HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Google

Plaso SQLite Plugin Scaffolder

Bachelorarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Frühlingssemester 2017

Autor:	Claudia Saxer
Betreuer:	Jürg Jucker
Projektpartner:	Google GmbH, Zürich Schweiz
Experte:	Maurin Egler
Gegenleser:	Andreas Steffen

Abstract

Ausgangslage

Plaso ist die Backend Engine für das Tool Log2timeline, welches Google für die forensische Datenanalyse verwendet. Log2timeline ist ein Tool, welches aus einem Image alle Informationen sammelt um eine Super Timeline zu erstellen, welches ein riesiges Logfile darstellt.
Plaso ist zu einem grossen Opensource Framework gewachsen, welches von vielen Security Engineers verwendet wird und beinhaltet eine Ansammlung von Plugins. Plugins werden von Log2timeline verwendet um Informationen von einem File zu extrahieren. Ein Grossteil der Plugins wurden für verschiedene SQLite Datenbanken entwickelt und die Nachfrage für neue Plugins für SQLite Datenbanken steigt.

Im Moment involviert das Hinzufügen eines Plugins das manuelle Verändern von mehreren Files in vorgegebener Reihenfolge.

Vorgehensweise

In einer ersten Phase soll deshalb das Erstellen von SQLite Plugins mit einem Gerüst vereinfacht und teilweise automatisiert werden. Somit sollen auch Security Engineers, welche nicht viel Erfahrung mit der Entwicklung haben, schnell und einfach ein Plugin zu Plaso hinzufügen können. Dies verringert den manuellen Aufwand für das Schreiben der Plugins, sowie für die Codereviews.
Das Tool soll ein ähnliches Konzept verwenden, welches in anderen Framework als "scaffolding" bekannt ist. Das Tool soll von dem Benutzer Eingaben entgegennehmen und daraus Files generieren sowie bestehende editieren um schnell ein neues SQLite Plugin zu erstellen.

Ergebnis

Im Rahmen dieses Projektes wurde ein Command Line Interface Tool erstellt, welches interaktiv Daten des Benutzers entgegennimmt und validiert sowie Daten einer SQLite Datenbank auswertet.

Die Applikation erlaubt durch ihren Aufbau das einfache Hinzufügen von weiteren Plugin Scaffolder. Der SQLite Scaffolder besitzt einerseits einen interaktiven Vorgang und erlaubt auch das Parametrisieren von einigen Eingaben. Die Kommunikation zwischen User und Applikation läuft mit dem Command Line Interface Tool Click, welches auf optparse aufbaut. Die Files Templates wurden mit Jinja2 erstellt und sind einfach anpassbar. Das Tool beinhaltet das Editieren, Erstellen sowie Kopieren von Files. Die generierten Files sowie das Programm sind mit Pylint validiert.

Erklärung

Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde.
- dass ich sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe.
- dass ich keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt habe.

Rapperswil, 16. Juni 2017

Claudia Saxer

Inhalt

Abstract **2**
Erklärung **3**
Inhalt **4**
1 Management Summary **6**
1.1 Ausgangslage 6
1.2 Vorgehen und Technologien 6
1.3 Ergebnisse 6
1.4 Ausblick 6
2 Einleitung **7**
2.1 Aufgabenstellung 7
2.2 Problemstellung, Vision 7
2.3 Ziele 7
2.4 Rahmenbedingungen 7
2.5 Methode / Vorgehen 7
3 Plaso **8**
3.1 Geschichte 8
3.2 Definition 8
3.3 Ziel 8
3.4 Benutzung 9
3.5 Fall Beispiel 10
4 Plugin Aufbau **11**
4.1 Eingaben vom User 12
4.2 Parser 13
4.3 Formatter 15
4.4 INIT 16
4.5 Formatter Test 17
4.6 Parser Test 18
5 Software Architektur **19**
5.1 Architekturpattern 19
5.2 Programmierprinzipien 22
5.3 Code 23
5.4 Aufbau 25
5.5 Build 25
5.6 Schichtenmodell 26
5.7 Source Code 27
5.8 Automatisches Testverfahren 28
5.9 Besonderheiten und Erwähnenswertes 31
5.10 SQL Query Testen 36
5.11 Typ Herausfinden 37
5.12 Yapf 41
6 Proof of Concept **43**
6.1 Setuptools 43
6.2 Click 44
6.3 Visual Studio PTVS versus PyCharm 46
6.4 Sequenzdiagramm mit seqdiag 47

7 Technologien 49
7.1 Verwendete Technologien 49
7.2 Lizenzen 51
8 Implementation 52
8.1 Command Line Interface 52
8.2 Komponenten Diagramm 55
8.3 Klassendiagramme 56
8.4 Datenbank 77
8.5 Ablauf 78
9 Testplan 100
10 Code Analyse 118
10.1 Testabdeckung 118
10.2 Travis 119
10.3 AppVeyor 119
10.4 Github 120
10.5 Git Statistik 120
11 Zusammenfassung 124
11.1 Ausblick 124
11.2 Ergebnisse 126
12 Verzeichnisse 127
12.1 Abbildungsverzeichnis 127
12.2 Tabellenverzeichnis 128
13 Literatur und Quellenverzeichnis 129
14 Danksagung 131
15 Anhang 132
15.1 Persönlicher Bericht 132
15.2 Projekt Management 133
15.3 Testbericht 137
16 Separate Dokumente 143

1 Management Summary

1.1 Ausgangslage

Die Bachelorarbeit befasst sich mit der Planung und Umsetzung eines Code Generator für Plaso als Command Line Interface Applikation.

Für Security Engineers welche unerfahren in der Programmierung sind, soll ein Code Generator zu Verfügung stehen, mit welchem sie SQLite Plugins für Plaso generieren können. Die meisten neuen Anfragen für weitere Plugins handeln sich um solche für SQLite Datenbanken. Diese Plugins werden von dem Tool Log2timeline von Plaso verwendet um Informationen aus einer SQLite Datenbank zu parsen.

Die Command Line Interface Applikation begleitet die User interaktiv durch den Generierungsprozess und unterstützt den User durch intensive Input Validierung und Hilfsinformationen, um mit wenig Eingaben ein funktionierendes Plugin für Plaso zu generieren.

1.2 Vorgehen und Technologien

Die Entwicklung wurde grob in eine Analyse-, Architektur-, Entwicklungs- und Testphase unterteilt, in welcher hauptsächlich die beschriebene Arbeit geleistet wurde. Am Ende der Arbeit wurde noch Zeit für die Dokumentation und allfällige Bugfixes eingeplant.

In der Analyse- und Architekturphase wurden jeweils die Anforderungen mit dem Auftragsgeber und dem Betreuer besprochen, sowie die grobe Architektur definiert.
Über die ganze Arbeit hinweg wurde agil und iterativ gearbeitet. Die Arbeit wurde in wöchentlichen Sprints geplant, und beinhaltete immer Analyse, Architektur, Entwicklung und Testen.

Die iterative Arbeit ermöglicht einen einfachen Start in die Entwicklung mit einer groben Architektur, welche über die Zeit mehrere Verbesserungen durchlief und zu einem komplexen gut strukturierten Projekt wuchs.
Nach jedem Sprint wurde dem Auftragsgeber der aktuelle Stand vorgestellt, die nächsten Features priorisiert und vorhandene Codereviews besprochen, damit der Code den erwarteten Standards entspricht.

Somit vergrösserte sich das allgemeine Wissen über Plaso, sowie den Aufbau der Plugins fortlaufend.

Entwickelt wurde die Command Line Interface Applikation mit Python 3.6 und für die Datenbank kommt SQLite zum Einsatz.

1.3 Ergebnisse

Das Ergebnis dieser Arbeit ist somit ein Code Generator für Plaso als Command Line Interface Applikation, mit einer Dokumentation der Architektur, sowie eine gründliche Dokumentation der API.

Einige optionalen Features, welche den Rahmen der Bachelorarbeit gesprengt hätten, können dank der guten API Dokumentation und Struktur mit wenig Aufwand auch nachträglich noch implementiert werden. Jedoch wurde mehr implementiert als am Anfang gefordert wurde.

Die Applikation erstellt und editiert für den Benutzer 6 Files und generiert durchschnittlich 600 Zeilen Code.

1.4 Ausblick

Der Code Generator kann durch weitere Module erweitert werden, damit auch Plugins für andere Datenbanken und Files generiert werden können. Auch das SQLite Modul kann noch erweitert werden, da noch keine vollständig automatisierte Codegenerierung stattfindet.

2 Einleitung

2.1 Aufgabenstellung

Die Aufgabe der Bachelorarbeit ist es, ein interaktives Command Line Tool zu erstellen, welches ein SQLite File entgegennimmt, sowie eine beliebige Anzahl von Queries. Daraus soll als Output basierend auf den Input, Code für Plaso generiert werden. Die Architektur soll die Integration von weiteren Arten von Plugins für andere Filetypen erlauben. Auch soll eine Ansammlung von Templates bestehen, welche benutzt werden können um verschiedene Arten von Files zu erstellen. Für diese Templates ist Jinja2 präferiert.

Da es ein Teil des Open Source Projektes ist, wird es die Lizenz von Plaso erben.

Der Code muss die Log2timeline Code Reviews von Plaso überstehen, sowie die Styleguides von Log2timeline befolgen.

2.2 Problemstellung, Vision

Diese Bachelorarbeit wurde durch die Firma Google, von Martin Suess, Joachim Metz und Daniel White aus der Security Abteilung, in Auftrag gegeben.

Das Opensource Tool Plaso wird für die forensische Datenanalyse in ihrem Arbeitsalltag verwendet.

Im Moment wird Plaso unteranderem von unerfahrenen Entwicklern um SQLite Plugins erweitert. Dies braucht jedoch viel manuellen Aufwand von Seiten der Security Engineers sowie Zeitaufwand für die Reviews von den Projektinhabern, da die Code Standards eingehalten werden müssen.

Um die Codebasis schneller zu erweitern braucht es ein Tool welches die Security Engineers bei der Erstellung unterstützt.

2.3 Ziele

Das Ziel dieser Arbeit ist es, eine Command Line Interface Applikation zu erstellen, welches die Security Engineers bei der Erstellung von SQLite Plugins unterstützt und somit auch den Zeitaufwand für Codereviews verringert.

Das Tool soll vom Security Engineer unter anderem das Datenbankschema sowie SQLite Queries entgegennehmen und daraus ein SQLite Plugin für Plaso generieren.

Die Security Engineers sollen nicht mehr mühsam Code zusammen kopieren und sich mit der Entwicklung auseinander setzten müssen, sondern sollen sich mehr auf ihr Fachgebiet konzentrieren können. Dies beschleunigt die forensische Datenanalyse und Vervollständigung von Plaso durch mehr SQLite Plugins.

Die Entwickler von Plaso sollen weniger Zeit für Code Reviews in Beanspruchung nehmen müssen und somit mehr Zeit für Features brauchen können, um Plaso fortlaufend zu verbessern.

2.4 Rahmenbedingungen

Der Benutzer der Command Line Interface Applikation mit dem Namen Plaso Scaffolder, muss das Tool mit Python 3.6 ausführen und kann es mit pip vom Python Package Index installieren.

2.5 Methode / Vorgehen

In diesem Dokument wird auf die Software Architektur und die eingesetzten Technologien für die Entwicklung eingegangen.

3 Plaso

3.1 Geschichte

Um 2000 hat man Timelineanalysen nur mit den Timestamps der Filesysteme durchgeführt.

Es gab ein Perl Skript von Harlan Carvey für die Register mit dem Namen regtime.pl, sowie ein File von Brian Carrier mit dem Namen mactime, welches die Daten mit einem Datumfilter sortierte und in ein CSV Schema brachte.

Um 2009 wurde Log2timeline erstellt, welches sich zuerst um eine Kollektion von Skripts handelte und sich später zu einem Framework erweiterte. [1]

3.2 Definition

Plaso ist die Abkürzung des Frameworks mit dem Namen «Plaso Langar Að Safna Öllu».

Log2timeline ist das hauptsächliche Frontend.

Plaso beinhaltet strukturierte Events, Metadaten, granulares Filtern, parsen von Bildern, parsen von vss, tagging sowie gezielte Kollektionen.

3.3 Ziel

Das Ziel war es, die Analyse von Super Timelines zu vereinfachen und sie einfacher zu erstellen.

Plaso beinhaltet eine Ansammlung von Tools, welche von Security Analysten gebraucht werden.

- Das Tool log2timeline.py extrahiert Daten anhand deren Zeitstempel.
- Das Tool psort.py ist für die Nachbearbeitung zuständig.
- Das Tool plasm.py ist zuständig für das taggen.
- Das Tool pshell.py ist eine IPython shell.
- Das Tool preg.py ist PoC für das Parsen von Register.
- Das Tool pprof.py ist für das Profiling von Runtimes für Devs zuständig.

3.4 Benutzung

Das Tool kann mit ***log2timeline.py [OPTIONS] output_file input_file*** gestartet werden. Und resultiert in einem Dump von Events. Ein Event stellt einen Logeintrag mit einem Timestamp dar. Danach lässt man psort.py laufen um die Daten zu sortieren. Und am Schluss analysiert man sie.

Folgend werden die verschiedenen Tools und deren Ablauf erläutert.

3.4.1 Log2timeline

log2timeline.py storage.dump /mnt/e01/ewf1

Dieses Kommando startet das Log2timeline Programm mit dem Outputfile storage.dump und als Inputfile ein gemountetes Image.

1. Zuerst verarbeitet Log2timeline die Parameter.
2. Danach mounted es das Image oder all die anderen Fileformate die es unterstützt und beginnt Informationen vom System zu sammeln.
3. Dann werden verschiedene Prozesse gestartet.
    a. Ein *Collector* sucht die Files welche Plaso parsen kann.
    b. *Workers* öffnen die Files und versuchen daraus Events zu extrahieren.
    c. *Storage* ist zuständig für das Serialisieren der Events um sie danach auf der Disk zu speichern.

Zu den Worker Tasks:

Zuerst öffnen sie ein File und prüfen danach, ob es ein Parser gibt, welcher in der Lage ist das File zu parsen, um danach die Events zu extrahieren.

3.4.2 Psort

psort.py storage.dump

Plaso speichert alle Daten in einem komprimierten Container. Man muss dann 'psort' benutzen um den Output zu konvertieren. Der Standard von psort dumpt den ganzen Inhalt von CSV zu stdout.

Hilfreich ist es, vor dem Sortieren die Daten zu taggen. Mehr zum tagging unter plasm.

3.4.3 Pinfo

pinfo.py [-v] storage.dump

Mit Pinfo kann man sich die Metadaten vom Plaso Storage File anschauen. Diese Metadaten beinhalten Informationen wie zum Beispiel den Zeitpunkt zu dem das Tool gestartet wurde, welche Parameter benutzt wurden, welche Parsers geladen wurden und die Anzahl Events im Storage File.

Der Parameter -v steht für verbose und retourniert mehr Daten.

3.4.4 Plasm – tagging

plasm.py tag -tagfile=somefile.txt storage.dump

Dieses Kommando startet das Plasm Programm und tagt alle Events welche im File somefile.txt vorgegeben wurden. Dazu werden die Files im storage.dump, welches von Log2timeline erstellt wurde, verwendet.

Mit einem einfachen Definitionsfile kann man seine Events taggen.

Und wenn man sein File getaggt hat, kann man somit danach nach den tags filtern.

psort.py storage.dump "select datetime, … where tag contains TagName and …"

3.5 Fall Beispiel

Das folgende Beispiel demonstriert nur einen Verwendungszweck von Plaso.
Plaso wird häufig gebraucht um herauszufinden, wie und wann ein Computer mit Malware infiziert wurde. Oder auch um herauszufinden, welche Aktivitäten auf einem unbeaufsichtigten Gerät ausgeführt wurden.

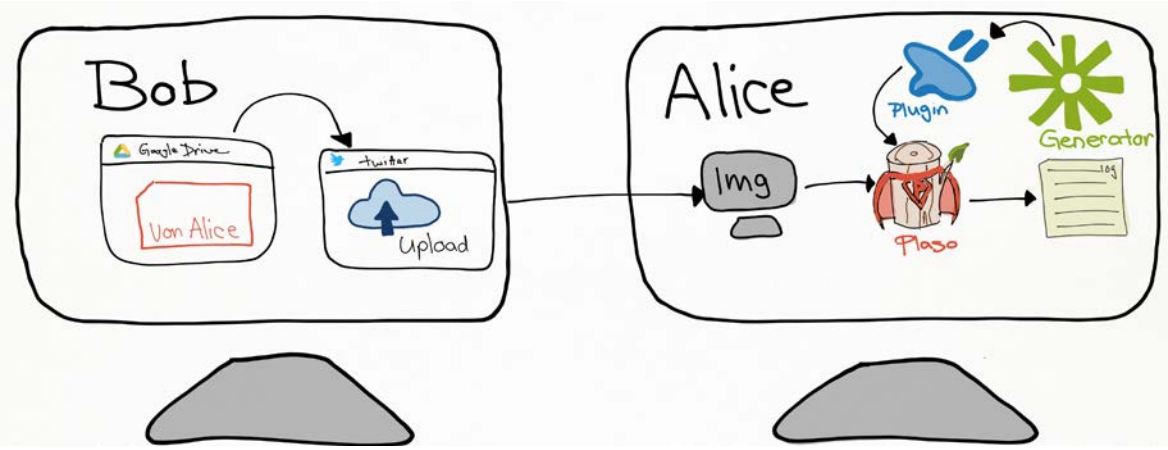


Abbildung 1: Fall Beispiel

Alice möchte wissen, ob Bob ihre Arbeit, welche sie in Google Drive gespeichert hat, geöffnet hat.
Bob verneint es ihr und gibt Alice Zugang zu seinem Gerät.
Dafür analysiert sie mit Plaso zuerst den Browserverlauf, um zu sehen ob das File geöffnet wurde.
Alice sieht einen Eintrag bei welchem ihr File aufgerufen wird und bemerkt, dass Bob danach Twitter aufgerufen hat.
Alice verdächtigt Bob das File über Twitter geteilt zu haben.
Alice möchte nun die SQLite Datenbank von Twitter auf Bobs Gerät analysieren, ob ein Upload Eintrag mit ihrem File existiert.
Da Plaso noch keine Twitter Datenbanken analysieren kann möchte sie selbst Plaso mit einem Plugin dafür erweitern.

Ursprünglich müsste Alice nun dafür 4 Files erstellen und 2 Files editieren, welches ca. 600 Zeilen Code beinhaltet. Danach gäbe es ein Codereview, welches durch die starken Richtlinien von Plaso bestimmt einmal abgelehnt wird. Bis Plaso das neue Plugin von Alice beinhaltet, hat Bob schon längst die Arbeit von Alice verbreitet.

Nun sieht Alice jedoch, dass ein Code Generator für SQLite Plugins existiert. Als Security Engineer kennt Alice sich mit SQL aus, und weiss auch, welche Daten sie von der Twitter Datenbank sehen möchte. Somit kann sie innerhalb weniger Minuten ein Plugin für Twitter erstellen. Nach wenigen manuellen Anpassungen wird es auch schon in Plaso gemerged, da es den Richtlinien entspricht.

Nun analysiert Alice mit Plaso die Twitter Datenbank und sieht, dass Bob das File auf Twitter geladen hat. Bob sieht sich geschlagen. Und Plaso wurde mit einem weiteren nützlichen Plugin erweitert.

4 Plugin Aufbau

Wie in der Anforderungsspezifikation ersichtlich, besteht das Plugin aus Änderung bei 7 Files[1].

- plaso/formatters/__init__.py — Init File für Formatters
- plaso/formatters/*pluginname*.py — Formatter für Plugins
- plaso/parsers/sqlite_plugins/__init__.py — Init File für Parsers für SQLite
- plaso/parsers/sqllite_plugins/*pluginname*.py — Parser
- test_data/*pluginname*.db — Testdatenbank
- tests/formatters/*pluginname*.py — Tests für de Formatter
- tests/parsers/sqlite_plugins/*pluginname*.py — Tests für den Parser

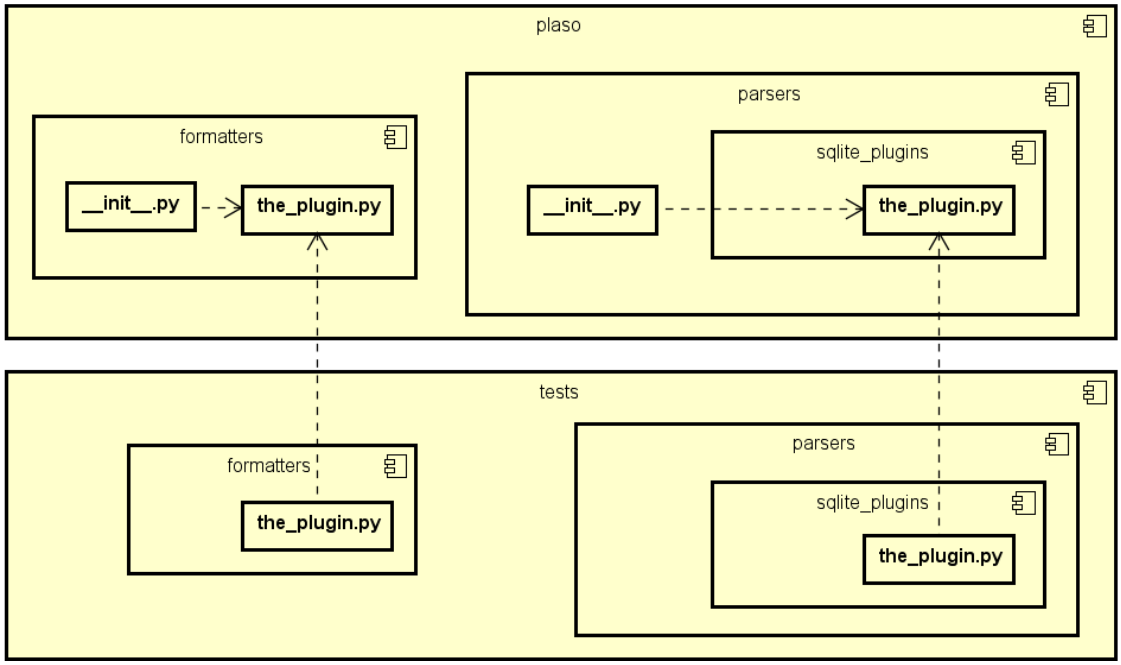


Abbildung 2: Komponenten Diagramm Plugin mit Astah[2]

Die Grundbausteine eines Plugins für Plaso sind das Parser und das Formatter File.
Der Parser parst die Daten aus der Datenbank und der Formatter stellt die Daten für Menschen lesbar dar.

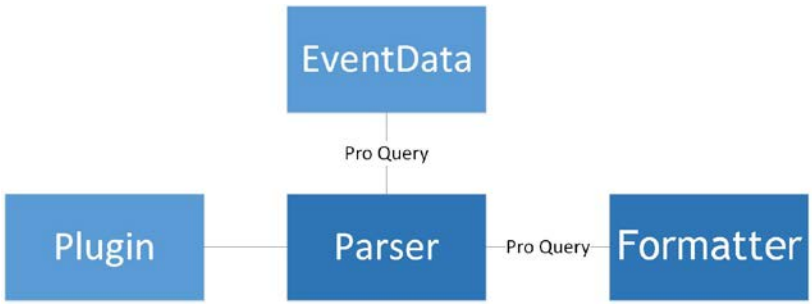


Abbildung 3: Plaso Plugin Aufbau

Pro Parser existiert eine Plugin Klasse. Diese Plugin Klasse erbt von `interface.SQLitePlugin`, da der Scaffolder SQLite Plugins generiert.

[1] Ersichtlich aus dem Codereview für ein Twitter iOS Plugin [4]
[2] Astah ersichtlich unter: [12]

4.1 Eingaben vom User

Der Benutzer muss dem Tool verschiedene Daten mitteilen, welche für die Generierung der Files benutzt werden.

Datenbankschema	Das SQLite Datenbankschema für das Plugin.
Queries	Die SQLite Queries für welche Events erstellt werden sollen.
Query Name	Der Namen des Queries
Timestamps	Die Timestamps pro Query, welche Events widerspiegeln.
Plugin Name	Der Name des Plugins.
Attribute mit Anpassungen	Die Attribute welche Spaltennamen des Queries sind, welche weitere Anpassungen brauchen.

4.1.1 Datenbankschema Validierung

Da das Format des Files einen SQLite Datenbankschema Aufbau haben muss, wird zuerst validiert ob ein File an jenem Pfad besteht und ob es das korrekte Format besitzt.

4.1.2 Queries Validierung

Zum schnellen entdecken der Fehler werden die Queries validiert.
Ausserdem darf der User nur SELECT Statements verwenden, damit nicht versehentlich produktive Daten gelöscht werden.
Damit für die Spalte der Alias und die dazugehörende Tabelle korrekt geparst werden kann, wird dem User vorgeschrieben keine speziellen Sonderzeichen wie Klammern und Anführungszeichen zu verwenden und Aliase für gleiche Spaltennamen zu definieren

4.1.3 Query Name Validierung

Da der Query Name für Teile des Klassennamens benötigt wird, darf er keine Sonderzeichen beinhalten und muss mit einem Kapitalbuchstaben beginnen.

4.1.4 Timestamps Validierung

Da ein Timestamp eine Spalte aus dem Query ist, werden dem User nur solche Namen erlaubt.
Auch wird dem User vorgeschlagen Spalten welche ‘time’ oder ‘date’ im Namen haben als Timestamp hinzuzufügen.

4.1.5 Plugin Name Validierung

Da der Plugin Name auch für den Filenamen verwendet wird, wird dem Benutzer nur erlaubt eindeutige Namen pro Directory zu vergeben.

Die Konvention verlangt, dass der Filename kleingeschrieben wird, als Separator ist ‘_’ zu verwenden.

4.1.6 Attribute mit Anpassungen Validierung

Da ein Attribut welches Anpassungen braucht eine Spalte aus dem Query ist, wird dem User wie bei den Timestamps nur solche Namen erlaubt.
Auch wird dem User nicht erlaubt Attribute zu definieren, welche er schon als Timestamps markiert hat, da diese anders behandelt werden.

4.2 Parser

Der Parser ist zuständig die Daten aus dem Datenbank zu parsen und für jedes Resultat aus dem Query ein Event zu erstellen.

4.2.1 Event Data

Pro SQL Query existiert eine Event Datenklasse, welche von `events.EventData` erbt.

Pro Event Datenklasse existieren folgende Konstanten:

`DATA_TYPE: str`	Die Datentyp Konstante, welche hilft die Event Datenklassen einzuordnen.

Auch wird beim Initialisieren pro Spalte, welche in der Query Resultat Zeile steht, eine Variable definiert und auf `None` gesetzt.
Somit existiert eine Datenklasse für genau diese Query, welches einen Event widerspiegelt.

4.2.2 Plugin Klasse

Pro Plugin Klasse existieren folgende Konstanten:

`NAME: str`	Der Name des Plugins.
`DESCRIPTION: str`	Eine Beschreibung für den Parser.
`QUERIES: [str]`	Alle Queries, welche für das Plugin existieren.
`REQUIRED_TABLES: [str]`	Alle Tabellen, welche in der Datenbank existieren.

Tabelle 1: Plugin Klassen Konstante

Um die Queries richtig zu parsen, existiert in der Plugin Klasse pro Query eine ParseRow Methode
In dieser Methode wird die resultierende SQLite Zeile eines ResultSet, welche man vom Query zurückbekommt, in die Event Daten für dieses Query geschrieben.

```
event_data = TwitterIOSContactEventData()
event_data.description = row['description']
…
```

Pro Query existiert 1 bis n Zeitstempel, damit die Daten in die Timeline eingefügt werden können.
Für jeden dieser Timestamps wird ein Event mit Hilfe der Event Daten erstellt.
Dazu wird der Timestamp zuerst in einen universell verwendeten Timestamp umgewandelt und danach daraus ein Event erstellt.

```
timestamp = row['createdDate']
if timestamp:
  # Convert the floating point value to an integer.
  timestamp = int(timestamp)
  date_time = dfdatetime_posix_time.PosixTime(timestamp=timestamp)
  event = time_events.DateTimeValuesEvent(
      date_time, eventdata.EventTimestamp.CREATION_TIME)
  parser_mediator.ProduceEventWithEventData(event, event_data)
```

4.2.3 Verwendung der Daten

Um den Parser zu generieren, werden vom User der Name des Plugins, das Datenbankschema, die Queries, der Query-Name, sowie die Timestamps verwendet.

Plugin Name wird für folgende Werte benötigt:

- Für den Namen des Parser File.

- Für den ersten Teil des Namens für alle Klassen im Parser.
- Für Teile der Beschreibung.
- Für die ersten Teile für den `DATA-TYPE`.

Datenbankschema wird für folgende Werte benötigt:

- Für den Datentyp in der Attributbeschreibung für die Event Daten.
- Für die REQUIRED_TABLES in der Plugin Klasse.
- Für das Ausführen der Queries um die Daten daraus zu lesen.

Queries werden für folgende Werte benötigt:

- Für die QUERIES in der Plugin Klasse.
- Die Namen der Attribute werden verwendet für:
  - Die Variablen in der Event Daten Klassen.
  - Die Variablen Beschreibung der Event Daten.
  - Die Variablen in der ParseRow Methode.
  - Das Holen der Daten aus dem ResultSet in der ParseRow Methode.
- Die tangierten Tabellen des Queries werden für einen Vorschlag für zweiten Teil des Namens der Event Datenklasse benötigt.

Query Name wird für folgende Werte benötigt:

- Für den Namen des Queries in der Plugin Klasse.
- Für den mittleren Teil des Namens der ParseRow Methode in der Plugin Klasse.
- Für die Klassenbeschreibung.
- Für den mittleren Teil des Namens für die Event Datenklasse

Timestamps werden für folgende Werte benötigt:

- Für die Timestamps in der ParseRow Klasse.

4.2.4 Manuelle Anpassungen

Da sehr viele verschiedene Arten von time_events existieren, muss der Benutzer diesen für die Timestamps nachträglich definieren.

Da es unsinnig ist eine detaillierte Beschreibung für die Attribute zu generieren, muss der Benutzer diese nachträglich erfassen, da der Codestyleguide von Plaso dies verlangt.

4.3 Formatter

Der Formatter ist zuständig, die Daten welche vom Parser kommen, für Menschen lesbar darzustellen.

Pro Query existiert eine Formatter Klasse, welche von interface.ConditionalEventFormatter erbt. Dieser Formatter ist die übliche und meistverwendete Art von Formatter für SQLite Plugins.

Pro Formatter Klasse existieren folgende Konstanten.
Anhand dieser wird der Log Output für Log2timeline erstellt und in das Dump File geschrieben.[3]

`DATA_TYPE: str`	Die Konstante des Datentyps, welche hilft die Event Daten Klassen einzuordnen.
`FORMAT_STRING_PIECES: [str]`	Dieses String Array beinhaltet alle Attribute des Parsers für das Query, sowie eine Beschreibung für die Spalten.
`FORMAT_STRING_SHORT_PIECES: [str]`	Dieses String Array stellt die verkürzte Version von FORMAT_STRING_PIECES dar. Falls der Output länge als 80 Zeichen ist, wird er nach 77 Zeichen abgeschnitten und mit … fortgesetzt. Dies hat historische Gründe. Man möchte das der Output auf 80 Zeichen Platz hat.
`SOURCE_LONG: str`	Das ist die lange Beschreibung des Formatters und besteht aus dem Namen des Plugins sowie des Queries.
`SOURCE_SHORT: str`	Das ist die kurze Beschreibung des Formatters und besteht aus dem Namen des Plugins.
`_ATTRIBUTNAME``: {object:str}`	Falls es Attribute gibt, welche weitere Anpassungen brauchen, gibt es ein Dictionary welches den Wert zu einem lesbaren String mappt.

Tabelle 2: Formatter Klassen Konstante

Falls es im Query für welches der Formatter existiert, ein bis mehrere Attribute gibt, welche Anpassungen nötig haben, wird eine GetMessage Methode erstellt.
In dieser Methode, werden alle Werte der Attribute welche Anpassungen brauchen auf den Wert gemappt, welche in *_ATTRIBUTENAME* Dictionary angegeben wurde, oder `u'UNKNOWN'`[4] falls zum Key kein Wert gefunden wurde.

Am Ende des Formatter Modul werden alle Formatter im Formatter Manager registriert.

4.3.1 Verwendung der Daten

Um den Formatter zu generieren, werden vom Benutzer der Name des Plugins, das Datenbankschema, die Queries, die Namen der Queries, die Timestamps sowie die Attribute mit Anpassungen verwendet.

Plugin Name wird für folgende Werte benötigt:

- Für den Namen des Formatters File.
- Für den ersten Teil des Namens für alle Formatter.
- Für Teile der Beschreibung.
- Für die ersten Teile für den DATA-TYPE.

[3] Siehe Abschnitt: 3.4.1

[4] Da Plaso noch nicht Python 3 unterstützt, bei welchem der Standard String Typ Unicode ist, müssen für den generierten Code alle Unicode Strings mit *u* dekoriert werden.

Datenbankschema wird für folgendes benötigt:

- Für das Ausführen der Queries um die Daten daraus zu lesen.

Queries werden für die Namen der Attribute verwendet, welche für folgende Werte benötigt werden:

- Für die FORMAT_STRING_PIECES.
- Für die FORMAT_STRING_SHORT_PIECES.
- Für den letzten Teil des SOURCE_LONG Strings.
- Für das Zuweisen des neuen Werte des Dictionary in der GetMessages Methode.
- Für den Namen der Konstante mit dem Dictionary, falls das Attribut Anpassungen braucht.

Query Name wird für folgende Werte benötigt:

- Für den zweiten Teil des Namens für die Formatter Klasse.
- Für die Klassenbeschreibung.

Timestamps werden für folgende Werte benötigt:

- Damit man weiss, welche Attribute nicht in den Event Daten sind.

4.3.2 Manuelle Anpassungen

Falls man Attribute hat, welche Anpassungen brauchen, muss man die Werte für das Dictionary erfassen.
Falls man einen die FORMAT_STRING_SHORT_PIECES anpassen möchte, kann man beliebige Zeilen löschen.
Ansonsten werden keine manuellen Anpassungen benötigt.

4.4 INIT

Für die Formatter sowie für die SQLite Parsers existiert ein __init__ File, welches editiert werden muss, oder erstellt falls es noch nicht existiert.
Es wird ein simples Import Statement für das neu erstellte Modul erstellt.

4.4.1 Manuelle Anpassungen

Für die Init Files werden keine manuellen Anpassungen benötigt.

4.5 Formatter Test

Es besteht ein Formatter Test File. Für jeden Formatter Klasse braucht es eine Testklasse.
Pro Testklasse gibt es 2 Test Methoden.
Die Methode testInitialization testet die Initialisierung eines Formatters.
Die Methode testGetFormatStringAttributeNames testet die Format String Attribute.

4.5.1 Verwendung der Daten

Um den Formatter Test zu generieren, werden vom Benutzer der Name des Plugins, das Datenbankschema, die Queries, der Name der Queries sowie die Timestamps verwendet.

Plugin Name wird für folgende Werte benötigt:
- Für den Namen des Formatters Test File.
- Für den ersten Teil des Namens für alle Formatter Test Klassen.
- Für Teile der Beschreibung.
- Für den ersten Teil des Namens für den Formatter, der zu initialisieren ist.

Datenbankschema wird für folgende Werte benötigt:
- Für das Ausführen der Queries um die Daten daraus zu lesen.

Timestamps werden für folgende Werte benötigt:
- Damit man weiss, welche Attribute nicht in den Event Daten sind.

Queries werden für folgende Werte benötigt:
- Die Namen der Attribute für die erwarteten Attribute Namen.

Query Name wird für folgende Werte benötigt:
- Den zweiten Teil des Namens für die Formatter Testklasse.
- Den zweiten Teil des Namens für den Formatter, der zu initialisieren ist.
- Die Klassenbeschreibung.

4.5.2 Manuelle Anpassungen

Es werden keine manuellen Anpassungen benötigt.

4.6 Parser Test

Es besteht ein Parser Test File mit einer Testklasse.
Der Test besteht aus mehreren Abschnitten. Pro Query sowie pro Event gibt es einen Source Code Abschnitt, welcher sich nur durch die verschiedenen Event Daten unterscheiden jedoch die gleiche Logik besitzt.

4.6.1 Verwendung der Daten

Um den Parser Test zu generieren, werden vom Benutzer der Name des Plugins, das Datenbankschema, die Queries sowie die Timestamps verwendet.

Plugin Name wird für folgende Werte benötigt:
- Für den Namen des Parser Test File.
- Für den ersten Teil des Namens für alle Parser Testklassen.
- Für Teile der Beschreibung.
- Für den ersten Teil des Namens für den Plugin, der zu Initialisieren ist.

Datenbankschema wird für folgendes benötigt:
- Für das ausführen der Queries um die Daten daraus zu lesen.
- Für das initialisieren des storage_writer, welcher das Datenbank File mit Hilfe des Plugins parst.
- Um den Datentyp der Event Daten herauszufinden, um das erwartete Resultat mit dem, welches das Plugin zurückgibt, zu vergleichen.
- Um die Anzahl Events welche im Datenbankschema existieren, herauszufinden.

Timestamps werden für folgende Werte benötigt:
- Damit man weiss, welche Attribute nicht in den Event Daten sind.
- Für das Erstellen eines Test Abschnitt pro Query pro Timestamp.
- Um dem User Hinweise zu geben, für welchen Timestamp aus der Datenbank er den formatierten Timestamp definieren muss.

Queries werden für folgende Werte benötigt:
- Die Namen der Attribute, welche mit den Daten aus der Datenbank verglichen werden sollen. Somit die Attribute der Test Event Datenklasse.

4.6.2 Manuelle Anpassungen

Der User muss nachträglich den formatierten Timestamp eintragen, sowie der Typ des Events, welcher er auch im Parser angegeben hatte.
Falls er Felder mit Anpassungen erstellt hat, muss er den Inhalt des Dictionary anpassen.
Es wird eine Hilfsfunktion generiert um den erwarteten Event aus dem Event Array zu lesen, diese soll der Benutzer löschen und mit der ID ersetzten, welche er bei der ersten lokalen Ausführung als Resultat bekommt.

5 Software Architektur

5.1 Architekturpattern

5.1.1 MVC

Für Konsolenapplikationen gibt es keine typischen Patterns.
Deshalb wurden Patterns als Inspiration benutzt, welche eher in die Webentwicklung und im Desktop-Berreich benutzt werden.

Als grundsätzliche Abtrennung der Komponenten wurde als Inspiration das MVC Pattern verwendet.

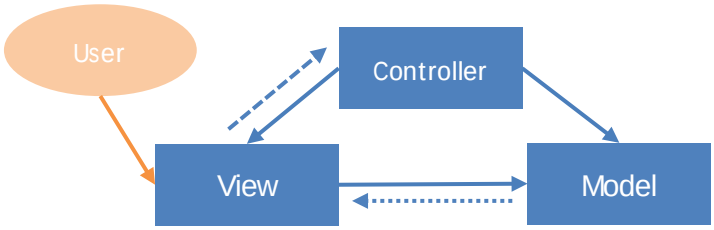


Abbildung 4: MVC simple

Die View ist die Präsentationsschicht für die Darstellung der Daten und kann bei einer Konsolenapplikation am ehesten mit dem Command Line Interface in Verbindung gebracht werden, da dieses Daten vom Model darstellt.
Das Model ist der Data Access Layer, welcher beim MVC Pattern Datenbank Operationen ausführt und auch für die Geschäftslogik zuständig ist.
Der Controller kontrolliert den Austausch zwischen der View und dem Model, es enthält den Input vom User, kommuniziert mit dem Model und entscheidet was anzuzeigen ist. Auch wenn in einer Konsolenapplikation keine dynamische View besteht, ist das Prinzip das Gleiche.

Zusätzlich für die Abtrennung der Erstellung der Output Files, wurde als Inspiration das MVVM Pattern benutzt.

Abbildung 5: MVVM

Bei dem typischen Model View ViewModel bestehen zwischen der View und dem ViewModel verschiedene DataBindings. Diese sind hier nicht beidseitig, jedoch besteht zwischen den Templates und den Daten ein AttributeBinding.
Das Model stellt somit die anzuzeigenden Daten dar.
Die View stellt die Präsentation Ansicht dar, welches hier die generierten Files sind.
Was beim MVVM die ViewModels sind, sind hier die Templates für die Output Files.

Wie beim MVVM Pattern informiert das Model das ViewModel über die Datenänderung. Jedoch nur einmal beim Generierungsvorlauf.
Das Model gibt die aufbereiteten Daten weiter an das ViewModel via AttributeBinding.
Die View, welches die generierten Files sind, enthält die Daten des ViewModels.

Kombiniert man die Idee von MVC und MVVM entsteht die folgende Architektur.

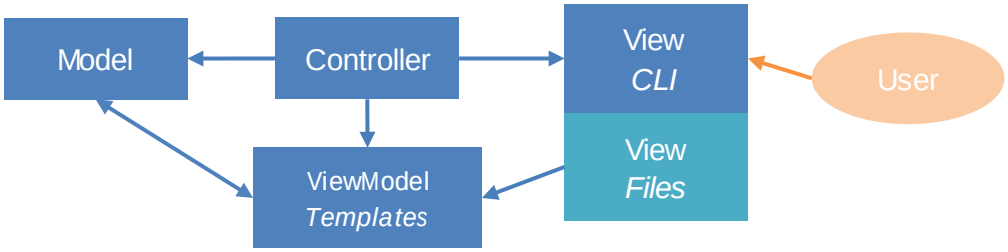


Abbildung 6: MVC + MVVM

Einerseits ist das Prinzip des MVC Patterns vorhanden für das Command Line Interface, welches die View vom Model und Controller trennt.
Zusätzlich besteht eine Abstraktion für den Output, welches die Files mit Hilfe der Templates, sowie den Datenmodel Klassen, erstellt.

Da somit die Daten, die Geschäftslogik sowie der Zugriff auf die Datenbank noch nicht abgetrennt sind, wird für eine schöne Abstrahierung die Applikation weiter abstrahiert mit den gängigen Schichten.

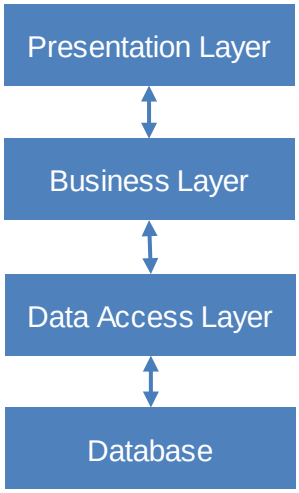


Abbildung 7: Schichtenarchitektur

Der Presentation Layer ist für die Präsentation der Daten verantwortlich und beinhaltet die View und den Controller.
Der Business Layer ist für die Business Logik zuständig.
Der Data Access Layer ist für den Datenbankzugriff verantwortlich.

Alle Patterns zusammen ergeben die Projekt Architektur des Plaso SQLite Plugin Scaffolder mit Inspiration von MVC, MVVM sowie der Schichtenarchitektur.

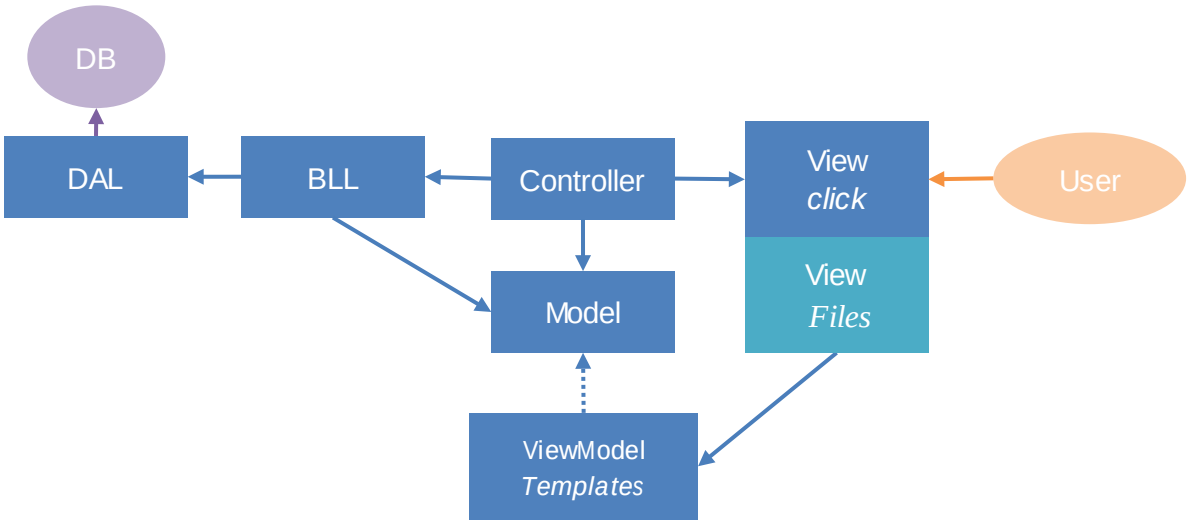


Abbildung 8: Finale MVC Architektur

All diese Komponenten sind lose gekoppelt und austauschbar.

5.2 Programmierprinzipien

Die Programmierprinzipien werden durchgehend eingehalten.

5.2.1 DRY – Don't repeat yourself

Falls Code doppelt gebraucht wird, wird er in eine neue Methode ausgelagert.

5.2.2 KISS – Keep it simple stupid

Der Code wurden simple gehalten, Statements welche auf einer Zeile geschrieben werden könnten, werden aufgeteilt.
Lange komplexe Methoden werden verhindert und in private Hilfsmethoden unterteilt. Dies erleichtert das Debuggen, welches sich sonst mit Python sehr schwierig verhaltet.

5.2.3 YAGANI – you ain't gonna need it

Es wird nur umgesetzt was auch gebraucht wird.

5.2.4 Verständlichkeit

Der Code beinhaltet keine Magic Numbers oder Variablen als Einzelzeichen.

5.2.5 Wartbarkeit

Der Code ist sauber formatiert, beinhaltet keine Pylint Warnungen und ist somit mit Pylint überprüfbar. Die Wartbarkeit ist durch eine sinnvolle API Dokumentation sichergestellt.

5.2.6 Überprüfbarkeit

Der Code ist durch die hohe Testabdeckung mit Unittests, Integrationstests sowie automatisierten End-to-End Tests vollständig überprüfbar.
Um die Portierbarkeit zu garantieren und plattformunabhängig zu sein wird dies mit automatischen Tests mit Python 3.6 auf Linux mit Ubuntu 14.04 sowie auf Windows mit dem Visual C++ Compiler 14.0 sichergestellt.

5.2.7 Lose Koppelung

Es besteht eine sehr lose Koppelung, da keine Module direkt ein anderes Modul ansprechen.
Die Kommunikation zwischen den Modulen ist über Basisklassen, welche in Python die Rolle von Interfaces übernimmt, geregelt. Falls ein Modul ein anderes Modul ansprechen möchte, wird eine Klasse des Typs der Basisklasse erwartet, und über den Konstruktor mitgegeben. Oder selten wird das Modul auch als Parameter der Methode überreicht, falls das Modul in der Klasse nur für eine spezifische Methode verwendet wird.
Da es in Python sehr ungewöhnlich ist Dependency Injection oder Inversion of Control Frameworks sowie Container zu benutzten, wurde die lose Koppelung erhöht indem die Klassen im Controller initialisiert werden und somit die Schichten unabhängig voneinander austauschbar sind.

5.2.8 Hohe Kohäsion

Es besteht logische Kohäsion durch die Aufteilung in Klassen sowie Methoden nach ihrer Logik.
Es besteht zeitliche Kohäsion durch die verschiedenen Schichten, welche durch Module aufgeteilt wurden.
Es besteht kommunikative Kohäsion dadurch, dass die einzelnen Teile des Moduls auf gemeinsame Daten zugreifen.
Es besteht sequentielle Kohäsion, da die Ausgabe der validierten Daten die Eingabe für die Generation sind.

5.3 Code

5.3.1 Code Analyse mit Pylint

Um die Code Qualität sicherzustellen wird Pylint[5] verwendet.
Da die Standard Konfiguration von Pylint, welche auf PEP8 basiert von dem Codestyleguide von Plaso[6] abweicht, wird Pylint mit dem pylintrc File konfiguriert. Da Plaso bereits ein pylintrc File besitzt, wird dieses übernommen.

Es gibt einige Informationen und Warnungen welche ignoriert werden, da sie im verwendeten Kontext sinnfrei sind, diese sind nachfolgend erklärt.
Ignorieren kann man diese mit einer Kommentarzeile im Format: `#pylint: disable=Warnung`.
Bei jedem Verwenden einer Kommentarzeile um eine Information, Warnung oder Error zu ignorieren, ist dies im Code auch dokumentiert.

protected-access
Da man in Python keine privaten sowie protected Methoden oder Variablen ansprechen sollte, welche mit einem vorangehenden _ gekennzeichnet werden.
Jedoch werden in den Unittests oftmals private Methoden getestet. Dies ist bei den Tests keinen Fehler, da es gut ist komplexere Methoden als alleinstehende Komponente zu testen und somit White-Box-Tests durchzuführen.

anomalous-backslash-in-string
Bei den End-to-End Tests wird Pexpect verwendet. Da Pexpect ein Regex erwartet, müssen alle verwendeten Sonderzeichen im String escaped werden.

no-member
Pylint erkennt manche Methoden von sqlite3 nicht, wie zum Beispiel connect, Error und Warning. Da diese jedoch eindeutig bei sqlite3 vorkommen, wird dieser Error ignoriert.
Dieser Fehler ist bei anderen Bibliotheken wie zum Beispiel numpy bekannt.[7]

dangerous-default-value
Bei den Fakes wird zum Beispiel ein leeres Array als default Wert zugewiesen, da dieser Wert erwartet wird, wenn die Methode keine Werte gefunden hat.
Die von Pylint als gefährlich bezeichneten Standard Werte, werden nur in den Fakes verwendet, um Schreibaufwand zu sparen.

unused-argument
Dies kommt nur bei einigen Fakes vor, falls die Methodendefinition Parameter beinhalten, welche in den Methoden nicht verwendet werden. Da der Fake jedoch die gleiche Signatur wie die richtige Implementierung braucht, wird diese Warnung ignoriert.
Ein anderer Fall ist in den Tests für Methoden, welche Click verwenden, bei welchen ein Patch verwendet wird um die Click API zu ersetzten. Hier braucht es ein Argument für den Patch, welcher jedoch selbst nicht verwendet wird.

bare-except
Dies wird nur ignoriert beim Testen der Click API, da der Test bei jedem Fehler fehlschlagen soll.

[5] Pylint ersichtlich unter: [1]
[6] Styleguides von Plaso ersichtlich unter: [2]
[7] numpy Fehler ersichtlich in stackoverflow unter: [17]

redundant-unittest-assert

Dies wird nur ignoriert beim Testen von Methoden die Click verwenden, da hier mehrmals das gleichaussehende Assert-Statement verwendet wird.

missing-docstring

Dies wird nur direkt bei der Verwendung von Click verwendet, beim Main und dem Start File für SQLite. Und zwar werden docstrings bei Click verwendet um Output zu generieren. Da es jedoch keinen Sinn macht, die docstrings im Programmablauf dem User anzuzeigen, haben einige Methoden in diesen Files keine Kommentarzeilen.

skip-file

Skip File bedeutet, dass das ganze File ignoriert wird.
Dies wird bei einem File getan und ist die einzige Information welche von Pylint geschrieben wird, wenn man Pylint über das gesamte Projekt laufen lässt.

```
************* Module plasoscaffolder.bll.services.sqlite_plugin_helper
I:  3, 0: Ignoring entire file (file-ignored)
```

Das File wird aufgrund eines Fehlers in Pylint ignoriert. Der Fehler ist beschrieben unter: https://github.com/PyCQA/astroid/issues/362. Sobald der Fehler gelöst wird, kann die Pylint Kommentarzeile wieder entfernt werden. Der Fehler ist, dass Pylint eine 'AttributeError' Exception beim Importieren von dem Package functool wirft.

5.4 Aufbau

.src Beinhaltet den Source Code.
.doc Beinhaltet die Konfigurationfiles für die API Dokumentation.
.build Beinhaltet die Buildfiles für den Build.
docs Beinhaltet die generierte API Dokumentation, welche als github Page verwendet wird.

5.5 Build

Das Projekt ist plattformunabhängig und unterstützt somit Linux und Windows. Deshalb ist es sinnvoll die Applikation auf verschiedenen Plattformen automatisch zu testen. Als Continuous Integration Server wird Appvoyer für Windows und Travis für Linux verwendet.

5.5.1 Appvoyer

Auf Appvoyer wird die Applikation unter Windows getestet.
Das Konfigurationsfile für Appvoyer liegt im Root und heisst appvoyer.yml.

1. Alle Packages werden mit Python3.6 von requirements.txt mit pip installiert.
2. Die Tests werden mit run_tests.py ausgeführt

5.5.2 Travis

Auf Travis wird die Applikation unter Linux getestet.
Das Konfigurationsfile für Travis liegt im Root und heisst travis.yml.

1. Alle Packages werden mit Python3.6 von requierements.txt mit pip installiert.
2. Coveralls wird mit pip installiert.
3. Die Tests werden mit run_tests.py ausgeführt
4. Die Tests werden mit run_tests_coverage.py mit Testabdeckung ausgeführt.
5. Falls die Tests fehlerfrei durchgeführt wurden, wird Coveralls gestartet.

5.5.3 Coveralls

Wie schon erwähnt, wird bei einer erfolgreichen Ausführung der Tests mit Travis auf Linux, Coveralls[8] gestartet. Somit wird basierend auf den Tests, welche davor mit Testabdeckung ausgeführt wurden, eine Auswertung bei Coveralls für die Testabdeckung des Codes erstellt.

[8] Siehe Coveralls unter: [19]

5.6 Schichtenmodell

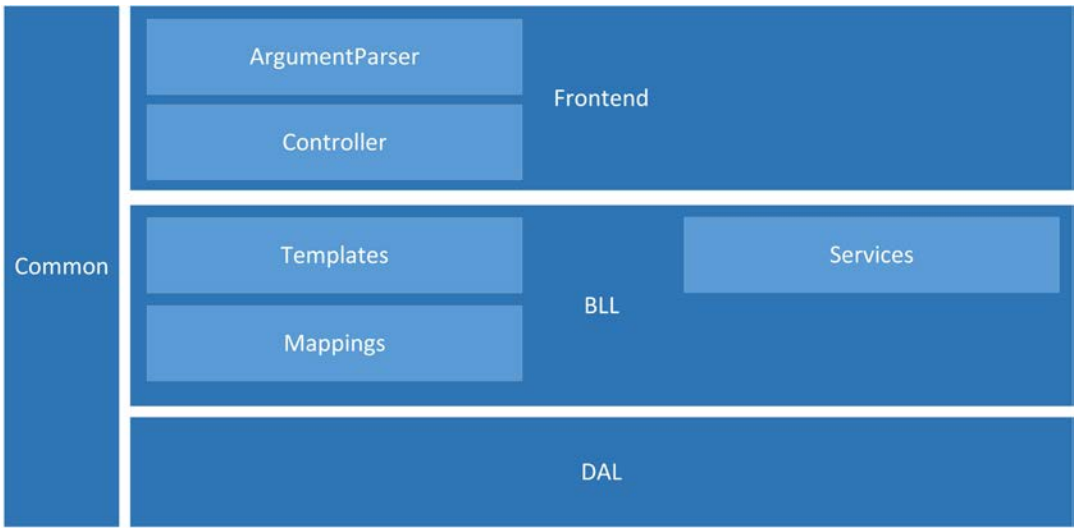


Abbildung 9: Schichtenmodell

Die Architektur wird in 3 Schichten aufgeteilt.

- Frontend
- BLL (Business Logic Layer)
- DAL (Data Access Layer)

Das Front-End beinhaltet den Teil des Argument Parser mit Click und der Controller welche die Aktionen beinhaltet.

Der BLL beinhaltet die Templates, Mappings und weitere Services.

Der DAL beinhaltet die Schnittstellen zu der SQLite Datenbanken.

Der Common Teil beinhaltet gemeinsam genutzte Funktionen über die 3 Schichten.

5.6.1 Involvierte Files

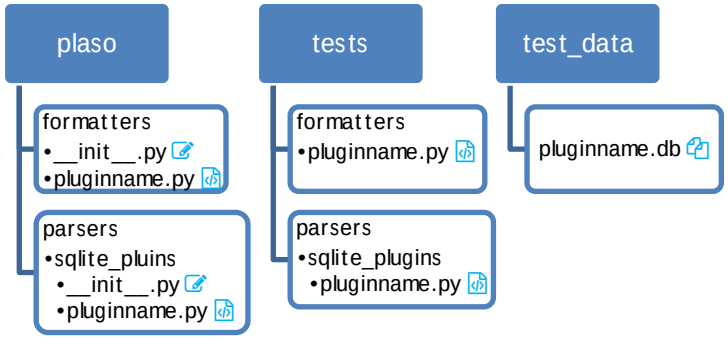


Abbildung 10: Involvierte Files Tree

Erstellen von Files:

- plaso/formatters/*pluginname*.py — Formatter
- plaso/parsers/sqllite_plugins/*pluginname*.py — Parser
- tests/formatters/*pluginname*.py — Test für Formatter
- tests/parsers/sqlite_plugins/*pluginname*.py — Test für Parser

Kopieren von einem File:

- test_data/*pluginname*.db — Testdatenbank

Editieren von zwei Files

- plaso/formatters/__init__.py — Formatter Init File
- plaso/parsers/sqlite_plugins/__init__.py — SQLite Plugin Init File

5.7 Source Code

Der Source Code ist im Ordner src vorhanden.

5.7.1 src

plasoscaffolder	Ordner für die Implementation des Plaso Scaffolder.
tests	Ordner für die Tests des Plaso SQLite Plugin Scaffolder.
setup.py	Setup Files, welches mit 'python setup.py install' die Applikation installiert.
run_tests.py	File, welches mit 'python run_tests.py' ausgeführt werden kann und alle Tests, ausser die Tests im Ordner end_to_end_tests, ausführt.
run_test_for_coverage.py	File, welches mit 'python run_tests.py' ausgeführt werden kann und alle Tests, ausser die Tests in den Ordner end_to_end_tests und integrationstest, ausführt.
requirements.txt	File, welches mit 'pip install -r requirements.txt' ausgeführt werden kann und somit alle abhängigen Packages für das Projekt mit pip installiert.
pylintrc	Das Konfigurationsfile für Pylint mit welchem man die Programmierrichtlinien überprüfen kann.

Tabelle 3: Code Struktur unter src

5.7.2 src/plasoscaffolder

Unter src/plasoscaffolder befindet sich die Implementation des Plaso Scaffolder.

bll	Ordner für den Business Logic Layer.
bll/**mappings**	Ordner für Klassen welche die Daten der Datenmodels zu einem Dictionary mappt und ein Template rendern.
bll/**services**	Ordner für Services, welche aufgerufen werden um das Plugin zu generieren.
bll/**templates**	Ordner für die Jinja2 Templates um die 6 Files zu generieren.
common	Ordner für die Files, welche über mehrere Schichten und verschiedene Plugin Scaffolder die folgen können, verwendet werden kann.
dal	Ordner für den Data Access Layer. Nur Methoden in diesem Ordner greifen auf die Datenbank zu.
frontend	Ordner für das Frontend. Nur hier wird der User mit Hilfe von Click aufgefordert Informationen einzugeben.
fronted/**controller**	Ordner, der den Controller für die verschiedenen Plugin Scaffolder beinhaltet.
fronted/**main.py**	Startpunkt der Applikation.
fronted/**sqliteplugin**	Order für den Startpunkt der Applikation für SQLite Plugins.
model	Ordner für die Daten Modelklassen.
style.yapf	Konfigurationsfile für Yapf welches für die Tests verwendet wird.
dependencies	Die verwendeten Dependencies und deren Version, wie es in Plaso dokumentiert wird, um die Integration des Projektes in Plaso zu vereinfachen.

Tabelle 4: Code Struktur unter src/plasoscaffolder

5.7.3 src/tests

Unter src/tests befindet sich die Tests für den Plaso SQLite Plugin Scaffolder.

bll	Ordner für Tests zu den Files unter plasoscaffolder/bll.
common	Ordner für Tests zu den Files unter plasoscaffolder/common.
dal	Ordner für Tests zu den Files unter plasoscaffolder/dal.
frontend	Ordner für Tests zu den Files unter plasoscaffolder/fontend
model	Ordner für Tests zu den Files unter plasoscaffolder/model.
other	Ordner für Tests zu den Files welche sicht nicht unter plasoscaffolder/* befinden.
integrationtests	Ordner für die Integrationstests, welche über alle Layer ausser dem Frontend laufen.
end_to_end_test	Ordner für die End-to-End Tests, welche über alle Layer laufen und auch manuell durchgeführt werden können.
fake	Ordner für die Fake Klassen, welche die Basisklassen implementieren und für die Tests benötigt werden.
test_database	Ordner für die Datenbanken, welche für die Tests verwendet werden.
test_helper	Ordner für die Hilfsklassen für die Tests.
test_template	Ordner für die Jinja2 Templates, welche für die Tests verwendet werden.
style.yapf	Konfigurationsfile für Yapf, welches für die Tests verwendet wird.

Tabelle 5: Code Struktur unter src/tests

5.8 Automatisches Testverfahren

5.8.1 Unittest

Der Code wird mit Unittests gründlich getestet.
Das Package unittest ist in der Standardbibliothek von Python vorhanden.[9]
Unteranderem ist es bereits eine Abhängigkeit von Plaso und sollte deshalb als Unittest Framework verwendet werden.

Ein Testcase ist eine Subklasse von unittest.TestCase. Damit die Testmethoden gefunden werden können, beginnen sie mit 'test'.
Damit die Tests ausgeführt werden können beinhalten sie ein Main.
Mit den implementierten Methoden setUp() und tearDown() kann man repetitive Arbeiten auslagern.

5.8.2 Faking

Um eine gut testbare und entkoppelte Applikation mit Python zu verwirklichen werden abstrakte Basisklassen mit `abc.ABCMeta` erstellt, von welchen geerbt wird. Mit `@abc.abstractmethod` werden die abstrakten Methoden dekoriert.
Für diese abstrakten Klassen existieren Fake Implementationen, welche beim Aufruf von der getesteten Methode als Parameter mitgegeben werden. Somit wird immer nur die gewünschte Funktionalität getestet und der Rest gefakt.
Diese Fakes befinden sich im Projekt unter dem Pfad: src/tests/fake.

[9] unittest.TestCase unter: [14]

5.8.3 Mocking

Wenn private Hilfsmethoden viel Logik beinhalten, und es Sinn macht jene abgekapselt zu testen, werden sie mit MagicMock[10] gemockt. Dies ist jedoch selten der Fall, da alles gefakt wird.

Mocking mit MagicMock ist in Python sehr einfach. Da Python dynamisch typisiert ist, wird die Methode welche gemockt wird mit einer MagicMock Methode ersetzt.
Ein Einsatz von MagicMock ist im File test/frontend/controller/test_sqlite_controller.py ersichtlich.

```
controller.CreateSQLQueryModelWithUserInput = mock.MagicMock(
          return_value=base_sql_query_execution.SQLQueryData(
              data='test', has_error=False, error_message=None))
```

Hier wird der Rückgabewert für die Hilfsmethode definiert, welche eigenständig gründlich getestet wird. Somit fallen viele Branches weg, da der Rückgabewert dieser Methode keine Rolle für den weiteren Ablauf spielt.

Falls die Methode nicht mit einem MagicMock gemockt werden kann, da sie von einer externen Bibliothek stammt, wird Patch eingesetzt.

Ein Einsatz von mock.patch ist zum Beispiel im File test/common/test_output_handler_click.py ersichtlich.

```
from unittest.mock import patch
…
  @patch('click.prompt', return_value='yes')
  def testPromptInfo(self, prompt):
    """test prompt info"""
    result = self.click.PromptInfo('test')
    self.assertEqual('yes', result)
```

Mit dem `@patch` Decorator, kann man Objekte während dem Testen ersetzen.
Man kann `click.prompt` nicht testen, da hier auf eine Usereingabe gewartet wird, deshalb wird dieser Aufruf gemockt und bei jedem Aufruf im Test wird stattdessen ‘yes’ zurückgegeben.

[10] unittest.mock unter: [15]

5.8.4 End-to-End Tests

End-to-End Tests werden in dieser Bachelorarbeit automatisch durchgeführt.
Jedoch könnten sie auch manuell durchgeführt werden.
Dafür wurde ein Testplan erstellt, welcher einzelne Tests beinhaltet, die folgende Struktur haben:

Test X	
Funktionalität	
Vorbedingung	
Abhängigkeiten	
Ablauf	
Erwartete Resultate	

Tabelle 6: Manuelles Testverfahren Vorlage

Jeder End-to-End Test:

- hat eine Nummer.
- beschreibt welche Funktionalität dieser testet.
- beschreibt was die Vorbedingungen für diesen Test sind.
- beschreibt von welchen anderen Tests dieser abhängig ist.
- beschreibt wie der Testablauf aussieht.
- Beschreibt was das erwartete Resultat ist.

Für die Durchführung der End-to-End Tests gibt es noch eine Tabelle, in welcher eingetragen wird, ob die einzelnen Tests erfolgreich waren und im Falle eines Fehlers, was fehlgelaufen ist.
Diese ist im Angang im Kapitel 15.3.2 vorzufinden.

Automatische Durchführung mit Pexpect

Die End-to-End Tests wurden mit der Bibliothek Pexpect[11] erstellt.
Pexpect läuft im Moment nur auf Linux, da es auf interne Funktionen zugreift.
Da man jedoch stdin und stdout von Click nicht überschreiben kann, ist Pexpect die beste Option die End-to-End Tests mit Python zu automatisieren. Falls man die Tests trotzdem mit Windows laufen lässt, stösst man auf eine NotImplementedError Exception mit der Erklärung, dass diese Tests erst für die Linux Plattform implementiert sind.

Pexpect ist sehr einfach zu verwenden.
Mit pexpect.span, kann man ein Kommando ausführen.

```
pexpect.spawn(python path_to_the_main sqlite)
```

Danach kann man mit dem Keyword expect den Output auf einen Regex prüfen. Falls dieser nicht gefunden wird, terminiert das Programm mit einem Fehler.

```
child.expect('frage_zum_pfad')
```

Wenn der String gefunden wurde, wird fortgefahren. Mit sendline, kann man dem Input einen String mitgeben.

```
child.sendline('antwort_zum_pfad')
```

So wird die ganze Applikation interaktiv durchgeführt.

[11] Siehe pexpect Dokumentation unter: [18]

5.9 Besonderheiten und Erwähnenswertes

In diesem Kapitel werden Besonderheiten und Erwähnenswertes zur Architektur notiert.

5.9.1 Logging

Es wird für die Applikation kein Log geschrieben.
Der Grund dafür ist, dass der Input, Output sowie Fehlermeldungen des Programmes auf der Konsole sichtbar sind und somit genügen, um den exakten Programmdurchlauf zu reproduzieren.

5.9.2 Exception Handling

Allgemein werden Exceptions durch viel Input Validierung vermieden.
Erwartete Exceptions (z.b SQL Error) werden am Ort wo sie entstehen abgefangen und danach an geeigneter Stelle behandelt, so dass das Programm normal fortfahren kann. Exceptions werden an den korrekten Ort weitergeleitet, jedoch meist gerade am aufgetretenen Ort durch Rückgabewerte der aufrufenden Methode mitgeteilt.
Für den User ist somit eine Meldung sichtbar und das Programm fragt nach einer anderen Eingabe.

5.9.3 Pfad zu Orten innerhalb des Projektes

Der Pfad zum Template Ordner sowie zum Style File von Yapf werden schon im Frontend im File commands.py erstellt und danach bis zum Modul weitergereicht, bei welchem es verwendet wird.
Der Grund ist, dass die Struktur des Projektes beim Erstellen eines Packages, wie zum Beispiel wheel[12], verändert wird. Damit der Pfad an den korrekten Ort zeigt und das File gefunden werden kann, wird der Ort beim Frontend angegeben. Am Ort wo die Applikation aufgerufen wird, kennt man mit Sicherheit die Architektur.

5.9.4 Parser Test Workaround

Im Parser Test Template ist ein Workaround ersichtlich.
Dieser Workaround wurde am 5. Mai vom Betriebspartner abgesegnet.

Aufbau
Alle Events für alle Timestamps und alle Queries vom Parser befinden sich im gleichen Resultat Array.
Zuerst werden alle Events für den ersten Query, welcher sich im Parser befindet, vorgefunden und danach der zweite.
Somit hat man bei einem Parser mit 2 Queries mit 20 und 25 Events, welche je zwei Timestamps beinhaltet, schlussendlich 90 Events im Resultat vom Parser.

Das Test File für den Parser vergleicht pro Timestamp der Queries ein Event, welcher vom Parser retourniert wird, mit den Daten welche man erwartet. Es wird der Event ausgewählt, welcher für den Timestamp zeitlich als erstes vorkommt.
Somit wird bei einem Query mit zwei angegebenen Timestamps, zwei unterschiedliche Events getestet.

Problem
Die Events, welche vom Parser zurückgegeben wurden, befinden sich in einer anderen Reihenfolge als von der Datenbank zurückgegeben, da pro angegebenen Timestamp die Daten nach deren Timestamp sortiert werden.
Damit der Scaffolder weiss, wie die Events sortiert werden, müsste der gleiche Sortierungsmechanismus wie in Plaso verwendet werden.

[12] Siehe wheel im Kapitel: 6.1

Die Methode `_GetSortedEvents` welche in Plaso für die Sortierung verwendet wird, ist im Plaso Ordner unter tests/parsers/test_lib.py[13] ersichtlich. Die Methode sortiert die Events mit einem Heap. Unter Push Event sind die Attribute ersichtlich, nach welchen sortiert wird. Zuerst wird nach dem Timestamp sortiert, welcher aus Milisekunden seit 1970 besteht, danach nach der Timestamp Beschreibung, dann nach dem Attribute Namen und zum Schluss nach der Speicheradresse des Events selbst. So wird sichergestellt, dass die Sortierung idempotent ist und deterministisch bleibt. Ein solcher Event der sortiert wird, ist eine Value Klasse mit dem Namen EventObject, welche sich im Plaso Ordner unter plaso/containers/events.py[14] befindet.

Um die Sortierungsmethode `_GetSortedEvents` von Plaso zu verwenden, müsste somit für jede Resultatszeile des ResultSets ein solcher Event erstellt werden.

Für die korrekte Erstellung des EventObject ist somit der Timestamp in Milisekunden seit 1970 essentiell.

Die Problematik ist jedoch, dass das Konvertieren eines beliebigen Timestamps, dessen Format dem Plaso Scaffolder unbekannt ist, keine simple Sache darstellt. Dies ist auch in Plaso eine komplizierte Implementation und es lohnt sich nicht dies für den Scaffolder nochmals zu implementieren.

Somit muss ein Workaround gefunden werden, um den Test für den Parser, ohne die Information des Timestamps in Milisekunden seit 1970, zu generieren.

Lösung

Der Timestamp in Milisekunden wird einerseits für die Sortierung der Events und andererseits auch für den Vergleich des erwarteten und wirklichen Resultat verwendet.

Der Plaso Scaffolder ist nicht in der Lage den unbekannten Timestamp zu Milisekunden umzuwandeln. Somit muss der User die Aufgabe übernehmen den Timestamp, welcher das Query beim ResultSet retourniert, zu einem universellen Timestamp zu konvertieren

```
expected_timestamp = timelib.Timestamp.CopyFromString(u'2007-04-22 14:42:37')
```

Diese Zeile, welche im Parser Test zu stehen hat, wird somit im schlussendlich generierten File wie folgt dargestellt.

```
# TODO add expected formatted timestamp for timestamp in database: 449070544.333328
expected_timestamp = timelib.Timestamp.CopyFromString(u'TODO')
```

Der grössere Workaround ist jedoch die Sortierung. Da der Plaso Scaffolder die Sortierung nicht kennt, wird nicht der erste, sondern ein beliebiger Event getestet, die einzige Bedienung ist, dass nicht mehrmals der gleiche Event getestet wird.

Somit hat man die erwarteten Werte, jedoch noch nicht den effektiven Event, mit welchem getestet werden soll.

```
test_event = storage_writer.events[position]
```

Um die Position des Events zu finden, wird zuvor nach dem Event im Resultat vom Parser gesucht.

```
position = storage_writer.index(guessed_event)
```

[13] Siehe File unter: github.com/log2timeline/plaso/blob/master/tests/parsers/test_lib.py

[14] Siehe File unter: github.com/log2timeline/plaso/blob/master/plaso/containers/events.py

Der guessed_event ist somit der Event, welche die erwarteten Daten besitzt.

```
guessed_event = [
        e for e in storage_writer.events
        if e.x == 0 and e.y == 'z' and …
```

Dies hilft dem User den Event zu finden und soll vor dem Einchecken in Plaso noch mit der aktuellen ID ersetzt werden.

In der Zukunft kann dies anders gelöst werden.[15]

5.9.5 Styleguide

Für den Code wird der Styleguide[16] von Plaso verwendet, welches eine Anforderung ist.
Der Styleguide von Plaso unterscheidet sich vom Styleguide von PEP8 einerseits bei den Docstrings, welche bei der API Dokumentation im Kapitel 5.9.6 beschrieben sind sowie beim Naming.

Die Funktionen und Methoden werden in PascalCase geschrieben: `MethodenName()`.
Protected Funktionen starten mit einem Underline, Private mit zwei und ungebraucht Funktionen enden mit einem Underline: `__ProtectedFunktion()` `unused_`. Jedoch wurde vom Betriebspartner gewünscht auch private Funktionen und Methoden nur mit einem Underline zu beginnen.
Die Konstanten sind in Kapitalbuchstaben zu schreiben: `ACONSTANT`.
Argumente sowie Attribute sind SnakeCase zu schreiben: `some_attribute`.
Akronyme wie `SQLite` sollen beibehalten werden.

Die Argumentbeschreibung sowie die Signaturbeschreibung muss für den Input sowie den Output den Typ beinhalten: `def SomeMethod(a_input: str) -> int`.

Alle Klassen erben von `object` und rufen in der `__init__` Methode die Init Methode der Superklasse auf: `super().__init__()`.

Alle Files starten mit dem Encoding: `# -*- coding: utf-8 -*-`.
Falls das File ausführbar ist, folgt vor dem Encoding noch der Ordner des Programm um das File auszuführen: `# !/usr/bin/python`.

Die Import Statements importieren immer das Modul und nicht direkt die Klasse, damit im Code der Ursprung der Klasse ersichtlich ist.

```
from plasoscaffolder.model import sql_query_column_model
… sql_query_column_model.SQLColumnModel …
```

5.9.6 API Dokumentation

Alle Module, Klassen sowie Methoden sind mit Docstring kommentiert.
Für die Docstrings wird der Styleguide[17] von Plaso verwendet, welches eine Anforderung ist.

Die Docstrings starten und enden mit 3 Anführungszeichen.
Die erste Zeile ist die Standardbeschreibung und muss ein Satz sein.
Falls eine Detailbeschreibung folgt, wird dazwischen eine neue Zeile erwartet.

[15] Lösungsmöglichkeit für die Zukunft im Kapitel:11.1.1
[16] Styleguide ersichtlich unter: [29]
[17] Styleguide für die Docstrings ersichtlich unter: [29]

`Args:` bezeichnet denn Start für die Beschreibung der Parameter Argumente, welche in der Reihenfolge in der sie angegeben werden beschrieben werden sollen.
Dazu gehört der Name des Arguments, der Typ und die Beschreibung. Falls die Beschreibung auf einer neuen Zeile fortgeführt werden muss, folgen 4 Spaces.

```
def __init__(
    self, path: str, name: str
    queries: [sql_query_model.SQLQueryModel],
    output_handler: base_output_handler.BaseOutputHandler
):
  """Initializes a SQLite Generator.

  Args:
    path (str): the path of the plaso folder
    name (str): the name of the plugin
    queries ([sql_query_model.SQLQueryModel]): list of queries
    output_handler (base_output_handler.BaseOutputHandler): the output
        handler for the generation information
  """
```

Falls ein Wert zurückgegeben wird, muss dieser mit dem Keyword `Returns:` angekündigt werden und den Typ sowie die Beschreibung beinhalten. Dies folgt nach der Argumentbeschreibung.

```
Returns:
  bool: true if the row name is valid
```

Öffentliche Attribute werden mit dem Keyword `Attributes:` angekündigt und werden im gleichen Schema wie die Argumente kommentiert.

```
Attributes:
  formatter_file_path (str): the path to the formatter file
```

5.9.7 Sphinx

Mit Sphinx ist es möglich aus den Docstrings des Codes eine API Dokumentation zu generieren.
Im Ordner .doc ist das Konfigurationsfile, mit dem Namen conf.py, für die Generierung der Sphinx Dokumentation ersichtlich.
Damit von den Docstrings, welche den Styleguide von Plaso befolgen, die Dokumentation generiert werden kann, braucht es die Napolean Erweiterung, welche Google Style Docstrings lesen kann.

Damit die Dokumentation generiert werden kann müssen zuerst die '.rst' Files generiert werden, welche mit einer Markup Syntax das Layout der API Dokumentation beschreiben.
Dazu muss der Command `spinx-apidoc -o sourcedir applicationdir` ausgeführt werden, welcher beim Plaso SQLite Plugin Scaffolder wie folgt aussieht: `spinx-apidoc -o . ..\src`.
Für diesen Befehl wurde im make.bat File ein Argument erstellt und kann nun mit `make.bat rst` gestartet werden.

Um die Dokumentation zu generieren, muss der Command: `sphinx-build -M html sourcedir builddir` ausgeführt werden, welcher beim Plaso SQLite Plugin Scaffolder wie folgt aussieht: `sphinx-build -M html . ..\docs`.
Damit daraus eine Github Page erstellt werden kann, muss das index.html im Root von ..\docs liegen. Dazu muss der Ordner doctrees gelöscht werden und alle Files vom Ordner html in den Root verschoben werden.
Für diesen Befehl wurde im make.bat File ein Argument erstellt und kann nun mit `make.bat onlyhtml` gestartet werden.

5.9.8 Read The Docs

Damit die API Dokumentation auch Online schnell zu finden ist, wurde sie mit Read the Docs unter: http://plasoscaffolder.readthedocs.io/en/latest/ veröffentlicht.
Da Read the Docs, Github sowie Sphinx unterstützt, wird die Dokumentation nach jedem commit auf dem master branch automatisch aktualisiert.

5.9.9 Jinja2

Jinja2 wird, wie von den Betriebspartner gewünscht, als Template Engine benutzt. Jinja2 ist eine von den meistbenutzten Template Engines für Python und wurde vom Django Templating System inspiriert und erweitert.
Durch ein Binding mit Attributen können für das Erstellen der Files Variablen sowie Klassen übergeben werden und bei Wünschen für eine Änderung am File muss nur das Template angepasst werden.

Folgend ein Ausschnitt aus dem Template File für den Formatter Test, welcher die Verwendung von `for`, `if` sowie Funktionen aufzeigt.

```
{% for query in queries %}
class {{ class_name }}{{ query.name|capitalize
}}FormatterTest(test_lib.EventFormatterTestCase):
  """Tests the {{ plugin_name|replace('_',' ')|title}} {{ query.name|lower }} event
formatter."""

  def testInitialization(self):
    """Tests the initialization."""
    event_formatter = {{ plugin_name }}.{{ class_name }}{{ query.name|capitalize
}}Formatter()
    self.assertIsNotNone(event_formatter)

  def testGetFormatStringAttributeNames(self):
    """Tests the GetFormatStringAttributeNames function."""
    event_formatter = {{ plugin_name }}.{{ class_name }}{{ query.name|capitalize
}}Formatter()

    expected_attribute_names = [
        {% for column in query.columns|sort(attribute='sql_column') %}u'{{
column.GetColumnAsSnakeCase() }}'{% if not loop.last %},{% endif %}
        {% endfor %}]

    self._TestGetFormatStringAttributeNames(
        event_formatter, expected_attribute_names)
{% endfor %}
```

5.10 SQL Query Testen

Es wird davon ausgegangen, dass der User des Plaso Scaffolders kein Entwickler ist und leider auch nicht perfekte SQL Statements schreiben kann.

Deshalb muss das Query getestet werden.

Einerseits kann dies der User selbst überprüfen, in dem ihm Beispielresultate angezeigt werden. Das Query wird auf der angegebenen Testdatenbank ausgeführt und die ersten drei Resultate vom ResultSet werden angezeigt, der User hat hier die Möglichkeit selbst seinen Fehler zu bemerken und das Query zu korrigieren.

Damit nicht versehentlich Datenmanipulationen durchgeführt werden, existiert ein Schutz.
Es wird dem User nur erlaubt read-only Queries zu definieren.
Jegliche Drop, Delete und sonstige Queries mit folgender Datenänderungen werden dem User unterbunden, damit diese nicht versehentlich im Plaso Projekt auftauchen und somit eine produktive Datenbank manipulieren.

Es gibt verschiedene Wege dies zu unterbinden.

5.10.1 **Query überprüfen.**

Eine Option ist, den Query auf Keywörter zu überprüfen welche die Datenmanipulation ermöglichen und diese zu verbieten.
Jedoch ist es auch möglich Keywörter als Aliase zu benutzen.
Bei SQLite müssen diese jedoch escaped werden, welches bedeutet, es wäre möglich auf alleinstehende Keywörter zu prüfen.[18]

5.10.2 **Rollback**

Eine andere Möglichkeit wäre direkt im Template File für den Plaso Parser eine Transaktion für das Query zu erstellen, und am Ende ein Rollback durchzuführen.[19]
Der Nachteil ist jedoch, dass im Plaso Code danach nicht mehr ein simples Query steht, sondern nun jedes Query aus einer Transaktion mit anschliessendem Rollback besteht.
Einerseits verringert dies die Übersicht und die Performance wird verschlechtert.
Der Vorteil wäre jedoch, dass hier bestimmt keine Daten verändert werden können.
Da Log2timeline von Plaso jedoch eine riesen Datenmenge durcharbeiten muss, ist der Performanceverlust nicht verzeihbar.

5.10.3 **Explain**

Es gibt die Möglichkeit den EXPLAIN Command auszuführen.[20]
Dieses Kommando ist jedoch nur für interaktives Debugging gedacht.
Was jedoch ein sehr wertvoller Bestandteil ist, ist der opcode.
Es gibt 163 opcodes, welche von andere SQLite Versionen differenzieren können, die relevanten codes sind OpenRead und OpenWrite.
OpenRead bedeutet, es wird ein read-only Cursor und ein read lock bei der Datenbank erstellt.
OpenWrite bedeutet, es wird ein read/write Cursor erstellt.
Somit kann man auf den opcode für OpenWrite prüfen. Wenn er existiert bedeutet dies, dass das SQL Query nicht read-only ist.

[18] SQLite Keywords unter: [11]
[19] Siehe Transaction unter: [13]
[20] Siehe EXPLAIN unter: [12]

5.10.4 Überprüfung versus Rollback versus Explain

Eine Transaktion mit Rollback mit Hilfe des Template direkt im Parser File zu generieren ist keine gute Idee, da dies auf Performanceverlust sowie duplicated Code im Plaso Projekt beim Ausführen von Log2timeline hinausläuft.

Den Query String zu überprüfen ist möglich, jedoch keine perfekte Lösung. Da man Keywörter auch als Spaltennamen gebrauchen kann, wenn man sie escaped, ist eine sorgfältige Überprüfung nötig. Auch müssen sichergestellt werden, dass alle Keywörter welche Daten manipulieren können unterbunden werden, und dass man dem User nichts verbietet, welches SQLite erlaubt.
Solch eine Prüfung ist deshalb sehr Fehleranfällig.

Der EXPLAIN Command, ist zwar nur für das Debugging gedacht und nicht sehr performant, jedoch wird der User den Performanceverlust beim Ausführen des Plaso Scaffolders nicht bemerken.
Und da damit geprüft werden kann, ob die Datenbank für den Query nur im read-only Zustand geöffnet wird, kann man damit sicherstellen, dass keine Daten verändert werden.

Da das Rollback im Parser File einen direkten Performanceverlust bei Log2timeline von Plaso fordert ist es nicht gut geeignet. Deshalb ist der EXPLAIN Command, welcher nur zu Performanceverlust beim Plaso Scaffolder führt, bei welchem Performance kein kritisches Problem ist, viel besser geeignet.
Und da man mit der Query Überprüfung nicht ohne aufwendiges Query Parsing und mögliches verbieten von Keywörtern in Select Statements, sicherstellen kann, dass keine Datenmanipulation durchgeführt wird, ist der EXPLAIN Command die beste Lösung.

5.11 Typ Herausfinden

Es ist eine Herausforderung herauszufinden was für Typen die Attribute haben.
Dies wird einerseits für die API Beschreibung gebraucht und andererseits wird bei den Parser Tests ein Boolean anders als ein String oder Integer behandelt.
Es ist somit eine grosse Hilfe für den User, da er die Datentypen nicht selbst aus der Datenbank suchen muss.

5.11.1 Typ per Beispieldaten

Wenn man im Internet danach sucht, wie man typischerweise den Datentyp von einer SQLite Datenbank mit Python herausfinden kann, bekommt man nicht viele Hinweise.
Vorgeschlagen wurde auf die Daten zuzugreifen und daraus mit `type(attribute_data)` den Datentyp der Daten herauszufinden.
In einem ersten Versuch wurde dies so implementiert und man ist auf verschiedene Probleme gestossen.

Keine Daten

Wenn in der Datenbank keine Daten zu diesem Query vorhanden sind, können die Datentypen nicht geparst werden.

Null Werte

Ein Attribut kann Null Werte für Int oder Strings beinhalten. Dies bedeutet, dass man solange Datenzeilen parsen muss, bis ein Wert für dieses Attribut gefunden wird. Jedoch ist nicht sichergestellt ob ein Wert vorhanden ist.

Zuverlässigkeit

Ein Boolean wird als 1 oder 0 abgespeichert. Dies bedeutet, wenn man den Typen für die Beispieldaten liest, wird bei `type(1)` ein `class<int>` retourniert und nicht der tatsächliche `class<bool>`.

Somit kann mit der Builtin Funktion `type` der Typ nicht zuverlässig gefunden werden.

5.11.2 Typ per Schema

Da der Typ nicht über die Testdaten herausgefunden werden kann, muss man diesen mit dem Schema herausfinden.

Tabellen Information

Mit dem SQL Command `PRAGMA table_info(table)` werden alle Attribute Namen, ihr Typ, Default Wert, ob sie ein Primary Key sind oder ob sie Null sein können, retourniert.

Hier sieht man ein Teil des ResultSet für die Tabellen Information von der Tabelle Users der Twitter SQLite Datenbank.

Result Set (35 items)

cid	name	type	notnull	dflt_value	pk
0	id	INTEGER	0	*null*	1
1	screenName	TEXT	0	*null*	0
2	profileImageUrl	TEXT	0	*null*	0
3	profileBannerUrl	TEXT	0	*null*	0
4	profileLinkColorHexTriplet	INTEGER	0	*null*	0
5	name	TEXT	0	*null*	0
6	location	TEXT	0	*null*	0

Abbildung 11: ResultSet PRAGMA table_info(users)

Parsen der Attribute Namen

Aus der Cursor Beschreibung werden die Namen der benutzten Attribute des SELECT Queries geholt. Dieser Name ist entweder der Name des Attributes oder falls angegeben, der Alias.

Nun muss für jedes dieser Namen herausgefunden werden, zu welchem SQL Attribut es gehört.

Dazu kann der SQL Query geparst werden.

Attribute zur Datenbank mappen

Falls der User ein Query schreibt, welches nur Daten aus einer Tabelle selektiert, ist es kein Problem die Datentypen zu den Attributen aus dem PRAGMA Tabelleninformation ResultSet zu lesen.

Falls der User jedoch Daten aus mehreren Tabellen selektiert, muss herausgefunden werden, welches Attribut zu welcher Tabelle gehört.

Dies kann man aus dem Query parsen[21].

Falls jedoch der User ein Attribut im Query benutzt, welches Unique über die Tabellen ist und somit kein Tabellenprefix notiert, muss mit try and error das Attribut aus den Tabellen gelesen werden.

[21] Das Parsen des Queries wird im Kapitel 5.11.4 und 5.11.5 erläutert.

5.11.3 Mapping

Wenn man den Datentyp für das Attribut der SQLite Datenbank herausgefunden hat, muss dieser Typ auf einen Typ von Python gemappt werden.
Dazu wird ein Dictionary für alle bekannten Datentypen von SQLite[22] verwendet.

Python Datentyp	SQLite Datentyp
bytes	BLOB
integer	INT, INTEGER, TINYINT, SMALLINT, MEDIUMINT, BIGINT, UNSIGNED BIG INT, INT2, INT8
str	CHARACTER(20), VARCHAR(255), VARYING CHARACTER(255), NCHAR(55), NATIVE CHARACTER(70), NVARCHAR(100), TEXT, CLOB
float	REAL, DOUBLE, DOUBLE PRECISION, FLOAT
int	NUMERIC, DECIMAL(10,5), BOOLEAN, DATE, DATETIME

Tabelle 7: Mapping zwischen SQLite und Python Datentypen

5.11.4 Per Explain das Query Parsen

Da das manuelle Parsen des Queries keine einfache Sache ist, wäre die Benutzung von Explain eine simplere Lösung und wird deshalb zuerst ausprobiert.
Mit `EXPLAIN select_query` kann man auslesen wie das Query geparst wurde.

Result Set (20 items)

addr	opcode	p1	p2	p3	p4	p5	comment
0	Trace	0	0	0		00	null
1	Goto	0	15	0		00	null
2	OpenRead	0	6	0	6	00	null
3	OpenRead	1	3	0	9	00	null
4	Rewind	0	12	0		00	null
5	Rewind	1	11	0		00	null
6	Rowid	0	1	0		00	null
7	Column	0	5	2		00	null
8	Column	1	8	3		00	null
9	ResultRow	1	3	0		00	null
10	Next	1	6	0		01	null
11	Next	0	5	0		01	null
12	Close	0	0	0		00	null
13	Close	1	0	0		00	null
14	Halt	0	0	0		00	null
15	Transaction	0	0	0		00	null
16	VerifyCookie	0	10	0		00	null
17	TableLock	0	6	0	Users	00	null
18	TableLock	0	3	0	Statuses	00	null
19	Goto	0	2	0		00	null
190		6	96	5			

Abbildung 12: EXPLAIN select users.id, name, card from users join statuses

Mit dem opcode TableLock werden die verschiedenen Tabellen angegeben, auf welche zugegriffen wurde. Der p2 des TableLock gibt den Tabellen eine ID.
Mit dem opcode OpenRead, werden die Tabellen zum Lesen geöffnet, der p2 ist die ID der Tabellen des TableLock und der p1 ist die ID der Tabellen für die Spalte.

[22] Siehe SQLiteDatentypen unter: [23]

Der opcode Column beschreibt die Spalten, auf welche zugegriffen wurde. Der p1 ordnet die Spalte zur ID der Tabelle von OpenRead p1, der p2 ist die CID[23] der Spalte in der Tabelle und der p3 ist die Position des Attributes im Select Statement.
Zusätzlich gibt es noch den opcode Rowid, welches die ID der Spalte darstellt. P1, p2 und p3 bedeuten das Gleiche wie bei den Column.

Falls man denn Query noch sortiert, verändert sich der Output ein wenig mit zusätzlichen Informationen.

Result Set (20 items)

addr	opcode	p1	p2	p3	p4	p5	comment
20	OpenPseudo	3	4	3		00	null
21	OpenPseudo	4	9	3		00	null
22	SorterSort	2	30	0		00	null
23	SorterData	2	9	0		00	null
24	Column	4	2	4		20	null
25	Column	3	0	1		20	null
26	Column	3	1	2		00	null
27	Column	3	2	3		00	null

Abbildung 13: EXPLAIN select users.id, name, card from users join statuses order by name

Der opcode OpenPseude öffnet die Pseudo Tabelle mit der ID in p1, dazu gibt es nun neue opcode für Column, welche mit p1 die Pseudo Tabelle referenziert, mit p2 die Column ID p1 der vorherigen COlumn referenziert und p3 die Sortierung vorgibt. Zusätzlich gibt es noch die Column welche für die Sortierung benutzt wurde.

Diese Inhalte genügen um die Attribute zu den Aliasen und zu den Tabellen zuzuordnen.
Dies wurde auch so implementiert.

Problem

Dies funktionierte wunderbar lokal auf der Windows sowie Ubuntu Maschine.
Jedoch hat man herausgefunden, dass der EXPLAIN Command nicht gut für Applikationen geeignet ist und nur für Debugging benutzt werden soll. Denn der Output kann pro SQLite Version sehr variieren. Deshalb sind die Tests auf Travis fehlgeschlagen.
Daraufhin wurde die Idee dieser Lösungsmöglichkeit verworfen.

5.11.5 Per Python das Query Parsen

Da der Versuch mit dem EXPLAIN Command fehlgeschlagen ist. Muss das Query manuell mit Python geparst werden.
Dazu wird für jeden Attribute Namen oder Alias, das SQL Attribute und die Tabelle direkt aus dem Query geparst.

1. Zuerst wird die Position des Attribute Namens gesucht.
2. Danach wird herausgefunden, ob es sich um einen Alias oder den Attribute Namen handelt.
3. Falls es ein Alias ist, wird das 'as' Keyword davor gesucht und danach das SQL Attribute und der Tabellen Prefix gesucht.
4. Falls es kein Alias ist, wird nur der Tabellen Prefix gesucht.
5. Falls kein Tabellen Prefix angegeben wurde, wird über die Tabellen Information herausgefunden, zu welcher Tabelle das Attribut gehört. Siehe: 5.11.2
6. Danach wird der Typ, mit Hilfe der Tabellen Information, für das Attribut aus der Tabelle gelesen. Siehe: 5.11.2
7. Zum Schluss wird noch der SQL Datentyp zu einem Python Datentyp gemappt. Siehe: 0

Da das Query nun manuell geparst wird, gibt es einige Restriktionen für den User.
Diese können schrittweise entfernt werden, wenn der Parsing Algorithmus verfeinert wird.

- Der User darf kein Sonderzeichen benutzen ausser `; , * = _`

[23] Die CID ist ersichtlich bei der Abbildung 11: ResultSet PRAGMA table_info(users)

Mit den restlichen Sonderzeichen kann der Parsing Algorithmus noch nicht umgehen.
- Der User darf keinen Alias für den Tabellen Namen benutzen, da der Prefix der Attribute im Moment als Tabelle identifiziert wird.
- Falls zwei Attribute von verschiedenen Tabellen gleich heissen, muss mindestens ein Attribut einen Alias haben. Einerseits ist dies nötig, damit korrekt geparst werden kann, andererseits ist dies eine Anforderung, da die Namen der Attribute in den Output Files als Variable benutzt werden und diese verschiedene Namen haben müssen.

5.12 Yapf

Yapf steht für 'Yet another Python Formatter' und kann für das Formatieren von Python Files verwendet werden.
Um die Formatierung zu konfigurieren, kann man in einem File mit dem Namen '.style.yapf' Parameter definieren.

```
[style]
based_on_style = chromium
COALESCE_BRACKETS = True
SPLIT_BEFORE_FIRST_ARGUMENT = True
```

Der Styleguide von Plaso basiert auf Chromium[24].

5.12.1 Formattierung ausschalten

Da Yapf noch ein junges Programm ist und noch nicht alle Features unterstützt welche man gerne hätte, kann man Programmteile definieren, welche man nicht von Yapf formatiert haben möchte.

Mit den Kommentarzeilen `# yapf: disable` wird die Formatierung für den Codeabschnitt solange im Originalzustand beigelassen bis der Kommentar `# yapf: enable` folgt.
Dies wird eingesetzt, um gewisse Codezeilen leserlicher und besser editierbar zu halten.

Diese Kommentarzeilen für Yapf sollen jedoch im generierten File nicht vorhanden sein. Deshalb werden nach der Formatierung mit Yapf diese Kommentare entfernt.

Zusätzlich bestehen ohne die Formatierung mit Yapf einige unschöne Spaces am Ende der Zeilen, welche durch das Verwenden der Jinja2 Syntax nicht vermieden werden können.
Deshalb werden nach der Formatierung mit Yapf noch alle Spaces am Ende der Zeilen entfernt.

5.12.2 Style File finden

In Yapf wird beschrieben, dass das File mit dem Namen. style.yapf automatisch im Projekt gefunden wird, falls es in derselben Ordnerstruktur existiert.
Dies ist jedoch noch nicht der Fall. Deshalb wird der Pfad angegeben.

5.12.3 Zeilenumbruch

Yapf ermöglicht es noch nicht, dass Unicode Strings nach 80 Zeichen automatisch auf der nächsten Zeile fortgesetzt werden.
Um dies zu ermöglichen, wird der Umbruch mit Jinja2 definiert.

```
u'{{datacolumn.GetFirstDataForTimeEvent(timestampcolumn.sql_column)|e|wordwrap(widt
h=64,wrapstring="'\n        u'")|indent(8)}}'
```

[24] Siehe Chromium Styleguide unter: [24]

Wordwrap ist eine Funktion, welche von Jinja2 unterstützt wird und nach der angegebenen `width` den String unterbricht und den Inhalt von `wrapstring` dazwischen einfügt.
Da die Funktion `wordwrap` nur für Strings implementiert wurde, muss der String zuerst mit `e` escaped werden. Die verschiedenen Funktionen können bei Jinja2 mit einer Pipe `|` definiert werden. Die Funktion `indent` fügt vor dem gesamten String noch die angegebene Anzahl von Spaces hinzu.

Da der String zuerst escaped werden muss, bevor der String gewrappt werden kann, ist folgender Nebeneffekt möglich.
Es kann sein, dass der String genau beim escape `\\` unterbrochen wird. Dies führt jedoch zu ungewollten Nebeneffekten und resultiert zu invalidem Python Code.
Da Yapf das File mit Python zuerst validiert, würde Yapf an diesem Punkt fehlschlagen.

Dieser fehlerhafte Effekt wird somit nach dem Generieren des Files mit Jinja2 und vor dem Formattieren mit Yapf entfernt.
Dafür werden alle Zeilen modifiziert, welche ein Backslash am Ende der Zeile und ein Unicode Startzeichen `u'` sowie ein Backslash auf den nächsten Zeilen beinhalten.
Somit wird der Codeabschnitt mit folgendem Muster modifiziert, da dieser Fehler nur an solchen Stellen auftreten kann.

```
\'newline            u'\
```

Der Backslash wird auf die nächste Zeile geschrieben und im Jinja2 Template wird bewusst nicht nach 80, sondern nach 79 Zeichen umgebrochen, damit ein zusätzliches Zeichen noch Platz hat.

```
'newline             u'\\
```

Falls Yapf in Zukunft die Formatierung von Unicode Strings unterstützt oder Plaso auf Python 3 portiert wird, bei welchem Unicode der Standard für Strings ist, kann das Template angepasst und die Modifizierung entfernt werden.

6 Proof of Concept

In diesem Kapitel sind Tools und Libraries dokumentiert welche für welche ein Proof of Concept durchgeführt wurde.

6.1 Setuptools

Um die Konsolenapplikation schlussendlich einfach installieren zu können. Wurde setuptools[25] getestet.

Im setup.py File wird der Pfad zum Startpunkt der Applikation sowie die zu installierenden Packages angegeben.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
"""This is the setup file for the project."""
from setuptools import setup, find_packages

setup(
    name='plasoscaffolder',
    version='0.1',
    packages=find_packages(),
    include_package_data=True,
    install_requires=['Click',…],
    entry_points='''
        [console_scripts]
        plasoscaffolder=plasoscaffolder.mainclick:main''',
)
```

Der Startpunkt der Applikation ist hier die Funktion main im File mainclick welche im Package plasoscaffolder liegt. Der Applikation wurde den Namen plasoscaffolder gegeben, mit welchem das Programm aufgerufen werden kann.

Für das einfache Installieren als Package kann man folgende Schritte ausführen.

1. Virtualenv erstellen → virtualenv venv
2. Virtualenv aktivieren → venv/Scripts/activate … (je nach OS)
3. Package installieren → python setup.py install
4. Starten → plasoscaffolder

Mit Hilfe von Setuptools und twine[26] wurde die Applikation in ein Python wheel[27] Package gepackt und auf PyPI geladen.

1. Twine installieren → pip install twine
2. Distribution erstellen → python setup.py bdist_wheel
3. Generiertes PKG-INFO welches sich im Ordner dist befindet, auf PyPI laden.
4. Erstelltes wheel File welches sich im Ordner dist befindet, auf PyPi laden.

Nun kann man das Package mit pip installieren.

Das Projekt muss dafür nicht mehr Heruntergeladen werden und das setup.py muss nicht mehr separat für die Installation der Packages ausgeführt werden.

1. Plasoscaffolder installieren → pip install plasoscaffolder
2. Starten → plasoscaffolder

[25] Setuptool ersichtlich unter: [5]

[26] Twine ersichtlich unter: [20]

[27] Wheel ersichtlich unter: [19]

6.1.1 Schwierigkeiten

Da standardmässig mit setuptools nur Python Files kopiert werden, wurden Jinja2 Template-Files und Konfigurationsfiles nicht gefunden.
Dazu kann man mit `package_data ={'Destinationspfad', ['FileRegex']}` die restlichen Filearten angeben.

Da zuerst keine konkrete Version angegeben wurde. Hat ein Update eines Packages meinen Tests fehlschlagen lassen.[28] Setuptools erlaubt es Package Version im Format `package >= version` anzugeben.

6.2 Click

Benutzt wird Click[29] als Command Line Tool library.
Der grosse Vorteil von Click gegenüber argparse und anderen Alternativen ist, das Click das Nesting von Commands unterstützt. Da der Scaffolder auch simple für andere Plugins ausgebaut werden soll, baut der Plaso Scaffolder auf diesem Prinzip auf.

Um zu testen ob Click für die Applikation in Frage kommt, wurde ein kleiner Prototyp geschrieben.

Dies ist die Filestruktur des Prototyps.

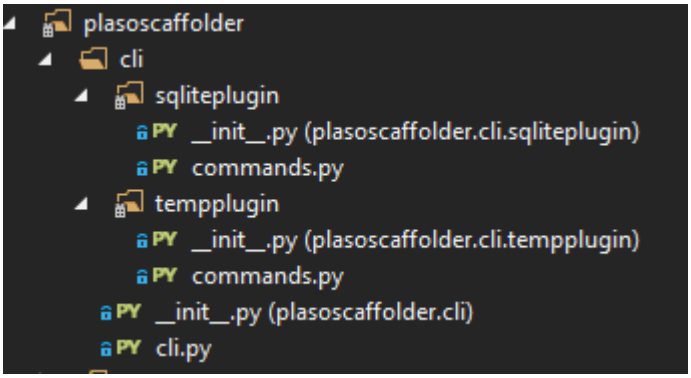


Abbildung 14: Click Prototyp Filestruktur

6.2.1 cli.py

Cli ist das Hauptprogramm.

```
import click

from tempplugin import commands as tempplugin
from sqliteplugin import commands as sqliteplugin


@click.group()
def entry_point():
    pass

entry_point.add_command(tempplugin.temp)
entry_point.add_command(sqliteplugin.sqlite)

if __name__ == '__main__':
    entry_point()
```

[28] Siehe denn Testbericht im Kapitel: 15.3.2.3
[29] Click ersichtlich unter: [6]

Hier wird die Hauptgruppe für die verschiedenen Plugins erstellt.
Als Versuch wurde ein 'temp' Plugin sowie ein 'sqlite' Plugin als Command hinzugefügt.

Beim Ausführen des Programms hat man somit auf erster Stufe diese zwei Commands zur Auswahl.

```
(venv) C:\temp> python C:\git\PlasoScaffolder\.src\plasoscaffolder\cli\cli.py
Usage: cli.py [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  sqlite
  temp
```

Abbildung 15: Prototyp Click Output 1

Bei der Auswahl des Commands 'sqlite' hat man danach eine Gruppe von nested Options zur Auswahl.
Diese kann man direkt als Options beim Aufruf des Programes angeben.
Falls die Option jedoch nicht angegeben wurde, wird man interaktiv danach gefragt.

```
(venv) C:\temp> python C:\git\PlasoScaffolder\.src\plasoscaffolder\cli\cli.py sqlite --help
Usage: cli.py sqlite [OPTIONS]

Options:
  -n, --name TEXT  The plugin name
  --tfp TEXT       The testfile path
  --tables TEXT    database tables
  --event INTEGER  event types
  --testfile TEXT  The testfile path
  --help           Show this message and exit.
```

Abbildung 16: Prototyp Click Output 2

Der Guide führt den User mit sinnvollen Fragen durch die Generierung. Hier wurde ein solcher Ablauf vereinfacht abgebildet, um die verschiedenen Möglichkeiten auszuprobieren.

```
(venv) C:\temp> python C:\git\PlasoScaffolder\.src\plasoscaffolder\cli\cli.py sqlite
What's the name of the plugin?: pluginname
What's the path to your test file?: testfile/thisandthat
What tables need to exist in the database?: this, that, me, you
Which different event types do you have? [3]: 2
Enter the query for the0 [12]: me and you
Error: me and you is not a valid integer
Enter the query for the0 [12]:
12
Enter the query for the1 [12]:
12
What's the path to your test file?:
What's the path to your test file?: Aborted!
(venv) C:\temp>
(venv) C:\temp>
(venv) C:\temp> python C:\git\PlasoScaffolder\.src\plasoscaffolder\cli\cli.py sqlite
What's the name of the plugin?: pluginxy
What's the path to your test file?: testfile/test/bla
What tables need to exist in the database?: this,that,all
Which different event types do you have? [3]:
Enter the query for the0 [12]:
12
Enter the query for the1 [12]:
12
Enter the query for the2 [12]:
12
What's the path to your test file?: testfile.db
Do you want to generate the files? [y/N]: y
generated
pluginxy
testfile/test/bla
this,that,all

testfile.db
```

Abbildung 17: Prototyp Click Output 3

Dies ist die simple Implementation ohne nötige Kommentare und Tests:

```
import click

def events(ctx,param,value):
  for x in range(value):
    value = click.prompt('Enter the query for the'+str(x), default=12)
    click.echo(value)

@click.command()
@click.option('--name', '-n', prompt="What's the name of the plugin?",
              help='The plugin name')
@click.option('--tfp', prompt="What's the path to your test file?",
              help='The testfile path')
@click.option('--tables', prompt="What tables need to exist in the database?",
              help='database tables')
@click.option('--event', default=3, prompt="Which different event types do you have?",
              help='event types',callback=events)
@click.option('--testfile', prompt="What's the path to your test file?",
              help='The testfile path')
def sqlite(name,tfp,tables,event,testfile):
  click.confirm('Do you want to generate the files?', abort=False)
  click.echo("generated")
  click.echo(name)
  click.echo(tfp)
  click.echo(tables)
  click.echo(event)
  click.echo(testfile)
```

6.3 Visual Studio PTVS versus PyCharm

Es wurde Visual Studio mit Python Tools[30] ausprobiert. Jedoch war die Unterstützung von IntelliSense nicht optimal und es gab keine gute automatische Formatierungsoption.
Ein weiterer Grund war, dass man den PYTHONPATH selbst setzen musste, damit die Tests gefunden werden und laufen.

PyCharm[31] ist von JetBrains und extra für Python gemacht. Deshalb ist die Unterstützung mit Autocompletion und Testen einiges besser ausgearbeitet.
Da schon ein Settingsfile mit den Einstellungen für die Plaso Richtlinien existieren, kann man sich diesen Teil der Arbeit ersparen.

[30] PTVS unter: [7]

[31] Pycharm unter: [8]

6.4 Sequenzdiagramm mit seqdiag

Es wurde überlegt, wie das gewünschte Sequenzdiagramm erstellt werden soll.
Optimal wäre gewesen, das Diagramm aus dem Code zu generieren, leider gibt es dafür keine Möglichkeit.
Danach wurde versucht das Sequenz Diagramm mit Visio von Microsoft sowie mit Astah[32] zu erstellen, bevor das Tool seqdiag gefunden wurde.
Visio ist für kleine Sequenzdiagrammen geeignet, jedoch je grösser das Diagramm wurde, je umständlicher wird es die grafischen Elemente korrekt anzuordnen, vor allem da man jede Form separat zeichnen und platzieren muss.
Deshalb wurde danach das Sequenzdiagramm mit Astah gezeichnet. Astah funktioniert grundsätzlich gut, und übernimmt das Generieren sowie Gliedern von einigen grafischen Elementen selbst.
Jedoch ist der Vorgang für das Erstellen immer noch sehr zeitintensiv und mit vielen Mausklicken verbunden.
Danach wurde seqdiag[33] ausprobiert, welches ein in Python geschriebenes Tool ist. Der Vorteil von seqdiag ist, dass das Diagramm vollständig generiert wird und mit einer einfach definierten Syntax definiert werden kann. Der einzige Nachteil von seqdiag ist, dass keine Syntax definiert wurde, um einem loop oder alt Element eine Kondition anzufügen. Deshalb wurde dies mit Notizen gelöst.

Folgend ein Beispiel für ein einfaches Diagramm[34], welches mit dem Command *seqdiag filename* generiert werden kann.

```
seqdiag {
  browser  -> webserver [label = "GET /index.html"];
  browser <-- webserver;
  browser  -> webserver [label = "POST /blog/comment"];
              webserver  -> database [label = "INSERT comment"];
              webserver <-- database;
  browser <-- webserver;
}
```

Daraus entsteht folgendes Diagramm:

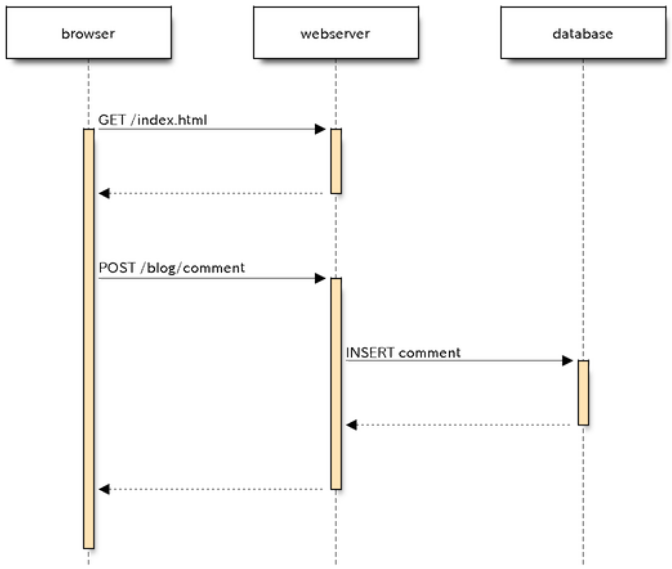


Abbildung 18: Einfaches seqdiag Diagram

[32] Astah unter: [12]
[33] seqdiag unter: [10]
[34] Diagramm Beispiel vorhanden unter: [10]

Das Tool kann mit pip unter Linux installiert werden.

- virtualenv venv — *Erstellen eines virtualen Environments*
- source venv/bin/activate — aktivieren des Environment
- pip install seqdiag — *installieren von seqdiag*

Bei einigen Systemen gibt es Probleme mit Pillow, mit diesen Schritten wird die korrekte Version installiert:

- pip uninstall pillow
- sudo apt-get install libjpeg-dev zbli1g-dev
- pip install -I Pillow

Um die Schriftart zu ändern müssen folgende Schritte getan werden:

- sudo apt-get install libfreetype6-dev
- pip uninstall pillow
- pip install –no-cache-dir pillow

Nun kann das Diagramm mit *seqdiag -f font.ttf* mit der gewünschten Schriftart generiert werden.

7 Technologien

7.1 Verwendete Technologien

Folgend eine Übersicht über die verschiedenen Tools, Applikationen und Packages, welche während sowie in der Arbeit verwendet werden.

7.1.1 **Tools**

Name	Verwendung	Kapitel
Plaso [2]	Projekt für welches Source Code generiert wird.	3
Pylint [3]	Ein Source Code Analyzer der nach Programmierungsfehlern sowie Code Smells sucht und hilft die Codierungsstandards einzuhalten.	5.3.1
Appvoyer [4]	Continuous Integration und Deployment Service für Windows, welcher für das automatische Testen auf Windows verwendet wird.	5.5.1
Travis [5]	Continuous Integration und Deployment Service für Linux und OSX, welcher für das automatische Testen auf Linux verwendet wird.	5.5.2
Coveralls [6]	Tool, welches die Testabdeckung für den Source Code berechnet und anzeigt welche Source Codeteile mit Tests abgedeckt wurden.	5.5.3
Docstring	Art der Dokumentation, welche im Code für die Kommentare verwendet wurde.	5.9.6
Sphinx [7]	Ein Tool, welches ermöglicht eine HTML API Dokumentation aus dem Code mit den Docstrings zu generieren.	5.9.7
Read the Docs [8]	Seite, die Dokumentationen hostet und Github sowie Sphinx unterstützt.	5.9.8
Twine [9]	Tool, welches verwendet wurde um ein wheel Package zu erstellen.	6.1
PyPI [10]	Der Python Package Index über welchen Packages heruntergeladen werden und der Plaso SQLite Plugin Scaffolder veröffentlicht wurde.	6.1
Virtualenv [11]	Tool, welches eine isolierte Python Umgebung sowie einen Ordner erstellt, der alle ausführbaren Packages beinhaltet die ein Python Projekt standardmässig braucht.	6.1
wheel [12]	Tool, welches verwendet wird um ein Standard Distribution Package zu erstellen, welches auf PyPI geladen wurde.	6.1
Pycharm 2016.3.2 [13]	Entwicklungsumgebung unter welcher der Plaso SQLite Plugin Scaffolder entwickelt wurde.	6.3
seqdiag [14]	Python Tool, mit welchem man mit einer vorgegebenen Syntax Sequenzdiagramme erstellen kann.	6.4
Astah [15]	Tool, welches für die Erstellung der Use Cases in der Anforderungsspezifikation (Anhang) sowie die Erstellung des Komponenten Diagramm verwendet wird.	8.2, Analyse
GitHub [16]	Version Control Repository mit Git, welches für die Source Code Verwaltung sowie Codereviews verwendet wird.	10.4

gitstats [17]	Ein Tool, welches eine Statistik über ein Git Repository generieren kann.	10.5
YouTrack [18]	Management Tool für die Projektplanung, Zeiterfassung und Sprintplanung.	15.2
Google Drive [19]	Dokumentationsablage der Arbeit.	Projektplan
Ubuntu 16.04.2 LTS [20]	Betriebssystem für Linux spezifische Arbeit.	Projektplan
Windows 10	Betriebssystem des Gerätes auf welchem die primäre Arbeit geleistet wurde.	Projektplan
MS Word	Tool, welches für die Erstellung der Dokumente verwendet wurde.	Projektplan

Tabelle 8: Verwendete Tools

7.1.2 Packages

Name	Verwendung	Kapitel
Python 3.6	Die verwendetete Python Version für die Programmierung des Plaso SQLite Plugin Scaffolder.	2.4
unittest [21]	Standard Test Framework von Python, welches für die Unittests verwendet wird und bereits Teil von Python ist.	5.8.1
Pexpect [22]	Ein Package, welches unter Linux verwendet werden kann um interaktive Command Line Interface Applikationen automatisch zu testen.	5.8.4
Jinja2 [23]	Eine benutzerfreundliche Templating Sprache für Python, welche nach Djangos Templates modelliert wurde.	5.9.9
Yapf [24]	Ein Tool für die Formatierung von Python Files, welches unter anderem die aktuellen PEP 8 Guidelines befolgt.	5.12
Setuptools [25]	Ein Package, welches die Erstellung von Python Packages erleichtert indem es Metadaten, Plattform spezifische Daten, Python Packages und Modul Informationen an einem Ort kapselt.	6.1
Click [26]	Ein Package, welches hilft, schöne und übersichtliche Command Line Interfaces zu erstellen und auf optparse aufbaut.	6.2
Colorama [27]	Ermöglicht die Verwendung von Farbe unter MS Windows, welche standardmässig von Linux unterstützt wird.	8.1
SQLite [28]	Die SQLite Datenbank Engine, welche für den Zugriff auf Daten der SQLite Datenbanken benutzt wird.	8.4

Tabelle 9: Verwendete Packages

7.2 Lizenzen

Folgende Tabelle beinhaltet die Lizenzen der Abhängigkeiten des Plaso SQLite Plugin Scaffolder, sowie die Lizenz von Plaso und dem Scaffolder.

Obwohl der Plaso SQLite Plugin Scaffolder unter der Lizenz Apache License 2.0 lizenziert ist, da er die Lizenz von Plaso erbt[35], beinhaltet er Third Party Code welcher unter anderen Lizenzen verfügbar gemacht wurden.

Name	Lizenz	Link
Plaso SQLite Plugin Scaffolder	Apache License 2.0 [36]	https://github.com/ClaudiaSaxer/PlasoScaffolder/blob/master/LICENSE
Plaso	Apache License 2.0	https://github.com/log2timeline/plaso/blob/master/LICENSE
Python	Python Software Foundation License [37]	https://docs.python.org/2/license.html
Click	BSD License [38]	https://github.com/pallets/click/blob/master/LICENSE
Setuptools	MIT License [39]	https://github.com/pypa/setuptools/blob/master/LICENSE
Jinja2	BSD License	https://github.com/pallets/jinja/blob/master/LICENSE
Colorama	BSD License	https://github.com/tartley/colorama/blob/master/LICENSE
Yapf	Apache License 2.0	https://github.com/google/yapf/blob/master/LICENSE
Pexpect	ISC License [40]	https://github.com/pexpect/pexpect/blob/master/LICENSE
SQLite	Public domain[41]	http://www.apache.org/licenses/LICENSE-2.0

Tabelle 10: Lizenzen

[35] Dies ist eine Anforderung und ist in der Aufgabenstellung ersichtlich unter: 16
[36] Apache License Informationen ersichtlich unter: [25]
[37] Python Foundation License Informationen ersichtlich unter: [26]
[38] BSD License Informationen ersichtlich unter: [27]
[39] MIT License Informationen ersichtlich unter: [28]
[40] ISC License Informationen ersichtlich unter: [29]
[41] Public Domain Informationen ersichtlich unter: [30]

8 Implementation

8.1 Command Line Interface

8.1.1 Installieren der Applikation

Die Applikation lässt sich mit pip installieren.

```
C:\>pip install plasoscaffolder
Collecting plasoscaffolder
  Using cached plasoscaffolder-0.1-py3-none-any.whl
Requirement already satisfied: setuptools>=35.0.2 in c:\program files (x86)\python36-32\lib\site-packages (from plasoscaffolder)
Requirement already satisfied: pexpect>=4.2.1 in c:\program files (x86)\python36-32\lib\site-packages (from plasoscaffolder)
Requirement already satisfied: jinja2>=2.9.6 in c:\program files (x86)\python36-32\lib\site-packages (from plasoscaffolder)
Requirement already satisfied: Click>=6.7 in c:\program files (x86)\python36-32\lib\site-packages (from plasoscaffolder)
Requirement already satisfied: colorama>=0.3.7 in c:\program files (x86)\python36-32\lib\site-packages\colorama-0.3.7-py3.6.egg (
from plasoscaffolder)
Requirement already satisfied: yapf==0.16.1 in c:\program files (x86)\python36-32\lib\site-packages (from plasoscaffolder)
Requirement already satisfied: ptyprocess>=0.5 in c:\program files (x86)\python36-32\lib\site-packages (from pexpect>=4.2.1->plas
oscaffolder)
Requirement already satisfied: MarkupSafe>=0.23 in c:\program files (x86)\python36-32\lib\site-packages\markupsafe-1.0-py3.6-win3
2.egg (from jinja2>=2.9.6->plasoscaffolder)
Installing collected packages: plasoscaffolder
Successfully installed plasoscaffolder-0.1
```

Abbildung 19: Screenshot pip install plasoscaffolder

8.1.2 Einfaches Ablaufen

Beim normalen Ablauf der Applikation ohne Fehler, werden die Fragen in weiss dargestellt.
Default Werte sind grossgeschrieben und müssen nicht eingegeben werden.
Informationen werden hellblau ausgegeben, falls Farben unterstützt werden.

```
C:\>plasoscaffolder sqlite
What's the path to the plaso project?: C:/temp/plaso
What's the name of the plugin?: the_plugin
What's the path to your test file?: C:/temp/twitter_ios.db
Do you want to have a output example for your SQL Query? [Y/n]: y
Please write your SQL script for the plugin: select id, name, updatedAt from users
Your query output could look like this.
['id', 'name', 'updatedAt']
(5402612, 'BBC Breaking News', 1449070544.333328)
(13334762, 'GitHub', 1449070777.57104)
(14388264, 'Tom Pohl', 1449070777.562435)
Do you want to add this query? [Y/n]: y
Do you want to name the query parse row: Users ? [Y/n]: y
Is the column a time event? updatedAt [Y/n]: y
Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: abort
Does the event Users need customizing? [y/N]: y
Enter columns that are customizable [columnName,aliasName...] or [abort]: name
Added: name
Failed:
Do you want to add more columns that are customizable? [y/N]: n
Do you want to add another Query? [Y/n]: n
Do you want to Generate the files? [Y/n]: y
create C:/temp/plaso\plaso\formatters\the_plugin.py
create C:/temp/plaso\plaso\parsers\sqlite_plugins\the_plugin.py
create C:/temp/plaso\tests\formatters\the_plugin.py
create C:/temp/plaso\tests\parsers\sqlite_plugins\the_plugin.py
copy C:/temp/plaso\test_data\the_plugin.db
edit C:/temp/plaso\plaso\parsers\sqlite_plugins\__init__.py
edit C:/temp/plaso\plaso\formatters\__init__.py
```

Abbildung 20: Screenshot normaler Ablauf

8.1.3 Fehlermeldungen

Bei einem Ablauf mit Fehlermeldungen, werden die Meldungen rot ausgegeben.
Das Programm wird bei fehlerhaften Eingaben nicht beendet, sondern fragt den User nach einer neuen Eingabe.

```
C:\>plasoscaffolder sqlite
What's the path to the plaso project?: C:/temp/doesnotexist
Folder does not exists. Enter correct one: C:/temp/plaso
What's the name of the plugin?: the_plugin
Plugin exists. Choose new Name: the plugin
Plugin is not in a valid format. Choose new Name [plugin_name_...]: The_plugin
Plugin is not in a valid format. Choose new Name [plugin_name_...]: the_plugin_2
Plugin is not in a valid format. Choose new Name [plugin_name_...]: the_next_plugin
What's the path to your test file?: C:/temp/doesnotexist
File does not exists. Choose another.: C:/temp/twitter_ios_wrong.db
Unable to open the database file. Choose another.: C:/temp/twitter_ios.db
Do you want to have a output example for your SQL Query? [Y/n]: y
Please write your SQL script for the plugin: select doesnotexist from users
Error: no such column: doesnotexist
Please write your SQL script for the plugin: select name from doesnotexist
Error: no such table: doesnotexist
Please write your SQL script for the plugin: select name from users; select name from users
Warning: You can only execute one statement at a time.
Please write your SQL script for the plugin: select name as [name] from users
Warning: Don't use any characters beside a-z A-Z 0-9 . ; , * = _
Please write your SQL script for the plugin: select users.id, statuses.id from users join statuses
Please use an alias (AS) for those column names: id
Please write your SQL script for the plugin: select users.id as userid, statuses.id as statusid from users join statu
ses
Your query output could look like this.
['userid', 'statusid']
(2220776716, 510031570397577216)
(2220776716, 666269245827747840)
(2220776716, 666271414903824384)
Do you want to add this query? [Y/n]: y
Do you want to name the query parse row: Usersstatuses ? [Y/n]: n
What row does the SQL Query parse?: UserStatus
Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: abort
Does the event UserStatus need customizing? [y/N]: n
Do you want to add another Query? [Y/n]: n
Do you want to Generate the files? [Y/n]: y
create C:/temp/plaso\plaso\formatters\the_next_plugin.py
create C:/temp/plaso\plaso\parsers\sqlite_plugins\the_next_plugin.py
create C:/temp/plaso\tests\formatters\the_next_plugin.py
create C:/temp/plaso\tests\parsers\sqlite_plugins\the_next_plugin.py
copy C:/temp/plaso\test_data\the_next_plugin.db
edit C:/temp/plaso\plaso\parsers\sqlite_plugins\__init__.py
edit C:/temp/plaso\plaso\formatters\__init__.py
```

Abbildung 21: Screenshot Ablauf mit Fehlermeldungen

8.1.4 Mit Options Eingabe

Es ist möglich beim Start einige Optionen anzugeben.

```
C:\Users\Claudia>plasoscaffolder sqlite --help
Usage: plasoscaffolder sqlite [OPTIONS]

Options:
  --path TEXT       The path to plaso
  --name TEXT       The plugin name
  --testfile TEXT   The testfile path
  --sql / --no-sql  The output example flag for the SQL Query for the plugin.
  --help            Show this message and exit.
```

Abbildung 22: Screenshot plasoscaffolder sqlite --help

Optionen welche schon beim Start angegeben wurden, müssen später vom User nicht erneut eingegeben werden. Jedoch werden sie immer noch validiert.

```
C:\>plasoscaffolder sqlite --path C:/temp/plaso --name mighty_plugin --testfile C:/temp/twitter_ios.db --sql
Please write your SQL script for the plugin: select id, name, updatedAt from users
Your query output could look like this.
['id', 'name', 'updatedAt']
(5402612, 'BBC Breaking News', 1449070544.333328)
(13334762, 'GitHub', 1449070777.57104)
(14388264, 'Tom Pohl', 1449070777.562435)
Do you want to add this query? [Y/n]: y
Do you want to name the query parse row: Users ? [Y/n]:
Is the column a time event? updatedAt [Y/n]:
Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: abort
Does the event Users need customizing? [y/N]: n
Do you want to add another Query? [Y/n]: n
Do you want to Generate the files? [Y/n]: n
Aborted!
```

Abbildung 23: Screenshot normaler Ablauf mit Options

Falls eine Option welche beim Start angegeben wurde invalid ist, wird man um eine neue Eingabe gefragt. Der Validierungsablauf ist derselbe für die interaktive Eingabe und die beim Start eingegebenen Optionen.

```
C:\>plasoscaffolder sqlite --path C:/temp/plaso --name the_plugin --testfile C:/temp/twitter_ios_wrond.db --sql
Plugin exists. Choose new Name: new_plugin
File does not exists. Choose another.: C:/temp/twitter_ios.db
Please write your SQL script for the plugin: select id from users
Your query output could look like this.
['id']
(2220776716,)
(382267114,)
(3150614614,)
Do you want to add this query? [Y/n]:
Do you want to name the query parse row: Users ? [Y/n]:
Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: abort
Does the event Users need customizing? [y/N]:
Do you want to add another Query? [Y/n]: n
Do you want to Generate the files? [Y/n]:
create C:/temp/plaso\plaso\formatters\new_plugin.py
create C:/temp/plaso\plaso\parsers\sqlite_plugins\new_plugin.py
create C:/temp/plaso\tests\formatters\new_plugin.py
create C:/temp/plaso\tests\parsers\sqlite_plugins\new_plugin.py
copy C:/temp/plaso\test_data\new_plugin.db
edit C:/temp/plaso\plaso\parsers\sqlite_plugins\__init__.py
edit C:/temp/plaso\plaso\formatters\__init__.py
```

Abbildung 24: Screenshot fehlerhafter Ablauf mit Options

8.2 Komponenten Diagramm

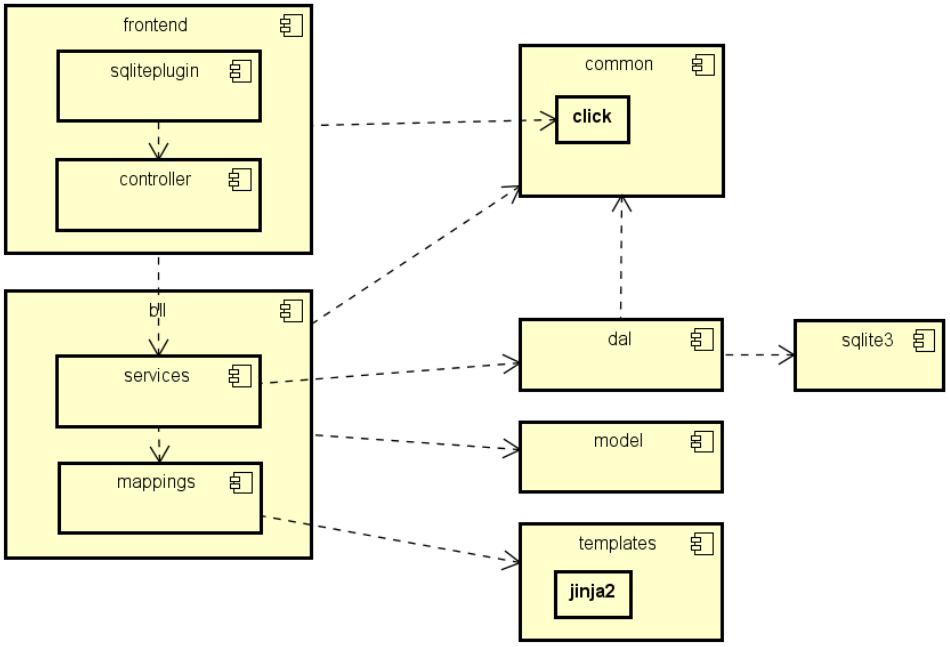


Abbildung 25: Komponenten Diagramm

Dieses Diagramm zeigt die wichtigsten Komponenten und deren Beziehungen auf.

Das Common Modul wird vom BLL, Frontend und DAL verwendet aber nur das Frontend greift auf die Klasse für Click zu.

Der Controller ruft Services auf, welche das Mappings Modul benutzt.

Die Templates werden von den Mappings verwendet.

Nur der DAL macht mit sqlite3 Zugriffe auf die Datenbank.

Der BLL verwendet die Model um die Daten einheitlich zu strukturieren.

8.3 Klassendiagramme

8.3.1 bll.mappings Modul - Mapping Helper

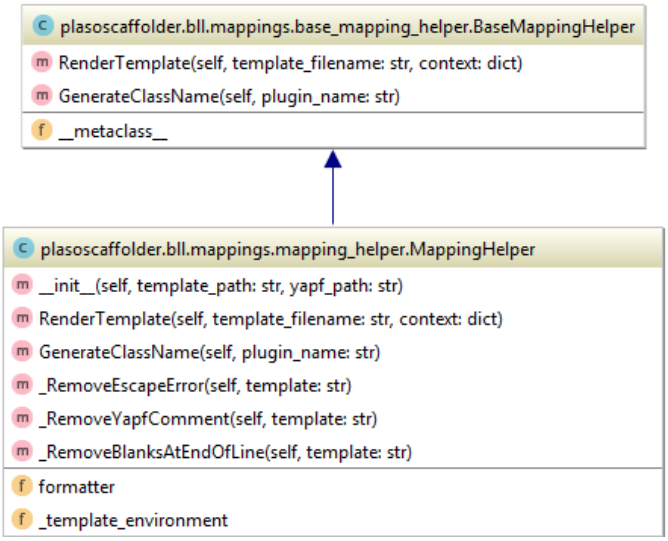


Abbildung 26: Klassendiagramm BLL MappingHelper

BaseMappingHelper[42]
Die Basis Klasse für den Mapping Helper.

`GenerateClassName(plugin_name: str) → str`
Generiert den Klassen Namen aus dem Plugin Namen
Parameters: plugin_name (str) – Der Name des Plugins
Returns: the name of the class
Return type: str

`RenderTemplate(template_filename: str, context: dict) → str`
Rendert das Template mit dem context und retuniert das Generierte File.
Parameters: template_filename (str) – der Name des templates
context (dict) – der Context des Template als Dictionary
Returns: Das gerenderete Template als String
Return type: str

MappingHelper[43]
Helper Klasse für die Konkreten Mapper.

[42] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.bll.mappings.html#module-plasoscaffolder.bll.mappings.base_mapping_helper für die genaue Beschreibung.
[43] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.bll.mappings.html#module-plasoscaffolder.bll.mappings.mapping_helper für die genaue Beschreibung.

8.3.2 bll.mappings Modul - Konkrete Mapper

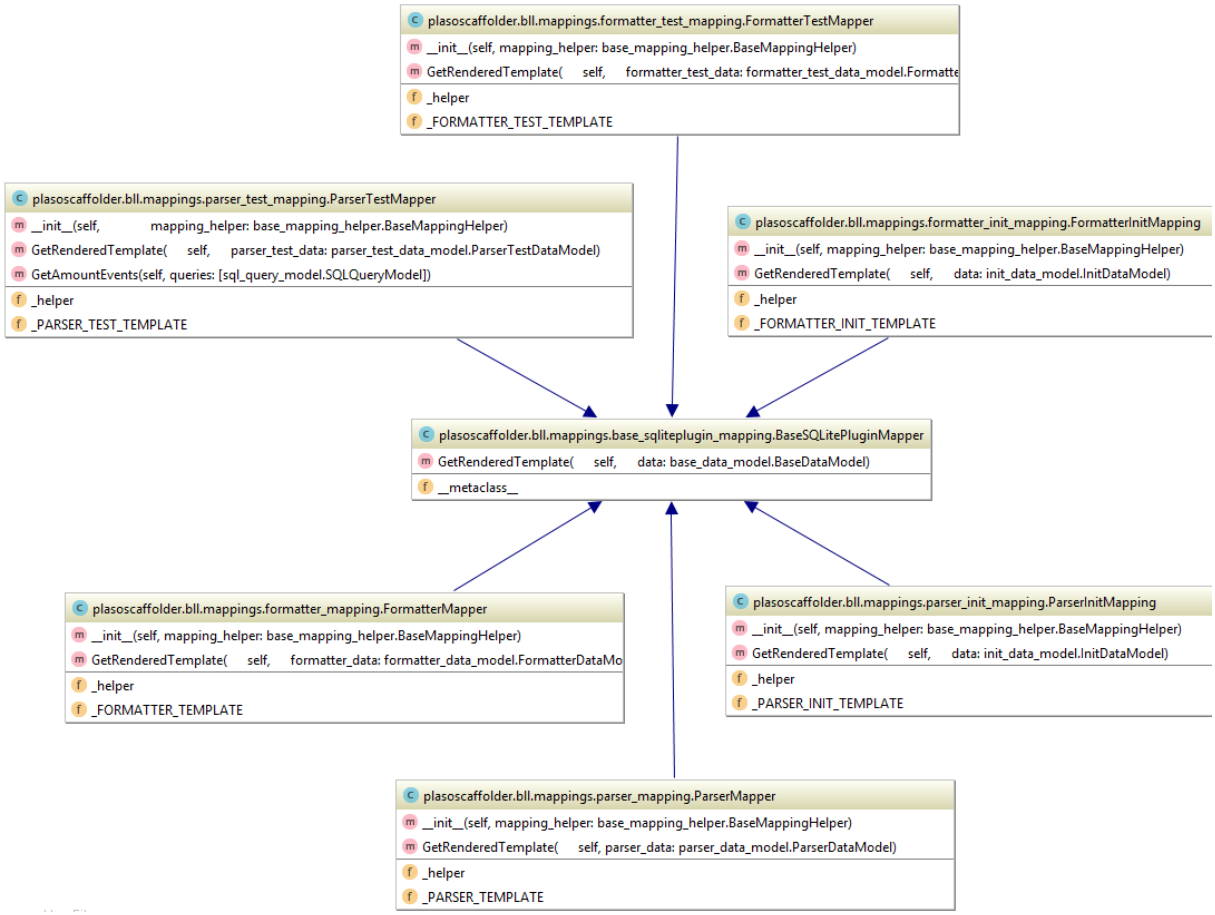


Abbildung 27: Klassendiagramm BLL Mapper

BaseSQLitePluginMapper[44]

Die Basis Klasse für die konkreten Mapper.

```
GetRenderedTemplate(data: plasoscaffolder.model.base_data_model.BaseDataModel) → str
```

Retourniert das generierte Template.

Parameters: data (base_data_model.BaseDataModel) - die Daten für das Template

Returns: das gerenderte Template

Return type: str

FormatterInitMapping[45]

Klasse repräsentiert den Mapper für das Formatter init File und implementiert die BaseSQLitePluginMapper Klasse.

[44] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.bll.mappings.html#module-plasoscaffolder.bll.mappings.base_sqliteplugin_mapping für die genaue Beschreibung.

[45] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.bll.mappings.html#module-plasoscaffolder.bll.mappings.formatter_init_mapping für die genaue Beschreibung.

FormatterMapper[46]

Klasse repräsentiert den Mapper für das Formatter File und implementiert die BaseSQLitePluginMapper Klasse.

FormatterTestMapper[47]

Klasse repräsentiert den Mapper für das Formatter Test File und implementiert die BaseSQLitePluginMapper Klasse.

ParserMapper[48]

Klasse repräsentiert den Mapper für das Parser File und implementiert die BaseSQLitePluginMapper Klasse.

ParserInitMapper[49]

Klasse repräsentiert den Mapper für das Parser init File und implementiert die BaseSQLitePluginMapper Klasse.

ParserTestMapper[50]

Klasse repräsentiert den Mapper für das Parser Test File und implementiert die BaseSQLitePluginMapper Klasse.

```
GetAmountEvents(queries: [<class
'plasoscaffolder.model.sql_query_model.SQLQueryModel'>])  → int
```

Berechnet die Anzahl der Events für das Query.

Parameters: queries ([sql_query_model.SQLQueryModel]) – die Queries

Returns: Die Anzahl der Events

Return type: int

[46] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.bll.mappings.html#module-plasoscaffolder.bll.mappings.formatter_mapping für die genaue Beschreibung.

[47] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.bll.mappings.html#module-plasoscaffolder.bll.mappings.formatter_test_mapping für die genaue Beschreibung.

[48] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.bll.mappings.html#module-plasoscaffolder.bll.mappings.parser_mapping für die genaue Beschreibung.

[49] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.bll.mappings.html#module-plasoscaffolder.bll.mappings.parser_init_mapping für die genaue Beschreibung.

[50] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.bll.mappings.html#module-plasoscaffolder.bll.mappings.parser_test_mapping für die genaue Beschreibung.

8.3.3 bll.services Modul - PathHelper

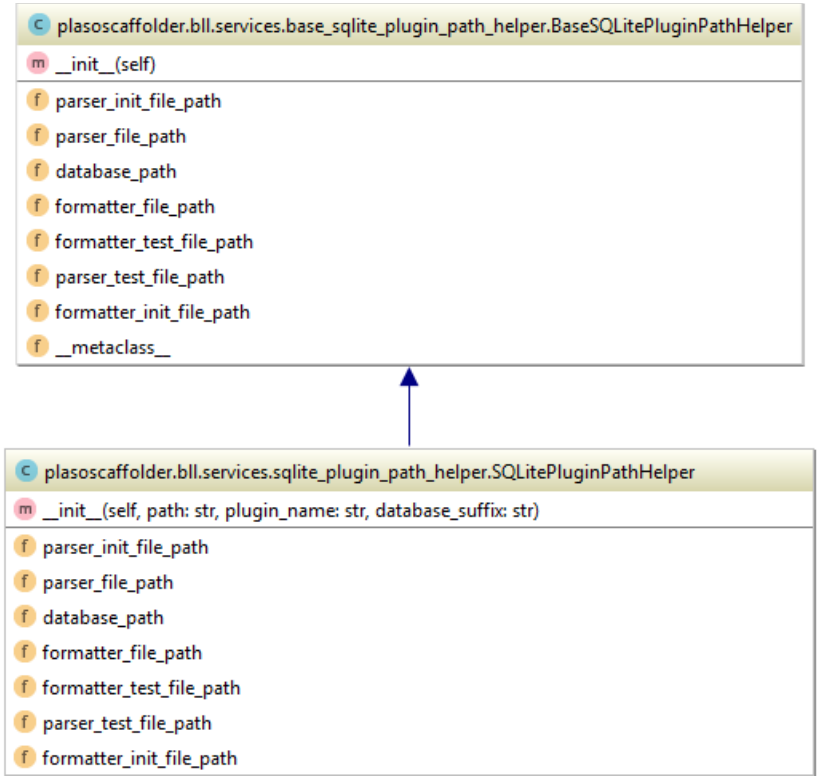


Abbildung 28: Klassendiagramm BLL PathHelper

BaseSQLitePluginPathHelper[51]

Die Basis Klasse für die SQLitePluginPathHelper Klasse.

`formatter_file_path`

str – der Pfad zum formatter file

`parser_file_path`

str – der Pfad zum parser file

`formatter_test_file_path`

str – der Pfad zum formatter test file

`parser_test_file_path`

str – der Pfad zum parser test file

`database_path`

str – der Pfad zum database file

`parser_init_file_path`

str – der Pfad zum parser init file

`formatter_init_file_path`

str – der Pfad zum formatter init file

SQLitePluginPathHelper[52]

Der Path Helper beinhaltet Variablen zur Hilfe für die korrekten Pfade zu den Files welche erstellt werden sollten. Und implementiert somit den BaseSQLitePluginPathHelper.

[51] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.bll.services.html#module-plasoscaffolder.bll.services.base_sqlite_plugin_path_helper für die genaue Beschreibung.

[52] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.bll.services.html#module-plasoscaffolder.bll.services.sqlite_plugin_path_helper für die genaue Beschreibung.

8.3.4 bll.services Modul - PluginHelper

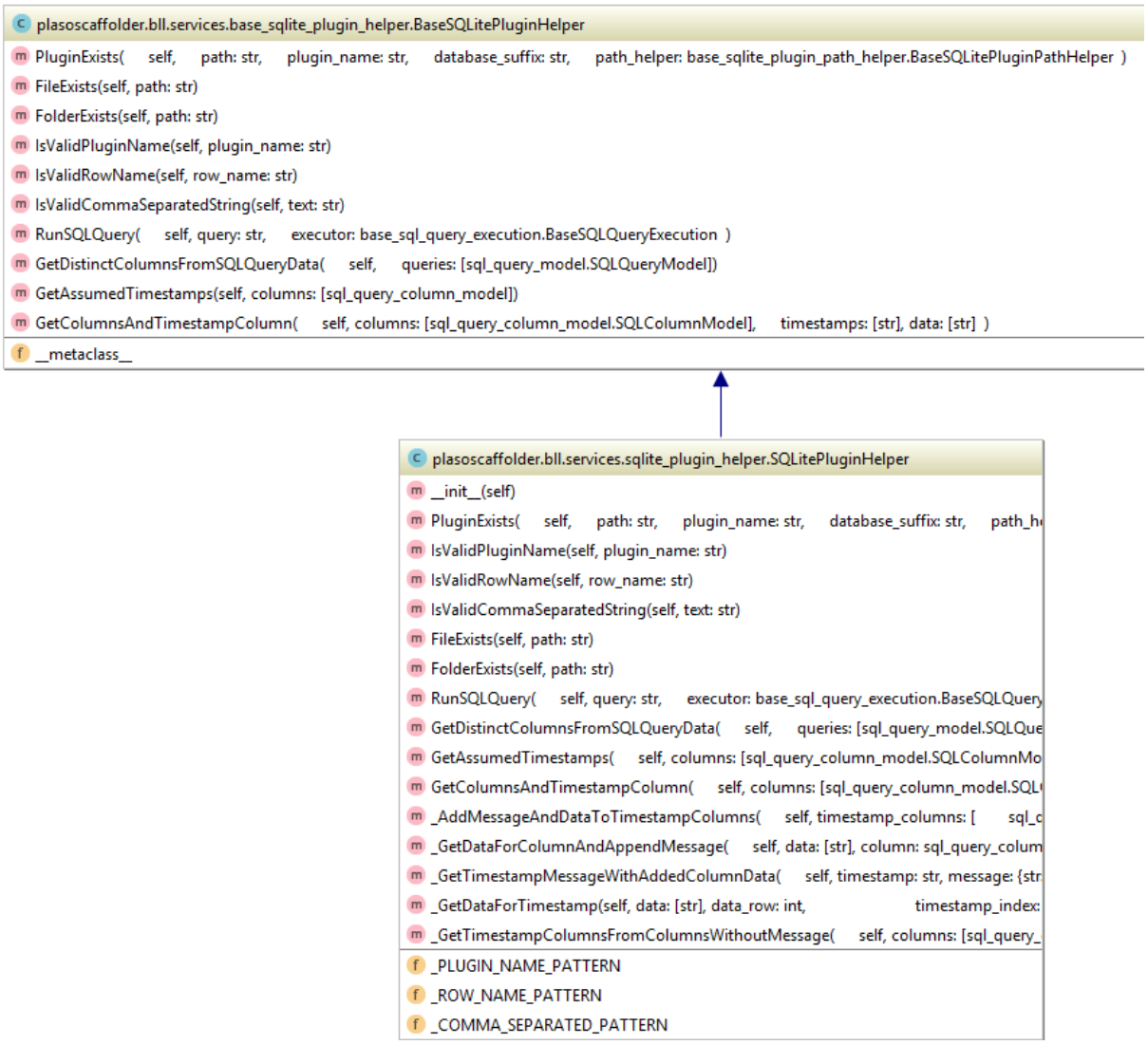


Abbildung 29: Klassendiagramm BLL PluginHelper

BaseSQLitePluginHelper[53]

Basis Klasse für den SQLitePluginHelper.

```
FileExists(path: str) → bool
```

Prüft ob das File existiert.
Parameters: path (str) – der File Pfad

```
FolderExists(path: str) → bool
```

Prüft ob der Ordner existiert.
Parameters: path (str) – der Ordner Pfad

[53] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.bll.services.html#module-plasoscaffolder.bll.services.base_sqlite_plugin_helper für die genaue Beschreibung.

```
GetAssumedTimestamps(columns: [<plasoscaffolder.model.sql_query_column_model>]) →
[<class 'str'>]
```

Sucht alle Columns bei welchen angenohmen wird, dass sie timestamps sind.

Parameters: columns ([sql_query_column_model]) – die Columns des Query

Returns: die Namen bei welchen angenommen wird, dass sie Timestamps sind

Return type: [str]

```
GetColumnsAndTimestampColumn(columns:
[<'plasoscaffolder.model.sql_query_column_model.SQLColumnModel'>], timestamps:
[<class 'str'>], data: [<class 'str'>]) -> ([<class
'plasoscaffolder.model.sql_query_column_model_data.SQLColumnModelData'>], [<class
'plasoscaffolder.model.sql_query_column_model_timestamp.SQLColumnModelTimestamp'>])
```

Teilt die columns Liste auf in eine Liste von normalen columns und eine liste für timestamp event columns und fügt Daten zu den normalen columns hinzu.

Parameters: columns ([SQLColumnModelData]) – die Columns vom SQL query
timestamps ([str]) – die Timestamp Events
data ([str]) – die Daten vom Cursor

Returns:
([sql_query_column_model_data.SQLColumnModelData],
[sql_query_column_model_timestamp.SQLColumnModelTimestamp]): Ein Tuple von Columns, das erste ist sind normale Columns, das zweite sind Timestamp Colums.

```
GetDistinctColumnsFromSQLQueryData(queries: [<class
'plasoscaffolder.model.sql_query_model.SQLQueryModel'>]) → [<class 'str'>]
```

Eine eindeutige Liste von allen Attributen der Queries

Parameters: queries ([sql_query_model.SQLQueryModel]) – Ein Array von SQL query data objects

Returns: Alle Attribute welche in den Queries gebraucht werden

Return type: list[str]

```
IsValidCommaSeparatedString(text: str) → bool
```

Validiert einen mit Komma separierten String

Parameters: text (str) – der zu validierende String.

Returns: true, wenn der Text korrekt ist.

Return type: bool

```
IsValidPluginName(plugin_name: str) → bool
```

Validiert den Plugin Name

Parameters: plugin_name (str) – Der Plugin Name

Returns: true, Wenn der Plugin Name korrekt ist.

Return type: bool

```
IsValidRowName(row_name: str) → bool
```

Validiert den Zeilen Name.

Parameters: row_name (str) – der Zeilen Name

Returns: true, wenn der Zeilen Name korrekt ist

Return type: bool

```
PluginExists(path: str, plugin_name: str, database_suffix: str, path_helper:
plasoscaffolder.bll.services.base_sqlite_plugin_path_helper.BaseSQLitePluginPathHel
per) → bool
```

Prüft ob das Plugin valide ist.

Parameters: database_suffix – der suffix des Datenbank Files.
path (str) – der Pfad zur Souce von Plaso
plugin_name (str) – der Name des Plugins
path_helper (BaseSQLitePluginPathHelper) – der SQLite plugin helper

Returns: True, wenn das Plugin bereits existiert. False, wenn das Plugin noch nicht existiert.

Return type: bool

```
RunSQLQuery(query: str, executor:
plasoscaffolder.dal.base_sql_query_execution.BaseSQLQueryExecution) →
plasoscaffolder.dal.sql_query_data.SQLQueryData
```

Validiert das SQL Query

Parameters:
executor (SQLiteQueryExecution) – der Executor, um das SQL Query auszuführen
query (str) – das SQL Query

Returns: Die Daten retourniert bei der Ausführung des Queries mit dem Executor.

Return type: sql_query_data.SQLQueryData

SQLitePluginHelper[54]

Die Klasse mit den Hilfsfunktionen für das SQLite Plugin und implentiert den BaseSQLitePluginHelper.

[54] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.bll.services.html#module-plasoscaffolder.bll.services.sqlite_plugin_helper für die genaue Beschreibung.

8.3.5 bll.services Modul - Generator

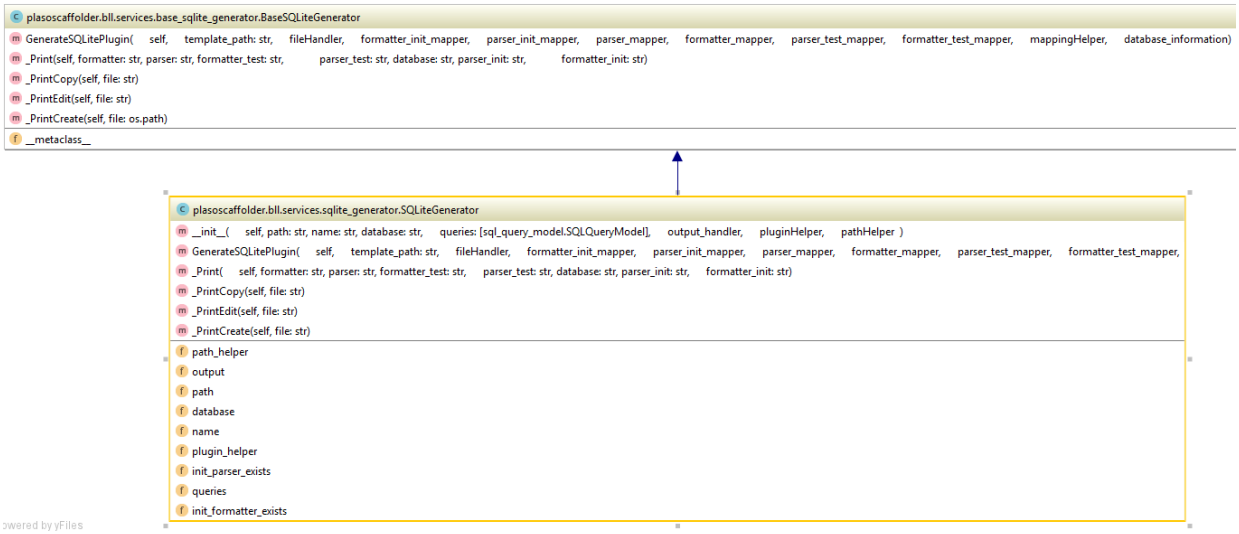


Abbildung 30: Klassendiagramm BLL SQLiteGenerator

BaseSQLiteGenerator[55]

Klasse repräsentiert die Basis Klasse für den SQLite Generator.

```
GenerateSQLitePlugin(template_path: str,
fileHandler: plasoscaffolder.common.base_file_handler.BaseFileHandler,
formatter_init_mapper: plasoscaffolder.bll.mappings.base_sqliteplugin_mapping.BaseSQLitePluginMapper,
parser_init_mapper: plasoscaffolder.bll.mappings.base_sqliteplugin_mapping.BaseSQLitePluginMapper,
parser_mapper: plasoscaffolder.bll.mappings.base_sqliteplugin_mapping.BaseSQLitePluginMapper,
formatter_mapper: plasoscaffolder.bll.mappings.base_sqliteplugin_mapping.BaseSQLitePluginMapper,
parser_test_mapper: plasoscaffolder.bll.mappings.base_sqliteplugin_mapping.BaseSQLitePluginMapper,
formatter_test_mapper: plasoscaffolder.bll.mappings.base_sqliteplugin_mapping.BaseSQLitePluginMapper,
mappingHelper: plasoscaffolder.bll.mappings.base_mapping_helper.BaseMappingHelper,
database_information: plasoscaffolder.dal.base_database_information.BaseDatabaseInformation)
```

Generiert das gesamte SQLite Plugin.

Parameters:
- template_path (str) – Der Pfad zum template Ordner
- fileHandler (base_file_handler.BaseFileHandler) – Der handler for das file
- formatter_init_mapper (BaseSQLitePluginMapper) – Der init formatter mapper
- parser_init_mapper (BaseSQLitePluginMapper) – Der init parser mapper
- parser_mapper (BaseSQLitePluginMapper) – Der parser mapper
- formatter_mapper (BaseSQLitePluginMapper) – Der mapper für den formatter
- parser_test_mapper (BaseSQLitePluginMapper) – Der mapper für den formatter test
- formatter_test_mapper (BaseSQLitePluginMapper) – Der mapper für den parser test
- mappingHelper (BaseMappingHelper) – Der mapping helper
- database_information (BaseDatabaseInformation) – Die Hilfeklasse für DB Infos

SQLiteGenerator[56]

Generator für die SQLiteFiles und implementiert den BaseSQLiteGenerator.

[55] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.bll.services.html#plasoscaffolder.bll.services.base_sqlite_generator für die genaue Beschreibung.

[56] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.bll.services.html#module-plasoscaffolder.bll.services.sqlite_generator für die genaue Beschreibung

8.3.6 Common Modul

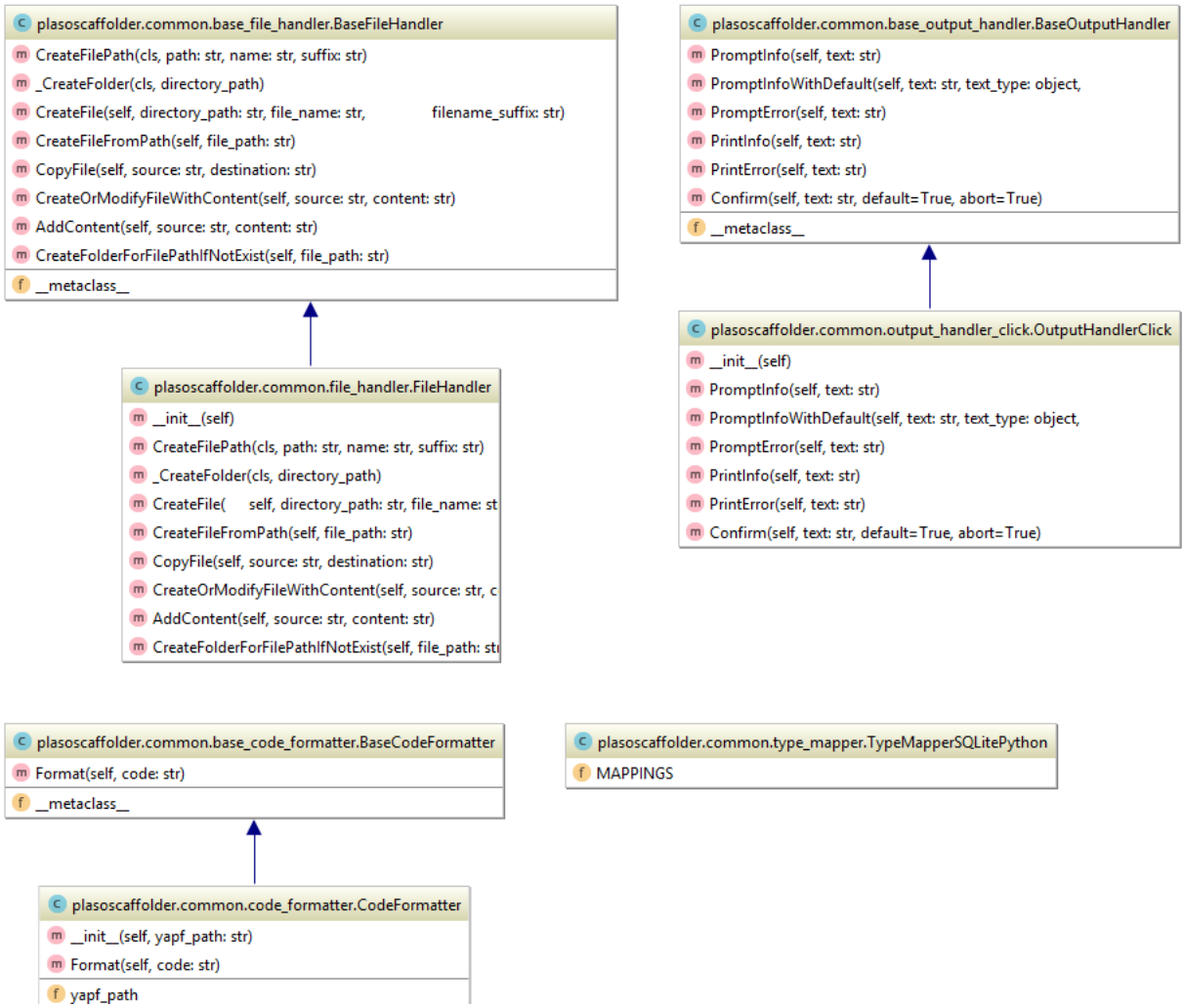


Abbildung 31: Klassendiagramm Common

BaseFileHandler[57]

Die Basis Klasse für den FileHandler.

`AddContent(source: str, content: str) → str`

Füge Daten einen File hinzu. Falls das File noch nicht existiert, wird das File zuerst erstellt.

Parameters: source (str) – Der Pfad zum File, welches editiert wird.
content (str) – Der Inhalt, welcher dem File angehängt werden soll

Returns: Der Pfad zum editierten File

Return type: str

`CopyFile(source: str, destination: str) → str`

Kopiert ein File.

Parameters: source (str) – Der Pfad zum File, welches kopiert werden soll.
destination (str) – Der Pfad zum File an welchen es kopiert werden soll.

Returns: Der Pfad zum neuen File.

Return type: str

[57] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.common.html#module-plasoscaffolder.common.base_file_handler für die genaue Beschreibung.

`CreateFile(directory_path: str, file_name: str, filename_suffix: str)`

Erstellt ein leeres File anhand des Pfades zum Ordner und dem File Namen sowie Suffix.

Parameters: directory_path (str) – Der Pfad zum Ordner in welchem das File erstellt werden soll
file_name (str) – Der Name des neuen Files.
filename_suffix (str) – Der suffix des neuen Files.

Returns: Der Pfad zum erstellten File

Return type: str

`CreateFileFromPath(file_path: str) → str`

Erstellt ein leeres File anhand des Pfades.

Parameters: file_path (str) – Der Pfad zum File.

Returns: Der Pfad zum erstellten File.

Return type: str

`classmethod CreateFilePath(path: str, name: str, suffix: str) → str`

Erstellt den Filepfad aus dem Ordnerpfad, dem Filenamen sowie dem Suffix.

Parameters: path (str) – Der Pfad zum Ordner
name (str) – Der Name des Files
suffix (str) – Das Suffix des Files

Returns: Der gesammte Pfad zum File

Return type: str

`CreateFolderForFilePathIfNotExist(file_path: str)`

Erstellt einen Ordner für den angegebenen Pfad, falls er noch nicht existiert.

Parameters: file_path (str) – Der Pfad zum File

Returns: Der Pfad zum erstellten oder existierenden Ordner

Return type: str

`CreateOrModifyFileWithContent(source: str, content: str)`

Füge Inhalt zum File hinzu falls das File schon existiert, oder erstelle ein neues File mit dem Inhalt falls es noch nicht existiert. Und erstelle den Ordner zum Filepfad, falls er noch nicht existiert.

Parameters: source (str) – Das File welches zu editieren oder erstellen und editieren ist.
content (str) – Der Inhalt welches in das File zu schreiben ist

FileHandler[58]

Diese Klasse ist für das Erstellen sowie Editieren für Files und Ordner zuständig und implementiert / erbt von der FileHandlerBase Klasse.

BaseOutputHandler[59]

Die Basis Klasse für den OutputHandler.

[58] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.common.html#module-plasoscaffolder.common.file_handler für die genaue Beschreibung.

[59] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.common.html#module-plasoscaffolder.common.base_output_handler für die genaue Beschreibung.

`Confirm(text: str, default=True, abort=True)`

Frage für eine Zustimmung oder Ablehnung zu einer Frage. Entweder y[es] oder n[o].

Parameters: text (str) – die Frage welche gestellt wird.
default (bool) – Der Default für die Frage. True für Y(es) und False für N(o)
abort (bool) – Ob das Programm abbrechen soll, falls die Antwort no (default) ist.

Returns: False wenn der User no eingegeben hat, True wenn der User yes eingegeben hat

Return type: bool

`PrintError(text: str)`

Zeige dem User einen Fehler an.

Parameters: text (str) – Der anzuzeigende Fehler.

`PrintInfo(text: str)`

Zeige dem User eine Information an

Parameters: text (str) – Die anzuzeigende Information.

`PromptError(text: str) → str[source]`

Frage den User um eine neue Eingabe da die vorherige Eingabe ungültig war.

Parameters: text (str) – Die anzuzeigende Frage.

Returns: die Eingabe des Users

Return type: str

`PromptInfo(text: str) → str`

Frage den User um eine Eingabe.

Parameters: text (str) – Die anzuzeigende Frage.

Returns: die Eingabe des Users

Return type: str

`PromptInfoWithDefault(text: str, text_type: object, default: object) → str`

Frage den User um eine Eingabe mit einem Standardwert zuwie einen vorgegebenen Typ.

Parameters: text (str) – der anzuzeigende Text
text_type (object) – der Typ welcher der Input haben muss
default (object) – der Standardwert

Returns: die Eingabe des Users

Return type: str

OutputHandlerClick[60]

Die Implementierung der OutputHandlerBase Klasse mit Hilfe des Click Packages.

BaseCodeFormatter[61]

Basis Klasse für den CodeFormatter.

[60] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.common.html#module-plasoscaffolder.common.output_handler_click für die genaue Beschreibung.

[61] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.common.html#module-plasoscaffolder.common.base_code_formatter für die genaue Beschreibung.

```
Format(code: str) → str
```

Formattiert den Code
Parameters: code (str) – Der zu formattierende Code.
Returns: der Formattierte Code
Return type: str

CodeFormatter[62]
Die Implementierung der CodeFormatterBase Klasse mit Hilfe des Yapf Packages.

TypeMapperSQLitePython[63]
Mappt die Datentypen von SQLite zu Python.
Die Mapping Information wurde von der Seite https://sqlite.org/datatype3.html genommen.

[62] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.common.html#module-plasoscaffolder.common.code_formatter für die genaue Beschreibung.
[63] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.common.html#module-plasoscaffolder.common.type_mapper für die genaue Beschreibung.

8.3.7 DAL Modul - TypeHelper

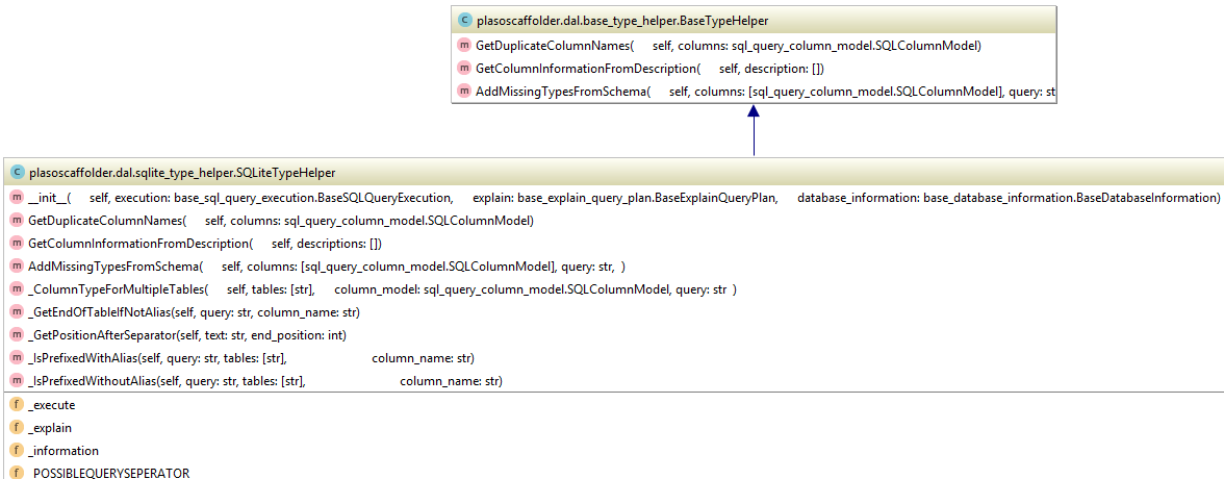


Abbildung 32: Klassendiagramm DAL TypeHelper

BaseTypeHelper[64]

Repräsentiert die Basisklasse für den Type Helper für SQLite.

```
AddMissingTypesFromSchema(columns: [<SQLColumnModel>], query: str) →
[<SQLColumnModel>]
```

Parst Informationen für die Typen aus dem Schema mit Hilfe des Queries.

Parameters: columns ([SQLColumnModel]) – Alle Zeilen mit den Zeilennamen
query – das Query
Returns: eine Liste mit allen Zeilen sowie den Typen.
Return type: [sql_query_column_model.SQLColumnModel]

```
GetColumnInformationFromDescription(description: []) → [<SQLColumnModel>]
```

Parst Informationen für die Result Set Zeilen aus der Cursor Beschreibung.

Parameters: description – die Cursor Beschreibung des ausgeführten Queries.
Returns: Eine List mit allen Namen der Zeilen ohne die Typen
Return type: [sql_query_column_model.SQLColumnModel]

```
GetDuplicateColumnNames(columns:
plasoscaffolder.model.sql_query_column_model.SQLColumnModel) → [str]
```

Findet heraus ob doppelte Zeilen Namen verwendet werden und somit ein Alias angegeben werden muss.

Parameters: columns (sql_query_column_model.SQLColumnModel) – alle Zeilen vom Cursor.
Returns: eine Liste mit allen doppelten Zeilennamen, und leer falls distinct.
Return type: [str]

SQLiteTypeHelper[65]

Ist zuständig für das Herausfinden des Typen der Zeilen und beinhaltet somit Hilfsmethoden für die Erstellung des Plugins. Es implementiert den BaseTypeHelper.

[64] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.dal.html#module-plasoscaffolder.dal.base_type_helper für die genaue Beschreibung

[65] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.dal.html#module-plasoscaffolder.dal.sqlite_type_helper für die genaue Beschreibung.

8.3.8 DAL Modul - Query

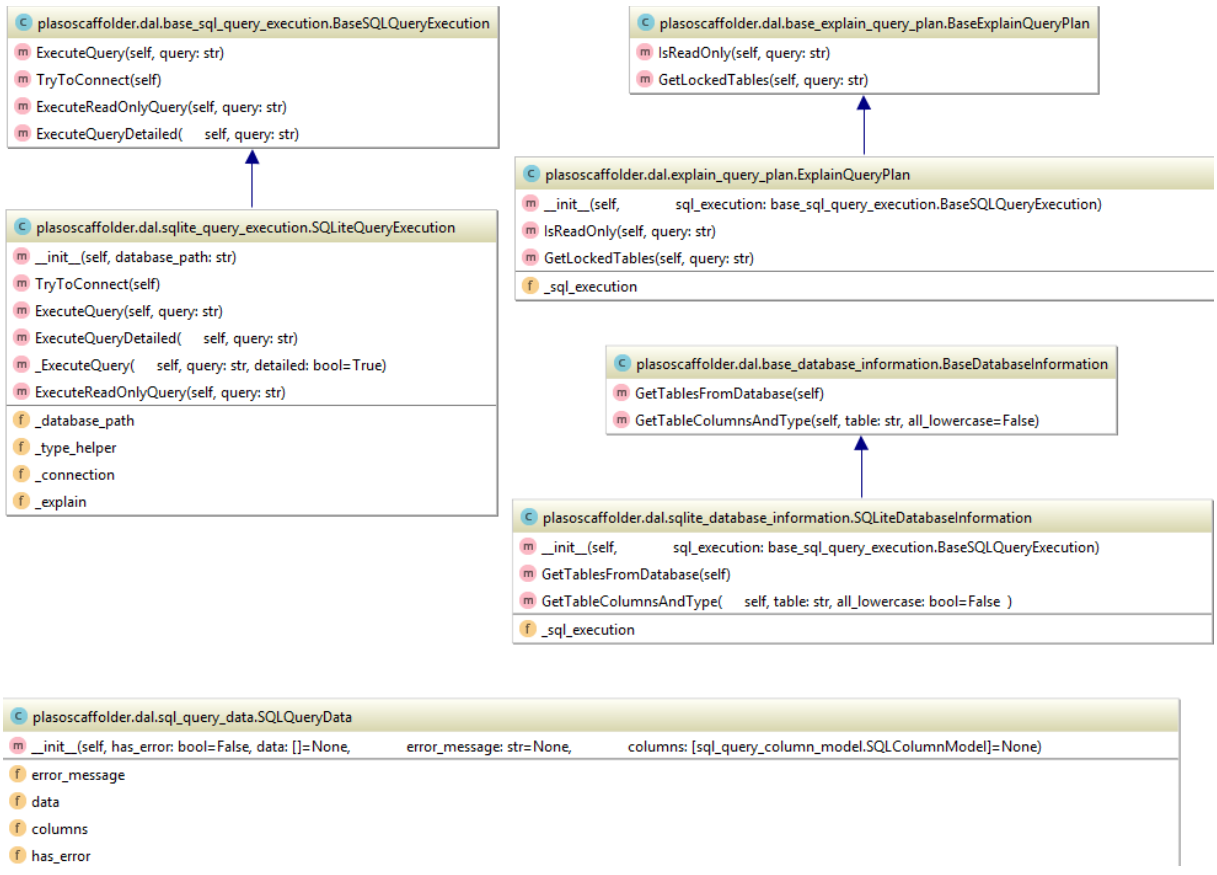


Abbildung 33: Klassendiagramm DAL Query

BaseSQLQueryExecution[66]

Ist die Basis Klasse für die SQLiteQueryExecution Klasse.

`ExecuteQuery(query: str) → plasoscaffolder.dal.sql_query_data.SQLQueryData`

Führt den SQL Query aus und retourniert die Daten im SQLQueryData Modul, und informiert somit über die resultierenden Daten oder den Fehler.

Parameters: query (str) – Das SQL Query welches auf der SQLite Datenbank auszuführen ist.
Returns: Die Daten zum ausgeführten SQL Query.
Return type: sql_query_data.SQLQueryData

`ExecuteQueryDetailed(query: str) → plasoscaffolder.dal.sql_query_data.SQLQueryData`

Führ das SQL Query aus und sammelt zusätzliche detaillierte Informationen dazu.

Parameters: query (str) – Das SQL Query welches auf der SQLite Datenbank auszuführen ist.
Returns: Die Daten zum ausgeführten SQL Query.
Return type: sql_query_data.SQLQueryData

`ExecuteReadOnlyQuery(query: str) → plasoscaffolder.dal.sql_query_data.SQLQueryData`

Führt das SQL Query auf der Datenbank aus und retourniert die Daten falls es valide ist, und ein reines Select Statement, somit readonly.

Parameters: query (str) – Das SQL Query welches auf der SQLite Datenbank auszuführen ist.
Returns: Die Daten zum ausgeführten SQL Query.
Return type: sql_query_data.SQLQueryData

[66] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.dal.html#module-plasoscaffolder.dal.base_sql_query_execution für die genaue Beschreibung

`TryToConnect() → bool`

Versucht sich mit dem Datenbankfile zu verbinden. Muss am Start vor anderen Methoden dieser Klasse erfolgreich durchgeführt werden.

Returns: True wenn das File geöffnet werden kann und es ein gültiges SQLite Datenbank File ist.

Return type: bool

SQLiteQueryExecution[67]

Die Klasse ist zuständig für das Ausführen sowie das Validierung der Queries und implementiert die BaseSQLQuery Execution Klasse.

BaseExplainQueryPlan[68]

Die Klasse ist die Basis Klasse für ExplainQueryPlan und führt Queries mit Hilfe des explain Statements von SQLite aus, welches Informationen über das ausgeführte Query liefert.

`GetLockedTables(query: str) → [<class 'str'>]`

Ermittelt die Tabellen welche vom SQL Query tangiert werden.

Parameters: query (str) – Das Query für welche die Tabellennamen ermittelt werden soll.

Returns: Eine Liste von Namen der Tabellen

Return type: [str]

`IsReadOnly(query: str) → bool`

Ermittelt ob das Query ein reines Select Statement und somit read-only ist.

Parameters: query (str) – Das Query für welches ermittelt werden sol lob es read-only ist.

Returns: True falls es read-only ist, False wenn nicht.

Return type: bool

ExplainQueryPlan[69]

Implementiert die BaseExplainQueryPlan Klasse mit Hilfe des sqlite3 Packages.

BaseDatabaseInformation[70]

Ist die Basisklasse von der Database-Information Klasse und ist zuständig um allgemeine Daten über die Datenbank herauszufinden.

`GetTableColumnsAndType(table: str, all_lowercase: bool = False) → [<class 'str'>]`

Ermittelt alle die Typen aller Zeilen der Datenbank für eine spezifische Tabelle.

Parameters:

table (str) – der Name der Tabelle

all_lowercase (bool) – True falls alle Namen und Typen in Kleinbuchstaben retourniert werden sollen.

Returns: alle Zeilen für die Tabellen mit ihrem Namen und Types.

Return type: {name, type}

[67] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.dal.html#module-plasoscaffolder.dal.sqlite_query_execution für die genaue Beschreibung.

[68] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.dal.html#module-plasoscaffolder.dal.base_explain_query_plan für die genaue Beschreibung

[69] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.dal.html#module-plasoscaffolder.dal.explain_query_plan für die genaue Beschreibung

[70] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.dal.html#module-plasoscaffolder.dal.base_database_information für die genaue Beschreibung

`GetTablesFromDatabase() → [<class 'str'>]`

Ermittelt alle Tabellen von der Datenbank.

Returns: Die Namen der Tabellen der Datenbank

Return type: [str]

SQLiteDatabaseInformation[71]

Implementiert die Klasse BaseSQLiteDatabaseInformation mit Hilfe des sqlite3 Packages.

SQLQueryData[72]

Das Datenmodul ist alleine dafür zuständig die Daten zum ausgeführten Query strukturiert zurück zu geben.

`has_error`

bool – True falls das Query einen Fehler hatte. False falls das Query fehlerfrei ausgeführt wurde.

`data`

[str] – Die Datenzeile für das ausgeführte Query, falls die Ausführung keinen Fehler hatte, sonst None.

`error_message`

str – Die Fehlermeldung für das Query, falls es nicht fehlerfrei ausgeführt wurde.

`columns`

[sql_query_column_model.SQLColumnModel] – Die Namen der Zeilen welche das Query beinhaltet.

[71] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.dal.html#module-plasoscaffolder.dal.sqlite_database_information für die genaue Beschreibung.

[72] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.dal.html#plasoscaffolder-dal-sql-query-data-module für die genaue Beschreibung.

8.3.9 Frontend Modul

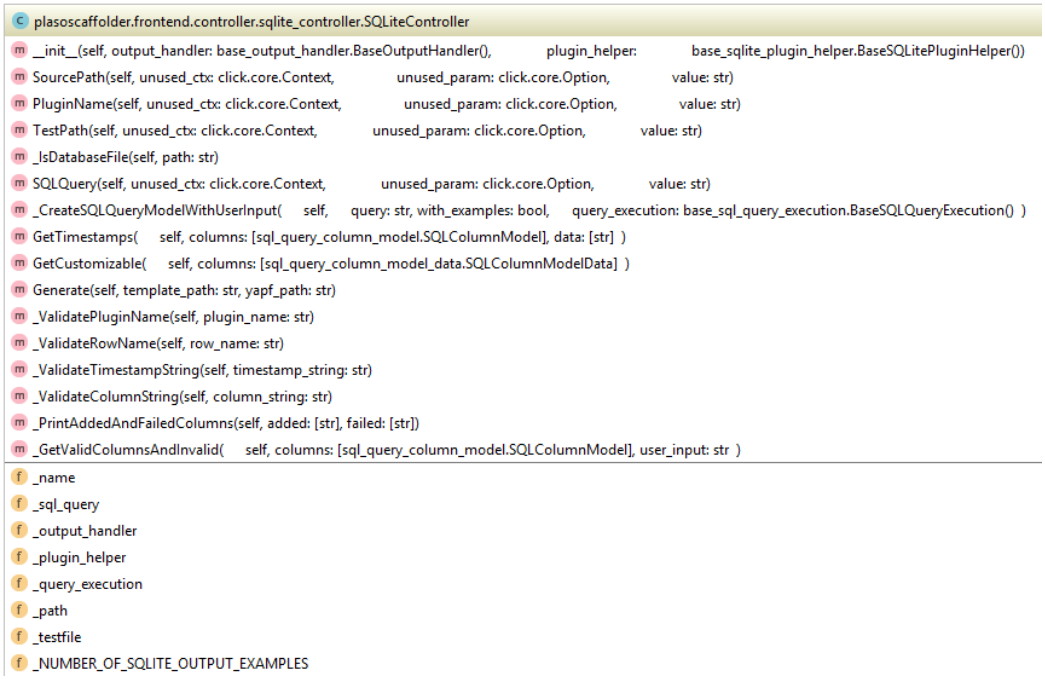


Abbildung 34: Klassendiagramm Frontend

SQLiteController[73]

Repräsentiert den Controller für die SQLite Plugins und verarbeitet die Eingaben des Users vor dem Generieren des Plugins.

```
Generate(template_path: str, yapf_path: str)
```

Generiert die Files

Parameters: template_path (str) – der Pfad zum Ordner mit den Templates.
yapf_path (str) – der Pfad zum yapf Konfigurationsfile

```
GetCustomizable(columns: [<SQLColumnModelData>]) → [<SQLColumnModelData>]
```

Fragt den User nach den Zeilen welche Anpassungen brauchen.

Parameters: columns ([SQLColumnModelData]) – die Zeilen vom SQL Query
Returns: die Zeilen mit der Information ob die Zeile Anpassungen braucht.
Return type: [sql_query_column_model_data.SQLColumnModelData]

```
GetTimestamps(columns: [<SQLColumnModel>], data: [str]) -> ([<SQLColumnModelData>],
[<SQLColumnModelTimestamp>])
```

Fragt den User nach den Timestamps für die Zeilen des Queries.

Parameters: columns ([sql_query_column_model.SQLColumnModel]) – die Zeilen vom SQL Query
data ([str]) – die Daten vom cursor
Returns: [sql_query_column_model_data.SQLColumnModelData], [sql_query_column_model_timestamp.SQLColumnModelTimestamp]: Ein tuple von Zeilen. Die erste Liste repräsentiert die normalen Datenzeilen, die zweite Liste repräsentiert die Zeilen welche Timestamps sind.

73 http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.frontend.controller.html#module-plasoscaffolder.frontend.controller.sqlite_controller für die genaue Beschreibung.

`PluginName(unused_ctx: click.core.Context, unused_param: click.core.Option, value: str) → str`

Fragt den User nach dem Namen des Plugins.

Parameters: unused_ctx (click.core.Context) – der Click context (automatisch via callback)
unused_param (click.core.Option) – der Click command (automatisch via callback)
value (str) – der Plugin Name (automatisch via callback)

Returns: Der validierte Name des Plugins

Return type: str

`SQLQuery(unused_ctx: click.core.Context, unused_param: click.core.Option, value: str) → [<class 'plasoscaffolder.model.sql_query_model.SQLQueryModel'>]`

Fragt den User nach den SQL Queries des Plugins.

Parameters: unused_ctx (click.core.Context) – der Click context (automatisch via callback)
unused_param (click.core.Option) – der Click command (automatisch via callback)
value (str) – ob er Informationen sehen möchte(automatisch via callback)

Returns: Eine Liste von SQL Queries

Return type: [sql_query_model.SQLQueryModel]

`SourcePath(unused_ctx: click.core.Context, unused_param: click.core.Option, value: str) → str`

Fragt den User nach dem Source Pfad von Plaso.

Parameters: unused_ctx (click.core.Context) – der Click context (automatisch via callback)
unused_param (click.core.Option) – der Click command (automatisch via callback)
value (str) – der Source Pfad von Plaso (automatisch via callback)

Returns: Der validierte Source Pfad von Plaso.

Return type: str

`TestPath(unused_ctx: click.core.Context, unused_param: click.core.Option, value: str) → str`

Fragt den User nach dem Datenbankfile

Parameters: unused_ctx (click.core.Context) – der Click context (automatisch via callback)
unused_param (click.core.Option) – der Click command (automatisch via callback)
value (str) – der Pfad zum Datenbank File (automatisch via callback)

Returns: Der validierte Pfad zum Datenbank File

Return type: str

8.3.10 Model Modul - DataModel

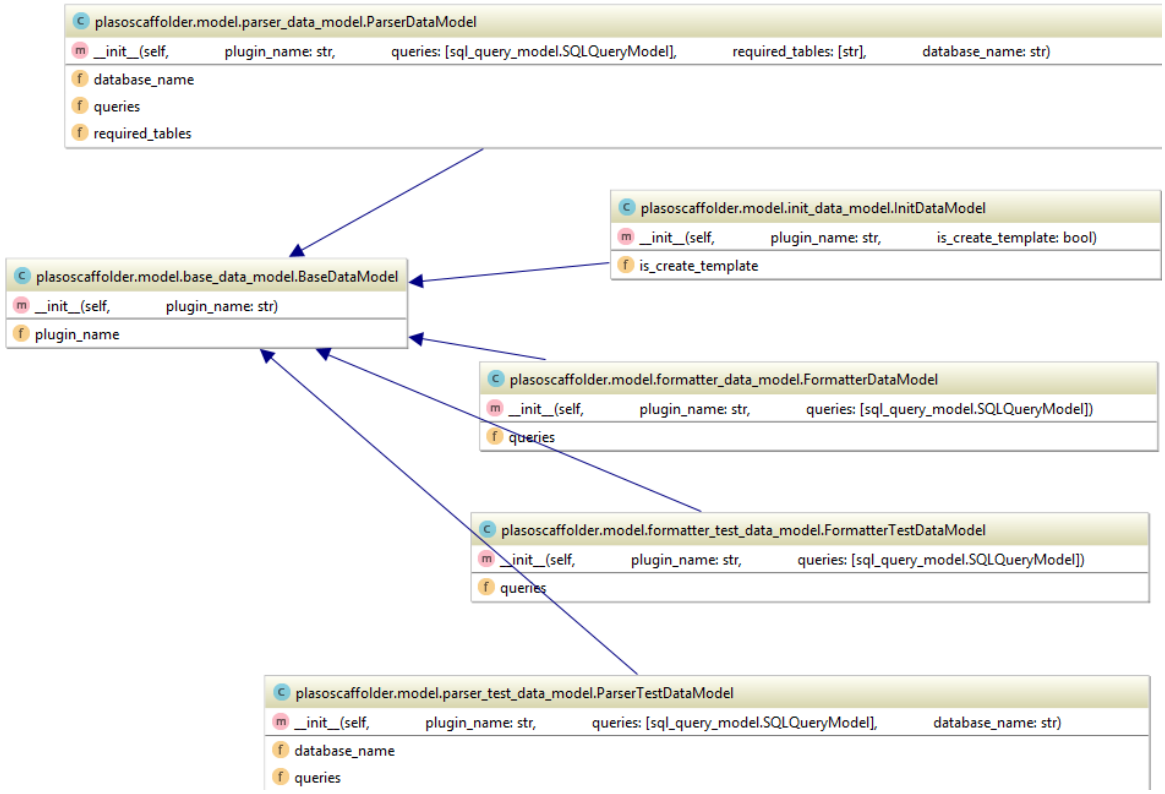


Abbildung 35: Klassendiagramm Model DataModel

BaseDataModel[74]

Das BaseDataModel ist die Basisklasse aller Datenmodel Module. Es binhaltet den Plugin

ParserDataModel[75]

Parser Datenmodul, welches zusätzlich zum Name des Plugins noch die Queries, die Tabellen der Datenbank, und den Datenbankfile Name beinhaltet. Diese Daten werden für das Template für das Parser File benötigt.

InitDataModel[76]

Init Datenmodul, welches zusätzlich zum Plugin Name noch beinhaltet ob das File existiert oder neu erstellt werden muss. Diese Daten werden für das Template der Init Files benötigt.

FormatterDataModel[77]

Formatter Datenmodul, welches zusätzlich zum Plugin Name noch die Queries, beinhaltet. Diese Daten werden für das Template für das Formatter File benötigt.

[74] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.model.html#module-plasoscaffolder.model.base_data_model für die genaue Beschreibung.

[75] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.model.html#module-plasoscaffolder.model.parser_data_model für die genaue Beschreibung

[76] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.model.html#module-plasoscaffolder.model.init_data_model für die genaue Beschreibung

[77] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.model.html#module-plasoscaffolder.model.formatter_data_model für die genaue Beschreibung

FormatterTestDataModel[78]

Formatter Test Daten Modul, welches zusätzlich zum Plugin Name noch die Queries, beinhaltet. Diese Daten werden für das Template für das Formatter Test File benötigt.

ParserTestDataModel[79]

Parser Test Daten Modul, welches zusätzlich zum Plugin Name noch die Queries und der Datenbankfile Name beinhaltet. Diese Daten werden für das Template des Parser Test File benötigt.

8.3.11 Model Modul - SQLColumnModel

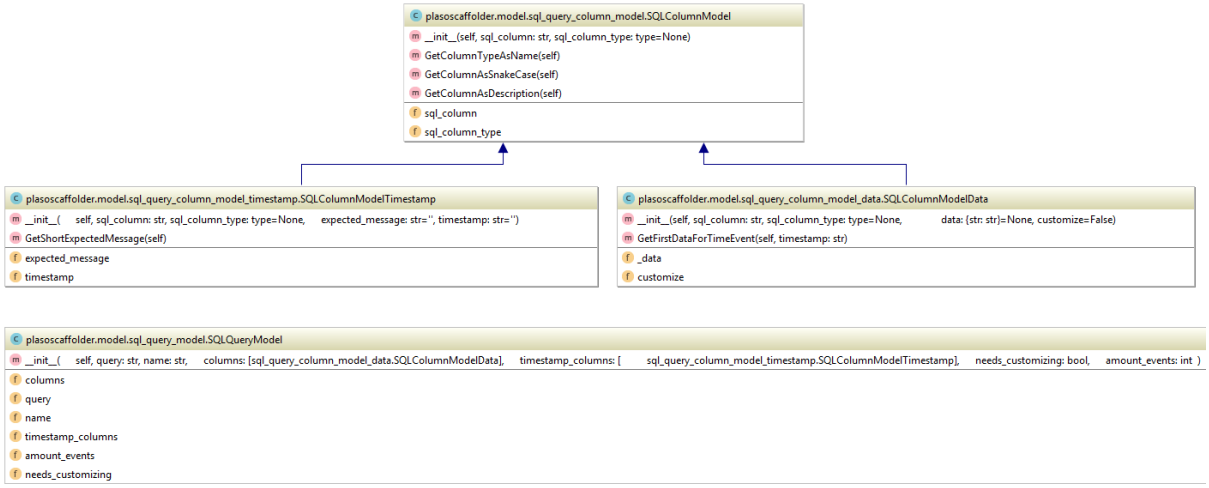


Abbildung 36: Klassendiagramm Model ColumnModel

SQLColumnModel[80]

Daten Model für die Queries, welche alle konkreten SQLColumModel beinhalten. Dies beinhaltet den Namen der Zeile sowie deren Datentyp und die folgenden Methoden.

`GetColumnAsDescription() → str`

Generiert aus dem Namen der Zeile eine Beschreibung.

Returns: der Name der Zeile in Beschreibungsform

Return type: str

`GetColumnAsSnakeCase() → str`

Generiert aus dem Zeilennamen einen Namen in SnakeCase, welcher als Variable verwendet wird.

Returns: der Zeilennamen in SnakeCase

Return type: str

`GetColumnTypeAsName() → str`

Ermittelt die Beschreibung für den Typen der Zeile. Beispiel: <class 'int'> wird als int retourniert.

Returns: der Type als Name

Return type: str

[78] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.model.html#module-plasoscaffolder.model.formatter_test_data_model für die genaue Beschreibung.

[79] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.model.html#module-plasoscaffolder.model.parser_test_data_model für die genaue Beschreibung.

[80] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.model.html#module-plasoscaffolder.model.sql_query_column_model für die genaue Beschreibung.

SQLColumnModelTimestamp[81]

Erbt vom SQLColumnModel und beinhaltet zusätzlich noch die erwartete Nachricht für den Event des Timestamps sowie eine Methode für die Kurzversion der Nachricht.

```
GetShortExpectedMessage() → str
```

Ermittelt die kurze Version der Nachricht. Welche höchstens 80 Buchstaben beinhaltet.

Returns: die kurze Version der Nachricht

Return type: str

SQLColumnModelData[82]

Erbt vom SQLColumnModel und beinhaltet zusätzlich noch die Daten zur Zeile sowie ob diese Zeile Anpassungen braucht und eine Methode um für die erste Datenzeile für einen Timestamp welche im ParserTest Template benötigt wird.

```
GetFirstDataForTimeEvent(timestamp: str) → str
```

Ermittelt die Daten für ein Timestamp.

Parameters: timestamp (str) – Der Zeilennamen des Timestamp

Returns: die Daten zum Timestamp

Return type: str

SQLQueryModel[83]

Die Model Klasse für die SQL Queries. Dieses beinhaltet die Daten für das Query, den Namen für das Query, die Zeilen des Queries der normalen Daten, die Zeilen des Queries für die Timestamps, ob das Query Anpassungen braucht, sowie die Anzahl von Events welche sich für das Query in der Testdatenbank befinden.

[81] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.model.html#module-plasoscaffolder.model.sql_query_column_model_timestamp für die genaue Beschreibung.

[82] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.model.html#plasoscaffolder.model.sql_query_column_model_data.SQLColumnModelData für die genaue Beschreibung.

[83] http://plasoscaffolder.readthedocs.io/en/latest/plasoscaffolder.model.html#module-plasoscaffolder.model.sql_query_model für die genaue Beschreibung.

8.4 Datenbank

Die Datenbank ist eine beliebige SQLite Datenbank, auf welche das Programm mit Hilfe einer Schnittstelle im sqlite3 Modul zugreift.

Um die Queries zu validieren und dem User Resultate die Bespieldaten anzuzeigen, werden die Queries vor dem Hinzufügen zu Plaso, ausgeführt.
Da alle Queries vom User auf der Datenbank laufen gelassen werden, werden diese Queries innerhalb einer Transaktion ausgeführt, und anschliessend wird ein Rollback der Datenbank ausgeführt.

```
connection = sqlite3.connect('some path')
connection.execute('BEGIN')
connection.execute('query')
connection.execute('ROLLBACK')
```

Um sicherzugehen, dass das File welches der User angegeben hat eine gültige SQLite Datenbank ist, wird zuerst darauf ein Query ausgeführt. Dies ist notwendig, da ein connect auf ein File nicht fehlschlägt, wenn es keine SQLiteDatenbank ist. Auch ist die Validierung des Pfades wichtig, da bei einem connect auf ein nichtexistierendes File, automatisch ein File erstellt wird.

```
connection.execute('PRAGMA schema_version')
```

Da PRAGMA ein SQLite spezifisches Statement sowie nicht kompatibel mit anderen Datenbanken ist, kann damit zuverlässig getestet werden, ob die Datenbank eine SQLite Datenbank ist.

8.5 Ablauf

Hier wird der grobe Ablauf des ganzen Programmes mit Hilfe von Sequenzdiagrammen beschrieben. Mehr zu seqdiag unter Kapitel 6.4.

Folgend das Schichtenmodell mit den Modulen welche in den Sequenzdiagrammen abgebildet werden.

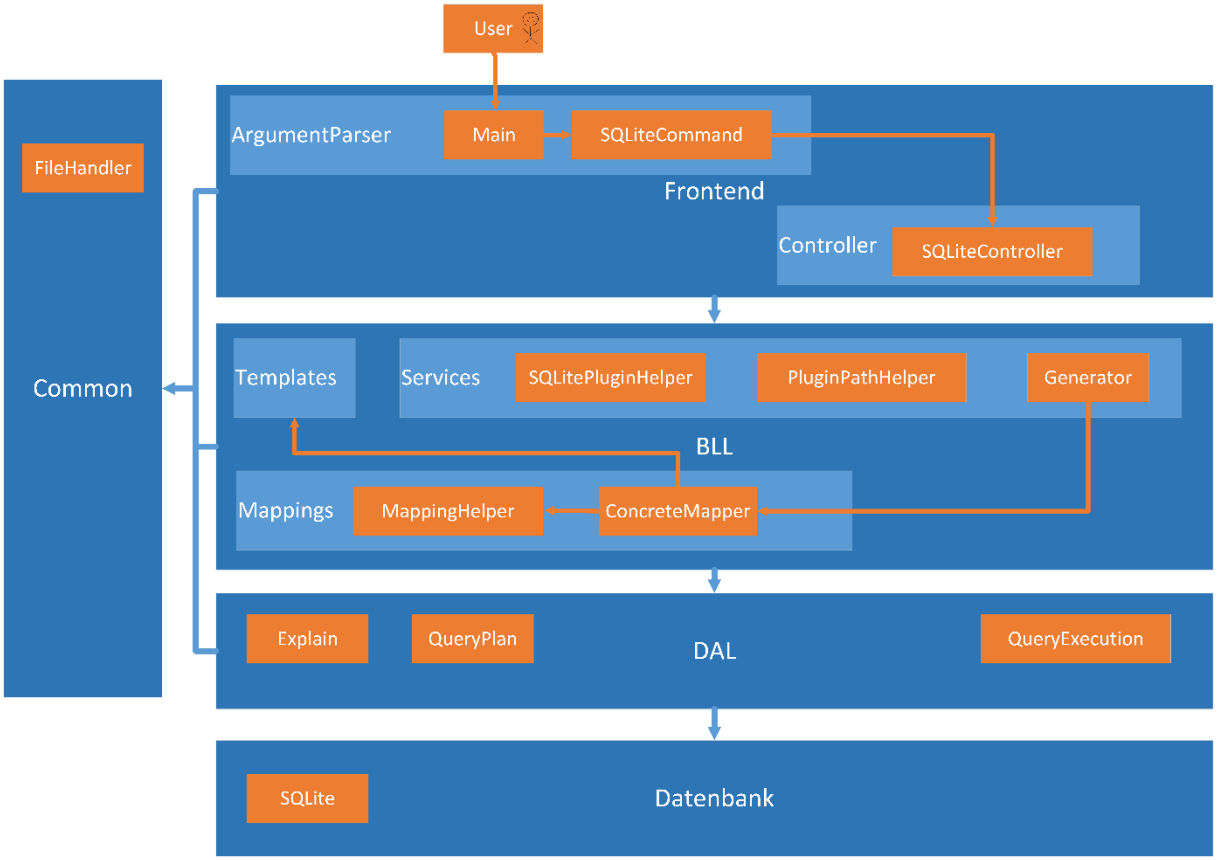


Abbildung 37: Schichtenmodell der in den Sequenzdiagrammen abgebildeten Module

User	Der User, welcher das Plugin ausführt.
Main	Das Hauptprogramm, welches den Einstiegspunkt zum Scaffolder abbildet. Ist im Frontend Layer vorhanden.
SQLiteCommand	Der Einstiegspunkt für das scaffolden eines SQLite Plugins. Ist im Frontend Layer vorhanden.
SQLiteController	Der Controller für das SQLite Plugin, welche die Logik des SQLiteCommand beinhaltet, und nur vom SQLiteCommand aufgerufen wird. Ist im Frontend Layer unter den Controller vorhanden.
SQLitePluginHelper	Der Plugin Helper für die SQLite Plugins, beinhaltet Hilfsklassen welche vom SQLiteController aufgerufen werden. Ist im BLL unter den Services vorhanden.
PluginPathHelper	Der Plugin Path Helper beinhaltet die Pfade zu den verschiedenen Files welche generiert, editiert oder kopiert werden. Ist im BLL unter Services vorhanden.
Generator	Der Generator wird aufgerufen nachdem alle Daten vom User aufgenommen wurden, um das Plugin zu generieren.

ConcreteMapper	Das Modul stellt die verschiedenen konkreten Mapper für die zu generierenden Files dar. Es existiert für jedes File ein eigener Mapper.
MappingHelper	Der MappingHelper besitzt Methoden, welche von allen konkreten Mapper benötigt werden.
Explain	Das Explain Modul besitzt Methoden welche sich dem 'explain' Keyword von SQLite bedienen. Um Informationen über das Query herauszufinden.
QueryPlan	Das QueryPlan Modul besitzt Methoden welche sich dem 'query plan' Keyword von sqlite bedienen. Um Informationen über die Ausführung des Query herauszufinden.
QueryExecution	Das QueryExecution Modul führt Queries auf der Datenbank aus und liest je nach Methode zusätzliche Daten vom ausgeführten Query aus.
SQLite	Das SQLite Modul ist eine Abhängigkeit mit welchem ich Queries auf der SQLite Datenbank ausführe.
FileHandler	Der FileHandler ist eines der common Module welches für die Erstellung der Files benutzt wird.

Tabelle 11: Beschreibung der in den Sequenzdiagrammen abgebildeten Module

8.5.1 Übersicht

Folgend eine abstrahierte Übersicht der Interaktion zwischen User und den Modulen Main, SqliteCommand sowie SQLiteController. Jeder der Aufrufe von SQLiteCommand zu SQLiteController wird noch detailliert erläutert.

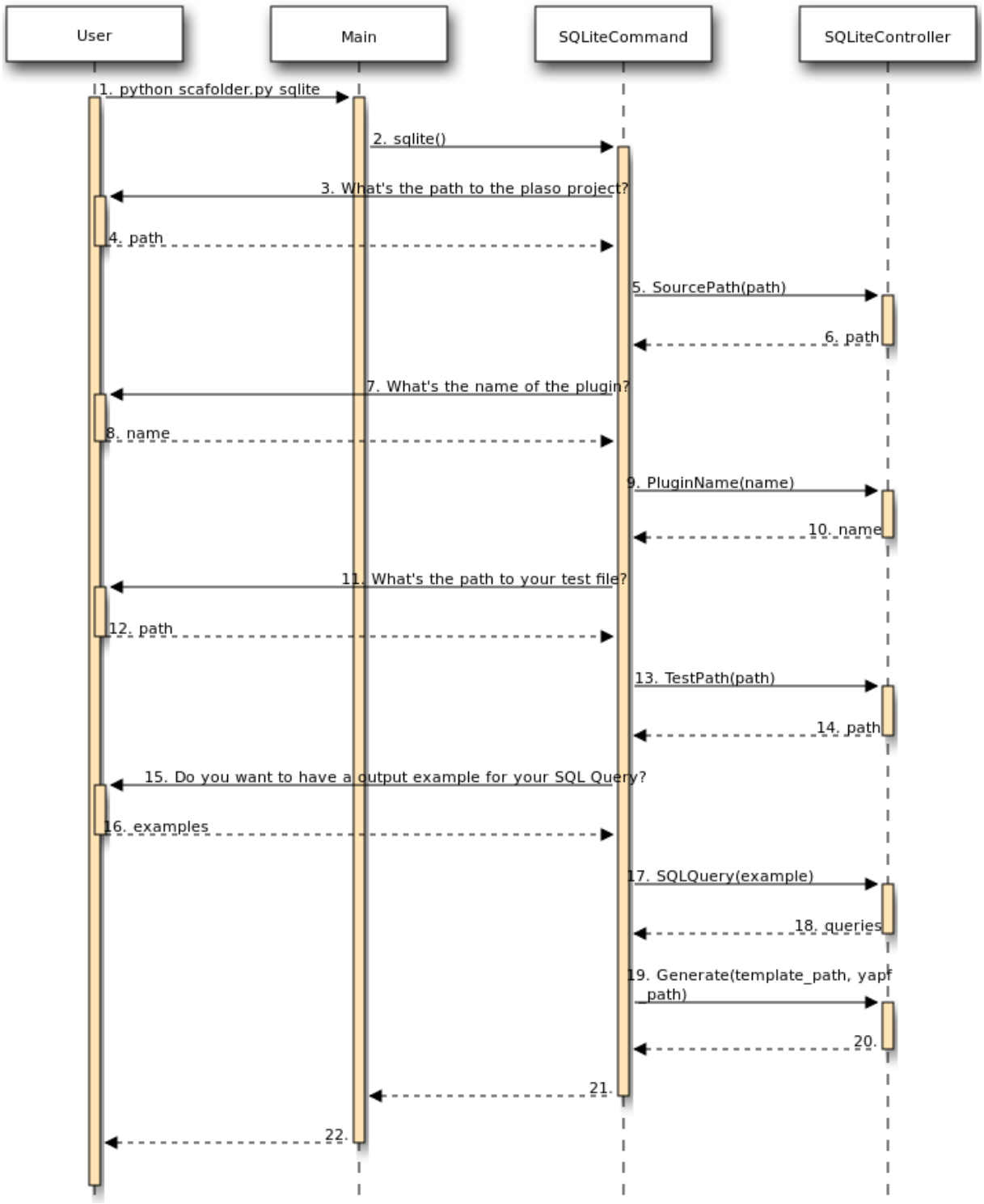


Abbildung 38: Sequenzdiagramm einer Übersicht

1. Der User ruft das Programm mit dem Command ‘sqlite’ auf.
2. Der Startpunkt des Programmes ruft aufgrund des Commands ‘sqlite’ die Methode sqlite() auf, welcher den Startpunkt für SQLite Plugins darstellt.
3. SQLiteCommand fragt den User nach dem Pfad zum Plaso Projektordner.
4. Der User gibt den Pfad ein.
5. SQLiteCommand ruft die Methode SourcePath mit dem Pfad des Users beim SQLiteController auf.
6. SQLiteController verarbeitet die Eingabe und retourniert den Pfad. Der Ablauf ist ersichtlich unter: 8.5.3
7. SQLiteCommand fragt den User nach dem Namen des Plugins.
8. Der User gibt den Plugin Namen ein.
9. SQLiteCommand ruft die Methode PluginName mit dem Namen des Users beim SQLiteController auf.
10. SQLiteController verarbeitet die Eingabe und retourniert den Namen. Der Ablauf ist ersichtlich unter: 8.5.4
11. SQLiteCommand fragt den User nach dem Pfad zum Test File.
12. Der User gibt den Pfad ein.
13. SQLiteCommand ruft die Methode TestPath mit dem Pfad des Users beim SQLiteController auf.
14. SQLiteController verarbeitet die Eingabe und retourniert den Pfad. Der Ablauf ist ersichtlich unter: 8.5.5.
15. SQLiteCommand fragt den User ob er einige Zeilen des ResultSets für die SQL Queries sehen möchte.
16. Der User gibt die Antwort ein.
17. SQLiteCommand ruft die Methode SQLQuery mit der Antwort des Users beim SQLiteController auf.
18. SQLiteController verarbeitet die Eingabe und retourniert die Queries. Der Ablauf ist ersichtlich unter: 8.5.6
19. SQLiteCommand ruft die Methode Generate mit dem Template Pfad und dem yapf Pfad beim SQLiteController auf.[84]

[84] Erläuterung warum die Pfade beim SQLiteCommand Modul mitgegeben werden unter: 5.9.3

8.5.2 Start

Hier wird der Start der Applikation aufgezeigt.

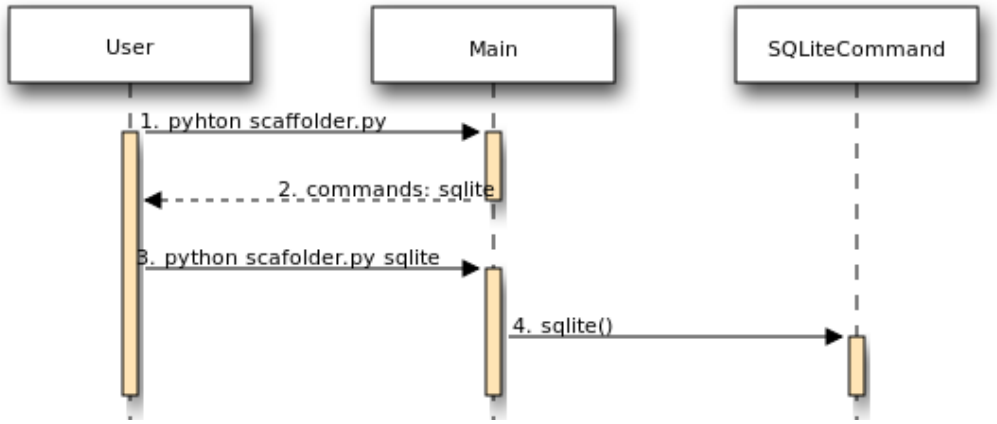


Abbildung 39: Sequenzdiagramm vom Start

1. Der User ruft die Applikation ohne Optionen auf.
2. Das Main retourniert dem User eine Übersicht über die verschiedenen Commands, welche er benutzen kann um verschiedene Plugins zu generieren. Im Moment existiert nur die Option 'sqlite'.
3. Der User ruft die Applikation mit der Option sqlite auf
4. Das Main ruft mit der sqlite() Methode das Modul SQLiteCommand auf, welches der SQLite Plugin Scaffolder ist.

Da die Erweiterung des Plaso Scaffolders mit weiteren Plugin Generatoren möglich sein muss, ist die Designentscheidung gefallen, diese als Commands am Start der Applikation zu definieren. Somit sind die verschiedenen Plaso Plugin Scaffolders sauber voneinander abgetrennt und trotzdem über einen gemeinsamen Einstiegspunkt erreichbar.

8.5.3 SourcePath

Dieses Sequenzdiagramm beschreibt den Ablauf für die Eingabe des Pfades. Der Pfad wird benötigt, damit man weiss wo das Originalprojekt Plaso liegt. Dies ist notwendig, damit am korrekten Ort innerhalb von Plaso die Files generiert und modifiziert werden können.

Damit man jedoch die Files nach Belieben auch an einem anderen Ort generieren kann, wird konkret geprüft ob es wirklich das Plaso Projekt ist, nur der Pfad muss existieren.

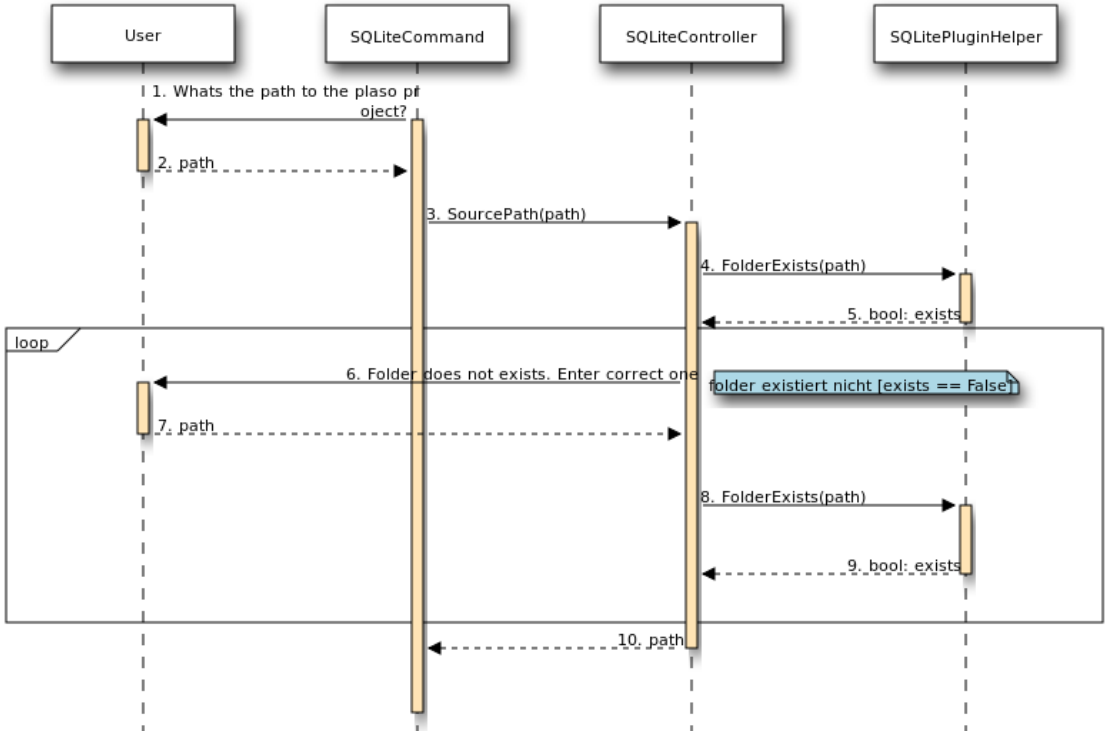


Abbildung 40: Sequenzdiagramm der Methode SourcePath()

1. Das SQLiteCommand Modul fragt den User nach dem Pfad zum Plaso Projekt.
2. Der User macht eine Eingabe und gibt den Pfad ein.
3. Das SQLiteCommand Modul ruft den SQLiteController mit dem Pfad des Users auf.
4. Der SQLiteController ruft die Methode FolderExists(path) des SQLitePluginHelpers auf, um zu prüfen ob der Order welcher im Pfad angegeben wurde existiert.
5. Der SQLitePluginHelper prüft ob der angegebene Plaso Ordner existiert und gibt den SQLiteController einen Boolean zurück.

Falls die Rückgabe des SQLitePluginHelper 'False' war und der Ordner nicht existiert wird solange wie die Variable 'exists' False ist, die folgende Sequenz wiederholt.

6. Der SQliteController gibt dem User Bescheid, dass der Order nicht existiert und fragt ihn nach einer weiteren Eingabe-
7. Der User gibt den neuen Pfad ein
8. Siehe 4.
9. Siehe 5.

Falls die Rückgabe des SQLitePluginHelper 'True' war und der Ordner existiert, wird hier fortgefahren.

10. Der SQLiteController retourniert dem SQLiteCommand Modul den Pfad und beendet somit die Validierung des Source Pfades.

8.5.4 PluginName

Dieses Sequenzdiagramm beschreibt den Ablauf für die Eingabe des Namens für das Plugins. Der Name des Plugins ist wichtig, da dieser Wert mehrmals für die Generierung verwendet wird. Siehe *PluginName* unter Kapitel 4, Plugin Aufbau.

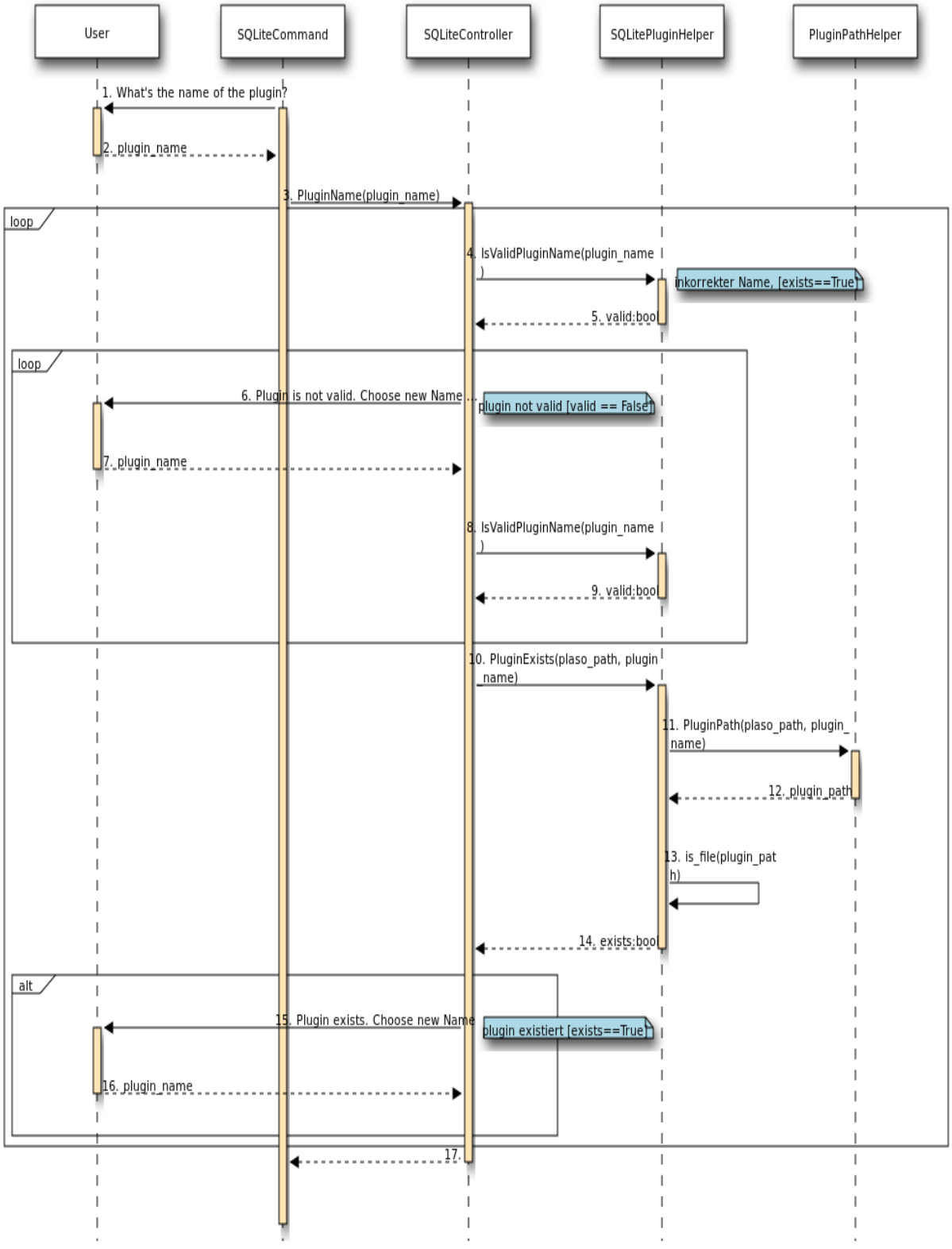


Abbildung 41: Sequenzdiagramm der Methode PluginName()

1. Das SQLCommand Modul fragt den User nach dem Namen des Plugin.
2. Der User gibt den Namen ein.
3. Das SQLiteCommand Modul ruft die Methode PluginName des SQLiteControllers auf um den Namen zu validieren.

Solange der Plugin Name nicht korrekt ist, wird 4-16 durchgeführt, so wie beim ersten Mal.

4. Der SQLiteController ruft die Methode IsValidPluginName vom SQLitePluginHelper auf damit mit einem regex Pattern geprüft wird, ob der Name in einem gültigen Format eingeben wurde.
5. Der SQLitePluginHelper retourniert einen Boolean. Welcher bei False bedeutet, dass der Pluginname nicht valide ist.

Solange der Pluginname kein gültiges Format hat wird 6-9 wiederholt.

6. Der SQLiteController fragt den User nach einem neuen Pluginnamen mit dem Hinweis wie das Format aussieht.
7. Der User gibt einen neuen Pluginnamen ein.
8. Siehe 4.
9. Siehe 5.

Wenn der Pluginname gültig ist und somit der SQLitePluginHelper True retourniert wird hier weitergearbeitet.

10. Der SQLiteController ruft die Methode PluginExists vom SQLitePluginHelper auf mit den Parameter des Plaso Pfades, welcher der User zuvor eingegeben hat, und dem Namen des Plugins.
11. Der SQLitePluginHelper ruft den PluginPathHelper auf. Um den Pfad der verschiedenen Files für das Plugin zu bekommen.
12. Der PluginPathHelper retourniert den Pfad der Files für das Plugin
13. Der SQLitePluginHelper prüft mit der privaten Hilfemethode is_file und Pfad als Parameter, ob eines der Files schon existiert.
14. Falls eines der Files existiert, retourniert der SQLitePluginHelper dem SQLiteController True.

Falls der SQLiteController True retourniert und somit der das Plugin existiert wird 15. Und 16 ausgeführt.

15. Der SQLiteController teilt dem User mit, dass schon ein Plugin mit dem angegebenen Pluginnamen existiert und fragt ihn nach einem neuen Namen
16. Der User gibt einen neuen Namen an. Und der Loop wird wiederholt, und somit bei 4. weitergearbeitet.

Falls der SQLiteController False retourniert und noch kein Plugin mit diesem Namen existiert wird hier fortgefahren.

17. Der SQLiteController gibt den Namen dem SQLiteCommand zurück und die Validierung ist abgeschlossen.

8.5.5 TestPath

Dieses Sequenzdiagramm beschreibt den Ablauf für die Eingabe des Pfades zur Testdatenbank. Die Testdatenbank ist wichtig, da der Pfad einerseits als Kommentar in den Files verwendet wird und andererseits mit der Datenbank verbunden wird um Schema Informationen für die Generierung der Files sowie Daten für den Parser Test auszulesen.

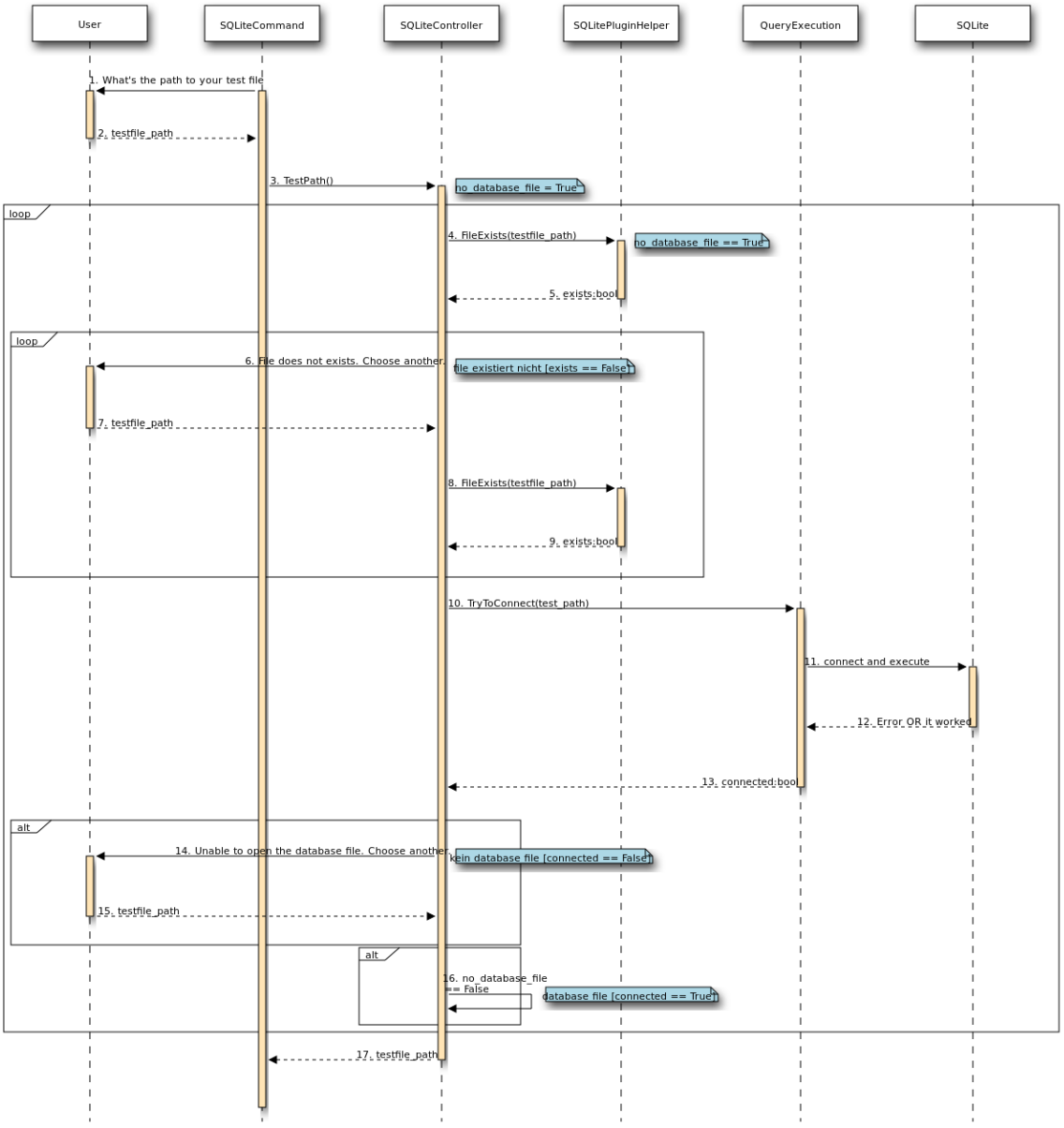


Abbildung 42: Sequenzdiagramm der Methode TestPath()

1. Das SQLiteCommand Modul fragt den User nach dem Pfad zur Datenbank
2. Der User gibt den Pfad zur Datenbank an.
3. Das SQLiteCommand Modul ruft die Methode TestPath() vom SQLiteController auf.

Der Teil 4 – 16 wird solange ausgeführt, wie die Variable no_database_file den Boolean True beinhaltet, somit es kein Datenbankfile ist, sowie beim ersten Durchlauf, da als Initialwert True gesetzt wird.

4. Der SQLiteController ruft die Methode FileExists mit dem Pfad als Argument beim SQLitePluginHelper auf.
5. Der SQLitePluginHelper prüft ob ein File an diesem Ort existiert und retourniert denn Boolean True falls ein File gefunden wurde.

Der Teil 6 – 9 wird solange ausgeführt wie die Methode FileExists False retourniert und somit kein File am angegebenen Pfad existiert.

6. Der SQLiteController teilt dem User mit, dass an jenem Pfad kein File existiert und fragt nach einer erneuten Eingabe.
7. Der User gibt einen neuen File Pfad ein.
8. Siehe 4.
9. Siehe 5.

Falls das File existiert wird hier fortgefahren.

10. Der SQLiteController ruft die Methode TryToConnect mit dem Pfad zur Datenbank als Parameter beim QueryExecution Modul auf.
11. Das QueryExecution Modul versucht sich mit SQLite bei der Datenbank zu verbinden und ein TestQuery laufen zu lassen, um sicher zu gehen, dass das File eine SQLite Datenbank ist.
12. SQLite retourniert einen Error, falls das Ausführen des Queries nicht funktionierte.
13. Das QueryExecution Modul retourniert True falls erfolgreich eine Verbindung aufgebaut werden konnte und es sich um ein SQLite Datenbank File handelt und sonst False.

Falls False zurückgegeben wurde und es sich nicht um ein SQLite Datenbank File handelt ,wird 14 – 15 ausgeführt.

14. Der SQLiteController teilt dem User mit dass es keine Verbindung mit dem File herstellen konnte, und verlangt eine neue Eingabe.
15. Der User gibt einen neuen Pfad zu seinem Test File ein. Da die Variable no_database_file immer noch den Wert False beinhaltet, wird wieder bei Punkt 4 weitergefahren.

Falls True zurückgegeben wurde und die Verbindung mit der SQLite Datenbank gelungen ist wird hier fortgefahren.

16. Der boolen no_database_file wird auf False gesetzt und somit der Loop beendet.
17. Der SQLiteController gibt dem SQLiteCommand Modul den Pfad zurück.

8.5.6 SQLQuery

Dieses Sequenzdiagramm beschreibt den Ablauf für die Eingabe der Queries. Dieser Teil ist der Komplizierteste, weshalb die folgenden Diagramme vereinfacht wurden. Für die wichtigsten Methodenaufrufe wurden separate Diagramme erstellt. Auch weicht der Code ein wenig ab, da er schöner geschrieben wurde als das Sequenzdiagramm.

Dieses Sequenzdiagramm zeigt den Ablauf der SQLQuery Methode.
Die Methode GetSQLQueryModelWithUserInput wurde hier abstrahiert und wird anschliessend unter 8.5.6.1 erklärt.

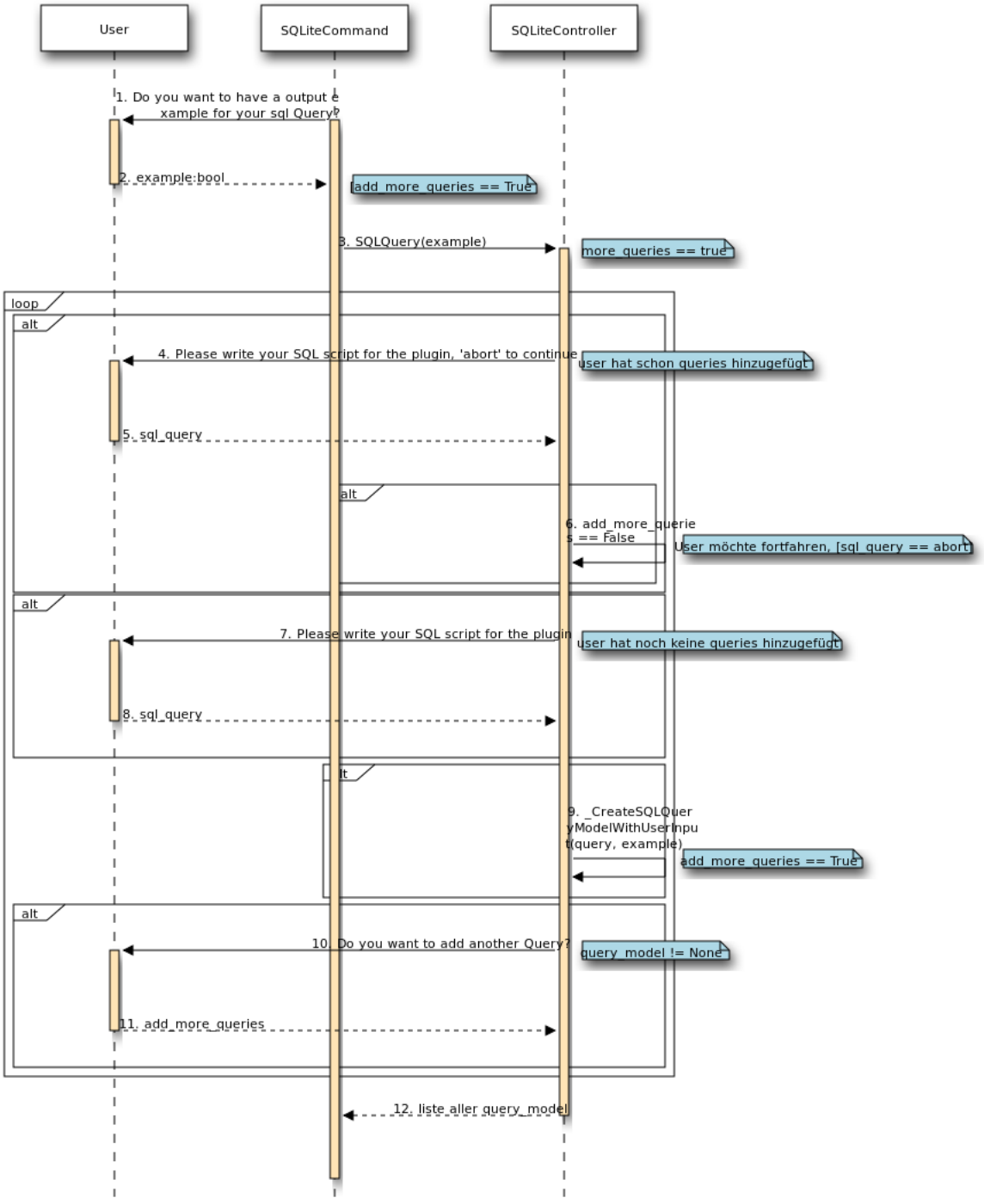


Abbildung 43: Sequenzdiagramm der Methode SQLQuery()

1. Das SQLiteCommand Modul fragt den User ob er einige Zeilen des ResultSets für die Queries angezeigt haben möchte.
2. Der User gibt Y oder N ein für die Eingabe (falls nicht die Abfrage solange wiederholt bis er valide Eingabe macht)
3. Das SQLCommandModul ruft vom SQLiteController die Methode SQLQuery auf.

Dieser Teil 4-11 wird solange ausgeführt wie der User Queries hinzufügen möchte.

Fall der User schon Queries hinzugefügt hatte:

4. Der User wird für neue Queries gefragt, kann jedoch auch den Teil SQLQuery verlassen indem er abort eingibt.[85]
5. Der User gibt das SQLQuery oder abort zurück.

Falls der User abort eingegeben hatte.

6. Die Variable add_more_queries wird auf False gesetzt und der Ablauf für SQLQuery wird beendet. (Man kommt zu Punkt 12)

Falls der User noch keine Queries hinzugefügt hatte:

7. Der User wird aufgefordert ein Query einzugeben.
8. Der User gibt ein Query ein.

Nachdem der User das Query eingegeben hatte und add_more_queries auf True gesetzt ist.:

9. Die Methode CreateSQLQueryModelWithUserInput mit dem Query als Parameter wird ausgeführt. Siehe 8.5.6.1

Falls die Methode CreateSQLQueryModelWithUserInput nicht None retourniert. Ansonsten war das Query nicht valide.

10. Der User wird gefragt, ob er ein weiteres Query hinzufügen möchte.
11. Der User gibt ein, ob er ein weiteres Query hinzufügen möchte.

Falls der User keine Queries mehr hinzufügen möchte und add_more_queries somit False ist:

12. Der SQLiteController gibt dem SQLiteCommand alle SQL Queries mit zusätzlichen Daten zurück.

[85] Dies war eine Designentscheidung, Falls der User sich vertippt und trotzdem kein Query hinzufügen möchte, kann er somit einfach aussteigen und muss das Programm nicht neustarten.

8.5.6.1 CreateSQLQueryModelWithUserInput

Diese Methode validiert das Query und sammelt mehr Daten zum Query.
Dieses Sequenzdiagramm wurde in 3 Teile aufgeteilt, damit es übersichtlicher ist.

Teil 1

Dieses Sequenzdiagramm zeigt den ersten Teil an.
Da seqdiag nicht detaillierte Aufrufe auf das eigene Modul darstellt, gibt es beim Diagramm die QueryExecutionMethode, welche eine Methode in QueryExecution Modul darstellt.

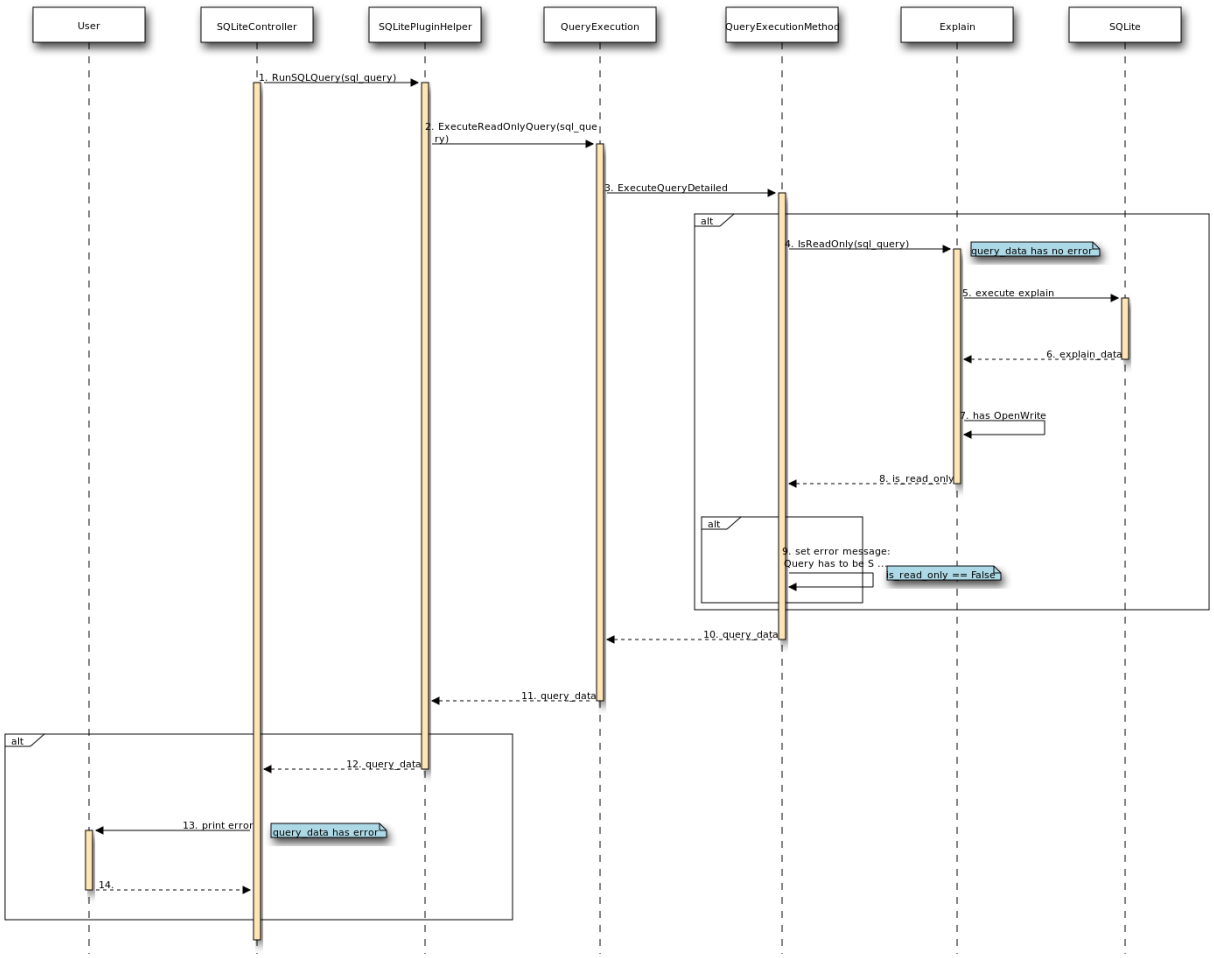


Abbildung 44: Sequenzdiagramm für die Methode CreateSQLQueryModelWithUserInput() Teil 1

1. Der SQLiteController ruft die Methode RunSQLQuery beim SQLitePluginHelper auf um das Query auszuführen und gleichzeitig zu validieren.
2. Der SQLitePluginHelper ruft die Methode ExecuteReadOnlyQuery beim QueryExecution Modul auf, da das Query nur lesende Zugriffe machen darf.
3. Das QueryExecutionModul ruft seine Methode ExecuteQueryDetailed auf, um zusätzliche Informationen zu dem Query zu sammeln. Diese Methode ruft selbst noch eine weitere Hilfsmethode auf, welche nach dem execute ein Rollback ausführt.[86]

Falls das Query keinen Fehler hat:

4. Das QueryExecution Modul ruft die Methode IsReadOnly beim Explain Modul auf, um herauszufinden ob es sich um ein read-only Query handelt.[87]
5. Das Explain Modul führt mit SQLite auf der Datenbank ein Explain Statement aus.
6. Das SQLite Modul für den Query aus und retourniert das Resultat.
7. Das Explain Modul testet, dass im Explain Resultat nie ein OpenWrite Keyword existiert. Somit kann sichergegangen werden, dass es sich um ein Select Statement handelt.
8. Das Explain Modul retourniert True falls es sich um ein Select Statement handelt, und andererseits False.

Falls das Query einen Fehler hat:

9. Im Query Datenmodel wird der Fehler sowie eine Fehlermeldung vermerkt.
10. Die Methode ExecuteQuery retourniert das Query Datenmodel.
11. Das QueryExecutionModul retourniert das Query Datenmodel dem SQLitePluginHelper

Falls in den Query Daten ein Fehler vermerkt wurde.

12. Das SQLitePluginHelper Modul retourniert das Query Datenmodel dem SQLiteController (falls ein Fehler vermerkt wurde).
13. Der SQLiteController zeigt dem User eine Fehlermeldung an.

[86] Dies war eine Designentscheidung. Da der User einen Fehler machen könnte, und nicht gewollt ist, dass er versehentlich seine Datenbank löscht, wird sicherheitshalber immer ein Rollback gemacht.

[87] Wenn Queries hinzugefügt werden, welche Daten modifizieren, und dies in keinem Review erkannt wurde, würden produktive Daten verändert werden, was dramatisch wäre.

Teil 2

Dieses Sequenzdiagramm zeigt den zweiten Teil an.

Dieses Modul zeigt einen Teil des Ablaufes an, welcher durchgeführt wird, falls das Query fehlerfrei ist.

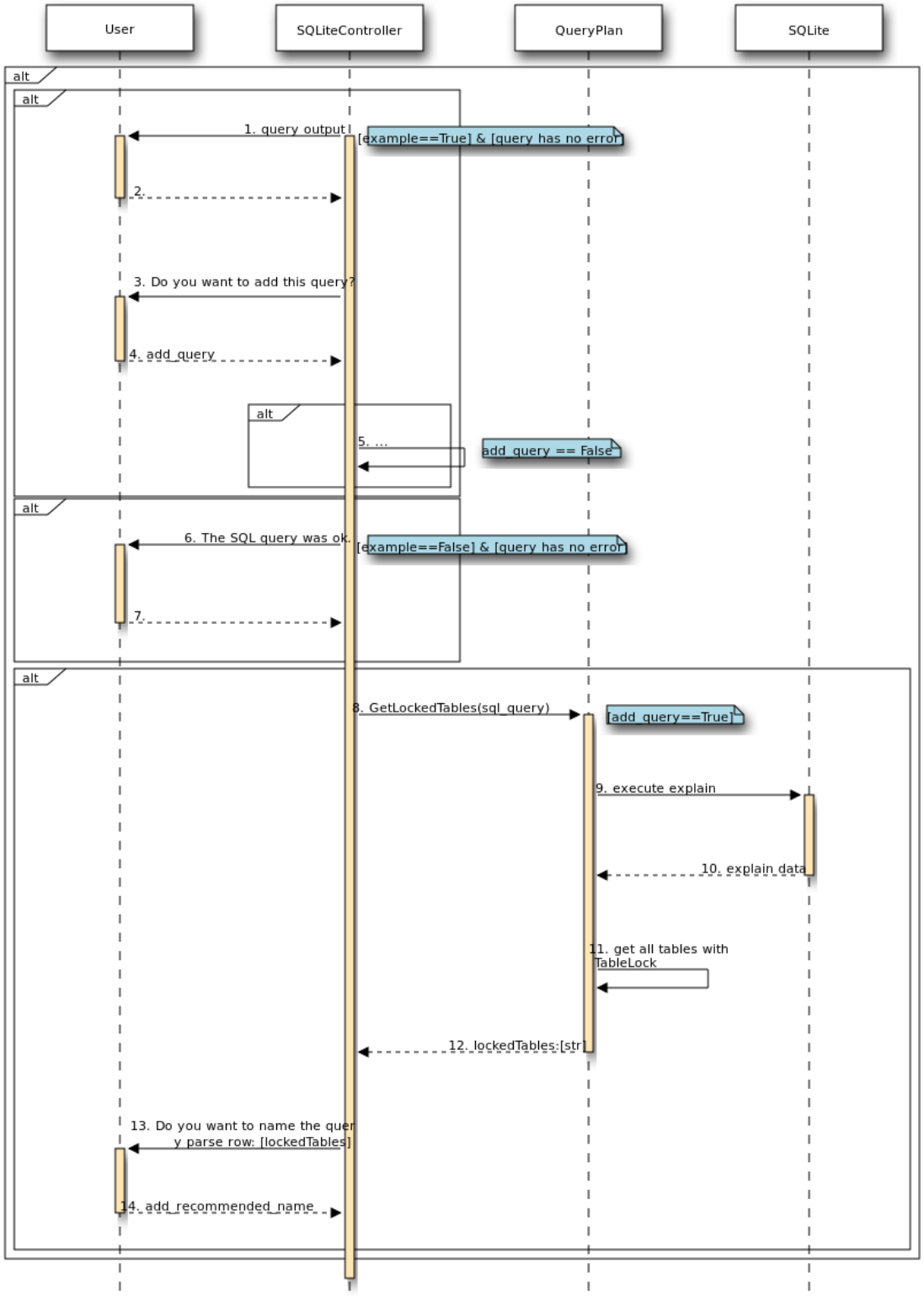


Abbildung 45: Sequenzdiagramm für die Methode CreateSQLQueryModelWithUserInput() Teil 2

Falls der User ein Beispiel für das Query möchte.

1. Der SQLiteControler zeigt dem User die ersten drei Zeilen des ResultSets für das Query.
2. *Es wird keine Eingabe gefordert.*
3. Der SQLiteController fragt den User ob er das Query hinzufügen möchte.
4. Der User gibt ein ob er das Query hinzufügen möchte.

Falls der User das Query nicht hinzufügen möchte:

5. Der SQLiteController beendet den Vorgang für CreateSQLQueryModelWithUserInput.

Falls der User kein Beispiel für das Query möchte.

6. Der SQLiteController gibt dem User Bescheid, dass der Query in Ordnung war. Und da der User somit nicht mehr Informationen bekam, wird die add_query Variable auf True gestellt.
7. *Es wird keine Eingabe gefordert.*

Falls die add_query Variable True ist. Somit der User das Query hinzufügen möchte.

8. Der SQLiteController ruft die Methode GetLockedTables vom QueryPlan Modul auf.
9. Der QueryPlan führt das explain Query mit SQLite auf der Datenbank auf.
10. Das SQLite Modul gibt die Daten des explain Queries zurück.
11. Das QueryPlan Modul sucht alle Datenzeile mit der QueryLock Id, um die Tabellen abzufragen auf welchen das Query zugegriffen hatte.
12. Der QueryPlan gibt eine Liste mit allen Namen der Tabellen zurück, auf welche das Query zugegriffen hatte.
13. Der SQLiteController erstellt aus der Liste der Tabellennamen einen Namen für das Query und gibt dem User die Möglichkeit den Namen zu ändern.
14. Der User gibt ein, ob er den vorgeschlagenen Namen annimmt.[88]

[88] Es wird nicht geprüft ob der Name schon existiert, da es möglich sein muss, den gleichen Namen zu benutzen. Dies ist eine Vorgabe.

Teil 3

Dieses Sequenzdiagramm zeigt den dritten Teil an.

Da seqdiag nicht detaillierte Aufrufe auf das eigene Modul darstellt, gibt es beim Diagramm die SQLiteControllerMethod, welche eine Methode im SQLiteController Modul darstellt.

Dieses Modul zeigt einen Teil des Ablaufes an, welcher durchgeführt wird, falls das Query fehlerfrei war.

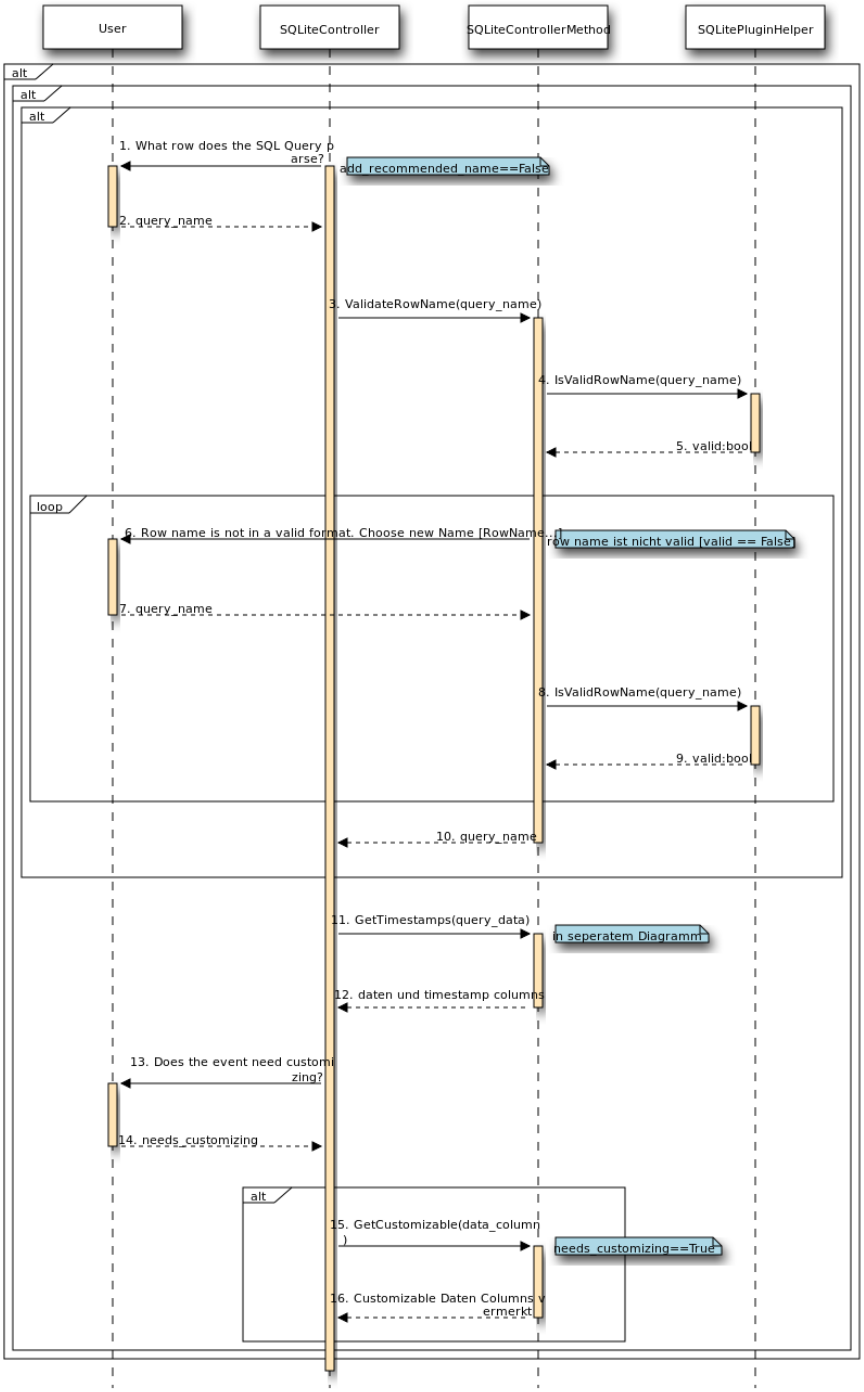


Abbildung 46: Sequenzdiagramm für die Methode CreateSQLQueryModelWithUserInput() Teil 3

Der User möchte einen eigenen Namen für das Query angeben:

1. Der SQLiteController fragt den User nach dem Namen des SQL Queries, welches die zu parsende Zeile beschreibt.
2. Der User gibt einen Namen ein.
3. Der SQLiteController ruft die Methode ValidateRowNamen bei sich auf, auf um den Namen zu validieren.
4. Der SQLiteController ruft die Methode IsValidRowNamen beim SQLitePluginHelper auf.
5. Der SQLitePluginHelper prüft den Namen mithilfe eines regex auf das korrekte Format. Er retourniert True falls der Namen valid ist und ansonsten False.

Solange der Row Name ungültig ist:

6. Der SQLiteController teilt dem User mit das der Name nicht in einem gültigen Format ist und fordert ihn auf einen neuen Namen einzugeben.
7. Der User gibt einen neuen Namen ein.
8. Siehe 4.
9. Siehe 5.
10. Die Methode ValidateRowName gibt dem SQLiteController den validen Namen zurück

Der User nimmt den vorgeschlagenen Namen an oder es wurde ein valider Name ausgesucht.

11. Der SQLiteController ruft die Methode GetTimstamps auf.[89]
12. Die Methode GetTimestamps retourniert Daten sowie Timestamps Zeilen.
13. Der SQLiteController fragt den User ob Zeilen im Query Anpassungen brauchen.
14. Der User macht eine Eingabe.
15. Der SQLiteController ruft die Methode GetCustomizable auf, welche den User im gleichen Style wie bei GetTimestamps auf die Spaltennamen abfragt, welche Anpassungen brauchen.
16. Die Methode GetCustomizable retourniert die Daten Zeilen mit Bemerkung welche Anpassungen brauchen.

[89] Zu dieser Methode existiert gerade nachfolgend ein Sequenzdiagramm.

GetTimestamps

Dieses Sequenzdiagramm beschreibt den Ablauf für die Eingabe der Timestamps.

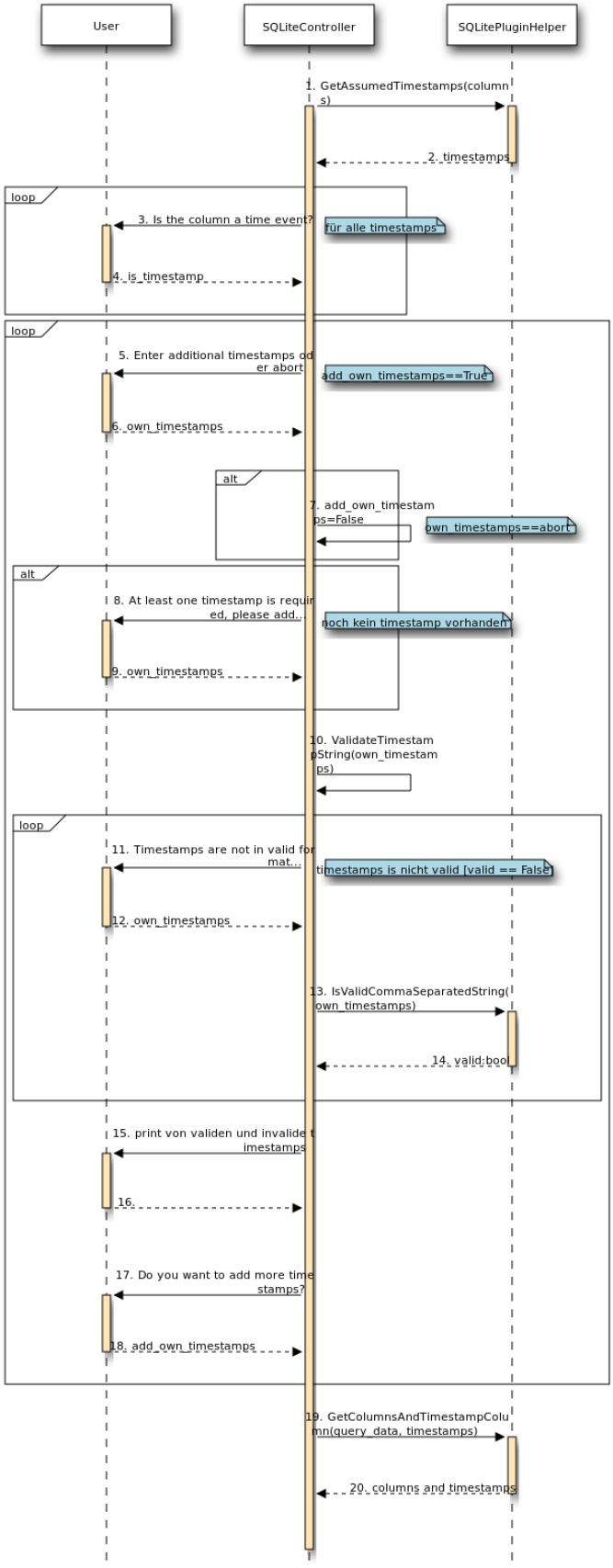


Abbildung 47: Sequenzdiagramm für die Methode Timestamps()

1. Der SQLiteController ruft das SQLitePluginHelper Modul auf, um Spaltenvorschläge zu bekommen.[90]
2. Der SQLitePluginHelper gibt alle Spaltennamen zurück welche er als Timestamp vorschlägt.

Punkt 2-3 wird für jeden Timestamp welcher vorgeschlagen wurde ausgeführt

3. Der SQLiteController fragt den User ob der vorgeschlagene Timestamp ein Timestamp ist.
4. Der User gibt ein ob es ein Timestamp ist.

Punkt 5 – 18 wird solange ausgeführt wie die variable add_own_timestamps True ist sowie bei der ersten Durchführung und der User somit weitere timestamps hinzufügen möchte.

5. Der SQLiteController fragt den User ob er weitere timestamps hinzufügen möchte oder diesen Teil mit abort abbrechen möchte.
6. Der User gibt ein ob er weitere timestamps möchte.
7. Falls der User abort eingibt, wird die Variable add_own_timestamps auf False gestellt.

Falls der User Abort Eingabe und noch kein Timestamp vorhanden ist:

8. Der SQLiteController teilt dem User mit, dass er mindestens eine Spalte als Timestamp identifiezieren muss.
9. Der User macht eine Eingabe.
10. Der SQliteController ruft die Methode ValidateTimestampString auf, um den Timestamp mit einem regex Pattern zu validieren. Siehe 13 + 14

Punkt 11 – 14 wird solange ausgeführt wie die Eingabe des Users nicht valide ist.

11. Der SQLiteController teilt dem User mit das seine Eingabe nicht valide ist.
12. Der User gibt neue Timestamps ein.
13. Der SQLiteController ruft die Methode IsValidCommaSeperatedString beim SQLitePluginHelper auf um die Eingabe zu validieren.
14. Falls die Eingabe valide ist retourniert der SQLitePluginHelper True, ansonsten False.

Die Eingabe ist valide.

15. Der SQLiteController zeigt dem User alle Timestamps an welche hinzugefügt wurden, und die jenigen, welche ungültig waren.
16. –
17. Der SQliteController fragt den User ob er noch mehr timestamps hinzufügen möchte.
18. Der User gibt ein ob er noch mehr timestamps hinzufügen möchte. Falls True wird wieder bei 5 gestartet, ansonsten wird dieser Teil der Verarbeitung verlassen.

Der User hat alle Timestamps eingegeben:

19. Der SQLiteController ruft die Methode GetColumnsAndTimestampColumns beim SQLitePluginHelper auf um die Informationen vom User zu verabeiten und in das Datenmodell zu übernehmen
20. Der SQLitePluginHelper retourniert die Daten sowie Timestamp Zeilen.

[90] Dies beinhaltet alle Spalten welche keine Timestamps sind und das Wort data oder time beinhalten.

8.5.7 Generate

Dies ist der letzte Teil des Programmes und wird aufgerufen sobald alle Daten vom User eingegeben und validiert wurden. Die Integrationstests testen nur diesen Teil. Deshalb sind die automatischen End-to-End Tests eine wichtige Vervollständigung.

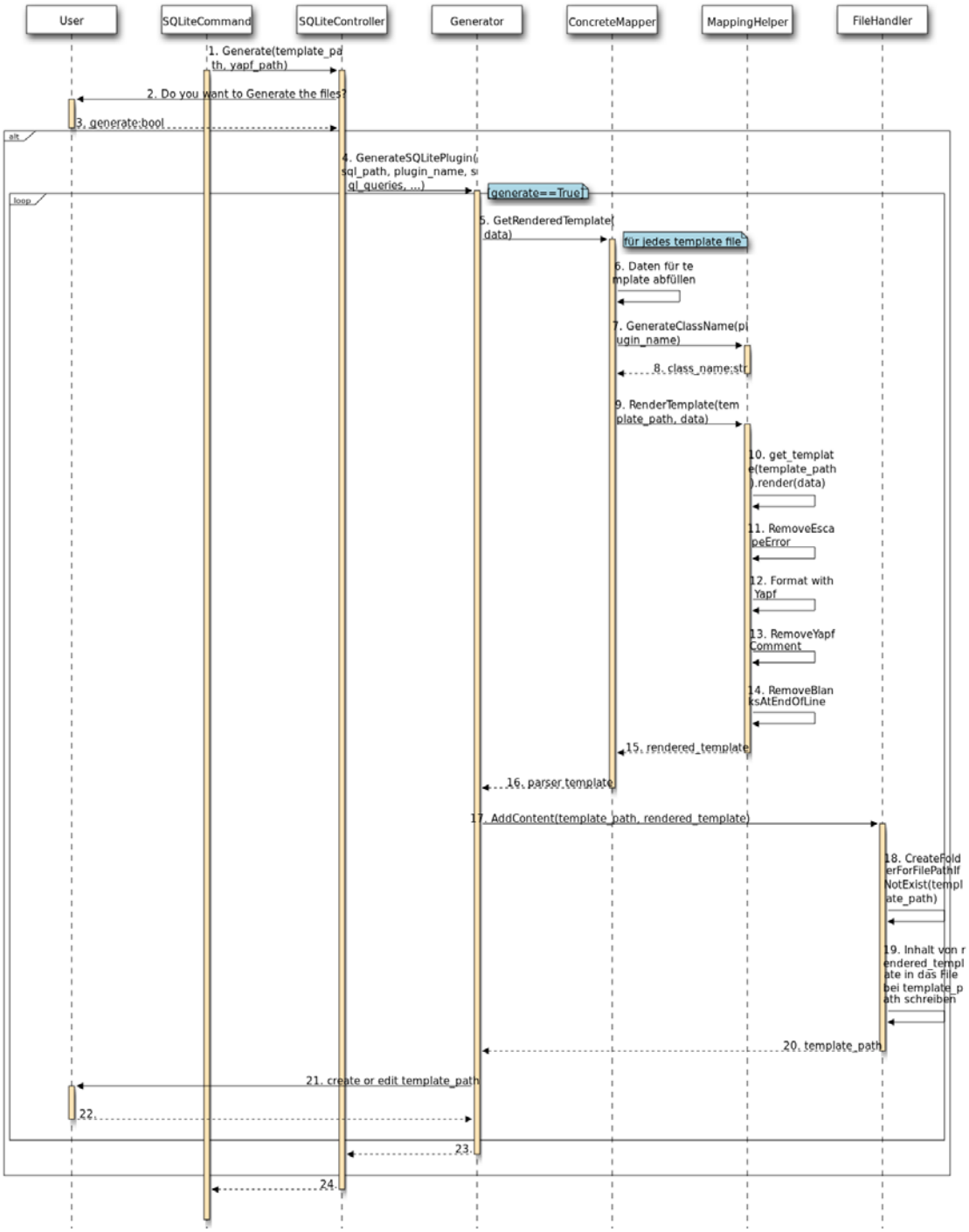


Abbildung 48: Sequenzdiagramm für die Methode Generate()

1. Das SQLiteCommand Modul ruft den SQliteController mit der Methode Generate sowie dem template und yapf Pfad auf.
2. Der SQLiteController fragt den User ob er die Files nun generieren möchte.
3. Der User macht eine Eingabe.

Der User möchte die Files generieren:

4. Der SQliteController ruft die Methode GenerateSQLitePlugin vom Generator auf und übergibt als Parameter alle Daten welche zuvor vom User Eingegeben wurde sowie berechnet.

Dies wird für jedes File welche generiert wird im gleichen Style ausgeführt.

5. Der Generator ruft die Methode GetRenderedTemplate vom Mapper für das zu erstellende File auf.
6. Der ConcreteMapper füllt die Daten für das Template in einem Dictionary ab.
7. De ConcreteMapper ruft die Methode GenerateClassName vom MappingHelper auf um den Klassennamen aus dem Pluginnamen zu generieren.
8. Der Mapping Helper gibt den Klassennamen zurück.
9. Der ConcreteMapper ruft die Methode RenderTemplate vom MappingHelper auf und gibt als Parameter den Pfad zum Templatefile mit, sowie dass Dictionary mit den Daten.
10. Der MappingHelper rendered das Template mit den Daten und bekommt in einem String den Inhalt des zu erstellenden Files zurück.
11. Der MappingHelper führt die Methode RemoveEscapeError auf um escape Fehler zu vermeiden.[91]
12. Der MappingHelper formatiert das File mit Hilfe von Yapf.[92]
13. Der MappingHelper führt die Methode RemoveYapfComment auf um die Kommentare für Yapf zu entfernen. [93]
14. Der Mapping Helper führt die Methode RemoveBlanksAtEndOfLine aus um leere Blanks am Ende von Zeilen zu entfernen.[94]
15. Der MappingHelper retourniert den Inhalt des zu erstellenden Files als String.
16. Der ConcreteMapper retourniert den Inhalt des zu erstellenden Files vom MappingHelper.
17. Der Generator führt die Methode AddContent auf dem FileHandler auf und gibt den String für das File sowie den Pfad wo das File danach liegen sollte als Parameter mit.
18. Der FileHandler ruft die Methode CreateFolderForPathIfNotExists auf, um ein File am entsprechenden Ort zu erstellen, falls es noch keines gibt.
19. Der FileHandler fügt den Inhalt des neuen Files vom MappingHelper dem File hinzu, welches neu erstellt wurde oder schon existiert.
20. Der FileHandler retourniert den Pfad zum editierten oder erstellten File.
21. Der Generator gibt dem User an welches File erstellt oder editiert wurde.
22. -

Der User möchte die Files nicht generieren: Der User sieht die Ausgabe: ‘aborted!’.

23. -
24. Das Programm wurde beendet.

[91] Siehe Abschnitt Zeilenumbruch unter: 5.12.3

[92] Mehr zu Yapf unter: 5.12

[93] Siehe Abschnitt Formattierung ausschalten unter: 5.12.1

[94] Siehe Abschnitt Formattierung ausschalten unter: 5.12.1

9 Testplan

Um die Tests automatisch laufen zu lassen, müssen diese Schritte auf Linux[95] durchgeführt werden:

1. Clonen des Repositories: git clone https://github.com/ClaudiaSaxer/PlasoScaffolder.git
2. Herunterladen der Requirements: pip install requirements.txt
3. Laufen lassen mit pycharm mit Python 3 oder direkt.

Manuelles Testen:

1. pip install plasoscaffolder

Empfehlung: zuvor virtual Environment erstellen:

1. virtualenv virtualenvplasoscaffolder
2. source virtualenvplasoscaffolder/bin/activate

Test 1 - Hilfe Funktion	
Funktionalität	Help Option von plasoscaffolder Testen
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - Plasoscaffolder lässt sich aufrufen
Abhängigkeiten	- Keine
Ablauf	1. Plasoscaffolder mit –help Command starten: *plasoscaffolder --help*
Erwartete Resultate	**Konsolen Output:** Usage: plasoscaffolder [OPTIONS] COMMAND [ARGS]... Options: --help Show this message and exit. Commands: Sqlite
	File Output: Es wurden keine Files generiert
Besonderheiten	Falls das Commando von wo anders gestartet wird kann Usage: plasoscaffolder … abweichen. Zum Beispiel zu Usage: main.py…
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_help_function.py

Tabelle 12: End-to-End-Test 1

[95] Erklärung wieso die Tests auf Linux ausgeführt werden müssen: 5.8.4

Test 2 - Hilfe Funktion von SQLite	
Funktionalität	Help Option von SQLite Testen
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - Plasoscaffolder lässt sich aufrufen
Abhängigkeiten	- Keine
Ablauf	1. Plasoscaffolder mit sqlite --help Command starten: *plasoscaffolder sqlite --help*
Erwartete Resultate	**Konsolen Output:** Usage: plasoscaffolder sqlite [OPTIONS] Options:' --path TEXT The path to plaso --name TEXT The plugin name --testfile TEXT The testfile path --sql / --no-sql The output example flag for the SQL Query for the plugin. --help Show this message and exit.
	File Output: Es wurden keine Files generiert
Besonderheiten	Falls das Commando von wo anders gestartet wird kann Usage: plasoscaffolder ... abweichen. Zum Beispiel zu Usage: main.py...
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_help_function_of_sqlite.py

Tabelle 13: End-to-End-Test 2

Test 3 - Einfaches Generieren	
Funktionalität	Testen eines Einfachen Ablaufes mit Generieren
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - [pfad]: Existierender pfad zu einem leeren Ordner - [pfad_file]: Pfad zum twitter_ios.db File - twitter_ios.db File ist in einem gültigen Zustand
Abhängigkeiten	- Keine
Ablauf	1. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite* 2. What's the path to the plaso project?: *[pfad]* 3. What's the name of the plugin?: *test* 4. What's the path to your test file?: [pfad_file]: 5. Do you want to have a output example for your SQL Query? [Y/n]: *n* 6. Please write your SQL script for the plugin: *select * from users* 7. The SQL query was ok. 8. Do you want to name the query parse row: Users ? [Y/n]: *Y* 9. Is the column a time event? updatedAt [Y/n]: *Y* 10. Is the column a time event? createdDate [Y/n]: *Y* 11. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: *abort* 12. Does the event Users need customizing? [y/N]: *N* 13. Do you want to add another Query? [Y/n]: *n* 14. Do you want to Generate the files [Y/n]: Y
Erwartete Resultate	**Konsolen Output:** create [pfad]/plaso/formatters/test.py create [pfad]/plaso/parsers/sqlite_plugins/test.py create [pfad]/tests/formatters/test.py create [pfad]/tests/parsers/sqlite_plugins/test.py copy [pfad]/test_data/test.db create [pfad]/plaso/parsers/sqlite_plugins/__init__.py create [pfad]/plaso/formatters/__init__.py
	File Output: Inhalt der generierten Files stimmen überein mit Files unter: PlasoScaffolder/src/tests/end-to-end-test/ExpectedEasyGenerationFiles
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_easy_generation.py

Tabelle 14: End-to-End-Test 3

Test 4 - Einfaches Generieren mit Abbruch	
Funktionalität	**Testen eines Einfachen Ablaufes mit Generieren**
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - [pfad]: Existierender pfad zu einem leeren Ordner - [pfad_file]: Pfad zum twitter_ios.db File - twitter_ios.db File ist in einem gültigen Zustand
Abhängigkeiten	- Test 3
Ablauf	1. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite* 2. What's the path to the plaso project?: *[pfad]* 3. What's the name of the plugin?: *test* 4. What's the path to your test file?: *[pfad_file]* 5. Do you want to have a output example for your SQL Query? [Y/n]: *n* 6. Please write your SQL script for the plugin: *select * from users* 7. The SQL query was ok. 8. Do you want to name the query parse row: Users ? [Y/n]: *Y* 9. Is the column a time event? updatedAt [Y/n]: *Y* 10. Is the column a time event? createdDate [Y/n]: *Y* 11. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: *abort* 12. Does the event Users need customizing? [y/N]: *N* 13. Do you want to add another Query? [Y/n]: *n* 14. Do you want to Generate the files [Y/n]: n
Erwartete Resultate	**Konsolen Output:** Aborted!
	File Output: Es wurden keine Files generiert
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_easy_generation_with_abort.py

Tabelle 15: End-to-End-Test 4

Test 5: Falsche Pfad Eingaben	
Funktionalität	Testen von Eingabe eines nicht existierenden Pfades.
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - [pfad]: Existierender pfad zu einem leeren Ordner - [pfad_falsch]: Pfad welcher nicht existiert - [pfad _file]: Pfad zum twitter_ios.db File - twitter_ios.db File ist in einem gültigen Zustand
Abhängigkeiten	- Test 3
Ablauf	1. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite* 2. What's the path to the plaso project?: *[pfad_falsch]* 3. Folder does not exists. Enter correct one: *[pfad _falsch]* 4. Folder does not exists. Enter correct one: *[pfad]* 5. What's the name of the plugin?: *test* 6. What's the path to your test file?: *[pfad _file]* 7. Do you want to have a output example for your SQL Query? [Y/n]: *n* *8.* Please write your SQL script for the plugin: *select * from users* 9. The SQL query was ok. 10. Do you want to name the query parse row: Users ? [Y/n]: *Y* 11. Is the column a time event? updatedAt [Y/n]: *Y* 12. Is the column a time event? createdDate [Y/n]: *Y* 13. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: *abort* 14. Does the event Users need customizing? [y/N]: *N* 15. Do you want to add another Query? [Y/n]: *n* 16. Do you want to Generate the files [Y/n]: Y
Erwartete Resultate	**Konsolen Output:** create [pfad]/plaso/formatters/test.py create [pfad]/plaso/parsers/sqlite_plugins/test.py create [pfad]/tests/formatters/test.py create [pfad]/tests/parsers/sqlite_plugins/test.py copy [pfad]/test_data/test.db create [pfad]/plaso/parsers/sqlite_plugins/__init__.py create [pfad]/plaso/formatters/__init__.py
	File Output: Inhalt der generierten Files stimmen überein mit Files unter: PlasoScaffolder/src/tests/end-to-end-test/ExpectedEasyGenerationFiles
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_wrong_path.py

Tabelle 16: End-to-End-Test 5

Test 6 - Falsche Plugin Name Eingabe	
Funktionalität	Testen von Eingaben von nicht gültigen Pluginnamen sowie Pluginnamen mit einem Unterstrich.
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - [pfad]: Existierender pfad zu einem leeren Ordner - [pfad_file]: Pfad zum twitter_ios.db File - twitter_ios.db File ist in einem gültigen Zustand
Abhängigkeiten	- Test 3
Ablauf	1. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite* 2. What's the path to the plaso project?: *[pfad]* 3. What's the name of the plugin?: *test plugin* 4. Plugin is not in a valid format. Choose new Name [plugin_name_...]: *TestPlugin* 5. Plugin is not in a valid format. Choose new Name [plugin_name_...]: *test123_plugin* 6. Plugin is not in a valid format. Choose new Name [plugin_name_...]: *test_plugin* 7. What's the path to your test file?: *[pfad_file]* 8. Do you want to have a output example for your SQL Query? [Y/n]: *n* 9. Please write your SQL script for the plugin: *select * from users* 10. The SQL query was ok. 11. Do you want to name the query parse row: Users ? [Y/n]: *Y* 12. Is the column a time event? updatedAt [Y/n]: *Y* 13. Is the column a time event? createdDate [Y/n]: *Y* 14. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: *abort* 15. Does the event Users need customizing? [y/N]: *N* 16. Do you want to add another Query? [Y/n]: *n* 17. Do you want to Generate the files [Y/n]: Y
Erwartete Resultate	**Konsolen Output:** create [pfad]/plaso/formatters/*test_plugin*.py create [pfad]/plaso/parsers/sqlite_plugins/*test_plugin*.py create [pfad]/tests/formatters/ *test_plugin*.py create [pfad]/tests/parsers/sqlite_plugins/*test_plugin*.py copy [pfad]/test_data/*test_plugin*.db create [pfad]/plaso/parsers/sqlite_plugins/__init__.py create [pfad]/plaso/formatters/__init__.py
	File Output: Inhalt der generierten Files stimmen überein mit Files unter: PlasoScaffolder/src/tests/end-to-end-test/ExpectedEasyGenerationLongNameFiles
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_wrong_plugin_name.py

Tabelle 17: End-to-End-Test 6

Test 7 - Falsches Test File	
Funktionalität	Testen von Eingabe eines nicht existierenden Testfilepfad sowie einem nicht validen Test File.
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - [pfad]: Existierender pfad zu einem leeren Ordner - [pfad_nicht_existierend]: Ein nicht existierender Pfad - [pfad_file]: Pfad zum twitter_ios.db File - [pfad_file_error]: Pfad zum twitter_ios_error.db File - twitter_ios.db File ist in einem gültigen Zustand. - twitter_ios_error.db File ist nicht in einem gültigen Zustand.
Abhängigkeiten	- Test 3
Ablauf	1. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite* 2. What's the path to the plaso project?: *[pfad]* 3. What's the name of the plugin?: *test* 4. What's the path to your test file?: [pfad_nicht_existierend] 5. File does not exists. Choose another.: [pfad_file_error]: 6. Unable to open the database file. Choose another.: [pfad_file] 7. Do you want to have a output example for your SQL Query? [Y/n]: *n* *8.* Please write your SQL script for the plugin: *select * from users* 9. The SQL query was ok. 10. Do you want to name the query parse row: Users ? [Y/n]: *Y* 11. Is the column a time event? updatedAt [Y/n]: *Y* 12. Is the column a time event? createdDate [Y/n]: *Y* 13. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: *abort* 14. Does the event Users need customizing? [y/N]: *N* 15. Do you want to add another Query? [Y/n]: *n* 16. Do you want to Generate the files [Y/n]: Y
Erwartete Resultate	**Konsolen Output:** create [pfad]/plaso/formatters/test.py create [pfad]/plaso/parsers/sqlite_plugins/test.py create [pfad]/tests/formatters/test.py create [pfad]/tests/parsers/sqlite_plugins/test.py copy [pfad]/test_data/test.db create [pfad]/plaso/parsers/sqlite_plugins/__init__.py create [pfad]/plaso/formatters/__init__.py
	File Output: Inhalt der generierten Files stimmen überein mit Files unter: PlasoScaffolder/src/tests/end-to-end-test/ExpectedEasyGenerationFiles
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_wrong_test_file.py

Tabelle 18: End-to-End-Test 7

Test 8 – Mit Output Beispiel	
Funktionalität	Testen eines einfachen Ablaufes mit Generieren sowie einem Output Beispiel
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - [pfad]: Existierender pfad zu einem leeren Ordner - [pfad_file]: Pfad zum twitter_ios.db File - twitter_ios.db File ist in einem gültigen Zustand.
Abhängigkeiten	- Test 3
Ablauf	1. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite* 2. What's the path to the plaso project?: *[pfad]* 3. What's the name of the plugin?: *test* 4. What's the path to your test file?: [pfad_file] 5. Do you want to have a output example for your SQL Query? [Y/n]: *Y* 6. Please write your SQL script for the plugin: *select id, name from users* 7. *Your query output could look like this.* *['id', 'name']* *(5402612, 'BBC Breaking News')* *(13334762, 'GitHub')* *(14388264, 'Tom Pohl')* 8. Do you want to add this query? [Y/n]: 9. Do you want to name the query parse row: Users ? [Y/n]: *Y* 10. Is the column a time event? updatedAt [Y/n]: *Y* 11. Is the column a time event? createdDate [Y/n]: *Y* 12. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: *abort* 13. Does the event Users need customizing? [y/N]: *N* 14. Do you want to add another Query? [Y/n]: *n* 15. Do you want to Generate the files [Y/n]: Y
Erwartete Resultate	**Konsolen Output:** create [pfad]/plaso/formatters/test.py create [pfad]/plaso/parsers/sqlite_plugins/test.py create [pfad]/tests/formatters/test.py create [pfad]/tests/parsers/sqlite_plugins/test.py copy [pfad]/test_data/test.db create [pfad]/plaso/parsers/sqlite_plugins/__init__.py create [pfad]/plaso/formatters/__init__.py
	File Output: Inhalt der generierten Files stimmen überein mit Files unter: PlasoScaffolder/src/tests/end-to-end-test/ExpectedEasyGenerationWithOutputFiles
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_easy_generation_with_abort.py

Tabelle 19: End-to-End-Test 8

Test 9 - Mit Fehlerhaften SQL Eingaben	
Funktionalität	Testen mit verschiedenen fehlerhaften SQL Eingaben. - Select von einer nicht gefundenen Tabelle - 2 Selects nacheinander - Drop Table Statement
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - [pfad]: Existierender pfad zu einem leeren Ordner - [pfad_file]: Pfad zum twitter_ios.db File - twitter_ios.db File ist in einem gültigen Zustand.
Abhängigkeiten	- Test 3
Ablauf	1. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite* 2. What's the path to the plaso project?: *[pfad]* 3. What's the name of the plugin?: *test* 4. What's the path to your test file?: [pfad_file] 5. Do you want to have a output example for your SQL Query? [Y/n]: n *6.* Please write your SQL script for the plugin: *select * from user* 7. Error: no such table: user 8. Please write your SQL script for the plugin: *select * from users; select * from users* 9. Warning: You can only execute one statement at a time. 10. Please write your SQL script for the plugin: *drop table users* 11. Query has to be a single SELECT query. 12. Please write your SQL script for the plugin: *select * from users* 13. The SQL query was ok. 14. Do you want to name the query parse row: Users ? [Y/n]: *Y* 15. Is the column a time event? updatedAt [Y/n]: *Y* 16. Is the column a time event? createdDate [Y/n]: *Y* 17. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: *abort* 18. Does the event Users need customizing? [y/N]: *N* 19. Do you want to add another Query? [Y/n]: *y* 20. Please write your SQL script for the plugin ['abort' to continue]: *abort* 21. Do you want to Generate the files [Y/n]: Y
Erwartete Resultate	**Konsolen Output:** create [pfad]/plaso/formatters/test.py create [pfad]/plaso/parsers/sqlite_plugins/test.py create [pfad]/tests/formatters/test.py create [pfad]/tests/parsers/sqlite_plugins/test.py copy [pfad]/test_data/test.db create [pfad]/plaso/parsers/sqlite_plugins/__init__.py create [pfad]/plaso/formatters/__init__.py
	File Output: Inhalt der generierten Files stimmen überein mit Files unter: PlasoScaffolder/src/tests/end-to-end-test/ExpectedEasyGeneration
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_wrong_sql_query_input.py

Tabelle 20: End-to-End-Test 9

Test 10 - Mit mehreren Selects	
Funktionalität	Testen mit mehreren Selects
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - [pfad]: Existierender pfad zu einem leeren Ordner - [pfad_file]: Pfad zum twitter_ios.db File - twitter_ios.db File ist in einem gültigen Zustand.
Abhängigkeiten	- Test 3
Ablauf	1. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite* 2. What's the path to the plaso project?: *[pfad]* 3. What's the name of the plugin?: *test* 4. What's the path to your test file?: [pfad_file] 5. Do you want to have a output example for your SQL Query? [Y/n]: n 6. Please write your SQL script for the plugin: *select * from users* 7. The SQL query was ok. 8. Do you want to name the query parse row: Users ? [Y/n]: *Y* 9. Is the column a time event? updatedAt [Y/n]: *Y* 10. Is the column a time event? createdDate [Y/n]: *Y* 11. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: *abort* 12. Does the event Users need customizing? [y/N]: *N* 13. Do you want to add another Query? [Y/n]: *Y* 14. Please write your SQL script for the plugin: *select * from statuses* 15. The SQL query was ok. 16. Do you want to name the query parse row: Statuses ? [Y/n]: *Y* 17. Is the column a time event? date [Y/n]: *Y* 18. Is the column a time event? updatedAt [Y/n]: *Y* 19. Is the column a time event? includeInProfileTimeline [Y/n]: *N* 20. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: *abort* 21. Does the event Users need customizing? [y/N]: *N* 22. Do you want to add another Query? [Y/n]: *Y* 23. Please write your SQL script for the plugin ['abort' to continue]: *abort* 24. Do you want to Generate the files [Y/n]: Y
Erwartete Resultate	**Konsolen Output:** create [pfad]/plaso/formatters/test.py create [pfad]/plaso/parsers/sqlite_plugins/test.py create [pfad]/tests/formatters/test.py create [pfad]/tests/parsers/sqlite_plugins/test.py copy [pfad]/test_data/test.db create [pfad]/plaso/parsers/sqlite_plugins/__init__.py create [pfad]/plaso/formatters/__init__.py
	File Output: Inhalt der generierten Files stimmen überein mit Files unter: PlasoScaffolder/src/tests/end-to-end-test/ExpectedEasyGenerationMultipleSelects
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_easy_generation_with_multiple_selects.py

Tabelle 21: End-to-End-Test 10

Test 11 - Mit eigenem Row Name	
Funktionalität	Testen des Eingebens eines eigenen Row Namens
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - [pfad]: Existierender pfad zu einem leeren Ordner - [pfad_file]: Pfad zum twitter_ios.db File - twitter_ios.db File ist in einem gültigen Zustand
Abhängigkeiten	- Test 3
Ablauf	1. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite* 2. What's the path to the plaso project?: *[pfad]* 3. What's the name of the plugin?: *test* 4. What's the path to your test file?: [pfad_file] 5. Do you want to have a output example for your SQL Query? [Y/n]: n 6. Please write your SQL script for the plugin: *select * from users* 7. The SQL query was ok. 8. Do you want to name the query parse row: Users ? [Y/n]: *n* 9. What row does the SQL Query parse?: *The User* 10. Row name is not in a valid format. Choose new Name [RowName...]: theuser 11. Row name is not in a valid format. Choose new Name [RowName...]: TheUser123 12. Row name is not in a valid format. Choose new Name [RowName...]: TheUser 13. Is the column a time event? updatedAt [Y/n]: *Y* 14. Is the column a time event? createdDate [Y/n]: *Y* 15. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: *abort* 16. Does the event Users need customizing? [y/N]: *N* 17. Do you want to add another Query? [Y/n]: *n* 18. Do you want to Generate the files [Y/n]: Y
Erwartete Resultate	**Konsolen Output:** create [pfad]/plaso/formatters/test.py create [pfad]/plaso/parsers/sqlite_plugins/test.py create [pfad]/tests/formatters/test.py create [pfad]/tests/parsers/sqlite_plugins/test.py copy [pfad]/test_data/test.db create [pfad]/plaso/parsers/sqlite_plugins/__init__.py create [pfad]/plaso/formatters/__init__.py
	File Output: Inhalt der generierten Files stimmen überein mit Files unter: PlasoScaffolder/src/tests/end-to-end-test/ExpectedEasyGenerationOwnColumnNameFiles
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_easy_generation_with_own_row_name.py

Tabelle 22: End-to-End-Test 11

Test 12 - Mit eigenen Time Events	
Funktionalität	Testen der Eingabe eigener Time Events sowie Ablehnen von vorgeschlagenen
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - [pfad]: Existierender pfad zu einem leeren Ordner - [pfad_file]: Pfad zum twitter_ios.db File - twitter_ios.db File ist in einem gültigen Zustand
Abhängigkeiten	- Test 3
Ablauf	1. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite* 2. What's the path to the plaso project?: *[pfad]* 3. What's the name of the plugin?: *test* 4. What's the path to your test file?: [pfad_file] 5. Do you want to have a output example for your SQL Query? [Y/n]: n 6. Please write your SQL script for the plugin: *select * from users* 7. The SQL query was ok. 8. Do you want to name the query parse row: Users ? [Y/n]: *Y* 9. Is the column a time event? updatedAt [Y/n]: *n* 10. Is the column a time event? createdDate [Y/n]: *Y* 11. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: id that this 12. Timestamps are not in valid format. Reenter them correctly [name,name...]: id, that, this 13. Timestamps are not in valid format. Reenter them correctly [name,name...]:id,that,this 14. Added: createdDate,id Failed: that,this 15. Do you want to add more timestamps? [y/N]: y 16. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: name,that 17. Added: createdDate,id,name 18. Failed: that,this 19. Do you want to add more timestamps? [y/N]: n 20. Does the event Users need customizing? [y/N]: *N* 21. Do you want to add another Query? [Y/n]: *n* 22. Do you want to Generate the files [Y/n]: Y
Erwartete Resultate	**Konsolen Output:** create [pfad]/plaso/formatters/test.py create [pfad]/plaso/parsers/sqlite_plugins/test.py create [pfad]/tests/formatters/test.py create [pfad]/tests/parsers/sqlite_plugins/test.py copy [pfad]/test_data/test.db create [pfad]/plaso/parsers/sqlite_plugins/__init__.py create [pfad]/plaso/formatters/__init__.py
	File Output: Inhalt der generierten Files stimmen überein mit Files unter: PlasoScaffolder/src/tests/end-to-end-test/ExpectedEasyGenerationOwnTimestampFiles
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_easy_generation_with_own_time_event.py

Tabelle 23: End-to-End-Test 12

Test 13 - Mit Customizing	
Funktionalität	Testen von einem Event der Anpassungen braucht
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - [pfad]: Existierender pfad zu einem leeren Ordner - [pfad_file]: Pfad zum twitter_ios.db File - twitter_ios.db File ist in einem gültigen Zustand
Abhängigkeiten	- Test 3
Ablauf	1. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite* 2. What's the path to the plaso project?: *[pfad]* 3. What's the name of the plugin?: *test* 4. What's the path to your test file?: [pfad_file] 5. Do you want to have a output example for your SQL Query? [Y/n]: n 6. Please write your SQL script for the plugin: *select * from users* 7. The SQL query was ok. 8. Do you want to name the query parse row: Users ? [Y/n]: *Y* 9. Is the column a time event? updatedAt [Y/n]: *Y* 10. Is the column a time event? createdDate [Y/n]: *Y* 11. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: abort 12. Does the event Users need customizing? [y/N]: *Y* 13. Enter columns that are customizable [columnName,aliasName...] or [abort]: *this that* 14. Column names are not in valid format. Reenter them correctly [name,name...]: *this, that* 15. Column names are not in valid format. Reenter them correctly [name,name...]: *this,that* 16. Added: Failed: that,this 17. Do you want to add more columns that are customizable? [y/N]: *Y* 18. Enter columns that are customizable [columnName,aliasName...] or [abort]: *this,location,id* 19. Added: location Failed: id,that,this 20. Do you want to add more columns that are customizable? [y/N]: *N* 21. Do you want to add another Query? [Y/n]: *n* 22. Do you want to Generate the files [Y/n]: Y
Erwartete Resultate	**Konsolen Output:** create [pfad]/plaso/formatters/test.py create [pfad]/plaso/parsers/sqlite_plugins/test.py create [pfad]/tests/formatters/test.py create [pfad]/tests/parsers/sqlite_plugins/test.py copy [pfad]/test_data/test.db create [pfad]/plaso/parsers/sqlite_plugins/__init__.py create [pfad]/plaso/formatters/__init__.py
	File Output: Inhalt der generierten Files übereinstimmen mit Files unter: PlasoScaffolder/src/tests/end-to-end-test/ExpectedEasyGenerationWithCustomizationFiles
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_easy_generation_with_customizing.py

Tabelle 24: End-to-End-Test 13

Test 14 - Mit Select mit einfachem Join	
Funktionalität	Testen von einfachen Join im Select
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - [pfad]: Existierender pfad zu einem leeren Ordner - [pfad_file]: Pfad zum twitter_ios.db File - twitter_ios.db File ist in einem gültigen Zustand
Abhängigkeiten	- Test 3
Ablauf	1. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite* 2. What's the path to the plaso project?: *[pfad]* 3. What's the name of the plugin?: *test* 4. What's the path to your test file?: [pfad_file] 5. Do you want to have a output example for your SQL Query? [Y/n]: n *6.* Please write your SQL script for the plugin: *select * from users join statuses* 7. Please use an alias (AS) for those column names: id updatedAt *8.* Please write your SQL script for the plugin: *select id as userid join statuses* 9. Error: near "join": syntax error *10.* Please write your SQL script for the plugin]: *select id as userid from users join statuses* 11. Error: ambiguous column name: id 12. Please write your SQL script for the plugin: select users.id as userid, users.updatedAt, createdDate from users join statuses 13. The SQL query was ok. 14. Do you want to name the query parse row: Usersstatuses? [Y/n]: *Y* 15. Is the column a time event? updatedAt [Y/n]: *Y* 16. Is the column a time event? createdDate [Y/n]: *Y* 17. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: *abort* 18. Does the event Users need customizing? [y/N]: *N* 19. Do you want to add another Query? [Y/n]: *N* 20. Do you want to Generate the files [Y/n]: Y
Erwartete Resultate	**Konsolen Output:** create [pfad]/plaso/formatters/test.py create [pfad]/plaso/parsers/sqlite_plugins/test.py create [pfad]/tests/formatters/test.py create [pfad]/tests/parsers/sqlite_plugins/test.py copy [pfad]/test_data/test.db create [pfad]/plaso/parsers/sqlite_plugins/__init__.py create [pfad]/plaso/formatters/__init__.py
	File Output: Inhalt der generierten Files übereinstimmen mit Files unter: PlasoScaffolder/src/tests/end-to-end-test/ExpectedWithJoinFiles
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_select_easy_join.py

Tabelle 25: End-to-End-Test 14

Test 15 - Mit Select mit Sonderzeichen	
Funktionalität	Testen von Select Query mit Sonderzeichen
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - [pfad]: Existierender pfad zu einem leeren Ordner - [pfad_file]: Pfad zum twitter_ios.db File - twitter_ios.db File ist in einem gültigen Zustand
Abhängigkeiten	- Test 3 - Test 14
Ablauf	1. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite* 2. What's the path to the plaso project?: *[pfad]* 3. What's the name of the plugin?: *test* 4. What's the path to your test file?: [pfad_file] 5. Do you want to have a output example for your SQL Query? [Y/n]: n 6. Please write your SQL script for the plugin: select users.id as "userid" from users join statuses 7. Warning: Don't use any characters beside a-z A-Z 0-9 . ; , * = _' 8. Please write your SQL script for the plugin: 9. Warning: Don't use any characters beside a-z A-Z 0-9 . ; , * = _' 10. Please write your SQL script for the plugin: select [AS].[ID] AS [the alias], [AS].name AS "the name" , S.id from users as [AS] join statuses as "S" 11. Warning: Don't use any characters beside a-z A-Z 0-9 . ; , * = _' 12. Please write your SQL script for the plugin: select id as the_id from users 13. Warning: Don't use any characters beside a-z A-Z 0-9 . ; , * = _' 14. Please write your SQL script for the plugin: select somealias.id from users as somealias 15. Warning: Don't use any alias for a table name 16. Please write your SQL script for the plugin: *select * from users* 17. The SQL query was ok. 18. Do you want to name the query parse row: Users ? [Y/n]: *Y* 19. Is the column a time event? updatedAt [Y/n]: *Y* 20. Is the column a time event? createdDate [Y/n]: *Y* 21. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: *abort* 22. Does the event Users need customizing? [y/N]: *N* 23. Do you want to add another Query? [Y/n]: *n* 24. Do you want to Generate the files [Y/n]: Y
Erwartete Resultate	**Konsolen Output:** create [pfad]/plaso/formatters/test.py create [pfad]/plaso/parsers/sqlite_plugins/test.py create [pfad]/tests/formatters/test.py create [pfad]/tests/parsers/sqlite_plugins/test.py copy [pfad]/test_data/test.db create [pfad]/plaso/parsers/sqlite_plugins/__init__.py create [pfad]/plaso/formatters/__init__.py
	File Output: Inhalt der generierten Files übereinstimmen mit Files unter: PlasoScaffolder/src/tests/end-to-end-test/ExpectedEasyGenerationFiles
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_select_with_special_characters.py

Tabelle 26: End-to-End-Test 15

Test 16 - Generieren mit Angabe von Optionen beim Aufrufen	
Funktionalität	Ablauf testen, mit der Eingabe der möglichen Optionen beim Aufruf
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - [pfad]: Existierender pfad zu einem leeren Ordner - [pfad_file]: Pfad zum twitter_ios.db File - twitter_ios.db File ist in einem gültigen Zustand
Abhängigkeiten	- Test 3
Ablauf	1. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite --path [pfad] --name test --testfile [pfad_file] –no-sql* 2. Please write your SQL script for the plugin: *select * from users* 3. The SQL query was ok. 4. Do you want to name the query parse row: Users ? [Y/n]: *Y* 5. Is the column a time event? updatedAt [Y/n]: *Y* 6. Is the column a time event? createdDate [Y/n]: *Y* 7. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: *abort* 8. Does the event Users need customizing? [y/N]: *N* 9. Do you want to add another Query? [Y/n]: *n* 10. Do you want to Generate the files [Y/n]: Y
Erwartete Resultate	**Konsolen Output:** create [pfad]/plaso/formatters/test.py create [pfad]/plaso/parsers/sqlite_plugins/test.py create [pfad]/tests/formatters/test.py create [pfad]/tests/parsers/sqlite_plugins/test.py copy [pfad]/test_data/test.db create [pfad]/plaso/parsers/sqlite_plugins/__init__.py create [pfad]/plaso/formatters/__init__.py
	File Output: Inhalt der generierten Files übereinstimmen mit Files unter: PlasoScaffolder/src/tests/end-to-end-test/ExpectedEasyGenerationFiles
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_generation_with_option_at_tool_call.py

Tabelle 27: End-to-End-Test 16

Test 17 - Einfaches Generieren mit Name der schon existiert	
Funktionalität	Ablauf testen, was geschieht, wenn schon ein Plugin mit dem gleichen Namen bereits existiert.
Vorbedingung	- Python 3.6 ist vorhanden - Alle Module vom requirements.txt sind installiert - [pfad]: Existierender pfad zu einem leeren Ordner - [pfad_file]: Pfad zum twitter_ios.db File - twitter_ios.db File ist in einem gültigen Zustand
Abhängigkeiten	- Test 3
Ablauf	1. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite* 2. What's the path to the plaso project?: *[pfad]* 3. What's the name of the plugin?: *test* 4. What's the path to your test file?: [pfad_file]: 5. Do you want to have a output example for your SQL Query? [Y/n]: *n* 6. Please write your SQL script for the plugin: *select * from users* 7. The SQL query was ok. 8. Do you want to name the query parse row: Users ? [Y/n]: *Y* 9. Is the column a time event? updatedAt [Y/n]: *Y* 10. Is the column a time event? createdDate [Y/n]: *Y* 11. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: *abort* 12. Does the event Users need customizing? [y/N]: *N* 13. Do you want to add another Query? [Y/n]: *n* 14. Do you want to Generate the files [Y/n]: Y 15. Plasoscaffolder mit sqlite Command starten: *plasoscaffolder sqlite* 16. What's the path to the plaso project?: *[pfad]* 17. What's the name of the plugin?: *test* 18. Plugin exists. Choose new Name: *test_plugin* 19. What's the path to your test file?: [pfad_file]: 20. Do you want to have a output example for your SQL Query? [Y/n]: *n* 21. Please write your SQL script for the plugin: *select * from users* 22. The SQL query was ok. 23. Do you want to name the query parse row: Users ? [Y/n]: *Y* 24. Is the column a time event? updatedAt [Y/n]: *Y* 25. Is the column a time event? createdDate [Y/n]: *Y* 26. Enter (additional) timestamp events from the query [columnName,aliasName...] or [abort]: *abort* 27. Does the event Users need customizing? [y/N]: *N* 28. Do you want to add another Query? [Y/n]: *n* 29. Do you want to Generate the files [Y/n]: Y
Erwartete Resultate	**Konsolen Output bei 14:** create [pfad]/plaso/formatters/test.py create [pfad]/plaso/parsers/sqlite_plugins/test.py create [pfad]/tests/formatters/test.py create [pfad]/tests/parsers/sqlite_plugins/test.py copy [pfad]/test_data/test.db create [pfad]/plaso/parsers/sqlite_plugins/__init__.py create [pfad]/plaso/formatters/__init__.py **Konsolen Output bei 29:** create [pfad]/plaso/formatters/test.py create [pfad]/plaso/parsers/sqlite_plugins/test.py create [pfad]/tests/formatters/test.py create [pfad]/tests/parsers/sqlite_plugins/test.py copy [pfad]/test_data/test.db

	edit [pfad]/plaso/parsers/sqlite_plugins/__init__.py edit [pfad]/plaso/formatters/__init__.py
	File Output: Inhalt der generierten Files übereinstimmen mit Files unter: PlasoScaffolder/src/tests/end-to-end-test/ExpectedEasyGenerationFiles für nach dem Generieren des ersten Plugins und ExpectedTwoPluginsFiles nach dem Generieren des 2ten Plugins
Pfad zum automatisierten Test	plasoscaffolder/tests/end_to_end_test/test_generation_with_an_existing_name.py

Tabelle 28: End-to-End-Test 17

10 Code Analyse

Der Projektstatus vom letzten Commit während dieser Bachelorarbeit ist, dass der Travis-CI Test sowie der AppVeyor Test fehlerfrei durchgeführt wurden, sowie Coveralls eine Testabdeckung von 100% berechnet.

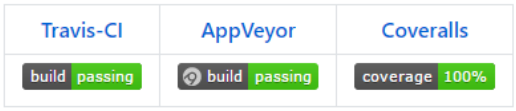


Abbildung 49: Projekt Status

10.1 Testabdeckung

Gefordert wurde eine Testabdeckung von 70%.
Nur mit den Unittests, ohne Integrationstests und ohne End-to-End Tests wurde eine Testabdeckung von 99.62% erzielt. Die Zeilen, welche mit den Unittests nicht getestet werden konnten, wurden jedoch mit den End-to-End Tests getestet.

Abbildung 50: Coveralls Test Coverage

10.2 Travis

Insgesamt wurden 517 Builds auf Travis ausgeführt.
Dies bedeutet die Applikation läuft fehlerfrei auf Linux mit Python 3.6.

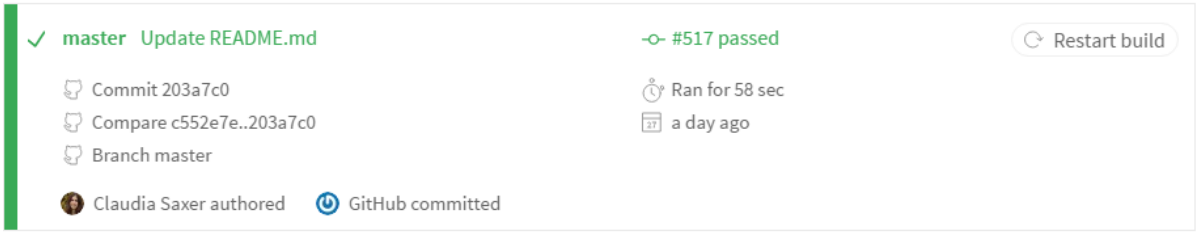


Abbildung 51: Travis Build

10.3 AppVeyor

Insgesamt wurden 531 Builds auf AppVeyor ausgeführt.
Dies bedeutet die Applikation läuft fehlerfrei auf Windows mit Python 3.6.
Da AppVeyor for Travis eingerichtet wurde, besitzt es einige Builds mehr.

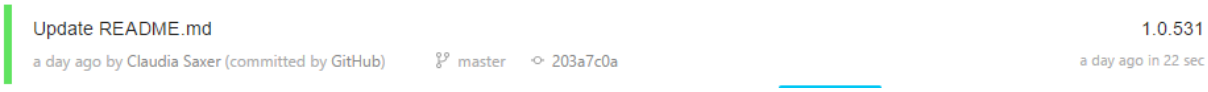


Abbildung 52: Appvoyer Build

10.4 Github

10.4.1 Pullrequest

Es wurden insgesamt 10 Pull Requests mit Github erstellt.
Die Antworten waren hauptsächlich Änderungswünsche zum Codestyle Guide von Plaso.

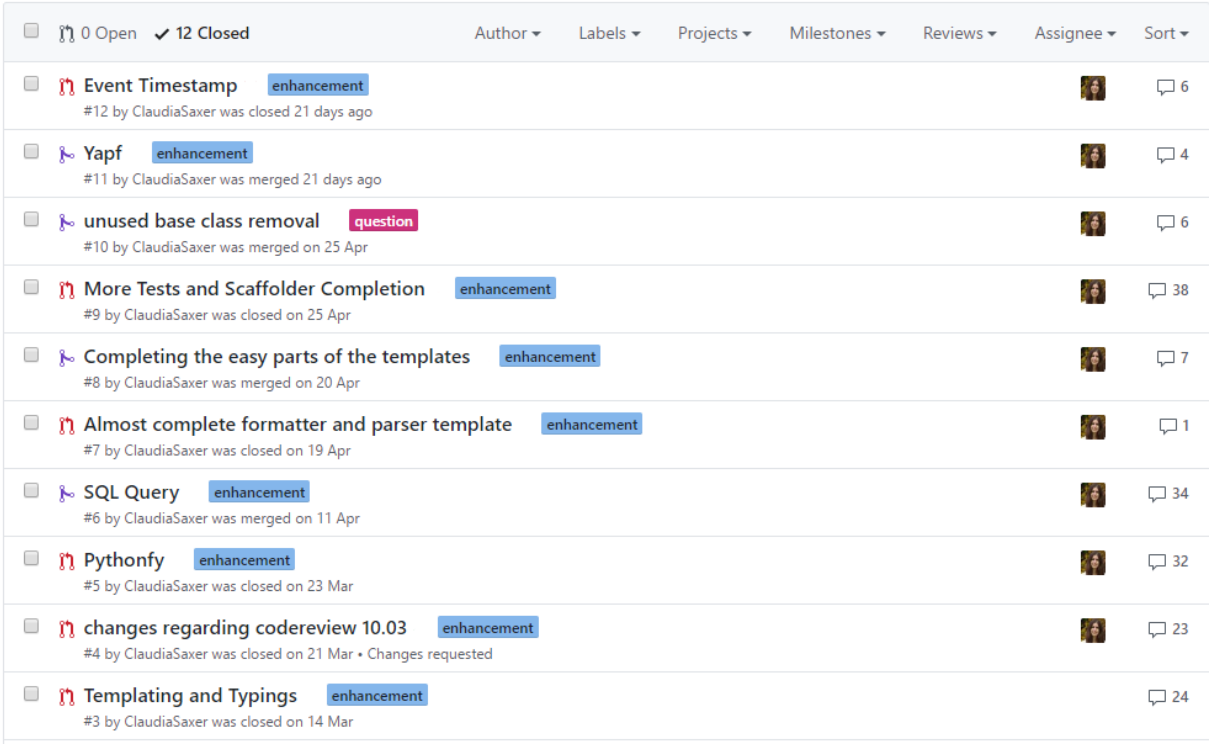


Abbildung 53: Pullrequest

10.5 Git Statistik

Mit gitstats[96] wurde eine Statistik über das Repository erstellt.
Diese Statistik ist in der Abgabe unter gitstats ersichtlich.

10.5.1 Code Zeilen

Interessant ist die Auswertung der Anzahl Code Zeilen pro Filetyp.
Es wurden 202 Files in Python erstellt sowie 18737 Zeilen Code dafür geschrieben.

[96] Siehe gitstats unter: [22]

Da im Order docs die generierte API Dokumentation für die github Page liegt, ist die Auswertung ein wenig verfälscht. Dazu gehören die Files für html, js, rst, css, png, inv, md und gif.

Extension	Files (%)	Lines (%)	Lines/file
py	202 (45.09%)	18737 (27.90%)	92
db	4 (0.89%)	527 (0.78%)	131
jinja2	7 (1.56%)	233 (0.35%)	33
yml	2 (0.45%)	59 (0.09%)	29
bat	1 (0.22%)	54 (0.08%)	54
yapf	2 (0.45%)	6 (0.01%)	3
html	128 (28.57%)	31056 (46.25%)	242
js	9 (2.01%)	13117 (19.53%)	1457
txt	34 (7.59%)	1237 (1.84%)	36
rst	33 (7.37%)	1232 (1.83%)	37
css	4 (0.89%)	960 (1.43%)	240
png	10 (2.23%)	43 (0.06%)	4
inv	1 (0.22%)	29 (0.04%)	29
md	1 (0.22%)	15 (0.02%)	15
gif	1 (0.22%)	9 (0.01%)	9
	9 (2.01%)	339 (0.50%)	37

Tabelle 29: Code Zeilen gesamtes Projek

Wenn man sich nur die Auswertung für den selbst geschrieben Code ansieht, sieht dies anders aus. Auch wurden die Datenbankfiles abgezogen, da hier unter anderem die Testdatenbank twitter_ios.db von Plaso verwendet wurde, jedoch wurden auch zwei Datenbanken mit Testwerten selbst erstellt, um alle Typen und verschiedene Namen zu testen.

Extension	Files (%)	Lines (%)	Lines/file
py	202 (94.09%)	18737 (98.14%)	92
jinja2	7 (1.56%)	233 (1.23%)	33
yml	2 (0.9%)	59 (0.32%)	29
bat	1 (0.5%)	54 (0.28%)	54
yapf	2 (0.9%)	6 (0.04%)	3

Tabelle 30: Codezeilen nur selbst geschriebener Code

10.5.2 Commit

Es wurden insgesamt 459 Commits erstellt.

Es ist schön ersichtlich, dass während der gesamten Projektzeit Code geschrieben wurde, da iterativ gearbeitet wurde. Jedoch im Februar weniger, da hier mehr Zeit für den Projektplan und die Anforderungsspezifikation gebraucht wurde. Im Juni wurde nur noch an der Dokumentation gearbeitet.

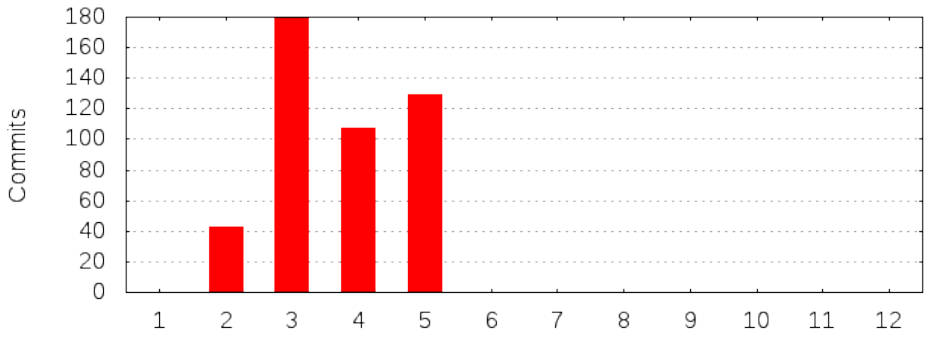


Abbildung 54: Commits pro Monat

Folgend eine genaure Übersicht aus GitHub.

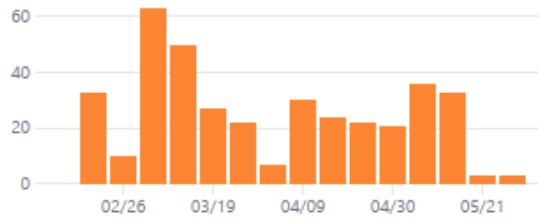


Abbildung 55: GitHub Aktivität

10.5.3 Autor

Es bestehen drei Autoren, weil mit verschiedenen Geräten gearbeitet und der Autor nicht auf allen Geräten gleich angegeben wurde.
Der Autor wird mit den Git Settings vor dem ersten Commit definiert.

```
git config --global user.name "Claudia Saxer"
git config --global user.email ...@gmail.com
```

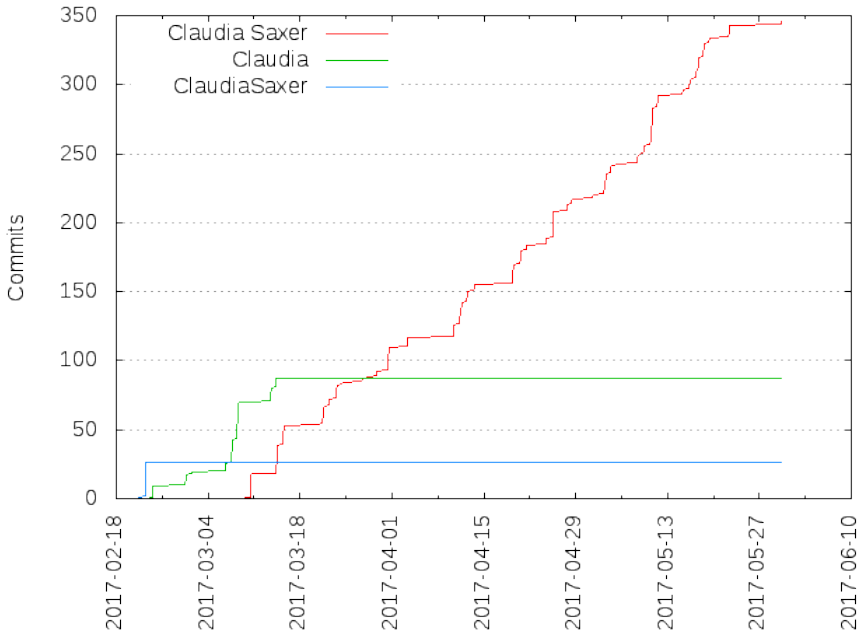


Abbildung 56: Commits pro Autor

- Claudia Saxer ist der Rechner bei der HSR
- Claudia ist der privater Rechner
- ClaudiaSaxer war ein Ersatzgerät, da der private Rechner in Reparatur war.

11 Zusammenfassung

11.1 Ausblick

11.1.1 Einbetten in Plaso

In den nächsten Wochen nach der Arbeit wird der Code des Plaso Scaffolder in das Repository von Plaso hinzugefügt und zukünftige Features werden definiert.

11.1.2 Verbesserungsvorschläge

In diesem Kapitel sind einige Features beschrieben, welche für den Plaso SQLite Plugin Scaffolder zu einem späteren Zeitpunkt hinzugefügt werden können.

Yapf
Da Plaso auch Yapf verwendet, kann das Style File von Plaso direkt verwendet werden, indem das File im Plaso Ordner lokalisiert wird. Falls das File nicht gefunden wird, soll der User nach dem Pfad zu Style File gefragt werden. Somit braucht der Plaso Scaffolder kein Style File mehr und Änderungen müssen nicht mehrfach nachgetragen werden.

Darstellung
Um die Ausgabe des ResultSet des ausgeführten Queries schöner anzuzeigen, kann die Ausgabe mit einem Jinja2 Template File strukturiert werden.

Query parsen
Um dem User mehr Möglichkeiten für das Schreiben des SQL Statement zu geben, kann der Parsing Algorithmus des Queries erweitert werden.

Logging
Um Logging während des Entwickelns zu ermöglichen, kann ein weiterer Parameter für das Starten des Tools erstellt werden, welcher standardmässig nicht abgefragt wird, um damit den Entwickler beim Erweitern des Programmes nicht nur mit dem Debugger zu unterstützen.

11.1.3 Erweiterungsmöglichkeiten

In diesem Kapitel sind einige Erweiterungsmöglichkeiten beschrieben, welche einerseits für den Plaso SQLite Plugin Scaffolder spezifisch oder auch für den Plaso Scaffolder im Allgemeinen sind.

Weitere Plugin Scaffolder

Plaso beinhaltet nicht nur Plugins für SQLite sondern auch weitere Plugins für andere Dateisysteme. Geplant ist, das Tool um weitere Plaso Plugin Scaffolder mit Unterstützung anderer Dateisystemen zu erweitern.

Eingaben abspeichern

Da die Anforderung war ein interaktives Tool zu schreiben, ist einige Userinteraktion gefordert. In Zukunft könnte man die Eingaben des Users abspeichern und die Möglichkeit hinzufügen, das Programm mit einem Inputfile zu starten. Somit kann das Programm ohne Userinteraktion durchgeführt und wiederholt werden.

Hilfestellung

Die Hilfestellungen und Informationen für den User sind möglichst kurz und verständlich gehalten. Um den User noch mehr zu unterstützen, kann ein neues Argument mit dem Namen verbose erstellt werden, welches steuert, dass dem User noch mehr Informationen zu den geforderten Eingaben geliefert werden. Damit wird der Ablauf für die Generation durchsichtiger und dem User wird bewusst, für was welche Eingaben verwendet werden.

Diese Informationen können in einem separaten File gespeichert werden, welches auch gleichzeitig verwendet werden kann um mit Jinja2 Templates ein Wiki zu generieren, welches erklärt welche Eingaben wofür verwendet werden.

Verknüpfung mit Plaso

Da sich die Verwendung der Timestamps schwieriger gestaltete als bisher vom Betriebspartner angenommen und ein Workaround für die Parser Testfile existiert[97] kann ein Update den Workaround entfernen.

Es braucht nicht mehr viel um die vollständige Generierung des Plugins zu beenden.

Wenn das Plugin ohne zusätzliche Editierung vom User auf Plaso lautbar ist, kann der Plaso Scaffolder noch ein Schritt weiter mit Plaso verknüpft werden.

Nach dem Generieren des Parser File und Formatter File sowie dem Editieren der Init Files kann das Plugin automatisch mit dem Scaffolder auf Plaso ausgeführt werden. Dies ermöglicht, dass die tatsächlichen Events von Plaso für das Plugin, welches noch in Generierung ist, verfügbar sind.

Somit kann das Parser Testfile mit den effektiven Daten von Plaso erstellt werden und abgesehen von einigen Kommentarinformationen braucht es keine nachträgliche Veränderung der Files mehr vom User.

[97] Siehe Workaround unter Kapitel: 5.9.4

11.2 Ergebnisse

Das Ergebnis dieser Bachelorarbeit ist eine Command Line Interface Applikation für die Generation beliebiger SQLite Plugins für Plaso. Der User hat die Möglichkeit das Tool mit Parametern zu starten oder direkt in den interaktiven Ablauf einzusteigen und innerhalb wenigen Minuten ein Plugin von durchschnittlich 600 Zeilen Code zu generieren.

Der User kann Dank der Validierung seiner Eingaben gleichzeitig kontrollieren, ob seine Eingaben korrekt sind und somit noch die Richtigkeit seiner Eingaben erhöhen. Dem User wird nicht nur Kopierarbeit erspart, sondern auch viele Codierungsarbeit, welche individuell wären.

Die generierten Files sind korrekt formatiert und verfolgen die grundsätzlichen Codierungsrichtlinien.

Um die Applikation zu testen wurden Continuous Integration Server für Linux sowie Windows aufgesetzt und Unittests für die gesamte Applikation geschrieben, welche eine Testabdeckung von 100% erzielen. Zusätzlich wurden End-to-End Tests geschrieben, die automatisch auf Linux oder manuell mit Hilfe des Testplans ausgeführt werden können.

Die Command Line Interface Applikation wurde mit einer gut strukturierten Software Architektur umgesetzt, welche einfach mit weiteren Plugin Scaffolder für andere Dateiformate erweiterbar ist. Zusätzlich können, durch die lose Koppelung, einzelne Module unkompliziert ausgetauscht werden.

Durch die vollständige Dokumentation des Codes mit Docstring Kommentaren und Generierung einer API Dokumentation ist das Programm für weitere Entwicklungen und Unterhalt bestens vorbereitet.

Da die Files mit Jinja2 Templates generiert wurden, können mit wenigen Anpassungen in den Templates das Design und der Aufbau der Files ohne Änderung in der Logik angepasst und nach Belieben verändert werden. Nur die Integrationstests und End-to-End Tests müssen angepasst werden.

Durch das Verwenden von Click, welches die Kommunikation mit dem User behandelt, können die mit dem User geteilten Informationen an einer Stelle zentral angepasst werden.

Die Applikation ist in Python 3.6 programmiert und durch Pylint komplett validiert.

Die gesamte Applikation wurde den Anforderungen entsprechend codiert indem die Styleguides eingehalten wurden, die Codereviews bestanden und die Testabdeckung auf Linux und Windows hoch übertroffen wurden.

Obwohl es keine Anforderung war, ist der Plaso SQLite Plugin Scaffolder bereits auf PyPI mit derselben Lizenz wie Plaso veröffentlicht und somit mit pip schnell und unkompliziert installier- und ausführbar.

Das in dieser Bachelorarbeit entwickelte Tool ist somit direkt produktiv verwendbar und kann jederzeit mit weiteren Features erweitert werden.

Alle geforderten Anforderungen und Features sowie einige optionale Features wurden umgesetzt und getestet.

Das Ziel dieser Bachelorarbeit ein Programm zu entwickeln, welches dem User mit einem interaktiven Command Line Interface Tool einen Programmierrahmen generiert, der beim Erstellen von SQLite Plugins hilft, wurde übertroffen.

12 Verzeichnisse

12.1 Abbildungsverzeichnis

Abbildung 1: Fall Beispiel 10
Abbildung 2: Komponenten Diagramm Plugin mit Astah 11
Abbildung 3: Plaso Plugin Aufbau 11
Abbildung 4: MVC simple 19
Abbildung 5: MVVM 19
Abbildung 6: MVC + MVVM 20
Abbildung 7: Schichtenarchitektur 20
Abbildung 8: Finale MVC Architektur 21
Abbildung 9: Schichtenmodell 26
Abbildung 10: Involvierte Files Tree 26
Abbildung 11: ResultSet PRAGMA table_info(users) 38
Abbildung 12: EXPLAIN select users.id, name, card from users join statuses 39
Abbildung 13: EXPLAIN select users.id, name, card from users join statuses order by name 40
Abbildung 14: Click Prototyp Filestruktur 44
Abbildung 15: Prototyp Click Output 1 45
Abbildung 16: Prototyp Click Output 2 45
Abbildung 17: Prototyp Click Output 3 45
Abbildung 18: Einfaches seqdiag Diagram 47
Abbildung 19: Screenshot pip install plasoscaffolder 52
Abbildung 20: Screenshot normaler Ablauf 52
Abbildung 21: Screenshot Ablauf mit Fehlermeldungen 53
Abbildung 22: Screenshot plasoscaffolder sqlite --help 54
Abbildung 23: Screenshot normaler Ablauf mit Options 54
Abbildung 24: Screenshot fehlerhafter Ablauf mit Options 54
Abbildung 25: Komponenten Diagramm 55
Abbildung 26: Klassendiagramm BLL MappingHelper 56
Abbildung 27: Klassendiagramm BLL Mapper 57
Abbildung 28: Klassendiagramm BLL PathHelper 59
Abbildung 29: Klassendiagramm BLL PluginHelper 60
Abbildung 30: Klassendiagramm BLL SQLiteGenerator 63
Abbildung 31: Klassendiagramm Common 64
Abbildung 32: Klassendiagramm DAL TypeHelper 68
Abbildung 33: Klassendiagramm DAL Query 69
Abbildung 34: Klassendiagramm Frontend 72
Abbildung 35: Klassendiagramm Model DataModel 74
Abbildung 36: Klassendiagramm Model ColumnModel 75
Abbildung 37: Schichtenmodell der in den Sequenzdiagrammen abgebildeten Module 78
Abbildung 38: Sequenzdiagramm einer Übersicht 80
Abbildung 39: Sequenzdiagramm vom Start 82
Abbildung 40: Sequenzdiagramm der Methode SourcePath() 83
Abbildung 41: Sequenzdiagramm der Methode PluginName() 84
Abbildung 42: Sequenzdiagramm der Methode TestPath() 86
Abbildung 43: Sequenzdiagramm der Methode SQLQuery() 88
Abbildung 44: Sequenzdiagramm für die Methode CreateSQLQueryModelWithUserInput() Teil 1 90
Abbildung 45: Sequenzdiagramm für die Methode CreateSQLQueryModelWithUserInput() Teil 2 92

Abbildung 46: Sequenzdiagramm für die Methode CreateSQLQueryModelWithUserInput() Teil 3 94
Abbildung 47: Sequenzdiagramm für die Methode Timestamps() 96
Abbildung 48: Sequenzdiagramm für die Methode Generate() 98
Abbildung 49: Projekt Status 118
Abbildung 50: Coveralls Test Coverage 118
Abbildung 51: Travis Build 119
Abbildung 52: Appvoyer Build 119
Abbildung 53: Pullrequest 120
Abbildung 54: Commits pro Monat 122
Abbildung 55: GitHub Aktivität 122
Abbildung 56: Commits pro Autor 123

12.2 Tabellenverzeichnis

Tabelle 1: Plugin Klassen Konstante 13
Tabelle 2: Formatter Klassen Konstante 15
Tabelle 3: Code Struktur unter src 27
Tabelle 4: Code Struktur unter src/plasoscaffolder 27
Tabelle 5: Code Struktur unter src/tests 28
Tabelle 6: Manuelles Testverfahren Vorlage 30
Tabelle 7: Mapping zwischen SQLite und Python Datentypen 39
Tabelle 8: Verwendete Tools 50
Tabelle 9: Verwendete Packages 50
Tabelle 10: Lizenzen 51
Tabelle 11: Beschreibung der in den Sequenzdiagrammen abgebildeten Module 79
Tabelle 12: End-to-End-Test 1 100
Tabelle 13: End-to-End-Test 2 101
Tabelle 14: End-to-End-Test 3 102
Tabelle 15: End-to-End-Test 4 103
Tabelle 16: End-to-End-Test 5 104
Tabelle 17: End-to-End-Test 6 105
Tabelle 18: End-to-End-Test 7 106
Tabelle 19: End-to-End-Test 8 107
Tabelle 20: End-to-End-Test 9 108
Tabelle 21: End-to-End-Test 10 109
Tabelle 22: End-to-End-Test 11 110
Tabelle 23: End-to-End-Test 12 111
Tabelle 24: End-to-End-Test 13 112
Tabelle 25: End-to-End-Test 14 113
Tabelle 26: End-to-End-Test 15 114
Tabelle 27: End-to-End-Test 16 115
Tabelle 28: End-to-End-Test 17 117
Tabelle 29: Code Zeilen gesamtes Projek 121
Tabelle 30: Codezeilen nur selbst geschriebener Code 121

13 Literatur und Quellenverzeichnis

[1] K. Gudjonsson, «2013 Open Source Digital Forensics Conference,» in *Google Drive*, Internet, 2013.

[2] Google, «Plaso Github,» 2017. [Online]. Available: https://github.com/log2timeline/plaso. [Zugriff am 12 06 2017].

[3] «Pylint,» [Online]. Available: https://www.pylint.org/. [Zugriff am 28 02 2017].

[4] AppVeyor, «AppVeyor,» 2017. [Online]. Available: https://www.appveyor.com/. [Zugriff am 12 06 2017].

[5] Travis, «Travis,» 2017. [Online]. Available: https://travis-ci.org/. [Zugriff am 12 06 2017].

[6] Coveralls, «Coveralls,» [Online]. Available: https://coveralls.io/. [Zugriff am 01 05 2017].

[7] G. Brandl, «Sphinx,» 2017. [Online]. Available: http://www.sphinx-doc.org/en/stable/. [Zugriff am 12 06 2017].

[8] R. t. Docs, «Read the Docs,» 2017. [Online]. Available: https://readthedocs.org/. [Zugriff am 12 06 2017].

[9] pypa, «twine github,» [Online]. Available: https://github.com/pypa/twine. [Zugriff am 02 06 2017].

[10] PyPI, «PyPI,» 2017. [Online]. Available: https://pypi.python.org/pypi. [Zugriff am 12 06 2017].

[11] pypa, «Virtualenv,» 2017. [Online]. Available: https://virtualenv.pypa.io/en/stable/. [Zugriff am 12 06 2017].

[12] pythonwheels, «Wheel,» 2017. [Online]. Available: http://pythonwheels.com/. [Zugriff am 12 06 2017].

[13] JetBrains, «PyCharm,» [Online]. Available: https://www.jetbrains.com/pycharm/. [Zugriff am 08 03 2017].

[14] blockdiag, «seqdiag,» 05 04 2017. [Online]. Available: http://blockdiag.com/en/seqdiag/. [Zugriff am 05 04 2017].

[15] Astah, «Astah,» 05 04 2017. [Online]. Available: http://astah.net/de. [Zugriff am 05 04 217].

[16] Github, «Github,» 2017. [Online]. Available: https://github.com/. [Zugriff am 12 06 2017].

[17] hoxu, «sourceforge gitstats,» [Online]. Available: http://gitstats.sourceforge.net/. [Zugriff am 02 06 2017].

[18] JetBrains, «Youtrack,» 2017. [Online]. Available: https://www.jetbrains.com/youtrack/. [Zugriff am 12 06 2017].

[19] Google, «Google Drive,» 2017. [Online]. Available: https://www.google.com/intl/de_ALL/drive/. [Zugriff am 12 06 2017].

[20] Ubuntu, «Ubuntu 10.04.2 LTS Download,» 2017. [Online]. Available: https://www.ubuntu.com/download/desktop/contribute?version=16.04.2&architecture=amd64. [Zugriff am 12 06 2017].

[21] docs.python, «unittest - unit testing framework,» 25 4 2017. [Online]. Available: https://docs.python.org/3/library/unittest.html. [Zugriff am 25 4 2017].

[22] pexpect, «readthedocs pexpect,» [Online]. Available: https://pexpect.readthedocs.io/en/stable/. [Zugriff am 05 30 2017].

[23] A. Ronacher, «Jinja2,» 03 04 2017. [Online]. Available: http://jinja.pocoo.org/docs/2.9/. [Zugriff am 12 06 2017].

[24] Google, «Yapf Github,» 09 06 2017. [Online]. Available: https://github.com/google/yapf. [Zugriff am 12 06 2017].

[25] «setuptols,» [Online]. Available: https://setuptools.readthedocs.io/en/latest/. [Zugriff am 01 03 2017].

[26] A. Ronacher, «Click,» 2017. [Online]. Available: http://click.pocoo.org/5/. [Zugriff am 1 3 2017].

[27] A. Yaari, «Colorama PyPi,» 27 04 2017. [Online]. Available: https://pypi.python.org/pypi/colorama. [Zugriff am 12 06 2017].

[28] SQLite, «SQLite,» 12 04 2017. [Online]. Available: https://sqlite.org/lang_keywords.html. [Zugriff am 12 04 2017].

[29] J. Metz, «Style Guide Plaso,» 25 07 2016. [Online]. Available: https://github.com/log2timeline/plaso/wiki/Style-guide. [Zugriff am 21 02 2017].

[30] J. Metz, «Github pylintrc,» 19 03 2016. [Online]. Available: https://github.com/log2timeline/plaso/blob/master/utils/pylintrc. [Zugriff am 28 02 2017].

[31] «Add Twitter iOS Plugin #715,» 02 06 2017. [Online]. Available: https://codereview.appspot.com/296280043/ . [Zugriff am 28 02 2017].

[32] Microsoft, «Python Tools for Visual Studio,» [Online]. Available: https://microsoft.github.io/PTVS/. [Zugriff am 08 03 2017].

[33] SQLite, «Explain Query Plan,» 12 04 2017. [Online]. Available: http://www.sqlite.org/lang_explain.html. [Zugriff am 12 04 2017].

[34] SQLite, «SQLite Transaction,» 12 04 2017. [Online]. Available: https://sqlite.org/lang_transaction.html. [Zugriff am 12 04 2017].

[35] docs.python, «unittest.mock - mock object library,» 25 04 2017. [Online]. Available: https://docs.python.org/3/library/unittest.mock.html. [Zugriff am 25 04 2017].

[36] stackoverflow, «stackoverflow how-do-i-get-pylint-to-recognize-numpy-members,» [Online]. Available: https://stackoverflow.com/questions/20553551/how-do-i-get-pylint-to-recognize-numpy-members. [Zugriff am 30 05 2017].

[37] AIDaniel, «github cloc,» [Online]. Available: https://github.com/AlDanial/cloc. [Zugriff am 02 06 2017].

[38] SQLite, «SQLite Datatype,» [Online]. Available: https://sqlite.org/datatype3.html. [Zugriff am 05 05 2017].

[39] Chromium, «Chromium coding style,» 01 01 2017. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/master/styleguide/styleguide.md. [Zugriff am 06 06 2017].

[40] Apache, «Apache License 2.0,» 04 2004. [Online]. Available: http://www.apache.org/licenses/LICENSE-2.0. [Zugriff am 12 06 2017].

[41] Python, «Python Software Foundation License,» 2017. [Online]. Available: http://docs.python.org/2/license.html. [Zugriff am 12 06 2017].

[42] BSD, «BSD License,» 2017. [Online]. Available: https://opensource.org/licenses/BSD-3-Clause. [Zugriff am 12 06 2017].

[43] MIT, «MIT License,» 2017. [Online]. Available: https://opensource.org/licenses/MIT. [Zugriff am 12 06 2017].

[44] ISC, «ISC License,» 2017. [Online]. Available: https://opensource.org/licenses/ISC. [Zugriff am 12 06 2017].

[45] Wikipedia, «Public domain,» 08 06 2017. [Online]. Available: https://en.wikipedia.org/wiki/Public_domain. [Zugriff am 12 06 2017].

14 Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während dieser Bachelorarbeit unterstützt und motiviert haben.

Ich möchte meinen Dank an folgende Personen aussprechen:

- Herrn Jürg Jucker von HSR, Rapperswil, für die Betreuung der Bachelorarbeit.
- Joachim Metz und Daniel White für die hilfreichen Anregungen und konstruktiven Kritiken.
- Martin Suess für die Organisation und Ermöglichung der Bachelorarbeit.
- Meiner Familie und Freunden für die tatkräftige Unterstützung während der Bachelorarbeit.

15 Anhang

15.1 Persönlicher Bericht

Die Bachelorarbeit habe ich alleine bestritten. Es gab Momente, in denen ich eine Teamarbeit vermisst habe und es hilfreich gewesen wäre Architekturentscheide zu besprechen und Wissen auszutauschen. Ich bereue es jedoch nicht, die Arbeit alleine gemacht zu haben. Ich habe mich alleine auf die Suche nach einer spannenden und herausfordernden Arbeit gemacht und diese schlussendlich mit Martin Suess von Google als Betriebspartner gefunden.

Obwohl ich noch nie etwas mit Python programmiert hatte und ich zuerst ein wenig bedenken davor hatte, habe ich mich der Herausforderung gestellt eine neue Programmiersprache während der Bachelorarbeit zu lernen. Ich hatte die Möglichkeit einen wertvollen Beitrag für die Opensource Community zu leisten und gleichzeitig Neues über die Forensische Datenanalyse zu erfahren und eine neue Programmiersprache zu lernen.

Ich durfte ein Projekt von Anfang bis Ende bearbeiten und hatte grosse Freiheit bei der Architektur, Programmierung sowie beim Design.

Da ich noch nicht viel über Python oder Plaso wusste, habe ich iterativ gearbeitet, was auch ein Anliegen von Seiten Google war, und habe meinen Wissensstand parallel mit dem Projektfortschritt aufgebaut. Trotzdem war der Anfang schwer und ich musste mich in der Freizeit während der Erstellung des Projektplans in Python einarbeiten.

Zudem habe ich für jedes Package, welches ich verwenden wollte, zuerst ein kleines Demo Projekt gemacht, damit ich sicher war, dass es gut geeignet ist.

Der Code unterlief vielen Refactorings, da immer mehr Code hinzukam und somit immer mehr ausgelagert wurde. Da sich die Codebasis so schnell veränderte, habe ich es schlussendlich auch aufgegeben, gewisse Dinge zu dokumentieren, da sich diese genauso schnell veränderten. Deshalb habe ich mich im Laufe der Entwicklungsphase dazu entschieden eine Woche der Testphase zu streichen und dafür eine Woche früher mit der Arbeit am Technischen Bericht zu beginnen. Das kürzen der Testphase war kein Problem, da ich während der Entwicklung konstant getestet habe.

Die Applikation nach jedem Commit automatisch mit Continuous Integration Server auf Linux und Windows zu testen und Coveralls einzurichten hat sich bewährt. So habe ich innert kürzester Zeit Bugs gefunden, welche ich sonst nicht so schnell gefunden hätte. Einerseits lag das daran, dass ich auf Windows entwickelte und mit Travis auch auf Linux getestet wurde. Andererseits war die gesamte Umgebung anders, da ich die Unittests lokal direkt in PyCharm ausführte. Coveralls hat mir geholfen schnell eine Übersicht zu erhalten, welche Codezeilen noch ungetestet sind. Somit konnte ich den gesamten Code sauber testen.

Da ich schon viel mit Git gearbeitet habe und Plaso auf GitHub existiert, war es für mich klar, dass ich auch mit Git arbeiten werde. Ich habe schon vor der Bachelorabeit viele gute Erfahrungen mit Git gesammelt, ob im Team oder alleine. Ich bevorzuge Git über SVN und GitHub ist durch die Möglichkeit von Forks auch ideal für Opensource Projekte geeignet. Ich würde nie mehr ein Entwicklungsprojekt ohne Git durchführen.

Ich hatte die Möglichkeit mich fast jeden Freitag mit Joachim Metz und Martin Suess, welche beide Autoren von Plaso sind, bei Google in Zürich zu treffen. Somit habe ich Ihnen wöchentlich eine Demo des Arbeitfortschritts gezeigt und besprochen, welche Features sie als Nächstes implementiert haben möchten. Gleichzeitig konnte ich sie über Plaso sowie Python ausfragen, da für jedes weitere Feature mehr Knowhow über den Aufbau eines SQLite Plugins gebraucht wurde. Joachim und Daniel waren während des ganzen Projekts überaus freundlich und hilfsbereit und haben mir die Möglichkeit gegeben, ihnen jederzeit Fragen zu Plaso zu stellen, die sie mir innert Kürze beantwortet haben.

Auch gaben sie mir die Möglichkeit Pullrequests via Github zu erstellen zu welchen sie mir Coderreviews gegeben hatten. Diese haben mir geholfen den Styleguide von Plaso einzuhalten und sauberen Python Code zu schreiben.

Durch diese enge Zusammenarbeit habe ich mich während der Bachelorarbeit nie verloren oder alleine gelassen gefühlt. Dafür möchte ich mich bei ihnen herzlich bedanken.

Ich freue mich mein Studium mit einer solch guten Arbeit als Highlight zu beenden und gleichzeitig bin ich traurig, dass diese Zeit auch schon vorbei ist.

15.2 Projekt Management

Mit YouTrack wurde die Zeit aufgezeichnet, Sprints und Task erstellt.

15.2.1 Sprints

Die Sprints hatten eine Grösse von 1 bis 3 Wochen und stimmen mit den Phasen des Projektplanes überein.
Insgesamt wurde für die Bachelorarbeit 363 Stunden und 20 Minuten investiert.
Die Zeit während der ich in der Freizeit Python gelernt habe, habe ich nicht aufgezeichnet.

Sprints	Zeitdauer	Verwendete Zeit
Kickoff	20 Februar – 26 Februar	17h 30min
Planungs-Phase	27 Februar – 5 März	19h 5min
Architektur-Phase 1	6 März – 18 März	46h 30min
Architektur-Phase 2	19 März – 1 April	40h 45min
Development-Phase 1	3 April – 16 April	44h 55min
Development-Phase 2	17 April – 30 April	36h 5min
Development-Phase 3	1 Mai – 14 Mai	33h 40min
Test-Phase	15 Mai – 4 Juni	69h 20min
Dokumentation	5 Juni – 16 Juni	54h 30min

15.2.2 Tasks

Die Tasks wurden in verschiedene Aktivitäten zugeteilt.
Pro Task gab es die Status Open, In Progress, To Verify, Done, Duplicate und Won't fix.

Open	Task wurde noch nicht begonnen und erst erstellt.
In Progress	Task ist in Bearbeitung.
To Verify	Resultat muss bestätigt werden.
Done	Task wurde beendet.
Duplicate	Es gibt schon ein Task mit derselben Aufgabe.
Won't fix	Der Task lässt sich nicht erfolgreich beenden.

15.2.3 User Story

Jeder Task wurde einer User Story zugeordnet.
Ersichtlich ist, dass am meisten Zeit für die Entwicklung verwendet wurde und ein grosser Teil für den Technischen Bericht.

Use Case	Phasen	Verwendete Zeit
Proof of Concept	Planungs-Phase, Architektur-Phase 1	4h 15min
Meetings	Alle Phasen	33h
Dokumentation	Alle Phasen	32h
Technischer Bericht	Alle Phasen ausser Kickoff	102h 50m
Umgebung / Platz einrichten	Kickoff, Planungs-Phase, Architektur-Phase 1	10h 20m
Zwischenpräsentation	Test-Phase	6h
Coding	Architektur-Phase, Development-Phase, Test-Phase	164h 35m
Requirement Definition	Planungs-Phase	2h 45m
Planung	Planungs-Phase	3h 35m

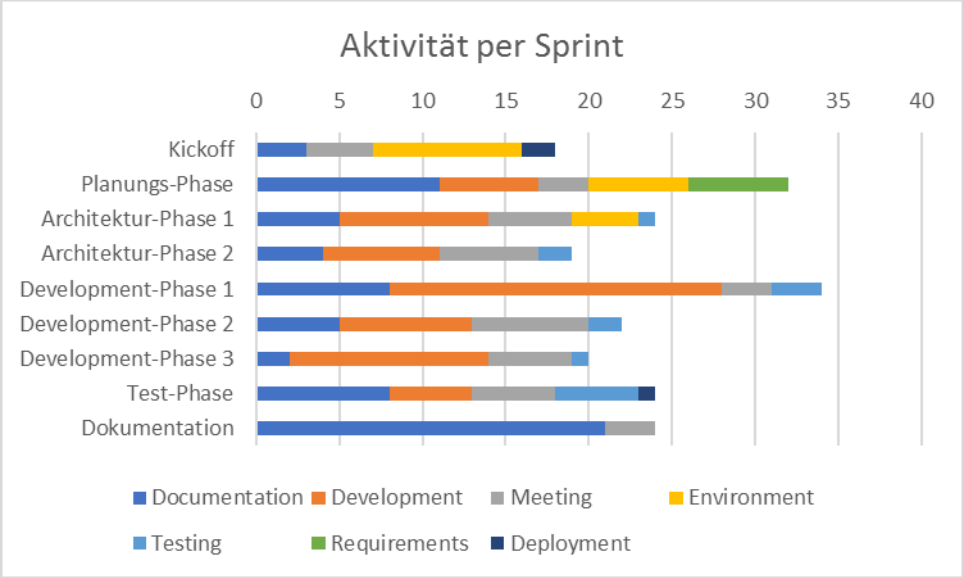
15.2.4 Aktivität

Pro Tasks kann eine Aktivität zur Kategorisierung ausgewählt werden.

Documentation	Task hat mit der Dokumentation zu tun.
Development	Task hat mit der Entwicklung zu tun.
Meeting	Task ist ein Meeting.
Environment	Task hat mit der Entwicklungsumgebung zu tun.
Testing	Task hat mit dem Testen zu tun.
Requirements	Task hat mit den Anforderungen zu tun.
Deployment	Task hat mit dem Deployen der Applikation zu tun.

Aktivität per Sprint

Im Diagramm ist ersichtlich, wie viele Tasks pro Aktivität per Sprint erstellt wurden.



In der Kickoff Phase wurde vor allem an der Entwicklungsumgebung und am Deployment gearbeitet und es wurde noch nicht entwickelt.

In der Planungsphase wurde Zeit in die Anforderungsspezifikation gesteckt und es wurde an den Anforderungen gearbeitet.

In der Architektur Phase wurde noch an der Entwicklungsumgebung gearbeitet sowie Zeit verwendet um die Architektur zu planen, dokumentieren und mit Code umzusetzen. Es werden auch schon bereits die entwickelten Teile getestet.

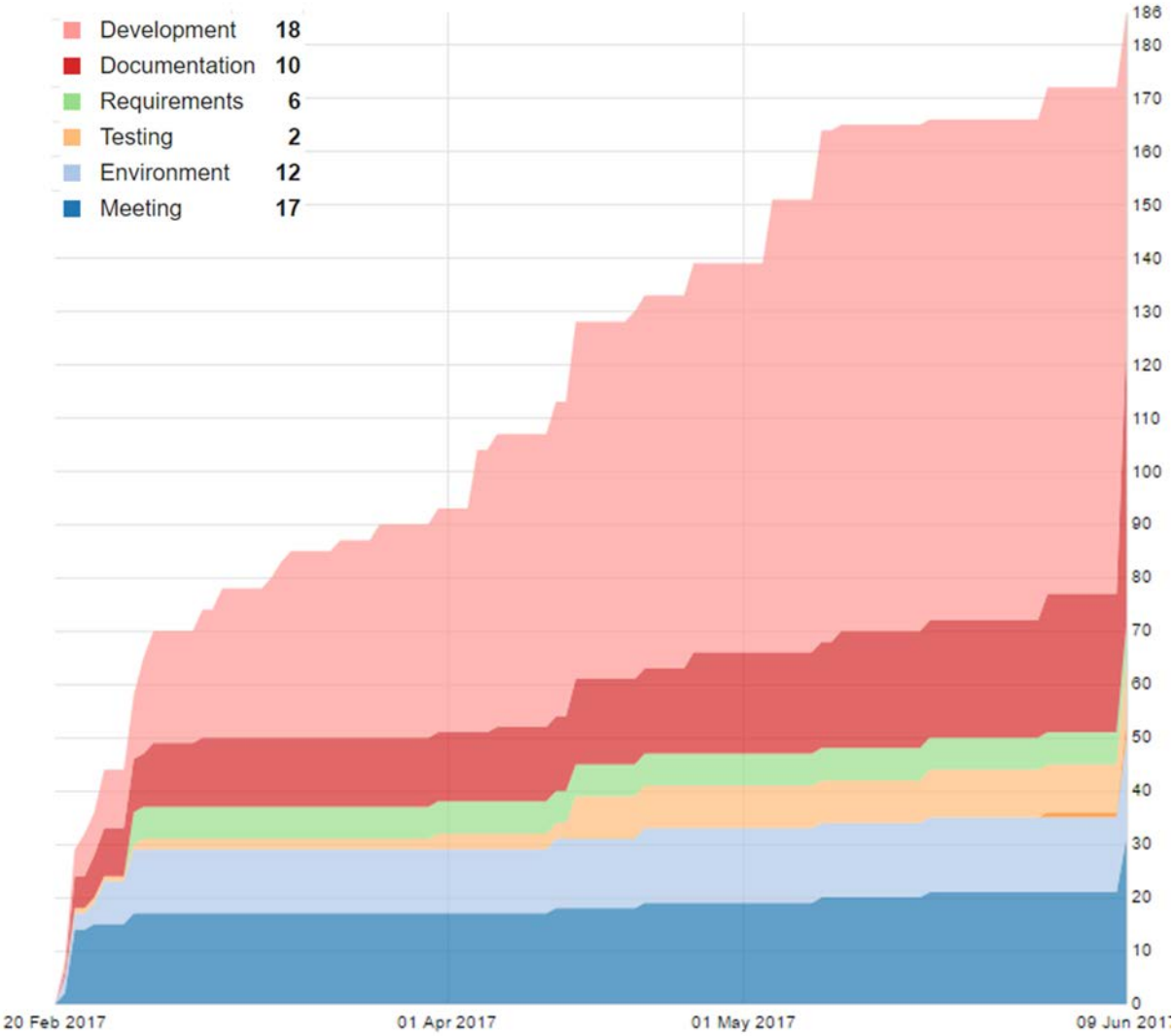
In der Development Phase wurde vor allem entwickelt, jedoch auch gleichzeitig getestet und ein wenig dokumentiert.

In der Test-Phase wurde vor allem getestet, die End-to-End Tests wurden programmiert und dokumentiert und es wurde auch bereits am technischen Bericht gearbeitet, was auch im Projektplan ersichtlich ist.

In der Dokumentation Phase wurde nur noch dokumentiert und die Abgabe vorbereitet.

Kumulierte Aktivität

Das Diagramm für die kumulierte Aktivität veranschaulicht wann Tasks mit welcher Aktivität erstellt wurden.



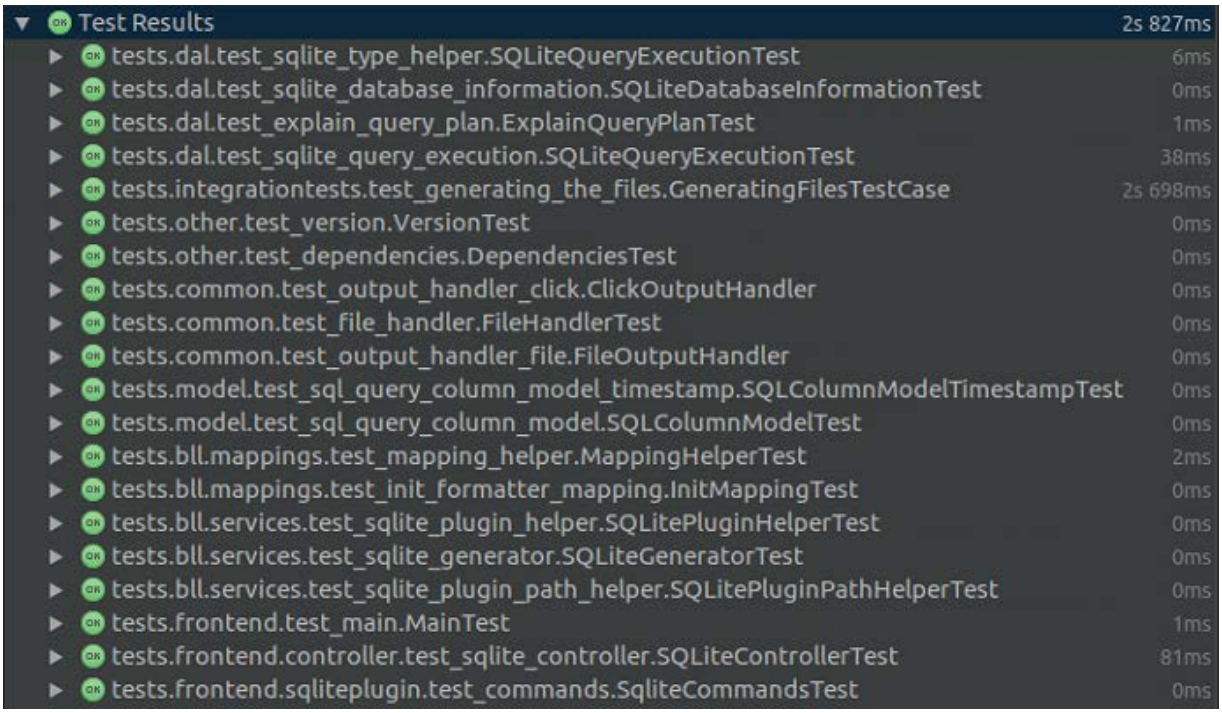
Da schon einige Tasks in der Startup Phase erstellt wurden, verändert sich zum Beispiel die Anzahl Tasks von Meetings fast nicht mehr. Jedoch ist gut ersichtlich, dass die Development Tasks durchgehend über die Zeit erstellt wurden, da in iterativen Phasen gearbeitet wurde und immer neue Features gewünscht wurden.

15.3 Testbericht

Folgend Testberichte zu den Unit Tests sowie zu den End-to-End Tests welche beim Testplan[98] dokumentiert wurden.

15.3.1 **Unit Test**

Die Unittests wurden iterativ geschrieben und bei jedem push ausgeführt.
Folgend ist der Stand vom 12. Juni 2017.
Die Beschreibung zu allen Test ist in der API Dokumentation ersichtlich:
http://plasoscaffolder.readthedocs.io/en/latest/tests.html#module-tests.
Im Bild sind nur die Module welche die Tests beinhaltet ersichtlich. Insgesamt existieren, exklusiv der End-to-End Tests, 203 Unittests.

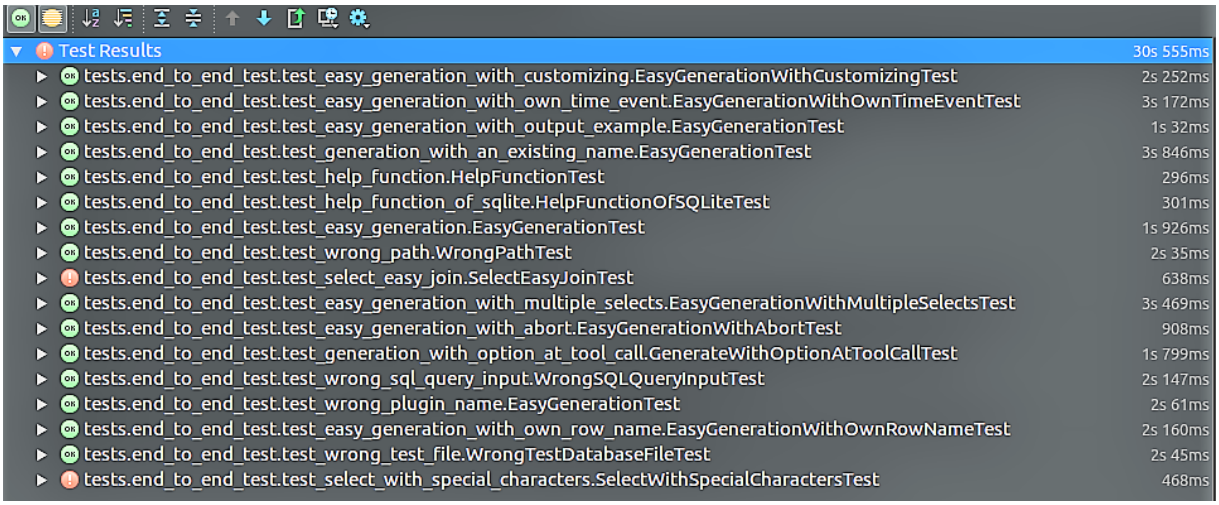


[98] Testplan ersichtlich im Kapitel unter: 9

15.3.2 End-to-End Test

15.3.2.1 Ergebnisse vom End-to-End Test 1 - 17.05.2017

Erste Durchführung der End-to-End Tests nach dem finalen Feature.



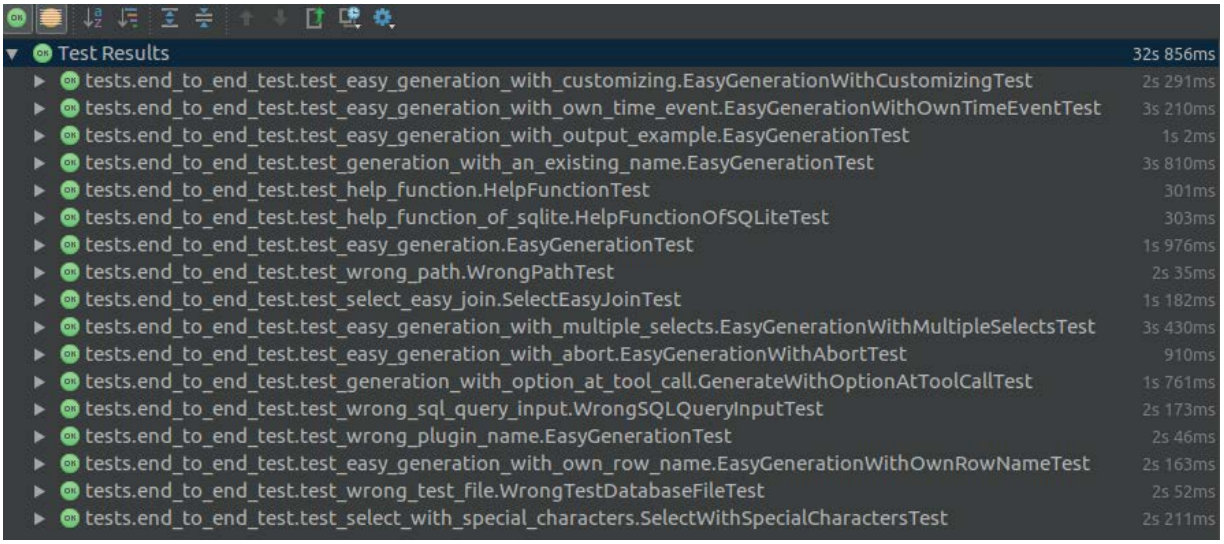
	Passed	Failed	Bemerkung
Test 01	X		
Test 02	X		
Test 03	X		
Test 04	X		
Test 05	X		
Test 06	X		
Test 07	X		
Test 08	X		
Test 09	X		
Test 10	X		
Test 11	X		
Test 12	X		
Test 13	X		
Test 14		X	Es ist nicht implementiert, dass man keinen Alias als Tabellen Namen gebrauchen darf.
Test 15		X	Der Select wird noch nicht auf Zeichen geprüft, die man nicht benutzen darf.
Test 16	X		
Test 17	X		

Dass die Fälle zu Test 14 und Test 15 nicht abgefangen wurden, wurde erst beim ersten End-to-End Test mit produktiven Daten bewusst. Die produktiven Daten kamen von User von Plaso, welche Ideen zu neuen SQLite Plugins hatten.

Als nächsten Schritt werden Unittest erstellt, welche diese Fälle abfangen und fehlschlagen und danach werden diese zwei notwendigen Features noch implementieren.

15.3.2.2 Ergebnisse vom End-to-End Test 1 - 18.05.2017

Durchführung der End-to-End Tests nach Behebung der Fehler von End-to-End Test 1.



	Passed	Failed	Bemerkung
Test 01	X		
Test 02	X		
Test 03	X		
Test 04	X		
Test 05	X		
Test 06	X		
Test 07	X		
Test 08	X		
Test 09	X		
Test 10	X		
Test 11	X		
Test 12	X		
Test 13	X		
Test 14	x		
Test 15	x		
Test 16	X		
Test 17	X		

15.3.2.3 Ergebnisse vom End-to-End Test 3 - 22.05.2017

Tests nach dem finalen Clean Up.

Problem war, dass Yapf 2 Tage zuvor ein Versionsupdate hatte, und die Version im Requirements File nicht definiert wurde.

	Passed	Failed	Bemerkung
Test 01		X	Andere Formatierung der Output Files
Test 02		X	Andere Formatierung der Output Files
Test 03		X	Andere Formatierung der Output Files
Test 04		X	Andere Formatierung der Output Files
Test 05		X	Andere Formatierung der Output Files
Test 06		X	Andere Formatierung der Output Files
Test 07		X	Andere Formatierung der Output Files
Test 08		X	Andere Formatierung der Output Files
Test 09		X	Andere Formatierung der Output Files
Test 10		X	Andere Formatierung der Output Files
Test 11		X	Andere Formatierung der Output Files
Test 12		X	Andere Formatierung der Output Files
Test 13		X	Andere Formatierung der Output Files
Test 14		X	Andere Formatierung der Output Files
Test 15		X	Andere Formatierung der Output Files
Test 16		X	Andere Formatierung der Output Files
Test 17		X	Andere Formatierung der Output Files

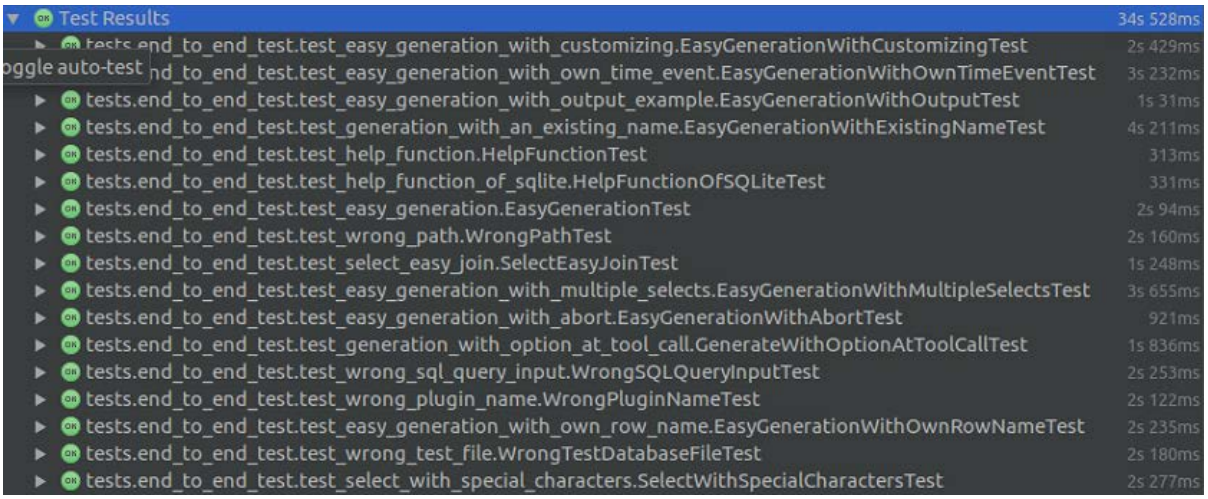
15.3.2.4 Ergebnisse vom End-to-End Test 4 – 22.05.2017

Die Version für alle Packages, welche installiert werden müssen wurde im requirements File sowie im setup.py Files angegeben.

	Passed	Failed	Bemerkung
Test 01	X		
Test 02	X		
Test 03	X		
Test 04	X		
Test 05	X		
Test 06	X		
Test 07	X		
Test 08	X		
Test 09	X		
Test 10	X		
Test 11	X		
Test 12	X		
Test 13	X		
Test 14	X		
Test 15	X		
Test 16	X		
Test 17	X		

15.3.2.5 Ergebnisse vom End-to-End Test 5 – 12.06.2017

Nach der letzten Änderung des Source Code mit Anpassung des Readme und Hinzufügung der Lizenz.



	Passed	Failed	Bemerkung
Test 01	X		
Test 02	X		
Test 03	X		
Test 04	X		
Test 05	X		
Test 06	X		
Test 07	X		
Test 08	X		
Test 09	X		
Test 10	X		
Test 11	X		
Test 12	X		
Test 13	X		
Test 14	X		
Test 15	X		
Test 16	X		
Test 17	X		

16 Separate Dokumente

- Aufgabenstellung HSR
- Aufgabenstellung Google
- Projektplan
- Anforderungsspezifikation
- Sitzungsprotokolle
- Einverständniserklärung
- Urheber- und Nutzungsrechte Vereinbarung
- Poster
- Source Code verfügbar unter: https://github.com/ClaudiaSaxer/PlasoScaffolder
- API Beschreibung verfügbar unter: https://plasoscaffolder.readthedocs.io