.3 Preliminary activities

[Preliminary activities.]

3.4 New learning contents

[New learning contents.]

4 TODO Procedure

4.1 Organization of work

4.1.1 Meetings

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

4.1.2 Phases of the project and milestones

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

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

4.1.3 Literate programming

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

> **TODO** Provide more information about literate programming.
>
> Citations, explain fragments, explain referencing fragments, code structure does not have to be "normal"

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

> **TODO** Insert reference/link to test cases here.

4.2 Standards and principles

4.2.1 Code

[Code.]

4.2.2 Diagrams

[Diagrams.]

4.2.3 Project structure

[Project structure.]

5 TODO Implementation

5.1 Requirements

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

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

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

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

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

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

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

```
1 install 3.6.0
```

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

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

```
1 pyenv virtualenv 3.6.0 qde
```

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

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

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

Listing 4: Installation of the projects' required dependencies.

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

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

5.2 Project structure

This chapter describes the planned directory structure as well as an instruction on how to set up that structure.

The whole source code shall be placed in the `src` directory underneath the main directory.

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

In der ersten Phase des Projektes soll der Editor erstellt werden. Dieser dient der Erstellung und Verwaltung von Echtzeit-Animationen [1, S. 29].

5.3 Editor

Der Editor soll sich im Verzeichnis `editor` unterhalb des `src/qde`-Verzeichnisses befinden.

Um sicherzustellen, dass Module als solche verwendet werden können, muss pro Modul und Namespace eine Datei zur Initialisierung vorhanden sein. Es handelt sich dabei um Dateien namens `__init__.py`, welche im minimalen Fall leer sind. Diese können aber auch regulären Programmcode, wie zum Beispiel Klassen oder Methoden enthalten.

Im weiteren Verlauf des Dokumentes wird darauf verzichtet diese Dateien explizit zu erwähnen, sie werden direkt in den entsprechenden Codeblöcken erstellt und als gegeben angesehen.

Nun kann mit der eigentlichen Erstellung des Editors begonnen werden.

Der Einstiegspunkt einer Qt-Applikation mit grafischer Oberfläche ist die Klasse `QtApplication`. Gemäss [5] kann die Klasse direkt instanziert und benutzt werden, es ist unter Umständen jedoch sinnvoller die Klasse zu kapseln, was schlussendlich eine höhere Flexibilität bei der Umsetzung bietet. Es soll daher die Klasse `Application` erstellt werden, welche diese Abstraktion bietet.

An dieser Stelle macht es Sinn, sich zu überlegen, welche Funktionalität die Applikation selbst haben soll. Es ist nicht nötig selbst einen Event-Loop zu implementieren, da ein solcher bereits durch Qt vorhanden ist[6].

Die Applikation hat die Aufgabe die Kernelemente der Applikation zu initialisieren. So fungiert das Modul als Knotenpunkt zwischen den verschiedenen Ebenen der Architektur, indem es diese mittels Signalen verbindet.[1, S. 37 bis 38]

Weiter soll es nützliche Schnittstellen, wie zum Beispiel das Protokollieren von Meldungen, bereitstellen. Und schliesslich soll das Modul eine Möglichkeit bieten beim Verlassen der Applikation zusätzliche Aufgaben, wie etwa das Entfernen von temporären Dateien, zu bieten.

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

Die (Datei-) Struktur zur Erstellung und Benennung der Module erfolgt ab diesem Zeitpunkt nach dem Schichten-Modell gemäss [1, S. 40].

Die Protokollierung auf Klassen-Basis stattfinden. Vorerst sollen Protokollierungen als Stream ausgegeben werden. Pro Klasse muss also eine `logging`-Instanz instanziert und mit dem entsprechenden Handler ausgestattet werden. Um den Programmcode nicht unnötig wiederholen zu müssen, bietet sich hierfür das Decorator-Pattern von Python an[7].

Die Klasse zur Protokollierung soll also Folgendes tun:

- Einen Logger-Namen auf Basis des aktuellen Moduls und der aktuellen Klasse setzen

```
1 logger_name = "%s.%s" % (cls.__module__, cls.__name__)
```

Listing 5: Setzen des Logger-Names auf Basis des aktuellen Modules und Klasse.

- Einen Stream-Handler nutzen

[4] https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces
[5] http://doc.qt.io/Qt-5/qapplication.html
[6] http://doc.qt.io/Qt-5/qapplication.html#exec
[7] https://www.python.org/dev/peps/pep-0318/

```
stream_handler = logging.StreamHandler()
```

Listing 6: Initialisieren eines Stream-Handlers.

- Die Stufe der Protokollierung abhängig von der aktuellen Konfiguration setzen

```
# TODO: Do this according to config.
stream_handler.setLevel(logging.DEBUG)
```

Listing 7: Setzen des `DEBUG` Log-Levels.

- Protokoll-Einträge ansprechend formatieren

```
# TODO: Set up formatter in debug mode only
formatter = logging.Formatter("%(asctime)s - %(levelname)-7s - %(name)s.%(funcName)s::%(lineno)s: %(message)s")
stream_handler.setFormatter(formatter)
```

Listing 8: Anpassung der Ausgabe von Protokoll-Meldungen.

- Eine einfache Schnittstelle zur Protokollierung bieten

```
cls.logger = logging.getLogger(logger_name)
cls.logger.propagate = False
cls.logger.addHandler(stream_handler)

return cls
```

Listing 9: Nutzung des erstellten Stream-Handlers und Rückgabe der Klasse.

Nun kann die eigentliche Funktionalität implementiert werden.

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

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

# System imports
import logging


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

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

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

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

Listing 10: Das `common`-Modul und eine Methode zur Protokollierung in Klassen.

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

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

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

Listing 11: Definition der Klasse `Application` mit dem `with_logger`-Dekorator des `common`-Modules.

Damit die Protokollierung jedoch nicht nur via STDOUT in der Konsole statt findet, muss diese entsprechend konfiguriert werden. Das *logging*-Modul von Python bietet hierzu vielfältige Möglichkeiten.[8] So kann die Protokollierung mittels der "Configuration API" konfiguriert werden. Hier bietet sich die Konfiguration via Dictionary an. Ein Dictionary kann zum Beispiel sehr einfach aus einer JSON-Datei generiert werden.

Die Haupt-Applikation soll die Protokollierung folgendermassen vornehmen:

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

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

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

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

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

    if value:
        path = value

    if os.path.exists(path):
        with open(path, 'rt') as f:

            config = json.load(f)
            logging.config.dictConfig(config)
    else:
        logging.basicConfig(level=default_level)
```

Listing 12: Methode zum Initialisieren der Protokollierung der Applikation.

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

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

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

<<app-imports>>

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

Listing 13: Haupt-Modul und Einstiegspunkt der Applikation.

```
<<app-system-imports>>

<<app-project-imports>>
```

Listing 14: Definition der Importe des Haupt-Modules.

```
# System imports
from   PyQt5.Qt import QApplication
from   PyQt5.Qt import QIcon
import logging
import os
```

Listing 15: Importe von Python-eigenen Modulen im Haupt-Modul.

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

Listing 16: Importe von selbst verfassten Modulen im Haupt-Modul.

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

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

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

    self.setup_logging()
```

Listing 17: Konstruktor des Haupt-Modules.

Der Konstruktor und die Methode zum Einrichten der Protokollierung werden schliesslich der Klasse hinzugefügt.

```
<<app-constructor>>

<<app-setup-logging>>
```

Listing 18: Hinzufügen des Konstruktors sowie der Methode zum Einrichten der Protokollierung zum Körper des Haupt-Modules.

6 Working log

<2017-02-20 Mon> Initial set up and structuring of the document. *<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.

7 Bibliography

Bibliography

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

8 Appendix

8.1 Test cases

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

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

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

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

Listing 19: Header des Test-Modules, «test-app-header».

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

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

Listing 20: Aufruf des Main-Modules mit zwei Argumenten, `argument1` und `argument2`.

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

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

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

Listing 21: Definition der Importe für das Modul zum Testen der Applikation.

```
# System imports
import sys
```

Listing 22: Importe von Python-eigenen Modulen im Modul zum Testen der Applikation.

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

Listing 23: Importe von selbst verfassten Modulen im Modul zum Testen der Applikation.

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

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

Listing 24: Methode zum Testen des Konstruktors der Applikation.

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

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

<<test-app-imports>>

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

Listing 25: Modul zum Testen der Applikation.

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

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

Listing 26: Aufruf zum Testen des Applkations-Modules.

Um sicherzustellen, dass die Protokollierung wie gewünscht funktioniert, wird diese durch die entsprechenden Testfälle abgedeckt.
Der einfachste Testfall ist die Standardkonfiguration, also ein Aufruf ohne Parameter.

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

Listing 27: Testfall 1 der Protkollierung der Hauptapplikation: Aufruf ohne Argumente.

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

```
import logging
```

Listing 28: Erweiterung des Importes von System-Modulen im Modul zum Testen der Applikation.

Und der Testfall muss den Testfällen hinzugefügt werden.

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

Listing 29: Hinzufügen des Testfalles 1 zu den bestehenden Testfällen im Modul zum Testen der Applikation.

Auch hierfür werden wiederum zuerst die Testfälle verfasst.

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

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

# System imports
import logging

# Project imports
from qde.editor.foundation import common


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

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

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

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

Listing 30: Testfälle der Hilfsmethode zur Protokollierung.

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

8.2 Meeting minutes

8.2.1 Meeting mintutes 2017-02-23

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

Kick-off meeting for the thesis.

1. Presentation and discussion of the current state of work
   - Presentation of the workflow. Emacs and Org-Mode is used to write the documentation as well as the actual code. (SO)
     - This is a very interesting approach. The question remains if the effort of this method does not prevail the method of developing the application and the documentation in parallel. It is important to reach a certain state of the application. Also the report should not exceed around 80 pages. (CF)
       - A decision about the used method is made until the end of this week. (SO)
   - The code will unit-tested using py.test and / or hypothesis. (SO)
   - Presentation of the structure of the documentation. It follows the schematics of the preceding documentations. (SO)
2. Further steps / proceedings
   - The expert of the thesis, Mr. Dubuis, puts mainly emphasis on the documentation. The code of the thesis is respected too, but is clearly not the main aspect. (CF)

- Mr. Dubuis also puts emphasis on code metrics. Therefore the code needs to be (automatically) tested and a coverage of at least 60 to 70 percent must be reached. (CF)
- A meeting with Mr. Dubuis shall be scheduled at the end of March or beginning of April 2017. (CF)
- The administrative aspects as well as the scope should be written until end of March 2017 for being able to present them to Mr. Dubuis. (CF)
- Mr. Dubuis should be asked if the publicly available access to the whole thesis is enough or if he wishes to receive the particular status right before the meetings. (CF)
- Regularly meetings will be held, but the frequency is to be defined yet. Further information follows per e-mail. (CF)
- At the beginning of the studies, a workplace at the Berne University of applied sciences in Biel was offered. Is this possibility still available? (SO)
  - Yes, that possibility is still available and details will be clarified and follow per e-mail. (CF)

3. To do for the next meeting
   a) **DONE** Create GitHub repository for the thesis. (SO)
      i. **DONE** Inform Mr. Fuhrer about the creation of the repository. (SO)
   b) **DONE** Ask Mr. Dubuis by mail how he wants to receive the documentation. (SO)
      i. **TODO** Await answer of Mr. Dubuis (ED)
   c) **DONE** Set up appointments with Mr. Dubuis (CF)
      i. **TODO** Await answer of Mr. Dubuis (ED)
   d) **DONE** Clarify possibility of a workplace at Berne University of applied sciences in Biel. (CF)

      i. A workplace was found at the RISIS laboratory and may be used instantly. (CF)
   e) **DONE** Decide about the method used for developing this thesis. (SO)
      i. After discussions with a colleague the method of literate programming is kept. The documentation containing the literate program will although be attached as appendix as it most likely will exceed 80 pages. Instead the method will be introduced in the report and the report will be endowed with examples from the literate program.
   f) **TODO** Describe procedure and set up a time schedule including milestones. (SO)

4. Scheduling of the next meeting
   - To be defined