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

[Administrative aspects].

3 TODO Scope

[Scope]

4 TODO Procedure

- Literate programming
- Agile

4.1 Project schedule

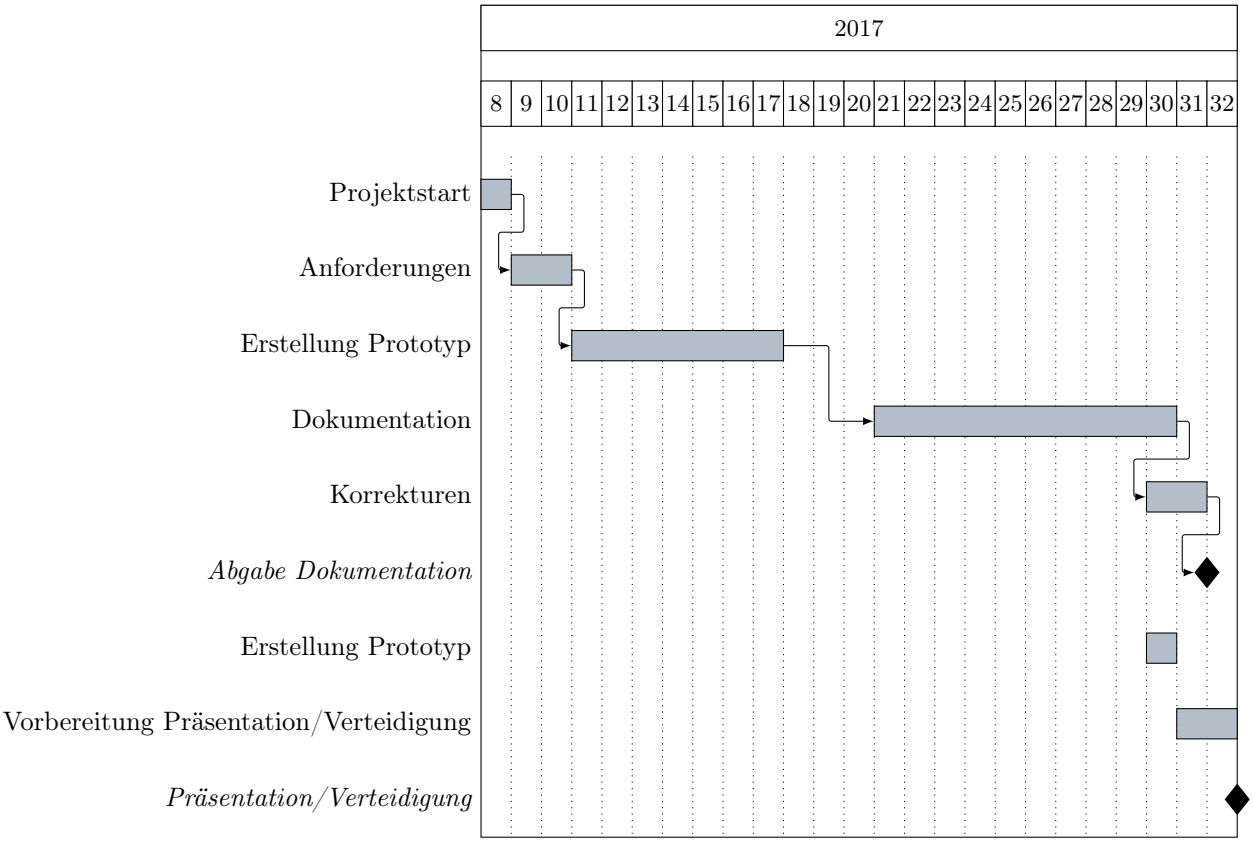

Figure 4.1: Zeitplan; Der Titel stellt Jahreszahlen, der Untertitel Kalenderwochen dar

5 TODO Implementation

Die Umsetzung des Projektes findet mittels Literate Programming statt. Der Programmcode wird von Grund auf direkt aus dieser Dokumentation erzeugt.

Den Aufbau betreffend, wird erst die Funktionalität erklärt, dann wird die Funktionalität implementiert. (Unit-) Testfälle werden getrennt von dieser Dokumentation verfasst und befinden sich im Anhang unter

TODO Insert reference/link to test cases here.

.

Voraussetzungen, um den Programmcode direkt aus dieser Dokumentation zu erstellen, sind zum aktuellen Zeitpunkt die folgenden:

- Ein Unix-Derivat als Betriebssystem (Linux, macOS)
- Python in der Version 3.5.x oder höher[1]
- Pyenv[2]
- Pyenv-virtualenv[3]

Als erster Schritt wird eine passende Version von Python für virtualenv installiert. Die verfügbaren Versionen lassen sich wie folgt anzeigen.

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

Listing 1: Anzeige der verfügbaren Python-Versionen für Pyenv.

Die gewünschte Version wird wie folgt installiert. In diesem Beispiel handelt es sich um Version 3.6.0.

```
1 install 3.6.0
```

Listing 2: Installation von Python in der Version 3.6.0 für Pyenv.

Es empfiehlt sich für das Projekt eine eigene, virtuelle Umgebung für Python zu erstellen. Darin werden alle Abhängigkeiten installiert und somit werden die Python-Pakete des Betriebsystem nicht kompromitiert. Es wird zuerst die gewünschte (und zuvor installierte) Version, dann der gewünschte Name der virtuellen Umgebung angegeben.

```
1 pyenv virtualenv 3.6.0 qde
```

Listing 3: Erstellung einer virtuellen Python-Umgebung mit Python Version 3.6.0.

Nun können die benötigten Abhängigkeiten für das Projekt problemlos installiert werden. Diese befinden sich in der Datei `python_requirements.txt` und werden mittels `pip` installiert.

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

Listing 4: Installation der benötigten Abhängigkeiten des Projektes.

Somit sind nun alle Voraussetzungen erfüllt und die eigentliche Umsetzung kann beginnen.

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

Der gesamte Programmcode soll im Verzeichnis `src` unterhalb des Hauptverzeichnisses liegen.

```
mkdir -p src
```

Listing 5: Erstellung des *src*-Unterverzeichnisses.

Um zu verhindern, dass mehrere Module denselben Namen verwenden, werden Namespaces verwendet.[4] Der Haupt-Namespace des Projektes soll `qde` lauten.

```
mkdir -p src/qde
```

Listing 6: Erstellung des `qde`-Namespaces.

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

5.1 Editor

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

```
mkdir -p src/qde/editor
```

Listing 7: Erstellung des `editor`-Namespaces.

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.

```
touch src/qde/__init__.py
touch src/qde/editor/__init__.py
```

Listing 8: Erstellung des `qde`-Namespaces und des `editor`-Namespaces.

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. Wie unter beschrieben, werden für alle Funktionalitäten erst (Unit-) Tests verfasst und erst dann die eigentliche Funktionalität umgesetzt.

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.

```
mkdir -p src/qde/editor/application
touch src/qde/editor/application/__init__.py
```

Listing 9: Erstellung des `application`-Namespaces.

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 10: Header des Test-Modules, `«test-app-header»`.

[4] `https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces`
[5] `http://doc.qt.io/Qt-5/qapplication.html`

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 11: 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 12: Definition der Importe für das Modul zum Testen der Applikation.

```
# System imports
import sys
```

Listing 13: Importe von Python-eigenen Modulen im Modul zum Testen der Applikation.

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

Listing 14: 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 15: 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 16: 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 17: Aufruf zum Testen des Applkations-Modules.

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]

[6] `http://doc.qt.io/Qt-5/qapplication.html#exec`

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].

```
mkdir -p src/qde/editor/foundation
touch src/qde/editor/foundation/__init__.py
```

Listing 18: Erstellung und Initialisierung des `foundation`-Namespaces.

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

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

Listing 19: Setzen des Logger-Names auf Basis des aktuellen Modules und Klasse.

- Einen Stream-Handler nutzen

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

Listing 20: 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 21: 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 22: 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 23: Nutzung des erstellten Stream-Handlers und Rückgabe der Klasse.

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

Auch hierfür werden wiederum zuerst die Testfälle verfasst.

```
# -*- 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 24: Testfälle der Hilfsmethode zur Protokollierung.

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

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

Führt man nun die Testfälle erneut aus, so schlagen diese nicht mehr fehl.

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

Listing 26: Ausführen der Testfälle für das `common`-Modul.

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 27: 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.

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 28: 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 29: Erweiterung des Importes von System-Modulen im Modul zum Testen der Applikation.

Und der Testfall muss den Testfällen hinzugefügt werden.

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

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

Listing 30: Hinzufügen des Testfalles 1 zu den bestehenden Testfällen im Modul zum Testen der Applikation.

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 31: Methode zum Initialisieren der Protokollierung der Applikation.

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

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

<<app-imports>>

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

Listing 32: Haupt-Modul und Einstiegspunkt der Applikation.

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

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

Listing 33: Definition der Importe des Haupt-Modules.

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

Listing 34: Importe von Python-eigenen Modulen im Haupt-Modul.

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

Listing 35: 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 36: Konstruktor des Haupt-Modules.

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

```
<<app-constructor>>

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

Listing 37: Hinzufügen des Konstruktors sowie der Methode zum Einrichten der Protokollierung zum Körper des Haupt-Modules.

Somit ist es nun möglich die Testfälle der Applikation auszuführen.

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

Listing 38: Ausführen der Testfälle für das Haupt-Modul.

6 Worklog

<2017-02-20 Mon> Initiale Struktur des Dokumentes

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 Meeting minutes

8.1.1 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) **TODO** Create GitHub repository for the thesis. (SO)
      i. **TODO** Inform Mr. Fuhrer about the creation of the repository. (SO)
   b) **TODO** Ask Mr. Dubuis by mail how he wants to receive the documentation. (SO)

c) **TODO** Set up appointments with Mr. Dubuis (CF)

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.

4. Scheduling of the next meeting

   - To be defined