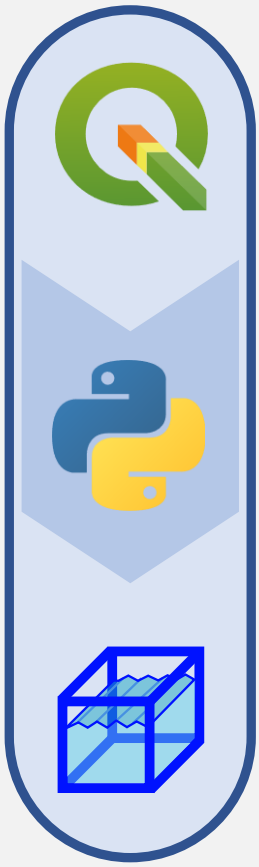
THM
TECHNISCHE HOCHSCHULE MITTELHESSEN

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

QGIS-Schnittstelle für das NA-Modell BlueM

QGIS-Interface for the simulation model BlueM



Abschlussarbeit zur Erlangung des akademischen Grades Master of Engineering im Studiengang Infrastrukturmanagement der Frankfurt University of Applied Sciences (UAS) in Kooperation mit der Technischen Hochschule Mittelhessen (THM) in Gießen

Erstprüfer **Prof. Dr.-Ing. Steffen Heusch**
steffen.heusch@bau.thm.de

Zweitprüfer **Prof. Dr.-Ing. Michael Bach**
m.bach@bluemodel.org

Kandidat **Martin Großhaus, B.Eng.**
mgrosshaus@gmail.com

vorgelegt am 14.02.2022

Danke an die üblichen Verdächtigen.

Im Geiste der Open-Source-Philosophie von QGIS ist diese Arbeit an alle gerichtet, die sich mit der in-silico Modellierung unserer Welt befassen. Um eine bessere Lesbarkeit und automatisierte Übersetzbarkeit zu gewährleisten, wird jedoch hauptsächlich das generische Maskulinum verwendet. Weibliche und anderweitige Geschlechteridentitäten werden dabei ausdrücklich mitgemeint, soweit es für die Aussage erforderlich ist.

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Masterthesis mit dem Titel „QGIS-Schnittstelle für das NA-Modell BlueM" (Englisch: *QGIS-Interface for the simulation model BlueM*) selbstständig und ohne fremde Hilfe angefertigt habe. Dass alle Textstellen, die wörtlich oder inhaltlich auf fremdes Gedankengut zurückgreifen, kenntlich gemacht wurden. Diese Masterthesis wurde noch nie einem anderen Prüfungsamt vorgelegt oder veröffentlicht.

Grünberg, ***14.02.2022***

Martin Großhaus, B.Eng.

Notizen

Abstract

This dissertation is concerned with an interface that enables the processing of data from the geographic information system QGIS for use in the rainfall-runoff-simulation-model BlueM.

Initially the thematic background is explained from which the theoretical conception of the interface is derived. Building on this, the technical implementation of the interface as a python-based plugin for QGIS is described in detail.

The application of the plugin is shown in a typical usage scenario; a user manual can be found in the appendix. There the source code of the plugin is also preserved.

Kurzbeschreibung

Die vorliegende Ausarbeitung beschäftigt sich mit einer Schnittstelle, die es ermöglicht Daten aus dem Geographischen Informationssystem QGIS für die Nutzung im Niederschlags-Abfluss-Simulationsmodell BlueM aufzubereiten.

Zuerst wird der fachliche Hintergrund erläutert, aus welchem sich die theoretische Konzeption der Schnittstelle ableitet. Darauf aufbauend wird die technische Umsetzung der Schnittstelle als mit Python programmiertes Plugin für QGIS detailliert beschrieben.

Die Nutzung des Plugins wird anhand eines typischen Anwendungsszenarios gezeigt, weiterhin wird auf das Benutzerhandbuch (in Englisch) im Anhang verwiesen. Hier findet sich auch der Quellcode des Plugins.

Inhaltsverzeichnis

Abbildungsverzeichnis

Bildquellen

Abb. 1 & 2: Aus den im Text genannten Literaturquellen
Abb. 3, 6 & 7: Logos von den Webseiten der Programme
Abb. 4, 8 bis 19: Eigene Werke & Screenshots
Abb. 5: Aus den Folien der Vorlesung „Geographische Informations-systeme" von Prof. Dr.-Ing. Joaquín Díaz, THM (09.04.2014)
Deckblatt: Grafik und Bild sind eigene Werke

Tabellenverzeichnis

Abkürzungsverzeichnis

ANSI	Erweiterung für ASCII (256 statt 128 Zeichen; inkl. ä, ö, ü, ß)
API	Application Programming Interface (Programmierschnittstelle)
ASCII	American Standard Code for Information Interchange
CSV	Comma-separated values
DWD	Deutscher Wetterdienst
ESRI	Environmental Systems Research Institute (US-amerikanischer Softwarehersteller)
GIS	Geographisches Informationssystem
GUI	Graphical User Interface (Graphische Benutzeroberfläche)
HLNUG	Hessisches Landesamt für Naturschutz, Umwelt und Geologie
HRU	Hydrologic Response Unit
IDE	Integrated Development Environment (Integrierte Entwicklungsumgebung)
OGC	Open Geospatial Consortium
OpenMI	Open Modeling Interface (offene Modell-Schnittstelle; Standard)
PDF	Portable Document Format (ein Dokumentenformat)
PSF	Python Software Foundation
QGIS	Ein freies & quelloffenes geographisches Informationssystem
SMUSI	Ein Schmutzfrachtsimulationsmodell
SWAT	„Soil & Water Assessment Tool" -Weit verbreitetes hydrologisches Modell mit besonderem Bezug zur Landwirtschaft
SWMM	Storm Water Management Model
UCS	Universal Coded Character Set
UI	User Interface (Benutzeroberfläche)
UTF-8	UCS Transformation Format mit 8 Bit
XML	Extensible Markup Language (eine Auszeichnungssprache)

1. Einleitung

Wasser bildet den Hauptbestandteil aller dem Menschen bekannten Lebewesen; und Gewässer waren nach aktuellem Stand der Wissenschaft gar die Wiege allen Lebens auf der Erde.

Doch sind es nicht nur die inneren chemischen Prozesse der Bewohner des „blauen Planeten", die durch Wasser bestimmt werden. Auch die Beschaffenheit ihrer Umgebung wird durch das feuchte Element geformt. Beispielhaft erwähnt sei hier der durch den Colorado River entstandene Grand Canyon, wobei festzuhalten ist, dass prinzipiell jedes Gelände durch Erosion und Sedimentation über Jahrmillionen sein heutiges Gesicht bekommen hat.

Zwar sind geologische Erscheinungen wie Vulkanismus und Kontinentaldrift für das „Grobe" wie die Entstehung von Gebirgsketten verantwortlich, aber die Ausgestaltung der einzelnen Täler wurde auch hier durch das Herausschwemmen von lockerem Felsen & Boden über kleine Bergbäche und reißende Flüsse bewerkstelligt.

Diese, durch Wasserbewegungen hervorgerufenen, Prozesse sind auch noch nicht abgeschlossen. Das kann positiv sein, wie man am Beispiel des Golfstroms sehen kann: Durch diese atlantische Meeresströmung werden unfassbare Mengen warmen Wassers gen Norden transportiert, welche Europa ein deutlich milderes Klima bescheren, als es für dessen Breitengrade zu erwarten wäre.

Da Wasser insbesondere für den Menschen lebenswichtig ist, spielte sich dieses Leben schon immer in der Nähe von Gewässern ab. Als historische Beispiele seien hier genannt das antike Ägypten am Nil und Mesopotamien (griechisch wörtlich „zwischen den Flüssen", gemeint sind Euphrat und Tigris).

Diese Fließgewässer boten den Menschen bereits in frühster Zeit Zugang zu Frischwasser für sich selbst, ihre Nutztiere und die Bewässerung der Felder, boten eine schnelle Entsorgung von Exkrementen und anderen Abfällen und ermöglichten das Reisen und Transportieren von Gütern, lange bevor die erste Straße nach Rom führte.

Doch diese räumliche Nähe von Mensch und Wasser kann auch negative Folgen haben. Als vor gut 2.500 Jahren der griechische Epiker Choirilos von Samos feststellte *„Steter Tropfen höhlt den Stein";* erkannte etwa zur selben Zeit der chinesische Philosoph 老子 (Lao Tzu) *„Nichts ist weicher oder flexibler als Wasser - und doch kann ihm nichts widerstehen*".

Dieses Zerstörungspotential wird deutlich, wenn wir uns an die eben erwähnte Formung von Tälern erinnern. Das Wasser eines ausufernden Bergbachs nach einem starken Regen oder während der Schneeschmelze differenziert nicht, ob es sich gravitationsgetrieben seinen Weg durch den Boden der Talsohle frisst oder eine pittoreske Alm hinwegreist.

Um dieses Potential zu quantifizieren, lohnt ein Blick auf die Darstellung des Wasserkreislaufes in einer Ausarbeitung des US-amerikanischen *National Center for Atmospheric Research* aus 2006 [1]:

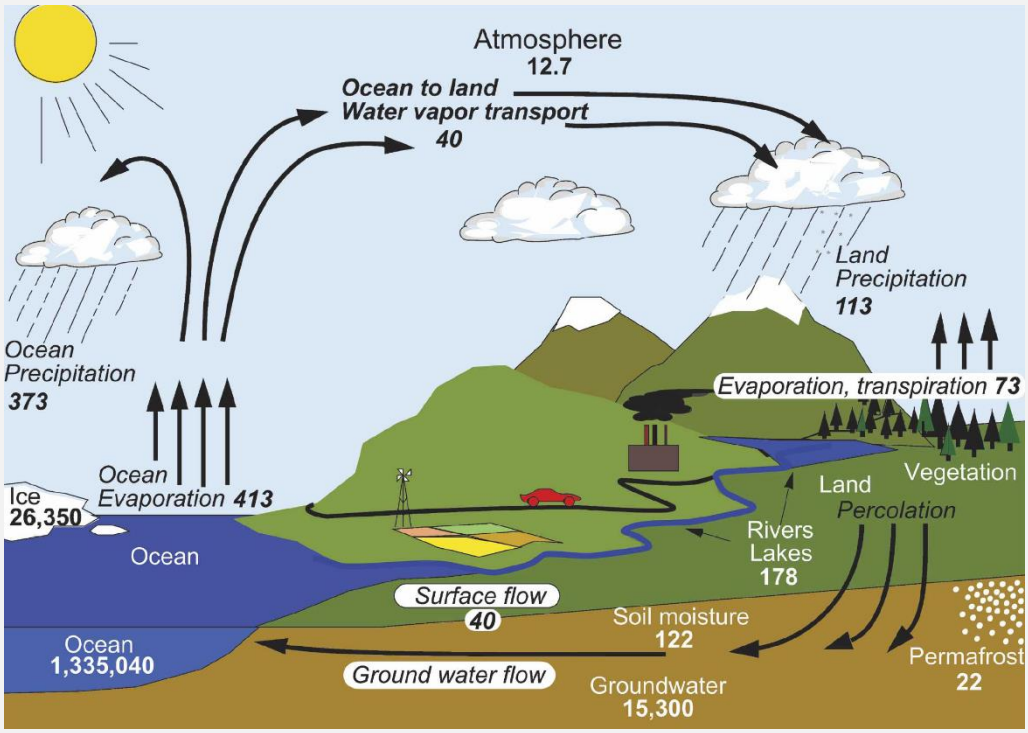


Abb. 1 Wasserkreislauf mit Kennzahlen

Die Kennzahlen der Abbildung sind in einfacher Schrift für Speicher (in 1.000 km³) und kursiver Schrift für Transport (1.000 km³ pro Jahr).

Aus diesen Untersuchungen von *Trenberth et al.* geht hervor, dass jährlich etwa 113.000 km³ Wasser als Niederschlag über Land niedergehen. Umgerechnet bedeutet dies eine Masse von etwa 3,5 Millionen Kubikmetern pro Sekunde – genug um den leeren Bodensee (Volumen: 48,5 km³) in weniger als 4 Stunden zu füllen.

Diese schwer fassbaren Mengen sind nicht gleichmäßig über den Globus verteilt und fallen auch nicht konstant, vielmehr gibt es große Unterschiede in Zeit und Raum. So vermeldet der *Deutsche Wetterdienst (DWD)* für die Bundesrepublik, die einen durchschnittlichen jährlichen Niederschlag von circa 800 mm (1 mm Niederschlagshöhe entspricht 1 Liter Wasser pro Quadratmeter) verbucht, folgende Rekorde [2]:

Dauer	Menge	Ort	Datum
8 Minuten	126 mm	bei Füssen (Ostallgäu)	25.05.1920
2 Stunden	245 mm	Münster (NRW)	28.07.2014
1 Tag	354 mm	Zinnwald-Georgenfeld (Erzgebirge)	12./13.08.2002
1 Woche	515 mm	Schneizlreuth-Weißbach (Berchtesgadener Land)	07.-14.09.1899
1 Monat	779 mm	Stein (Kreis Rosenheim)	Juli 1954

Tab. 1 Niederschlagsrekorde in Deutschland (laut DWD)

[1] Trenberth, Kevin E., et al. 2006
[2] Junghänel, T., et al. 2021

Diese beeindruckenden Extreme werden international sogar noch deutlich übertroffen, allerdings sind globale Rekorde aus über 100 Jahren Wetteraufzeichnungen nicht das maßgebende Kriterium für Maßnahmen an einem bestimmten Ort. Hier sind ortsspezifische Daten in Form von Jahresganglinien bzw. historischen Regenereignissen etc. relevant (Stichwort *KOSTRA-Atlas*).

In diesem Zusammenhang besonders interessant ist eine Untersuchung des US-amerikanischen *National Center for Atmospheric Research* in Zusammenarbeit mit der *ETH Zürich* [3], die besagt, dass global betrachtet 50% des gesamten jährlichen Niederschlags an einem Ort in dessen 12 „nassesten" Tagen fallen – für Deutschland sind es etwa die 20 „nassesten" Tage.

Auch muss erwähnt werden, dass nicht jeder gefallene Tropfen Niederschlag bis in einen Ozean abgeleitet werden muss; vielmehr wird laut *Trenberth et al* beinahe 2/3 des Niederschlags durch Evaporation und Transpiration wieder an die Atmosphäre abgegeben. Dies geschieht jedoch in starker Abhängigkeit von der Umgebungssituation: Niederschlag über Wald und Wiesen wird zu einem Großteil wieder gasförmig, bevor es weit geflossen ist. Auf von Menschen versiegelten Flächen, wie beispielsweise Straßen, die auch die Versickerung unmöglich machen, hat das Niederschlagswasser hingegen keine Zeit, zu evaporieren, bevor es aus Gründen der Verkehrssicherheit in einem Kanal verschwinden muss – eine Evapotranspiration kann also erst nach Ableitung des Niederschlagswassers aus der menschengemachten Umgebung erfolgen.

Es muss dabei bedacht werden, dass die von Starkregen ausgehenden Gefahren in Zukunft aufgrund des Klimawandels eher zunehmen werden. Der Bericht des DWD [4] zur hydro-klimatologischen Einordnung der Starkregenereignisse im Juli 2021 (Stichwort Ahrtal) kommt zu der Abschätzung, dass die jährlichen Niederschlagsmengen sich zwar in Zukunft nicht stark erhöhen, jedoch auf weniger, dafür extremere Starkregenereignisse konzentrieren werden.

Doch die Behandlungen des Wassers erfolgen nicht nur, um den Menschen vor dessen Zerstörungspotential zu schützen, oft muss auch das Wasser vor menschlichen Einflüssen in Form von Verunreinigungen (Toxische Stoffe, Hormone, Mikroplastik, etc.) geschützt werden, die die Gewässerökologie flussabwärts gefährden könnten.

Bauliche Anlagen, die der Behandlung von Niederschlagswasser dienen, gehören einerseits zu den monumentalsten Bauwerken der Menschheit (beispielsweise die Drei-Schluchten-Talsperre am Jangtsekiang, VR China), können andererseits aber auch enorm kleinteilig und filigran sein, wie ein städtisches Netz aus Regenwasserkanälen mit unzähligen Anschlüssen für Dächer, Terrassen und Verkehrsflächen.

Bau und Betrieb solch enormer Anlagen erfordern ein hohes Maß an Planung. Um wassertechnische Anlagen passend zu dimensionieren, müssen mithilfe von hydrologischen Modellen bestehende oder zukünftige Abflussszenarien simuliert und deren Ergebnisse korrekt interpretiert werden – hiervon hängt

[3] Pendergrass, Angeline G. und Knutti, Reto. 2018
[4] Junghänel, T., et al. 2021

Abb. 2 Unterirdischer Regenwasserspeicher in Tokio, Japan

grundsätzlich die Sinnhaftigkeit massiver Investitionen, im Speziellen sogar das Retten von Leben ab.

Der abgebildete Regenwasserspeicher (25,4 x 78 x 177 Meter) ist Teil eines Systems, um Tokio vor Überschwemmungen durch Taifune zu schützen. Baukosten des Systems: ca. 2 Milliarden Euro [5].

Diese hydrologischen Modelle nutzen als Eingangsdaten unter anderem die zu verarbeitende Bemessungsregenspende, natürliche & künstliche Abflüsse, Informationen über Zusammensetzung & Zustand der vorkommenden Böden, sowie Daten über bestehende oder zukünftige wasserbauliche Anlagen (Kapazitäten von Speichern & Leitungen, etc.) und deren Abhängigkeiten in einem System.

Daraus simuliert das hydrologische Modell ein spezifisches Regenereignis und wie es sich auf das erfasste System im Verlauf der Simulation auswirkt. Hierdurch können schnell Zwangspunkte identifiziert werden, die bei zu häufigem Eintreten eliminiert werden müssen.

Eines dieser hydrologischen Simulationsmodelle ist das Programm BlueM. Um viele der von BlueM benötigten Daten zu ermitteln beziehungsweise aufzubereiten, ist es zweckdienlich ein sogenanntes Geographisches Informationssystem (beispielsweise QGIS) zu nutzen.

Um die gemeinsame Nutzung beider Systeme effizienter zu gestalten, wurde im Rahmen der vorliegenden Arbeit eine Schnittstelle entwickelt, die es ermöglicht, die in QGIS vorliegenden Daten in einer durch BlueM nutzbaren Formatierung zu exportieren.

[5] Kaffka, Ines. 2017

2. Theoretische Grundlagen

2.1 Hydrologische Modelle

2.1.1 Allgemein

Ein hydrologisches Modell ist eine vereinfachte Repräsentation eines Systems in der realen Welt. Das beste Modell ist dabei das, welches mit den wenigsten Parametern und der geringsten Komplexität Vorhersagen trifft, die der Realität nahekommen.[6]

Laut Devi, et al (2015) [7] lassen sich hydrologische Modelle in 3 Kategorien einteilen:

- ***Empirische Modelle (black box)***
  Diese orientieren sich hauptsächlich an aus Beobachtung ähnlicher Situationen gewonnenen Daten und versuchen damit ein funktionales Verhältnis von Systemeingängen und Ausgängen zu konstruieren. Daraus können dann Rückschlüsse auf die Untersuchungssituation gezogen werden. Ein tieferes Verständnis der genauen Vorgänge ist nicht erforderlich (Stichwort „Maschinelles Lernen").

- ***Konzeptionelle Modelle (grey box)***
  „Conceptual models" analysieren die grundlegenden Zusammenhänge des vorliegenden Systems in Bezug auf Evaporation, Perkolation, etc. und versuchen durch Kalibrierung der Parameter ein bekanntes Ergebnis nachzuvollziehen. Mit diesen angepassten Parametern können danach die Auswirkungen einzelner Veränderungen abgeschätzt werden.
  Beispiel: Stanford Watershed Model IV (SWM)

- ***Physikalische Modelle (white box)***
  Hier wird versucht die Realität am genausten durch das rechnerische Nachvollziehen der beteiligten physikalischen Prozesse zu imitieren. Der technische Aufwand ist dabei der größte der drei Kategorien, dafür lässt sich die Systematik am einfachsten auf neue Umstände anpassen, da keine Kalibrierung oder umfassende empirische Erhebungen notwendig sind.
  Beispiel: MIKE SHE und SWAT; auch BlueM fällt in diese Kategorie

Das Programm SWAT (Soil & Water Assessment Tool) des US-amerikanischen Landwirtschaftsministeriums (Programm daher in *public domain*), ist das international wohl am weitesten verbreitete hydrologische Modell. So weist die

[6] Sorooshian, Soroosh und Moradkhani, Hamid. 2008 (Seite 291)
[7] Devi, Gayathri K, Ganasri, B P und Dwarakish, G S. 2015

eigene Website der *„SWAT Literature Database*"seit 1984 beinahe 5.000 *peer-reviewed* Artikel zu diesem Modell auf.

2.1.2 Das Programm BlueM

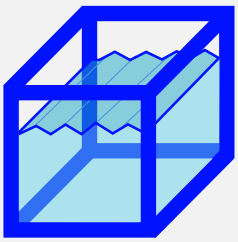


Abb. 3 BlueM - Logo

Laut dem Development Team ist BlueM ein Softwarepaket zur integrierten Flusseinzugsgebietsbewirtschaftung bzw. -modellierung.

Das nach Registrierung frei verfügbare Programm wird in Zusammenarbeit des Development Teams mit dem Fachgebiet für Ingenieurhydrologie und Wasserbewirtschaftung (ihwb) der Technischen Universität Darmstadt entwickelt.

BlueM (früher auch Blue Model genannt) bietet nach Aussage des Dev Teams dem Nutzer die Möglichkeit, die Entladungs- und Verschmutzungsbelastungen von ruralen und urbanen Einzugsgebieten zu simulieren, zu analysieren und gegebenenfalls zu optimieren.

Die Software-Suite kann in drei Hauptwerkzeuge unterteilt werden:

- **BlueM.Sim** ist der in der Programmiersprache *Fortran* entwickelte hydrologische Rechenkern, welcher die physikalischen Prozesse simuliert.
- **BlueM.Wave** ermöglicht das Behandeln und die Visualisierung von Niederschlags-Zeitreihen.
- **BlueM.Opt** wird zur Optimierung bzw. Kalibrierung von Modellparametern genutzt.

Für nähere Informationen zum Programm BlueM wird auf die Website www.bluemodel.org inklusive verlinktes Wiki verwiesen. Für eine detaillierte Beschreibung der Funktionsweise des Rechenkerns von BlueM.Sim wird die Masterthesis von Tobias Rosskopf [8] empfohlen. Der Quellcode kann via GitHub (*www.github.com/bluemodel*) eingesehen werden.

2.1.3 Die BlueM Dateien

Die Ein- & Ausgabe von Daten erfolgt bei BlueM in Form von Textdateien mit spezifischen Endungen, die auf ihren jeweiligen Inhalt hinweisen.

Diese Dateien sind im ANSI Zeichensatz kodiert, welcher den Standard ASCII-Code (*American Standard Code for Information Interchange*) von 7 auf 8 Bit pro Zeichen erweitert. Dies führt dazu, dass mit ANSI nicht nur die 128 ASCII-Zeichen verwendet werden können, sondern eine Auswahl von 256 Zeichen (inklusive den in Deutschland verwendeten Umlauten ä, ö, ü & ß) zur Verfügung steht.

[8] Rosskopf, Tobias. 2016

Eine BlueM-Datei enthält im Allgemeinen eine Tabelle – im Fall der TAL-Datei sogar mehrere Tabellen, nur die ALL-Datei ist mehr wie eine Liste aufgebaut.

Diese Tabellen verfügen über einen Überschriftenblock, welcher aufgrund von Sternchen („Asterisk" / „*") am Anfang jeder Zeile als solcher erkannt und nicht durch das Programm eingelesen wird. Vielmehr ist dieser Abschnitt dazu da, den eigentlichen Tabelleninhalt für einen menschlichen Nutzer verständlich zu machen. Neben den Überschriften der einzelnen Tabellenspalten können hier auch weitere Informationen, wie beispielsweise Datenherkunft und Aktualität, hinterlegt werden.

Unter dem Überschriftenblock befinden sich die eigentlichen Daten über Systemelemente, Niederschlags-Ganglinien oder Ähnliches. Eine Besonderheit der BlueM-Dateien ist, dass die einzelnen Einträge in der Vertikalen durch die einzelnen Zeilen getrennt sind, in der Horizontalen aber nicht durch Kommata oder Semikola, wie es beispielsweise bei CSV-Dateien üblich wäre.

Auch wenn einzelne Tabellenwerte teilweise durch vertikale Striche („|") getrennt sind, so sind diese lediglich visuelle Hilfen für den Nutzer. Vielmehr ist es so, dass BlueM die Werte anhand ihrer Zeichenstellung zuweist.

Beispielsweise könnte die Bezeichnung eines Elements zwischen dem 3. und 23. Zeichen einer Zeile stehen, ein dazugehöriger Rohrdurchmesser würde dabei aus dem 30., 31., & 32. Zeichen derselben Zeile ausgelesen.

Die Kodierung der Werte in dieser Form bewirkt eine optisch ansprechende Gestaltung der Tabelle, die den Nutzer zu manuellen Anpassungen geradezu einlädt. Gleichzeitig erschwert es eine manuelle Konstruktion der Datei mit vorhandenen Werten jedoch enorm, da diese mühselig und fehleranfällig ist.

Hier versucht diese Arbeit anzusetzen, um im Rahmen einer Schnittstelle die BlueM-konforme Formatierung von Dateien zu gewährleisten – zum einen aus mit QGIS erarbeiteten Daten, zum anderen aber auch aus üblichen Excel oder CSV-Dateien, die via QGIS umgewandelt werden.

```
*Talsperren (*.TAL)
*==================
*
*|------|------------|------------|-----|----------|----------------------|----------------------|
*| Bez  |  Anfangs-  |  Maximal   | Ge- |  Sohle   |      HW-Entlastung   |      Krone           |
*|      |  volumen   |  volumen   |wicht|          |   Kante       S-Vol  |   Kante       S-Vol  |
*|------|------------|------------|-----|----------|----------------------|----------------------|
*|      | [Tsd.cbm]  | [Tsd.cbm]  |  -  |   [mNN]  |   [mNN]  | [Tsd.cbm] |   [mNN]  | [Tsd.cbm] |
*|-<-->-|-<--------->-|-<--------->-|-<->-|-<------>-|-<------>-|-<--------->-|-<------>-|-<--------->-|
 | T170 |        600 |       3650 |  1  |     114.9 |   118.25 |       2650 |   119.25 |       3650 |
 | THRB |        500 |       2800 |  1  |       116 |      121 |       2500 |      122 |       2800 |
*|------|------------|------------|-----|----------|----------------------|----------------------|
*
0Funktionen für T170
*===================
*
*|------|------------------------------|-------------------------------------|------|-----|----------
*| Bez.-|       Beschreibung           | F u n k t i o n e n                 |      |     |
*|------|                              | Anz. Anz. Anz. Anz.                 |      |     |
*|   -  |                              | Spe  Str  Grz  Abh                  |      |     |
*|-<-->-|-<--------------------------->-|-<->--<->--<->--<->------------------|------|-----|----------
 | T170 | EMTALSPER                    |  0    1    0    0                    |      |     |
*|      |------------------------------|-------------------------------------|------|-----|----------
1|      |----------------------- SPEICHERKENNLINIE -------------------------|------|-----|----------
*|      |                              |  WSP     Volumen    Oberfl.         |      |     |
*|      |                              |  mNN     Tsd.cbm      ha            |      |     |
*|      |------------------------------|-<------>-<--------><------>--------|------|-----|----------
 |      |                              |   114.9          0         0       |      |     |
 |      |                              |     115 0.00273973       0.1       |      |     |
 |      |                              |   116.5  0.2739726       100       |      |     |
```

Abb. 4 Ausschnitt aus einer BlueM-TAL-Datei

2.1.4 Übersicht über die BlueM-Dateitypen

Die 23 Eingangsdaten (engl. input-files) für BlueM können grob in 6 Kategorien eingeteilt werden:

① *ALLGEMEINES (General)*

- *ALL-file* *General Simulation Options*
  Hier werden allgemeine Informationen über die geplante Simulation hinterlegt (Bezeichnung, Zeitraum, etc.), sowie Optionen gesteuert

- *SYS-file* *System Plan*
  Enthält die Systemlogik, welche die Abhängigkeiten zwischen den einzelnen Elementen beschreibt

② *FUNKTIONEN (Functions)*

- *FKT-file* *Functions*
  Allgemeine Funktionen wie beispielsweise Verdunstungsparameter

- *KTR-file* *Control Functions*
  Steuerungsfunktionen für Systemelemente

③ GANGLINIEN (Time Series)

- *EXT-file* *Time Series Management*
  Verwaltet die einzelnen Gangliniendateien und deren Einheiten (l/s etc.)

- *JGG-file* *Annual Series*
  Enthält Niederschlags-Jahresganglinien

- *TGG-file* *Day Series*
  Enthält Niederschlags-Tagesganglinien

- *WGG-file* *Week Series*
  Enthält Niederschlags-Wochenganglinien

④ ELEMENTE (Elements)

- *BEK-file* *Rain Overflow Basin*
  Informationen über Becken, insbesondere Regenüberlaufbecken

- *EIN-file* *Individual Discharges*
  Informationen über Einzeleinleitungen

- *EZG-file* *Natural Catchments*
  Natürliche Einzugsgebiete; Tabelle mit den meisten Spalten: 39

- *FKA-file* *Urban Catchments*
  Informationen über kanalisierte (städtische) Flächen

- *HYA-file* *Hydraulic Elements*
  Hydraulische Elemente (Wehre, Rohrleitungen, Turbinen)

- *RUE-file* *Rain Overflow Discharge*
  Informationen über Regenüberläufe

- *TAL-file* *Dams*
  Informationen über Talsperren; die Datei enthält eine Übersichtstabelle und jeweils 6 Tabellen für jedes Element der Übersichtstabelle

- *TRS-file* *Transport Routes*
  Informationen über Transportelemente (Rohrleitungen, offene Gerinne)

- *URB-file* *Consumer*
  Daten über Wasserverbraucher im System

- *VER-file* *Branching*
  Daten über Verzweigungen im System

⑤ BODENINFORMATIONEN (Soil Information)

- *BOA-file* *Soil Texture*
  Informationen über Bodenarten im Untersuchungsgebiet

- *BOD-file* *Soil Types*
  Informationen über Bodentypen im Untersuchungsgebiet

- *EFL-file* *Elementary Surfaces*
  Listet die einzelnen homogenen Teilflächen (Hydrological Response Unit, kurz HRU) für die spätere Berechnungen auf

- *LNZ-file* *Landuse*
  Liefert Parameter für verschiedene Arten der Landnutzung im Untersuchungsgebiet

⑥ DIFFUSE QUELLEN (Diffuse Sources)

- *DIF-file* *Diffuse Sources*
  Beschreibt einzelne Diffuse Quellen mit deren zu erwartenden Stoffeinträgen, Temperatur und Sauerstoffsättigung

2.2 GIS

2.2.1 Allgemein

Geographische Informationssysteme (kurz GIS) sind Computerprogramme, die versuchen die räumliche Verteilung von Elementen möglichst realitätsnah abzubilden. Dabei wird nicht nur die geographische Ausdehnung der Elemente dargestellt, sondern auch weitergehende Informationen (Attribute) über diese verarbeitet.

Diese Elemente gelangen einzeln oder in thematischen Gruppen als Ebene (engl. „Layer") in das GIS. Für diese Layer wird hauptsächlich zwischen zwei Datentypen unterschieden.

Rasterdaten bestehen aus einem räumlich definierten Feld einzelner Quadrate (Pixel), denen jeweils ein Wert zugeordnet ist – im Prinzip wie bei einem Foto, dessen einzelne Pixel einen Farbwert haben. Ein Beispiel für ein solches Dateiformat ist GeoTIFF. Die Nachteile von Rasterdaten liegen dabei in der fehlenden Skalierbarkeit und dem geringen Informationsgehalt für einzelne Punkte bzw. Elemente – stellen aber oft den einzig verfügbaren Datentyp dar (beispielsweise für Luftbilder).

Das komplexere Gegenstück dazu sind sogenannte Vektordaten. Diese bestehen aus über Koordinaten definierten Punkten, welche wiederum zu Linien und Flächen verbunden sein können. Die dadurch geometrisch exakt definierte Ausdehnung der Elemente ermöglicht verlustfreie Skalierungen und vereinfacht geometrische Operationen mit den dargestellten Entitäten. Gleichzeitig kann jedem durch Koordinaten definierten Bereich oder Element eine theoretisch unbegrenzte Menge an Informationen zugeordnet (engl. „to attribute") werden. Die Datenhaltung erfolgt bei Vektordaten hauptsächlich in Form von ESRI Shapefiles bzw. den GeoPackages des OGC.

Neben der einfachen Darstellung bestehender Informationen, liegt die Hauptaufgabe von geographischen Informationssystemen in der Gewinnung neuer Erkenntnisse aus bereits vorhandenen Daten.

Wenn die einzelnen Layer mit geographischen Daten in einem gemeinsamen Referenzsystem vorliegen, können diese Informationen „übereinander" gelegt und miteinander verschnitten werden. Ein anschauliches Beispiel einer solchen „räumlichen Analyse" bietet der englische Mediziner John Snow [9]:

Als 1854 der Londoner Stadtteil Soho von einer Cholera-Epidemie heimgesucht wurde, dokumentierte Dr. Snow die Wohnorte der Erkrankten auf einem Stadtplan. Ebenfalls wurden dort die zur Trinkwasserversorgung genutzten öffentlichen Pumpen verzeichnet. Dabei wurde schnell deutlich, dass sich die meisten Cholerafälle um die Pumpe in der Broad Street konzentrierten. Snow schloss daraus, dass es sich bei dieser Pumpe um die Quelle der Infektionen handelte. Dies wurde durch einen Rückgang der Erkrankungen nach Sperrung

[9] Snow, J., 1854

(Entfernung des Schwengels) der Pumpe bestätigt, auch wenn Snows Ergebnisse zur damaligen Zeit hoch umstritten waren.

Heutzutage werden Geographische Informationssysteme in einem weiten Spektrum verwendet; insbesondere bei der Auswahl geeigneter Standorte für Maßnahmen. Sei es für die Windkraft, durch Vergleich von Naturschutzgebieten, Entfernung zu Wohnbebauung und Karten mit ortsspezifischen Winddaten. Oder einer neuen Supermarktfiliale durch Analyse bestehender Versorgungsgebiete bzw. der Ermittlung des Kundenpotentials anhand von Bevölkerungsdaten innerhalb eines X-minütigen Bewegungsradius.

Die Vielzahl an Anwendungsmöglichkeiten hat ein breites Angebot an fach-spezifischen und allgemein gehaltenen GIS-Programmen hervorgerufen. Neben dem später genauer behandelten QGIS sind hier vor allem ArcGIS und GeoMedia zu erwähnen, sowie das aus dem Gießener Raum stammende INGRADA.

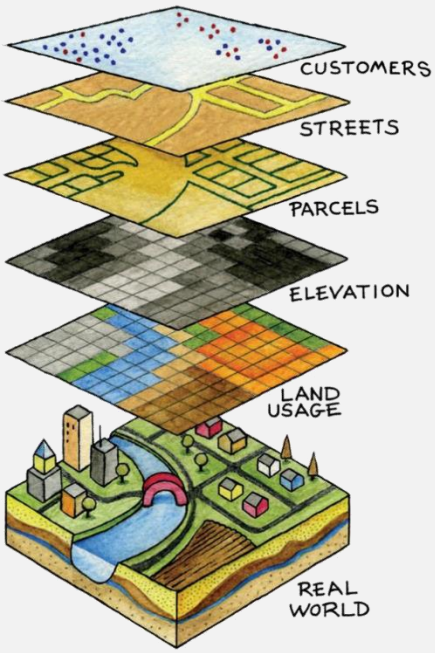


Abb. 5 "Übereinanderlegen" von Daten in einem GIS

2.2.2 GIS in der Hydrologie

Die Hydrologie ist aufgrund der inhärenten Eigenschaften des Untersuchungsobjekts Wasser ein sehr raumgreifendes Wissenschaftsfeld; sei es aufgrund des flächigen Auftretens von Niederschlägen, oder der daraus entstehenden Gewässernetze, welche sich über tausende von Kilometern erstrecken können. Gleichzeitig wird das Wasser in jeder Situation von seinem Umfeld beeinflusst und hat im Gegenzug großen Einfluss auf die umgebenden Räume. Der Einsatz von GIS ist in der Hydrologie daher unerlässlich.

Das Management eines kommunalen Kanalsystems beispielsweise, macht außerhalb eines GIS wenig Sinn, da jede einzelne Leitung direkten Wechselwirkungen mit den anderen Systemelementen ausgesetzt ist und zusätzlich von der Umgebung beeinflusst wird. Die Stärke eines GIS ist dabei, dass die reale Welt möglichst genau dargestellt werden kann und die Übersichtlichkeit durch Abstraktion der Realität (Blick von oben; Leitungen sind nicht unter der Erde versteckt, sondern gut erkennbar) sogar noch gesteigert wird. Gleichzeit sind weitergehende Informationen zu jeder Leitung (Baujahr, Material, Durchmesser, etc.) im Idealfall nur einen Klick entfernt in den Attributen des jeweiligen Elements.

Für weitere, umfangreichere Informationen zur Anwendung von GIS in der Hydrologie wird auf die Bücher „*Distributed Hydrologic Modeling Using GIS*“ [10] und „*GIS and Geocomputation for Water Resource Science and Engineering*“ [11] verwiesen.

Als Beispiel für die Nutzung eines GIS zum Erzeugen von Eingangsdaten für das Programm BlueM wird die Arbeit „Eignung von QGIS für die Erstellung von Eingangsdaten für das Modellsystem BlueM“ [12] von Fabian Schweizer empfohlen. Dort werden die einzelnen Arbeitsschritte zur Erzeugung der HRUs (Hydrological Response Units) für BlueM detailliert aufgeschlüsselt.

2.2.3 Das Programm QGIS

Abb. 6 QGIS - Logo

Laut programmeigener Website [13] ist QGIS (früher auch Quantum-GIS) eine freie open-source Software, welche seit 2002 als Community-Projekt entwickelt wird.

Die kostenlose Verfügbarkeit unterscheidet es dabei von den anderen, bereits genannten Systemen. Dies führt zusammen mit der offenen Philosophie zu einer weiten Verbreitung von QGIS im akademischen Milieu. Wobei es seit wenigen Jahren auch Gerüchte über die Bewegung von privatwirtschaftlichen und öffentlichen GIS-Anwendern in Richtung QGIS gibt.

Diese wurden bisher oftmals von dieser „Offenheit“ eher abgeschreckt, da es beispielsweise bedeutete, dass auftretende Betriebsstörungen oder unvorhergesehene Problemstellungen nicht durch einen zentralen Kundenservice gelöst werden können, wie dies bei den kommerziellen Programmen ArcGIS (ESRI) und GeoMedia (Hexagon/Intergraph) von Firmen mit jeweils über 3.000 Mitarbeitern der Fall wäre.

Bei QGIS ist vielmehr die „Community“ der Ansprechpartner. Da diese sehr umfangreich ist, findet man bei einem Problem in der Regel online sehr schnell einen anderen Nutzer, der vor einiger Zeit ein ähnliches Problem hatte, welches bereits von der Community gelöst wurde. Dies ist jedoch keine Garantie und selbst eine gefundene Lösung erfordert oftmals ein tieferes Verständnis für das zugrundeliegende Problem. Für einen forschenden oder hobbymäßigen Nutzer ist dies kein Ausschlusskriterium, für ein risikoaverses Unternehmen, dass auf planbare Abläufe angewiesen ist, jedoch schon.

Ein anschauliches Beispiel für ein solches Problem ist der schon länger bestehende Bug bei mathematischen Operationen mit Dezimalzahlen:

[10] Vieux, B. E., 2016
[11] Dixon, B. & Uddameri, V., 2016
[12] Schweizer, F., 2021
[13] QGIS-Community, 2022

Die Berechnung von 0,2 + 0,1 in einer QGIS Attributtabelle würde beispielsweise das Ergebnis 0,30000000000000004 liefern. Die überzähligen Nachkommastellen lassen sich hier auf ein interessantes zahlentheoretisches Problem zurückführen.

EXKURS: Brüche können nur korrekt als Kommazahl dargestellt werden, wenn der Nenner des Bruches ein Primfaktor (oder vielfaches davon) der Basis des genutzten Zahlensystems ist.

Bei dem üblicherweise genutzten Zahlensystem ist die Basis 10 (dezimal), mit den Primfaktoren 2 und 5. Das bedeutet die Brüche 1/2, 1/4, 1/5, 1/6, 1/8, 1/10 können als Kommazahl dargestellt werden. Die Brüche 1/3, 1/7, 1/9 hingegen nicht, da sie unendlich viele Nachkommastellen hätten.

Ein Computer benutzt hingegen ein Zahlensystem mit der Basis 2 (binär), welche nur 2 als Primfaktor besitzt. Das bedeutet, dass Brüche wie 1/5 oder 1/10 in Binärcode nicht mit einer endlichen Zahl an Nachkommastellen korrekt ausgedrückt werden können, sondern gerundet werden müssen, um überhaupt abgespeichert werden zu können. Durch dieses Runden geht jedoch der exakte Wert verloren, was wiederum das Berechnungsergebnis verfälscht.

Die für Anwender einfache Addition der Dezimalzahlen 0,2 und 0,1 (1/5 bzw. 1/10) stellt den Computer daher vor ein komplexeres Problem als auf den ersten (menschlichen) Blick ersichtlich. Im Prinzip ist es als würde man einem handelsüblichen Taschenrechner die Aufgabe 0,3333333 + 0,6666666 stellen und sich dann wundern, dass das Ergebnis nicht 1 ist.

Solche Probleme wurden bei kommerziellen Softwareprodukten vermutlich schon in der Betaphase erkannt und behoben, da sie für den beruflichen Anwender zu unangenehmen Rückfragen bei der Abgabe von Projekten führen dürften. Bei QGIS besteht dieses Problem jedoch bereits seit Jahren, da erwartet wird, dass der informierte Nutzer weiß, wie er solche Probleme überwinden kann – es in manchen Kreisen ja sogar als Nachweis der Kompetenz des Anwenders gilt.

Insgesamt kann QGIS jedoch als robustes und verlässliches Programm angesehen werden, dass durch seine Offenheit gut für akademische Projekte geeignet ist – insbesondere für die Zusammenarbeit mit dem ebenfalls offenen BlueM.

2.2.4 Was ist ein Plugin?

Neben den von der Community entwickelten, „offiziellen" Anwendungen, die Teil der jeweiligen QGIS-Version sind, hat der Nutzer auch die Möglichkeit die

Funktionalitäten des Programms nach eigenen Wünschen zu erweitern. Dies geschieht über sogenannte Plugins, die entweder manuell installiert oder aus dem QGIS Plugin Repository bezogen werden können.

Diese Plugins sind kostenlos und meistens auf sehr spezielle Anwendungsszenarien ausgerichtet, die nicht in der standard QGIS-Version behandelt werden, da sie nur von einer kleinen Untergruppe der Nutzer benötigt werden.

Am nächsten kommt dieses System dabei den herunterladbaren Apps für Smartphones: Ein gewöhnliches Smartphone umfasst werksmäßig Applikationen zum Telefonieren, Fotografieren und einen Internetbrowser, da diese von praktisch jedem genutzt werden – dies entspricht QGIS mit seinen Standardfunktionalitäten. Der jeweilige Besitzer kann zusätzlich Apps installieren, die spezifische Aufgaben übernehmen, die ihm persönlich wichtig sind, aber nicht von jedem benötigt werden. Diese Apps entsprechen den sogenannten Plugins und der App-Store / Play-Store / etc. dem Plugin Repository.

QGIS Plugins werden in der Regel von unabhängigen Developern mit Python oder C++ entwickelt. Im aktuellen Repository (*www.plugins.qgis.org* ; Stand 24.01.2022) finden sich 1.574 Python Plugins, welche wiederum zwischen wenigen Dutzend bis zu 3,5 Millionen (für das *OpenLayers-Plugin*) Downloads verzeichnen.

Da die Entwicklung einer BlueM-Schnittstelle für QGIS als Nischenanwendung gut zur genannten Definition passt, wurde beschloss diese als Plugin in QGIS zu realisieren.

Alternativ wäre auch ein eigenständiges Programm denkbar gewesen, welches direkt CSV-, Excel- oder Shape-Dateien verarbeiten würde. Dies hätte jedoch einen deutlich höheren Programmier-Aufwand, ohne wirklichen Mehrwert für den späteren Anwender bedeutet. Vielmehr ist es so, dass ein QGIS-Plugin allein schon durch die Bekanntheit & den Ruf des Basis-Programms deutlich bessere Verbreitungschancen hat, als es eine Standalone-Lösung hätte.

Ein weiteres Zeichen für die Vitalität der QGIS-Community und insbesondere der Plugin-Szene ist das Programm „Plugin-Builder". Dieses bereits über 60.000-Mal heruntergeladene Plugin ermöglicht die automatische Generierung eines mit QGIS verknüpften Startpunktes zur weiteren Entwicklung eines Plugins.

Der standardisierte Ansatz sichert dabei die Kompatibilität zwischen Plugin und QGIS, bietet einen Anknüpfungspunkt für andere Anwendungen und vereinfacht den Veröffentlichungsprozess im QGIS Repository.

Es wurde darauf geachtet den durch den Plugin-Builder (Version 3.2.1) erzeugten Code (weniger als 200 Zeilen) strikt von dem Code zu trennen, der im Rahmen dieser Abschlussarbeit erzeugt wurde (über 3.000 Zeilen). Letzterer wurde daher in eine separate Datei (plugin_functions.py) ausgelagert und durch kleinere Anpassungen des Builder-Codes mit diesem verknüpft.

2.3 Python

Abb. 7 Python - Logo

Python ist, ähnlich wie das bereits erwähnte Fortran, eine höhere Programmiersprache. Das Adjektiv „höher" beschreibt dabei das Abstraktionslevel der Sprache, wobei „höher" ein besseres intuitives Verständnis des geschriebenen Codes durch den Leser bedeutet. Ein niedriges Level der Abstraktion, wie beispielsweise bei Machine-Code oder Assembly Language, ist hingegen für den Menschen auf den ersten Blick nur schwer zu verstehen, jedoch im Allgemeinen kürzer, präziser und für die Hardware schneller umzusetzen.

Die Anfang der 90er Jahre durch den Niederländer Guido von Rossum entwickelte Sprache liegt mittlerweile (seit 2008) in der 3. Version vor und wird durch die gemeinnützige Python Software Foundation (PSF) gemanagt [14]. Python hat es dabei geschafft, laut PYPL-Index im Januar 2022, die mit großem Abstand vor Java beliebteste Programmiersprache weltweit zu sein (28,74% Marktanteil).

Dies liegt unter anderem an der einsteigerfreundlichen Struktur der Sprache, die aber gleichzeitig auch leistungsfähig genug ist, um komplexe Vorgänge mit großen Datenmengen zu bearbeiten.

Die sehr offene & freie Gestaltung der Python Lizenzierung durch die PSF hat dazu geführt, dass sich um Python eine sehr breit aufgestellte Community gebildet hat, welche mit Ihren Beiträgen nur zur weiteren Steigerung der Attraktivität dieser Sprache beiträgt.

Ein gutes Beispiel für diese Beiträge sind die zahl- & kostenlosen Module, welche in Python eingebunden werden können. Beispielhaft erwähnt seien hier „NumPy" (für mehrdimensionale Daten-Arrays), „csv" (für die Behandlung von CSV-Dateien) und „Matplotlib" (für mathematische Grafiken).

Ähnlich wie bei QGIS bietet auch diese Community online viele Ressourcen zur Problemlösung und auch praktische Hinweise, wie man seinen Code mehr „pythonic" gestalten kann.

Da QGIS selbst (neben C++) größtenteils mit Python entwickelt wurde und die Sprache, dank ihrer weiten Verbreitung und großen Community, für zukünftige Weiterentwicklungen bestens geeignet erscheint, wurde beschlossen, die BlueM-Schnittstelle für QGIS mit Python 3 zu programmieren.

Als Entwicklungsumgebung (*integrated development environment* – „IDE") wurde sich für PyCharm der tschechischen Softwarefirma JetBrains entschieden. Eine Alternative wäre Microsofts Visual Studio Code gewesen.

Der in Python geschriebene Code des Plugins befindet sich im Anhang.

[14] The Python Software Foundation. 2022

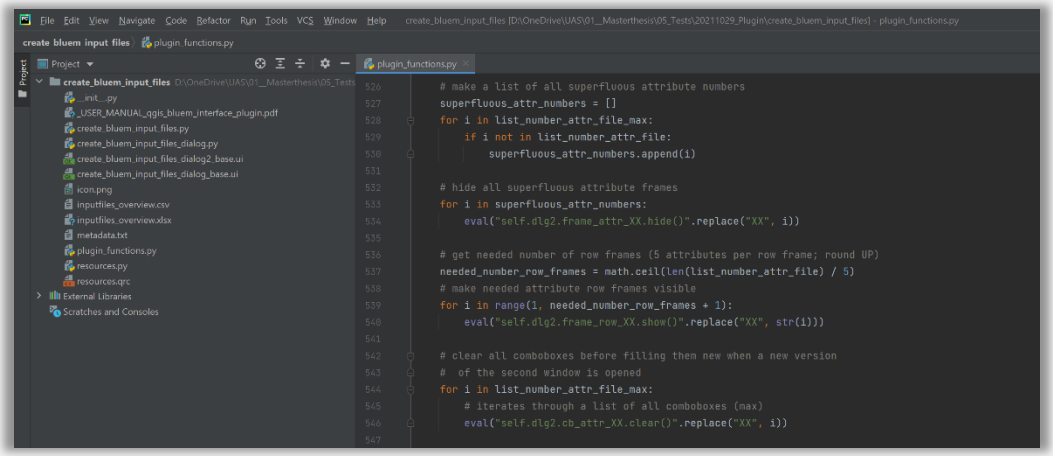


Abb. 8 Python Code in PyCharm

2.4 Qt Designer

Als Toolkit zur Entwicklung der grafischen Benutzeroberfläche (graphical user interface – kurz GUI) des Plugins wurde Qt (gesprochen wie engl. *„cute"*) ausgewählt. Dieses ist wie QGIS ebenfalls quelloffen und kann für nicht-kommerzielle Anwendungen kostenlos verwendet werden. Qt selbst verwendet C++ bzw. XML-Code, bietet aber mit „PyQt" eine sogenannte Anbindung an Python, die es ermöglicht das Userinterface mit Python-Befehlen anzusteuern.

Neben bekannten Anwendungen wie *Teamviewer*, *Google Earth* und *Telegram* (desktop client) verwendet QGIS selbst Qt als Bibliothek für grafische Kontrollelemente (genannt Widgets). Die Interaktion geht so weit, dass Qt Designer beim Download von QGIS standardmäßig mitgeliefert wird und diese Version sogar spezielle Widgets für die Anwendung in QGIS-Plugins bereitstellt.

Eine mögliche Alternative wäre das ebenfalls freie Toolkit von *Tkinter* gewesen.

Der XML-Code der Benutzeroberflächen ist leider zu umfassend für den Anhang dieser Arbeit und ist daher nur in den beiden UI-files einzusehen.

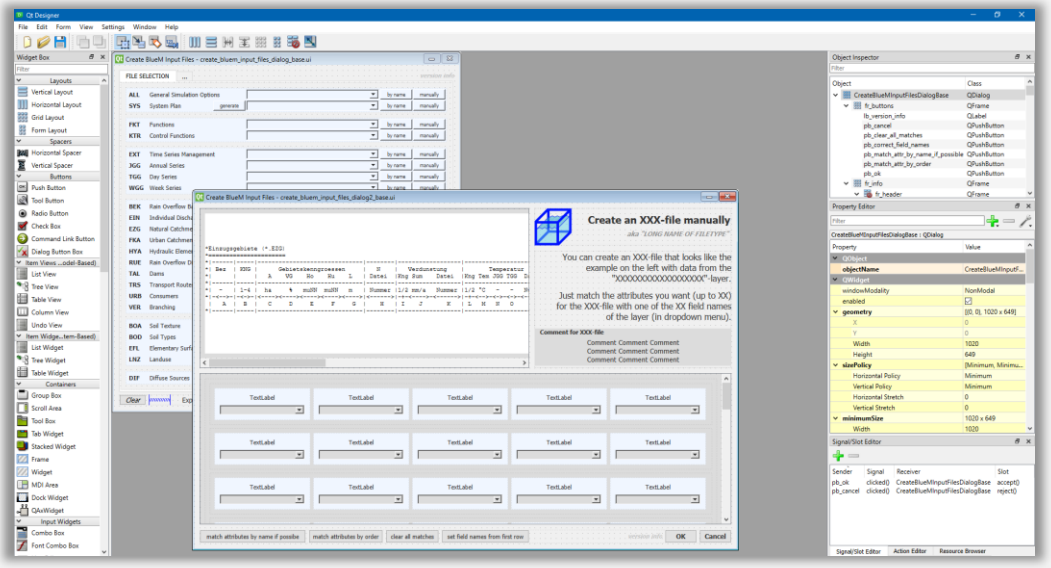


Abb. 9 Bearbeitung der Benutzeroberfläche in Qt Designer

3. Anwendung der Schnittstelle

Auf den folgenden Seiten werden zuerst die Anwendungsmöglichkeiten der QGIS-Schnittstelle für das NA-Modell BlueM gezeigt. Danach werden die zugrundeliegende Funktionsschemata des Plugins (Kapitel 4) und dessen technische Details (Kapitel 5) erläutert.

3.1 Export von BlueM-Dateien

Nach dem Klick auf das BlueM-Icon in der QGIS-Toolbar öffnet sich für den Anwender dieses Dialogfenster:

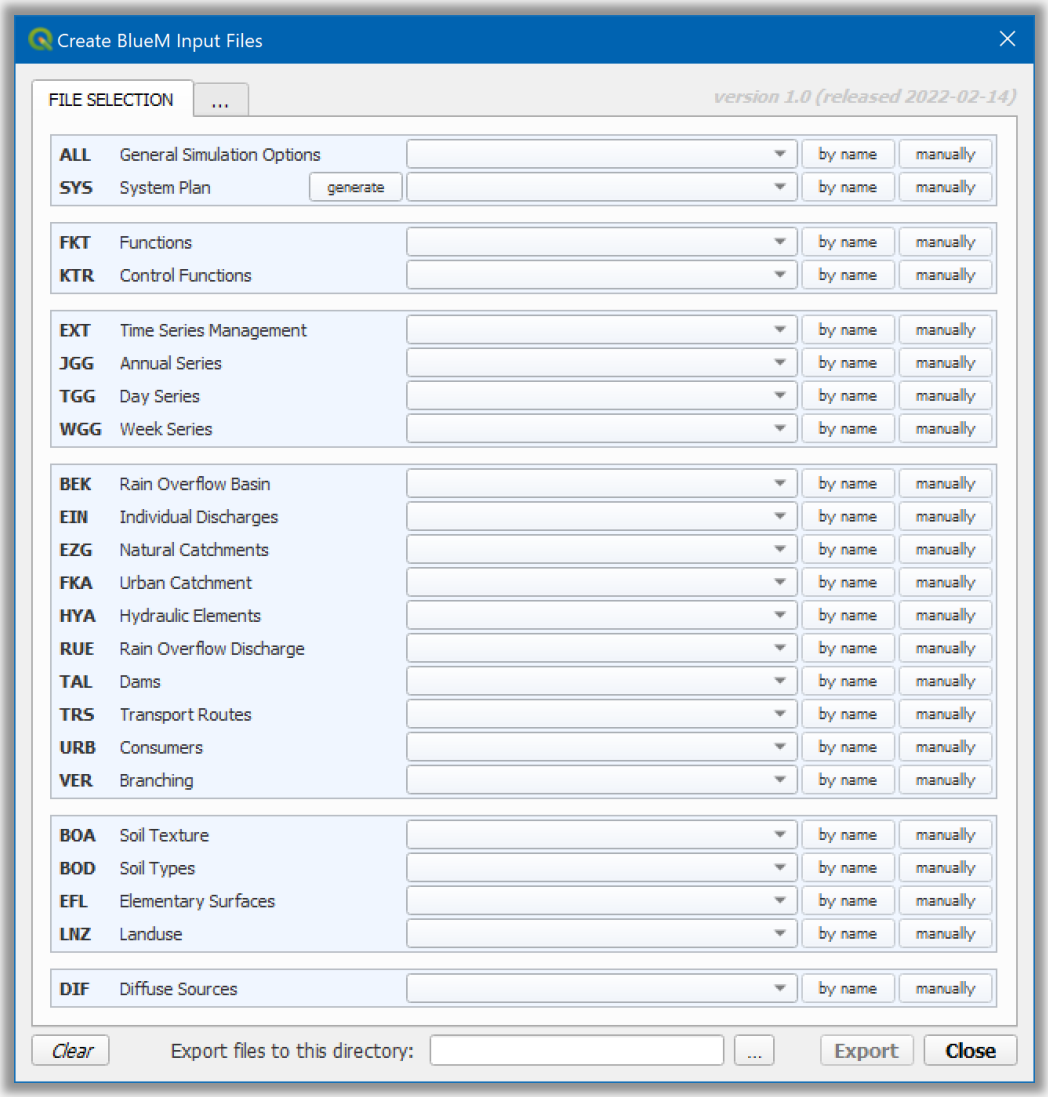


Abb. 10 Erstes Dialogfenster, Tab 1

Die dargestellte Benutzeroberfläche ist für eine schnelle und übersichtliche Nutzung durch einen informierten Anwender optimiert; dies führt dazu, dass die

große Anzahl an Auswahlfeldern und Knöpfen auf einen neuen Anwender zuerst überfordernd wirken kann. Dieses Gefühl sollte jedoch schnell verfliegen, wenn die zugrundeliegende Systematik der Benutzeroberfläche erläutert wurde.

Im Kern dieses Fensters steht eine Tabelle. Die einzelnen Zeilen dieser Tabelle stellen die 23 BlueM-Dateitypen dar, die mit diesem Plugin erzeugt werden können. Die einzelnen Zeilen sind wiederum in Gruppen von verwandten Dateitypen (blaue Kästen) sortiert.

In jeder Zeile steht das Kürzel des Dateityps, gefolgt von dessen englischer Bezeichnung. Rechts davon befindet sich ein Dropdownmenü (auch „Combobox" genannt); in diesem kann ein Vektorlayer bzw. Layer ohne Geometrie des aktuellen QGIS-Projekts ausgewählt werden. Dieser Layer sollte die Informationen enthalten, die in die jeweilige BlueM-Datei umgeformt werden sollen.

Der *„generate"*-Button der SYS-Datei kann vorerst ignoriert werden und wird in einem späteren Kapitel näher erläutert.

Sobald für einen Dateityp ein Layer ausgewählt wurde, werden die sich rechts daneben befindenden beiden Buttons *„by name"* und *„manually"* freigeschaltet. Diese sind dazu da, die einzelnen Spalten der Attributtabelle des Layers den entsprechenden Attributen der zu erzeugenden BlueM-Datei zuzuweisen.

Der *„by name"*-Button überprüft die Überschriften der Attributtabelle des Layers und wenn er dort Spalten findet, die den gleichen Titel wie ein Attribut der BlueM-Datei haben, wird der Inhalt der jeweiligen Spalte dem entsprechenden BlueM-Attribut zugeordnet.

Eine komplexere Art der Zuordnung bietet der *„manually"*-Button, dieser öffnet dazu ein zweites Dialogfenster, welches für den jeweiligen BlueM-Dateityp spezifisch ausgestattet ist:

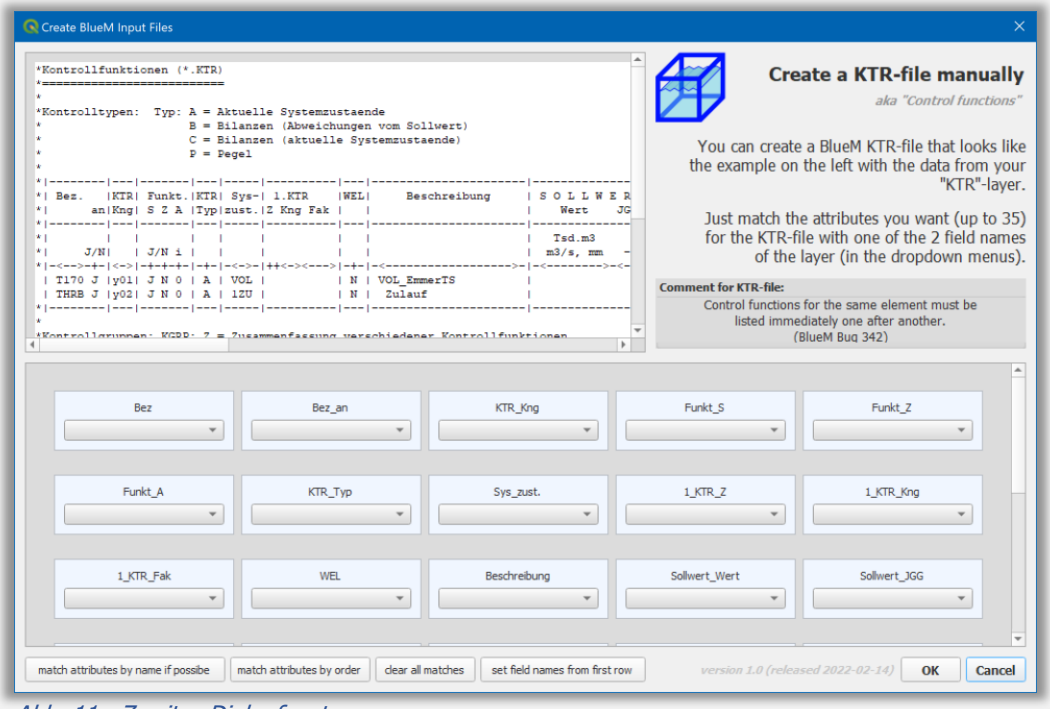


Abb. 11 Zweites Dialogfenster

Dieses Dialogfenster besteht aus 4 Elementen:

- Rechs oben befindet sich ein Textblock mit Informationen zum aktuellen Dateityp und dem dafür gewählten Layer.
- Links oben kann ein Beispiel für den ausgewählten Dateityp angesehen werden. Da die Beispiele sehr unterschiedliche Größen haben, sind sie in einem scroll-baren Bereich eingefügt.
- Der Hauptteil in der Mitte des Fensters wird von einem Block eingenommen, der alle Attributbezeichnungen der aktuell ausgewählten BlueM-Datei verzeichnet. Da die Anzahl der Attribute je nach Dateityp stark schwankt, ist hier gegebenenfalls auch Scrollen möglich.
- Am unteren Rand des Fensters finden sich Buttons zum weiteren Steuern des Plugins.

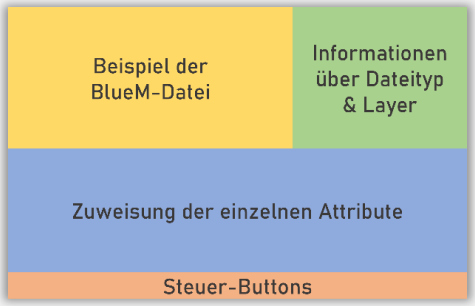


Abb. 12 Schematische Aufteilung des zweiten Dialogfensters

Der Anwender kann nun für jedes einzelne BlueM-Attribut in einem Dropdownmenü eine Layer-Spalte manuell zuordnen.

Dieser, bei bis zu 80 BlueM-Attributen sehr arbeitsintensive Vorgang, kann durch das Plugin unterstützt werden, indem via Button versucht wird Attribute anhand ihrer Namen zuzuweisen. Alternativ können auch alle Layer-Spalten ihrer Reihenfolge nach den BlueM-Attributen zugewiesen werden. Die Zuweisungen können dann manuell weiter verfeinert oder per Klick auf den entsprechenden *„clear all matches"*-Button zurückgesetzt werden.

Sobald der Anwender mit der Zuweisung der Attribute fertig ist, kann er diese mit einem Klick auf *„OK"* speichern oder via *„Cancel"* verwerfen.

Das Verfahren „Layer auswählen", dann „Attribute zuweisen" muss nun für alle BlueM-Dateitypen wiederholt werden, die exportiert werden sollen.

Danach kann der Anwender am unteren Rand des ersten Dialogfensters einen Pfad zu einem Zielordner eingeben (bzw. über das Kontextmenü auswählen) und den „Export"-Button drücken.

Nun werden alle BlueM-Dateien, für die ein „by name" oder „manually"-Button aktiviert ist, mit den Daten des zugewiesenen Layers gefüllt und in das gewählte Verzeichnis geschrieben.

Standardmäßig exportiert das Plugin zusätzlich eine Datei mit detaillierten Informationen über den Exportvorgang („Export_Log"). Danach schließt sich das Plugin selbstständig und öffnet ein Windows-Fenster des gewählten Exportverzeichnisses.

3.2 Einstellungen

Der im vorherigen Kapitel beschriebene Exportvorgang kann durch den Anwender durch verschiedene Einstellungsmöglichkeiten beeinflusst werden. Diese befinden sich auf dem zweiten Tab des ersten Dialogfensters, welches über die drei Punkte rechts neben „FILE SELECTION" erreicht werden kann.

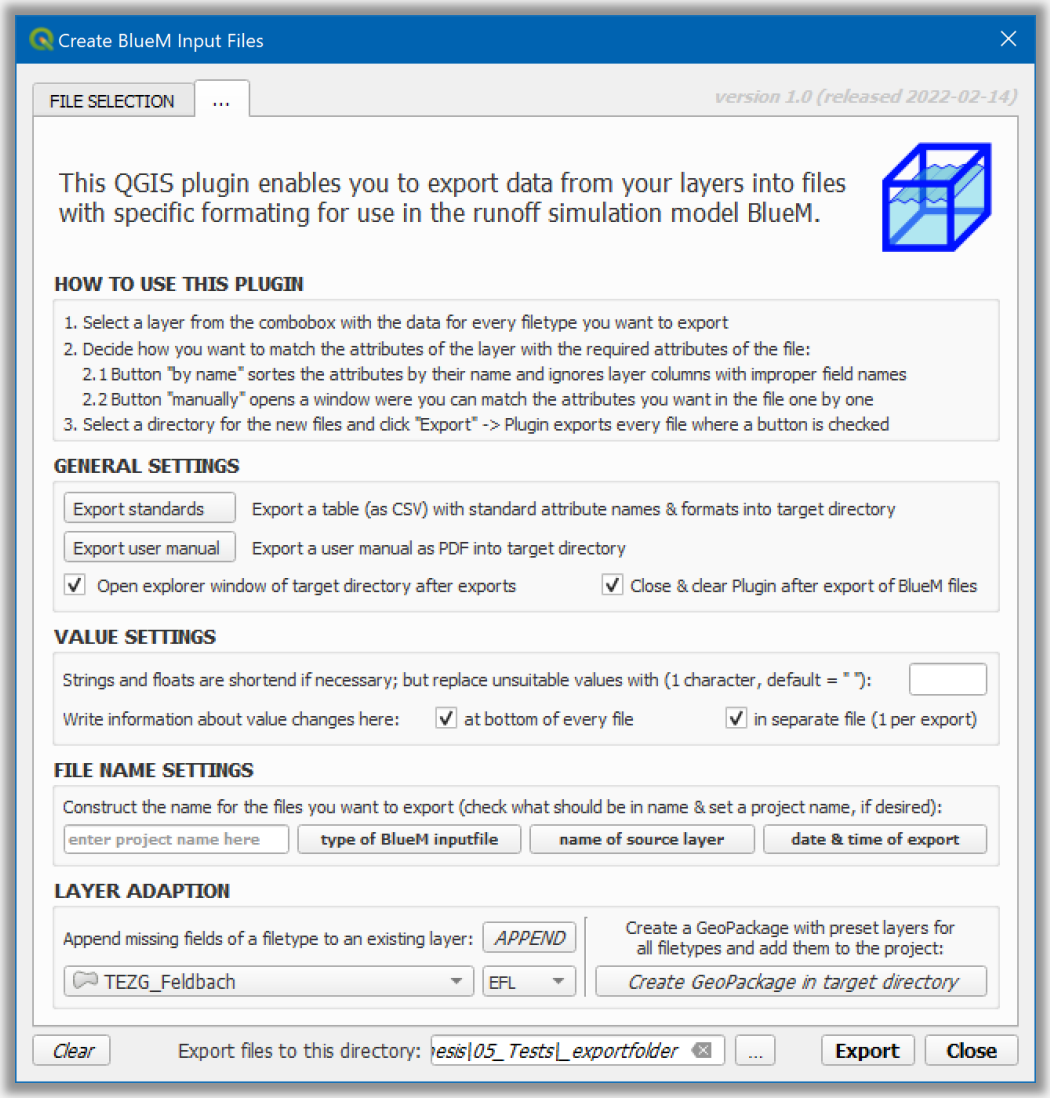


Abb. 13 Erstes Dialogfenster, Tab 2

Nach einer kurzen Anwendungsbeschreibung bietet sich dem Anwender unter **„GENERAL SETTINGS"** die Möglichkeit eine CSV-Datei zu exportieren, die alle 428 Attributdefinitionen der 23 BlueM-Dateien enthält. Diese können als Grundlage für die Erzeugung einer Excel- oder CSV-Datei genutzt werden, die wiederum in QGIS einen neuen Quell-Layer bilden kann.

Weiterhin gibt es dort die Möglichkeit ein kurzes Benutzerhandbuch (englisch) zu exportieren, sowie zu entscheiden, ob nach dem Export von BlueM-Dateien ein entsprechendes Windowsfenster geöffnet und das Plugin geschlossen werden soll.

In den **„VALUE SETTINGS"** kann der Anwender definieren welches Zeichen (standardmäßig Leerzeichen) anstelle von nicht in BlueM schreibbaren Werten eingesetzt werden soll. Beispielsweise für zu große Zahlen oder ungültige Datumsangaben.

Diese und andere Werteänderungen werden zusammen mit weiteren Informationen während des Exportvorgangs gesammelt. Der Anwender kann hier entscheiden, ob diese Export-Informationen in die jeweilige BlueM-Datei und / oder eine Sammlung für alle Dateien („Export_LOG") geschrieben werden soll.

Mit den **„FILE NAME SETTINGS"** ist es dem Anwender möglich, die Dateinamen der ausgegeben BlueM-Dateien zu beeinflussen. Es kann ein einheitlicher Projektname vergeben werden, sowie einzelne Informationen über den jeweiligen Dateityp, den Namen des Quell-Layers und den Exportzeitpunkt angefügt werden.

Eine BlueM-Datei kann daher standardmäßig als *„unnamed.tal"* oder als *„ProjektXY_TAL_file_from_Example_Layer_20220214_1230.tal"* ausgeben werden.

3.3 Weitere Tools

Neben dem Export von Layer-Daten in einem von BlueM verwendbaren Format, bietet das Plugin noch weitere Funktionalitäten, um für den Anwender die Erzeugung von BlueM-Daten komfortabler zu gestalten:

→ Korrektur fehlerhafter Spaltentitel im Layer

Beim Einlesen von Excel-Dateien in QGIS kann es dazu kommen, dass die erste Zeile der Daten nicht als die Überschriften der Spalten der Attributtabelle erkannt werden. Stattdessen nennt QGIS diese Spalten dann „Field1", „Field2", etc.

Dies steht einer automatisierten Zuweisung der Layer-Spalten zu den gesuchten Attributen des BlueM-Dateityps im Wege und auch eine manuelle Zuweisung wird dadurch verkompliziert.

Sollte das Plugin solche fehlerhaft eingelesen Datensätze erkennen, wird im zweiten Dialogfenster automatisch ein Button sichtbar, der mit einem Klick die Spalten der Attributtabelle des selektierten Layers in die Eintragungen aus dessen erster Zeile umbenennt. Damit kann dann die übliche automatisierte Zuweisung erfolgen.

→ Adaption vorhandener Layer

Ein bereits im Projekt vorhandener Layer enthält eventuell noch nicht alle, oder sogar gar keine, der für eine bestimmte BlueM-Datei benötigten Spalten.

Um dieses Problem zu lösen, kann der genannte Layer im zweiten Tab des ersten Dialogfensters unter „LAYER ADAPTION" ausgewählt werden. Danach muss nur noch der gewünschte Dateityp ausgewählt und auf den Button „APPEND" gedrückt werden.

Das Plugin wird nun automatisch alle für den gewählten Dateityp möglichen Spalten dem vorgenannten Layer hinzufügen. Sollten bereits passende Spalten vorhanden sein, werden diese beibehalten und nicht zusätzlich hinzugefügt.

Die neu hinzugefügten Spalten werden zusätzlich den Anforderungen entsprechend definiert, sodass beispielsweise in einer Spalte, die in der späteren BlueM-Datei einen Integer (Ganzzahl) darstellen soll, im Layer keine Buchstaben eingetragen werden können.

→ Dropdownmenüs in SYS-Attributtabelle

Während ein Layer für die Benutzung als Grundlage für das SYS-file adaptiert wird, werden in dessen Attributtabelle die 3 „Zulauf" und 3 „Ablauf"-Spalten automatisch in Dropdownmenüs umformatiert. In diesen Dropdownmenüs befinden sich dann alle in der „Nr"-Spalte eingetragenen Systemelemente. Dadurch wird das manuelle Ausfüllen der 6 genannten Spalten deutlich vereinfacht.

→ Erzeugen eines GeoPackages mit Quell-Layern

Neben der Anpassung vorhandener Layer gibt es bei der „LAYER ADAPTION" auch die Möglichkeit, durch Knopfdruck ein neues GeoPackage im gewählten Zielverzeichnis zu generieren.

Dieses GeoPackage enthält 23 einzelne Layer, die für jeweils einen der BlueM-Dateitypen optimiert sind (analog zur Adaption vorhandener Layer).

Das GeoPackage wird standardmäßig *„GPKG_BlueM_Inputfiles"* genannt, alternativ kann aber auch über das Feld *„enter project name here"* der „FILE NAME SETTINGS" ein Projektname gesetzt werden, der sich im GeoPackage wiederfindet.

Es ist zu beachten, dass die Erzeugung des ca. 408 KB großen GeoPackages einige Zeit (je nach System etwa 20 Sekunden) in Anspruch nimmt; der Prozess kann im automatisch geöffneten Windowsfenster verfolgt werden. Nach Abschluss des Exports wird das GeoPackage automatisch dem aktuellen Projekt hinzugefügt, wobei der Anwender im Importdialog entscheiden kann, welche neuen Layer er im Projekt haben möchte.

→ Automatisches Generieren eines SYS-Layers

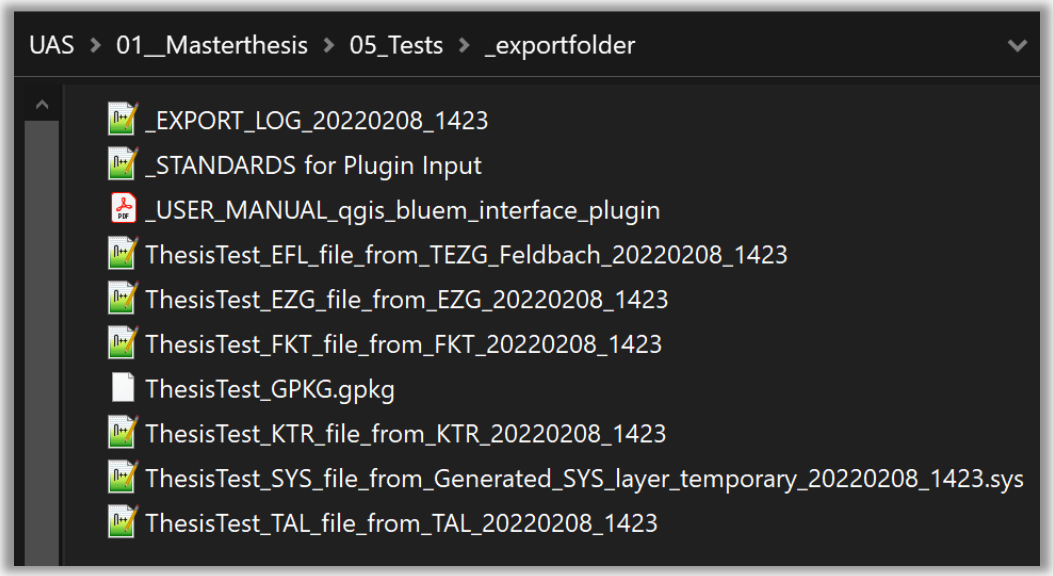
Die Nutzung von Dropdownmenüs bei der Erarbeitung der SYS-Daten ist bereits eine enorme Erleichterung für den Anwender, es gibt jedoch ebenfalls eine Möglichkeit, die Abbildung der Systemverbindung (Systemplan) komplett automatisiert zu erzeugen:

Bei einem Klick auf den *„generate"*-Button neben der SYS-Layerselektion werden automatisch alle für die 10 Element-Dateien selektierten Layer durchsucht. Dabei werden alle enthaltenen Elemente mitsamt ihren in der *„Output_to"*-Spalte aufgelisteten Abflüssen (einzelne Abflüsse können durch Leerzeichen, Komma oder Semikolon getrennt sein) verzeichnet und entsprechend in einem neuen Layer *(„Generated_SYS_layer_temporary")* einsortiert.

Die Werte für die entsprechenden Zulaufspalten werden ebenfalls automatisch errechnet, der neue Layer in der Layerselektion ausgewählt und der „by name"-Button gedrückt. Weitere Eintragungen im Layer sind jedoch möglich.

Neben den hier erfolgten Erläuterungen zu den Anwendungsmöglichkeiten wird ebenfalls auf die im Plugin befindlichen Tooltips (Mauszeiger über Elementen schweben lassen), sowie das Benutzerhandbuch (englisch) verwiesen.

Abb. 14 Zielordner nach verschiedenen Exporten durch das Plugin



4. Das Plugin

4.1 Programmatische Grundlagen des Plugins

Das vorliegende Python-Plugin für QGIS wurde nach dem KISS-Prinzip (in der Variante *„Keep it simple and straightforward"*) konzipiert. Es wurde daher besonderer Wert darauf gelegt, dass die einzelnen Algorithmen im Code in wiederkehrenden Mustern aufgebaut sind und dabei Schritt für Schritt vorgehen, auch wenn Python eine kompaktere und performantere Schreibweise zulassen würde.

Performanz ist jedoch bei den vorgesehenen Standardprozessen aufgrund der relativ geringen Datenmengen kein Zwangspunkt. Ebenfalls wurde jeder Schritt im Code mit ausführlichen Kommentaren versehen.

Die verschiedenen Arbeitsabläufe im Code sind dabei in einzelne Funktionen aufgeteilt, die jeweils bei Bedarf mit den relevanten Parametern angesteuert werden. Auch sich wiederholende Abläufe wurden in eigene Funktionen ausgelagert, um wiederum von unterschiedlichen Prozessen genutzt werden zu können (Stichwort „Modularität").

Dies alles vereinfacht ein zukünftiges Weiterentwickeln des Plugin Codes durch die BlueM-Community (bspw. via GitHub).

Ebenfalls um künftige Anpassungen zu erleichtern, sind beinahe alle für das Plugin relevanten Parameter in einer einfach zu erreichenden CSV-Datei definiert. In dieser können ohne jegliche Python-Kenntnisse für alle BlueM-Dateitypen die Bezeichnungen der Attribute und die dazugehörigen Definitionen verändert werden. Dabei können sogar Attribute entfernt oder neu hinzugefügt werden. Voraussetzung dafür ist die ebenfalls in dieser Definitionsdatei vorzunehmende Anpassung der Tabellenstrukturen der BlueM-Dateien (Näheres dazu im Kapitel „Definitionsdatei").

Neben den Parametern zur Dateierzeugung ist auch die Benutzeroberfläche selbst für eine einfache Anpassung konzipiert: Sämtliche Dialoge der manuellen Zuweisung der 23 Dateitypen sind nicht einzeln in Qt programmiert, sondern es gibt lediglich eine Version des Zuweisungsdialogs, welche jeweils mit den Daten des selektierten QGIS-Layers und den Dateityp-Parametern aus der Definitionsdatei für die aktuelle Aufgabe angepasst wird. Eine Änderung dieser Benutzeroberfläche ist daher seltener nötig und wenn doch, für alle Dateitypen in einem Arbeitsschritt durchführbar.

Um das Vertrauen des Anwenders in die Professionalität des Plugins hoch zu halten, wurde besonders auf die grammatikalische Korrektheit der englischsprachigen Texte im Plugin geachtet.

So wurde beispielsweise Wert darauf gelegt bei den spezifisch konstruierten Titeln des zweiten Dialogfensters automatisiert den korrekten Artikel (*„a"* bzw. *„an"*) vor den Namen des Dateityps zu setzten. Ebenfalls wurden vordefinierte

Informationstexte passend zu automatisch eingesetzten Zahlenwerten definiert. Beispiel: „No values…", „1 value…", „2 values…" in den Exportinformationen.

Ein weiteres besonderes Charakteristikum des Plugins ist die Datenhaltung. Üblicherweise werden Daten, die von einer Datei in eine andere Datei formatiert und dabei überprüft werden sollen, in einer Datenbank zwischengespeichert.

Dies ist hier anders. Aus der automatisierten oder durch den Nutzer manuell erfolgten Zuordnung der Layer-Daten zu den entsprechenden Attributen des gewünschten BlueM-Dateityps, wird ein Verzeichnis erzeugt. Die einzelnen Layer-Features entsprechen dabei den Zeilen der späteren BlueM-Datei. So ist jedem gewünschten Wert für die spätere BlueM-Datei eine konkrete Zelle der Attributtabelle des Layers zugeordnet.

Beim Generieren der BlueM-Datei für den Export wird dann Zeile für Zeile und Spalte für Spalte, die jeweils an diesem Punkt in die Datei zu schreibende Zelle, bestimmt. Über das Zuordnungsverzeichnis wird der entsprechende Wert im Layer ermittelt, *„on-the-fly*" anhand der Definitionsdatei überprüft, gegebenenfalls angepasst und dann in das passende Dateimuster eingefügt.

Das Programmierparadigma des Plugins ist als imperativ zu bezeichnen, da verschiedene Vorgänge strikt definierten Abläufen bzw. Prozeduren folgen. Es ist dabei allerdings nicht klar abzugrenzen, ob das Programm als „prozedural" oder „objektorientiert" zu beschreiben ist, da es sich in einer Grauzone dazwischen befindet. Einerseits kann die Fixierung auf die Layer-Daten und BlueM-Dateien als „Objektorientierung" ausgelegt werden. Andererseits sind diese Objekte nur Protagonisten der angewandten Zuordnungs- und Bearbeitungsprozeduren. Einen tieferen Einblick in die Begriffsdefinitionen bietet hier die Ausarbeitung *„Programming Paradigms […]*" von Greg Michaelson (2020). [15]

Die einzelnen Abläufe des Plugins werden in den folgenden Kapiteln näher erläutert, zusätzlich wird als Teil der „Philosophie" des Plugins auf das Kapitel zur Fehlerresilienz (4.5) verwiesen.

4.2 Funktionsschema des Exports von BlueM-Dateien

Nach dem Start des Plugins entscheidet der Anwender, welche BlueM-Dateitypen er generieren möchte. Für jeden dieser Dateitypen selektiert er einen Layer und wählt die Zuordnungsmethode aus. Entweder werden die Spalten des Layers anhand ihrer Bezeichnungen automatisch den späteren Attributen der BlueM-Datei zugeordnet oder es findet eine manuelle Zuordnung (unterstützt durch weitere Algorithmen) im zweiten Dialogfenster statt.

Die vorgenommenen Zuordnungen werden in einem sogenannten Dictionary in der Form *„[Schlüssel]: [Wert]*"spezifisch für diesen Dateityp verzeichnet. Dabei bekommt jedes Attribut der BlueM-Datei eine Spalte der Attributtabelle des

[15] Michaelson, Greg. 2020

Layers zugeordnet. Sollte einem Attribut keine Layer-Spalte zugeordnet sein, verbleibt der *„Wert"* des *„Schlüssels"* für dieses Attribut leer.

Nachdem alle gewünschten BlueM-Dateitypen einen Quell-Layer inklusive der entsprechenden Zuordnungsverzeichnisse erhalten haben, kann der Anwender einen Ziel-Pfad definieren und den Export per Klick auf den entsprechenden Button starten.

Dies triggert die Funktion *„def export_clicked"*, die überprüft für welche Dateitypen eine Zuordnung vorgenommen wurde. Für diese Dateitypen (außer TAL) wird dann nacheinander die Funktion *„def export_file"* gestartet.

In dieser Exportfunktion wird anhand des übergebenen Dateityps die entsprechende Layerselektion ausgelesen und das zu verwendende Zuordnungs-Dictionary ermittelt. Weiterhin werden über die Support-Funktion *„def get_type_info"* alle Formatierungsinformationen für den aktuellen Dateityp aus der Definitionsdatei gesammelt. Danach liefert die Funktion *„def construct_filename"* den zu verwendenden Dateinamen inklusive dem aus der Nutzereingabe ausgelesenen Dateipfads, sowie den Exportzeitpunkt.

Danach beginnt das Plugin die neue BlueM-Datei zu schreiben. Zuerst werden die passenden Titel und Tabellenüberschriften aus der Definitionsdatei in die neue Datei geschrieben, danach folgen die Werte aus dem Quell-Layer.

Jedes einzelne Feature des Quell-Layers wird wie eine Zeile der späteren BlueM-Datei behandelt. Dabei wird in der Reihenfolge der Datei-Attribute der Inhalt der zugeordneten Layer-Spalte für das aktuelle Feature ausgelesen.

Der Inhalt wird dann, samt der Definition des jeweiligen Attributes aus der Definitionsdatei, an die Prüf-Funktion *„def check_value"* übergeben.

In dieser Funktion wird anhand der mitgelieferten Attribut-Definition überprüft, ob der Inhalt aus dem Layer in die BlueM-Datei geschrieben werden kann. Falls nötig, wird der Inhalt so angepasst, dass er als Wert in BlueM verwendet werden kann; dies umfasst beispielsweise Kürzungen von Nachkommastellen oder zu langen Texten.

Sollte der Inhalt aus dem Layer nicht in einen nutzbaren Wert umgewandelt werden können, so wird er durch Leerzeichen bzw. das vom Anwender in den Einstellungen festgelegte Zeichen ersetzt.

Falls dem aktuellen BlueM-Attribut im Verzeichnis keine Layer-Spalte zugewiesen wurde, wird die entsprechende Stelle des erwarteten Wertes mit der passenden Anzahl an Leerzeichen aufgefüllt.

Alle überprüften Werte bzw. Leerzeichen des Features werden in eine Liste für die aktuelle Zeile der Datei aufgenommen. Nach Verarbeitung aller Attribute dieser Zeile, werden alle Elemente dieser Liste anhand eines Musters aus der Definitionsdatei in die BlueM-Datei geschrieben. Das genannte Muster enthält die vordefinierten Abstände der Werte und vertikale Striche, die dafür sorgen, dass sich die Zeile nahtlos in die Tabellenstruktur der BlueM-Datei einfügt.

Dies wird für alle Features des Layer wiederholt.

Sollten einzelne Werte bei der Überprüfung verändert worden sein, wurde für jede Änderung ein Vermerk erzeugt, der den betroffenen Wert und die vorgenommene Veränderung beschreibt. Diese Vermerke werden gesammelt und zusammen mit weiteren Informationen (Exportzeitpunkt, Anzahl der zugewiesenen Attribute, etc.) in einem Textblock an das Ende der Datei geschrieben.

Die einzelnen Zeilen dieses Textblocks beginnen, wie die Designelemente der restlichen Datei, mit einem Sternchen („*"), sodass BlueM nicht versucht diese einzulesen. Der Textblock ist lediglich als Information für den Anwender gedacht.

Der Anwender kann das Schreiben dieser Informationen in die Datei auch abschalten, sowie zusätzlich oder stattdessen alle Informationen aus allen Dateitypen eines Exportvorganges gesammelt in einem sogenannten Export Log ausgeben lassen (siehe Einstellungen).

Der standard Exportvorgang ist im folgenden Flowchart vereinfacht dargestellt; Entscheidungen des Anwenders sind gelb, seine Eingaben blau, Elemente des Plugins sind grün.

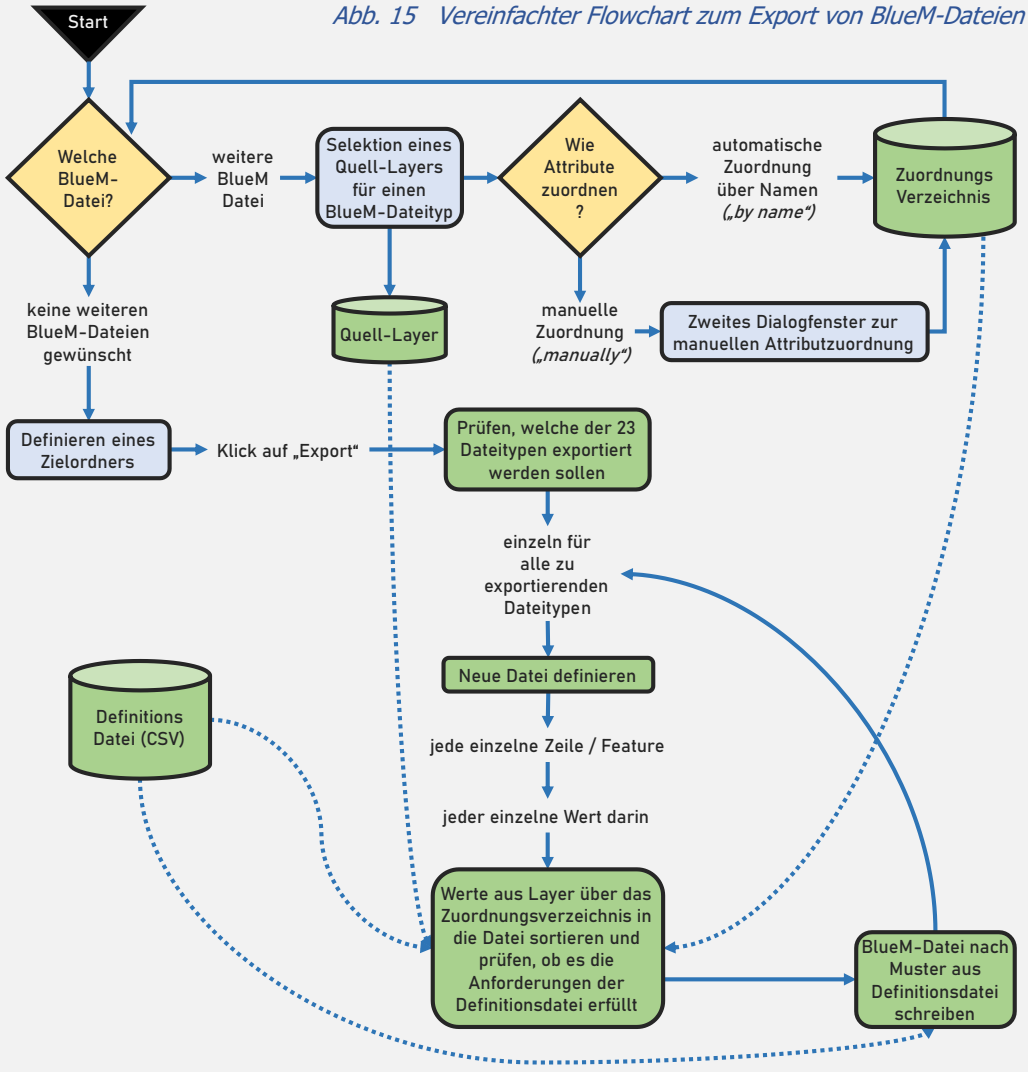


Abb. 15 Vereinfachter Flowchart zum Export von BlueM-Dateien

Während des standard Exportvorgangs wird auch auf kleinere Eigenarten der Dateitypen eingegangen.

- Für das FKT-file werden 3 verschiedene Arten von Zwischenlinien in die Tabelle eingepflegt, sodass beispielsweise einfacher zu erkennen ist, wann eine neue Funktion beginnt.
- Beim ALL-file wird lediglich das erste Feature des Layers behandelt, da mehrere Daten-Zeilen nicht in die als Liste formatierte Datei passen würden.
- Das KTR-file wird anhand der Bezeichnungen im ersten Attribut in zwei verschiedene Tabellen mit eigenen Überschriften etc. aufgespalten.

Der Unterschied zwischen der Struktur eines TAL-files und allen anderen Dateitypen ist jedoch so groß, dass zur Erzeugung eines Solchen eine völlig andere Funktion benötigt wird. Sollte ein TAL-file konstruiert werden müssen, wird daher nach dem Klick auf „Export", die Funktion *„def export_tal_file"* angesteuert.

Ein TAL-file besteht aus einer Übersichtstabelle mit einzelnen Talsperren-IDs und weiteren Parametern zu diesen Talsperren. Zusätzlich enthält die Datei 6 weitere Tabellen für jede einzelne Talsperre in der Übersichtstabelle. Eine TAL-Datei für 3 Talsperren würde daher bereits aus 19 einzelnen Tabellen bestehen.

Um dies zu realisieren, müssen die Bereiche der Attributtabelle des Layers nicht nur, wie üblich, nach Spalten den Attributen zugeordnet werden. Vielmehr müssen die Spalten zu Gruppen für die einzelnen Tabellen sortiert werden, sowie ebenfalls die Zeilen der Attributtabelle für spezifische Datei-Tabellen gebündelt werden.

Dreh- & Angelpunkt für diese Einteilung sind die sogenannten Talsperren-IDs. Da eine weitere Konstruktion der TAL-Datei ohne diese keinen Sinn macht, wird zuerst überprüft, ob der Quell-Layer Talsperren-IDs enthält.

Sollten keine Talsperren-IDs gefunden werden, wird die Funktion *„def export_tal_file_failed"* ausgelöst. Diese informiert den Anwender zum einen direkt über eine Message-Bar in QGIS über den Fehler, zum anderen wird im Zielverzeichnis eine Datei mit dem entsprechenden Hinweis generiert, sowie die Informationen an das Export Log gesendet. Dabei wird unterschieden, ob keine passende Spalte für das Attribut „Talsperren-IDs" zugeordnet wurde, oder ob die zugeordnete Spalte keine Talsperren-IDs enthält.

Sollten Talsperren-IDs (diese beginnen mit „T") im Layer gefunden worden sein, wird die Funktion *„export_tal_file_possible"* gestartet.

Diese startet, ähnlich wie die standard Exportfunktion, mit dem Sammeln von Informationen für den aktuellen Dateityp. Danach werden jedoch deutliche Unterschiede sichtbar, wenn die Funktion beginnt die Layer-Daten horizontal einzuteilen.

Dazu wird das Füllmuster aus der Definitionsdatei anhand von Zeilenumbrüchen in 7 Elemente aufgeteilt, welche wiederum die 7 verschiedenen Tabellentypen

widerspiegeln. Nun wird gezählt wie viele Attribute in den einzelnen Füllmustern Platz finden und daraus ein Bereich von Attribut-Nummern definiert, die der jeweiligen Tabelle zugewiesen werden. In Verbindung mit dem Zuordnungsverzeichnis wird dadurch bestimmt, welche Layer-Spalten zu welcher Tabelle der TAL-Datei gehören.

Im nächsten Schritt werden die Layer-Daten vertikal aufgeteilt. Dazu wird die Layer-Spalte, die laut Zuordnung die Talsperren-IDs enthält, durchgegangen und notiert, in welcher Zeile der Attributtabelle des Layers eine Talsperren-ID eingetragen ist. Aus den einzelnen Zeilenindices der Talsperren-IDs werden dann wiederum vertikale Bereiche (mit Start und Ende) definiert, die einer einzelnen Talsperre zugeordnet werden können.

Aus der ersten Zeile eines jeden vertikalen Bereichs, kann nun die Übersichtstabelle zusammengesetzt werden, wobei nur die der Übersichtstabelle zugewiesenen Attribute (horizontaler Bereich) verwendet werden.

Die erste und alle weiteren Zeilen eines bestimmten vertikalen Bereichs (entspricht einer bestimmten Talsperren-ID) werden danach verwendet, um die 6 Unter-Tabellen für diese Talsperren-ID zu füllen. Die Daten werden wieder anhand ihrer, den Spalten zugeordneten Datei-Attribute, in die 6 Gruppen eingeteilt.

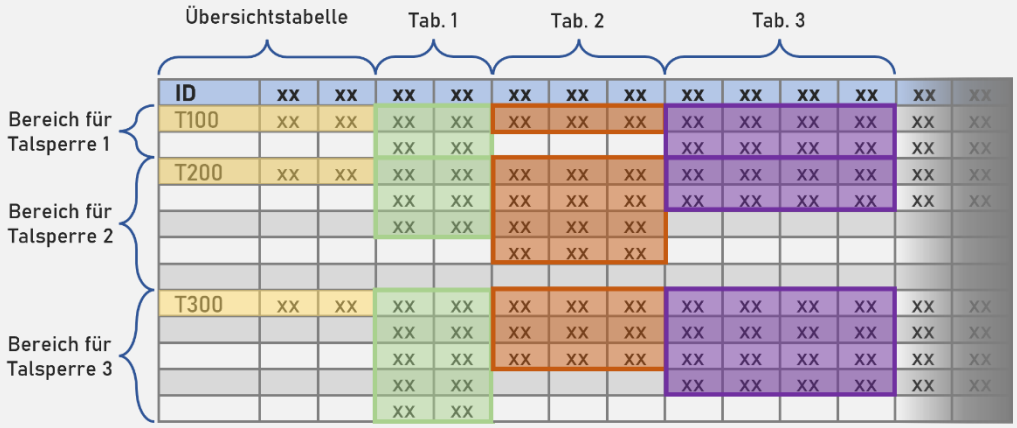


Abb. 16 Einteilung der Attributtabelle eines Layers für das TAL-file

Das obenstehende Beispiel zeigt die für die Übersichtstabelle gesammelten Daten in Gelb, sowie die einzelnen Blöcke von Zellen für 3 der 6 Untertabellen der einzelnen Talsperren in Grün, Braun und Violett. Die Darstellung ist dabei stark vereinfacht, da das TAL-file bis zu 72 Layer-Spalten verwenden kann, wobei nicht alle Spalten des Layers in der Tal-Datei genutzt werden müssen.

Die Spalten für eine bestimmte Untertabelle müssen auch nicht unbedingt nebeneinander liegen, da die Gruppierung über die den Spalten zugewiesenen Datei-Attribute erfolgt.

Die Erstellung der 6 Unter-Tabellen wird für alle Talsperren-IDs wiederholt, wobei der Platz zwischen den Daten-Feldern mit vorgefertigten Überschriftsblöcken aus der Definitionsdatei gefüllt wird. Dabei wird der erste Überschriftsblock einer Gruppe von Untertabellen für die jeweilige Talsperren-ID angepasst, damit BlueM sie korrekt identifizieren kann.

Auf diese Weise kann die eine Attributtabelle eines Layers in viele kleinere Tabellen einer BlueM-Datei umformatiert werden.

Die Überprüfung und Anpassung der einzelnen Werte, sowie die Änderungsvermerke und Exportinformationen werden dabei analog zur standard Exportfunktion behandelt.

4.3 Funktionsschemata der weiteren Tools

Im Folgenden werden die Funktionsabläufe von unterstützenden Werkzeugen (Tools) des Plugins näher erläutert, deren Anwendung in 3.3 beschrieben ist.

→ Korrektur fehlerhafter Spaltentitel im Layer

Bei der Anpassung des zweiten Dialogfensters wird automatisch geprüft, ob die Spalten-Titel der Attributtabelle des gewählten Layers von QGIS korrekt eingelesen wurden. Falls erkannt wird, dass der erste Spaltentitel des Layers „Field1" lautet, also vermutlich automatisch generiert wurde, wird für den Benutzer im zweiten Dialogfenster ein Button sichtbar.

Dieser Button mit der Aufschrift *„set field names from first row"* ruft auf Knopfdruck die Support-Funktion *„def correct_field_names"* auf.

In dieser Funktion wird der aktuell für den Dateityp des zweiten Dialogfensters selektierte Layer bestimmt und jede Spalte seiner Attributtabelle untersucht. Falls die ersten 5 Buchstaben des jeweiligen Spaltentitels „Field" lauten, wird dieser Titel durch den Wert des ersten Features in dieser Spalte ersetzt.

→ Adaption vorhandener Layer

Die Adaption eines vorhandenen Layers erfordert die Auswahl eines Layers des aktuellen QGIS-Projekts, sowie eines BlueM-Dateityps durch den Nutzer. Erst dann wird der *„Append"*-Button freigeschaltet.

Dieser startet die Funktion *„def append_layer"*, welche den gewählten Layer und den gewünschten Dateityp ausliest, mit diesen eine Nachricht über den aktuellen Vorgang in QGIS anzeigt und dann an die Funktion *„def append_layer_generic"* weitergibt.

In dieser, auch später für die einzelnen Layer des GeoPackages verwendeten, Funktion werden dann die aus der Definitionsdatei gefüllte Liste der

Attributbezeichnungen und das Dictionary der Attributdefinitionen für den gewählten Dateityp geladen.

Mit diesen Informationen wird ein neues Dictionary aufgestellt, welches festlegt, welchen Datentyp die neue Spalte für ein bestimmtes Attribut haben soll. Dabei stehen Ganzzahl, Dezimalzahl, Datum/Uhrzeit und Text zur Auswahl.

Danach wird geprüft, welche Layer-Spalten bereits zu den Attributen der BlueM-Datei passende Titel tragen; diese werden nicht verändert. Alle weiteren (noch nicht vorhandenen) Spalten werden mit dem jeweils passenden Datentyp dem Layer angefügt.

→ Dropdownmenüs in SYS-Attributtabelle

Sollte in der Funktion *"def append_layer"* erkannt werden, dass es sich bei dem gewünschten Dateityp um eine SYS-Datei handelt, wird nach der weiteren Adaption des Layers die Funktion *„def change_widget_type_sys"* aufgerufen.

Diese listet die einzelnen Spalten des SYS-files auf, welche als Dropdownmenü ausgeführt werden sollen (3 Zuläufe und 3 Abläufe), sowie die Spalte des Layers, die als Quelle für diese verwendet werden soll (Spalte *„Nr"*). Mit diesen Informationen, sowie dem gewählten Layer als Parameter, wird die allgemein gehaltene Funktion *„def change_widget_type"* für jede der 6 anzupassenden Spalten in einem Loop aufgerufen.

In dieser wird wiederum ein Setup definiert, das alle Befehle für ein QGIS-eigenes Widget enthält, das die eigentliche Operation zur Änderung der Spalte von normalen Zellen hin zu Dropdownmenüs durchführt.

→ Export eines GeoPackages als Quell-Layer

Neben der Adaption vorhandener Layer bietet das Plugin auch die Möglichkeit komplett neue Layer für alle BlueM-Dateitypen als GeoPackage in einem Zielverzeichnis zu erzeugen.

Nach Eingabe eines validen Pfades wird der Button *„Create GeoPackage in target directory"* freigeschaltet, welcher bei einem Klick die Funktion *„def create_geopackage"* ausführt.

Mit dieser Funktion werden nach und nach 23 Layer erzeugt, wobei sich die Geometrie-Art des Layers (Polygon, Multipolygon, Point oder keine Geometrie) an dessen späteren Verwendungsmöglichkeiten orientiert.

Die einzelnen Layer werden ebenfalls über die bereits bekannte Funktion *„def append_layer_generic"* an ihre zugewiesenen Dateitypen angepasst.

Da dies ein ungewöhnlich langwieriger Prozess ist (je nach System etwa 20 Sekunden), wird der Anwender darüber in der Message-Bar informiert und kann die Entstehung der 408kB großen GeoPackage-Datei im automatisch geöffneten Windowsfenster verfolgen.

→ Automatisches Generieren eines SYS-Layers

Das automatische Generieren eines SYS-Layers wird über den *„generate"*-Button gestartet, welcher die Funktion *„def generate_sys_layer"* aufruft.

Diese Funktion durchsucht alle Layer, welche für eine der 10 BlueM-Element-Dateien selektiert sind – die Layer müssen dabei nicht über einen aktiven Sortier-Button verfügen, es reicht, wenn sie in der Combobox ausgewählt sind.

In all diesen Layern werden die für die Element-ID und den Ablauf definierten Spalten ausgelesen und die Elemente in einem Dictionary mit ihren zugewiesenen Abläufen gespeichert. Es können dabei mehrere Abläufe, durch Leerzeichen, Komma oder Semikolon getrennt, für ein Element gelistet sein.

Diesem Dictionary wird, nach allen Elementen aus den Layern, automatisch ein Zielpegel (*„ZPG"*) hinzugefügt.

Danach wird für jedes einzelne Element des Dictionaries geprüft, in welchen Ablauf-Listen es genannt wird. Das entsprechende Element wird dann als Zulauf des erstgenannten Elements in einem Dictionary für Zuläufe verzeichnet.

Da die spätere BlueM-Datei nur jeweils 3 Spalten für Zu- und Abläufe vorsieht, werden für jedes Element nur diese 6 anderen Elemente zugeordnet. Sollten in den Quell-Layern mehr Zu- und Abläufe verzeichnet sein, werden diese ignoriert.

Die Daten der beiden Dictionaries werden dann in einen neuen temporären Layer mit dem Namen *"Generated_SYS_layer_temporary"* eingefügt, welcher bereits passend zur SYS-Datei formatiert wurde.

Dieser neue Layer wird ebenfalls direkt in der zur SYS-Datei gehörenden Layerselektion ausgewählt und der *„by name"*-Button aktiviert. Eine Anpassung des Layers ist jedoch weiterhin möglich; beispielsweise für das Auswählen der *„WEL"*- und *„BIL"*-Ausgabe.

Damit die Abläufe der einzelnen Elemente eingetragen werden können, wurde für die 10 Element-Dateien eine *„Output_to"*-Spalte in deren Definition hinzugefügt, sodass ein vom Plugin neu erzeugter oder angepasster Layer automatisch diese Spalte enthält. Da diese spezielle Spalte nicht Teil der späteren BlueM-Dateien sein soll, wird sie beim Exportvorgang ignoriert.

4.4 Definitionsdatei

Die Definitionsdatei ist das Herzstück des Plugins und ermöglicht das Abändern einer Vielzahl von Parametern für die BlueM-Dateien, ohne den eigentlichen Python Code ändern zu müssen.

Die Datei befindet sich als „inputfiles_overview.csv" im Plugin-Verzeichnis *„…QGIS3\profiles\[QGIS-Profil]\python\plugins\create_bluem_input_files"* und

sollte zur Bearbeitung vorzugsweise in Excel geöffnet werden, da die hier vorhandene Möglichkeit zum Fixieren von Zeilen/Spalten und die anpassbaren Spaltenbreiten die enthaltene Tabelle deutlich übersichtlicher machen.

Für die 23 BlueM-Dateitypen sind folgende Parameter (in Spalten) hinterlegt:

- *„name_short"* Kurzbezeichnung (und Extension) des Dateityps
- *„name_long"* Längere, englische Bezeichnung des Dateityps
- *„comment_for_dlg2"* Kommentar für das zweite Dialogfenster
- *„example"* Beispiel für eine Datei dieses Typs
- *„headlines"* Titel und Überschriftenblock
- *„add_lines_[1 bis 7]"* Zusätzliche Textblöcke
- *„last_line"* Letzter Textblock der Tabelle
- *„pattern"* Füllmuster für die Zeilen (bei TAL 7 Stück)
- *„attr_count"* Anzahl der Attribute des Dateityps
- *„attr_[1 bis 80]"* Bezeichnung des jeweiligen Attributes der Datei
- *„attr_[1 bis 80]_type"* Definition des vorgenannten Attributes

Die Reihenfolge der Spalten ist dabei irrelevant und es können auch weitere Spalten (beispielsweise für Kommentare und Ähnliches) eingefügt werden, da das Plugin beim Start die genannten Parameter inklusive ihrer Spalten-Indices ermittelt und das Auslesen der Daten dann über diese Indices steuert.

Die Bezeichnungen der einzelnen Attribute wurden so gewählt, dass die dazugehöre Spalte der Beispieldatei möglichst einfach identifiziert werden kann. Dabei wurde darauf geachtet, die Bezeichnungen der 428 einzelnen Attribute möglichst kurz und dennoch einzigartig zu halten. Sowie dabei keine Leerzeichen und sprachspezifische Umlaute zu verwenden.

Aktuell sind die Datei und das Plugin auf maximal 80 Attribute pro Dateityp beschränkt, dies ist aber kein Zwangspunkt des Codes, vielmehr sind im zweiten Dialogfenster nur 80 individuelle Felder für Attribute vorhanden. Um die maximale Attributanzahl pro Dateityp zu erhöhen, müssten daher im Qt Designer mehr Felder in den Scroll-Bereich eingefügt werden. Dies ist einfach möglich, wurde bis jetzt jedoch nicht für nötig erachtet.

Die Definitionen der einzelnen Attribute sind dabei nach einem einfachen System aufgebaut:

- Die Anzahl der Zeichen der Definition beschreibt die maximale Länge des Wertes in der späteren BlueM-Datei.
- Der Buchstabe „s" definiert den Datentyp des Attributes als *„String"* (Text).
- Ein „i" bedeutet, dass der Wert ein *„Integer"* (Ganzzahl) sein soll.
- Ein „f" markiert einen *„Float"* (Dezimalzahl).
- Bei einem *„Y"* ist ein Entscheidungswert gefordert, der in der späteren BlueM-Datei als *„J"* oder *„N"* erscheint.

- Alle weiteren Definitionen sind Varianten von Datum & Uhrzeit und erscheinen in diesen Formaten: „TT.MM", „hh:mm", „TT.MM hh:mm" und „TT.MM.JJJJ hh:mm"
- Ein *String* wird später linksbündig formatiert, *Integer* und *Floats* hingegen rechtsbündig. Entscheidungswerte und Datum/Uhrzeit passen exakt in ihre Felder.

Der Inhalt der Definitionsdatei kann im Anhang eingesehen werden.

4.5 Fehlerresilienz & Informationspolitik

Die einzelnen Funktionalitäten des Plugins sind so konzipiert, dass möglichst wenige Fehler produziert werden. Dies geschieht zum einen durch ein Leiten des Anwenders, sodass er keine Eingaben vornehmen kann, die zu einem Fehler führen würden. Zum anderen wird versucht, Fehler in den Eingangsdaten automatisch zu beheben. Gleichzeitig wird angestrebt, den Anwender über diese Vorgänge informiert zu halten.

Beispiele hierfür sind:

➔ Sperrung von Elementen der Benutzeroberfläche

Die Buttons zur Auswahl der Zuordnungsmethode (*„by name*" & *„manually*") werden erst zum „Anklicken" freigeschaltet, wenn ein Layer für diesen Dateityp selektiert wurde. Umgekehrt wird die entsprechende Layerselektion fixiert, solange ein Zuordnungsbutton aktiviert ist.
Auch werden die Buttons zum Export der BlueM-Dateien, der Standard-Definition, des Benutzerhandbuchs und des GeoPackages nur freigegeben, wenn der Anwender einen validen Pfad zu einem Verzeichnis eingegeben hat. Über dies wird der Anwender durch die Label auf den betroffenen Buttons, sowie im Text vor dem Pfad-Eingabe-Feld informiert.
Es wird dabei überprüft, ob ein Pfad eingegeben wurde, dieser Leerzeichen enthält, oder das Verzeichnis nicht existiert – die Informationen werden jeweils entsprechend aktualisiert.

➔ Anpassung von Werten

Eine wichtige Funktion des Plugins ist das Beheben kleinerer Fehler in den Eingangsdaten, damit die erzeugten Dateien korrekt in BlueM eingelesen werden können.
Dies wird beispielsweise umgesetzt, indem in Zahlenwerten als Dezimaltrennzeichen verwendete Kommata in Punkte umgewandelt werden,

oder fehlerhaft eingelesene Texte („*Null*", oder „*None*") durch Leerzeichen ersetzt werden.
Auch wird darauf geachtet, ob die Werte des Layers aufgrund ihrer Zeichenzahl in die Spalten der BlueM-Datei passen werden. Hier wird nach Möglichkeit sinnvoll gekürzt.
Ebenfalls ist es nicht möglich den Exportvorgang der BlueM-Dateien durch unpassende Werte abstürzen zu lassen, da dieser „*ValueError*" automatisch abgefangen und registriert, der weitere Vorgang aber nicht gestört wird.

➔ Export Information

All diese Anpassungen der Werte werden automatisch protokolliert und samt allgemeiner Exportinformationen (über verwendete Layer, fehlende Zuordnungen, etc.) den BlueM-Dateien beigefügt oder in einem Export-Log für alle exportierten Dateien gesammelt.

```
* This EZG-file was created by the 'QGIS BlueM Interface Plugin'
*   with data from the 'EZG'-layer
*   on the 2022/02/03 at 17:57.
*
* Of the 40 file attributes, 38 found a matching field name in the layer.
* (including the not printed 'Output_to')
* These attributes found no match: ['attr_01', 'attr_13']
*
* 23 values had to be changed or were not suitable at all:
*
*  -> For attribute number 05 of feature 001:  !!! integer part of float was to long to fit
*  -> For attribute number 16 of feature 001:      string was shortened
*  -> For attribute number 17 of feature 001:      float rounded to fit
*  -> For attribute number 19 of feature 001:  !!! integer part of float was to long to fit
*  -> For attribute number 22 of feature 001:      float rounded to fit
*  -> For attribute number 40 of feature 001:      string was shortened
*  -> For attribute number 19 of feature 002:      float was changed to integer
*  -> For attribute number 28 of feature 002:      float rounded to fit
*  -> For attribute number 32 of feature 002:      float rounded to fit
*  -> For attribute number 34 of feature 002:      float rounded to fit
*  -> For attribute number 40 of feature 002:      string was shortened
*  -> For attribute number 06 of feature 003:      float rounded to fit
*  -> For attribute number 33 of feature 003:  !!! Y/N value was not recognized
*  -> For attribute number 34 of feature 003:      float rounded to fit
*  -> For attribute number 40 of feature 003:      string was shortened
*  -> For attribute number 09 of feature 004:      string was shortened
*  -> For attribute number 12 of feature 004:      string was shortened
*  -> For attribute number 14 of feature 004:      string was shortened
*  -> For attribute number 15 of feature 004:      string was shortened
*  -> For attribute number 18 of feature 004:      string was shortened
*  -> For attribute number 23 of feature 004:  !!! Y/N value was not recognized
*  -> For attribute number 29 of feature 004:  !!! Y/N value was not recognized
*  -> For attribute number 33 of feature 004:  !!! Y/N value was not recognized
```

Abb. 17 Beispiel der Export-Information einer BlueM-Datei

➔ Message-Bar

Neben den Einträgen in den Dateien und den Button-Beschriftungen wird die QGIS-eigene Message-Bar verwendet, um den Anwender über laufende Prozesse zu informieren. Er erhält hier zum Beispiel Hinweise, wenn er BlueM-Dateien exportieren will, ohne entsprechende Zuordnungen vorgenommen zu haben. Aber auch wenn ein TAL-file aufgrund fehlender Talsperren-IDs nicht erzeugt werden kann oder keine Layer für das automatische Generieren eines SYS-Layers vorliegen.
Die Hinweise sind dabei nicht immer nur auf Probleme bezogen, so wird der Anwender auch über den gestarteten Export von BlueM-Dateien oder die Anpassung eines Layers detailliert informiert.

➔ **Sonstige Hilfen**

Ebenfalls erwähnenswert ist die Vorsortierung der Layer nach Typ, sodass keine Rasterlayer selektiert werden können.
Oder auch das automatische Überspringen einer Datenzeile des Layers, wenn sie mit den Spaltentiteln übereinstimmt.

4.6 Systemvoraussetzungen & Installation

Das Plugin wurde unter **QGIS 3.16.11 (LTR)** entwickelt. Da für einige Funktionalitäten des Plugins die ab dieser QGIS LTR enthaltene Qt Version 5.12 benötigt wird, ist diese QGIS Version (oder höher) zwingend erforderlich.

Getestet wurde das Plugin unter Windows 10, theoretisch sollten auch Windows 11, 8 und 8.1 möglich sein. Unter Windows 7 funktioniert das Plugin nicht, weil dort keine Unterstützung für Python 3.9 vorliegt und dementsprechend auch die benötigte QGIS-Version nicht unterstützt werden würde.

Da das Plugin aktuell noch Teil einer Abschlussarbeit ist, wurde es noch nicht veröffentlicht und ist daher nicht über das übliche QGIS Plugin Repository zu beziehen. Es muss daher manuell installiert werden.

Hierzu klickt man in QGIS den Reiter *„Einstellungen"* (*„Settings"*), navigiert zu den *„Benutzerprofilen"* (*„User Profiles"*) und öffnet den Ordner des aktuellen Profils (*„Open Active Profile Folder"*).

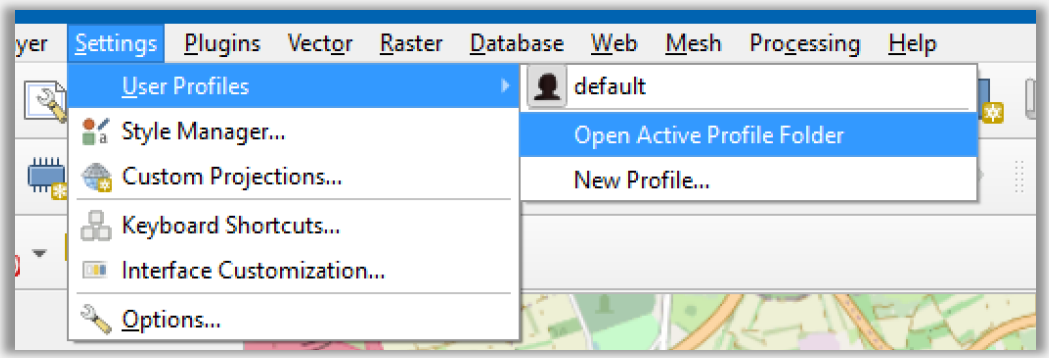


Abb. 18 Navigation zu aktivem Profil Ordner

In diesem Ordner befindet sich der Ordner *„python"* und darin der Ordner *„plugins"*. Sollten die Ordner nicht existieren, liegt es daran, dass noch keine Plugins für dieses Benutzerprofil installiert wurden. Die beiden Ordner müssen in diesem Fall manuell erstellt werden.

Nun werden die Plugin-Daten als 1 Ordner *„create_plugin_input_files"* in den bestehenden Ordner *„plugins"* kopiert. Danach muss QGIS neu gestartet werden.

Nach dem Neustart navigiert man über den Reiter *„Erweiterungen"* (*„Plugins"*) zur Pluginverwaltung (*„Manage and Install Plugins…"*). Hier wird nun manuell nach dem Plugin gesucht (*„bluem"* im Suchfeld für *„All / Alle"* sollte ausreichen) und der entsprechende Haken vor *„Create BlueM Input Files"* gesetzt.

Nachdem der Dialog geschlossen wurde, erscheint das Plugin in der Toolbar (als BlueM-Icon) und kann verwendet werden.

4.7 Dateiübersicht

Das Plugin benötigt etwa 1 Megabyte an Speicherplatz und besteht aus 13 einzelnen Dateien, die im Folgenden kurz beschrieben sind:

→ ***__init__.py***
Python-Datei zur Initialisierung des Plugins

→ ***_USER_MANUAL_qgis_bluem_interface_plugin.pdf***
Kurzes Benutzerhandbuch (englisch) zum Plugin. Damit der Anwender nicht in den Plugin-Daten danach suchen muss, wird es auf Knopfdruck in ein gewünschtes Verzeichnis kopiert.

→ ***create_bluem_input_files.py***
Durch die *init*-Datei angesteuerte Hauptdatei zum Aufrufen des Plugins. Diese standardisierte Datei wurde durch den Plugin-Builder erstellt, weshalb der im Rahmen dieser Thesis erarbeitete Code in die Datei *plugin_functions.py* ausgelagert wurde.

→ ***create_bluem_input_files_dialog.py***
Python-Datei zur Verbindung des Python Codes mit den beiden Userinterface-Definitionsdateien.

→ ***create_bluem_input_files_dialog_base.ui***
Diese mit Qt erzeugte UI-Datei enthält die in XML kodierte grafische Benutzeroberfläche für das erste Dialogfenster des Plugins.

→ ***create_bluem_input_files_dialog2_base.ui***
Diese mit Qt erzeugte UI-Datei enthält die in XML kodierte grafische Benutzeroberfläche für das zweite Dialogfenster des Plugins.

→ ***icon.png***
Diese Bilddatei enthält das BlueM-Logo, welches in QGIS als Icon für das Plugin verwendet wird.

→ ***inputfiles_overview.csv***
Dies ist die Definitionsdatei, welche für alle 23 BlueM-Inputfiles Informationen über deren Attribute, Textblöcke und Tabellenmuster enthält. Hier können auch durch Anwender, die den eigentlichen Programmcode nicht ändern wollen, signifikante Anpassungen der späteren BlueM-Dateien vorgenommen werden.

→ ***inputfiles_overview.xlsx***
Excel-Version der CSV-Definitionsdatei. Die CSV-Datei könnte auch direkt in Excel geöffnet werden, jedoch speichert eine CSV keine Spaltenbreiten oder „Fixierungen" von Zeilen & Spalten. Es ist daher ratsam die Bearbeitung der Definitionsdatei im übersichtlich gestalteten Excel-Format durchzuführen und die CSV-Datei danach zu überschreiben.

→ ***metadata.txt***
Diese Textdatei enthält die relevanten Metadaten des Plugins.

→ ***plugin_functions.py***
In dieser Python-Datei sind die für das Plugin im Rahmen dieser Abschlussarbeit geschriebenen Funktionen gespeichert. Von hier werden sie in die Hauptdatei (create_bluem_input_files.py) importiert.

→ ***resources.py***
Automatisch erzeugte Python-Version der gleichnamigen QRC-Datei.

→ ***resources.qrc***
Diese QRC-Datei ist ein Verzeichnis für die Pfade aller weiteren, für das Plugin benötigten Elemente (außer den verlinkten Code-Dateien). Aktuell ist hier lediglich das Icon verzeichnet.

Zusätzlich zu diesen 13 Dateien gibt es 2 Ordner, die nicht integraler Bestandteil des Plugins sind, aber unter Umständen automatisch erzeugt werden:

Der Ordner ***„.idea"*** wird erzeugt, wenn der Python Code in einer IDE (hier: PyCharm) verwendet wird und enthält die dort gesetzten Bookmarks, etc.

Im Ordner ***„__pycache__"*** befinden sich kompilierte Versionen der 5 im Plugin verwendeten Python-Dateien. Diese sind für den Computer schneller einzulesen als die ursprünglichen Dateien. Der Ordner kann gelöscht bzw. nicht mit übergeben werden, da er bei jedem Start des Plugins neu erzeugt werden würde.

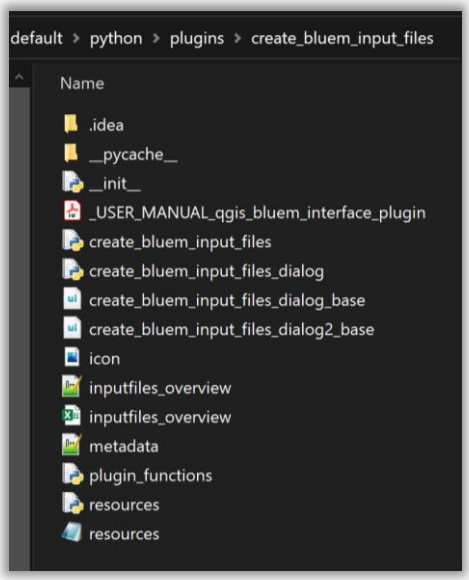


Abb. 19 Die Dateien des Plugins

4.8 Bekannte Fehler (Known Bugs)

Aktuell sind folgende geringfügige Fehler („Bugs") im Plugin bekannt:

→ Tooltip des Pfad-Eingabefeldes nicht auf Englisch

Die Benutzeroberfläche, Tooltips und alle Ausgaben des Plugins sind grundsätzlich in Englischer Sprache verfasst. Das speziell für QGIS entwickelte Qt-Widget *„QgsFileWidget"*, welches im Plugin den Pfad zum Exportverzeichnis aufnimmt, lässt in der aktuellen Version jedoch kein Setzen eines selbstdefinierten Tooltips zu.

Stattdessen wird ein standardisierter Tooltip in der jeweils von QGIS genutzten Sprache angezeigt. Ist der Tooltip des FileWidgets also nicht, wie alles andere, auf Englisch, so liegt das daran, dass das genutzte QGIS aktuell nicht Englisch als Sprache verwendet.

In einer zukünftigen Version des Widgets bzw. von Qt Designer kann dieses Problem eventuell behoben werden.

→ Probleme bei der Windows-Skalierung

Es ist beobachtet worden, dass das Plugin unter einer erhöhten Windowsskalierung verzerrt dargestellt werden kann. In diesem Fall werden die Texte hochskaliert, die entsprechenden Textfelder jedoch nicht. Dies führt dazu, dass die Texte zu groß für ihre Textfelder werden und dementsprechend „abgeschnitten" erscheinen.

Dieses Problem tritt jedoch nicht bei allen Anwendern auf. Aktuell wird vermutet, dass es etwas mit der verwendeten Version von Windows 10 zu tun hat. Unter Version 21H1 erscheint das Problem nicht, unter 21H2 ist es jedoch beobachtet worden.

Der Grund konnte nicht abschließend geklärt werden. Es wird vermutet, dass es an Inkompatibilitäten zwischen Windows und Qt liegt, die vermutlich in zukünftigen Versionen gelöst werden.

Aktuell gibt es 2 Lösungsmöglichkeiten:

- Setzen der Windowsskalierung auf 100%, sodass es zu keinen Verzerrungen kommt.
- Die räumliche Fixierung der Dialogfenster des Plugins wurde zu Lösung dieses Problems aufgehoben. Die Fenster können daher mit der Maus größer „gezogen" werden, wodurch sich auch die Textfelder vergrößern und der enthaltene Text lesbarer wird.

→ Dopplung der ersten Zeile nach Anpassung der Spaltentitel

Das Plugin bietet die Möglichkeit die Spaltentitel der Attributtabelle eines Layers mit den Einträgen der ersten Zeile zu überschreiben, falls die Spaltentitel einer CSV- oder Excel-Datei von QGIS nicht automatisch erkannt wurden.

Die erste Zeile des Layers besteht jedoch weiterhin unverändert fort. Dies führt dazu, dass in der Quelldatei des Layers nach der Anpassung die erste Zeile doppelt auftaucht.

Es wäre technisch möglich diese erste Zeile automatisch zu löschen, aus Prinzip wird jedoch darauf verzichtet, Löschbefehle auf vom Benutzer eingebrachte Daten anzuwenden.

Stattdessen wurde in die Exportfunktionen ein Algorithmus eingebaut, welcher erkennt, wenn eine Zeile eines Quell-Layers identisch mit den jeweiligen Spaltentiteln ist. Diese Zeile wird dann nicht für die BlueM-Datei verwendet.

→ Fehler durch Löschen eines Layers während der Exportauswahl

Sobald für einen Dateityp ein Quell-Layer selektiert und entweder der *„by name*" oder der *„manually*"-Button aktiviert wurde, wird die Layerauswahl gesperrt, sodass der Quell-Layer nicht mehr gewechselt werden kann. Dies geht nur, wenn der aktivierte Button zuerst deaktiviert wird.

Wenn jedoch zwischen dem Aktivieren eines Sortier-Buttons und dem abschließenden Klick auf den *„Export*"-Button der für den entsprechenden Dateityp gewählte Layer vom Anwender aus dem Projekt gelöscht wird, kommt es zu einem Fehler.

Durch das Löschen des in der Layerauswahl selektierten Layers rutscht die Auswahl (bei gleichbleibendem Index) automatisch auf den nächsten Layer der Liste. Das Plugin versucht dann während des Exports die aus dem ursprünglichen Layer gespeicherten & zugewiesenen Spaltentitel in diesem neuen Layer zu finden, was nur funktionieren würde, wenn der neue Layer identische Spaltentitel hätte. Es wird daher ein Fehler ausgeworfen.

Theoretisch kann diese unglückliche Verkettung von Umständen im Code abgefangen werden, es stellt sich jedoch die Frage nach der Verhältnismäßigkeit.

4.9 Bei Änderungen der BlueM-Dateitypen beachten

➔ Änderungen der Attributnamen & Definitionen

Die vorhandenen Attributnamen und entsprechenden Definitionen können in der Definitionsdatei ohne Umstände angepasst werden. Auch bei kleineren Längenänderungen muss das Füllmuster (*„pattern"*) angepasst werden.

➔ Änderung der Dateistruktur

Eine Änderung der Struktur einer vorhandenen Datei erfordert das Anpassen des Füllmusters (*„pattern"*), sowie der Textblöcke in der Definitionsdatei.

Aktuell können bis zu 80 Datei-Attribute definiert werden; diese Zahl hängt jedoch nur von der Benutzeroberfläche ab. Es müsste dort lediglich der unbegrenzte Platz des Scroll-Bereichs mit mehr Attribut-Feldern gefüllt werden. Im Code muss dann lediglich an einer Stelle in der *„prepare_plugin"*-Funktion die Variable *„max_number_attr_file"* entsprechend erhöht werden.

➔ Hinzufügen neuer Dateitypen

Das Hinzufügen eines völlig neuen Dateityps kann in der Definitionsdatei durch Ausfüllen einer neuen Zeile erfolgen. Das Plugin wird automatisch alle relevanten Parameter einlesen und zuweisen.

Der Zwangspunkt ist auch hier wieder die Benutzeroberfläche: Alle Layerselektionen und Zuordnungs-Buttons haben in ihrer Elementkennung die Bezeichnung des dazugehörigen Dateityps. Dies war nötig, um Prozesse für alle 23 Dateitypen zu automatisieren.

Für einen neuen Dateityp müssten die entsprechenden Elemente der Benutzeroberfläche hinzugefügt oder die Elemente eines wegfallenden Dateityps umbenannt werden.

4.10 Konkrete Verbesserungsvorschläge

➔ Zweites Dialogfenster erneut öffnen

Nach der Aktivierung aus dem zweiten Dialogfenster deaktiviert ein zweiter Klick auf den *„manually"*-Button diesen lediglich, sodass der entsprechende Dateityp nicht exportiert werden würde.

Alternativ wäre es sinnvoll, dass ein zweiter Klick auf den *„manually"*-Button das zweite Dialogfenster wieder öffnet und dabei die vorher getätigten Zuordnungen wieder darstellt. Diese könnten dann angepasst werden. Ob der

„manually"-Button nach dem zweiten Klick deaktiviert wird oder aktiviert bleibt, hängt dann von den *„OK"* bzw. *„Cancel"*-Buttons des zweiten Dialogfensters ab.

➔ Attributzuordnungen vor Export überprüfen

Das Plugin prüft aktuell die im Quell-Layer vorhandenen Werte erst während des Exportvorgangs.

Es wäre für den Anwender hilfreich, wenn er bei der manuellen Zuordnung eines Attributes im zweiten Dialogfenster eine Warnung im jeweiligen Attribut-Feld angezeigt bekommen würde, falls er eine unpassende Zuordnung plant.

Beispielsweise könnte drauf hingewiesen werden, dass in der gewählten Layer-Spalte Texte vorhanden sind, die nicht zum aktuellen Datei-Attribut passen, da es sich bei diesem um eine Zahl handeln soll.

➔ Beispiel in Dateivorschau umwandeln

Das Beispiel für den jeweiligen Dateityp im zweiten Dialogfenster zeigt aktuell eine vordefinierte Datei ohne Verbindung zu aktuellen Daten.

Es wäre möglich im zweiten Fenster einen Button einzubauen, der die üblichen Funktionen zum Generieren der BlueM-Datei startet, jedoch ohne diese zu exportieren. Stattdessen könnte der Inhalt des vorhandenen Beispiel-Textfeldes durch die Daten ersetzt werden, die einen Export mit den bisher erfolgten Zuordnungen simulieren.

➔ Manuelle Selektion der *„BlueM_ID"* und *„Output_to"*-Spalten

Die automatische Generierung eines Layers für die SYS-Daten prüft aktuell auf die Spalten *„BlueM_ID"* und *„Output_to"* in allen selektierten Layern für Element-Dateien.

Zusätzlich dazu könnte der Algorithmus prüfen, ob für den selektierten Layer bereits eine manuelle Zuordnung vorgenommen wurde. In diesem Fall würden die jeweiligen Spalten durchsucht, welche den Datei-Attributen *„BlueM_ID"* und *„Output_to" zugeordnet sind.*

➔ Kodierung der Attributdefinitionen

Im Moment erfolgt die Kodierung der Attributdefinition durch eine Zeichenfolge, deren Länge der des Attributes in der späteren Datei entspricht (Beispiel: *„ssss"* für einen Text-String der maximal 4 Zeichen lang sein kann).

Dies ist intuitiv und leicht verständlich, jedoch in der Dokumentation schwierig. So gibt es beispielsweise Text-Strings mit mehr als 20 Zeichen.

Alternativ könnte man in die Zelle der Attributdefinition ein *„s, 20"* schreiben, um einen Text-String mit maximal 20 Zeichen zu kodieren. Dies wäre ebenfalls

intuitiv nachvollziehbar und durch weitere (durch Komma getrennte) Eingaben wie beispielsweise Textausrichtung oder Relevanz/Unverzichtbarkeit des Attributes leicht zu ergänzen.

➔ Einstellungen für nächste Session speichern

Die aktuell im Plugin getätigten Einstellungen (beispielsweise für automatisches Schließen des Plugins nach Export oder Erzeugen eines Export-Logs) werden bisher bei jedem Neustart des Plugins auf den vordefinierten Standard zurückgesetzt.

Mit einer *„QgsSettings*"-Methode wäre es möglich solche Eingaben im QGIS-Projekt für die nächste Session zu speichern.

➔ Vorhandene BlueM-Datei zu Layer umwandeln

Zur Abrundung des Behandlungsportfolios wäre eine Rückabwicklung bereits vorhandener BlueM-Dateien in einen Layer denkbar. Der Anwender müsste lediglich in einem Auswahlfeld eine Datei auswählen. Das Plugin würde aus der Endung den Dateityp ermitteln, die entsprechende Formatierung aus der Definitionsdatei entnehmen und damit die Daten aus der BlueM-Datei dekodieren um damit eine reguläre Attributtabelle zu füllen.

Die Attributtabelle des erzeugten Layers würde dann eine Bearbeitung des BlueM-Datensatzes deutlich vereinfachen und gleichzeitig eine Grundlage für weitere Berechnungen bzw. GIS-Anwendungen bieten.

➔ Änderung der Elementbezeichnungen der Benutzeroberfläche

Aktuell sind die Elemente der Benutzeroberfläche des ersten Dialogfensters anhand ihrer Namen einem BlueM-Dateityp zugeordnet. So lautet die Bezeichnung der Combobox für die Layerselektion des FKA-files „cb_fka_layerselection".

Stattdessen könnte jedem Dateityp in der Definitionsdatei eine numerische ID zugeordnet werden, mit der auch auf die entsprechenden Elemente zugegriffen werden kann.

Dies würde eine spätere Änderung der BlueM-Dateitypen-Kurzbezeichnung deutlich vereinfachen.

5. Detaillierte Beschreibung des Python Codes

Die folgenden Seiten enthalten detaillierte Erklärungen zu den im Rahmen dieser Abschlussarbeit mit Python programmierten Elementen des Plugins. Diese sind in einzelne Funktionen (mit dem Präfix *„def"*) aufgeteilt, welche im Verlauf der Nutzung im Code aufgerufen werden. Es wird empfohlen während der Lektüre der Beschreibungen die einzelnen Funktionen in PyCharm oder einer anderen IDE geöffnet zu haben, um das Verständnis mithilfe des eigentlichen Codes und den darin enthaltenen Kommentaren zu erhöhen.

Der behandelte Code befindet sich (mit Ausnahme der beiden Elemente in Kapitel 5.8) in der Datei *„plugin_functions.py"*.

5.1 Vorbereitung & Allgemeine Funktionen

Imports

Bevor die einzelnen Funktionen in Python aufgerufen werden können, müssen zuerst notwendige Module importiert werden, die zusätzliche Funktionalitäten bereitstellen. Diese & ihre Anwendungsbereiche sind:

- csv Behandlung von CSV-Dateien
- numpy Arrays
- os Pfadverifikation
- math Aufrunden
- shutil Kopieren von Dateien
- datetime Datum und Uhrzeit

- Elemente aus PyQt für die Interaktion zwischen Python und Qt

- Elemente aus QGIS für die Interaktion zwischen dem Plugin und QGIS (diese werden in der IDE eventuell als nicht bekannt gekennzeichnet, aber wenn das Plugin in QGIS gestartet wird, werden die Elemente gefunden)

- Die beiden Python-Dateien, die die Benutzeroberfläche definieren bzw. zu den Qt-UI-Dateien verweisen

Globale Variablen

Damit Variablen zwischen einzelnen Funktionen des Python-Codes ausgetauscht werden können, müssen diese außerhalb der Funktionen als globale Variable definiert sein. Dies wird hier für einige String-Parameter, Listen und Dictionaries getan.

Alternativ können globale Variablen auch innerhalb von Funktionen mit dem *globals()*-Befehl definiert werden – was beispielsweise für Dateityp-spezifische Listen und Dictionaries getan wurde.

```
def prepare_plugin(add_self, first_start):
```

Diese Funktion wird automatisch beim Start des Plugins aufgerufen und steuert alle weiteren Funktionen um Daten aus der Definitionsdatei (CSV) einzulesen und das erste Dialogfenster vorzubereiten.

Im ersten Abschnitt wird die aus der standardisierten *„create_bluem_input_files.py"*-Datei übernommene *„self"*-Variable als globale Variable umgesetzt, sodass sie für spätere Funktionen nicht mehr weitergegeben werden muss. Auch werden die beiden importierten Dialogfensterdefinitionen in kürzere Variablen umbenannt und das erste Dialogfenster mit dem *„show()"*-Befehl aufgerufen.

Für dieses und das zweite Dialogfenster wird direkt die aktuelle Versionsnummer des Plugins inklusive Veröffentlichungsdatum definiert und die entsprechende Anzeige aktualisiert.

Im folgenden Abschnitt „EXECUTION" wird die Datei *„inputfiles_overview.csv"* aus dem Pluginverzeichnis in ein Numpy-Array eingelesen und die einzelnen Datenspalten (für Namen, Attributanzahl, Füllungsmuster, Attributdefinitionen, etc.) indiziert. Sowie aus den einzelnen Zeilen des Arrays eine Liste von BlueM-Dateitypen erstellt.

Danach wird für die maximale Anzahl an Dateityp-Attributen (aktuell durch die GUI auf 80 limitiert) ein Dictionary erstellt, welche den jeweiligen Index im Array für die jeweilige Attributnummer enthält.

Im nächsten Schritt wird für jeden Dateityp eine globale Liste mit seinen jeweiligen Attributbezeichnungen angelegt, gefolgt von einem globalen Dictionary, das die Array-Indizes der jeweiligen Attributdefinitionen verzeichnet. Aus diesen Beiden wird dann aus dem Array für jeden Dateityp ein eigenes, globales Dictionary erzeugt, welches die jeweiligen Attributdefinitionen enthält.

Als Letztes in diesem Abschnitt wird ein Validator definiert, der die Nutzereingaben im Textfeld *„project_name"* überprüft und dort nur standard Groß- und Kleinbuchstaben, Zahlen und Unterstriche zulässt.

Im folgenden Abschnitt „LAYER ADAPTION" werden die Layer für die Layer-Combobox gefiltert (nur Vektor oder NoGeometry, keine Raster), sowie die Combobox für die Auswahl des BlueM-Dateityps mit einer leeren Auswahl und der Liste der Dateitypen gefüllt.

Der Abschnitt „GENERAL HOUSEKEEPING" definiert eine (anfangs leere) globale Liste der Dateitypen, die exportiert werden sollen, sowie eine Liste von validen Zeichen zur späteren Verwendung.

Ebenfalls findet sich hier eine Liste von Dateitypen, deren Export nicht funktioniert. Diese ist aktuell leer. Für Dateitypen in dieser Liste würde im Folgenden die Layerselektion blockiert und ein Hinweis in der GUI angezeigt.

Weiterhin wird eine Liste für Dateitypen definiert, deren Exportvorgang nicht dem Standard entspricht. Dies ist aktuell nur die TAL-Datei, da alle anderen

dateityp-spezifischen Vorgänge in der normalen Export-Funktion behandelt werden können.

Aus der Liste der nicht-standard Dateitypen wird dann eine Liste der standard Dateitypen entwickelt und ebenfalls die Prüfung des vom Nutzer definierten Ersatzzeichens für unpassende Zellenwerte ausgelöst.

Im letzten Abschnitt („GUI CONNECTIONS") wird die „first_start"-Variabel auf „False" gesetzt, damit die ebenfalls hier aufgerufenen GUI-Funktionen nicht bei jedem Klick auf das Plugin-Icon ausgeführt werden.

```
def open_second_window(filetype):
```

In dieser Funktion wird das zweite Dialogfenster, welches sich mit der manuellen Zuweisung der Layer-Spalten zu den Dateityp-Attributen beschäftigt, definiert und dateitypspezifisch vorbereitet.

Zuerst wird eine Liste von Vokalen definiert und geprüft, ob der Dateityp, für den gerade das zweite Dialogfenster vorbereitet wird, mit einem solchen beginnt. Falls der Dateityp mit einem Vokal beginnt, wird in einer entsprechenden Variable (*„a_an"*) der englische unbestimmte Artikel *„an"* für die Verwendung in den kommenden Textfeldern gesetzt. Sollte der Dateityp nicht mit einem Vokal beginnen, wird ein *„a"* eingesetzt. Zwar sind die Grammatik-Regeln zur Verwendung von *„an"* bzw. *„a"* komplizierter als hier abgebildet, doch für die Abkürzungen der Dateitypen ist die hier vorgenommene Unterscheidung ausreichend, um die Texte grammatikalisch korrekt zu halten.

Als Nächstes wird mit einer Support-Funktion der Zeilenindex im Array des mitgelieferten Dateityps bestimmt, mit diesem das passende Dateibeispiel aus dem Array kopiert und in das Beispielfeld des Dialogs eingefügt, sowie ein grammatikalisch korrekter (a/an) Tooltip gesetzt. Weiterhin werden der für diesen Dateityp selektierte Layer und sein Name, die lange Dateityp-Bezeichnung, der Kommentar und die Anzahl der Attribute des Dateityps aus dem Array ausgelesen.

Danach wird eine Liste der Attribute (Spalten) des Layers generiert und ihre Anzahl bestimmt.

Als Nächstes wird eine Liste aller Zahlen zwischen 1 und der Anzahl der für den Dateityp möglichen Attribute erstellt, wobei einstellige Zahlen eine führende Null bekommen.

Im darauffolgenden Abschnitt „ATTRIBUTE FRAMES" geht es um die einzelnen Felder für die manuelle Attributzuordnung im Scroll-Bereich. Diese werden zuerst in einem Loop für die maximale Anzahl der Attribute mit *„show()"*sichtbar gemacht, während alle Reihen-Rahmen (5 Attribute pro Reihe) unsichtbar geschaltet werden.

Danach wird aus der Liste für die maximale Anzahl an Attributen und der Liste für die aktuelle Anzahl der Attribute eine Liste gebildet, welche die aktuell nicht benötigten Attributnummern auflistet. Die Felder der nicht benötigten Attributnummern werden daraufhin unsichtbar geschaltet. Die benötigten

Reihen-Rahmen werden hingegen sichtbar gemacht. Die Reihenfolge bzw. der Ablauf des sichtbar und unsichtbar Machens kann dabei auch anders gestaltet werden, wichtig ist, dass am Ende nur die benötigten Attribut-Felder sichtbar sind.

Im nächsten Schritt werden die Inhalte aus allen Comboboxen der einzelnen Attribut-Felder gelöscht und eine Liste der Attribute des aktuellen Dateityps aus einer globalen Variable entnommen.

In Loops werden nun die Titel der einzelnen Attributfelder aus den Listen der Attributnummern und der Attribute des aktuellen Dateityps gesetzt, sowie die dazugehörigen Comboboxen mit allen im Layer vorhandenen Spalten gefüllt.

In nächsten Abschnitt („SET TEXTS") geht es nun um die den Dateityp beschreibenden Texte des zweiten Dialogfensters: Es werden die Überschrift inklusive Unterüberschrift, zwei Textfelder und die Kommentar-Box mit Informationen (Anzahl Layer-Attribute, Dateityp-Attribute, etc.) gesetzt. Weiterhin wird geprüft, ob eine Kommentar-Box für den aktuellen Dateityp in der Definitionsdatei vorgesehen ist und diese dann eventuell ausgeblendet.

Nun wird das zweite Dialogfenster mit dem *„show()"*-Befehl aufgerufen.

Im letzten Abschnitt werden globale Variablen mit Informationen über den aktuellen Dateityp und Layer im zweiten Dialogfenster überschrieben, sodass andere Funktionen direkt auf diese zugreifen können.

```
def reject_second_window():
```

Diese Funktion definiert, was passieren soll, wenn das zweite Dialogfenster mit dem *„Cancel"*-Button beendet wird.

Da diese Funktion durch Klick auf einen Button ausgelöst wird und dabei keine Parameter übergeben werden können, wird der aktuelle Dateityp des zweiten Dialogfensters aus einer globalen Variable ausgelesen.

Die zu diesem Dateityp gehörige Combobox für die Layerselektion wird daraufhin freigeschaltet. Dies ist aktuell nicht unbedingt nötig, da die Combobox schon mit Klick auf den *„manually"*-Button freigeschaltet wurde, nachdem der *„by name"*-Button sie vorher eventuell gesperrt hatte. Dies ist allerdings eine neue Entwicklung, um eine weitere Sperrung der Combobox durch irreguläres Beenden des zweiten Dialogfensters zu verhindern.

Diese Funktion könnte daher zur Vermeidung von Redundanzen gelöscht werden, der Gewinn an Performanz und Speicherplatz ist aber marginal. Es wurde entschieden sie vorerst beizubehalten, da weitere Entwicklungen in diesem Bereich nicht auszuschließen sind.

```
def execute_second_window():
```

Diese Funktion steuert das Verhalten des Plugins für den Fall, dass die Benutzereingaben im zweiten Dialogfenster mit einem Klick auf „OK" akzeptiert werden.

Da auch diese Funktion über einen Button getriggert wird und daher keine Parameter mitnehmen konnte, muss der aktuell im zweiten Dialogfenster behandelte Dateityp aus der entsprechenden globalen Variable ausgelesen werden.

Weiterhin wird der passende *„manually"*-Button im ersten Dialogfenster als *„checked"* markiert, der aktuelle Dateityp in die Liste der zu exportierenden Typen aufgenommen und die entsprechende Combobox für die Layerselektion gesperrt.

Abschließend werden die vom Benutzer im zweiten Dialogfenster den einzelnen Attributen (des Dateityps) zugewiesenen Spalten (des Layers) in einem globalen Dictionary für den Dateityp gespeichert. Hierbei kommen wieder die *„exec()"*- und *„eval()"*-Befehle zum Einsatz, um die Anweisungen dateitypspezifisch auszuführen.

5.2 Sortier-Funktionen

```
def create_dict_by_name(filetype):
```

Wenn der Benutzer auf einen *"by name"*-Button klickt, wird (neben den Reaktionen der GUI) diese Funktion zum Sortieren der Layer-Spalten anhand ihrer Titel angesteuert.

Da der Button jeweils für einen bestimmten Dateityp zuständig ist, gibt er der Funktion direkt den relevanten Dateityp beim Aufruf mit auf den Weg. Für diesen wird dann der passende Arrayindex und damit die Anzahl seiner Attribute ermittelt.

Aus dieser Anzahl wird dann ein Liste generiert, wobei einstellige Zahlen eine führende Null erhalten.

Auch wird eine Liste der Attributnamen des Dateityps aus einer globalen Variable entnommen, sowie der Daten-Quell-Layer bestimmt.

Für diesen Daten-Quell-Layer wird dann eine Liste der in ihm vorhandenen Spalten (fields) aufgestellt.

Im darauffolgenden Abschnitt „CREATE NEW DICT" wird zuerst ein neues temporäres Dictionary erstellt, welches dann mit Schlüsseln und leeren Werten für jedes Datei-Attribut gefüllt wird.

Danach werden in einem Loop alle Spalten des neuen Layers überprüft: Wenn der Titel der Spalte auch in der Liste der Attribute des Dateityps auftaucht, wird

der Index der Spalte bestimmt und um 1 erhöht, sowie (falls nötig) mit einer führenden Null versehen (Index startet bei 0, die Attributnummerierung bei 01).

Für diese Suche werden der Spaltentitel und die Liste der Attributnamen temporär in Großbuchstaben (*„uppercase"*) umgewandelt, sodass Groß- & Kleinschreibung beim Abgleichen keine Rolle spielt.

Aus dem veränderten Index wird nun der Schlüssel für das neue Dictionary generiert, welchem dann der Titel der zugewiesenen Layer-Spalte als Wert zugewiesen wird.

Abschließend wird das temporäre Dictionary mit seinen Layer-Spalten und leeren Werten als globale Variable für den aktuellen Dateityp gespeichert (bzw. gegebenenfalls überschrieben).

`def match_attributes_by_name_if_possible():`

Diese Funktion wird durch einen Button im zweiten Dialogfenster ausgelöst und versucht die manuelle Attributsortierung damit zu erleichtern, dass vom Titel her passende Layer-Spalten den entsprechenden Dateityp-Attributen zugewiesen werden.

In einem kombinierten Loop für die beiden globalen Listen, der für den gewählten Dateityp nötigen Attributnummern und Attributbezeichnungen, wird Folgendes getan:

Jedes Attribut-Feld des zweiten Dialogfensters hat eine Combobox mit allen Spalten des Layers. In jeder dieser Comboboxen wird nach dem zum jeweiligen Attribut-Feld passenden Titel der Spalte gesucht. Die *„.findText"*-Methode wird dabei mit dem Parameter *„MatchFixedString"* aufgerufen, der dafür sorgt, dass Groß- & Kleinschreibung ignoriert wird.

Das Ergebnis der Suche wird als Index ausgegeben; falls die Suche ergebnislos verläuft, ist der Index -1.

Im nächsten Schritt wird (falls der Index nicht -1 ist) die gerade untersuchte Combobox auf den Index des gefunden Spaltentitels gesetzt. Somit sind am Ende allen Attribut-Feldern, falls vorhanden, die passenden Spalten zugeordnet.

`def match_attributes_by_order():`

Diese Funktion gehört ebenfalls zu einem Button des zweiten Dialogfensters und ordnet den Attribut-Feldern die Layer-Spalten in ihrer originalen Reihenfolge zu.

In einem Loop für alle Nummern zwischen 1 und der Anzahl der im Layer vorhandenen Spalten + 1 (weil *„range()"*) passiert Folgendes:

Die aktuelle Nummer wird in eine zweite Variable kopiert und bekommt gegebenenfalls eine führende Null. In einer *„eval()"*-Funktion wird dann in der Combobox im Feld für das Attribut mit der aktuellen Nummer, der Wert ausgewählt, welcher als Index die aktuelle Nummer hat. Dies Funktioniert, da

all diese Comboboxen beim Index „0" einen leeren Wert eingesetzt bekamen, um gegebenenfalls „nichts" auswählen zu können. Daher verlaufen die Nummerierung der Attribute parallel zu den Indices.

```
def clear_all_matches():
```

Mit dieser Funktion werden alle Zuweisungen von Layer-Spalten zu Dateityp-Attributen im zweiten Dialogfenster zurückgesetzt.

In einem Loop für alle Nummern der im aktuellen Dateityp vorhandenen Attribute wird mit einer *„eval()"*-Funktion die jeweilige Combobox angesprochen und deren Index auf „0" gesetzt, wo sich bei diesen Comboboxen ein leerer Wert befindet.

5.3 Erzeugen & Export von BlueM-Dateien

```
def export_clicked():
```

Diese Funktion wird ausgelöst, wenn der Benutzer auf den *„Export"*-Button klickt und steuert den weiteren Export-Ablauf.

Zuerst wird die Anzahl der in der globalen Liste gespeicherten, für den Export vorgesehen Dateitypen ermittelt. Sollte diese 0 sein, wird dem Benutzer ein entsprechender Hinweis gegeben. Falls jedoch Dateitypen in der Liste vorhanden sind, werden diese in einer Nachricht für den Nutzer angezeigt.

Weiterhin wird ein Windowsfenster des gewählten Exportverzeichnisses geöffnet.

Als Nächstes wird in einem Loop geprüft, welche *„by name"*- & *„manually"*-Buttons aktuell gecheckt sind. Dies geschieht für die Standardexporte in einem Loop mit Verweis auf die allgemeine Exportfunktion; für das TAL-file jedoch einzeln, da es auf einen TAL-spezifischen Export verweist.

Alternativ zur Button-Abfrage wäre es auch möglich, die einzelnen Exporte aus der Liste der zu exportierenden Dateien anzuweisen.

Abschließend wird geprüft, ob der Benutzer eine separate Datei mit den gesammelten Export-Informationen wünscht (was gegebenenfalls beauftragt wird) und ob das Plugin nach dem Export geschlossen werden soll.

```
def export_file(filetype):
```

Für jeden zu exportierenden Standarddateityp wird diese Funktion aufgerufen, welche die Informationen aus dem selektierten Layer für die entsprechende BlueM-Datei formatiert und dann abspeichert.

Zuerst werden alle relevanten Information zum aktuellen Dateityp (selektierter Layer, Füllmuster, Tabellenüberschriften, etc.) mit der Support-Funktion *„get_type_info(filetype)"* bezogen, sowie die zum Dateityp passenden Dictionaries für Attribut-Namen & -Definitionen aus den globalen Variablen in temporäre Variablen umgesetzt.

Aus dem Attribut-Namen-Dictionary (entweder aus *„by name"* oder *„manually"*-Zuordnung) wird nun eine Liste der Dateityp-Attribute erstellt, die keine Layer-Spalte zugewiesen bekommen haben. Ebenfalls wird gezählt, wie viele Dateityp-Attribute eine entsprechende Layer-Spalte besitzen.

Am Ende des ersten Bereichs wird der spätere BlueM-Dateiname mithilfe einer Support-Funktion *„construct_filename(filetype)"* konstruiert, sowie eine leere Liste für potenzielle Warnmeldungen definiert.

Im Abschnitt „WRITE THE FILE" wird nun begonnen die mit ANSI kodierte BlueM-Eingangsdatei zu schreiben. Wobei mit dem simplen Einfügen des Überschriftenblocks (*„headlines"*) aus der Definitionsdatei angefangen wird.

Danach wird in einem Loop durch alle Elemente (*„feature"*) des Layers iteriert, wobei für jedes Element eine temporäre Liste (*„current_row"*) für eine Zeile in der späteren BlueM-Datei definiert und dann gefüllt wird.

Dazu wird in einem weiteren Loop für alle Attributnummern des Dateityps Folgendes getan:

- Die Definition des Attributes mit der aktuellen Nummer wird aus dem entsprechenden Dictionary entnommen, formatiert (Leerzeichen waren eventuell nicht korrekt eingelesen worden) und die maximale Zeichenzahl bestimmt.
- Falls dem aktuellen Attribut keine Layer-Spalte zugewiesen wurde, wird eine passende Anzahl Leerzeichen in die Liste für die aktuelle Zeile aufgenommen.
- Sollte eine Layer-Spalte zugewiesen worden sein, wird der Zellenwert in dieser Spalte für das aktuelle Layer-Element ausgelesen.
- Für diesen Wert wird dann die später beschriebene Funktion *„check_value()"* aufgerufen, welche einen geprüften (und eventuell korrigierten) Wert samt einer möglichen Warnung über Wertänderungen zurückgibt.
- Sollte eine Warnung zurückgekommen sein, wird diese zusammen mit den Koordinaten des betreffenden Wertes (Element & Attributnummer) in die Liste der Warnmeldungen für diesen Dateityp aufgenommen.

Im nächsten Abschnitt „IMPLEMENT FILETYPE SPECIFIC FORMATS" werden kleinere Besonderheiten des jeweiligen Dateityps behandelt.

Sollte gerade eine FKT-Datei erstellt werden, so wird ihre spezielle Tabellenaufteilung beachtet:

- Ist ein Eintrag für die Datei-Spalte „Bezeichnung" vorhanden, wird ein die ganze Tabelle durchziehender, horizontaler Trennstrich über der aktuellen Zeile eingefügt.
- Falls die „Bezeichnung" leer, aber eine „Funktion" benannt ist, wird ab der Spalte „Funktion" ein Trennstrich eingefügt.

- Analog wird ein Trennstrich eingefügt, wenn in der dritten Spalte ein neuer Abschnitt beginnt.

Für die Erstellung einer KTR-Datei ist zu beachten, dass diese aus zwei Tabellen besteht: Zuerst die Kontroll-Funktionen und dann die Kontroll-Gruppen (KGRP); dazwischen befindet sich eine Erläuterung der Kontrollgruppen.

Dies wird realisiert indem, sobald der erste Eintrag einer Zeile mit „K" beginnt, geprüft wird, ob die Variable *„kgrp_count*" existiert. Sollte sie existieren, passiert nichts. Falls sie aber noch nicht existiert, wird ein *„UnboundLocalError*" ausgeworfen. Da diese Prüfung mit einem vorangestellten *„try*"versucht wurde, wird dieser spezifische Fehler registriert und automatisch damit behandelt, dass die gesuchte Variable erzeugt und die entsprechende Trennung der Tabellen durch Einfügen eines vordefinierten Textblocks vollzogen wird. Diese etwas unorthodoxe Methode funktioniert, solange alle KGRPs im Layer unter den Kontrollfunktionen stehen. Falls nicht würde eine Nicht-KGRP in der unteren Tabelle stehen, was jedoch hauptsächlich ein Übersichtlichkeitsproblem und keinen kritischen Fehler darstellt.

Auch die „ALL"-Datei muss besonders behandelt werden, da sie nicht die Form einer Tabelle hat, sondern eher einer Liste entspricht. Diese Liste wird nur mit den Einträgen aus einem Element gefüllt, weshalb das Plugin nur das erste Element des Quell-Layers für die ALL-Datei in diese schreibt. Alle weiteren Elemente werden ignoriert.

Danach wird geprüft, ob der erste Eintrag einer Zeile dem Spaltentitel entspricht. Dies kann vorkommen, wenn die Spaltentitel aus Excel-Dateien nicht automatisch von QGIS als solche erkannt werden und stattdessen als Erstes Element behandelt werden. Das Plugin ermöglicht für diesen Fall das Setzen der Spaltentitel aus der ersten Zeile des Layers, wobei diese jedoch erhalten bleibt. Die aktuelle Funktion prüft daher auf Zeilen, die den Spaltentiteln entsprechen und schreibt diese nicht in die BlueM-Datei.

Als Letztes in diesem Abschnitt wird noch ein Textblock in die Datei geschrieben, welcher den Abschluss der Tabelle bildet.

Der nächste Abschnitt „GET THE EXPORT INFORMATION INTO THE FILE" beschäftigt sich damit, weiterführende Informationen (Quell-Layer, Exportdatum, etc.), sowie die gesammelten Warnmeldungen der Werte, in die BlueM-Datei zu schreiben.

Zuerst wird aus Textbausteinen und Variablen ein Informationstext zusammengesetzt, an welchen dann eine Liste von Attributen angehängt wird, die im Layer keine Entsprechung gefunden haben.

Danach wird eine auf die Anzahl der Warnmeldungen grammatikalisch abgestimmte Überschrift für die Liste der Warnmeldungen generiert und mit dieser entsprechend formatierten Liste dem restlichen Informationstext hinzugefügt.

Nun wird geprüft, ob der Nutzer diese Export-Information in der jeweiligen BlueM-Datei wünscht, wo sie dann gegebenenfalls auch eingefügt wird. Weiterhin wird die generierte Exportinformation in einer globalen Variable

gespeichert, von wo sie dann (sofern gewünscht) in eine Gesamt-Export-Informationsdatei (siehe *„def export_file_export_information()"*) geschrieben werden kann.

```
def check_value(value, req_length, req_type):
```

Mit dieser Funktion wird ein mitgelieferter Wert auf seine Länge und seinen Datentyp geprüft, falls nötig angepasst und dann als Text samt eventueller Warnmeldungen zurückgegeben.

Zuerst wird die Variable für die Warnmeldungen als leer definiert. Danach wird aus der mitgelieferten Attributdefinition (*„req_type"*), welche aus der Definitionsdatei stammt und gleichzeitig die maximale Zeichenlänge definiert, der gewünschte Datentyp des Wertes bestimmt.

Im nächsten Schritt wird „versucht" (mit *„try"*) die einzelnen Datentypen zu behandeln:

Soll der Wert in der BlueM-Datei als ***„String"*** (Text) auftauchen, so wird versucht den Wert mittels *„str()"*-Funktion als String zu definieren. Weiterhin werden eventuelle Sonderzeichen (eckige Klammern, etc.) am Anfang und Ende des Textes entfernt; diese können beim Einlesen von Daten in QGIS automatisch gesetzt werden.

Auch wird geprüft ob der Text „NULL" oder „None" ist, was fälschlicherweise von QGIS für leere Werte automatisch gesetzt, oder sogar schon in der Quelldatei des Layers so definiert worden sein könnte. Diese werden durch Leerzeichen ersetzt.

Danach wird geprüft, ob der Text länger ist als später in die Spalte der BlueM-Datei passen würde. Falls ja, wird der Text gekürzt und eine entsprechende Warnmeldung gespeichert. Sollte der Text kürzer sein als notwendig, wird er mit Leerzeichen aufgefüllt.

Falls der Zellenwert eine Ganzzahl oder Kommazahl sein soll und ein Komma enthält, wird dieses durch einen Punkt ersetzt, da BlueM (und auch Python) einen Punkt als Dezimaltrennzeichen verwendet.

Im nächsten Schritt wird der Wert behandelt, falls es sich um einen ***„Integer"*** (Ganzzahl) handeln soll:

Da der Quell-Layer eventuell unpassend formatiert war und beispielsweise eine „3" als „3.0" gespeichert hatte oder dem Benutzer nicht bekannt war, dass hier nur Ganzzahlen verwendet werden können, wird der Wert zuerst als Dezimalzahl behandelt und auf Null Nachkommastellen gerundet. Danach wird der Wert in einen Integer umgewandelt. Sollte der gerundete Wert nicht dem Original entsprechen, wird eine Warnmeldung über die Rundung abgespeichert.

Danach wird die Zeichenlänge der Ganzzahl geprüft: Sollte sie zu lang sein, um in die spätere Spalte zu passen, wird sie durch das benutzerdefinierte Ersatzzeichen ersetzt und eine entsprechende strenge (mit „!!!") Warnmeldung gespeichert. Falls die Zahl zu kurz ist, werden davor Leerzeichen eingefügt.

Als Nächstes werden Werte geprüft, die eine Dezimalzahl (***„float"***) sein sollen:

Es wird direkt versucht den Wert als eine Dezimalzahl zu definieren (was passiert, falls dieser Versuch fehlschlägt, wird später erläutert).

Danach wird die Zeichenlänge der Dezimalzahl vor dem Komma bestimmt. Falls diese länger als die zulässige Zeichenlänge ist, wird die Zahl durch das Ersatzzeichen ersetzt und eine entsprechende Warnmeldung gespeichert.

Falls die maximale Zeichenlänge dafür sorgen würde, dass der Dezimal-Punkt das letzte schreibbare Zeichen der Zahl wäre, wird die Dezimalzahl in eine Ganzzahl umgewandelt.

Sollte die Zahl aufgrund ihrer Nachkommastellen zu lang für die Spalte der BlueM-Datei werden, wird sie auf die maximal abbildbare Anzahl an Nachkommastellen gerundet (inklusive Warnmeldung); wobei ein dadurch erzeugtes „.0" am Ende der Zahl abgeschnitten wird.

Sollte die Dezimalzahl zu kurz für die Spalte sein, werden ihr Leerzeichen vorangestellt.

Nun werden Werte behandelt, die später einer **J/N Entscheidung** (gewähltes Datentyp-Kürzel: „Y") entsprechen sollen:

Hier wurden zwei Listen definiert, die jeweils Werte enthalten, die als positive oder negative Entscheidung gewertet werden können. Neben den vordefinierten „J"/"N" finden sich hier „Ja"/"Nein", verschiedene Fremdsprachen, sowie „a"/„r", welche in Exceltabellen (in der Sprache Marlett) gerne als Häkchen beziehungsweise ansprechenderes „X" verwendet werden.

Wird der Zellenwert des Layers in einer der beiden Liste wiedergefunden, wird der Wert mit „J" bzw. „N" überschrieben. Falls der Wert nicht erkannt wird, kommt wieder das benutzerdefinierte Ersatzzeichen zum Einsatz und eine entsprechende Warnmeldung wird gespeichert.

Als letzter Datentyp wird **Datum & Uhrzeit** verarbeitet:

Auch hier werden zuerst etwaige Sonderzeichen entfernt, sowie „NULL" und None etc. durch Leerzeichen ersetzt.

Es kommt häufig vor, dass Datums- bzw. Uhrzeitangaben aus Excel von QGIS automatisch in sogenannte *„QdateTime"*-Elemente umgewandelt werden. Dieser Fall wird vom Plugin abgefangen, indem geprüft wird, ob der aktuelle Wert als String mehr als die (ansonsten für Datum & Uhrzeit) maximal nötigen 16 Zeichen lang ist.

Dies wäre der Fall, wenn Python den Wert von QGIS als ein solches *„QdateTime"*-Elemente bekommen hat. Dieses Problem wird behoben, indem die einzelnen Informationen zu Jahr, Monat, Tag, Stunde und Minute aus diesem Element herausgezogen und als eigene Variablen gespeichert werden.

Aus diesen Variablen können dann die 4 unterschiedlichen DateTime-Formate von BlueM konstruiert werden.

Es werden aber nicht immer alle Datums- und/oder Uhrzeitangaben derartig behandelt; das von BlueM verwendete Format „TT.MM" wird beispielsweise weder in Excel noch QGIS als Datumsangabe verstanden. Dies verhindert auch den Einsatz eines spezialisierten Parser-Moduls wie beispielsweise „dateutil.parser".

Falls der Zellenwert also nicht mehr als 16 Zeichen lang ist, wird versucht ihn unformatiert in die BlueM-Datei zu schreiben. Falls er jedoch nicht in die Spalte passen würde, wird er durch das Ersatzzeichen ausgetauscht und eine entsprechende Warnmeldung gespeichert.

Auch wird der Fall behandelt, dass kein Datentyp für den Zellenwert festgestellt werden konnte (was nur bei einer fehlerhaften Definitionsdatei der Fall sein kann). Dies provoziert ebenfalls eine Warnmeldung.

Da alle Behandlungen der Datentypen unter dem *„try"*-Vorbehalt durchgeführt wurden, kann ein geworfener „ValueError" hier abgefangen werden: Sollte beim Versuch den Zellenwert in den gewünschten Datentyp umzuformen, dieser Fehler ausgelöst werden, wird ein eventuell „leerer" Wert durch Leerzeichen ersetzt. Sollte jedoch der Wert einen Inhalt haben, der nur nicht korrekt eingeordnet werden konnte, kommt das Ersatzzeichen zum Einsatz und ein entsprechender Warnhinweis wird abgesetzt.

Am Ende der Funktion werden behandelte Werte als Text, sowie die eventuell dazugehörige Warnmeldung zurückgegeben.

```
def export_tal_file():
```

Da das TAL-file aufgrund seiner internen Komplexität nicht über den standardisierten Dateiexport abgehandelt werden kann, gibt es hier eine spezielle Funktion, die von der Exportsteuerung explizit angesprochen wird und prüft, ob überhaupt ein TAL-file konstruiert werden kann.

Der zu untersuchende Dateityp wird selbstverständlich als *„TAL"* definiert. Mit dieser Information werden der selektierte Layer und das Attribut-Zuordnungs-Dictionary ermittelt.

Nun wird geprüft, ob dem ersten Attribut des TAL-files eine Layer-Spalte zugewiesen wurde. Falls nicht, wird die Funktion *„export_tal_file_failed(reason)"* mit einer entsprechenden Begründung aufgerufen, denn das TAL-file kann nicht ohne eine Spalte für das erste Attribut konstruiert werden.

Sollte eine Layer-Spalte für das erste Attribut definiert sein, wird diese als *„main_attribute_field_name"* gespeichert. Danach werden alle Werte in dieser Layer-Spalte einer Liste hinzugefügt. Anschließend wird diese Liste in eine Liste ohne potenzielle Duplikate umgesetzt, die auch nur Werte enthalten darf, die mit „T" beginnen.

Nun wird geprüft wie viele Elemente diese neue Liste besitzt: Sollte die Liste leer sein, wird die *„export_tal_file_failed(reason)"*-Funktion mit der Begründung gestartet, dass ohne Talsperren-ID (welche mit „T" beginnen) keine TAL-Datei konstruiert werden kann.

Ist die Liste jedoch nicht leer, also mindestens eine Talsperren-ID enthalten, wird die Funktion *„export_tal_file_possible()"* gestartet, welche dann das eigentliche TAL-file konstruiert und exportiert.

`def export_tal_file_failed(reason):`

Diese Funktion wird aufgerufen, um dem Nutzer mitzuteilen, dass kein TAL-file exportiert werden konnte und den Grund dafür zu erläutern.

Zuerst wird die beim Funktionsaufruf mitgelieferte Begründung (*„reason"*) für das Fehlschlagen des TAL-Exports in eine umfassendere Begründung ausgedehnt. Diese sind für die beiden Gründe (entweder keine Layer-Spalte für Talsperren-IDs definiert oder keine Talsperren-IDs in dieser Spalte) vorgefertigt.

Danach wird die aktuell in der QGIS-Message-Bar für den Nutzer sichtbare Meldung zum laufenden Export von Dateien entfernt und durch eine Warnmeldung zum Scheitern des TAL-Exports ersetzt.

Ebenfalls wird in das Exportverzeichnis eine Datei geschrieben, die im Namen und im Inhalt das Scheitern erläutert. Zusätzlich wird die Fehlermeldung auch in eine globale Variable geschrieben, von wo aus sie in die Gesamt-Export-Informationsdatei übernommen werden kann.

`def export_tal_file_possible():`

Sollte die Prüfung der für das TAL-file selektierten Layer-Daten grünes Licht geben, wird diese Funktion gestartet. Da die TAL-Datei aus einer Übersichtstabelle und 6 einzelnen Tabellen für jedes Element der Übersichtstabelle besteht, ist die hier vorgenommene Konstruktion der Ausgabedatei deutlich komplizierter als bei einem standard Exportvorgang.

Zuerst wird, wie bei der *„export_file"*-Funktion, mit dem Dateityp via *„get_type_info"* eine ganze Reihe an Informationen über den selektierten Layer und aus dem Definitions-Array bestimmt. Ebenfalls werden die Dictionaries für die Spaltenzuordnung und die Attributdefinitionen aus den entsprechenden globalen Variablen entnommen.

Danach wird, wie bei einem standard Exportvorgang, eine Liste von Attributen erstellt, denen keine Layer-Spalte zugeordnet werden konnte, die Anzahl der erfolgten Zuordnungen gezählt, der spätere Dateiname samt Zeitpunkt mit der *„construct_filename"*-Funktion bestimmt und eine leere Liste von Warnmeldungen definiert.

Im darauffolgenden Abschnitt „GET VARIABLES FOR ALL 7 SUB-TABLES" werden die im Layer als eine einzige Tabelle vorhandenen Daten horizontal aufgespalten, sodass daraus praktisch 7 Unter-Tabellen entstehen (1 für die Übersichtstabelle und 6 verschiedene Arten von weiteren Tabellen).

Als Erstes wird das Füllmuster (*„pattern"*) aus der Definitionsdatei, welches aus 7 Zeilen besteht in diese aufgeteilt, sodass 7 tabellenspezifische Füllmuster entstehen.

In jedem Füllmuster wird dann geprüft, wie viele Attribute darin Platz finden – ein Attributplatz wird durch ein „{}" im Füllmuster signalisiert. Aus diesen Attributzahlen wird nun für jede Tabelle ein Bereich (*„range"*) von Attributnummern definiert, die zu dieser Tabelle gehören (im Sinne von: Tabelle „X" enthält die Attribute 13 bis 21).

Für jede der 7 Tabellen wird dann ein eigenes Dictionary erstellt, welches mit den dazugehörigen Attributnummern und den entsprechenden Layer-Spalten gefüllt wird. Die Attributdefinitionen werden analog in 7 Dictionaries aufgespalten.

Im nächsten Abschnitt „DIVIDE LAYER FEATURES (ROWS) INTO MAIN ATTRIBUTES OF FILE" werden die Layer-Daten dann vertikal in die einzelnen Hauptelemente (Talsperren-IDs) aufgespalten.

Zuerst wird der, dem ersten Dateityps-Attribut im Layer zugewiesene Spaltentitel, aus dem Dictionary bestimmt. Ähnlich dem Vorgang beim Prüfen der Validität der Layer-Daten für den TAL-Export wird nun eine Liste der Talsperren-IDs (Diese beginnen mit „T") erzeugt.

Der Inhalt dieser Liste von Talsperren-IDs wird danach in einem Loop durch alle Elemente genutzt, um deren jeweilige Zeilennummer im Layer herauszufinden. Da die einzelnen Talsperren-IDs der Dreh- und Angelpunkt der gesamten TAL-Datei sind, werden die Zeilennummern des Beginns eines Abschnitts einer bestimmten Talsperren-ID, sowie dessen Endes in einem Dictionary gespeichert.

Aus diesen Informationen wird dann zur vertikalen Einteilung ein weiteres Dictionary mit den Bereichen (*„range"*; analog zur horizontalen Einteilung) erzeugt. Somit hat die laufende Funktion nun alle Informationen um die Daten des Layers den einzelnen Tabellen des TAL-files korrekt zuzuweisen.

Dies wird im folgenden Abschnitt „WRITE FILE" praktisch umgesetzt:

Zuerst wird mit dem konstruierten Dateinamen im Exportverzeichnis eine ANSI-kodierte Datei angelegt und der Überschriftenblock für die Übersichtstabelle (*„main table"*) eingefügt.

Danach werden in einem verschachtelten Loop für alle Layer-Elemente deren Informationen für alle Attribute der Übersichtstabelle entnommen, geprüft und einer Variable für die aktuelle Zeile im TAL-file hinzugefügt:

Im ersten Schritt wird jede Attributnummer im Bereich der Übersichtstabelle auf die zweistellige Schreibweise (mit eventuell führender Null) angepasst und

damit die entsprechende Attributdefinition ausgelesen. Diese Definition wird dann formatiert und deren Zeichenlänge bestimmt.

Sollte keine Layer-Spalte für das aktuelle Attribut der Übersichtstabelle vorhanden sein, wird die entsprechende Anzahl an Leerzeichen in die Zeilenvariable eingefügt.

Falls jedoch eine Layer-Spalte zugewiesen wurde, wird der für das aktuelle Element verzeichnete Wert ausgelesen und dieser in der bekannten *„check_value*"-Funktion untersucht. Danach wird der geprüfte Wert der aktuellen Zeilenvariable hinzugefügt und die eventuelle Warnmeldung aus der Wertüberprüfung mit den Koordinaten des aktuellen Wertes in die Liste der Warnmeldungen aufgenommen.

Da die einzelnen Unter-Tabellen des TAL-files aus den Layer-Elementen häufig Zeilen generieren, die komplett leer sind (da die einzelnen Tabellen unterschiedliche „Höhen" haben), wird nun geprüft, ob die aktuelle Zeilenvariable überhaupt einen validen Eintrag besitzt. Dazu kommt die vordefinierte Liste von validen Zeichen zum Einsatz. Sollte die Zeile valide Zeichen besitzen, wird sie mit dem passenden Füllmuster in das TAL-file geschrieben – ansonsten wird sie vom Plugin übersprungen.

Danach wird die Bearbeitung der Übersichtstabelle mit dem Einfügen eines abschließenden Zeichenblocks beendet.

Nun geht es um die Erzeugung der 6 einzelnen Untertabellen für jedes Element der Übersichtstabelle (Talsperren-ID): In einem Loop für jede Talsperren-ID (aus der Liste) wird der dazugehörige Layer-Zeilen-Bereich aus dem entsprechenden Dictionary entnommen, die Überschrift der folgenden Unter-Tabellen an die aktuelle Talsperren-ID angepasst und in die Datei geschrieben.

Danach wird in einem Loop für alle 6 Untertabellen-Typen Folgendes getan:

- In einem verschachtelten Loop wird für alle Layer-Elemente des aktuellen Bereichs (zur aktuellen Talsperren-ID gehörend) eine leere Zeilenvariable definiert.
- Ein weiterer Loop wird für alle zur aktuellen Untertabelle gehörenden Attribute gestartet.
- Dadurch ist der zweidimensionale Bereich des Layers definiert, welcher zur aktuellen Talsperren-ID die entsprechenden Attribute für die aktuelle Untertabelle enthält.
- In diesem Bereich werden nun alle im Layer enthaltenen Werte, samt der entsprechenden Attributdefinitionen, durch die aus der Übersichtstabelle und dem standard Exportvorgang bekannte Werteprüfung geschleust. Entsprechend abgeändert, in die Zeilenvariable eingefügt und die Zeile dann (sofern sie nicht leer ist) mit dem tabellenspezifischen Füllmuster in die Datei geschrieben.
- Nach allen, zur aktuellen Untertabelle gehörenden Zeilen, wird der, die Tabellen trennende und vordefinierte, Textblock eingefügt.
- Dies wird für alle 6 Untertabellen-Typen der aktuellen Talsperren-ID wiederholt und dann für alle weiteren Talsperren-IDs durchgeführt.

Danach werden (wie beim standard Exportvorgang ausführlich beschrieben) die Exportinformationen und gesammelten Warnmeldungen aus der Werteprüfung in Textform gegossen, sofern vom Benutzer gewünscht, ebenfalls in die Datei geschrieben und für den Gesamt-Export-Bericht abgespeichert.

5.4 Andere Exporte

```
def export_user_manual():
```

Dies ist eine kurze Funktion, welche das in den Plugin-Dateien gespeicherte Benutzerhandbuch (als PDF) in den vom Benutzer aktuell gewählten Export-Ordner kopiert, damit er nicht in den Tiefen des Programmordners danach suchen muss und dabei eventuell ungewollte Veränderungen vornimmt.

Zuerst wird der Dateiname des Benutzerhandbuchs, sowie der Pfad zu dieser als Variable festgelegt. *„__file__"* ist dabei die aktuell ausgeführte Python-Datei aus der über das Modul *„os"* der Verzeichnisname ausgelesen wird. Dieser ist derselbe wie beim Benutzerhandbuch, da beide im gleichen Ordner liegen. Weiterhin wird der vom Benutzer gewählte Zielordner aus dem Widget ausgelesen.

Das Modul *„shutil"* ermöglicht das Kopieren der PDF-Datei aus dem bekannten Verzeichnis in das ausgewählte Zielverzeichnis. Dieses Zielverzeichnis wird daraufhin via der Funktion *„open_explorer_window()"* geöffnet.

```
def create_geopackage():
```

Diese Funktion erzeugt ein GeoPackage mit Layern für alle potenziellen BlueM-Dateien, welche wiederum alle notwendigen Spalten mit passenden Namen und Formatierung enthalten.

Zu Beginn wird mit der Funktion *„open_explorer_window()"* das Zielverzeichnis geöffnet, damit der Nutzer bei der Erzeugung des GeoPackages „zuschauen" kann. Gleichzeit wird in QGIS eine Nachricht angezeigt, die darauf hinweist, dass dies ein langwieriger Prozess ist – diese Nachricht muss sogar durch den *„repaint"*-Befehl wiederholt werden, da sie ansonsten erst nach dem ressourcen-hungrigen Erzeugungsprozess angezeigt würde.

Weiterhin wird der Export-Pfad des GeoPackages aus dem Pfad-Widget ausgelesen und der Dateiname konstruiert (entweder aus dem Voreingestellten oder dem „project name"-Feld im Plugin).

Als Nächstes wird in einem Loop für alle gespeicherten BlueM-Dateitypen ein Layer erzeugt. Diesen Layern werden je nach Dateityp unterschiedliche Geometrietypen zugewiesen, die für die späteren Elemente des Layers am passendsten sind. Für den ersten Layer des Loops (durch *„enumerate"* werden die Layer im Loop durchnummeriert) wird ein GeoPackage mit dem ermittelten

Namen und Pfad erzeugt – alle weiteren Layer werden als „Update" in dieses GeoPackage eingefügt.

Abschließend wird das erzeugte GeoPackage mit QGIS aufgerufen, sodass der Nutzer auswählen kann, welchen Layer er aktuell ins laufende Projekt laden will.

`def export_standards():`

Ermöglicht den Export der im Plugin definierten Standards für BlueM-Dateien und ihre Attribute als CSV-Datei, um in Excel oder Ähnlichem mit der Vorbereitung von BlueM-Daten zu beginnen.

Zuerst wird ein Windows-Fenster des Zielordners geöffnet, der Ziel-Pfad aus dem Widget ausgelesen und der Dateiname gesetzt. Mit der *„with open"*-Methode wird das Schreiben einer Datei gestartet.

Die neue Datei wird zuerst mit einer vordefinierten Überschrift und Erklärungen gefüllt. Gefolgt von einem Loop für alle gespeicherten Dateitypen.

In diesem Loop werden als Erstes die Namen der Attribute des jeweiligen Dateityps, aus einer globalen Variable für diesen Typ, mithilfe einer *„eval"*-Funktion (wandelt einen String in eine Variable um) als Liste importiert. Für die Anzahl der Attribute des Dateityps wird nun eine Liste mit zweistelligen Nummern erzeugt. Auch wird ein globales Dictionary mit den Attribut-Beschreibungen des Dateityps importiert und diese Beschreibungen in eine Liste gepackt.

Nun wird für jeden Dateityp des Loops eine dreizeilige Tabelle in die Standards-Datei geschrieben. Zuerst eine Überschriften-Zeile mit dem Namen des Typs und der jeweiligen Nummer des Attributes. Darunter der Name des jeweiligen Attributes, gefolgt von der Beschreibung/Definition dieses Attributes. Die Informationen für diese Tabellen stammen aus den in der aktuellen Instanz des Loops erzeugten Listen.

`def export_file_export_information():`

Diese Funktion erzeugt im Export-Ziel-Ordner eine Textdatei mit den gesammelten Export-Informationen der einzelnen Datei-Exporte im aktuellen Exportvorgang (mit Zeitstempel), sodass die Informationen nicht in den einzelnen BlueM-Dateien stehen müssen und dem Benutzer hier einen Überblick geben können. Auch bekannt als „Export-Log".

Zu Anfang wird der Dateipfad ausgelesen, sowie das aktuelle Datum samt Uhrzeit bestimmt – der Zeitstempel wird dann für die Nutzung im Dateinamen und innerhalb der Datei in zwei Variablen formatiert.

Im nächsten Schritt wird der Dateiname inklusive Pfad konstruiert. Auch wird mittels einer globalen Variable (*„list_filetypes_for_export"*) geprüft, wie viele

Dateien im aktuellen Vorgang exportiert wurden und eine Variable (für diese Anzahl entsprechend) grammatikalisch korrekt formatiert.

Nun wird mit der *„with open"*-Methode eine neue Datei im Zielordner erzeugt und mit einer grammatikalisch korrekten Überschrift gefüllt.

Innerhalb eines Loops wird dann für jede exportierte BlueM-Datei dessen spezifische Export-Informationen aus einer globalen Variable importiert, deren Grammatik angepasst („This" -> „The" XXX-file) und dann in die Export-Übersichts-Datei geschrieben.

5.5 GUI-Funktionen

```
def general_gui_functions():
```

Diese sehr umfangreiche Funktion erfasst die Eingaben des Anwenders in der grafischen Benutzeroberfläche (GUI) und steuert die Reaktionen des Plugins auf Diese.

Im ersten Abschnitt werden weitere Funktionen definiert, welche das entsprechende Verhalten der GUI-Elemente für das TAL-file steuern. Diese Funktionen werden durch Benutzereingaben im Plugin getriggert. Dies wird später näher erläutert.

Zuerst wird beschrieben, was passieren soll, wenn die Combobox für die Layerselektion benutzt wird:

Falls nach der Nutzung der Combobox durch den Anwender kein Layer ausgewählt ist, werden automatisch die beiden Buttons zum Matching der Attribute (*„by name"* & *„manually"*) unchecked und zum „Anklicken" deaktiviert. Sollte ein Layer ausgewählt sein, werden beide Buttons zum „Anklicken" freigeschaltet.

Danach wird definiert, was passieren soll, wenn der *„by name"*-Button des TAL-files gedrückt wird:

Sollte der Button nach der Interaktion gecheckt sein, wird der *„manually"*-Button automatisch „unchecked" (falls vorher gecheckt) und die Layerselektion für Veränderungen durch den Nutzer gesperrt. Ebenfalls wird der Befehl ausgegeben zu versuchen die Spalten des aktuell selektierten Layers anhand ihrer Namen den benötigten Attributen des TAL-files zuzuweisen (*„create dict by name"*). Auch wird das TAL-file einer Liste der zu exportierenden Dateitypen zugewiesen.

Sollte der Button nach der Benutzung „unchecked" sein, wird die Layerselektion freigeschaltet und das TAL-file aus der eben genannten Liste entfernt.

Ähnliches gilt für den *„manually"*-Button:

Falls der Button nach dem „Anklicken" „gecheckt" ist, wird die Layerselektion aktiviert, beide Buttons unchecked und das TAL-file aus der Liste der zu

exportierenden Dateitypen entfernt. Dies ist auf den ersten Blick widersprüchlich, da der Button gerade gecheckt wurde um das TAL-file zum Export „anzumelden". Es ist aber nötig, da das ebenfalls nun aufgerufene zweite Dialogfenster zum manuellen zuweisen der Attribute durch den Nutzer auch über das „X" abgebrochen werden kann, ohne dass das Plugin weitere Informationen bekommt. Die notwendigen Informationen erhält das Plugin beim regulären Akzeptieren oder Zurückweisen des zweiten Dialogs (dazu in den entsprechenden Funktionen mehr).

Sollte der *„manually"*-Button gerade *„unchecked"* worden sein, wird die Layerselektion ebenfalls wieder freigeschaltet und das TAL-file aus der Exportliste entfernt.

All diese Angaben waren auf die spezifischen GUI-Elemente des TAL-files zugeschnitten, da diese mit ihren eindeutigen Namen angesteuert werden müssen. Als Nächstes müssten diese Code-Zeilen für die anderen 22 Dateitypen angepasst und ebenfalls niedergeschrieben werden. Da dies jedoch unnötig viel Platz einnehmen würde und selbst kleinste Änderungen bei allen Dateitypen wiederholt werden müssten, wurde sich für ein anderes System entschieden.

Die spezifisch für das TAL-file geschriebenen Befehle wurden (ohne die beschreibenden Kommentare) in 3 Textblöcke (*„Strings"*) aufgeteilt und jede Referenz zum TAL-file direkt durch „XXX" und jede Nennung von „tal" in einer Elementbezeichnung durch „xxx" ersetzt. Besonders ist hier darauf zu achten, die Zeilenumbrüche der Funktion durch „\n" und die für Python relevanten Einrückungen exakt nachzubilden.

Als Nächstes werden in einem Loop alle anderen Dateityp-Kürzel durchgegangen und die „XXX" bzw. „xxx" im Textblock mit dem jeweiligen Kürzel in Groß- oder Kleinbuchstaben ersetzt. Diese temporären Textblöcke werden dann mittels der *„exec"*-Methode als eigenständige Funktionen eingelesen und können danach wie die TAL-spezifischen Funktionen angesprochen werden.

Der nächste Abschnitt beschäftigt sich mit den Funktionen *„append_layer_changed()"* und *„filewidget_changed()"*. Diese sind thematisch in die größere Funktion eingebunden, verdienen aber eine eigene Beschreibung.

Als Nächstes werden sogenannte Tooltips gesetzt, welche dem Benutzer weitere Informationen zu einem GUI-Element liefern, wenn er mit der Maus länger darüber verweilt. Die meisten Tooltips wurden manuell im Programm Qt für die einzelnen Elemente eingetragen, aber für sich oft wiederholende Elemente wie die Layerselektion und die dazugehörigen Buttons wurden diese hier automatisiert gesetzt. Dabei wird, ähnlich wie bei den oben erwähnten, sich wiederholenden Befehlen, die eigentliche Auftragsbeschreibung einmal formuliert und dann in einem Loop für alle Dateitypen entsprechend angepasst, sodass das richtige Element eine jeweils passende Beschreibung erhält.

Im darauffolgenden Abschnitt „DETECT USER INPUT IN GUI" werden alle GUI-Elemente, mit denen der Benutzer interagieren kann, an die entsprechenden Funktionen angebunden, welche durch Interaktion getriggert werden sollen. Dies geschieht zum einen automatisiert für Anbindungen, die für alle 23 Dateitypen standardisiert ablaufen. Zum anderen werden einzigartige

Funktionsverbindungen (beispielsweise *„generate SYS-file*" und der Klick auf den Export-Button) einzeln definiert.

`def append_layer_changed():` [in general_gui_functions]

Diese Funktion steuert das direkte Feedback für den Benutzer bei der Funktion zur Anpassung eines Layers an einen bestimmten Dateityp.

Die Funktion wird aufgerufen, sobald der Benutzer eine der beiden Comboboxen der „Layer Adaption" verändert.

Sollte nach einer solchen Veränderung bei der einen Combobox ein Layer und in der anderen ein Dateityp ausgewählt sein, wird der dazugehörige Button freigeschaltet und mit der Bezeichnung „Append" versehen.

Falls nach einem Userinput eine der beiden Bedingungen nicht erfüllt sein sollte, wird der Button gesperrt und mit der Bezeichnung „what?" darauf hingewiesen, dass der Nutzer noch Eingaben vorzunehmen hat.

`def filewidget_changed():` [in general_gui_functions]

Hier werden die Benutzereingaben in der Spalte für den Export-Pfad geprüft und entsprechende Reaktionen getriggert.

QGIS verarbeitet Leerzeichen in Pfadangaben oft nicht korrekt. Es ist daher ratsam auf diese zu verzichten. Sollte der Benutzer den Export-Pfad nicht über das Kontext-Menü, sondern manuell eingeben bzw. verändern und dabei Leerzeichen benutzen, wird er automatisch gewarnt. Auch werden der Export-Button und alle anderen Exporte aus den Settings, die einen validen Pfad voraussetzen, gesperrt. Ebenfalls wird ein Hinweis neben der Eingabezeile und auf den einzelnen Buttons in den Settings angezeigt.

Im weiteren Verlauf der Funktion wird auch geprüft, ob überhaupt etwas eingegeben wurde (oder gerade gelöscht) und ob die Eingabe einen existierenden, validen Pfad enthält. Die Sperrung der Buttons und die Hinweise werden jedes Mal entsprechend angepasst.

`def set_tab_order():`

Diese automatisch beim Start aufgerufene Funktion ermöglicht einen sinnvollen Einsatz der Tabulatortaste zur Navigation im Plugin.

Hier wird jedes einzelne Element der grafischen Benutzeroberfläche des Plugins benannt und ein darauffolgendes Element definiert. Elemente die aktuell nicht verwendet werden können werden dabei automatisch übersprungen. Dies führt dazu, dass die Eingaben im ersten Dialogfenster komplett ohne Maus vorgenommen werden können. Das zweite Dialogfenster kann zwar ebenfalls zum größten Teil via Tabulator angesteuert werden, die einzelnen Eingabefelder für die manuelle Spaltenzuweisung im scrollbaren Feld sind so jedoch nicht erreichbar, da es hier nicht zielführend gewesen wäre.

5.6 Änderungen an bestehenden Layern

`def correct_field_names():`

Es kann vorkommen, dass QGIS beim Einlesen einer Excel-Datei die erste Zeile nicht automatisch als Spaltenüberschriften (engl. „field names") erkennt. Das Plugin bemerkt dieses Problem und schaltet einen Button im zweiten Fenster frei, der die Überschriften mit dieser Funktion korrigiert.

Zuerst wird der aktuell im zweiten Dialog-Fenster behandelte Dateityp aus einer globalen Variable importiert und dadurch die relevante Layerselection-Combobox identifiziert aus dem dann der inkorrekt eingelesene Layer entnommen wird.

Nun wird in einem Loop für alle Spalten des gewählten Layers, falls die Spaltenüberschrift mit „Field…" beginnt, Folgendes getan:

- Der Index der aktuell untersuchten Spalte wird in eine Variable gepackt.
- Der Wert der ersten Zeile dieser Spalte wird in eine weitere Variable kopiert.
- Der Name der aktuellen Spalte wird via *„renameAttribute*" mit dem Namen des Wertes aus der ersten Zeile überschrieben.

Die Bearbeitung des Layers wird danach übernommen und das zweite Dialogfenster für den aktuellen Dateityp erneut aufgerufen, damit die Veränderungen direkt sichtbar werden.

Aktuell sorgt dieses Vorgehen dafür, dass in der Datenquelle des inkorrekt eingelesenen Layers (Excel, CSV, etc.) die erste Spalte verdoppelt wird, was auch der Realität des Layers entspricht. Es wäre leicht möglich, die überzählige Version der ersten Zeile aus dem Layer bzw. der Datenquelle automatisch zu löschen – es wurde sich aber aus Prinzip dagegen entschieden Elemente der Datengrundlage zu löschen.

Während dem späteren Verarbeiten des Layers wird automatisch geprüft, ob der aktuelle Zellenwert der Überschrift der Spalte entspricht und in diesem Falle weggelassen. Die ursprünglichen Einträge, welche von QGIS nicht als Spaltenüberschriften erkannt wurden, gelangen daher nicht in die ausgegebenen BlueM-Dateien.

`def append_layer():`

Hier wird die Benutzereingabe aus der GUI zur Adaption eines bestehenden Layers verarbeitet und entsprechend weitergeleitet.

Zuerst wird der zu adaptierende Layer aus der entsprechenden Combobox ausgelesen und geprüft für welchen Dateityp (ebenfalls eine Combobox) er angepasst werden soll. Mit diesen Informationen wird dann die allgemeine Funktion zum Anpassen eines Layers gestartet und gleichzeitig dem Benutzer eine Benachrichtigung mit den Details angezeigt.

Ebenfalls wird überprüft, ob es sich beim gewählten Dateityp um *„SYS"* handelt. Falls ja, wird für den anzupassenden Layer eine Funktion (*„def change_widget_type_sys"*) gestartet, die die hilfreichen Dropdown-Menüs für den Systemplan in den Layer einfügt.

```
def append_layer_generic(layer_to_append, filetype):
```

Dies ist die allgemeine Funktion zum Adaptieren eines Layers an einen Dateityp, welche auch bei der Erzeugung der neuen GeoPackage-Layer hilft und den automatisch erzeugten SYS-Layer formatiert.

Die Funktion bekommt beim Aufrufen den zu bearbeitenden Layer und den gewünschten Dateityp mitgegeben. Mit diesem Dateityp konstruiert sich die Funktion mithilfe von *eval()* die entsprechenden Namen der globalen Variablen für die notwendige Attributliste und das Attribut-Definitions-Dictionary, welche dann in aktuelle Variablen kopiert werden.

Als Nächstes wird, in einem kombinierten Loop mit diesen beiden Variablen, ein neues Dictionary mit den *„Qvariant"*-Datentypen, für die entsprechenden Spalten (engl. Fields), gefüllt. Der Datentyp ergibt sich dabei aus dem ersten Buchstaben der jeweiligen Attribut-Definition.

Danach wird eine Liste mit den aktuell im Layer vorhandenen Spaltenüberschriften erstellt. Diese wird dann mit der Liste, der für diesen Dateityp notwendigen Spaltenüberschriften, vergleichen und als Ergebnis eine Liste mit noch hinzuzufügenden (engl. *„to append"*) Spalten erzeugt.

Nun wird für jedes Element in der „noch hinzuzufügen"-Liste im Layer eine Spalte hinzugefügt und diese entsprechend des Datentyp-Dictionary formatiert. Abschließend wird der Layer aktualisiert, um die Adaption sichtbar zu machen.

```
def change_widget_type_sys(layer):
```

Diese Funktion beinhaltet die Parameter zur Anpassung des Layers für die SYS-Datei in Bezug auf die Dropdownmenüs der Attributtabelle.

Der zu bearbeitende Layer wurde der Funktion beim Aufrufen übergeben. Nun werden hier die in Dropdownmenüs zu verändernden Spalten, sowie die Quell-Spalte für diese Menüs benannt. Die zu verändernden Spalten werden nun in einem Loop einzeln zusammen mit der Quell-Spalte an eine allgemeine Funktion zum Ausführen (*def change_widget_type*) übergeben.

```
def change_widget_type(layer, field_to_change,
field_source):
```

Dies ist die allgemein gehaltene Funktion zum Einfügen von Dropdownmenüs als Widget Type in bestehende Attributtabellen.

Die Funktion bekommt den betroffenen Layer, den Namen der zu ändernden Spalte und die Quell-Spalte der Dropdownmenüs angegeben. Der Index der zu

ändernden Spalte wird aus Ihrem Namen ermittelt. Danach wird ein Setup für den QGIS-eigenen Widget-Editor definiert. Hier ist besonders auf die Werte *„field_source"* als Quelle und die Bezeichnung *„ValueRelation"* als Synonym für Dropdownmenüs zu achten.

Am Schluss wird der Widget-Editor für den ermittelten Spaltenindex im gelieferten Layer mit dem gerade definierten Setup gestartet.

5.7 Support-Funktionen

```
def generate_sys_layer():
```

Mit dieser Funktion wird nach Klick auf den *„generate"*-Button ein temporärer Layer erzeugt, welcher alle Informationen über Systemabhängigkeiten aus für die Element-Dateien selektierten Layern bündelt und entsprechend formatiert, um daraus später die SYS-Datei zu erstellen.

Als Erstes werden die Namen der für die Systemlogik relevanten Spalten (*„BlueM_ID"* & *„Output_to"*) und eine Liste der Dateitypen, welche Systemelemente enthalten, definiert.

Nun wird in einem Loop für jede dieser Element-Dateien geprüft, ob ein Layer selektiert wurde (wenn in der Combobox nichts ausgewählt ist, wäre der entsprechende Index „-1"). Falls ein Layer selektiert wurde, wird dieser einer Liste von Element-Layern hinzugefügt.

Als Nächstes wird geprüft, ob diese Element-Layer-Liste leer ist; falls ja, wird ein Hinweis für den Benutzer angezeigt, dass keine Daten gesammelt werden konnten, da keine Layer gefunden wurden (ganz am Ende der Funktion unter *„else:"*).

Falls jedoch Layer mit Daten gefunden werden konnten, wird ein leeres Dictionary erzeugt und eine Liste mit unpassenden BlueM_ID-Einträgen (beispielsweise Leerzeichen oder „None") definiert. Danach wird in einem Loop für jeden Element-Layer geprüft, ob es sich bei dem Eintrag wirklich um einen Layer handelt (könnte sonst auch *„nonType"* sein). Falls es sich um einen Layer handelt, wird geprüft, ob die Spalten BlueM_ID und Output_to existieren.

Falls auch diese Bedingung erfüllt wird, werden alle Elemente des Layers auf ihre BlueM_ID geprüft und (sofern diese nicht in der Liste der unpassenden Einträge verzeichnet sind) samt ihrer Output_to-Einträge in das Dictionary aufgenommen. Zusätzlich wird automatisch ein zwingend vorhandenes Element Zielpegel („ZPG") eingefügt.

Im nächsten Abschnitt („CREATE A NEW LAYER") wird der Name für den neuen temporären Layer definiert und geprüft, ob ein Solcher schon im Projekt vorhanden ist. Falls ein solcher Layer aus einer früheren Operation vorhanden

ist, wird Dieser gelöscht, bevor ein neuer temporärer NoGeometry Layer erzeugt wird.

Auf Diesen werden dann die bereits bekannten Funktionen *„append_layer_generic"* und *„change_widget_type_sys"* angewendet um ihn für die SYS-Daten passend zu formatieren.

Der Abschnitt „FILL THAT LAYER WITH FEATURES" befasst sich mit dem Füllen des Layers mit den zuvor gesammelten Elementen. Zuerst wird für jedes Element im *„BlueM_ID: Output_to"*- Dictionary der *„Output_to"*-Wert untersucht, gegebenenfalls in bis zu 3 valide Elemente (als Liste) aufgeteilt und der alte Wert im Dictionary damit überschrieben. Die einzelnen Elemente des *„Output_to"*-Wertes können in der Datenquelle durch Leerzeichen, Kommata oder Semikola getrennt sein.

Im nächsten Schritt wird dann in einem verschachtelten Loop für jedes SYS-Element der *„Output_to"*-Wert jedes anderen Elementes untersucht. Wenn das aktuelle SYS-Element als *„Output_to"* eines anderen Elementes gefunden wird, ist dieses andere Element ein Zulauf des ursprünglichen Elementes und wird in einer speziellen Zulauf-Liste für Dieses gespeichert. Die Zulauf-Liste wird dann auf maximal 3 Einträge gekürzt, da das SYS-file nur 3 Spalten für Zuläufe zur Verfügung stellt.

Als Nächstes wird in einem Loop für jedes Systemelement aus dem Dictionary ein Feature erzeugt und mit den im neuen SYS-Layer vorhandenen Spalten ausgestattet. Diese Spalten werden nun mit der Element-ID und den 3 Zuläufen und 3 Abläufen aus den jeweiligen Dictionaries gefüllt. Das neue Feature wird dann einer Feature-Liste hinzugefügt, welche im nächsten Schritt mittels *„edit"* in den neuen SYS-Layer eingefügt wird.

Der abschließende Bereich „HOUSEKEEPING" wählt den neuen Layer in der Combobox für das SYS-file aus und simuliert einen Klick auf den *„by name"*-Button mit allen dazugehörigen Folgen. Abschließend wird dem Nutzer noch eine Nachricht zum aktuellen Vorgang gezeigt.

```
def open_explorer_window():
```

Diese Funktion öffnet ein Windows-Fenster zum Anzeigen des aktuellen Export-Ziel-Ordners.

Zuerst wird geprüft, ob (*„if"*) diese Funktion via Haken im Settings-Tab des Plugins aktiviert ist. Falls ja, wird der Export-Pfad aus dem Widget ausgelesen und über das Modul *„os.system"* ein Windows-Fenster mit diesem Pfad geöffnet.

```
def get_filetype_index(filetype):
```

Ermittelt den Index des mitgesendeten Dateityps im globalen Array der Datei-Informationen (*„inputfiles_overview"*).

Zuerst werden die Variablen für Spalten-Index (-1) und Dateityp-Index gesetzt. Letzterer muss bereits außerhalb des If-Statements gesetzt werden, da die IDE

ansonsten eine Warnung generiert, weil es sich nicht sicher ist, dass die Variable definiert wird, bevor sie später verwendet wird.

Nun wird in einem Loop für jede Zeile (row) des Arrays mit den Datei-Informationen („*inputfiles_overview*") in der Spalte mit den Kurzbezeichnungen („*name_short*") geprüft, ob diese dem gesuchten Dateityp entspricht. Es ist wichtig, dass nur in dieser Spalte gesucht wird, da an anderen Stellen des Info-Arrays ebenfalls Dateitypen-Bezeichnungen stehen können.

Sobald der gesuchte Dateityp gefunden wurde, wird der mitzählende Spalten-Index als Index des Dateityps gesetzt. Alternativ wäre hier auch ein Loop mit „*enumerate*" möglich gewesen.

Abschließend wird der ermittelte Index des Dateityps mit „*return*" zurückgegeben.

`def get_type_info(filetype):`

Diese Funktion sammelt alle relevanten Informationen über einen Dateityp aus der Benutzeroberfläche und dem Informations-Array, bearbeitet diese und gibt sie an die eigentlichen Konstruktions-Funktionen zurück.

Als Erstes werden der als Datenquelle zugewiesene Layer und der Index des gewählten Dateityps bestimmt. Auch wird die Anzahl der Attribute des Dateityps ermittelt und daraus eine Liste mit zweistelligen Zahlen erzeugt.

Nun werden folgende Strings (Textelemente) mithilfe des ermittelten Zeilen-Index aus dem Informations-Array extrahiert, beschnitten und formatiert:

- pattern → Muster für das Füllen der Datenzeilen
- headlines → Tabellenüberschriften
- add_lines_[1 bis 7] → Zwischenüberschriften
- last_line → Tabellenabschluss

Am Ende werden diese Informationen mit „*return*" an die Konstruktions-Funktionen zurückgegeben.

`def check_val_rep_char():`

Hier wird das in den Settings vom Benutzer eingegebene Ersatz-Zeichen für unpassende Werte aus den Layern geprüft und gegebenenfalls ersetzt.

Als Erstes wird das Ersatz-Zeichen aus dem Eingabefeld in den Settings ausgelesen und mit einer Liste von akzeptablen Zeichen aus der Funktion „*prepare_plugin() / Housekeeping*" verglichen. Falls das Zeichen nicht in dieser Liste erscheint, wird es durch ein Leerzeichen („ ") ersetzt.

Das geprüfte Zeichen (bzw. eventuell das Leerzeichen) wird nun als globale Variable gesetzt und der Inhalt des Eingabefelds in den Settings überschrieben.

```
def construct_filename(filename):
```

Mit dieser Funktion wird der spätere Name einer zu erzeugenden BlueM-Datei anhand des Dateityps und der vom Benutzer gewünschten Informationen konstruiert.

Zuerst werden der Ziel-Pfad und (falls vorhanden) der Projektname aus den Nutzereingaben in Variablen umgewandelt. Weiterhin werden Variablen mit dem Dateityp, dem Datenquell-Layer und dem aktuellen Datum samt Uhrzeit gesetzt und verständlich formatiert. Datum und Uhrzeit werden in zwei Variablen gespeichert - eine für den Dateinamen und eine für die spätere Nutzung in der Datei selbst.

Die eigentliche Konstruktion beginnt nun mit einem leeren String. Diesem werden, falls die entsprechenden Buttons in den Settings aktiviert sind, der Projektname (falls Feld nicht leer), Dateityp, Name des Datenquell-Layers und Zeitpunkt des Exports jeweils mit verbindendem Unterstrich („_ ") hinzugefügt.

Der konstruierte Dateiname wird nun überprüft:

- Falls kein Projektname gesetzt wurde, wird der führende Verbindungs-Unterstrich entfernt.
- Sollte der Projektname selbst ein Unterstrich sein (vom Anwender so gewünscht), wird der zweite/überflüssige Unterstrich entfernt.
- Für den Fall, dass kein Projektname gesetzt und auch sonst keine Informationen aktiviert wurden, wird der Dateiname als *„unnamed"* definiert.
- Sollte der Name zusammen mit dem Pfad der Datei länger als 250 Zeichen werden, so wird der Dateiname auf diese Länge gekürzt. Zwar kann Windows 10 Dateinamen mit mehr als den früher möglichen 255 Zeichen verarbeiten (250, da Extension noch fehlt), jedoch wäre die Nutzung mit älteren Systemen nicht mehr möglich.

Abschließend wird dem Dateinamen (inklusive Pfad) die Dateinamenserweiterung angefügt und dann zusammen mit einem Zeitstempel an die eigentliche Konstruktions-Funktion zurückgegeben.

Die Benennungsoptionen sind mit Vorsicht zu genießen, da alle Eingangsdaten in der aktuellen BlueM-Version denselben Namen haben müssen und nur anhand der Dateinamenserweiterung unterschieden werden.

```
def clear_dlg():
```

Diese Funktion löscht alle bisherigen Nutzereingaben aus dem aktuellen Plugin-Dialog und setzt Diesen auf den Standard zurück.

Zuerst wird die globale Variable, welche alle aktuell zu exportierenden Dateitypen als Liste enthält geleert.

Daraufhin wird in einem Loop für alle Dateitypen via *„eval"*-Funktion Folgendes vorgenommen:

- Die entsprechende Combobox zur Auswahl eines Daten-Layers wird freigeschaltet (war vorher nach Auswahl der Sortierungsmethode gesperrt) und auf den Index 0 gesetzt, welcher leer ist.
- Der *„by-name"*-Button des jeweiligen Dateityps wird deaktiviert und auch gesperrt, da aktuell kein Layer ausgewählt ist.
- Der *„manually"*-Button wird ebenfalls deaktiviert und gesperrt.

5.8 Code zur Anbindung an standard Dateien

Code in create_bluem_input_files.py

Um den automatisch generierten Code des Plugin-Builders sauber von dem im Rahmen dieser Abschlussarbeit geschriebenen Codes zu trennen, wurde Letzterer in die *„plugin-functions.py"*-Datei ausgelagert.

Damit der Basis-Code des Plugin-Builders, der die standardisierte Verbindung zu QGIS herstellt, auf den zu bewertenden Code der Abschlussarbeit zugreifen kann, muss folgende Verbindung in die Basis-Datei „create_bluem_input_files.py" eingefügt werden:

- In der Import-Sektion: *from .plugin_functions import **
  → Importiert alles („*") aus *„plugin-functions.py"*

- In der *„def run(self)"*-Funktion: *prepare_plugin(self, self.first_start)*
  → Startet die importierte Funktion "prepare_plugin", welche alle weiteren notwendigen Funktionen aufruft

Code in create_bluem_input_files_dialog.py

Standardmäßig enthält die durch den Plugin-Builder erzeugte Datei „create_bluem_input_files_dialog.py" nur eine Klasse (*„class"*) für eine einzige Qt-UI-Datei.

Da das vorliegende Plugin jedoch zwei grundverschiedene Benutzeroberflächen verwendet (1 für Layerauswahl & Settings; sowie 1, welches an die 23 Dateitypen für die manuelle Auswahl angepasst wird), ist es notwendig in dieser Datei eine zweite Klasse zu erstellen.

Der verwendete Code kann dazu von der ersten Klasse (und der *„FORM_CLASS"*) kopiert werden, es muss jedoch an vielen Stellen darauf geachtet werden, die entsprechenden Dateinamen und Variablen abzuändern.

6. Fazit & Ausblick

Wie bereits vor über 10 Jahren durch Liebscher & Mendel (2010) [16] festgestellt, sind Geographische Informationssysteme (wie QGIS) unverzichtbar, um die - im Vergleich zu empirischen Modellen sehr umfangreichen - nötigen Daten zum Modellieren komplexer Flusseinzugsgebiete zu erarbeiten.

Gleichzeitig wird die bisher schon hohe Relevanz von hydrologischen Modellen zur Unterstützung von wichtigen politischen & wirtschaftlichen Entscheidungsprozessen, im Zuge der durch den Klimawandel zunehmenden Starkregenereignisse [17] [18], nur weiter steigen.

Das im Rahmen dieser Abschlussarbeit entwickelte Interface bietet den Anwendern von BlueM dabei einen schnellen und komfortablen Weg, um die mit QGIS erarbeiteten Daten in eine von BlueM nutzbare Form zu bringen.

Zusätzlich können auch bereits als CSV-Datei oder in ähnlichen Formaten vorliegende Daten für hydrologische Modelle per Drag-and-Drop in QGIS eingefügt und von dort mit wenigen Klicks BlueM-konform ausgegeben werden. Wobei jeder einzelne Wert automatisch vorgeprüft wird und somit spätere Fehler in BlueM und entsprechende manuelle Prüfungen der Eingangsdaten reduziert werden.

Diese Automatisierung einfacher und repetitiver Aufgaben, deren manuelle Durchführung enormen Zeitaufwand bedeuten würde und gleichzeitig fehleranfälliger wäre, ist ein bedeutender Faktor bei der Wahl eines hydrologischen Modells (Stichwort *„Usability"*).

Das Interface wird BlueM daher einen Vorteil gegenüber anderen Modellen verschaffen, die nicht über eine solche Schnittstelle verfügen.

Das Plugin in seiner aktuellen Version 1.0 (vom 14.02.2022) stellt jedoch lediglich einen Grundstein dar.

Es ist geplant, das Plugin auf *GitHub* (einer Website für Software-Entwicklungsprojekte) zu veröffentlichen und dort offen weiterzuentwickeln (analog zu BlueM selbst).

Langfristig ist dabei angedacht, die Funktionalitäten des Plugins deutlich über die BlueM-konforme Ausgabe vorhandener Daten und das automatische Generieren des Systemplans (SYS-Datei) zu erweitern.

Als nächster großer Schritt erscheint dabei die automatische Erzeugung von Hydrological Response Units (HRUs) logisch. Diese ist bereits seit Längerem für das Programm SWAT via dessen QGIS Interface QSWAT3 möglich [19].

[16] Liebscher, Hans-Jürgen und Mendel, Hermann Gregor. 2010
[17] Singh, Shailesh Kumar und Marcy, Nelly. 2017
[18] Lozán, José L. 2020
[19] Dile, Yihun, Srinivasa, R. und George, Chris. 2020

Eine Roadmap für die Berechnung der HRUs und der daraus möglichen Gewinnung der Daten für die BlueM-Dateitypen EFL und EZG existiert bereits mit der in Kapitel 2 erwähnten Bachelorthesis von Fabian Schweizer (2021).

Insgesamt bietet das Plugin in seinem jetzigen Zustand eine schnelle und komfortable Bearbeitung von Eingangsdaten für BlueM - und sollte daher schnellstmöglich den Anwendern von QGIS und BlueM zur Verfügung gestellt werden. Hierfür bietet sich eine Veröffentlichung im QGIS Plugin Repository an.

Literaturverzeichnis

Bach, Michael und Kissel, Michael. 2017. *BlueM - Ein Softwarepaket zur integrierten Flusseinzugsgebietsmodellierung.* [Präsentation] s.l. : Sydro Consult / ihwb, Technische Universität Darmstadt, 2017.

Bach, Michael, et al. 2009. *BlueM - Ein Softwarepaket zur integrierten Flussgebietsbewirtschaftung.* s.l. : ihwb, Technische Universität Darmstadt, 2009.

BlueM Developer Group. 2022. BlueM Website. [Online] 2022. www.bluemodel.org.

—. 2022. BlueM Wiki. [Online] 2022. www.wiki.bluemodel.org.

Devi, Gayathri K, Ganasri, B P und Dwarakish, G S. 2015. A Review on Hydrological Models. *Aquatic Procedia.* 1001 – 1007, 2015. INTERNATIONAL CONFERENCE ON WATER RESOURCES, COASTAL AND OCEAN ENGINEERING (ICWRCOE 2015).

Dile, Yihun, Srinivasa, R. und George, Chris. 2020. *QGIS 3 Interface for SWAT (QSWAT3).* 2020.

Dixon, Barnali und Uddameri, Venkatesh. 2016. *GIS and Geocomputation for Water Resource Science and Engineering.* ISBN 978-1-118-35413-1. s.l. : John Wiley & Sons, Ltd, 2016.

Junghänel, T., et al. 2021. *Hydro-klimatologische Einordnung der Stark- und Dauerniederschläge in Teilen Deutschlands im Zusammenhang mit dem Tiefdruckgebiet „Bernd" vom 12. bis 19. Juli 2021.* s.l. : Deutscher Wetterdienst, 2021.

Kaffka, Ines. 2017. Die Wasser-Kathedrale - Tokyos Taifunschutz. *Der Spiegel - www.spiegel.de/wissenschaft/technik/christoffer-rudquist-taifunschutz-in-tokio-a-1162726.html.* 2017.

Kissel, Michael, et al. 2017. *Stand und Entwicklung des Integrierten Flusseinzugsgebietsmodells BlueM.* s.l. : ihbw, Technische Universität Darmstadt; BlueM Developer Group, 2017.

Liebscher, Hans-Jürgen und Mendel, Hermann Gregor. 2010. *Vom empirischen Modellansatz zum komplexen hydrologischen Flussgebietsmodell.* Koblenz : Bundesanstalt für Gewässerkunde, 2010.

Lozán, José L. 2020. Der Klimawandel und die Entwicklung der Niederschläge. *Warnsignal Klima.* 2020, https://www.klima-warnsignale.uni-hamburg.de/der-klimawandel-und-die-entwicklung-der-niederschlaege/.

Michaelson, Greg. 2020. Programming Paradigms, Turing Completeness and Computational Thinking. *The Art, Science, and Engineering of Programming.* Vol. 4, 2020, No. 3.

Pendergrass, Angeline G. und Knutti, Reto. 2018. The Uneven Nature of Daily Precipitation and Its Change. *Geophysical Research Letters (American Geophysical Union).* 2018, 10.1029/2018GL080298.

QGIS-Community. 2022. QGIS-Website. [Online] 2022. [Zitat vom: 20. 01 2022.] www.qgis.org.

Rosskopf, Tobias. 2016. *Implementierung von ausgewählten Elementen zur Modellierung der Abflussquantität und -qualität urbaner Entwässerungsstrukturen in das Softwarepaket BlueM.* s.l. : Masterthesis - Technische Universität Darmstadt, 2016.

Schweizer, Fabian. 2021. *Eignung von QGIS für die Erstellung von Eingangsdaten für das Modellsystem BlueM.* Stuttgart : Hochschule für Technik Stuttgart, 2021.

Singh, Shailesh Kumar und Marcy, Nelly. 2017. Comparison of Simple and Complex Hydrological Models for Predicting Catchment Discharge Under Climate Change. *AIMS Geosciences.* 2017, Bd. Vol. 3, Issue 3, 467-497.

Snow, John. London 1854. "The cholera near Golden-Square, and at Deptford. *The Medical Times and Gazette.* London 1854.

Sorooshian, Soroosh und Moradkhani, Hamid. 2008. *General review of rainfall-runoff modeling: model calibration, data assimilation, and uncertainty.* s.l. : Springer, 2008. ISBN 978-3-540-77842-4.

The Python Software Foundation. 2022. General Python FAQ. [Online] 2022. docs.python.org/3/faq/general.html#id2.

Trenberth, Kevin E., et al. 2006. Estimates of the Global Water Budget and Its Annual Cycle Using Observational and Model Data. [Hrsg.] American Meteorological Society. *Journal of Hydrometeorology.* 2006, #758.

Vieux, Baxter E. 2016. *Distributed Hydrologic Modeling Using GIS.* s.l. : Water Science and Technology Library - Springer, 2016.

Anhang A

Inhalt der Definitionsdatei

(16 Seiten)

Definitionsdatei (Teil 1 von 16)

name_short	name_long	comment_for_dlg2
ALL	General Simulation	Only first line of attributes will be exported.\nAny other lines are ignored.
SYS	System Plan	The system element "ZIELPEGEL (ZPG)" has to be\nthe last element in the SYS-file.
FKT	Functions	
KTR	Control functions	Control functions for the same element must be\nlisted immediately one after another.\n(BlueM Bug 342)
EXT	time series management	
JGG	Annual series	
TGG	Day series	
WGG	Week series	
BEK	Rain overflow basin	
EIN	Individual discharges	
EZG	Natural catchments	
FKA	Urban catchment	
HYA	Hydraulic Elements	
RUE	Rain Overflow Discharge	
TAL	Dams	The Identification of a dam must begin with a "T".\nThe TAL-file consists of several tables:\n1 as overview and 6 per
TRS	Transport routes	The maximum number of lines that can be entered\nper transport element is 26! (TRS_MAXSTZ)
URB	Consumers	
VER	Branching	
BOA	Soil texture	
BOD	Soil types	
EFL	Elementary surfaces	
LNZ	Landuse	
DIF	Diffuse sources	

Definitionsdatei (Teil 2 von 16)

<table>
<tr><th>name_short</th><th>...</th><th>pattern</th><th>attr_count</th></tr>
<tr><td>ALL</td><td rowspan="23">Die Spalten „example“, „headlines“, „add_lines_[1 bis 7]“ und "last_line" sind zu umfangreich, um hier sinnvoll dargestellt zu werden. Diese enthalten lediglich Textblöcke, die jederzeit aus den jeweiligen BlueM-Dateien entnommen werden können – daher wurde hier auf sie verzichtet. Ebenfalls ist das „pattern“ der ALL- und der TAL-Datei unvollständig.</td><td>*Allgemeine Angaben (*.ALL)
*==========================</td><td>32</td></tr>
<tr><td>SYS</td><td>| {} | {} | {} {} {} | {} {} {} | {} | {} | {}</td><td>11</td></tr>
<tr><td>FKT</td><td>| {} |{}| {} {}| {} {} {} | {} |</td><td>8</td></tr>
<tr><td>KTR</td><td>| {} {} |{}| {} {} {} | {} | {} |{}{}{}| {} | {} | {} {} | {} |{}| {} {} |{}|{}|
{} {} | {} {} |{}{}|{}|{} {}|{}|{} {} {} |{}</td><td>35</td></tr>
<tr><td>EXT</td><td>| {} | {} | {} | {} |{}</td><td>5</td></tr>
<tr><td>JGG</td><td>| {}{}| {} {} |{}</td><td>5</td></tr>
<tr><td>TGG</td><td>| {} | {} {} {} {} {} {} {} {} {} {} {} {} |</td><td>13</td></tr>
<tr><td>WGG</td><td>| {} | {}{}{}{}{}{}{} |</td><td>8</td></tr>
<tr><td>BEK</td><td>| {} {} {} | {} | {} {} {} |{} {} {} {} {}| {} | {} | {} |</td><td>16</td></tr>
<tr><td>EIN</td><td>| {} | {} | {} {} {} {} | {} {} | {} | {} |</td><td>11</td></tr>
<tr><td>EZG</td><td>| {} | {} |{}{}{}{}{}|{}| {} {} {}| {} {} {} {} {}|{} {}|{}|{}{}|{}| {} {} {}
{} {} {} {} {}| {} {}| {} {} {} {} {} {} | {} |</td><td>40</td></tr>
<tr><td>FKA</td><td>| {} | {} |{}{}{} |{}| {} {} {}|{}|{}{}| {} {} {} {} {} {} {} {} {} {} {} |</td><td>24</td></tr>
<tr><td>HYA</td><td>| {} | {} | {} | {} {} {} {} {}|{} {} {} {} {} {} {}| {} {} {} {} {} {} {}|</td><td>23</td></tr>
<tr><td>RUE</td><td>| {} | {} | {}{} | {} | {}{} | {} | {} |</td><td>10</td></tr>
<tr><td>TAL</td><td>| {} | {} | {} | {} | {} | {} | {} | {} | {} |
| {} | {} | {} {} {} {} | | | |
| |</td><td>73</td></tr>
<tr><td>TRS</td><td>| {} | {} |{}|{}|{}{}{}{}{} | {} {}{}{} {}{} {} | {}{}{} | {} | {} {} {} {}
{}|</td><td>26</td></tr>
<tr><td>URB</td><td>| {} | {} {} {} {} {} |{} | {} {} {} {} {}{} | {} {} {} {} {} |</td><td>19</td></tr>
<tr><td>VER</td><td>| {} | {} | {}{} | {} | {}{} | {} {} {} {} {}|</td><td>13</td></tr>
<tr><td>BOA</td><td>| {} | {} {} {} {} {} {} {} | {} |</td><td>9</td></tr>
<tr><td>BOD</td><td>| {} | {} {} {} {} {} {} {} {} {} {} {} {} {}| {} |</td><td>15</td></tr>
<tr><td>EFL</td><td>| {} | {} {} {} {} {} | {} {} |{}{} {}|{}| {} |</td><td>13</td></tr>
<tr><td>LNZ</td><td>| {} |{} {} {} {} {} {} {} {}|{}|</td><td>10</td></tr>
<tr><td>DIF</td><td>| {} |{}|{}{}{}{}{}{}{}|</td><td>9</td></tr>
</table>

Definitionsdatei (Teil 3 von 16)

name_short	attr_01	attr_01_type	attr_02	attr_02_type	attr_03	attr_03_type
ALL	Bezeichnung	sssssssssssssssssssssssssssssssssssss	Hauptueberschriften	sssssssssssssssssssssssssssssssssssss	SimBeginn	TT.MM.JJJJ hh:mm
SYS	Beschreibung	sssssssssssssssssss	Nr	ssss	Zulauf_1	ssss
FKT	Bez	ssss	Funktion	sssssssssssssssssss	Fkt_Nam	sss
KTR	Bez	ssss	Bez_an	Y	KTR_Kng	sss
EXT	Datei_Nr	sss	Einh	ssss	Intrp	sss
JGG	JGG	sss	V	s	Dat	TT.MM
TGG	KT	ss	h_1	hh:mm	1	iiii
WGG	WGG	sss	Monday	fffff	Tuesday	fffff
BEK	BlueM_ID	ssss	Basin_type	sss	(H/N)	s
EIN	BlueM_ID	ssss	Kng	sss	QMITTEL	ffffffff
EZG	BlueM_ID	ssss	KNG	sss	A	ffffff
FKA	BlueM_ID	ssss	KNG	sss	EZG_A	ffffff
HYA	BlueM_ID	sss	WSP	sss	Kng	sss
RUE	BlueM_ID	ssss	Kng	sss	Schwellenwert_Qdr	ffffff
TAL	BlueM_ID	ssss	Anfangsvolumen	fffffffff	Maximalvolumen	ffffffffff
TRS	BlueM_ID	ssss	Kng	sss	Laenge	iiiiii
URB	BlueM_ID	ssss	Zuschuss_Kng	sss	Zuschuss_QM	ffffff
VER	BlueM_ID	ssss	Kng	sss	Naeherung_Qswell	ffffff
BOA	ID	iiii	Typ	i	WP	ffffff
BOD	ID	iiii	anzsch	fff	d1	ffff
EFL	EZG	ssss	Gef	ffff	Flaeche	iiiiii
LNZ	ID	iiii	WE	fffff	BG_%	fffff
DIF	ID	iiii	Abfluss	sssssss	O2(%)	fffffff

Definitionsdatei (Teil 4 von 16)

name_short	attr_04	attr_04_type	attr_05	attr_05_type	attr_06	attr_06_type
ALL	SimEnde	TT.MM.JJJJ hh:mm	Zeitschrittlaenge	ssssssssss	Zeitschritt_Monat	ssssssssss
SYS	Zulauf_2	ssss	Zulauf_3	ssss	Ablauf_1	ssss
FKT	Fkt_Art	ss	h_mNN	iiiiii	X-Wert	ffffffffff
KTR	Funkt_S	Y	Funkt_Z	Y	Funkt_A	i
EXT	Pfad	ssssssssssssssssssssssss ssssssssssssssssssssssss ssssssssssssssssssssssss ssssssss	Bezeichnung	ssssssssss		
JGG	Fak	fffff	Additional_Info	sssssssssssssss		
TGG	h_2	hh:mm	2	iiii	h_3	hh:mm
WGG	Wednesday	fffff	Thursday	fffff	Friday	fffff
BEK	Kng	i	Naeherung_Vol	ffffff	Naeherung_Qab	ffffff
EIN	JGG	sss	WGG	sss	TGG	sss
EZG	VG	ffffff	Ho	ffffff	Hu	ffffff
FKA	EZG_VG	ffffff	EZG_tf	ffff	Datei	ssssssss
HYA	Wehrform	sss	Wehrkante	fffffff	Wehrbreite	fffff
RUE	Schwellenwert_Trenn	ffffff	proz_Aufteilg	ffffffffffffff	Qzu	ffffff
TAL	Gewicht	iii	Sohle	ffffffff	HW_Entl_Kante	ffffffff
TRS	translation_time	iiiiii	Rohrleitung_P	s	Rohrleitung_B_D	fffff
URB	Zuschuss_JGG	sss	Zuschuss_WGG	sss	Zuschuss_Datei	sss
VER	Naeherung_Trenn	ffffff	Naeherung_%	ffffffffffffff	Kennlinie_Qzu	ffffff
BOA	FK	ffffff	GPV	ffffff	kf	ffffff
BOD	boa1	ffff	d2	ffff	boa2	ffff
EFL	Bod	iiii	Lnz	iiii	CN	iiii
LNZ	BG_jgg	sss	BFI	fffff	BFI_jgg	sss
DIF	T(°C)	fffffff	BSB5(mg/l)	fffffff	NH4+(mg/l)	fffffff

Definitionsdatei (Teil 5 von 16)

name_short	attr_07	attr_07_type	attr_08	attr_08_type	attr_09	attr_09_type
ALL	Muldenverlust	iiiii	Anfangsverlust	iiiii	Gerinneabfluss	iiiii
SYS	Ablauf_2	ssss	Ablauf_3	ssss	WEL	Y
FKT	Y-Wert	ffffffff	Datum	TT.MM hh:mm		
KTR	KTR_Typ	s	Sys_zust.	sss	1_KTR_Z	ss
EXT						
JGG						
TGG	3	iiii	h_4	hh:mm	4	iiii
WGG	Saturday	fffff	Sunday	fffff		
BEK	Naeherung_Qku	ffffff	Kennlinien_h	ffffff	Kennlinien_Qab	ffffff
EIN	Datei_Nr	ssssssss	Faktor(%)	fffff	Spende	ffffff
EZG	L	ffffff	N_Datei	sssssss	Kng_Verd	s
FKA	Kng_Verd	i	Sum_Verd	ffffff	Datei_Verd	sssssss
HYA	mue	ffff	Fak_hue	sss	Einl_Kante	ffffff
RUE	Qdr	ffffff	ID	sss	ZR	sss
TAL	HW_Entl_SVol	fffffffff	Krone_Kante	ffffffff	Krone_Svol	fffffffff
TRS	Rohrleitung_Aquer	fffff	Rohrleitung_Js	fffff	Rohrleitung_kb	fffff
URB	Retention	iiiii	Verbrauch_Kng	sss	Verbrauch_Schwellenwert	ffffff
VER	Kennlinie_Qab	ffffff	ID	sss	Faktor	ffffff
BOA	maxInf	ffffff	maxKap	ffffff	Beschreibung	sssssssssssssssssssssssss
BOD	d3	ffff	boa3	ffff	d4	ffff
EFL	SimOpt_Typ	i	SimOpt_mi	Y	xPos	ffffff
LNZ	LNZ_jgg	sss	VG	fff	Kst	fffff
DIF	NO3-(mg/l)	fffffff	PO34-(mg/l)	fffffff	StoffX(mg/l)	fffffff

Definitionsdatei (Teil 6 von 16)

name_short	attr_10	attr_10_type	attr_11	attr_11_type	attr_12	attr_12_type
ALL	Anfangsbecken fuellung	iiiii	Anfangsbodenf euchte	iiiii	Kontrollfunktio nen	Y
SYS	BIL	Y	Add_Descriptio n	sssssssssssssssssssssssssssssssssss		
FKT						
KTR	1_KTR_Kng	sss	1_KTR_Fak	sssss	WEL	Y
EXT						
JGG						
TGG	h_5	hh:mm	5	iiii	h_6	hh:mm
WGG						
BEK	Kennlinien_Qk u	ffffff	Kennlinien_Qb u	ffffff	Kennlinien_V	ffffff
EIN	Flaeche	ffffff	Output_to	ssss		
EZG	Sum	ffffff	Datei_Verd	sssssss	Kng_Temp	s
FKA	PSI_1	fffff	CN	fffff	VorRg	fffff
HYA	Laenge	ffffff	D	fffff	A_quer	fffff
RUE	Output_to	ssss				
TAL	0_Bez	ssss	0_Beschreibun g	sssssssssssssssssssssss	0_Funk_Anz_S pe	iii
TRS	Vorland	s	Hoehe	ffffff	Links_Breite	ffffff
URB	Verbrauch_Tre nn	fffff	Verbraucht_Au ft	fffff	Verbrauch_Qz u	ffffff
VER	JGG	sss	WGG	sss	TGG	sss
BOA						
BOD	boa4	ffff	d5	ffff	boa5	ffff
EFL	yPos	ffffff	Symbol	iiiii	Diffuse_Quelle n_ID	iiiiiii
LNZ	Beschreibung	sssssssssssssssssssssssss				
DIF						

Definitionsdatei (Teil 7 von 16)

name_short	attr_13	attr_13_type	attr_14	attr_14_type	attr_15	attr_15_type
ALL	Exponent_Abw_Sollwert	iii	Fehlerabweichung_am_Pegel	iii	Fehlerabweichung_Zielfkt	iii
SYS						
FKT						
KTR	Beschreibung	sssssssssssssssssss	Sollwert_Wert	fffffffff	Sollwert_JGG	sss
EXT						
JGG						
TGG	6	iiii				
WGG						
BEK	Abs_Kl	s	Ktr_ID	sss	Str_ZR	sss
EIN						
EZG	Temp	ffff	JGG_Temp	sss	TGG	sss
FKA	Einw	iiiii	Qh	fffff	Qh_JGG	sss
HYA	Umfang	fffff	kB	fffff	Zeta	fffff
RUE						
TAL	0_Funk_Anz_Str	iii	0_Funk_Anz_Grz	iii	0_Funk_Anz_Abh	iii
TRS	Links_Kst	fffff	Rechts_Breite	ffffff	Rechts_Kst	fffff
URB	Verbrauch_Qab	ffffff	Bedarf_Kng	sss	Bedarf_QM	ffffff
VER	Output_to	ssss				
BOA						
BOD	d6	ffff	boa6	ffff	Beschreibung	sssssssssssssssssssssssss
EFL	WEL	Y				
LNZ						
DIF						

Definitionsdatei (Teil 8 von 16)

name_short	attr_16	attr_16_type	attr_17	attr_17_type	attr_18	attr_18_type
ALL	Ausgabe_Maximalereignisse	Y	Bilanzausgabe	Y	Wahrscheinlichkeiten	Y
SYS						
FKT						
KTR	Istwert_Kng	sss	Istwert_Zschr	sssss	Istwert_fest_Zeitraum_Start	tt.mm
EXT						
JGG						
TGG						
WGG						
BEK	Output_to	ssss				
EIN						
EZG	Datei_Temp	ssssssss	qB	fffff	JGG_QBASIS	sss
FKA	Qh_WGG	sss	Qh_TGG	sss	Qg	fffff
HYA	P_Leistg	ffffff	Qmax	fffff	Qmin	fffff
RUE						
TAL	1_Speicherkennlinie_WSP	fffffff	1_Speicherkennlinie_Vol	fffffffff	1_Speicherkennlinie_Oberfl	ffffffff
TRS	offenes_Gerinne_Js	fffff	Kennlinie_Hoehe	ffffff	Kennlinie_A_quer	ffffff
URB	Bedarf_JGG	sss	Bedarf_WGG	sss	Bedarf_Datei	sss
VER						
BOA						
BOD						
EFL						
LNZ						
DIF						

Definitionsdatei (Teil 9 von 16)

name_short	attr_19	attr_19_type	attr_20	attr_20_type	attr_21	attr_21_type
ALL	Ganglinienausgabe	Y	CSV_Format	Y	Binaerausgabe	Y
SYS						
FKT						
KTR	Istwert_fest_Zeitraum_Ende	tt.mm	Istwert_Monat	sssss	Istwert_XZsch	iiiii
EXT						
JGG						
TGG						
WGG						
BEK						
EIN						
EZG	PSI_1	fffff	CN	fffff	VorRg	fffff
FKA	Qg_JGG	sss	Qg_WGG	sss	Qg_TGG	sss
HYA	Hoehe	fffffff	Verlust	fffff	%_von_Qmax	fffff
RUE						
TAL	2_Grenzf_Funktion	sssssssssssssssssss	2_Grenzf_lfd_Nr	iii	2_Grenzf_Fkt_Kng	sss
TRS	Kennlinie_Qab	ffffff	MQ	iiiiii	Temp_Kng	i
URB	Output_to	ssss				
VER						
BOA						
BOD						
EFL						
LNZ						
DIF						

Definitionsdatei (Teil 10 von 16)

name_short	attr_22	attr_22_type	attr_23	attr_23_type	attr_24	attr_24_type
ALL	Animation	Y	Berechnung	ii	Anzahl_Stoffe	ii
SYS						
FKT						
KTR	Koordinaten_SystemZF_x	iiiiiiiii	Koordinaten_SystemZF_y	iiiiiiiii	Koordinaten_KontrollGF_x	iiiiiiiii
EXT						
JGG						
TGG						
WGG						
BEK						
EIN						
EZG	BF0_3	fffff	R_Retentionskonst	Y	K_VG	ffffff
FKA	Qf	fffff	Qf_JGG	sss	Output_to	ssss
HYA	eta	fffff	Output_to	ssss		
RUE						
TAL	2_Grenzf_Min_Wert	ffffffff	2_Grenzf_Zeiger_Nr	sss	2_Grenzf_y_Pos	fffffffff
TRS	Temp_Tem	ffff	Temp_JGG	sss	Temp_TGG	sss
URB						
VER						
BOA						
BOD						
EFL						
LNZ						
DIF						

Definitionsdatei (Teil 11 von 16)

name_short	attr_25	attr_25_type	attr_26	attr_26_type	attr_27	attr_27_type
ALL	EFL_Wellenausgabe	Y	EFL_Bodenkennwerte	Y	DIF_Testausgabe	Y
SYS						
FKT						
KTR	Koordinaten_KontrollGF_y	iiiiiiiii	Achsenbeschriftung_x	sssssssssssss	Achsenbeschriftung_y	sssssssssssss
EXT						
JGG						
TGG						
WGG						
BEK						
EIN						
EZG	K1	ffffff	K2	ffffff	Int	ffffff
FKA						
HYA						
RUE						
TAL	2_Grenzf_Achsbesch_X	sssssssssssss	2_Grenzf_Achsbesch_Y	sssssssssssss	3_Sp_Oberfl_Funktion	sssssssssssssssss
TRS	Temp_Datei	sssssss	Output_to	ssss		
URB						
VER						
BOA						
BOD						
EFL						
LNZ						
DIF						

Definitionsdatei (Teil 12 von 16)

name_short	attr_28	attr_28_type	attr_29	attr_29_type	attr_30	attr_30_type
ALL	Protokollausgabe	Y	LTSTOUT	Y	BLBFFKT	Y
SYS						
FKT						
KTR	Werte_Aenderung_Kng	sss	Werte_Aenderung_Typ	sss	Werte_Aenderung_Faktor	ffffff
EXT						
JGG						
TGG						
WGG						
BEK						
EIN						
EZG	Bas	ffffff	nLin	Y	Expo	fffff
FKA						
HYA						
RUE						
TAL	3_Sp_Oberfl_Fkt_Kng	sss	3_Sp_Oberfl_Kng_1_2	sss	3_Sp_Oberfl_JGG	sss
TRS						
URB						
VER						
BOA						
BOD						
EFL						
LNZ						
DIF						

Definitionsdatei (Teil 13 von 16)

name_short	attr_31	attr_31_type	attr_32	attr_32_type	attr_33	attr_33_type
ALL	LBFOUT	Y	Beschreibung	sssssssssssssssssssssssssssssssssssssssssssss		
SYS						
FKT						
KTR	Werte_Aenderung_Bezugsdatum	TT.MM.JJJJ hh:mm	Werte_Aenderung_JGG	sss	Werte_Aenderung_WGG	sss
EXT						
JGG						
TGG						
WGG						
BEK						
EIN						
EZG	Beta1	ffffff	Beta2	ffffff	R_Schneekonst	Y
FKA						
HYA						
RUE						
TAL	3_Sp_Oberfl_WGG	sss	3_Sp_Oberfl_TGG	sss	3_Sp_Oberfl_QM	ffffff
TRS						
URB						
VER						
BOA						
BOD						
EFL						
LNZ						
DIF						

Definitionsdatei (Teil 14 von 16)

name_short	attr_34	attr_34_type	attr_35	attr_35_type	attr_36	attr_36_type
ALL						
SYS						
FKT						
KTR	Werte_Aenderung_TGG	sss	Additional_Info	sssssssssssss		
EXT						
JGG						
TGG						
WGG						
BEK						
EIN						
EZG	GrT	fffff	NSchD	fffff	GrD	fffff
FKA						
HYA						
RUE						
TAL	3_Sp_Oberfl_Datei_Nr	sss	4_Steuer_Funktion	sssssssssssssssssss	4_Steuer_gibt_nach	ssss
TRS						
URB						
VER						
BOA						
BOD						
EFL						
LNZ						
DIF						

Definitionsdatei (Teil 15 von 16)

name_short	attr_37	attr_37_type	attr_38	attr_38_type	attr_39	attr_39_type	attr_40	attr_40_type
EZG	RateT	fffffff	RateSB	fffffff	MQ	fffffffffff	Output_to	ssss
TAL	4_Steuer_KTR_ID	ssss	4_Steuer_Fkt_Kng	sss	4_Steuer_S	Y	4_Steuer_Z	Y

name_short	attr_41	attr_41_type	attr_42	attr_42_type	attr_43	attr_43_type	attr_44	attr_44_type
TAL	4_Steuer_F	s	4_Steuer_Grz	sss	4_Steuer_A	Y	4_Steuer_Kng_1_4	sss

name_short	attr_45	attr_45_type	attr_46	attr_46_type	attr_47	attr_47_type	attr_48	attr_48_type
TAL	4_Steuer_HYA_ID	sss	4_Steuer_JGG	sss	4_Steuer_WGG	sss	4_Steuer_TGG	sss

name_short	attr_49	attr_49_type	attr_50	attr_50_type	attr_51	attr_51_type	attr_52	attr_52_type
TAL	4_Steuer_QM	ffffff	4_Steuer_Datei_Nr	sss	4_Steuer_y_Pos	fffffffff	4_Steuer_Achsenbesch_X	sssssssssssssss

name_short	attr_53	attr_53_type	attr_54	attr_54_type	attr_55	attr_55_type	attr_56	attr_56_type
TAL	4_Steuer_Achsenbesch_Y	sssssssssssssss	4_Steuer_Kng_0_3	sss	4_Steuer_Intervall_Typ	sss	4_Steuer_Intervall_Faktor	ffffff

name_short	attr_57	attr_57_type	attr_58	attr_58_type	attr_59	attr_59_type	attr_60	attr_60_type
TAL	4_Steuer_Bezugsdatum	TT.MM.JJJJ hh:mm	4_Steuer_W_Aenderung_JGG	sss	4_Steuer_W_Aenderung_WGG	sss	4_Steuer_W_Aenderung_TGG	sss

Definitionsdatei (Teil 16 von 16)

name_short	attr_61	attr_61_type	attr_62	attr_62_type	attr_63	attr_63_type	attr_64	attr_64_type
TAL	5_Interne_Abh_Fkt_Kng	sss	5_Interne_Abh_QM	ffffff	5_Interne_Abh_QM_JGG	sss	5_Interne_Abh_QM_WGG	sss

name_short	attr_65	attr_65_type	attr_66	attr_66_type	attr_67	attr_67_type	attr_68	attr_68_type
TAL	5_Interne_Abh_QM_TGG	sss	5_Interne_Abh_Speicher	fffffffff	5_Interne_Abh_Speicher_JGG	sss	5_Interne_Abh_Speicher_WGG	sss

name_short	attr_69	attr_69_type	attr_70	attr_70_type	attr_71	attr_71_type	attr_72	attr_72_type
TAL	5_Interne_Abh_fkt_Nam_1	sss	5_Interne_Abh_fkt_Nam_2	sss	5_Interne_Abh_fkt_Nam_3	sss	5_Interne_Abh_fkt_Nam_4	sss

name_short	attr_73	attr_73_type	attr_74	attr_74_type	attr_75	attr_75_type	attr_76	attr_76_type
TAL	Output_to	ssss						

name_short	attr_77	attr_77_type	attr_78	attr_78_type	attr_79	attr_79_type	attr_80	attr_80_type

Anhang B

User Manual (english)

(2 Seiten)

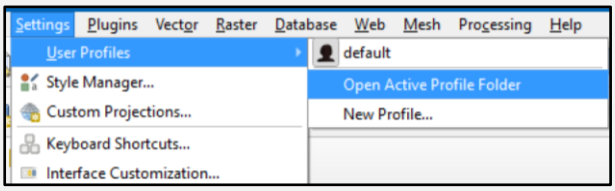
SHORT USER MANUAL
QGIS BlueM Interface Plugin

for version 1.0 (released 2022-02-14)

INSTALLATION & SYSTEM REQUIREMENTS

The plugin requires QGIS 3.16.11 (LTR) or higher; Windows 10 is recommended.
It is not yet part of the QGIS Plugin repository, therefore it must be installed manually:

- Copy folder "create_bluem_input_files".
- Open active profile folder (see right).
- Navigate in there to folder "python".
- Navigate to folder "plugins".
- Create it if it doesn't exist yet.
- Paste copied folder in there.



- Restart QGIS and go to *"Plugins"* – *"Manage and Install Plugins…"*.
- Go to tab *"Installed"* and tick the box in front of *"Create BlueM Input Files"; then "Close".*
  ➔ The plugin is now part of your QGIS toolbar (as BlueM icon).

HOW TO EXPORT BLUEM-FILES

- Open the plugin via click on the BlueM icon.
- Decide which BlueM files you want to create and select a source layer for those filetypes in the corresponding combo box (only *Vector* and *NoGeometry* possible).
- Decide how you want to match the fields of the layer attribute table to the attributes needed for the BlueM file:
  - ❖ Check "by name" if the attributes should match automatically by their names
  - ❖ Check "manually" if you want to match the attributes yourself:
    This will open a second window, where you can select the layer field names for every file attribute manually (the buttons down left may help you with this).
    You can save your matches for this filetype with a click on "OK".
- Enter a valid export path in the field at the bottom and press "Export".
  ➔ The plugin will now export all files where a match-button is checked.

OTHER TOOLS

- Adapt existing layers to better suit the requirements of BlueM-files
- Or create new GeoPackage layers for that purpose
- Export attribute definitions for all BlueM-files as a CSV
- Generate a new layer with SYS-data from element-layers

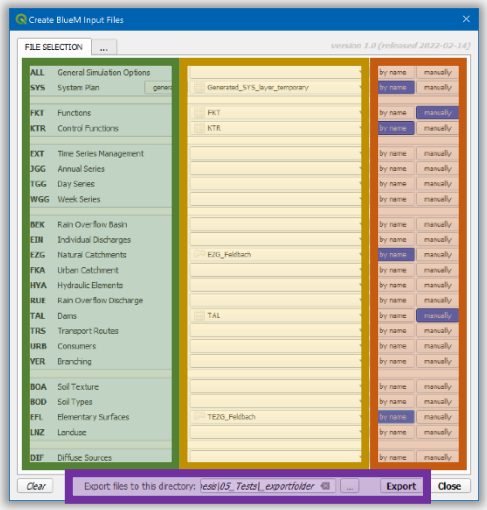
GOOD TO KNOW

- The settings in the "**…**"-tab are worth a look and self-explanatory.
- Sometimes QGIS has trouble identifying the first row of an attribute table of an Excel file as the field names (it calls them "Field1", etc.); the plugin can correct hat:
  If the plugin detects this, it shows a button in the "manually"-window to correct the issue.
  (At the moment this creates a doubling of the first row in the Excel source file.)

- If a string or float is to long for its target cell, the plugin will shorten/round them.
  If an integer is to long or the value does not match the required data type, it will be replaced with the replacement character you selected (default = " ").
  You can find information about these and other value changes in each file or the export log.

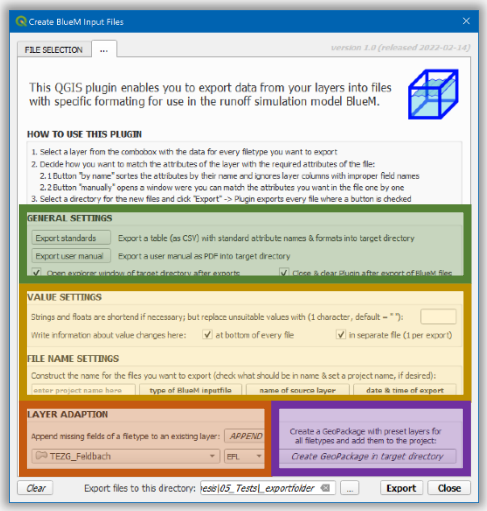
USER INTERFACE OVERVIEW

The user interface of the plugin consists of 3 parts:



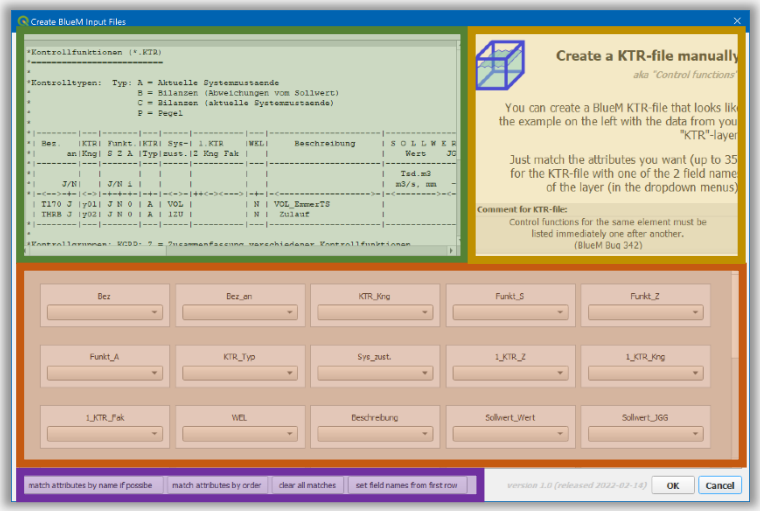
❶ Selection for BlueM Export

all 23 BlueM-files and generate-SYS-button
layer selection for all filetypes
the buttons to match their attributes
define export path and start export



❷ Settings and other tools

export standards & user manual
and general options
decide where the export information should be saved and define names for exported files
adapt an existing layer to a filetype
create a GeoPackage with layers for all filetypes



❸ Manual attribute matching

example of filetype
information about file & layer
select matches one by one
helpful tools

Anhang C

Code der Datei *plugin_functions.py*

(43 Seiten)

```python
# -*- coding: utf-8 -*-
"""
##########################################################################
##                                                                      ##
##  This file contains the specialized code to enable the functions of the ##
##  QGIS BlueM Interface Plugin    (Masterthesis Martin Großhaus Feb 2022) ##
##                                                                      ##
##########################################################################

   ____   _____  _____   _____
  / __ \ / ____||_   _| / ____|
 | |  | || |  __  | |  | (___    ______
 | |  | || | |_ | | |   \___ \  |______|
 | |__| || |__| |_| |_  ____) |
  \___\_\ \_____||_____||_____/

  _____         _                __
 |_   _|       | |              / _|
   | |   _ __  | |_   ___  _ __ | |_   __ _   ___   ___
   | |  | '_ \ | __| / _ \| '__||  _| / _` | / __| / _ \
  _| |_ | | | || |_ |  __/| |   | |  | (_| || (__ |  __/
 |_____||_| |_| \__| \___||_|   |_|   \__,_| \___| \___|

   __                 ____   _                __  __
  / _|               |  _ \ | |              |  \/  |
 | |_   ___   _ __   | |_) || | _   _   ___  | \  / |
 |  _| / _ \ | '__|  |  _ < | || | | | / _ \ | |\/| |
 | |  | (_) || |     | |_) || || |_| ||  __/ | |  | |
 |_|   \___/ |_|     |____/ |_| \__,_| \___| |_|  |_|

#------------------------------------------------------------------------#
|                                                                        |
|   Connect this file to the general code of the QGIS PluginBuilder by:  |
|                                                                        |
|   1) Copy this into the import section of "create_bluem_input_files.py":  |
|       from .plugin_functions import *                                  |
|                                                                        |
|   2) Replace the code IN the function "def run(self)" (same file) with:  |
|       prepare_plugin(self, self.first_start)                           |
|                                                                        |
|   3) In file "create_bluem_input_files_dialog.py":                     |
|       double the code for class & FORM_CLASS; "...Dialog" for "...Dialog2"|
|                                                                        |
#------------------------------------------------------------------------#

"""

##########################################################################


# IMPORTS

import csv  # to import csv data
import numpy as np  # for arrays
import os  # to verify file paths, open explorer etc.
import math  # for rounding up
import shutil  # to copy pdf files (manual)

# for date and time
from datetime import datetime

# used to change alignment of labels in warning
#  (only used in eval, therefore called "unused" by IDE)
from PyQt5.QtCore import Qt

from qgis import processing

# used for validation of text and setting type of new layer fields
from PyQt5.QtCore import QRegExp, QVariant
from PyQt5.QtGui import QRegExpValidator

# IDE (e.g. PyCharm) calls imports below unresolved, but QGIS knows them
from qgis.core import QgsMapLayer, QgsMapLayerProxyModel, QgsFeature, \
    QgsSettings, QgsField, QgsVectorLayer, \
    QgsEditorWidgetSetup, QgsProject, edit
```

```
# import dialog windows
from .create_bluem_input_files_dialog import CreateBlueMInputFilesDialog
from .create_bluem_input_files_dialog import CreateBlueMInputFilesDialog2


################################################################################


# DEFINE GLOBAL VARIABLES

# globals for array and its indices
global inputfiles_overview
global index_name_short
global index_name_long
global index_comment_for_dlg2
global index_attr_count
global index_example
global index_headlines
global index_add_lines_1
global index_add_lines_2
global index_add_lines_3
global index_add_lines_4
global index_add_lines_5
global index_add_lines_6
global index_add_lines_7
global index_last_line
global index_pattern

# global lists and dictionaries
global list_inputfile_types
global list_number_attr_file_max
global list_inputfile_types_standard
global list_filetypes_not_standard
global dict_indices_file_attr_names
global dict_indices_file_attr_types
global list_filetypes_for_export

# for current filetype in second window
global current_filetype_second_window
global current_list_file_attr_names
global current_list_number_attr_file
global current_number_attr_layer
global current_number_attr_file

# other globals
global self
global max_number_attr_file
global list_valid_chars_not_space
global val_rep_char


################################################################################
#   GENERAL FUNCTIONS  -  Allgemeine Funktionen (Kapitel 5.1)                  #
################################################################################


# This function prepares the first window and general plugin functions
def prepare_plugin(add_self, first_start):

    # set self for the following functions as a global, so that it does not
    # need to be passed on all the time (easier for eval()-functions)
    global self
    self = add_self

    # define dialog windows
    self.dlg = CreateBlueMInputFilesDialog()
    self.dlg2 = CreateBlueMInputFilesDialog2()

    # show the dialog
    self.dlg.show()

    # set info about plugin version in both dialog windows
    version_info = "version 1.0 (released 2022-02-14) "
    self.dlg.lb_version_info.setText(version_info)
    self.dlg2.lb_version_info.setText(version_info)

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
```

```
    # EXECUTION

    # import information about BlueM input file types from csv into array
    with open(os.path.dirname(  file  ) + str("\\")
              + "inputfiles_overview.csv", 'r', encoding="ANSI") as file:
        inputfiles_overview_data = list(csv.reader(file, delimiter=";"))

    # put all information into array
    global inputfiles_overview
    inputfiles_overview = np.array(inputfiles_overview_data)

    # check for correct indices of information in the array
    #  in case the columns got mixed up
    global index_name_short
    index_name_short = np.where(inputfiles_overview == "name_short")[1]
    global index_name_long
    index_name_long = np.where(inputfiles_overview == "name_long")[1]

    global index_comment_for_dlg2
    index_comment_for_dlg2 = \
        np.where(inputfiles_overview == "comment_for_dlg2")[1]
    global index_attr_count
    index_attr_count = np.where(inputfiles_overview == "attr_count")[1]
    global index_example
    index_example = np.where(inputfiles_overview == "example")[1]

    global index_headlines
    index_headlines = np.where(inputfiles_overview == "headlines")[1]
    global index_add_lines_1
    index_add_lines_1 = np.where(inputfiles_overview == "add_lines_1")[1]
    global index_add_lines_2
    index_add_lines_2 = np.where(inputfiles_overview == "add_lines_2")[1]
    global index_add_lines_3
    index_add_lines_3 = np.where(inputfiles_overview == "add_lines_3")[1]
    global index_add_lines_4
    index_add_lines_4 = np.where(inputfiles_overview == "add_lines_4")[1]
    global index_add_lines_5
    index_add_lines_5 = np.where(inputfiles_overview == "add_lines_5")[1]
    global index_add_lines_6
    index_add_lines_6 = np.where(inputfiles_overview == "add_lines_6")[1]
    global index_add_lines_7
    index_add_lines_7 = np.where(inputfiles_overview == "add_lines_7")[1]
    global index_last_line
    index_last_line = np.where(inputfiles_overview == "last_line")[1]

    global index_pattern
    index_pattern = np.where(inputfiles_overview == "pattern")[1]

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

    # get a global list of input file types
    global list_inputfile_types
    list_inputfile_types = []
    for row in inputfiles_overview:
        # only in this column
        list_inputfile_types.append(row[int(index_name_short)])
    # remove header
    list_inputfile_types.remove("name_short")
    # remove empty entry if necessary
    while '' in list_inputfile_types:
        list_inputfile_types.remove('')

    # set a maximum number of attributes for all filetypes and make it a list
    global max_number_attr_file
    max_number_attr_file = 80  # limited by number of frames in current GUI
    global list_number_attr_file_max
    list_number_attr_file_max = []
    for i in range(1, max_number_attr_file + 1):
        if len(str(i)) == 1:
            i = "0" + str(i)
        list_number_attr_file_max.append(str(i))

    # define a global dictionary of the indices of all file attributes
    global dict_indices_file_attr_names
    dict_indices_file_attr_names = {}
    # fill a global dictionary of the indices of all file attributes
    for i in list_number_attr_file_max:
        dict_indices_file_attr_names[str("index_attr_" + i)] = \
```

```
        np.where(inputfiles_overview == str("attr_" + i))[1]

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# create global list for every filetype with its attribute names
for i in list_inputfile_types:  # go through every filetype
    current_list = []  # make temporary variable for filetype
    # go through all attribute numbers of current filetype
    for j in range(1, int(inputfiles_overview
                          # raise by 1 because its from list (no header)
                          [int(list_inputfile_types.index(i) + 1)]
                          [int(index_attr_count)])
                          # raise by 1 because of how "range" works
                          + 1):
        if len(str(j)) == 1:  # give j a leading zero if necessary
            j = "0" + str(j)
        current_list.append(str(inputfiles_overview
                                # raise by 1 because its from list
                                # (with no header)
                                [int(list_inputfile_types.index(i) + 1)]
                                [int(dict_indices_file_attr_names
                                     ["index_attr_" + str(j)])]))

    # create global list for every filetype from temporary variable
    globals()[i.lower() + "_file_attr_list"] = current_list

# get a dictionary of the indices of all file attributes types
global dict_indices_file_attr_types
dict_indices_file_attr_types = {}
for i in list_number_attr_file_max:
    dict_indices_file_attr_types[str("index_attr_" + i + "_type")] = \
        np.where(inputfiles_overview == str("attr_" + i + "_type"))[1]

# create global dictionary for every filetype with its attribute types
for i in list_inputfile_types:  # go through every filetype
    current_dict = {}  # make temporary variable for filetype
    # go through all attribute numbers of current filetype
    for j in range(1, int(inputfiles_overview
                          # raise by 1 because its from list (no header)
                          [int(list_inputfile_types.index(i) + 1)]
                          [int(index_attr_count)]
                          # raise by 1 because of how "range" works
                          ) + 1):
        if len(str(j)) == 1:  # give j a leading zero if necessary
            j = "0" + str(j)
        current_dict["attr_" + str(j) + "_type"] = \
            str(inputfiles_overview
                # raise by 1 because its from list (no header)
                [int(list_inputfile_types.index(i) + 1)]
                [int(dict_indices_file_attr_types
                     ["index_attr_" + str(j) + "_type"])])

    # create global dictionary for every filetype from temporary variable
    globals()[i.lower() + "_file_attr_type_dict"] = current_dict

# Control input validation of line edit in second tab of first window.
# Only allow: standard letters (no "ä" etc), numbers and "_"
validator = QRegExpValidator(QRegExp("[A-Z-a-z-0-9_]+"))
self.dlg.le_project_name.setValidator(validator)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# LAYER ADAPTION (in second tab of first dialog)

# sort layer-combobox (remove unwanted layer types)
suitable_types_layer_adaption = (QgsMapLayerProxyModel.VectorLayer
                                 or QgsMapLayerProxyModel.NoGeometry)
self.dlg.cb_layer_adaption_selection \
    .setFilters(suitable_types_layer_adaption)

# fill filetype selection combobox with empty item and all filetypes
self.dlg.cb_filetype_combobox.addItem(str(""))
self.dlg.cb_filetype_combobox.addItems(list_inputfile_types)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# GENERAL HOUSEKEEPING
```

```
    # create empty list of files for export
    global list_filetypes_for_export
    list_filetypes_for_export = []

    # set a global list of valid characters, that are not a whitespace (" ")
    global list_valid_chars_not_space
    list_valid_chars_not_space = \
        ['A', 'a', 'B', 'b', 'C', 'c', 'D', 'd', 'E', 'e', 'F', 'f',
         'G', 'g', 'H', 'h', 'I', 'i', 'J', 'j', 'K', 'k', 'L', 'l',
         'M', 'm', 'N', 'n', 'O', 'o', 'P', 'p', 'Q', 'q', 'R', 'r',
         'S', 's', 'T', 't', 'U', 'u', 'V', 'v', 'W', 'w', 'X', 'x',
         'Y', 'y', 'Z', 'z',
         'Ä', 'ä', 'Ö', 'ö', 'Ü', 'ü', 'ß', '_', '!', '¿', '?', '&',
         '%', '$', '€', '#', '+', '-', '*', '/', '~', '§',
         '0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

    # a list of filetypes that are currently not working
    list_filetypes_not_working = []  # non, juhu! :-)

    # disable functions for filetypes that are not working
    for i in list_inputfile_types:
        if i in list_filetypes_not_working:
            # disable this combobox and set a warning
            eval("self.dlg.cb_xxx_layerselection.setEnabled(False)"
                 .replace("xxx", i.lower()))
            eval("self.dlg.lb_xxx_eng.setText('NOT WORKING')"
                 .replace("xxx", i.lower()))
            eval("self.dlg.lb_xxx_eng.setAlignment(Qt.AlignRight)"
                 .replace("xxx", i.lower()))

    # get a list of "not standard" filetypes
    global list_filetypes_not_standard
    list_filetypes_not_standard = [
        "TAL"  # has its own export function
    ]

    # get a global list of "standard" filetypes by removing non standard
    #  filetypes from list of all filetypes
    global list_inputfile_types_standard
    list_inputfile_types_standard = list_inputfile_types.copy()
    for i in list_filetypes_not_standard:
        list_inputfile_types_standard.remove(i.upper())

    # check the value replacement character
    check_val_rep_char()

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

    # GUI CONNECTIONS

    # establish GUI connections only once due to "first_start"-variable
    if first_start is True:
        first_start = False  # set first_start to FALSE for next round

        general_gui_functions()
        set_tab_order()


################################################################################


# Creates a second window and fills it with the necessary information
#  for the selected input file type
def open_second_window(filetype):

    # differentiate between a and an depending on first letter
    #  of the file type name (Vowels must be in a tuple)
    vowels = ("A", "E", "I", "O", "U", "a", "e", "i", "o", "u")
    if filetype.startswith(vowels):
        a_an = "an"
    else:
        a_an = "a"

    # get filetype index from function
    filetype_index = get_filetype_index(filetype)

    # get the example for the filetype from the array of csv data
    self.dlg2.lb_example.setText(
```

```
        str(inputfiles_overview[int(filetype_index)][int(index_example)]))

    # set tool tip for example
    self.dlg2.lb_example.setToolTip(
        "example of " + a_an + " " + filetype.upper() + "-file")

    # get the selected layer
    selected_layer = \
        eval("self.dlg.cb_xxx_layerselection.currentLayer()"
             .replace("xxx", str(filetype.lower())))

    # get the name of the selected layer
    selected_layer_name = \
        eval("self.dlg.cb_xxx_layerselection.currentText()"
             .replace("xxx", str(filetype.lower())))

    # get the long name of the filetype
    name_long = str(inputfiles_overview[int(filetype_index)]
                                       [int(index_name_long)])

    # get the comment for the filetype and format it
    comment = str(inputfiles_overview[int(filetype_index)]
                                     [int(index_comment_for_dlg2)])
    comment = comment.replace(r"\n", "\n")

    # get the number of attributes for the filetype
    number_attr_filetype = int(inputfiles_overview[int(filetype_index)]
                                                  [int(index_attr_count)])

    # get list of attributes of the layer
    layer_attr = []
    for field in selected_layer.fields():
        layer_attr.append(field.name())

    # get the number of attributes of the layer
    number_attr_layer = len(layer_attr)

    # create list of numbers of attributes for this filetype
    list_number_attr_file = []
    for i in range(1, number_attr_filetype+1):
        # give i a leading zero if necessary
        if len(str(i)) == 1:
            i = "0"+str(i)
        list_number_attr_file.append(str(i))

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
    # ATTRIBUTE FRAMES

    # make all attribute frames visible for the moment
    for i in list_number_attr_file_max:
        eval("self.dlg2.frame_attr_XX.show()".replace("XX", i))

    # get max number of row frames (5 attributes per row frame; round UP)
    max_number_row_frames = math.ceil(max_number_attr_file / 5)
    # make all attribute row frames invisible for the moment
    for i in range(1, max_number_row_frames + 1):
        eval("self.dlg2.frame_row_XX.hide()".replace("XX", str(i)))

    # make a list of all superfluous attribute numbers
    superfluous_attr_numbers = []
    for i in list_number_attr_file_max:
        if i not in list_number_attr_file:
            superfluous_attr_numbers.append(i)

    # hide all superfluous attribute frames
    for i in superfluous_attr_numbers:
        eval("self.dlg2.frame_attr_XX.hide()".replace("XX", i))

    # get needed number of row frames (5 attributes per row frame; round UP)
    needed_number_row_frames = math.ceil(len(list_number_attr_file) / 5)
    # make needed attribute row frames visible
    for i in range(1, needed_number_row_frames + 1):
        eval("self.dlg2.frame_row_XX.show()".replace("XX", str(i)))

    # clear all comboboxes before filling them new when a new version
    #  of the second window is opened
    for i in list_number_attr_file_max:
        # iterates through a list of all comboboxes (max)
```

```
        eval("self.dlg2.cb_attr_XX.clear()".replace("XX", i))

    # list of file attribute names for current filetype
    current file attr list = eval("xxx file attr list"
                                  .replace("xxx", filetype.lower()))

    # fill the labels above the comboboxes with the file attribute names
    for (i, j) in zip(list_number_attr_file, current_file_attr_list):
        eval("self.dlg2.lb attr XX.setText(j)".replace("XX", i))

    # fill comboboxes with layer attributes
    for i in list_number_attr_file:
        # empty item at the top
        eval("self.dlg2.cb_attr_XX.addItem(str(""))".replace("XX", i))
        # fill from list
        eval("self.dlg2.cb_attr_XX.addItems(layer_attr)".replace("XX", i))

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
    # SET TEXTS

    # set header in second window for the selected filetype
    self.dlg2.lb_header.setText(
        "Create " + a_an + " " + filetype.upper() + "-file manually")

    # set long name of filetype in second window
    self.dlg2.lb_name_long.setText(
        'aka "' + name_long + '" ')

    # set info_1 in second window for the selected filetype
    self.dlg2.lb_info_1.setText(
        "You can create a BlueM " + filetype.upper() +
        "-file that looks like\nthe example on the left with the "
        "data from your\n" + '"' + selected_layer_name + '"' + "-layer.")

    # set info_2 in second window for the selected filetype
    self.dlg2.lb_info_2.setText(
        "Just match the attributes you want (up to "
        + str(number_attr_filetype) + ")\n for the "
        + filetype.upper() + "-file with one of the "
        + str(number_attr_layer)
        + " field names\nof the layer (in the dropdown menus).")

    # set title for comment box
    self.dlg2.gb_comment.setTitle(
        " Comment for " + filetype.upper() + "-file:")

    # set comment
    self.dlg2.lb_comment.setText(comment)

    # hide comment box if comment is empty, and show if not empty
    if comment == "":
        self.dlg2.gb_comment.hide()
    else:
        self.dlg2.gb_comment.show()

    # handle button for "correct field names"
    self.dlg2.pb_correct_field_names.hide()
    # only show button if necessary (field name 1 = "Field1")
    if self.dlg2.cb_attr_01.itemText(1) == "Field1":
        self.dlg2.pb_correct_field_names.show()

    # open second window
    self.dlg2.show()

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

    # set global variables accordingly for use in other functions
    #  (e.g. correcting field names of faulty layer, etc.)

    # set current filetype of second window as global
    global current filetype second window
    current_filetype_second_window = filetype

    # set current list of attribute numbers of file as global
    global current_list_number_attr_file
    current_list_number_attr_file = list_number_attr_file

    # set current list of attribute names of file as global
```

```
    global current_list_file_attr_names
    current_list_file_attr_names = current_file_attr_list

    # set number of layer attributes of second window as global
    global current_number_attr_layer
    current_number_attr_layer = number_attr_layer

    # set number of file attributes of second window as global
    global current number attr file
    current_number_attr_file = number_attr_filetype


##############################################################################


# Function for "Cancel"-button in second window
def reject_second_window():
    # get filetype from open_second_window via global variable
    filetype = current_filetype_second_window

    # enables the layerselection, because "manually" is not checked
    eval("self.dlg.cb_xxx_layerselection.setEnabled(True)"
         .replace("xxx", str(filetype.lower())))


##############################################################################


# Function for "OK"-button in second window
def execute_second_window():

    # get filetype from open_second_window via global variable
    filetype = current_filetype_second_window

    # checks the "manually"-button, which had opened the second window
    eval("self.dlg.pb_xxx_manually.setChecked(True)"
         .replace("xxx", str(filetype.lower())))

    # put filename in list for export
    list_filetypes_for_export.append(filetype.upper())

    # disables the layerselection for this filetype so that it
    #  can not be changed unless the buttons are unchecked
    eval("self.dlg.cb_xxx_layerselection.setEnabled(False)"
         .replace("xxx", str(filetype.lower())))

    # saves the selections from all attributes of the second window
    #  in a file specific global dictionary
    exec("global xxx_attr_dict; xxx_attr_dict = {}"
         .replace("xxx", filetype.lower()))
    for xx in current_list_number_attr_file:
        eval("xxx_attr_dict"
             .replace("xxx", filetype.lower()))[str("attr_"+xx)] = \
            eval("self.dlg2.cb_attr_"+xx).currentText()


##############################################################################
#   SORTING FUNCTIONS  -  Sortier-Funktionen (Kapitel 5.2)                   #
##############################################################################


# Create a dictionary for this file type and match the layer fields to the
#  required file attributes by their names
def create_dict_by_name(filetype):

    # get filetype index from function
    filetype_index = get_filetype_index(filetype)

    # get the number of attributes for the filetype
    number_attr_filetype = \
        int(inputfiles overview[int(filetype index)]
                               [int(index_attr_count)])

    # create list of numbers of attributes for this filetype
    list_number_attr_file = []
    for i in range(1, number_attr_filetype + 1):
        if len(str(i)) == 1:  # give i a leading zero if necessary
            i = "0" + str(i)
```

```
        list_number_attr_file.append(str(i))

    # get list of file attribute names from global variable
    list names attr file = eval(filetype.lower()+" file attr list")

    # get the selected layer
    selected_layer = \
        eval("self.dlg.cb_xxx_layerselection.currentLayer()"
             .replace("xxx", str(filetype.lower())))

    # get list of layer attribute names
    list_names_attr_layer = []
    for field in selected_layer.fields():
        list_names_attr_layer.append(field.name())

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
    # CREATE NEW DICT

    # create current dictionary and fill it with all the necessary
    #  keys (with empty values) for this filetype
    current_dict = {}
    for number in list_number_attr_file:
        current_dict["attr_" + number] = ""

    # go through all attribute names of layer
    for name_attr_layer in list_names_attr_layer:
        # check if these names are used in the file
        # (make everything temporary upper case, to ignore case)
        if name_attr_layer.upper() in \
                [x.upper() for x in list_names_attr_file]:

            # get the index of that name in file attribute list
            # (make everything temporary upper case, to ignore case)
            index_attr_in_file = \
                [x.upper() for x in list_names_attr_file].index(
                    name_attr_layer.upper())

            # raise by one because the index comes from
            #  a list (starts at 0) but we start at "01"
            index_attr_in_file += 1

            # make it a string and add a leading zero if necessary
            index_attr_in_file = str(index_attr_in_file)
            if len(index_attr_in_file) == 1:
                index_attr_in_file = "0" + index_attr_in_file

            # write name of the layer attribute with the correct key in the
            #  current dictionary
            current_dict["attr_" + index_attr_in_file] = str(name_attr_layer)

    # transfer data from current dict into specific global dict
    globals()[filetype.lower() + "_attr_dict"] = current_dict


################################################################################


# Matches attributes in second window by name if possible
def match_attributes_by_name_if_possible():

    # for all attribute numbers and names of current file in second window
    for xx, yy in zip(current_list_number_attr_file,
                      current_list_file_attr_names):
        # try to find matching attribute name in every combobox
        index = eval("self.dlg2.cb_attr_xx.findText('yy', "
                     "Qt.MatchFixedString)"
                     .replace("xx", xx).replace("yy", yy))
        # if attribute name is not in combobox -> index is -1, therefore:
        if index != -1:
            # set index of combobox to the index of the found attribute name
            eval("self.dlg2.cb_attr_xx.setCurrentIndex(index)"
                 .replace("xx", xx))


################################################################################


# Matches attributes in order of fields of layer
```

```
def match_attributes_by_order():

    # for all fields in layer for filetype in second window
    for xx in range(1, current number attr layer + 1):
        # copy number, because one needs leading zero, the other does not
        xx = str(xx)
        yy = xx
        # give leading zero if necessary
        if len(str(xx)) < 2:
            yy = str("0" + yy)
        # set index of combobox to the number of layer attribute name
        eval("self.dlg2.cb_attr_yy.setCurrentIndex(int(xx))"
             .replace("yy", yy).replace("xx", xx))


#############################################################################


# Clears all attribute matches in second window
def clear_all_matches():

    # for all attribute numbers of current file in second window
    for xx in current_list_number_attr_file:
        # set index of combobox to 0 i.e. first value (empty)
        eval("self.dlg2.cb_attr_xx.setCurrentIndex(0)".replace("xx", xx))


#############################################################################
#   GENERATE & EXPORT BLUEM-FILES                                           #
#       Erzeugen & Export von BlueM-Dateien (Kapitel 5.3)                   #
#############################################################################


# When the user clicks on "export"
def export_clicked():

    # check how many files need to be exported
    number_files_to_export = len(list_filetypes_for_export)

    if number_files_to_export == 0:

        # show message in message bar
        self.iface.messageBar().pushMessage(
            "NO FILES FOR EXPORT SELECTED",
            "Please check the buttons to export BlueM files.",
            duration=10)

    else:

        # show message in message bar
        self.iface.messageBar().pushMessage(
            "Exporting BlueM files: " + str(number_files_to_export) +
            " - Types: " + str(list_filetypes_for_export))

        # open explorer window if checked in settings (GUI)
        open_explorer_window()

        # investigate which buttons are checked
        #  and call corresponding functions
        for i in list_inputfile_types_standard:
            if eval("self.dlg.pb_xxx_manually.isChecked()"
                    .replace("xxx", i.lower())):
                export_file(i)
            if eval("self.dlg.pb_xxx_byname.isChecked()"
                    .replace("xxx", i.lower())):
                export_file(i)

        # check if TAL has to be exported
        if self.dlg.pb_tal_manually.isChecked():
            export_tal_file()
        if self.dlg.pb tal byname.isChecked():
            export_tal_file()

        # export file with information about all file exports if
        #  checked in GUI
        if self.dlg.cb_separate_file.checkState():
            export_file_export_information()
```

```
        # close plugin dialog after export
        #  only if checked in settings (GUI, dlg, tab2)
        if self.dlg.cb_close_after_export.isChecked():
            self.dlg.accept()


################################################################################


# Export a file for a standard filetype
def export_file(filetype):

    # get info about this file type from csv i.e. array via function
    (current_layer, list_number_attr_file,
     pattern, headlines, last line,
     add_lines_1, add_lines_2, add_lines_3,
     add_lines_4, add_lines_5, add_lines_6,
     add_lines_7) = \
        get_type_info(filetype)

    # get dictionaries for current file type
    current_dict_name = eval(filetype.lower() + "_attr_dict")
    current_dict_type = eval(filetype.lower() + "_file_attr_type_dict")

    # get information about dictionaries
    number_of_attr_in_file = len(current_dict_name)
    number_of_matches = 0
    list_attrs_not_matched = []
    for attr_nr in current_dict_name.keys():
        if current_dict_name[attr_nr] == "":
            list_attrs_not_matched.append(attr_nr)
        else:
            number_of_matches += 1

    # get file name from function
    filename, time_of_export = construct_filename(filetype)

    # create empty list for the value warnings of this file
    current_file_value_warnings_list = []

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
    # WRITE THE FILE

    # "write" file with "ANSI" encoding
    with open(filename, 'w', encoding='ANSI') as file:

        # write headlines in file
        file.write(headlines)

        feature_counter = 0
        # go through every feature of layer
        for feature in current_layer.getFeatures():
            feature_counter += 1

            # create empty list for current row
            current_row = []
            # go through every attribute of filetype
            for xx in list_number_attr_file:

                # get the required attribute data type
                req_type = current_dict_type["attr_" + xx + "_type"]
                # replace possible \xa0 with real spaces
                req_type = req_type.replace(r'\xa0', " ")
                # get required length of output
                req length = len(str(req type))

                # check if there is an entry for this attribute
                if current_dict_name["attr_"+xx] == "":
                    # if no match in feature: fill with enough " "
                    current_row.append(str(" " * req_length))
                else:
                    # handle the corresponding value

                    # get the feature value for attribute from layer
                    value = feature[current_dict_name["attr_" + xx]]

                    # check the value in another function
                    #  and get potential warning
```

```
            value, value_warning = \
                check_value(value, req_length, req_type)

            # put value as string in list for current feature
            current_row.append(str(value))

            # if there is a value warning
            if value_warning != "":
                # add column and row to value warning
                value_warning = str(xx + ";" +
                                    str(feature_counter)
                                    + ";" + value_warning)

                # add the warning to the list for this file
                current file value warnings list\
                    .append(value_warning)

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
    # IMPLEMENT FILETYPE SPECIFIC FORMATS

    if filetype == "FKT":
        # FKT has 3 types of lines to make the table
        #  easier to read

        # line above new "Bez"
        if current_row[0][0] != " ":
            file.write(add_lines_1 + "\n")

        # line above new "Funktion"
        if current_row[0][0] == " " \
                and current_row[1][0] != " ":
            file.write(add_lines_2 + "\n")

        # line above new "Fkt_Nam"
        if current_row[0][0] == " " \
                and current_row[1][0] == " " \
                and current_row[2][0] != " ":
            file.write(add_lines_3 + "\n")

    if filetype == "KTR":
        # KTR is divided into two tables

        if current_row[0][0] == "K":
            # if "Bez" starts with "K" like in
            #  KGRP (Kontrollgruppen), insert new
            #  table header for it only once

            try:
                if kgrp_count in locals():
                    pass
            # KGRP_count should not exist at first and even the
            #  check provokes an error, therefore:
            except UnboundLocalError:
                kgrp_count = 1
                # Create KGRP_count variable, so next is "pass".
                # Could have been solved differently (define
                # variable earlier), but in this way all KTR-
                #  specific code is contained here.
                file.write(add_lines_1 + "\n")
                # Hopefully the table was sorted by the user,
                #  so that there is only KGRP in the second table.
                # The table should not be sorted by python, because
                #  that would cause more trouble than its worth due
                #  to the empty values in the first column.

    if filetype == "ALL" and feature_counter > 1:
        pass
        # only one feature / "row" is needed for an ALL-file
    else:
        # precede with writing row in file

        # check if current row contains field names
        #  (maybe check every attribute, could be valid though)
        # only if first attribute has a match
        if current_dict_name["attr_01"] != "":
            # is original first entry of first row identical
            #  to first field name?
            if current_dict_name["attr_01"] == \
```

```
                feature[current_dict_name["attr_01"]]:
                    # do not write row in file (it is field names)
                    # reduce feature counter
                    feature_counter -= 1
                else:
                    # write list for current feature with
                    #  file-specific pattern in file
                    file.write(pattern.format(*current_row))
            else:
                # write list for current feature with
                #  file-specific pattern in file
                file.write(pattern.format(*current_row))

    # write last line(s) of the table into the file
    file.write(last_line)

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
    # GET THE EXPORT INFORMATION INTO THE FILE

    # construct the export information for this file
    current_file_export_info = \
        str("* This " + filetype.upper() + "-file was created by "
            "the 'QGIS BlueM Interface Plugin'\n"
            "*   with data from the '" + current_layer.name() +
            "'-layer\n*   on the " + time_of_export + ".\n*\n" +
            "* Of the " + str(number_of_attr_in_file) +
            " file attributes, " + str(number_of_matches) +
            " found a matching field name in the layer.\n")

    # add another line to element-files
    if filetype in ["BEK", "EIN", "EZG", "FKA", "HYA",
                    "RUE", "TAL", "TRS", "URB", "VER"]:
        current_file_export_info = \
            str(current_file_export_info +
                "* (including the not printed 'Output_to')\n")

    # if some attributes did not find a match
    if number_of_attr_in_file > number_of_matches:
        current_file_export_info = \
            str(current_file_export_info +
                "* These attributes found no match: "
                + str(list_attrs_not_matched) + "\n")

    # if there are value warnings
    nr_warnings = len(current_file_value_warnings_list)

    if nr_warnings == 0:
        current_file_export_info = \
            str(current_file_export_info + "*\n* " +
                "No values had to be changed or were not "
                "suitable at all.\n*\n")

    if nr_warnings > 0:
        if nr_warnings == 1:
            current_file_export_info = \
                str(current_file_export_info + "*\n* " +
                    "1 value had to be changed or was not "
                    "suitable at all.\n")
        if nr_warnings > 1:
            current_file_export_info = \
                str(current_file_export_info + "*\n* " +
                    str(len(current_file_value_warnings_list)) +
                    " values had to be changed or were not "
                    "suitable at all:\n*\n")

        # write value warnings in file
        for warnings in current_file_value_warnings_list:
            warnings = warnings.split(";")
            current_file_export_info = \
                str(current_file_export_info +
                    "*  -> For attribute number "
                    + str(warnings[0]) + " of feature "
                    + str(warnings[1]).zfill(3) + ":  "
                    + str(warnings[2]) + "\n")

    # write information into file if checked in GUI
    if self.dlg.cb_bottom_every_file.checkState():
        file.write("\n*\n*\n*\n*\n*\n*\n"
```

```
                        + current_file_export_info)

    # set current file export info as global for filetype
    globals()[filetype.lower() + " file export info"] = \
        current_file_export_info


################################################################################


# Check if value is possible in output data type and reformat if necessary
def check_value(value, req_length, req_type):

    # create empty "value warning" for changes
    value warning = ""

    # get data type of attribute output
    if str(req_type)[0] == "s":
        attr_type = "string"
    elif str(req_type)[0] == "i":
        attr_type = "integer"
    elif str(req_type)[0] == "f":
        attr_type = "float"
    elif str(req_type)[0] == "Y":
        attr_type = "Y"  # yes(J) or no(N)
    elif str(req_type) == "TT.MM":
        attr_type = "TT.MM"
    elif str(req_type) == "hh:mm":
        attr_type = "hh:mm"
    elif str(req_type) == "TT.MM hh:mm":
        attr_type = "TT.MM hh:mm"
    elif str(req_type) == "TT.MM.JJJJ hh:mm":
        attr_type = "TT.MM.JJJJ hh:mm"
    else:
        attr_type = ""

    # "try" if conversion is possible
    try:
        # handle string
        if attr_type == "string":
            value = str(value)
            # crop both ends of string
            value = value.lstrip("['").rstrip("']")
            value = value.lstrip(" ").rstrip(" ")
            value = value.lstrip("(").rstrip(")")
            # remove "NULL" and "None"
            value = value.replace("NULL", "    ")
            value = value.replace("None", "    ")
            # cut value to appropriate length if necessary
            if len(value) > req_length:
                value = value[0:req_length]
                value_warning = "    string was shortened"
            # fill up with " " if string too short
            while len(value) < req_length:
                value = value + " "

        # in value replace "," with "." for numbers
        if attr_type == "integer" or attr_type == "float":
            value = str(value).replace(",", ".")

        # handle integer
        if attr_type == "integer":
            # accept floats and make them integer
            value = float(value)
            # need to round, because int() just cuts off
            value_r = round(value, 0)
            value_r = int(value_r)
            # check if float was rounded to integer
            if value != value_r:
                value_warning = "    float was rounded to work as integer"

            value = str(value_r)
            # check if number is too big
            if len(value) > req_length:
                value = str(val_rep_char * req_length)
                value_warning = "!!! integer was to long to fit"
            # fill up with " " if integer is too small
            while len(value) < req_length:
```

```
            value = " " + value

        # handle float
        if attr_type == "float":
            value = float(value)
            # check length before "."
            len_int = len(str(int(value)))
            # if integer part of float to big: cut it out
            if len_int > req_length:
                value = str(val_rep_char * req_length)
                value_warning = \
                    "!!! integer part of float was to long to fit"
            # if "." would be last character
            elif len_int == (req_length - 1):
                value = str(" " + str(int(round(value, 0))))
                value_warning = "    float was changed to integer"
            # if value has to many characters after "."
            elif len(str(value)) > req_length:
                value_warning = "    float rounded to fit"
                # get max character count after point
                len_after_point = req_length - 1 - len_int
                # round appropriately
                value = round(value, len_after_point)
                value = str(value)
                # convert to integer if last char ".0"
                if value[-2:] == ".0":
                    value = value[:-2]

            value = str(value)
            # fill up with " " if float is too small
            while len(value) < req_length:
                value = " " + value

        # handle YES or NO output
        if attr_type == "Y":
            value = str(value)
            # check for different versions
            yes_list = ["J", "j", "Ja", "JA", "ja",
                        "Y", "y", "Yes", "YES", "Yes",
                        "a", "oui", "etiam", "sic", "да"]
            no_list = ["N", "n", "Nein", "NEIN", "nein",
                       "No", "NO", "no",
                       "r", "non", "нет"]
            if value in yes_list:
                value = "J"
            elif value in no_list:
                value = "N"
            else:
                value = str(val_rep_char * req_length)
                # does not accept " " as replacement... therefore:
                #  fill up with " " if value is too short
                while len(value) < req_length:
                    value = " " + value
                value_warning = "!!! Y/N value was not recognized"

        # handle date and time outputs
        datetime_list = ["TT.MM.JJJJ hh:mm",
                         "TT.MM hh:mm",
                         "TT.MM",
                         "hh:mm"]
        if attr_type in datetime_list:
            value = str(value)
            # crop both ends of string
            value = value.lstrip("['").rstrip("']")
            value = value.lstrip(" ").rstrip(" ")
            # if value entry is empty: fill up
            if value == "" or \
                    value == "NULL" or \
                    value == "None" or \
                    value is None:
                value = str(" " * req_length)

            if len(value) > 16:
                # -> the value automatically turned into
                #  a QDateTime-format, e.g.: value =
                #  "PyQt5.QtCore.QDateTime(2021, 11, 1, 8, 15)"

                # now reformat into something useful
```

```
                value = value.lstrip("PyQt5.QtCore.QDateTime(")
                value = value.rstrip(")")
                value = value.replace(" ", "")
                value = value.split(",")

                year = int(value[0])
                month = int(value[1])
                day = int(value[2])
                hour = int(value[3])
                minute = int(value[4])

                # format every string as required
                if attr_type == "TT.MM.JJJJ hh:mm":
                    value = "{:02d}.{:02d}.{:04d} {:02d}:{:02d}" \
                        .format(day, month, year, hour, minute)
                if attr_type == "TT.MM hh:mm":
                    value = "{:02d}.{:02d} {:02d}:{:02d}" \
                        .format(day, month, hour, minute)
                if attr_type == "TT.MM":
                    value = "{:02d}.{:02d}".format(day, month)
                if attr_type == "hh:mm":
                    value = "{:02d}:{:02d}".format(hour, minute)

            else:
                value = str(value)
                # sadly use of parser not possible
                #  due to unrecognizable format "TT.MM"

                # check if length is suitable
                if len(value) != req_length:
                    value = str(val_rep_char * req_length)
                    value_warning = "!!! date/time value was not recognized"

            # does not accept " " as replacement... therefore:
            #  fill up with " " if value is too short
            while len(value) < req_length:
                value = " " + value

        # should not happen, if csv is complete
        if attr_type == "":
            value = str(val_rep_char * req_length)
            value_warning = "!!! value type not recognized"

    # if not possible to convert to required data type:
    except ValueError:
        # if value entry was empty all along: fill up
        if value == "" or \
                value == "NULL" or \
                value is None:
            value = str(" " * req_length)
        # if entry is somehow unsuitable: write replacement character
        else:
            value = str(val_rep_char * req_length)
            value_warning = "!!! value did not match required data type"

    # give it back
    return value, value_warning


################################################################################


# Checks if TAL-file can be exported correctly
#  (This could have been done in main TAL function, but make it too "wide".)
def export_tal_file():

    # set name of filetype
    filetype = "TAL"

    # get current layer from specific combobox
    current_layer = \
        eval("self.dlg.cb_xxx_layerselection.currentLayer()"
             .replace("xxx", filetype.lower()))

    # get name dictionary for current file type
    current_dict_name = eval(filetype.lower() + "_attr_dict")

    # check if main attribute field name is defined
```

```
    if current_dict_name["attr_01"] != "":

        # get field name of layer for first / main file attribute
        main attr field name = current dict name["attr 01"]

        # get a list of main attributes ("Talsperren"/"Dams") in layer
        main_attr_list_with_duplicates = []
        # go through all features of current layer
        for feature in current layer.getFeatures():
            main_attr_list_with_duplicates.\
                append(feature[main_attr_field_name])

        # get list without duplicates
        main_attr_list = []
        for n in main attr list with duplicates:
            if n not in main_attr_list:
                # only "T"-element
                if str(n).startswith("T"):
                    main_attr_list.append(n)

        # check if main attribute list contains any T-functions
        if len(main_attr_list) != 0:
            # raise "export possible"-function for TAL
            export_tal_file_possible()

        else:
            # raise "export failed"-function for TAL with "reason"
            export_tal_file_failed("T-FUNCTION NAME FAIL")

    else:
        # raise "failed" function for TAL with "reason"
        export_tal_file_failed("DICTIONARY FAIL")


##############################################################################


# Export of TAL-file failed
def export_tal_file_failed(reason):

    # differentiate the reasons for failure
    reason_long = []
    if reason == "DICTIONARY FAIL":
        reason_long = "The TAL-file could not be exported, " \
                      "because no layer field name was matched " \
                      "to the main attribute ('ID')."

    if reason == "T-FUNCTION NAME FAIL":
        reason_long = "The TAL-file could not be exported, " \
                      "because no T-functions ('TXXX') could " \
                      "be found in assigned column of source layer."

    # remove exporting message from message bar
    self.iface.messageBar().popWidget()

    # send a message to user
    self.iface.messageBar().pushWarning(reason, reason_long)

    # get export path
    export_path = self.dlg.fw_export_path.filePath()
    export_path = export_path.rstrip(str("\\"))

    # construct filename
    filename = export_path + "\\" + "_TAL_export_failed" + ".tal"

    # write file with warning and explanation
    with open(filename, 'w', encoding='ANSI') as file:
        file.write(reason + "\n" + reason_long)

    # set correct file export info as global for TAL
    globals()["tal file export info"] = str("* " + reason long + "\n")


##############################################################################


# Export a TAL-file, which is different from the standard filetype
#  (it has 1 main table and 6 tables for every main attribute (here: Dams))
```

```
def export_tal_file_possible():
    # (function is mostly kept universal, to enable use for other / future
    #  complicated filetypes with 1 + 6x different tables)

    # set name of filetype
    filetype = "TAL"

    # get info about this file type from csv i.e. array via function
    (current layer, list number attr file,
     pattern, headlines, last_line,
     add_lines_1, add_lines_2, add_lines_3,
     add_lines_4, add_lines_5, add_lines_6,
     add_lines_7) = \
        get_type_info(filetype)

    # get dictionaries for current file type
    current_dict_name = eval(filetype.lower() + "_attr_dict")
    current_dict_type = eval(filetype.lower() + "_file_attr_type_dict")

    # get information about dictionaries
    number_of_attr_in_file = len(current_dict_name)
    number_of_matches = 0
    list_attrs_not_matched = []
    for attr_nr in current_dict_name.keys():
        if current_dict_name[attr_nr] == "":
            list_attrs_not_matched.append(attr_nr)
        else:
            number_of_matches += 1

    # get file name from function
    filename, time_of_export = construct_filename(filetype)

    # create empty list for the value warnings of this file
    current_file_value_warnings_list = []

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
    # GET VARIABLES FOR ALL 7 SUB-TABLES
    #  (DIVIDE LAYER DATA HORIZONTALLY)

    # handle special file patterns
    pattern_list = pattern.split("\n")
    pattern_main = pattern_list[0]
    pattern_0 = pattern_list[1]
    pattern_1 = pattern_list[2]
    pattern_2 = pattern_list[3]
    pattern_3 = pattern_list[4]
    pattern_4 = pattern_list[5]
    pattern_5 = pattern_list[6]

    # get number of attributes in each pattern
    pattern_main_attr_count = pattern_main.count("{}")
    pattern_0_attr_count = pattern_0.count("{}")
    pattern_1_attr_count = pattern_1.count("{}")
    pattern_2_attr_count = pattern_2.count("{}")
    pattern_3_attr_count = pattern_3.count("{}")
    pattern_4_attr_count = pattern_4.count("{}")
    pattern_5_attr_count = pattern_5.count("{}")

    # get number ranges for attributes in each pattern
    pattern_main_attr_range = \
        range(1,
              1 + pattern_main_attr_count)

    pattern_0_attr_range = \
        range(1 + pattern main attr count,
              1 + pattern_main_attr_count + pattern_0_attr_count)

    pattern_1_attr_range = \
        range(1 + pattern_main_attr_count + pattern_0_attr_count,
              1 + pattern_main_attr_count + pattern_0_attr_count
              + pattern 1 attr count)

    pattern_2_attr_range = \
        range(1 + pattern_main_attr_count + pattern_0_attr_count
              + pattern_1_attr_count,
              1 + pattern_main_attr_count + pattern_0_attr_count
              + pattern_1_attr_count + pattern_2_attr_count)
```

```
    pattern_3_attr_range = \
        range(1 + pattern_main_attr_count + pattern_0_attr_count
              + pattern_1_attr_count + pattern_2_attr_count,
              1 + pattern main attr count + pattern 0 attr count
              + pattern_1_attr_count + pattern_2_attr_count
              + pattern_3_attr_count)

    pattern_4_attr_range = \
        range(1 + pattern main attr count + pattern 0 attr count
              + pattern_1_attr_count + pattern_2_attr_count
              + pattern_3_attr_count,
              1 + pattern_main_attr_count + pattern_0_attr_count
              + pattern_1_attr_count + pattern_2_attr_count
              + pattern_3_attr_count + pattern_4_attr_count)

    pattern_5_attr_range = \
        range(1 + pattern_main_attr_count + pattern_0_attr_count
              + pattern_1_attr_count + pattern_2_attr_count
              + pattern_3_attr_count + pattern_4_attr_count,
              1 + pattern_main_attr_count + pattern_0_attr_count
              + pattern_1_attr_count + pattern_2_attr_count
              + pattern_3_attr_count + pattern_4_attr_count
              + pattern_5_attr_count)

    # create attribute name dictionaries for specific parts
    pattern_main_attr_name_dict = {}
    pattern_0_attr_name_dict = {}
    pattern_1_attr_name_dict = {}
    pattern_2_attr_name_dict = {}
    pattern_3_attr_name_dict = {}
    pattern_4_attr_name_dict = {}
    pattern_5_attr_name_dict = {}

    # fill attribute name dictionaries for specific parts from name dict
    for count, attr_key in enumerate(current_dict_name, start=1):
        if count in pattern_main_attr_range:
            pattern_main_attr_name_dict[attr_key] = \
                current_dict_name[attr_key]
        if count in pattern_0_attr_range:
            pattern_0_attr_name_dict[attr_key] = \
                current_dict_name[attr_key]
        if count in pattern_1_attr_range:
            pattern_1_attr_name_dict[attr_key] = \
                current_dict_name[attr_key]
        if count in pattern_2_attr_range:
            pattern_2_attr_name_dict[attr_key] = \
                current_dict_name[attr_key]
        if count in pattern_3_attr_range:
            pattern_3_attr_name_dict[attr_key] = \
                current_dict_name[attr_key]
        if count in pattern_4_attr_range:
            pattern_4_attr_name_dict[attr_key] = \
                current_dict_name[attr_key]
        if count in pattern_5_attr_range:
            pattern_5_attr_name_dict[attr_key] = \
                current_dict_name[attr_key]

    # create attribute type dictionaries for specific parts
    pattern_main_attr_type_dict = {}
    pattern_0_attr_type_dict = {}
    pattern_1_attr_type_dict = {}
    pattern_2_attr_type_dict = {}
    pattern_3_attr_type_dict = {}
    pattern_4_attr_type_dict = {}
    pattern 5 attr type dict = {}

    # fill attribute type dictionaries for specific parts from name dict
    for count, attr_key in enumerate(current_dict_type, start=1):
        if count in pattern_main_attr_range:
            pattern_main_attr_type_dict[attr_key] = \
                current dict type[attr key]
        if count in pattern_0_attr_range:
            pattern_0_attr_type_dict[attr_key] = \
                current_dict_type[attr_key]
        if count in pattern_1_attr_range:
            pattern_1_attr_type_dict[attr_key] = \
                current_dict_type[attr_key]
        if count in pattern_2_attr_range:
```

```
            pattern_2_attr_type_dict[attr_key] = \
                current_dict_type[attr_key]
        if count in pattern_3_attr_range:
            pattern_3_attr_type_dict[attr_key] = \
                current_dict_type[attr_key]
        if count in pattern_4_attr_range:
            pattern_4_attr_type_dict[attr_key] = \
                current_dict_type[attr_key]
        if count in pattern_5_attr_range:
            pattern_5_attr_type_dict[attr_key] = \
                current_dict_type[attr_key]

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
    # DIVIDE LAYER FEATURES (ROWS) INTO MAIN ATTRIBUTES OF FILE
    #  (DIVIDE LAYER DATA VERTICALLY)

    # get field name of layer for first / main file attribute
    main_attr_field_name = current_dict_name["attr_01"]

    # get a list of main attributes ("Talsperren"/"Dams") in layer
    main_attr_list_with_duplicates = []
    # go through all features of current layer
    for feature in current_layer.getFeatures():
        main_attr_list_with_duplicates.append(feature[main_attr_field_name])
    # get list without duplicates
    main_attr_list = []
    for n in main_attr_list_with_duplicates:
        if n not in main_attr_list:
            # only "T"-element
            if str(n).startswith("T"):
                main_attr_list.append(n)

    # get dictionary of row indices of main attributes
    main_attr_row_index_dict = {}
    temp_list = []
    temp_previous = ""
    row_count_max = 0
    for row_count, feature in \
            enumerate(current_layer.getFeatures(), start=1):
        # check if this row has a main attribute
        if feature[main_attr_field_name] in main_attr_list:
            # check if main attribute is not in temp list (-> first time)
            if feature[main_attr_field_name] not in temp_list:
                # put main attr name in temp list, so it stays unique
                temp_list.append(feature[main_attr_field_name])
                # put main attr name with its start row index in dictionary
                main_attr_row_index_dict[
                    feature[main_attr_field_name]+"_start"] = \
                    row_count
                # it is not first main attribute, set end for previous
                if temp_previous != "":
                    main_attr_row_index_dict[
                        temp_previous + "_end"] = \
                        row_count - 1
                # set "previous" for next main attribute
                temp_previous = feature[main_attr_field_name]

        # count to max row count
        row_count_max = row_count

    # set end row index in dict for last main attribute (max row count)
    main_attr_row_index_dict[str(main_attr_list[-1]) + "_end"] =\
        row_count_max

    # get ranges for main attributes and put them in dictionary
    main_attr_ranges_dict = {}
    for attr in main_attr_list:
        main_attr_ranges_dict[attr] = \
            range(main_attr_row_index_dict[attr + "_start"],
                  main_attr_row_index_dict[attr + "_end"] + 1)

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
    # WRITE FILE

    # "write" file with "ANSI" encoding
    with open(filename, 'w', encoding='ANSI') as file:

        # write headlines in file
```

```
        file.write(headlines)

        # write pattern main table
        feature_counter = 0
        # go through every feature of layer
        for feature in current_layer.getFeatures():
            feature_counter += 1

            # create empty list for current row
            current_row = []
            # go through every attribute pattern main
            for xx in pattern_main_attr_range:
                xx = str(xx)

                # give xx leading 0 if necessary
                if len(xx) < 2:
                    xx = "0" + xx

                # get the required attribute data type
                req_type = \
                    pattern_main_attr_type_dict["attr_" + xx + "_type"]
                # replace possible \xa0 with real spaces
                req_type = req_type.replace(r'\xa0', " ")
                # get required length of output
                req_length = len(str(req_type))

                # check if there is an entry for this attribute
                if pattern_main_attr_name_dict["attr_" + xx] == "":
                    # if no match in feature: fill with enough " "
                    current_row.append(str(" " * req_length))
                else:
                    # handle the corresponding value

                    # get the feature value for attribute from layer
                    value = feature[current_dict_name["attr_" + xx]]

                    # check the value in another function
                    #  and get potential warning
                    value, value_warning = \
                        check_value(value, req_length, req_type)

                    # put value as string in list for current feature
                    current_row.append(str(value))

                    # if there is a value warning
                    if value_warning != "":
                        # add column and row to value warning
                        value_warning = str(xx + ";" +
                                            str(feature_counter)
                                            + ";" + value_warning)

                        # add the warning to the list for this file
                        current_file_value_warnings_list \
                            .append(value_warning)

            # check if current row contains more than empty space
            checkmark = 0
            for char in list_valid_chars_not_space:
                if checkmark == 0 and char in str(current_row):
                    checkmark += 1
                    # write row in file
                    file.write(pattern_main.format(*current_row) + "\n")

        # write lines at end of main table
        file.write(last line + "\n")

        # end of main table
        # ------------------------------------------------------------------ #
        # now loop for every main attribute

        for main attr in main attr list:

            # get range for this main attribute
            current_range = main_attr_ranges_dict[main_attr]

            # set current text for "Funktion für XXXX in add lines 1"
            current_add_lines_1 = add_lines_1.replace("XXXX", str(main_attr))
```

```
        # write add lines 1
        file.write(current_add_lines_1 + "\n")

        # loop for the six sub-tables
        for xxx in [0, 1, 2, 3, 4, 5]:

            # go through every feature of layer
            for feature_count, feature in \
                    enumerate(current layer.getFeatures(), start=1):

                # only use feature in current range (this main attr)
                if feature_count in current_range:

                    # create empty list for current row
                    current row = []
                    # go through every attribute of this pattern
                    for xx in eval(
                            "pattern_xxx_attr_range"
                            .replace("xxx", str(xxx))):
                        xx = str(xx)

                        # give xx leading 0 if necessary
                        if len(xx) < 2:
                            xx = "0" + xx

                        # get the required attribute data type
                        req_type = \
                            eval("pattern_xxx_attr_type_dict"
                                 "['attr_' + xx + '_type']"
                                 .replace("xxx", str(xxx)))
                        # replace possible \xa0 with real spaces
                        req_type = req_type.replace(r'\xa0', " ")
                        # get required length of output
                        req_length = len(str(req_type))

                        # check if there is an entry for this attribute
                        if eval("pattern_xxx_attr_name_dict"
                                "['attr_' + xx]"
                                .replace("xxx", str(xxx))) == "":
                            # if no match in feature:
                            #  fill with enough " "
                            current_row.append(str(" " * req_length))
                        else:
                            # handle the corresponding value

                            # get the feature value for attribute
                            #  from layer
                            value = \
                                feature[
                                    current_dict_name["attr_" + xx]]

                            # check the value in another function
                            #  and get potential warning
                            value, value_warning = \
                                check_value(
                                    value, req_length, req_type)

                            # put value as string in list for
                            #  current feature
                            current_row.append(str(value))

                    # check if current row contains more
                    #  than empty space
                    checkmark = 0
                    for char in list valid chars not space:
                        if checkmark == 0 and char in str(current_row):
                            checkmark += 1
                            # write row in file
                            file.write(eval("pattern_" + str(xxx))
                                       .format(*current_row) + "\n")

            # write matching lines at end of this table table
            #  plus 2 due to different numbering of add lines
            file.write(eval("add_lines_" + str(xxx + 2)) + "\n")

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
    # GET THE EXPORT INFORMATION INTO THE FILE
```

```
        # construct the export information for this file
        current_file_export_info = \
            str("* This " + filetype.upper() + "-file was created by "
                "the 'QGIS BlueM Interface Plugin'\n"
                "*   with data from the '" + current_layer.name() +
                "'-layer\n*   on the " + time_of_export + ".\n*\n" +
                "* Of the " + str(number_of_attr_in_file) +
                " file attributes, " + str(number_of_matches) +
                " found a matching field name in the layer.\n"
                "* (including the not printed 'Output_to')\n")

        # if some attributes did not find a match
        if number_of_attr_in_file > number_of_matches:
            current_file_export_info = \
                str(current file export info +
                    "* These attributes found no match: "
                    + str(list_attrs_not_matched) + "\n")

        # if there are value warnings
        nr_warnings = len(current_file_value_warnings_list)

        if nr_warnings == 0:
            current_file_export_info = \
                str(current_file_export_info + "*\n* " +
                    "No values had to be changed or were not "
                    "suitable at all.\n")

        if nr_warnings > 0:
            if nr_warnings == 1:
                current_file_export_info = \
                    str(current_file_export_info + "*\n* " +
                        "1 value had to be changed or was not "
                        "suitable at all:\n*\n")
            if nr_warnings > 1:
                current_file_export_info = \
                    str(current_file_export_info + "*\n* " +
                        str(len(current_file_value_warnings_list)) +
                        " values had to be changed or were not "
                        "suitable at all:\n*\n")

            # write value warnings in file
            for warnings in current_file_value_warnings_list:
                warnings = warnings.split(";")
                current_file_export_info = \
                    str(current_file_export_info +
                        "*  -> For attribute number "
                        + str(warnings[0]) + " of feature "
                        + str(warnings[1]).zfill(3) + ":  "
                        + str(warnings[2]) + "\n")

        # write information into file if checked in GUI
        if self.dlg.cb_bottom_every_file.checkState():
            file.write("*\n*\n*\n*\n*\n*\n*\n"
                       + current_file_export_info)

    # set current file export info as global for filetype
    globals()[filetype.lower() + "_file_export_info"] = \
        current_file_export_info


################################################################################
#   OTHER EXPORTS  -  Andere Exporte (Kapitel 5.4)                             #
################################################################################


# Export of user manual into target directory
def export_user_manual():

    user_manual_filename = "_USER_MANUAL_qgis_bluem_interface_plugin.pdf"
    source_path = os.path.dirname(__file__)
    target directory = self.dlg.fw export path.filePath()

    # copy user manuel from plugin files into target directory
    shutil.copy(os.path.join(source_path, user_manual_filename),
                target_directory)

    # open explorer window if checked in settings (GUI)
    open_explorer_window()
```

```
################################################################################


# Create GeoPackage with layers for all filetypes
def create_geopackage():

    # open explorer window of target directory
    open_explorer_window()

    # display warning message for the "lengthy process" in QGIS
    self.iface.messageBar().pushMessage(
        "LENGTHY PROCESS",
        "Creating 23 Layers with specific fields in a new GeoPackage is "
        "going to take a while...", duration=10)

    # repaint main QGIS window, otherwise the message would only be
    #  displayed after the lengthy and "resource-thirsty" process
    self.iface.mainWindow().repaint()

    # get export path
    export_path = self.dlg.fw_export_path.filePath()
    export_path = export_path.rstrip(str("\\"))

    # get a name for the GeoPackage (from project name if possible)
    if self.dlg.le_project_name.text() == "":
        gpkg_name = "GPKG_BlueM_Inputfiles"
    else:
        gpkg_name = str(self.dlg.le_project_name.text() + "_GPKG")

    # construct filename for the GeoPackage
    gpkg_file = str(export_path + "\\" + gpkg_name + ".gpkg")

    # for all filetypes
    for count, filetype in enumerate(list_inputfile_types, start=1):

        # create a temporary layer with polygon geometry
        if filetype in ["BEK", "EZG", "FKA", "HYA", "RUE",
                        "TAL", "TRS", "URB", "VER", "EFL", "DIF"]:
            temp_layer = QgsVectorLayer("Polygon", "temp_layer", "memory")

        # create a temporary layer with multipolygon geometry
        elif filetype in ["BOA", "BOD", "LNZ"]:
            temp_layer = QgsVectorLayer(
                "Multipolygon", "temp_layer", "memory")

        # create a temporary layer with point geometry
        elif filetype in ["EIN"]:
            temp_layer = QgsVectorLayer("Point", "temp_layer", "memory")

        # create a temporary layer without geometry
        #  (ALL, SYS, FKT, KTR, EXT, JGG, TGG, WGG)
        else:
            temp_layer = QgsVectorLayer("NoGeometry", "temp_layer", "memory")

        # add fields to the layer
        append_layer_generic(temp_layer, filetype)

        # create a name for the layer
        layer_name = filetype

        if count == 1:
            # save first layer to NEW GeoPackage (no "-update")
            parameters = {"INPUT": temp layer,
                          "OPTIONS": str("-nln " + layer_name),
                          "OUTPUT": gpkg_file}
            processing.run("gdal:convertformat", parameters)

        else:
            # save following layers to EXISTING GeoPackage (with "-update")
            parameters = {"INPUT": temp_layer,
                          "OPTIONS": str("-update -nln " + layer_name),
                          "OUTPUT": gpkg_file}
            processing.run("gdal:convertformat", parameters)

    # add the GeoPackage to the project
    self.iface.addVectorLayer(
```

```
        gpkg_file, gpkg_name, 'ogr')


##############################################################################


# Export a CSV-file with all attribute names and formats
def export_standards():

    # open explorer window if checked in settings (GUI)
    open_explorer_window()

    # get export path
    export_path = self.dlg.fw_export_path.filePath()
    export path = export path.rstrip(str("\\"))

    # construct filename
    filename = export_path + "\\" + "_STANDARDS for Plugin Input" + ".csv"

    with open(filename, 'w', encoding='ANSI') as file:

        # write headlines
        file.write("This table contains the standards regarding names and "
                   "formats of the layer attributes that can be used "
                   "in the BlueM Inputfiles Plugin for QGIS\n\n")

        # write explanations
        file.write("How to read the attribute types:\n"
                   "-> The maximum length of the attribute is the number "
                   "of the characters in its type\n"
                   "-> 's' stands for string (text)\n"
                   "-> 'f' stands for float (e.g. 12,345)\n"
                   "-> 'i' stands for integer (e.g. 12345)\n"
                   "-> 'Y' stands for a decision (e.g. Y or N, etc.)\n"
                   "-> 'TT.MM.JJJJ hh:mm' and similar are formats for date"
                   " and time values\n"
                   "       (T = day, M = month, J = year, h = hour, "
                   "m = minute)\n\n\n")

        # for every filetype
        for filetype in list_inputfile_types:

            # get list of attribute names
            list_attr_names = eval(filetype.lower() + "_file_attr_list")

            # get the number of attributes and make it a list
            number_attr = len(list_attr_names)
            list_number_attr = []
            for xx in range(1, number_attr + 1):
                xx = str(xx)
                if len(xx) < 2:
                    xx = "0" + xx
                list_number_attr.append(xx)

            # get a list of attribute types from global dict
            current_type_dict = \
                eval(filetype.lower() + "_file_attr_type_dict")
            list_attr_types = list(current_type_dict.values())

            # write a count for the attribute numbers
            current_row = str(filetype.upper() + "-file;")
            for xx in list_number_attr:
                current_row = current_row + filetype.upper() \
                              + "_attr_" + xx + ";"
            file.write(current row + "\n")

            # prepare and write attribute names in file
            current_row = str(filetype.upper() + " attr names;")
            for attr_name in list_attr_names:
                current_row = current_row + attr_name + ";"
            file.write(current row + "\n")

            # prepare and write attribute types in file
            current_row = str(filetype.upper() + " attr types;")
            for attr_type in list_attr_types:
                current_row = current_row + attr_type + ";"
            file.write(current_row + "\n\n")
```

```
################################################################################


# Export a file with information about the filetypes that were just exported
def export_file_export_information():

    # get export path
    export path = self.dlg.fw export path.filePath()
    export_path = export_path.rstrip(str("\\"))

    # get time of export
    now = datetime.now()
    # reformat suitably
    time of export = now.strftime("%Y%m%d %H%M")
    time_of_export_2 = now.strftime("%Y/%m/%d at %H:%M")

    # construct filename
    filename = export_path + "\\" + "_EXPORT_LOG_" + time_of_export + ".txt"

    # check how many files have been exported
    number_files_to_export = len(list_filetypes_for_export)

    # make it grammatically correct
    if number_files_to_export == 1:
        number_files_to_export = str("1 file that was")
    elif number_files_to_export > 1:
        number_files_to_export = str(str(number_files_to_export)
                                     + " files that were")

    # create file
    with open(filename, 'w', encoding='ANSI') as file:

        # write headlines
        file.write("* EXPORT LOG\n"
                   "* This file summarizes the export information for the "
                   + number_files_to_export + " exported on the "
                   + time_of_export_2 + ".\n*\n*\n" + 120*"*" + "\n")

        # for every exported file
        for exported_type in list_filetypes_for_export:
            # get current export info
            current_export_info = \
                eval(exported_type.lower() + "_file_export_info")

            # change wording
            current_export_info = current_export_info.replace("This", "The")

            # write info for this exported file
            file.write("*\n*\n" + current_export_info
                       + "*\n*\n" + 120*"*" + "\n")


################################################################################
#   GUI FUNCTIONS  -  GUI-Funktionen (Kapitel 5.5)                             #
################################################################################


# CONNECTIONS TO GUI (Graphical User Interface)
def general_gui_functions():

    # GUI FUNCTIONS

    # TAL (this filetype is used as an example,
    #  the others are handled in the exec-loops in the next section)

    # gui functions are marked as unused by IDE (e.g. PyCharm),
    #  but the connection is established later with eval()

    # enable buttons only if suitable layer is selected
    def tal layerselection used():
        if self.dlg.cb_tal_layerselection.currentLayer() is None:
            # uncheck both buttons if layerselection is empty
            self.dlg.pb_tal_byname.setChecked(False)
            self.dlg.pb_tal_manually.setChecked(False)
            # disable both buttons if layerselection is empty
            self.dlg.pb_tal_byname.setEnabled(False)
            self.dlg.pb_tal_manually.setEnabled(False)
```

```
        else:
            # enable both buttons if layerselection is not empty
            self.dlg.pb_tal_byname.setEnabled(True)
            self.dlg.pb tal manually.setEnabled(True)

    def tal_byname_clicked():
        if self.dlg.pb_tal_byname.isChecked():
            # uncheck other button if necessary
            self.dlg.pb tal manually.setChecked(False)
            # disable layerselection
            self.dlg.cb_tal_layerselection.setEnabled(False)
            # match attributes by name and fill dictionary
            create_dict_by_name("TAL")
            # put filename in list for export
            list filetypes for export.append("TAL")
        else:
            # enable layerselection
            self.dlg.cb_tal_layerselection.setEnabled(True)
            # remove filename from list of exports
            list_filetypes_for_export.remove("TAL")

    def tal_manually_clicked():
        if self.dlg.pb_tal_manually.isChecked():
            # enable layer selection (disable in def accept)
            self.dlg.cb_tal_layerselection.setEnabled(True)
            # check if "TAL" is in export list, remove if it is
            if "TAL" in list_filetypes_for_export:
                list_filetypes_for_export.remove("TAL")
            # uncheck other button if necessary
            self.dlg.pb_tal_byname.setChecked(False)
            # uncheck "manually" until second window checks it
            self.dlg.pb_tal_manually.setChecked(False)
            open_second_window("TAL")  # open second window
        else:
            # enable layerselection
            self.dlg.cb_tal_layerselection.setEnabled(True)
            # remove filename from list of exports
            list_filetypes_for_export.remove("TAL")

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

    # Create variables of gui functions for later loop
    #  (check TAL-functions above for detailed comments)

    xxx_layerselection_used = \
        "def xxx_layerselection_used():\n" \
        "    if self.dlg.cb_xxx_layerselection.currentLayer() is None:\n" \
        "        self.dlg.pb_xxx_byname.setChecked(False)\n" \
        "        self.dlg.pb_xxx_manually.setChecked(False)\n" \
        "        self.dlg.pb_xxx_byname.setEnabled(False)\n" \
        "        self.dlg.pb_xxx_manually.setEnabled(False)\n" \
        "    else:\n" + \
        "        self.dlg.pb_xxx_byname.setEnabled(True)\n" \
        "        self.dlg.pb_xxx_manually.setEnabled(True)\n"

    xxx_byname_clicked = \
        "def xxx_byname_clicked():\n" \
        "    if self.dlg.pb_xxx_byname.isChecked():\n"\
        "        self.dlg.pb_xxx_manually.setChecked(False)\n" \
        "        self.dlg.cb_xxx_layerselection.setEnabled(False)\n" \
        "        create_dict_by_name('XXX')\n" \
        "        list_filetypes_for_export.append('XXX')\n" \
        "    else:\n" \
        "        self.dlg.cb_xxx_layerselection.setEnabled(True)\n" \
        "        list filetypes for export.remove('XXX')"

    xxx_manually_clicked = \
        "def xxx_manually_clicked():\n" \
        "    if self.dlg.pb_xxx_manually.isChecked():\n" \
        "        self.dlg.cb_xxx_layerselection.setEnabled(True)\n" \
        "        if 'XXX' in list filetypes for export:\n" \
        "            list_filetypes_for_export.remove('XXX')\n" \
        "        self.dlg.pb_xxx_byname.setChecked(False)\n" \
        "        self.dlg.pb_xxx_manually.setChecked(False)\n" \
        "        open_second_window('XXX')\n" \
        "    else:\n" \
        "        self.dlg.cb_xxx_layerselection.setEnabled(True)\n" \
        "        list_filetypes_for_export.remove('XXX')"
```

```
    # execute loop for all standard filetypes
    for i in list_inputfile_types_standard:

        # set filetype names correctly in strings
        i_layerselection_used = \
            xxx_layerselection_used.replace("xxx", i.lower())
        i_byname_clicked = \
            xxx byname clicked.replace("xxx", i.lower())\
            .replace("XXX", i.upper())
        i_manually_clicked = \
            xxx_manually_clicked.replace("xxx", i.lower())\
            .replace("XXX", i.upper())

        # make functions from string with exec()
        exec(i_layerselection_used)
        exec(i_byname_clicked)
        exec(i_manually_clicked)

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

    # checks the inputs for appending fields to layer
    def append_layer_changed():
        if self.dlg.cb_layer_adaption_selection.currentIndex() != 0 and \
                self.dlg.cb_filetype_combobox.currentIndex() != 0:
            self.dlg.pb_append_layer.setEnabled(True)
            self.dlg.pb_append_layer.setText("APPEND")
        else:
            self.dlg.pb_append_layer.setEnabled(False)
            self.dlg.pb_append_layer.setText("what?")

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

    # checks user input in filewidget (export path)
    def filewidget_changed():
        # check if there are spaces in selected path (not allowed)
        if " " in self.dlg.fw_export_path.filePath():
            # print warning in label and disable exports
            self.dlg.lb_path_info.setText("NO SPACES IN PATH")
            self.dlg.lb_path_info.setStyleSheet("color: rgb(200, 0, 0);")
            self.dlg.pb_export.setEnabled(False)
            self.dlg.pb_export_attr_names.setEnabled(False)
            self.dlg.pb_export_attr_names.setText("  no path")
            self.dlg.pb_export_user_manual.setEnabled(False)
            self.dlg.pb_export_user_manual.setText("  no path")
            self.dlg.pb_create_geopackage.setEnabled(False)
            self.dlg.pb_create_geopackage.setText("no path")

        else:
            # check if path exists (is valid)
            if os.path.exists(self.dlg.fw_export_path.filePath()):
                # remove warning from label and enable exports
                self.dlg.lb_path_info\
                    .setText("Export files to this directory:")
                self.dlg.lb_path_info\
                    .setStyleSheet("color: rgb(50, 50, 50);")
                self.dlg.pb_export.setEnabled(True)
                self.dlg.pb_export_attr_names.setEnabled(True)
                self.dlg.pb_export_attr_names.setText("  Export standards")
                self.dlg.pb_export_user_manual.setEnabled(True)
                self.dlg.pb_export_user_manual\
                    .setText("  Export user manual")
                self.dlg.pb_create_geopackage.setEnabled(True)
                self.dlg.pb_create_geopackage\
                    .setText("Create GeoPackage in target directory")

            # remove warning if path is empty, but disable exports
            elif self.dlg.fw_export_path.filePath() == "":
                self.dlg.lb_path_info \
                    .setText("Export files to this directory:")
                self.dlg.lb path info \
                    .setStyleSheet("color: rgb(50, 50, 50);")
                self.dlg.pb_export.setEnabled(False)
                self.dlg.pb_export_attr_names.setEnabled(False)
                self.dlg.pb_export_attr_names.setText("  no path")
                self.dlg.pb_export_user_manual.setEnabled(False)
                self.dlg.pb_export_user_manual.setText("  no path")
                self.dlg.pb_create_geopackage.setEnabled(False)
```

```
            self.dlg.pb_create_geopackage.setText("no path")

        else:
            # print warning in label and disable exports
            self.dlg.lb_path_info\
                .setText("PATH DOES NOT EXIST")
            self.dlg.lb_path_info\
                .setStyleSheet("color: rgb(200, 0, 0);")
            self.dlg.pb export.setEnabled(False)
            self.dlg.pb_export_attr_names.setEnabled(False)
            self.dlg.pb_export_attr_names.setText("  no path")
            self.dlg.pb_export_user_manual.setEnabled(False)
            self.dlg.pb_export_user_manual.setText("  no path")
            self.dlg.pb_create_geopackage.setEnabled(False)
            self.dlg.pb create geopackage.setText("no path")

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# SET TOOL TIPS (the non repetitive ones are set in Qt Designer)

# set tool tips for layerselection comboboxes
for xxx in list_inputfile_types:
    eval('self.dlg.cb_xxx_layerselection'
         '.setToolTip("select layer with data for XXX-file")'
         .replace("xxx", xxx.lower()).replace("XXX", xxx.upper()))

# set tool tips for "by name"-buttons
for xxx in list_inputfile_types:
    eval('self.dlg.pb_xxx_byname'
         '.setToolTip("match attributes for the XXX-file by name")'
         .replace("xxx", xxx.lower()).replace("XXX", xxx.upper()))

# set tool tips for "manually"-buttons
for xxx in list_inputfile_types:
    eval('self.dlg.pb_xxx_manually'
         '.setToolTip("match attributes for the XXX-file manually")'
         .replace("xxx", xxx.lower()).replace("XXX", xxx.upper()))

# set tool tips for all attribute frames
for xx in list_number_attr_file_max:
    eval('self.dlg2.frame_attr_xx.setToolTip("Attribute xx")'
         .replace("xx", str(xx)))

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# DETECT USER INPUT IN GUI

# execute for all filetypes
for j in list_inputfile_types:

    # remove unsuitable layers from layer selection cb
    #  variable marked as unused by IDE, but it is used in eval
    suitable_layers = (QgsMapLayerProxyModel.VectorLayer
                       or QgsMapLayerProxyModel.NoGeometry)
    eval("self.dlg.cb_xxx_layerselection.setFilters(suitable_layers)"
         .replace("xxx", j.lower()))

    # triggers functions if the layer in the selection combo box
    #  is changed
    eval("self.dlg.cb_xxx_layerselection"
         ".layerChanged.connect(xxx_layerselection_used)"
         .replace("xxx", j.lower()))

    # triggers functions if the "by name"-buttons are clicked
    eval("self.dlg.pb xxx byname.clicked.connect(xxx byname clicked)"
         .replace("xxx", j.lower()))

    # triggers functions if the "manually"-buttons are clicked
    eval("self.dlg.pb_xxx_manually.clicked.connect(xxx_manually_clicked)"
         .replace("xxx", j.lower()))

# detect change of file path input
self.dlg.fw_export_path.fileChanged.connect(filewidget_changed)

# "generate"-button for SYS clicked
self.dlg.pb_sys_generate.clicked.connect(generate_sys_layer)

# "export"-button clicked
```

```
    self.dlg.pb_export.clicked.connect(export_clicked)

    # "Export user manual"-button clicked
    self.dlg.pb export user manual.clicked.connect(export user manual)

    # "Export standards"-button clicked
    self.dlg.pb_export_attr_names.clicked.connect(export_standards)

    # "Clear"-button clicked
    self.dlg.pb_clear.clicked.connect(clear_dlg)

    # Connect value replacement line edit to its check-function
    self.dlg.le_value_replacement.textChanged.connect(check_val_rep_char)

    # layer adaption layer combobox changed

self.dlg.cb_layer_adaption_selection.layerChanged.connect(append_layer_changed)

    # layer adaption filetype combobox changed
    self.dlg.cb_filetype_combobox.currentIndexChanged \
        .connect(append_layer_changed)

    # "APPEND"-button clicked
    self.dlg.pb_append_layer.clicked.connect(append_layer)

    # "Create GeoPackage"-button clicked
    self.dlg.pb_create_geopackage.clicked.connect(create_geopackage)

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

    # Buttons of second window:

    # "match attributes by name if possible" button in second window clicked
    self.dlg2.pb_match_attr_by_name_if_possible.clicked \
        .connect(match_attributes_by_name_if_possible)

    # "match attributes by order" button in second window clicked
    self.dlg2.pb_match_attr_by_order.clicked \
        .connect(match_attributes_by_order)

    # "clear all matches" button in second window clicked
    self.dlg2.pb_clear_all_matches.clicked \
        .connect(clear_all_matches)

    # "set field names from first row" button in second window clicked
    self.dlg2.pb_correct_field_names.clicked \
        .connect(correct_field_names)

    # ok button in second window clicked
    self.dlg2.pb_ok.clicked.connect(execute_second_window)

    # chancel button in second window clicked
    self.dlg2.pb_cancel.clicked.connect(reject_second_window)


##############################################################################


# Set the tab order
def set_tab_order():

    # tab order for first window

    self.dlg.setTabOrder(
        self.dlg.cb all layerselection, self.dlg.pb all byname)
    self.dlg.setTabOrder(
        self.dlg.pb_all_byname, self.dlg.pb_all_manually)
    self.dlg.setTabOrder(
        self.dlg.pb_all_manually, self.dlg.pb_sys_generate)

    self.dlg.setTabOrder(
        self.dlg.pb_sys_generate, self.dlg.cb_sys_layerselection)

    self.dlg.setTabOrder(
        self.dlg.cb_sys_layerselection, self.dlg.pb_sys_byname)
    self.dlg.setTabOrder(
        self.dlg.pb_sys_byname, self.dlg.pb_sys_manually)
    self.dlg.setTabOrder(
```

```
        self.dlg.pb_sys_manually, self.dlg.cb_fkt_layerselection)

    self.dlg.setTabOrder(
        self.dlg.cb_fkt_layerselection, self.dlg.pb_fkt_byname)
    self.dlg.setTabOrder(
        self.dlg.pb_fkt_byname, self.dlg.pb_fkt_manually)
    self.dlg.setTabOrder(
        self.dlg.pb_fkt_manually, self.dlg.cb_ktr_layerselection)

    self.dlg.setTabOrder(
        self.dlg.cb_ktr_layerselection, self.dlg.pb_ktr_byname)
    self.dlg.setTabOrder(
        self.dlg.pb_ktr_byname, self.dlg.pb_ktr_manually)
    self.dlg.setTabOrder(
        self.dlg.pb_ktr_manually, self.dlg.cb_ext_layerselection)

    self.dlg.setTabOrder(
        self.dlg.cb_ext_layerselection, self.dlg.pb_ext_byname)
    self.dlg.setTabOrder(
        self.dlg.pb_ext_byname, self.dlg.pb_ext_manually)
    self.dlg.setTabOrder(
        self.dlg.pb_ext_manually, self.dlg.cb_jgg_layerselection)

    self.dlg.setTabOrder(
        self.dlg.cb_jgg_layerselection, self.dlg.pb_jgg_byname)
    self.dlg.setTabOrder(
        self.dlg.pb_jgg_byname, self.dlg.pb_jgg_manually)
    self.dlg.setTabOrder(
        self.dlg.pb_jgg_manually, self.dlg.cb_tgg_layerselection)

    self.dlg.setTabOrder(
        self.dlg.cb_tgg_layerselection, self.dlg.pb_tgg_byname)
    self.dlg.setTabOrder(
        self.dlg.pb_tgg_byname, self.dlg.pb_tgg_manually)
    self.dlg.setTabOrder(
        self.dlg.pb_tgg_manually, self.dlg.cb_wgg_layerselection)

    self.dlg.setTabOrder(
        self.dlg.cb_wgg_layerselection, self.dlg.pb_wgg_byname)
    self.dlg.setTabOrder(
        self.dlg.pb_wgg_byname, self.dlg.pb_wgg_manually)
    self.dlg.setTabOrder(
        self.dlg.pb_wgg_manually, self.dlg.cb_bek_layerselection)

    self.dlg.setTabOrder(
        self.dlg.cb_bek_layerselection, self.dlg.pb_bek_byname)
    self.dlg.setTabOrder(
        self.dlg.pb_bek_byname, self.dlg.pb_bek_manually)
    self.dlg.setTabOrder(
        self.dlg.pb_bek_manually, self.dlg.cb_ein_layerselection)

    self.dlg.setTabOrder(
        self.dlg.cb_ein_layerselection, self.dlg.pb_ein_byname)
    self.dlg.setTabOrder(
        self.dlg.pb_ein_byname, self.dlg.pb_ein_manually)
    self.dlg.setTabOrder(
        self.dlg.pb_ein_manually, self.dlg.cb_ezg_layerselection)

    self.dlg.setTabOrder(
        self.dlg.cb_ezg_layerselection, self.dlg.pb_ezg_byname)
    self.dlg.setTabOrder(
        self.dlg.pb_ezg_byname, self.dlg.pb_ezg_manually)
    self.dlg.setTabOrder(
        self.dlg.pb_ezg_manually, self.dlg.cb_fka_layerselection)

    self.dlg.setTabOrder(
        self.dlg.cb_fka_layerselection, self.dlg.pb_fka_byname)
    self.dlg.setTabOrder(
        self.dlg.pb_fka_byname, self.dlg.pb_fka_manually)
    self.dlg.setTabOrder(
        self.dlg.pb_fka_manually, self.dlg.cb_hya_layerselection)

    self.dlg.setTabOrder(
        self.dlg.cb_hya_layerselection, self.dlg.pb_hya_byname)
    self.dlg.setTabOrder(
        self.dlg.pb_hya_byname, self.dlg.pb_hya_manually)
    self.dlg.setTabOrder(
        self.dlg.pb_hya_manually, self.dlg.cb_rue_layerselection)
```

```
        self.dlg.setTabOrder(
            self.dlg.cb_rue_layerselection, self.dlg.pb_rue_byname)
        self.dlg.setTabOrder(
            self.dlg.pb_rue_byname, self.dlg.pb_rue_manually)
        self.dlg.setTabOrder(
            self.dlg.pb_rue_manually, self.dlg.cb_tal_layerselection)

        self.dlg.setTabOrder(
            self.dlg.cb_tal_layerselection, self.dlg.pb_tal_byname)
        self.dlg.setTabOrder(
            self.dlg.pb_tal_byname, self.dlg.pb_tal_manually)
        self.dlg.setTabOrder(
            self.dlg.pb_tal_manually, self.dlg.cb_trs_layerselection)

        self.dlg.setTabOrder(
            self.dlg.cb_trs_layerselection, self.dlg.pb_trs_byname)
        self.dlg.setTabOrder(
            self.dlg.pb_trs_byname, self.dlg.pb_trs_manually)
        self.dlg.setTabOrder(
            self.dlg.pb_trs_manually, self.dlg.cb_urb_layerselection)

        self.dlg.setTabOrder(
            self.dlg.cb_urb_layerselection, self.dlg.pb_urb_byname)
        self.dlg.setTabOrder(
            self.dlg.pb_urb_byname, self.dlg.pb_urb_manually)
        self.dlg.setTabOrder(
            self.dlg.pb_urb_manually, self.dlg.cb_ver_layerselection)

        self.dlg.setTabOrder(
            self.dlg.cb_ver_layerselection, self.dlg.pb_ver_byname)
        self.dlg.setTabOrder(
            self.dlg.pb_ver_byname, self.dlg.pb_ver_manually)
        self.dlg.setTabOrder(
            self.dlg.pb_ver_manually, self.dlg.cb_boa_layerselection)

        self.dlg.setTabOrder(
            self.dlg.cb_boa_layerselection, self.dlg.pb_boa_byname)
        self.dlg.setTabOrder(
            self.dlg.pb_boa_byname, self.dlg.pb_boa_manually)
        self.dlg.setTabOrder(
            self.dlg.pb_boa_manually, self.dlg.cb_bod_layerselection)

        self.dlg.setTabOrder(
            self.dlg.cb_bod_layerselection, self.dlg.pb_bod_byname)
        self.dlg.setTabOrder(
            self.dlg.pb_bod_byname, self.dlg.pb_bod_manually)
        self.dlg.setTabOrder(
            self.dlg.pb_bod_manually, self.dlg.cb_efl_layerselection)

        self.dlg.setTabOrder(
            self.dlg.cb_efl_layerselection, self.dlg.pb_efl_byname)
        self.dlg.setTabOrder(
            self.dlg.pb_efl_byname, self.dlg.pb_efl_manually)
        self.dlg.setTabOrder(
            self.dlg.pb_efl_manually, self.dlg.cb_lnz_layerselection)

        self.dlg.setTabOrder(
            self.dlg.cb_lnz_layerselection, self.dlg.pb_lnz_byname)
        self.dlg.setTabOrder(
            self.dlg.pb_lnz_byname, self.dlg.pb_lnz_manually)
        self.dlg.setTabOrder(
            self.dlg.pb_lnz_manually, self.dlg.cb_dif_layerselection)

        self.dlg.setTabOrder(
            self.dlg.cb_dif_layerselection, self.dlg.pb_dif_byname)
        self.dlg.setTabOrder(
            self.dlg.pb_dif_byname, self.dlg.pb_dif_manually)
        self.dlg.setTabOrder(
            self.dlg.pb_dif_manually, self.dlg.fw_export_path)

        self.dlg.setTabOrder(self.dlg.fw_export_path, self.dlg.pb_export)
        self.dlg.setTabOrder(self.dlg.pb_export, self.dlg.pb_cancel)

        # second tab of first window

        self.dlg.setTabOrder(
            self.dlg.pb_cancel, self.dlg.pb_export_attr_names)
```

Code für plugin_functions.py – Seite 127

```
    self.dlg.setTabOrder(
        self.dlg.pb_export_attr_names, self.dlg.lb_export_user_manual)
    self.dlg.setTabOrder(
        self.dlg.lb export user manual, self.dlg.cb open explorer)
    self.dlg.setTabOrder(
        self.dlg.cb_open_explorer, self.dlg.cb_close_after_export)

    self.dlg.setTabOrder(
        self.dlg.cb close after export, self.dlg.le value replacement)
    self.dlg.setTabOrder(
        self.dlg.le_value_replacement, self.dlg.cb_bottom_every_file)
    self.dlg.setTabOrder(
        self.dlg.cb_bottom_every_file, self.dlg.cb_separate_file)

    self.dlg.setTabOrder(
        self.dlg.cb_separate_file, self.dlg.le_project_name)
    self.dlg.setTabOrder(
        self.dlg.le_project_name, self.dlg.pb_type_in_name)
    self.dlg.setTabOrder(
        self.dlg.pb_type_in_name, self.dlg.pb_layer_in_name)
    self.dlg.setTabOrder(
        self.dlg.pb_layer_in_name, self.dlg.pb_time_in_name)

    self.dlg.setTabOrder(
        self.dlg.pb_time_in_name, self.dlg.cb_layer_adaption_selection)
    self.dlg.setTabOrder(
        self.dlg.cb_layer_adaption_selection, self.dlg.cb_filetype_combobox)
    self.dlg.setTabOrder(
        self.dlg.cb_filetype_combobox, self.dlg.pb_append_layer)
    self.dlg.setTabOrder(
        self.dlg.pb_append_layer, self.dlg.pb_create_geopackage)

    self.dlg.setTabOrder(
        self.dlg.pb_create_geopackage, self.dlg.pb_clear)
    self.dlg.setTabOrder(
        self.dlg.pb_clear, self.dlg.fw_export_path)
    self.dlg.setTabOrder(
        self.dlg.fw_export_path, self.dlg.pb_export)
    self.dlg.setTabOrder(
        self.dlg.pb_export, self.dlg.pb_cancel)

    # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

    # tab order for second window

    self.dlg2.setTabOrder(
        self.dlg2.pb_ok,
        self.dlg2.pb_cancel)
    self.dlg2.setTabOrder(
        self.dlg2.pb_cancel,
        self.dlg2.scrollArea_example)

    self.dlg2.setTabOrder(
        self.dlg2.scrollArea_example,
        self.dlg2.scrollArea_attr_match)
    self.dlg2.setTabOrder(
        self.dlg2.scrollArea_attr_match,
        self.dlg2.pb_match_attr_by_name_if_possible)

    self.dlg2.setTabOrder(
        self.dlg2.pb_match_attr_by_name_if_possible,
        self.dlg2.pb_match_attr_by_order)
    self.dlg2.setTabOrder(
        self.dlg2.pb_match_attr_by_order,
        self.dlg2.pb clear all matches)

    self.dlg2.setTabOrder(
        self.dlg2.pb_clear_all_matches,
        self.dlg2.pb_correct_field_names)
    self.dlg2.setTabOrder(
        self.dlg2.pb correct field names,
        self.dlg2.pb_ok)


################################################################################
#   ADAPTING EXISTING LAYERS                                                   #
#       Änderungen an bestehenden Layern (Kapitel 5.6)                         #
################################################################################
```

```
# Correct i.e. set field names in attribute table of faulty layer
def correct field names():

    # get current filetype of second window from global variable
    filetype = current_filetype_second_window

    # get selected layer for current filetype
    faulty_layer = \
        eval("self.dlg.cb_xxx_layerselection.currentLayer()"
             .replace("xxx", str(filetype.lower())))

    # start editing layer
    faulty layer.startEditing()

    # change field names
    for field in faulty_layer.fields():
        # checks if names are really faulty
        if field.name()[:5] == "Field":
            # get index of current field
            index_faulty_name = \
                faulty_layer.fields().indexFromName(field.name())

            # get the features of the layer
            feature_count = 0
            value_first_row = ""
            for feature in faulty_layer.getFeatures():
                # get the first entry in that column
                while feature_count < 1:
                    feature_count += 1
                    value_first_row = feature[field.name()]

            # give column new field name
            faulty_layer.renameAttribute(
                index_faulty_name, str(value_first_row))

    # stop editing layer
    faulty_layer.commitChanges()

    # The first row (containing the field names) could have been deleted,
    #  but this would change the original data in the excel / csv file.
    # Therefore the row with the field names gets filtered out in the
    #  export file function.

    # If the QGIS layer is deleted later, it may create a copy of the
    #  first row in the original excel / csv file.

    # reopen second window so that changes take effect in the comboboxes
    open_second_window(filetype)


################################################################################


# Get information from GUI and give it to general function
def append_layer():

    # get layer which to append to
    layer_to_append = self.dlg.cb_layer_adaption_selection.currentLayer()

    # get filetype from combobox
    filetype = self.dlg.cb_filetype_combobox.currentText()

    # give to generic function
    append_layer_generic(layer_to_append, filetype)

    # display message in QGIS
    self.iface.messageBar().pushMessage(str(filetype.upper() +
                                            "-fields appended to layer: "),
                                        str(layer to append.name()))

    # do something special if filetype is SYS
    if filetype.upper() == "SYS":
        change_widget_type_sys(layer_to_append)


################################################################################
```

```
# Append fields of a specific filetype to a specific layer
def append layer generic(layer to append, filetype):

    # get a list of the field names for this filetype
    current_filetype_fields_list = eval(filetype.lower() + "_file_attr_list")

    # get a dictionary for the current field type definitions
    current_dict_type = eval(filetype.lower() + "_file_attr_type_dict")

    # create a new dictionary with field name as key
    #  and QVariant.XXX as value
    field_name_to_qvariant_dict = {}
    for field name, field type \
            in zip(current_filetype_fields_list, current_dict_type.values()):

        # set field type to int for integer
        if field_type.startswith("i"):
            field_name_to_qvariant_dict[str(field_name)] = QVariant.Int
        # set field type to double (other word for float)
        elif field_type.startswith("f"):
            field_name_to_qvariant_dict[str(field_name)] = QVariant.Double
        # set field type for dates or time to datetime
        elif field_type.startswith(("T", "M", "J", "h", "m")):
            field_name_to_qvariant_dict[str(field_name)] = QVariant.DateTime
        # set field type of a decision value or string to string
        else:
            field_name_to_qvariant_dict[str(field_name)] = QVariant.String

    # get a list of existing field names in layer
    existing_fields_list = []
    for field in layer_to_append.fields():
        existing_fields_list.append(field.name())

    # get a list of fields that are to append to the layer
    fields_to_append_list = []
    for field_name in current_filetype_fields_list:
        if field_name not in existing_fields_list:
            fields_to_append_list.append(field_name)

    # add all necessary fields to the layer with its correct type
    for field_name in fields_to_append_list:
        layer_to_append.dataProvider().addAttributes(
            [QgsField(str(field_name),
                      field_name_to_qvariant_dict[field_name])])

    # update the attribute table
    layer_to_append.updateFields()


################################################################################


# change field widget type for SYS
def change_widget_type_sys(layer):

    # list of fields in SYS file that need a drop down menu
    list_sys_fields_drop_down = ["Zulauf_1", "Zulauf_2", "Zulauf_3",
                                 "Ablauf_1", "Ablauf_2", "Ablauf_3"]

    # the source for the drop down menu should be this field
    field_source = "Nr"

    # loop through all fields that need a drop down menu & send to function
    for field in list_sys_fields_drop_down:
        change_widget_type(layer, field, field_source)


################################################################################


# change widget type of field in attribute table to a drop down menu
def change_widget_type(layer, field_to_change, field_source):

    # get index of field_to_change in layer
    i = layer.fields().indexFromName(field_to_change)
```

```
    # define the change (ValueRelation with values from field_source)
    new_setup = QgsEditorWidgetSetup('ValueRelation',
                                     {'AllowMulti': False,
                                      'AllowNull': True,
                                      'FilterExpression': '',
                                      'Key': field_source,
                                      'Layer': layer.id(),
                                      'NofColumns': 1,
                                      'OrderByValue': False,
                                      'UseCompleter': False,
                                      'Value': field_source})

    # execute change for field with index i
    layer.setEditorWidgetSetup(i, new_setup)


##############################################################################
#   SUPPORT FUNCTIONS  -  Support-Funktionen (Kapitel 5.7)                   #
##############################################################################


# create and layer and fill it with SYS-information from other Layers
def generate_sys_layer():

    # define field name of ID in element layers
    field_name_id = "BlueM_ID"

    # define field name of where they output to in element layers
    field_name_output_to = "Output_to"

    # list of filetypes that contain information about elements
    element_filetypes_list = ["BEK", "EIN", "EZG", "FKA", "HYA",
                              "RUE", "TAL", "TRS", "URB", "VER"]

    # get a list of layers with element-data
    element_data_layer_list = []

    for XXX in element_filetypes_list:

        # check for the current element-filetype if a layer is selected
        if eval("self.dlg.cb_xxx_layerselection.currentIndex()"
                .replace("xxx", XXX.lower())) != -1:

            # if a layer is selected, append it to the list
            element_data_layer_list.append(
                eval("self.dlg.cb_xxx_layerselection.currentLayer()"
                     .replace("xxx", XXX.lower())))

    # if there is at least one layer with element-data selected
    if len(element_data_layer_list) != 0:

        # create empty dictionary for sys elements
        sys_elements_dict = {}

        # create a list of invalid values for ID
        nono_list = [None, "", "NULL", " ", "None"]

        # go through every element data layer
        for layer in element_data_layer_list:

            # check if "layer" really is something
            if layer is not None:

                # check if both fields exist
                if layer.fields().indexOf(field name id) != -1 \
                 and layer.fields().indexOf(field_name_output_to) != -1:

                    # go through every feature of current layer
                    for feature in layer.getFeatures():

                        # get output to with ID into dictionary
                        sys_id = feature[field_name_id]
                        output_to = feature[field_name_output_to]
                        if sys_id not in nono_list:
                            sys_elements_dict[sys_id] = str(output_to)

        # Add ZPG ("Zielpegel") with no output to elements-dictionary
        sys_elements_dict["ZPG"] = ""
```

```
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# CREATE A NEW LAYER

# define name for new sys-layer
name_new_sys_layer = "Generated_SYS_layer_temporary"

# check if there is already a layer with that name
if len(QgsProject.instance().mapLayersByName(
        name_new_sys_layer)) != 0:

    # remove other layers
    for evil_layer in QgsProject.instance()\
            .mapLayersByName(name_new_sys_layer):
        QgsProject.instance().removeMapLayer(evil layer)

# create a new layer with that name and add it to the project
new_sys_layer = QgsVectorLayer(
    "NoGeometry", name_new_sys_layer, "memory")
QgsProject.instance().addMapLayer(new_sys_layer)

# get the field names for the new sys layer via other functions
append_layer_generic(new_sys_layer, "SYS")
change_widget_type_sys(new_sys_layer)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# FILL THAT LAYER WITH FEATURES

# for all sys elements, reformat output_to
for element in sys_elements_dict.keys():

    # divide up dict value, if there are multiple parts to it
    value = sys_elements_dict[element]
    value = value.replace(";", " ")
    value = value.replace(",", " ")
    value_list = value.split(" ")

    # remove empty values (from to many spaces)
    while "" in value_list:
        value_list.remove("")

    # make sure only a max of 3 items are in value list
    while len(value_list) > 3:
        value_list.pop()

    # put it back into the dictionary (overwrite old value)
    sys_elements_dict[element] = value_list

# create a new dictionary for input_from
sys_input_from = {}

# for all sys elements, build a reverse dictionary
for element1 in sys_elements_dict.keys():

    # list for element1 input_from
    element1_list = []

    # for every element check every dict entry
    for element2 in sys_elements_dict.keys():

        # check where the element is in the output_to_list
        if element1 in sys_elements_dict[element2]:

            # add element2 to the list of input_from for element1
            element1 list.append(element2)

    # cut element1 list, if it wrongly contains more than 3
    while len(element1_list) > 3:
        element1_list.pop()

    # add element1 and its list to the new dict
    sys_input_from[element1] = element1_list

# create empty list of features
feat_list = []

# for all sys elements, reformat output_to
for element in sys_elements_dict.keys():
```

```
            # add the feature with its info to the layer
            feat = QgsFeature()
            feat.setFields(new sys layer.fields())
            feat.setAttribute(1, element)

            # add input_from to attributes
            for x in range(0, len(sys_input_from[element])):
                feat.setAttribute(2 + x, sys input from[element][x])

            # add output_to to attributes
            for x in range(0, len(sys_elements_dict[element])):
                feat.setAttribute(5 + x, sys_elements_dict[element][x])

            # add feature to list of features
            feat_list.append(feat)

        # start editing layer and add features from list to it
        with edit(new_sys_layer):
            new_sys_layer.addFeatures(feat_list)

        # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
        # HOUSEKEEPING

        # select new layer in combobox
        self.dlg.cb_sys_layerselection.setLayer(new_sys_layer)

        # check "by name"-button and additional stuff (not the normal way)
        if self.dlg.pb_sys_byname.isChecked():
            create_dict_by_name('SYS')
        else:
            self.dlg.pb_sys_byname.setChecked(True)
            self.dlg.pb_sys_manually.setChecked(False)
            self.dlg.cb_sys_layerselection.setEnabled(False)
            create_dict_by_name('SYS')
            list_filetypes_for_export.append('SYS')

        # display message with information about SYS-layer creation
        self.iface.messageBar().pushMessage(
            "SYS-layer created",
            "A new layer was created from element-data of the other layers "
            "and added to the SYS-file layerselection")

    else:
        # display a warning if there are no layers with element data
        self.iface.messageBar().pushMessage(
            "No elements for new SYS-layer found",
            "Please select layers for the element-files with suitable data")


################################################################################


# Opens a windows explorer window of the export directory
def open_explorer_window():

    # only if option is checked in settings (GUI, dlg, tab2)
    if self.dlg.cb_open_explorer.isChecked():
        export_path = str(self.dlg.fw_export_path.filePath())
        os.system(str('start ' + export_path))


################################################################################


# Get the filetype index for this filetype
def get_filetype_index(filetype):

    # get the index of the current filetype in the array
    #  (only search in the "name_short" column, otherwise there might be
    #  more than one match)

    # start at -1 one because first row is then 0
    row_index = -1
    # filetype_index has to be set here to avoid a warning, but will be set
    #  correctly in the if-statement
    filetype_index = 1
    # go through every row of array and check the "name_short" column
```

```
    for row in inputfiles_overview:
        row_index += 1
        if row[int(index_name_short)] == filetype:
            filetype index = row index

    # give filetype index back
    return filetype_index


#############################################################################


# Get data about "this" type of file from csv i.e. array
def get_type_info(filetype):

    # get current layer from specific combobox
    current_layer = \
        eval("self.dlg.cb_xxx_layerselection.currentLayer()"
             .replace("xxx", filetype.lower()))

    # get filetype index from function
    filetype_index = get_filetype_index(filetype)

    # get the number of attributes for the filetype
    number_attr_filetype = \
        int(inputfiles_overview[int(filetype_index)]
                               [int(index_attr_count)])

    # create list of numbers of attributes for this filetype
    list_number_attr_file = []
    for i in range(1, number_attr_filetype + 1):
        if len(str(i)) == 1:  # give i a leading zero if necessary
            i = "0" + str(i)
        list_number_attr_file.append(str(i))

    # get attribute insertion pattern for this filetype from csv
    pattern = str(inputfiles_overview[filetype_index][index_pattern])
    # remove both ends of element
    pattern = pattern.lstrip("['").rstrip("']")
    # exchange "\n" if necessary for multi line pattern
    pattern = pattern.replace(r"\n", "\n")
    # add line escape to end of pattern
    pattern = pattern + "\n"
    # replace possible \xa0 with real spaces
    pattern = pattern.replace(r'\xa0', " ")

    # get matching headlines from csv
    headlines = str(inputfiles_overview
                    [filetype_index]
                    [index_headlines])
    # crop both ends of string
    headlines = headlines.lstrip("['").rstrip("']")
    # "\n" is a valid character in the original csv element,
    #  so it has to be replaced with a line escape
    headlines = headlines.replace(r"\n", "\n")
    # crop possible line escape from the end
    headlines = headlines.rstrip("\n")
    # add line escape at the end
    headlines = headlines + "\n"
    # replace possible \xa0 with real spaces
    headlines = headlines.replace(r'\xa0', " ")

    # get matching additional lines 1 ("add_lines_1") from csv
    add_lines_1 = \
        str(inputfiles overview[filetype index][index add lines 1])
    # crop both ends of string
    add_lines_1 = add_lines_1.lstrip("['").rstrip("']")
    # exchange "\n" if necessary for multi input
    add_lines_1 = add_lines_1.replace(r"\n", "\n")
    # crop possible line escape from the end
    add lines 1 = add lines 1.rstrip("\n")
    # replace possible \xa0 with real spaces
    add_lines_1 = add_lines_1.replace(r'\xa0', " ")

    # get matching additional lines 2 ("add_lines_2") from csv
    add_lines_2 = \
        str(inputfiles_overview[filetype_index][index_add_lines_2])
    # crop both ends of string
```

```
    add_lines_2 = add_lines_2.lstrip("['").rstrip("']")
    # exchange "\n" if necessary for multi input
    add_lines_2 = add_lines_2.replace(r"\n", "\n")
    # crop possible line escape from the end
    add_lines_2 = add_lines_2.rstrip("\n")
    # replace possible \xa0 with real spaces
    add_lines_2 = add_lines_2.replace(r'\xa0', " ")

    # get matching additional lines 3 ("add lines 3") from csv
    add_lines_3 = \
        str(inputfiles_overview[filetype_index][index_add_lines_3])
    # crop both ends of string
    add_lines_3 = add_lines_3.lstrip("['").rstrip("']")
    # exchange "\n" if necessary for multi input
    add_lines_3 = add_lines_3.replace(r"\n", "\n")
    # crop possible line escape from the end
    add_lines_3 = add_lines_3.rstrip("\n")
    # replace possible \xa0 with real spaces
    add_lines_3 = add_lines_3.replace(r'\xa0', " ")

    # get matching additional lines 4 ("add_lines_4") from csv
    add_lines_4 = \
        str(inputfiles_overview[filetype_index][index_add_lines_4])
    # crop both ends of string
    add_lines_4 = add_lines_4.lstrip("['").rstrip("']")
    # exchange "\n" if necessary for multi input
    add_lines_4 = add_lines_4.replace(r"\n", "\n")
    # crop possible line escape from the end
    add_lines_4 = add_lines_4.rstrip("\n")
    # replace possible \xa0 with real spaces
    add_lines_4 = add_lines_4.replace(r'\xa0', " ")

    # get matching additional lines 5 ("add_lines_5") from csv
    add_lines_5 = \
        str(inputfiles_overview[filetype_index][index_add_lines_5])
    # crop both ends of string
    add_lines_5 = add_lines_5.lstrip("['").rstrip("']")
    # exchange "\n" if necessary for multi input
    add_lines_5 = add_lines_5.replace(r"\n", "\n")
    # crop possible line escape from the end
    add_lines_5 = add_lines_5.rstrip("\n")
    # replace possible \xa0 with real spaces
    add_lines_5 = add_lines_5.replace(r'\xa0', " ")

    # get matching additional lines 6 ("add_lines_6") from csv
    add_lines_6 = \
        str(inputfiles_overview[filetype_index][index_add_lines_6])
    # crop both ends of string
    add_lines_6 = add_lines_6.lstrip("['").rstrip("']")
    # exchange "\n" if necessary for multi input
    add_lines_6 = add_lines_6.replace(r"\n", "\n")
    # crop possible line escape from the end
    add_lines_6 = add_lines_6.rstrip("\n")
    # replace possible \xa0 with real spaces
    add_lines_6 = add_lines_6.replace(r'\xa0', " ")

    # get matching additional lines 7 ("add_lines_7") from csv
    add_lines_7 = \
        str(inputfiles_overview[filetype_index][index_add_lines_7])
    # crop both ends of string
    add_lines_7 = add_lines_7.lstrip("['").rstrip("']")
    # exchange "\n" if necessary for multi input
    add_lines_7 = add_lines_7.replace(r"\n", "\n")
    # crop possible line escape from the end
    add_lines_7 = add_lines_7.rstrip("\n")
    # replace possible \xa0 with real spaces
    add_lines_7 = add_lines_7.replace(r'\xa0', " ")

    # get matching last line from csv
    last_line = \
        str(inputfiles_overview[filetype_index][index_last_line])
    # crop both ends of string
    last_line = last_line.lstrip("['").rstrip("']")
    # exchange "\n" if necessary for multi input
    last_line = last_line.replace(r"\n", "\n")
    # crop possible line escape from the end
    last_line = last_line.rstrip("\n")
    # replace possible \xa0 with real spaces
```

```
    last_line = last_line.replace(r'\xa0', " ")

    # give the data back
    return (current layer, list number attr file,
            pattern, headlines, last_line,
            add_lines_1, add_lines_2, add_lines_3,
            add_lines_4, add_lines_5, add_lines_6,
            add_lines_7)


################################################################################


# Check the value replacement character
def check val rep char():

    char = self.dlg.le_value_replacement.text()

    if char not in list_valid_chars_not_space:
        char = " "

    global val_rep_char
    val_rep_char = char

    # set text in line edit with something valid
    self.dlg.le_value_replacement.setText(val_rep_char)


################################################################################


# Construct a name for the file that is currently written
def construct_filename(filetype):

    # get export path
    export_path = self.dlg.fw_export_path.filePath()
    export_path = export_path.rstrip(str("\\"))

    # get project name from line edit
    project_name = self.dlg.le_project_name.text()

    # get filetype name in suitable format
    filetype_name = str(filetype.upper() + "_file")

    # get name of source layer
    name_source_layer = \
        eval("self.dlg.cb_xxx_layerselection.currentText()"
             .replace("xxx", filetype.lower()))
    # remove possible spaces from layer name
    name_source_layer = name_source_layer.replace(" ", "_")
    # add "from"
    name_source_layer = str("from_" + name_source_layer)

    # get time of export
    now = datetime.now()
    # reformat suitably
    time_of_export = now.strftime("%Y%m%d_%H%M")
    time_of_export_2 = now.strftime("%Y/%m/%d at %H:%M")

    # construct the name of the file
    filename = ""

    # add project name
    if project_name is not "":
        filename = self.dlg.le project name.text()

    # add filetype
    if self.dlg.pb_type_in_name.isChecked():
        filename = str(filename + "_" + filetype_name)

    # add source layer
    if self.dlg.pb_layer_in_name.isChecked():
        filename = str(filename + "_" + name_source_layer)

    # add time of export
    if self.dlg.pb_time_in_name.isChecked():
        filename = str(filename + "_" + time_of_export)
```

```
    # remove "_" from front if project name is empty
    if project_name == "":
        filename = filename.lstrip("_")

    # remove 1 char from front if project name is "_" and
    #  other things are in filename
    if project_name == "_" and len(filename) > 1:
        filename = filename[1:]

    # make name something, if it is not already
    if filename == "":
        filename = "unnamed"

    # add export path to filename
    filename = str(export path + str("\\") + filename)

    # shorten filename if necessary
    max_filename_len = 250  # not 255 because of extension
    #  32000+ characters possible since Windows 10

    if len(filename) > max_filename_len:
        # shorten filename to maximum possible
        filename = filename[0:max_filename_len]

    # add matching file extension to file name
    filename = str(filename + "." + filetype.lower())

    # give filename back
    return filename, time_of_export_2


##############################################################################


# Clear layerselection comboboxes and uncheck all buttons in first dialog
def clear_dlg():

    # clear list of files for export
    global list_filetypes_for_export
    list_filetypes_for_export = []

    # for all filetypes
    for xxx in list_inputfile_types:

        # enable and set layerselection combobox to index 0 (empty)
        eval("self.dlg.cb_xxx_layerselection.setEnabled(True)"
             .replace("xxx", xxx.lower()))
        eval("self.dlg.cb_xxx_layerselection.setCurrentIndex(0)"
             .replace("xxx", xxx.lower()))

        # uncheck and disable "by name" button
        eval("self.dlg.pb_xxx_byname.setChecked(False)"
             .replace("xxx", xxx.lower()))
        eval("self.dlg.pb_xxx_byname.setEnabled(False)"
             .replace("xxx", xxx.lower()))

        # uncheck and disable "manually" button
        eval("self.dlg.pb_xxx_manually.setChecked(False)"
             .replace("xxx", xxx.lower()))
        eval("self.dlg.pb_xxx_manually.setEnabled(False)"
             .replace("xxx", xxx.lower()))


##############################################################################
```

```
"""
                         ,&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
                      &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
                   .&&&&&&&&&&&,                                                 &&&&&&&&&&
                 &&&&&&&  &&&&&,                                             &&&&&&&.&&&&&
               &&&&&&&    &&&&&,                                          &&&&&&&   &&&&&
            &&&&&&&.      &&&&&,                                      &&&&&&&//    &&&&&
          &&&&&&&         &&&&&, &&,,,,(&  (&*,,,,&& *&*,&&&&&&&,,,,,,&&&&&&&
       &&&&&&&,           &&&##/,,,,,,,,,*,,,,,,,,,,,&&&&&&&/,,,,,,,&..&&&&&
     &&&&&&&             &#####*,,,,,,/,,,,,,,,,,/,,&&&&&&&,,,,,,,*&....&&&&&
  &&&&&&&*             #%,,(####*,,,,,,,,,,,,,,/,,&&&&&&&(,,,,,,,&,......&&&&&
%&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&,,,,,,,,&.........&&&&&
%&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&#,,,,,,,&(...........&&&&&
%&&&&/          &,,,,,,,,,,##(##*,,,,,,/,,,,,,,,&&&&&,,,,,,&.............&&&&&
%&&&&/       #%,,/,,,,,,,,*######*,,,*,,,,,,,,,*&&&&&,,,#%...............&&&&&
%&&&&/     &,,*,,,,,,,,,*,,#####*,,,,,,,,,,,,,,&&&&&,&.................&&&&&
%&&&&/  #%,,(,,,,,,,,,&&,,,,###%&&&,,,,,,,#&&,,,&&&&&.....................&&&&&
%&&&&&&&.....&&&&.....(&&&(((,...&&&.........&&&&&.....................&&&&&
%&&&&(.....................(((((,.............&&&&&.....................&&&&&
%&&&&(.....................(((((((((((((((((((&&&&&(((((((((((((((((((&&&&&
%&&&&(.................(((((((((((((((((((((&&&&&(((((((((((((((((((%&&&&&&
%&&&&(...............,(((((((.................&&&&&.................&&&&&&,
%&&&&(.............(((((((.....................&&&&&...............&&&&&&
%&&&&/..........,(((((((.......................&&&&&............%&&&&&&*
%&&&&/........(((((((.........................&&&&&..........&&&&&&
%&&&&/.....,(((((((...........................&&&&&.......%&&&&&(
%&&&&/...(((((((.............................&&&&&.....&&&&&&
%&&&&/.(((((((...............................&&&&&..(&&&&&#
%&&&&&(((((,.................................&&&&&&&&&&&
%&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&%
%&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

"""
```