HAW HAMBURG

BACHELORTHESIS
Dustin Fritsch

Schreiben eines Messreportgenerators in der Sprache Python (Version 3.9)

FAKULTÄT TECHNIK UND INFORMATIK
Department Informations- und Elektrotechnik

Faculty of Computer Science and Engineering
Department of Information and Electrical Engineering

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Dustin Fritsch

Schreiben eines Messreportgenerators in der Sprache Python (Version 3.9)

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Elektro- und Informationstechnik*
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Benno Radt
Zweitgutachter: Prof. Dr. Bjoern Lange

Eingereicht am: 6. Dezember 2021

Dustin Fritsch

Thema der Arbeit

Schreiben eines Messreportgenerators in der Sprache Python (Version 3.9)

Stichworte

Messreportgenerator, Python

Kurzzusammenfassung

In dieser Bachelorarbeit geht es um das Schreiben eines Messreportgenerators. Dies geschieht in der Programmiersprache Python. Dieser Code ist auf die am DESY genutzen Teststände angepasst...

Dustin Fritsch

Title of Thesis

Writing of a measurement reportgenerator in the programming language Python (Version 3.9)

Keywords

Measurement reportgenerator, Python

Abstract

The topic of this thesis is to write a measurement reportgenerator. This is done using the programming language Python. The resulting code is adjusted to the test stands used at DESY ...

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungen

ADC Analog to Digital Converters.

DESY Deutsche Elektronen-Synchrotron.

FLASH Freie-Elektronen-Laser in Hamburg.

FMC FPGA Mezzanine Card.

FPGA Field Programmable Gate Array.

IDE Integrated Development Environment.

LGPL3 GNU Lesser General Puplic License Version 3.

PEP Python Enhanced Proposal.

PETRA III Positron-Elektron-Tandem-Ring-Anlage III.

PVM Python Virtual Machine.

TMCB Temperature Measurement and Control Board.

UML Unified Modeling Language.

USB Universal Serial Bus.

XFEL X-Ray Free-Electron Laser.

1 Einleitung

Das Deutsche Elektronen-Synchrotron (DESY) ist eines der weltweit führenden Beschleunigerzentren. Die Teilchenbeschleuniger am DESY ermöglichen es Forschern, Untersuchungen im Mikrokosmos zu tätigen.[Vgl. 4] Die Teilchenbeschleuniger erfüllen dabei teilweise unterschiedliche Funktionen. Der Speicherring Positron-Elektron-Tandem-Ring-Anlage III (PETRA III) erzeugt Röntgenstrahlen, die um ein bis zu 5000-faches feiner sind als ein menschliches Haar. Dies lässt zu, dass man damit sehr kleine Proben untersucht. Des Weiteren kann PETRA III sehr kurzwellige Röntgenstrahlung erzeugen. Ein tieferes Eindringen in die Materie, als mit anderem Röntgenlicht ist damit möglich. Insbesondere in der Erforschung von Materialien für die Karosserien, die in der Automobilindustrie und Flugzeugindustrie hergestellt werden, ist dies wichtig. Der Linearbeschleuniger Freie-Elektronen-Laser in Hamburg (FLASH) hingegen ermöglicht es, Laserpulse aus langwelligem Röntgenlicht zu erzeugen. Diese sind tausendmal stärker als Pulse von vergleichbaren konventionellen Lasern und deutlich kürzer als die eines Speicherringes. Das detaillierte Verfolgen und Untersuchen extrem schneller Prozesse, wie zum Beispiel dem Schmelzen von Metallen, ist damit möglich. Der Linearbeschleuniger European X-Ray Free-Electron Laser (XFEL) erzeugt ebenfalls Röntgenblitze, ähnlich dem FLASH. Allerdings sind die Röntgenblitze deutlich kürzer als eine Picosekunde und um ein Milliardenfaches heller als das Licht eines Speicherringes. Halbleiterphysiker, Molekularbiologen, Mediziner, Chemiker, Astrophysiker und Geologen profitieren von den Forschungsmöglichkeiten, die sich daraus ergeben. Jedes dieser Geräte ist einzigartig und gigantisch.[Vgl. 7] Der PETRA-III-Ring hat einen Umfang von 2,3 km, Flash eine Länge von 300 m und European XFEL eine Länge von 3,4 km.[Vgl. 6][Vgl. 5] In diesen Großgeräten sind viele Bauteile verbaut, die zum großen Teil selber entwickelt und produziert werden. Um diese Bauteile zu entwickeln, sind Teststände vonnöten um die Funktionsparameter festzustellen. Des Weiteren sind Teststände zur Qualitätssicherung für die Produktion von solchen Bauteilen vonnöten. Bei den zu testenden Bauteilen kann es sich zum Beispiel um Geräte zur Aufnahme von Messwerten oder auch Kabel handeln. Eine wichtige Funktion eines solchen Teststandes ist es, automatisiert Berichte zu generieren,

die einem Rückschlüsse bieten, ob und wie ein Testdurchlauf bestanden wurde. Thema dieser Bachelorarbeit ist es, eine Möglichkeit zur komfortablen Generierung solcher Berichte zu finden und zu programmieren.[Vgl. 7]

2 Material und Methoden

2.1 Material

Als Material für diese Arbeit wurden folgende Programme verwendet:

- Python (Version: 3.9)
- PyCharm community edition (Version: 2021.2.3)
- GitLab (Version 14.5)

2.1.1 Python

Historie

Python wurde Anfang der 1990er Jahre entwickelt. Im Januar 1994 wurde Python mit der Version 1.0 veröffentlicht. Die damals noch nicht vorhandene Garbage Collection wurde mit Veröffentlichung der Version 2.0 im Oktober 2000 ergänzt. Eine Garbage Collection übernimmt, einfach ausgedrückt, das Löschen nicht mehr benötigter Daten. In der Ende 2008 veröffentlichten Version 3.0 wurden einige der bestehenden Sprachelemente grundlegend verändert. Dies führte zum Verlust der Kompatibilität zu vorherigen Versionen.[Vgl. 3, S.5] Bis Anfang 2020 wurde die Version 2.7 parallel zu den 3.X Versionen weiter unterstützt. Die im Dezember 2021 aktuellste Version ist die Version 3.10. [Vgl. 8]

Warum Python?

Python ist aufgrund der Benutzung eines Interpreters sowie einer Garbage Collection weniger performant als Sprachen wie z.B. C++. Des Weiteren ist es eine abstrakte Sprache. Das macht es unmöglich bis ins kleinste Detail zu optimieren. Dafür bietet Python andere Vorzüge. Aufgrund der verwendeten virtuellen Maschine, die für allen relevanten Systeme vorhanden ist, ist Python plattformunabhängig. Des Weiteren benötigt man zum Programmieren in abstrakteren Sprachen wie Python weniger Entwicklungszeit für Software. Das liegt daran, dass man sich als Programmierer nicht mehr um jedes Detail kümmern muss. Außerdem gilt Python als einfach zu erlernende Sprache. Für die Teststände wird eine eigene Software benötigt. Diese muss auch nach Veröffentlichung weiter gepflegt werden. Dafür macht es Sinn, abstrakte Sprachen zu wählen, um die Kosten der dadurch entstehenden Arbeitszeiten gering zu halten. Des Weiteren ist sie quelloffen und auf allen benötigten Plattformen verfügbar. Die weite Verbreitung und Leistungsfähigkeit sind weitere gute Argumente.[Vgl. 3, S.5]

Interpreter statt Compiler

Python ist eine Programmiersprache die einen Interpreter verwendet. Im Gegensatz zu Programmiersprachen die auf einen Compiler setzen, wird hier aus dem Quellcode kein lauffähiger Maschinencode erzeugt. Anstelle dessen wird aus dem Quellcode ein sogenannter Bytecode generiert. Dieser ist nur über eine entsprechende virtuelle Maschine ausführbar. Im Falle von Python heißt diese Python Virtual Machine (PVM). Dies hat zwar eine längere Laufzeit zur Folge, bietet dafür aber andere Vorteile. Ein einfacherer Entwicklungsprozess, insbesondere beim Debuggen des Codes ist einer davon. Dies ist der Fall, weil ein Interpreter den Quellcode Zeile für Zeile übersetzt und so Fehler oft genauer angeben kann. Des Weiteren kann eine Integrated Development Environment (IDE) den Quellcode noch während der Erstellung überprüfen. Ein Compiler hingegen betrachtet den Quellcode erst bei der Kompilierung. Dies führt oft dazu, dass Fehler zwar erkannt werden, aber die ausgegebenen Fehlerhinweise nicht unbedingt hilfreich sind.[Vgl. 3, S.7]

2.2 Methoden

2.2.1 PEP 8

Python ist eine Programmiersprache mit vielen Freiheiten um einem Quellcode einen eigenen Stil zu geben. Dies ist allerdings in den meisten Fällen unerwünscht. Um eine gute Lesbarkeit von Quellcode zu gewährleisten, ergibt es also Sinn, einen sogenannten Style Guide zu verwenden. Style Guides sind prinzipiell eine Art Formatierungsvorgabe für Quellcode. Im Falle von Python ist PEP 8 eine gute Wahl. Die verwendete IDE PyCharm community edition hat zu diesem Zwecke eine automatische integrierte Überprüfung. Das macht das Einhalten des Style Guides trivial.[Vgl. 3, S.26]

3 Beginn der Arbeiten

Zu Beginn der Arbeiten ist es wichtig, sich mit dem Anwendungsgebiet und dem Stand der Technik vertraut zu machen. In diesem Kapitel werden die Teststände beschrieben und die Ergebnisse einer Marktrecherche festgehalten. Bei der Marktrecherche sind auch die bisher getesteten Lösungen berücksichtigt.

3.1 Die Teststände

3.1.1 PZ16M-Teststand

Das PZ16M ist eine 19-Zoll-Box mit 16 Piezotreibern, also Elektronik, die Piezoelemente mit Hochspannung versorgen kann, so dass diese sich ausdehnen oder zusammenziehen. Das wird in Beschleunigern z.B. dazu benutzt, die Resonanzfrequenz der Hohlraumresonatoren, mit denen die Teilchen beschleunigt werden, einzustellen. Man drückt mit dem Piezoelement gegen die Wand des Hohlraumresonators, so dass sich die Resonanzfrequenz verstellt. Der Teststand ist in der Lage, die Sollfunktionalität einer Box zu untersuchen (z.B. ob alle Kanäle richtig funktionieren). Das dient sowohl der Qualitätssicherung, bevor die Geräte verbaut werden, als auch der Fehlersuche, wenn ein defektes Gerät aus dem Beschleuniger zurückkommt.

3.1.2 TMCB-Teststand

Das Temperature Measurement and Control Board (TMCB) ist ein Multifunktions-Board mit einem Field Programmable Gate Array (FPGA), der die Kontrolle über das Board hat und verschiedene Anwenderfirmware laufen lassen kann. Das Board ist mit mehreren Analog-Digital- und Digital-Analog-Konvertern ausgestattet, um Analogsignale messen und erzeugen zu können, so dass sich Regelkreise damit aufbauen lassen. Einige der Analog to Digital Converters (ADC) sind extra für Temperaturmessungen ausgelegt, daher

der Name. Des Weiteren hat es mehrere digitale Schnittstellen (z.B. I^2C-Bus), an die sich weitere Sensoren anbinden lassen. Außerdem verfügt es über einige General Purpose Input Output Pins. Des Weiteren hat das Board einen sogenannten FPGA Mezzanine Card (FMC)-Steckplatz, auf dem sich kleine Tochterkarten anschließen lassen, die ebenfalls vom FPGA bedient werden. Das TMCB hat eine Debug-Schnittstelle (seriell über einen Universal Serial Bus (USB)) und ist voll hotplug-fähig. Zur Kommunikation sind Ethernet (RJ45) und Glasfaseranschlüsse möglich. All diese vielen verschiedenen Hardwarekomponenten werden in dem Teststand auf Funktionalität geprüft. Der Teststand dient der Qualitätssicherung der Massenproduktion.

3.1.3 Teststand für Struck SIS8300L

Struck SIS8300L ist ein FPGA-basiertes ADC-Board. Die Aufgabe dieses Teststandes ist eine Kombination aus Qualitätssicherung und Fehlersuche.

3.1.4 Climate Lab

Dieser Teststand ist ein Forschungsstteststand zur Charakterisierung von Hochfrequenzkabeln. Die Kabel werden in einer Klimakammer betrieben, an der Temperatur und Luftfeuchtigkeit eingestellt werden können. Der Teststand ist in der Lage, verschiedene Temperatur- und Luftfeuchigkeitsprofile zu fahren und Messungen an den Hochfrequenzkabeln zu machen, um deren Reaktion auf die Temperatur- und Luftfeuchteänderungen zu untersuchen.

3.2 Stand der Technik

3.2.1 Marktrecherche

Bei der Marktrecherche wurde mit hilfe der Schlagwörter „Python“, „Reportgenerator“ und „PDF“ gesucht. Im Zuge dieser Marktrecherche mit den Suchmaschinen der Webseiten Google und Sciencedirect, sind folgende potenzielle Lösungen mit wenigen Vorbehalten gefunden worden:

- Pandas + HTML
- ReportLab
- LaTeX

Pandas + HTML

Pandas selbst ist ein Werkzeug zum Analysieren und Manipulieren von Daten in Python. Es bietet unter anderem auch die Möglichkeit, Daten in Tabellen in eine HTML-Datei zu überführen. Das Problem bei dieser Methode ist allerdings, dass für alles Weitere das HTML-Skript entsprechend angepasst werden müsste. Das heißt, es müsste Code für ein Interface geschrieben werden, der mit den gegebenen Daten eine HTML-Vorlage erstellt und füllt.
„Reportlab with Platypus could not process vector graphics and had issues with placing graphics. The usage was not as nice as announced (rather cryptic in some places), but could produce tables with the required highlighting. Due to the lack of graphics inclusion we had to fall back to creating individual pages as pdf and combine them with an additional tool. (State of 2014)“ [2]

ReportLab

ReportLab wurde bereits an einem Teststand getestet. Aufgrund der als umständlich empfundenen Bedienung und nicht zufriedenstellenden Formatierungsmöglichkeiten wurde es als unzureichend eingestuft.[Vgl. 2]

LaTeX

Auch LaTeX wurde bereits an einem Teststand verwendet, dort allerdings durch Generieren von „Schnipseln“ im Testbenchtool. Dieses Vorgehen hat zwar die besten Ergebnisse geliefert, war allerdings umständlich und schwierig zu debuggen. „Directly generating Latex snippets from the code had stability problems as open and closing braces / begin-end statements were produced at completely different places and could cause broken latex documents. The situation was difficult to debug. The produced output on the other hand was superior to all other solutions. “ [2]

3.2.2 Ergebnisse der Marktrecherche

Unter Abwägung der Vor- und Nachteile der vielversprechendsten Lösungsansätze wurde LaTeX als Basis gewählt. Die Idee ist, eine Bibliothek zu schaffen, die es ermöglicht in einem Python-Skript bequem die Daten in eine zusammenhängende LaTeX-Datei zu platzieren.[Vgl. 2]

4 Ergebnisse

4.1 Die Bibliothek „testreportgen.py“

4.1.1 Shebang, Imports und Konstanten

```
1 #!/usr/bin/env python3
2 """testreportgen.py: This libery is a tool to generate reports.
3 Requires Python 3.4 or later"""
4
5 __version__ = '0.5'
6 __author__ = 'Dustin Fritsch'
7
8 import abc  # abstract classes
9 import numpy as np
10 from enum import Enum
11
12 OK = 'Ok'
13 SKIPPED = 'Skipped'
14 FAILED = 'Failed'
15 Colors = Enum('Colors', 'RED GREEN BLUE CYAN MAGENTA YELLOW BLACK GRAY WHITE DARKGRAY LIGHTGRAY BROWN LIME OLIVE '
16                         'ORANGE PINK PURPLE TEAL VIOLET')
17 OK_COLOR = Colors.GREEN
18 SKIPPED_COLOR = Colors.ORANGE
19 FAILED_COLOR = Colors.RED
20
21
```

Abbildung 4.1: Shebang, Imports und Konstanten

Am Anfang des Codes wird der Zweck des Codes genannt und darauf hingewiesen, dass mindestens Python 3.4 benötigt wird. Dies liegt daran, dass erst in Python 3.4 Enumeration eingeführt worden ist. [Vgl. 1] Außerdem werden die Versionsnummer und Name des Autors genannt. Für den Code werden die Bibliotheken „abc“, „numpy“ und „enum“ benötigt. Die Bibliothek „abc“ wird zur Definierung einer Metaklasse benötigt. Numpy ist eine Bibliothek die zum Erstellen und Verarbeiten von Arrays benötigt wird. Die Klasse „Enum“ aus der Blibliothek „enum“ dient der Erstellung von Enumeration-Objekten. Die Konstanten „OK“, „SKIPPED“ und „FAILED“ definieren Strings, die im Bericht für die jeweiligen Ereignisse stehen sollen. Die um den Zusatz „_COLOR“ ergänzten Konstanten

definieren die Farben, die die jeweiligen Texte haben sollen, wenn sie in den Übersichtstabellen vorkommen. Das Enumeration-Objekt „Color“ definiert alle in LaTeX möglichen Farben und bietet einen komfortablen Umgang mit diesen.

4.1.2 Die Klassen

ReportBuildingBlock

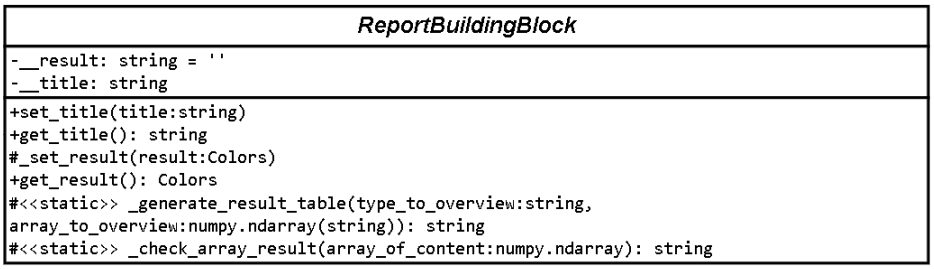

Abbildung 4.2: UML Diagramm der Klasse „ReportBuildingBlock“

In dieser Klasse sind einige allgemein notwendige Methoden und Attribute deklariert. Da es sich hierbei um eine Metaklasse handelt, ist es nicht vorgesehen, Instanzen davon zu generieren. Alle weiteren Klassen sind von dieser Klasse abgeleitet und erben somit alle Attribute und Methoden dieser Klasse.

```
22 class ReportBuildingBlock(metaclass=abc.ABCMeta):
23     """
24     Metaclass to define some functions needed in the other classes
25     Keyword arguments:
26     result -- Value to define whether a block is Failed, OK or Skipped.
27     title -- Title to name a certain block of the document.
28     """
29
30     def __init__(self, title):
31         assert isinstance(title, str)
32         self.__title = title
33         self.__result = ''
34
35     def get_result(self):
36         return self.__result
37
38     def _set_result(self, result):
39         self.__result = result
40
41     def get_title(self):
42         return self.__title
43
44     def set_title(self, title):
45         self.__title = title
46
```

Abbildung 4.3: Konstruktor, Getter und Setter der Klasse „ReportBuildingBlock“

Die Metaklasse „ReportBuildingBlock“ definiert die Attribute „__result“ und „__title“. Dabei handelt es sich um private Attribute, die in jedem Teil eines Berichts benötigt werden. Diese werden mit entsprechenden Gettern und Settern versehen. Der Setter des Attributes „__result“ wird dabei dem Nutzer unzugänglich gemacht und ist nur innerhalb der Klassen des Reportgenerators aufrufbar. Dies verhindert ein versehentliches Manipulieren dieses Attributs.

```
47      @staticmethod
48      def _generate_result_table(type_to_overview, array_to_overview):
49          """
50          Function to generate a quick table to overview results of certain parts of the document.
51          temp is just a temporary string to build the requested code in.
52          :param type_to_overview: Either sections, subsections or tests can be overviewed
53          :param array_to_overview: Array containing the sections, subsections or tests to overview
54          :return: String that contains the code to generate the desired table
55          """
56          temporary_latex_code = '\\begin{table}[h]\n\\centering\n\\begin{tabular}{|l|l|}\n\\hline\n'
57          temporary_latex_code += type_to_overview + ' & Result\\\\ \\hline\n'
58          for entry in array_to_overview:
59              temporary_latex_code += entry.get_title() + '&\\textcolor{'
60              if entry.get_result() == OK:
61                  temporary_latex_code += str.lower(OK_COLOR.name) + '}{Ok}'
62              elif entry.get_result() == SKIPPED:
63                  temporary_latex_code += str.lower(SKIPPED_COLOR.name) + '}{Skipped}'
64              elif entry.get_result() == FAILED:
65                  temporary_latex_code += str.lower(FAILED_COLOR.name) + '}{Failed}'
66              temporary_latex_code += '\\\\ \\hline\n'
67          temporary_latex_code += '\\end{tabular}\n\\end{table}\n'
68          return temporary_latex_code
69
```

Abbildung 4.4: Methode „_generate_result_table“ der Klasse „ReportBuildingBlock“

Die statische Methode „_generate_result_table“ dient der Erstellung von Übersichten über die Ergebnisse einzelner Abschnitte, zum Beispiel welche Tests in einem Kapitel bestanden wurden. Der Parameter „type_to_view“ gibt den Namen des Typs, über den eine Übersicht generiert werden soll, als String an. Das Array, das die benötigten Elemente des Typs enthält, wird mit dem Parameter „array_to_overview“ übergeben. Diese müssen in Form eines numpy.ndarray vorliegen. Dessen Elemente müssen Objekten entsprechen, die einer von „ReportBuildingBlock“ abgeleiteten Klasse angehören. Die Funktion erstellt aus den Ergebnissen den Code, der für die entsprechende Tabelle in LaTeX benötigt wird. Dieser Code wird in Form eines Strings als Rückgabewert verwendet.

```
70      @staticmethod
71      def _check_array_result(array_of_content):
72          """
73          Checks the results of an array to build up a summarized result
74          :return: Summarized result
75          """
76          temporary_result = OK
77          for entry in array_of_content:
78              if entry.get_result() != OK:
79                  temporary_result = SKIPPED
80          if temporary_result == SKIPPED:
81              for entry in array_of_content:
82                  if entry.get_result() != SKIPPED:
83                      temporary_result = FAILED
84          return temporary_result
85
86
```

Abbildung 4.5: Methode „_check_array_result“ der Klasse „ReportBuildingBlock“

Die statische Methode „_check_array_result“ wird zur Auswertung benötigt. Ihr wird ein numyp.ndarray übergeben. Dessen Elemente müssen Objekten entsprechen, die einer von „ReportBuildingBlock“ abgeleiteten Klasse angehören. Die Ergebnisse der Elemente werden zu einem Gesamtergebnis zusammengefasst. Wurden alle Elemente bestanden, so ist das Gesamtergebnis „OK“. Wurden alle Elemente übersprungen, so ist das Gesamtergebnis „SKIPPED“. In jedem anderen Fall ist das Gesamtergebnis „Failed“. Dieses Gesamtergebnis liegt in Form eines Strings vor und dient als Rückgabewert.

Document

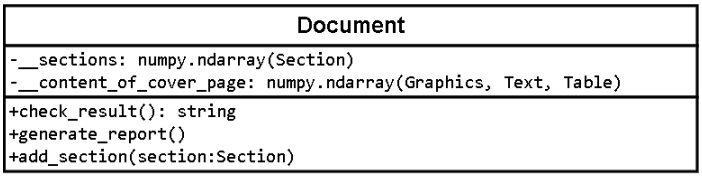

Abbildung 4.6: UML Diagramm der Klasse „Document“

Die Klasse „Document“ dient als eine Art Grundgerüst, dem die einzelnen Kapitel angehängt werden.

```
87 class Document(ReportBuildingBlock):
88     __document_class = '[a4paper,10pt]{article}'
89     __packages = ['[latin1]{inputenc}', '[T1]{fontenc}', '[english]{babel}', '{color}', '{pgf}',
90                   '{transparent}', '{float}']
91
92     def __init__(self, title, sections, content_of_cover_page):
93         """
94         :param title: Title string of the document
95         :param sections: Array of sections
96         :param content_of_cover_page: Array containing text, table's or graphics for the cover page
97         """
98         ReportBuildingBlock.__init__(self, title)
99         assert isinstance(sections, np.ndarray)
100        for entry in sections:
101            assert isinstance(entry, Section)
102        assert isinstance(content_of_cover_page, np.ndarray)
103        for entry in content_of_cover_page:
104            assert isinstance(entry, (Graphics, Text, Table))
105        self.__sections = sections
106        self.__content_of_cover_page = content_of_cover_page
107
```

Abbildung 4.7: Konstruktor der Klasse „Document“

In der Klasse „Document“ sind zwei statische Attribute definiert. „__document_class“ und „__packages“ enthalten Informationen über Pakete und Formatierungsvorschriften für die spätere LaTeX-Datei. Es werden außerdem die numpy.ndarray „__sections“ und „__content_of_cover_page“ definiert. „__sections“ enthält dabei Objekte der Klasse „Section“ und „__content_of_cover_page“ kann Objekte der Klassen „Graphic“, „Text“ und/oder „Table“ enthalten.

```
108     def check_result(self):
109         """
110         Checks the results of the sections
111         :return: Summarized result
112         """
113         for entry in self.__sections:
114             entry.check_result()
115         temporary_result = self._check_array_result(self.__sections)
116         self._set_result(temporary_result)
117         return temporary_result
118
```

Abbildung 4.8: Methode „check_result“ der Klasse „Document“

Die Methode „check_result“ lässt alle angehängten Kapitel in „__sections“ sich selbst auf ihre Ergebnisse überprüfen. Anschließend ermittelt sie aus den so ermittelten Ergebnissen ein Gesamtergebnis. Dieses dient in Form eines Strings als Rückgabewert.

```
119     def generate_report(self):
120         """
121         :return: Writes a latex file that contains the code to generate the report.
122         """
123         self.check_result()
124         # Generate the beginning of the code containing user packages
125         temporary_latex_code = ''
126         temporary_latex_code += '\\documentclass' + self.__document_class + '\n'
127         for package in self.__packages:
128             temporary_latex_code += '\\usepackage' + package + '\n'
129         temporary_latex_code += '\\begin{document}\n'
130 
131         # Generate cover page
132         temporary_latex_code += '\\title{' + self.get_title() + '}\n'
133         temporary_latex_code += '\\setlength{\\parindent}{0em}\n \\maketitle\n'
134         for entry in self.__content_of_cover_page:
135             temporary_latex_code += entry.convert_to_tex()
136 
137         # Generate table of contents page
138         temporary_latex_code += '\\newpage\n \\tableofcontents\n \\newpage'
139 
140         # Generate table of sections result page
141         temporary_latex_code += self._generate_result_table('Section', self.__sections)
142         temporary_latex_code += '\\newpage'
143 
144         # Generate code for the sections
145         for entry in self.__sections:
146             temporary_latex_code += entry.convert_to_tex()
147 
148         # Finish the code and write it into an latex file
149         temporary_latex_code += '\\end{document}'
150         file = open("latex.tex", "w")
151         file.write(temporary_latex_code)
152 
```

Abbildung 4.9: Methode „generate_report“ der Klasse „Document“

Die Methode „generate_report“ erstellt eine LaTeX-Datei. Diese enthält den Code, der zur Erstellung eines entsprechenden Berichts mittels eines Programms wie „pdflatex“ benötigt wird. Dazu wird zunächst die Methode „check_result“ des Dokuments aufgerufen. Nach dieser Kontrolle der Ergebnisse wird der LaTeX-Code generiert und in einem letzten Schritt in eine LaTeX-Datei geschrieben. Dieser enthält unter anderem auch eine Übersicht darüber, welche Ergebnisse die einzelnen Kapitel haben.

```
153     def add_section(self, section):
154         """
155         :param section: Section to add
156         :return: Adds section into the self.__sections array
157         """
158         assert isinstance(section, Section)
159         self.__sections = np.append(self.__sections, [section])
160 
161 
```

Abbildung 4.10: Add Methode der Klasse „Document“

Die Methode „add_section“ ermöglicht das nachträgliche Anhängen von weiteren Kapiteln. Der Funktion wird im Übergabeparameter „section“ ein Objekt der Klasse „Section“ übergeben. Dieses wird dann an das Array „__sections“ hinten angehängt.

Section

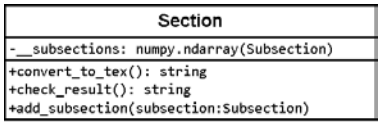

Abbildung 4.11: UML Diagramm der Klasse „Section“

Die Klasse „Section“ dient als Gerüst für die einzelnen Kapitel des Berichts. Objekten dieser Klasse können einzelne Unterkapitel angehängt werden.

```
162 class Section(ReportBuildingBlock):
163
164     def __init__(self, title, subsections):
165         """
166         :param title: Title string of the section
167         :param subsections: Array of subsections
168         """
169         ReportBuildingBlock.__init__(self, title)
170         assert isinstance(subsections, np.ndarray)
171         for entry in subsections:
172             assert isinstance(entry, Subsection)
173         self.__subsections = subsections
174
```

Abbildung 4.12: Konstruktor der Klasse „Section“

In der Klasse „Section“ wird das Attribut „__subsections“ definiert. Dies enthält ein numpy.ndarray. In diesem Array sind die Unterkapitel in Form von Objekten der Klasse „Subsection“ enthalten.

```
175     def convert_to_tex(self):
176         """
177         Generates latex code of the current section
178         :return: Latex code string needed to build the section
179         """
180         # Generate section title
181         temporary_latex_code = ''
182         temporary_latex_code += '\\section{' + self.get_title() + '}\n'
183
184         # Generate table of subsections result
185         temporary_latex_code += self._generate_result_table('Subsection', self.__subsections)
186
187         # Generate code for the subsections
188         for entry in self.__subsections:
189             temporary_latex_code += entry.convert_to_tex()
190         return temporary_latex_code
191
```

Abbildung 4.13: Methode „convert_to_tex“ der Klasse „Section“

Die Methode „convert_to_tex“ generiert den LaTeX-Code, der zur Einbindung des jeweiligen Kapitels nötig ist. Dieser dient in Form eines Strings als Rückgabewert. In dem Code ist eine Übersicht der Ergebnisse der einzelnen angehängten Unterkapitel sowie der Code eben dieser Unterkapitel enthalten.

```
192     def check_result(self):
193         """
194         Checks the results of the subsections
195         :return: Summarized result
196         """
197         for entry in self.__subsections:
198             entry.check_result()
199         temporary_result = self._check_array_result(self.__subsections)
200         self._set_result(temporary_result)
201         return temporary_result
202
```

Abbildung 4.14: Methode „check_result“ der Klasse „Section“

Die Methode „check_result“ lässt alle angehängten Unterkapitel in „__subsections“ sich selbst auf ihre Ergebnisse überprüfen. Anschließend ermittelt es aus den so ermittelten Ergebnissen ein Gesamtergebnis. Dieses dient in Form eines Strings als Rückgabewert.

```
203     def add_subsection(self, subsection):
204         """
205         :param subsection: Subsection to add
206         :return: Adds subsection to the self.__subsections array
207         """
208         assert isinstance(subsection, Subsection)
209         self.__subsections = np.append(self.__subsections, [subsection])
210
211
```

Abbildung 4.15: Add Methode Klasse „Section“

Die Methode „add_subsection“ ermöglicht das nachträgliche Anhängen von weiteren Unterkapiteln. Der Funktion wird im Übergabeparameter „subsection“ ein Objekt der Klasse „Subsection“ übergeben. Dieses wird dann an das Array „__subsections“ hinten angehängt.

Subsection

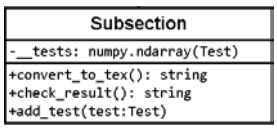

Abbildung 4.16: UML Diagramm der Klasse „Subsection“

Die Klasse „Subsection“ dient als Gerüst für die einzelnen Unterkapitel des Berichts. Objekten dieser Klasse können einzelne Tests angehängt werden.

```
212 class Subsection(ReportBuildingBlock):
213
214     def __init__(self, title, tests):
215         """
216         :param title: Title string of the subsection
217         :param tests: Array of tests
218         """
219         ReportBuildingBlock.__init__(self, title)
220         assert isinstance(tests, np.ndarray)
221         for entry in tests:
222             assert isinstance(entry, Test)
223         self.__tests = tests
224
```

Abbildung 4.17: Konstruktor der Klasse „Subsection“

In der Klasse „Subection“ wird das Attribut „__tests“ definiert. Dies enthält ein numpy.ndarray. In diesem Array sind die Tests in Form von Objekten der Klasse „Test“ enthalten.

```
225     def convert_to_tex(self):
226         """
227         Generates latex code of the current subsection
228         :return: Latex code string needed to build the subsection
229         """
230         # Generate section title
231         temporary_latex_code = ''
232         temporary_latex_code += '\\subsection{' + self.get_title() + '}\n'
233
234         # Generate table of test result
235         temporary_latex_code += self._generate_result_table('Test', self.__tests)
236         for entry in self.__tests:
237             temporary_latex_code += entry.convert_to_tex()
238         return temporary_latex_code
239
```

Abbildung 4.18: Methode „convert_to_tex“ der Klasse „Subsection“

Die Methode „convert_to_tex“ generiert den LaTeX-Code, der zur Einbindung des jeweiligen Unterkapitels nötig ist. Dieser dient in Form eines Strings als Rückgabewert. In

dem Code ist eine Übersicht der Ergebnisse der einzelnen angehängten Tests sowie der Code eben dieser Tests enthalten.

```
240     def check_result(self):
241         """
242         Checks the results of the tests
243         :return: Summarized result
244         """
245         temporary_result = self._check_array_result(self.__tests)
246         self._set_result(temporary_result)
247         return temporary_result
248
```

Abbildung 4.19: Methode „check_result“ der Klasse „Subsection“

Die Methode „check_result“ lässt alle angehängten Tests in „__tests“ sich selbst auf ihre Ergebnisse überprüfen. Anschließend ermittelt es aus den so ermittelten Ergebnissen ein Gesamtergebnis. Dieses dient in Form eines Strings als Rückgabewert.

```
249     def add_test(self, test):
250         """
251         :param test: Test to add
252         :return: Adds test to self.__tests array
253         """
254         assert isinstance(test, Test)
255         self.__tests = np.append(self.__tests, [test])
256
257
```

Abbildung 4.20: Add Methode der Klasse „Subsection“

Die Methode „add_test“ ermöglicht das nachträgliche Anhängen von weiteren Unterkapiteln. Der Funktion wird im Übergabeparameter „test“ ein Objekt der Klasse „Test“ übergeben. Dieses wird dann an das Array „__tests“ hinten angehängt.

Test

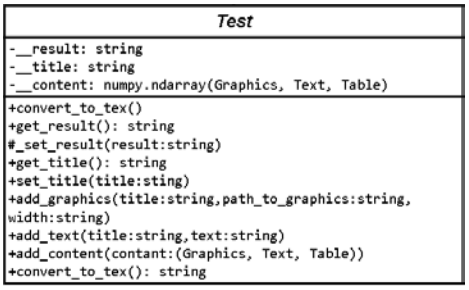

Abbildung 4.21: UML Diagramm der Klasse „Test“

Die Klasse „Test“ dient als Gerüst für die einzelnen Tests des Berichts. Objekten dieser Klasse können einzelne Bausteine angehängt werden. Die Bausteine können Texte, Grafiken oder auch Tabellen sein.

```
258 class Test(ReportBuildingBlock):
259
260     def __init__(self, result, title, content):
261         """
262         :param result: String to define whether this test is Failed, OK or Skipped.
263         :param title: Title string of the Test
264         :param content: Array containing text, graphics or table objects
265         """
266         assert isinstance(result, str)
267         assert isinstance(title, str)
268         assert result == FAILED or result == SKIPPED or result == OK
269         assert isinstance(content, np.ndarray)
270         for entry in content:
271             assert isinstance(entry, (Graphics, Text, Table))
272         self.__title = title
273         self.__result = result
274         self.__content = content
275
```

Abbildung 4.22: Konstruktor der Klasse „Test“

In der Klasse „Test“ wird das Attribut „__content“ definiert. Dies enthält ein numpy.ndarray. In diesem Array sind die Bausteine in Form von Objekten der Klassen „Grapics“, „Text“ und „Table“ enthalten. Des Weiteren werden die Attribute „__result“ und „__title“ in dieser Klasse überschrieben. Das ist notwendig, um eine Übergabe des Wertes „result“ im Konstruktor möglich zu machen. So kann der Nutzer bei Erstellen eines Objektes der Klasse Test angeben, welches Ergebnis dieser hatte.

```
276     def get_result(self):
277         return self.__result
278 
279     def _set_result(self, result):
280         self.__result = result
281 
282     def get_title(self):
283         return self.__title
284 
285     def set_title(self, title):
286         self.__title = title
287 
```

Abbildung 4.23: Getter und Setter der Klasse „Test“

Aufgrund der neuen Definition der Attribute „__result“ und „__title“ ist eine erneute Definition der entsprechenden Setter und Getter notwendig.

```
288     def add_graphics(self, title, path_to_graphics, width):
289         """
290         :param title: Caption string of the graphics
291         :param path_to_graphics: Location of the graphics (/path/to/graphics.ong)
292         :param width: Number in string format to define the size of the graphics
293         :return: Adds graphics to the test
294         """
295         self.__content = np.append(self.__content, [Graphics(title, path_to_graphics, width)])
296 
297     def add_text(self, title, text):
298         """
299         :param title: Caption string of the text
300         :param text: String containing actual text
301         :return: Adds text to the test
302         """
303         self.__content = np.append(self.__content, [Text(title, text)])
304 
305     def add_content(self, content):
306         """
307         :param content: Graphics, text or table to add
308         :return: Adds a graphics, text or table object to the test
309         """
310         assert isinstance(content, (Graphics, Text, Table))
311         self.__content = np.append(self.__content, [content])
312 
```

Abbildung 4.24: Add Methoden der Klasse „Test“

Die Methoden „add_graphics“, „add_text“ und „add_content“ ermöglichen das nachträgliche Anhängen von weiteren Bausteinen. Der Methode „add_graphics“ und „add_text“ werden jeweils die nötigen Werte für die Konstruktoren der entsprechenden Objekte „Grapics“ und „Text“ übergeben, um solche Objekte zu erzeugen. Der Methode „add_content“ wird über den Parameter „content“ ein Objekt übergeben. Dieses kann der Klasse „Graphics“, „Text“ oder „Table“ angehören. All diese Funktionen hängen die jeweiligen Objekte an das Array „__content“ des Tests hinten an.

```
313     def convert_to_tex(self):
314         """
315         Generates latex code of the current test
316         :return: Latex code string needed to build the test
317         """
318         temporary_latex_code = '\\textbf{' + self.get_title() + '}\n\\newline\n'
319         for entry in self.__content:
320             temporary_latex_code += entry.convert_to_tex()
321         return temporary_latex_code
322
323
```

Abbildung 4.25: Methode „convert_to_tex“ der Klasse „Test“

Die Methode „convert_to_tex“ generiert den LaTeX-Code, der zur Einbindung des jeweiligen Tests nötig ist. Dieser dient in Form eines Strings als Rückgabewert. In dem Code ist der Code zur Erstellung der angehängten Bausteine enthalten.

Graphics

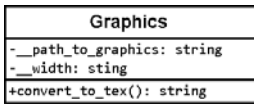

Abbildung 4.26: UML Diagramm der Klasse „Graphics“

Die Klasse „Graphics“ dient als Baustein für die einzelnen Tests des Berichts. Objekte dieser Klasse können einzelne Tests angehängt werden. Sie ermöglichen das Einbinden von Raster- und Vektorgrafiken. Die Übergabe eines Titels ist optional. Wird eine leerer String übergeben, so wird auf einen Titel für das Bild verzichtet.

```
324 class Graphics(ReportBuildingBlock):
325
326     def __init__(self, title, path_to_graphics, width):
327         """
328         :param title: Caption string of the graphics
329         :param path_to_graphics: path_to_graphics: Location of the graphics (/path/to/graphics.ong)
330         :param width: Number in string format to define the size of the graphics
331         """
332         ReportBuildingBlock.__init__(self, title)
333         assert isinstance(path_to_graphics, str)
334         assert isinstance(width, str)
335         self.__path_to_graphics = path_to_graphics
336         self.__width = width
337
```

Abbildung 4.27: Konstruktor der Klasse „Graphics“

In der Klasse „Graphics“ werden die Attribute „__path_to_graphics“ und „__width“ definiert. Diese enthalten Strings. „__path_to_graphics“ beschreibt dabei den Ordnerpfad zu der Grafik die angehängt werden soll. Das Attribut „__width“ hingegen enthält einen Zahlenwert als String, der die Größe der Grafik im fertigen Bericht beschreibt.

```
338     def convert_to_tex(self):
339         """
340         Generates latex code of the current graphics
341         :return: Latex code string needed to build the graphics
342         """
343         temporary_latex_code = '\\begin{figure}[H] \n\\centering \n'
344         # Vector graphics in pgf format require a different approach
345         if (self.__path_to_graphics[-3] == 'p') & (self.__path_to_graphics[-2] == 'g') & \
346                 (self.__path_to_graphics[-1] == 'f'):
347             temporary_latex_code += '\\def\\svgwidth{' + self.__width + 'pt} \n\\input{' + \
348                     self.__path_to_graphics + '}\n'
349         # then other graphics
350         else:
351             temporary_latex_code += '\\includegraphics[width=' + self.__width + 'pt]{' + \
352                     self.__path_to_graphics + '}\n'
353         # Only add caption, if it exists
354         if self.get_title() != '':
355             temporary_latex_code += '\\caption{' + self.get_title() + '}\n'
356         temporary_latex_code += '\\end{figure}\n'
357         return temporary_latex_code
358
359
```

Abbildung 4.28: Methode „convert_to_tex“ der Klasse „Graphics“

Die Methode „convert_to_tex“ generiert den LaTeX-Code, der zur Einbindung der jeweiligen Grafik nötig ist. Dieser dient in Form eines Strings als Rückgabewert.

Text

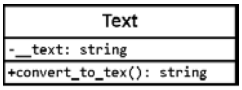

Abbildung 4.29: UML Diagramm der Klasse „Text“

Die Klasse „Text“ dient als Baustein für die einzelnen Tests des Berichts. Objekte dieser Klasse können einzelnen Tests angehängt werden. Sie ermöglichen das Einbinden von Texten. Die Übergabe eines Titels und des Textes ist optional. Wird ein leerer String übergeben, so wird auf den jeweiligen Text verzichtet.

```
360 class Text(ReportBuildingBlock):
361
362     def __init__(self, title, text):
363         """
364         :param title: Caption string of the text
365         :param text: String containing actual text
366         """
367         ReportBuildingBlock.__init__(self, title)
368         assert isinstance(text, str)
369         self.__text = text
370
371     def convert_to_tex(self):
372         """
373         Generates latex code of the current text
374         :return: Latex code string needed to build the text
375         """
376         temporary_latex_code = ''
377         # Only add caption, if it exists
378         if self.get_title() != '':
379             temporary_latex_code += '\\textbf{' + self.get_title() + '}\n\\newline\n'
380         # Only add text, if it exists
381         if self.__text != '':
382             temporary_latex_code += '\\textmd{' + self.__text + '}\n\\newline\n'
383         return temporary_latex_code
384
385
```

Abbildung 4.30: Die Klasse „Text“

In der Klasse „Text“ wird das Attribut „__text“ definiert. Dies enthält einen String zur Einbindung in den Bericht. Der String in „__title“ wird fettgedruckt in den Bericht eingebunden. Der String in „__text“ wird normalgedruckt in den Bericht eingebunden. Die Methode „convert_to_tex“ generiert den LaTeX-Code, der zur Einbindung des jeweiligen Textes nötig ist. Dieser dient in Form eines Strings als Rückgabewert.

Table

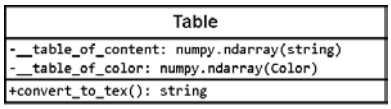

Abbildung 4.31: UML Diagramm der Klasse „Table“

Die Klasse „Table“ dient als Baustein für die einzelnen Tests des Berichts. Objekte dieser Klasse können einzelnen Tests angehängt werden. Sie ermöglichen das Einbinden von Tabellen. Da dies eine von „ReportBuildingBlock“ abgeleitete Klasse ist, erbt sie dessen Attribute und Methoden. Die Übergabe eines Titels ist optional. Wird ein leerer String übergeben, so wird auf einen Titel verzichtet.

```
386 class Table(ReportBuildingBlock):
387
388     def __init__(self, title, table_of_content, table_of_color):
389         """
390         :param title: Caption string of the table
391         :param table_of_content: 2D Array containing the strings to print into the table.
392         :param table_of_color: 2D Array containing the strings to color the table.
393         """
394         ReportBuildingBlock.__init__(self, title)
395         assert isinstance(table_of_content, np.ndarray)
396         assert isinstance(table_of_color, np.ndarray)
397         for list_of_color in table_of_color:
398             for color in list_of_color:
399                 assert isinstance(color, Colors)
400         assert table_of_content.shape == table_of_color.shape
401         self.__table_of_content = table_of_content
402         self.__table_of_color = table_of_color
403
```

Abbildung 4.32: Konstruktor der Klasse „Table“

In der Klasse „Table“ werden die Attribute „__table_of_content“ und „__table_of_color“ definiert. Diese enthalten jeweils ein zweidimensionales numpy.ndarray. Die beiden Arrays müssen dieselbe Form haben. „__table_of_content“ beinhaltet Strings, die den Inhalt der jeweiligen Zellen der Tabelle beschreiben. Das Array „__table_of_color“ hingegen beinhaltet Werte der Enumeration „Colors“. Es beschreibt die Farbe der Texte der jeweiligen Zellen.

```
404     def convert_to_tex(self):
405         """
406         Generates latex code of the current table
407         :return: Latex code string needed to build the table
408         """
409         temporary_latex_code = '\\begin{table}[h]\n\\centering\n'
410         # Only add caption, if it exists
411         if self.get_title() != '':
412             temporary_latex_code += '\\caption{' + self.get_title() + '}\n'
413         # add table format
414         temporary_latex_code += '\\begin{tabular}{|' + self.__table_of_content.shape[1] * 'l|' + '}\n\\hline\n'
415         # create flipped version of table. These are required for the following loop to write the corresponding code
416         table_of_content = np.rot90(np.fliplr(self.__table_of_content))
417         table_of_color = np.rot90(np.fliplr(self.__table_of_color))
418         for y in range(len(self.__table_of_content)):
419             for x in range(len(self.__table_of_content[y])):
420                 temporary_latex_code += '\\textcolor{' + str.lower(table_of_color[x, y].name) + '}{' + table_of_content
    [x, y] + '}'
421                 if x < len(self.__table_of_content[y]) - 1:
422                     temporary_latex_code += '&'
423             temporary_latex_code += '\\\\ \\hline\n'
424         temporary_latex_code += '\\end{tabular}\n\\end{table}\n'
425         return temporary_latex_code
426
427
```

Abbildung 4.33: Methode „convert_to_tex“ der Klasse „Table“

Die Methode „convert_to_tex“ generiert den LaTeX-Code, der zur Einbindung der jeweiligen Tabelle nötig ist. Dieser dient in Form eines Strings als Rückgabewert.

Small Test

```
428 # Small Test
429 """
430 text1 = Text('Text Title', 'This ist some Text')
431 text2 = Text('Text Title', '')
432 text3 = Text('', 'This ist some Text')
433 pic1 = Graphics('PNG Title', 'python.png', '100')
434 pic2 = Graphics('PGF Title', 'vgrafic.pgf', '100')
435 table1 = Table('Table Title', np.array([['0', '1'], ['2', '3']]),
436                np.array([[Colors.BLACK, Colors.BLACK], [Colors.RED, Colors.BLACK]]))
437 test1 = Test(OK, 'Test Title', np.array([text1, pic2, pic1, table1]))
438 test2 = Test(SKIPPED, 'Test Title', np.array([text2]))
439 test3 = Test(FAILED, 'Test Title', np.array([text1, pic2, pic1, table1]))
440 test2.add_graphics('', 'vgrafic.pgf', '15')
441 test2.add_graphics('', 'python.png', '15')
442 subsection1 = Subsection('Subsection Title', np.array([test1, test1]))
443 subsection2 = Subsection('Subsection Title', np.array([test2, test2]))
444 subsection3 = Subsection('Subsection Title', np.array([test3, test3]))
445 section1 = Section('Section Title', np.array([subsection1, subsection2, subsection1]))
446 section2 = Section('Section Title', np.array([subsection2, subsection2, subsection3]))
447 section3 = Section('Section Title', np.array([subsection1, subsection1]))
448 document1 = Document('Document Title', np.array([section1, section2, section3]), np.array([text2]))
449 document1.generate_report()
450 """
```

Abbildung 4.34: Schnelltestcode

Der „Small Test“ am Ende des Codes kann bei Modifikation der Bibliothek „testreport-gen.py“ genutzt werden, um einen kurzen Funktionstest zu machen. Sein Inhalt ist nahezu identisch mit dem des Skriptes „test.py“, auf das im Kapitel 4.2 näher eingegangen wird.

4.2 Tests

```python
import numpy as np
import testreportgen as rg

"""
Small test of "testreportgen"
This test creates a latex file using reportgen.
text1 contains both bold and normal text.
text2 just bold and text3 just normal text.
test1 and test2 will have graphics with captions, while Test 2 won't have captions.
test1 is OK, test2 is Skipped, test3 is Failed.
Subsection1 should be OK. Subsection2 should be Skipped. Subsection 3 should be Failed.
Section1 and section2 should be Failed. Section 3 should be OK.
"""

text1 = rg.Text('Text 1 Title', 'This ist some Text 1')
text2 = rg.Text('Text 2 Title', '')
text3 = rg.Text('', 'This ist some Text 3')
pic1 = rg.Graphics('PNG Title', 'python.png', '100')
pic2 = rg.Graphics('PGF Title', 'vgrafic.pgf', '100')
table1 = rg.Table('Table 1 Title', np.array([['0', '1'], ['2', '3']]),
                np.array([[rg.Colors.BLACK, rg.Colors.BLACK], [rg.Colors.RED, rg.Colors.BLACK]]))
test1 = rg.Test(rg.OK, 'Test 1 Title', np.array([text1, pic2, pic1, table1]))
test2 = rg.Test(rg.SKIPPED, 'Test 2 Title', np.array([text3, text2]))
test3 = rg.Test(rg.FAILED, 'Test 3 Title', np.array([text1, pic2, pic1, table1]))
test2.add_graphics('', 'vgrafic.pgf', '15')
test2.add_graphics('', 'python.png', '15')
subsection1 = rg.Subsection('Subsection Title', np.array([test1, test1]))
subsection2 = rg.Subsection('Subsection Title', np.array([test2, test2]))
subsection3 = rg.Subsection('Subsection Title', np.array([test3, test3]))
section1 = rg.Section('Section Title', np.array([subsection1, subsection2, subsection1]))
section2 = rg.Section('Section Title', np.array([subsection2, subsection2, subsection3]))
section3 = rg.Section('Section Title', np.array([subsection1, subsection1]))
document1 = rg.Document('Document Title', np.array([section1, section2, section3]), np.array([text2]))
document1.generate_report()

```

Abbildung 4.35: test.py

Die Tests erfolgten mit einem Testskript. Dies erzeugt mithilfe der Bibliothek „testreportgen.py“ eine Latex Datei. Ist diese kompilierfähig und erfüllt das so entstehende PDF die Vorgaben, so ist der Test bestanden. Die Vorgaben sind im Convluenzprotokoll festgehalten, dass sich im Anhang A.2 befindet.

Sowohl die Bibliothek „testreportgen.py“ als auch das Sktript „test.py“ funktionieren gemäß der Beschreibung.

5 Ausblick

Der Code befindet sich aktuell noch in der unfertigen Version 0.5. Vor der Veröffentlichung ist noch eine Einpflegung des Testskripts in Jenkins CI sowie die Anlegung einer README- und einer Lizenzdatei nötig. Als Lizens wurde bereits GNU Lesser General Puplic License Version 3 (LGPL3) ausgewählt.[Vgl. 2] Diese stellt den Code quelloffen und erlaubt Dinge, wie z.B. das Modifizieren des Codes und Erstellen von Ableitungen. Auch wurde die Software aktuell nur über ein Testskript getestet. Ein Feldtest in einem der vorhandenen Teststände steht noch aus. Sollte in der Lebenszeit von Python ein neuer Major Release erfolgen, so wäre eine Neuauflage der Bibliothek notwendig. Dies wird allerdings nicht vor Oktober 2026 der Fall sein, da die aktuell neueste Version Python 3.10 bis dahin unterstützt wird.[Vgl. 9]

5.1 Pflege und Kosten

Die Entwicklungsdauer des Codes betrug 160 h. Die Pflege des Codes, insbesondere beim Migrieren auf neue Python-Versionen, verursacht in Zukunft weitere Kosten. Wird davon ausgegangen, dass der daran arbeitende Ingenieur $120\frac{€}{h}$ kostet und die Dauer der Pflege 10 % der Entwicklungsdauer pro Jahr beträgt, kommen so folgende Kosten (K) zustande:

$$K = 160h \cdot 10\frac{\%}{a} \cdot 120\frac{€}{h} \tag{5.1}$$

$$K = 1920\frac{€}{a} \tag{5.2}$$

Jährlich kostet diese Software also ungefähr 1920€. Es sollte bei einem Migrieren also darauf geachtet werden, ob alternative Lösungen auf den Markt kommen, die geringere Kosten verursachen. Ein Einstellen der Pflege des entsprechenden Codes kann dann folglich ebenfalls Kosten sparen. In dem Falle sollte dann auch die Pflege des Codes eingestellt werden. Aufgrund der gewählten Lizenz sind veränderte Varianten auf dem Markt denkbar.

6 Fazit

In der Zeit der Bearbeitung dieses Projekts sind altbekannte Probleme durch die im Studium gewonnen Erfahrungen leicht zu meistern gewesen. Eine gute Planung und eine klare Zielsetzung ermöglichten ein zielgerichtetes Arbeiten. Zu den aufgetretenen Problemen zählten unter anderem eine zweiwöchige Phase zum Entfernen von Bugs und anderen Fehlern. Neue Probleme, die auftraten, waren zum Beispiel die Benennung der Software. In dieser Sache war es sinnvoll, das Internet nach potenziell gleichnamiger Software im Vorhinein zu durchsuchen. Darüber hinaus konnten erste Erfahrungen im Umgang mit einer bis dahin dem Verfasser unbekannten Programmiersprache gesammelt werden. Die regelmäßigen Meetings mit den Betreuern waren sehr lehrreich und trugen durch Weitergabe von Erfahrungen zur Lösung von Problemen bei.

Literaturverzeichnis

[1] Barry Warsaw, Ethan F.: *PEP 435 – Adding an Enum type to the Python standard library Python.org*. 2013. – URL https://docs.python.org/3/library/enum.html. – Zugriffsdatum: 01-12-2021

[2] Hoffmann, Matthias: *Report Generator*. 2021. – Interner Confluence Eintrag von DESY. Befindet sich im Anhang

[3] Kalista, Heiko: *Python 3 Einsteigen und Durchstarten*. Carl Hanser Verlag München, 2018. – URL https://www.hanser-elibrary.com/doi/book/10.3139/9783446456891. – ISBN 978-3-446-45689-1

[4] o.V.: *DESY im Überblick - Deutsches Elektronen-Synchrotron DESY*. o.J.. – URL https://www.desy.de/ueber_desy/desy/index_ger.html. – Zugriffsdatum: 01-12-2021

[5] o.V.: *Deutsches Elektronen-Synchrotron - Wikipedia*. o.J.. – URL https://de.wikipedia.org/wiki/Deutsches_Elektronen-Synchrotron. – Zugriffsdatum: 01-12-2021

[6] o.V.: *FLASH2020+ - Deutsches Elektronen-Synchrotron DESY*. o.J.. – URL https://www.desy.de/forschung/anlagen__projekte/flash2020/index_ger.html. – Zugriffsdatum: 01-12-2021

[7] o.V.: *Großgeräte für die Wissenschaft - Deutsches Elektronen-Synchrotron DESY*. o.J.. – URL https://www.desy.de/ueber_desy/desy/grossgeraete_fuer_die_wissenschaft/index_ger.html. – Zugriffsdatum: 01-12-2021

[8] Peterson, Benjamin: *PEP 373 – Python 2.7 Release Schedule Python.org*. 2008. – URL https://www.python.org/dev/peps/pep-0373/. – Zugriffsdatum: 01-12-2021

[9] Salgado, Pablo G.: *PEP 619 – Python 3.10 Release Schedule Python.org*. 2020. – URL https://www.python.org/dev/peps/pep-0619/. – Zugriffsdatum: 01-12-2021

A Anhang

A.1 Report Generator

Report Generator

Scope

MSK test stand software is usually written in Python. Up to now we were lacking a good way to generate test reports. Several approaches have been tried out, but all of them did not work sufficiently well.

- Directly generating Latex snippets from the code had stability problems as open and closing braces / begin-end statements were produced at completely different places and could cause broken latex documents. The situation was difficult to debug. The produced output on the other hand was superiour to all other solutions.
- The Robot Framework directly generated reports, but only in HTML. The browsable pages were not suitable as printable reports that can easily be archived.
- Markdown/ReST is not powerful enough to format tables (or was not when last evaluated a few years ago).
- Reportlab with Platypus could not process vector graphics and had issues with placing graphics. The usage was not as nice as announced (rather cryptic in some places), but could produce tables with the required highlighting. Due to the lack of graphics inclusion we had to fall back to creating individual pages as pdf and combine them with an additional tool. (State of 2014)

As all these findings are a couple of years old, the situation was re-evaluated by checking which tools are available at the moment, and what their performance is. If no matching tool is available, it is to be developed.

Technical requirements

1. Long term support is required (accelerators and this the hardware components are operating up to 20 years, and the test stand should be available for the whole lifetime). Preferably open source, as commercial vendors often don't have this long term support.
2. The tool is available for the current python relase 3 8, which is used on Ubuntu20.04 LTS
3. The tests can be structured in sections and sub-sections. Each section/subsection starts with a summary (test name and test result) of all sub-sections and tests, and a summed up test result
4. The cover page has several parts:
    a. A header section with title, test number, date, test software revision and further test-specific fields like hardware Ds etc. (extensible in each test software)
    b. The overall test result
    c. A summary of all test sections and section results
5. Each test has a result and can contain the following components
    a. Pixel graphics
    b. Vector graphics (should print/scale without quality loss)
    c. Free text blocks
    d. Tables
    Individual cells in tables can formatted with color highlighting, for instance to indicate which values caused the test to fail
6. Test results can be OK, Failed and Skipped, and are always shown in color code green, red and orange, respectively
    a. The status of sections/sub-sections and the overall test result are automatically calculated from the contained sections/sub-sections /tests
        i. Only if all sub-results are OK, a result is OK
        ii. If at least one sub-results is Failed, the result is failed
        iii. If all sub-results are Skipped, the result is Skipped. Any combination of Skipped with OK or Failed is results in Failed.
7. The resulting test report is a PDF document (A4 format) which is optimised for printing (e.g. no large coloured areas)
8. The code, code comments and documentation are written in English
9. The report is English

Market survey and design decisions

1. Marked survey revealed that the solutions on the marked are not sufficient.
2. The Marked survey was done using the Keywords "Python Report Generator" on the search engines of "www google.com" and "www.sciencedirekt.com"
3. Most promising results where the following:
    a. Pandas in combination with an HTML Template
    b. ReportLab
4. However, both had some caveats:
    a. Pandas would require a different HTML template for every protocol.
    So, it would require an Interface in python to generate the HTML file for the reports automatically.
    b. ReportLab's open-source solution doesn't support vector graphics.
    c. Neither of these Tools offer a convenient way of building the requested report structure and automatically calculated statues of sections/sub-sections and overall test results.
5. Because of this caveats a 3rd solution was made up.
6. 3rd solution is writing an API to create a Latex file in a convenient way within a python script.

Design decision

As only Latex can fulfill all requirements of the output as an open source solution, we are going for creating a library that acts as a wrapper around the creation of a latex file and provides an API to create a report with sections and put the contents conveniently from python.

API requirements

Definition of Done for v01.00

- [x] The software is checked in in https://gitlab.desy de/msk-sw/utilities/report-generator.git

- [ ] The software is published under the lgpl3 license
- [ ] Version v01 00.00 is tagged
- [x] There is an example "test stand software" which produces a report that show-cases all the required features
- [ ] The example test stand software is run in Jenkins CI to check that the library is working (smoke test to check that the example document can be produced). This has to be successful.
- [ ] Manual review of the API and how it is used in the example test stand software
- [ ] Manual review of the generated report to check that all technical requirements are met

A.2 testreportgen.py

File - E:\LatexCode\CodeGenerator\source\testreportgen.py

```
#!/usr/bin/env python3
"""testreportgen.py: This libery is a tool to generate reports.
Requires Python 3.4 or later"""

__version__ = '0.5'
__author__ = 'Dustin Fritsch'

import abc  # abstract classes
import numpy as np
from enum import Enum

OK = 'Ok'
SKIPPED = 'Skipped'
FAILED = 'Failed'
Colors = Enum('Colors', 'RED GREEN BLUE CYAN MAGENTA YELLOW BLACK GRAY WHITE DARKGRAY LIGHTGRAY BROWN LIME OLIVE '
                        'ORANGE PINK PURPLE TEAL VIOLET')
OK_COLOR = Colors.GREEN
SKIPPED_COLOR = Colors.ORANGE
FAILED_COLOR = Colors.RED


class ReportBuildingBlock(metaclass=abc.ABCMeta):
    """
    Metaclass to define some functions needed in the other classes
    Keyword arguments:
    result -- Value to define whether a block is Failed, OK or Skipped.
    title -- Title to name a certain block of the document.
    """

    def __init__(self, title):
        assert isinstance(title, str)
        self.__title = title
        self.__result = ''

    def get_result(self):
        return self.__result

    def _set_result(self, result):
        self.__result = result

    def get_title(self):
        return self.__title

    def set_title(self, title):
        self.__title = title

    @staticmethod
    def _generate_result_table(type_to_overview, array_to_overview):
        """
        Function to generate a quick table to overview results of certain parts of the document.
        temp is just a temporary string to build the requested code in.
        :param type_to_overview: Either sections, subsections or tests can be overviewed
        :param array_to_overview: Array containing the sections, subsections or tests to overview
        :return: String that contains the code to generate the desired table
        """
        temporary_latex_code = '\\begin{table}[h]\n\\centering\n\\begin{tabular}{|l|l|}\n\\hline\n'
        temporary_latex_code += type_to_overview + ' & Result\\\\ \\hline\n'
        for entry in array_to_overview:
            temporary_latex_code += entry.get_title() + '&\\textcolor{'
            if entry.get_result() == OK:
                temporary_latex_code += str.lower(OK_COLOR.name) + '}{Ok}'
            elif entry.get_result() == SKIPPED:
                temporary_latex_code += str.lower(SKIPPED_COLOR.name) + '}{Skipped}'
            elif entry.get_result() == FAILED:
                temporary_latex_code += str.lower(FAILED_COLOR.name) + '}{Failed}'
            temporary_latex_code += '\\\\ \\hline\n'
        temporary_latex_code += '\\end{tabular}\n\\end{table}\n'
        return temporary_latex_code

    @staticmethod
    def _check_array_result(array_of_content):
        """
        Checks the results of an array to build up a summarized result
        :return: Summarized result
        """
        temporary_result = OK
        for entry in array_of_content:
            if entry.get_result() != OK:
                temporary_result = SKIPPED
        if temporary_result == SKIPPED:
            for entry in array_of_content:
                if entry.get_result() != SKIPPED:
                    temporary_result = FAILED
        return temporary_result


class Document(ReportBuildingBlock):
    __document_class = '[a4paper,10pt]{article}'
    __packages = ['[latin1]{inputenc}', '[T1]{fontenc}', '[english]{babel}', '{color}', '{pgf}',
```

File - E:\LatexCode\CodeGenerator\source\testreportgen.py

```
                '{transparent}', '{float}']

    def __init__(self, title, sections, content_of_cover_page):
        """
        :param title: Title string of the document
        :param sections: Array of sections
        :param content_of_cover_page: Array containing text, table's or graphics for the cover page
        """
        ReportBuildingBlock.__init__(self, title)
        assert isinstance(sections, np.ndarray)
        for entry in sections:
            assert isinstance(entry, Section)
        assert isinstance(content_of_cover_page, np.ndarray)
        for entry in content_of_cover_page:
            assert isinstance(entry, (Graphics, Text, Table))
        self.__sections = sections
        self.__content_of_cover_page = content_of_cover_page

    def check_result(self):
        """
        Checks the results of the sections
        :return: Summarized result
        """
        for entry in self.__sections:
            entry.check_result()
        temporary_result = self._check_array_result(self.__sections)
        self._set_result(temporary_result)
        return temporary_result

    def generate_report(self):
        """
        :return: Writes a latex file that contains the code to generate the report.
        """
        self.check_result()
        # Generate the beginning of the code cointaining user packages
        temporary_latex_code = ''
        temporary_latex_code += '\\documentclass' + self.__document_class + '\n'
        for package in self.__packages:
            temporary_latex_code += '\\usepackage' + package + '\n'
        temporary_latex_code += '\\begin{document}\n'

        # Generate cover page
        temporary_latex_code += '\\title{' + self.get_title() + '}\n'
        temporary_latex_code += '\\setlength{\\parindent}{0em}\n \\maketitle\n'
        for entry in self.__content_of_cover_page:
            temporary_latex_code += entry.convert_to_tex()

        # Generate table of contents page
        temporary_latex_code += '\\newpage\n \\tableofcontents\n \\newpage'

        # Generate table of sections result page
        temporary_latex_code += self._generate_result_table('Section', self.__sections)
        temporary_latex_code += '\\newpage'

        # Generate code for the sections
        for entry in self.__sections:
            temporary_latex_code += entry.convert_to_tex()

        # Finish the code and write it into an latex file
        temporary_latex_code += '\\end{document}'
        file = open("latex.tex", "w")
        file.write(temporary_latex_code)

    def add_section(self, section):
        """
        :param section: Section to add
        :return: Adds section into the self.__sections array
        """
        assert isinstance(section, Section)
        self.__sections = np.append(self.__sections, [section])


class Section(ReportBuildingBlock):

    def __init__(self, title, subsections):
        """
        :param title: Title string of the section
        :param subsections: Array of subsections
        """
        ReportBuildingBlock.__init__(self, title)
        assert isinstance(subsections, np.ndarray)
        for entry in subsections:
            assert isinstance(entry, Subsection)
        self.__subsections = subsections

    def convert_to_tex(self):
        """
        Generates latex code of the current section
        :return: Latex code string needed to build the section
```

Page 2 of 6

File - E:\LatexCode\CodeGenerator\source\testreportgen.py

```
179         """
180         # Generate section title
181         temporary_latex_code = ''
182         temporary_latex_code += '\\section{' + self.get_title() + '}\n'
183 
184         # Generate table of subsections result
185         temporary_latex_code += self._generate_result_table('Subsection', self.__subsections)
186 
187         # Generate code for the subsections
188         for entry in self.__subsections:
189             temporary_latex_code += entry.convert_to_tex()
190         return temporary_latex_code
191 
192     def check_result(self):
193         """
194         Checks the results of the subsections
195         :return: Summarized result
196         """
197         for entry in self.__subsections:
198             entry.check_result()
199         temporary_result = self._check_array_result(self.__subsections)
200         self._set_result(temporary_result)
201         return temporary_result
202 
203     def add_subsection(self, subsection):
204         """
205         :param subsection: Subsection to add
206         :return: Adds subsection to the self.__subsections array
207         """
208         assert isinstance(subsection, Subsection)
209         self.__subsections = np.append(self.__subsections, [subsection])
210 
211 
212 class Subsection(ReportBuildingBlock):
213 
214     def __init__(self, title, tests):
215         """
216         :param title: Title string of the subsection
217         :param tests: Array of tests
218         """
219         ReportBuildingBlock.__init__(self, title)
220         assert isinstance(tests, np.ndarray)
221         for entry in tests:
222             assert isinstance(entry, Test)
223         self.__tests = tests
224 
225     def convert_to_tex(self):
226         """
227         Generates latex code of the current subsection
228         :return: Latex code string needed to build the subsection
229         """
230         # Generate section title
231         temporary_latex_code = ''
232         temporary_latex_code += '\\subsection{' + self.get_title() + '}\n'
233 
234         # Generate table of test result
235         temporary_latex_code += self._generate_result_table('Test', self.__tests)
236         for entry in self.__tests:
237             temporary_latex_code += entry.convert_to_tex()
238         return temporary_latex_code
239 
240     def check_result(self):
241         """
242         Checks the results of the tests
243         :return: Summarized result
244         """
245         temporary_result = self._check_array_result(self.__tests)
246         self._set_result(temporary_result)
247         return temporary_result
248 
249     def add_test(self, test):
250         """
251         :param test: Test to add
252         :return: Adds test to self.__tests array
253         """
254         assert isinstance(test, Test)
255         self.__tests = np.append(self.__tests, [test])
256 
257 
258 class Test(ReportBuildingBlock):
259 
260     def __init__(self, result, title, content):
261         """
262         :param result: String to define whether this test is Failed, OK or Skipped.
263         :param title: Title string of the Test
264         :param content: Array containing text, graphics or table objects
265         """
266         assert isinstance(result, str)
267         assert isinstance(title, str)
```

Page 3 of 6

File - E:\LatexCode\CodeGenerator\source\testreportgen.py

```
268         assert result == FAILED or result == SKIPPED or result == OK
269         assert isinstance(content, np.ndarray)
270         for entry in content:
271             assert isinstance(entry, (Graphics, Text, Table))
272         self.__title = title
273         self.__result = result
274         self.__content = content
275 
276     def get_result(self):
277         return self.__result
278 
279     def _set_result(self, result):
280         self.__result = result
281 
282     def get_title(self):
283         return self.__title
284 
285     def set_title(self, title):
286         self.__title = title
287 
288     def add_graphics(self, title, path_to_graphics, width):
289         """
290         :param title: Caption string of the graphics
291         :param path_to_graphics: Location of the graphics (/path/to/graphics.ong)
292         :param width: Number in string format to define the size of the graphics
293         :return: Adds graphics to the test
294         """
295         self.__content = np.append(self.__content, [Graphics(title, path_to_graphics, width)])
296 
297     def add_text(self, title, text):
298         """
299         :param title: Caption string of the text
300         :param text: String containing actual text
301         :return: Adds text to the test
302         """
303         self.__content = np.append(self.__content, [Text(title, text)])
304 
305     def add_content(self, content):
306         """
307         :param content: Graphics, text or table to add
308         :return: Adds a graphics, text or table object to the test
309         """
310         assert isinstance(content, (Graphics, Text, Table))
311         self.__content = np.append(self.__content, [content])
312 
313     def convert_to_tex(self):
314         """
315         Generates latex code of the current test
316         :return: Latex code string needed to build the test
317         """
318         temporary_latex_code = '\\textbf{' + self.get_title() + '}\n\\newline\n'
319         for entry in self.__content:
320             temporary_latex_code += entry.convert_to_tex()
321         return temporary_latex_code
322 
323 
324 class Graphics(ReportBuildingBlock):
325 
326     def __init__(self, title, path_to_graphics, width):
327         """
328         :param title: Caption string of the graphics
329         :param path_to_graphics: path_to_graphics: Location of the graphics (/path/to/graphics.ong)
330         :param width: Number in string format to define the size of the graphics
331         """
332         ReportBuildingBlock.__init__(self, title)
333         assert isinstance(path_to_graphics, str)
334         assert isinstance(width, str)
335         self.__path_to_graphics = path_to_graphics
336         self.__width = width
337 
338     def convert_to_tex(self):
339         """
340         Generates latex code of the current graphics
341         :return: Latex code string needed to build the graphics
342         """
343         temporary_latex_code = '\\begin{figure}[H] \n\\centering \n'
344         # Vector graphics in pgf format require a different approach
345         if (self.__path_to_graphics[-3] == 'p') & (self.__path_to_graphics[-2] == 'g') & \
346                 (self.__path_to_graphics[-1] == 'f'):
347             temporary_latex_code += '\\def\\svgwidth{' + self.__width + 'pt} \n\\input{' + \
348                     self.__path_to_graphics + '}\n'
349         # then other graphics
350         else:
351             temporary_latex_code += '\\includegraphics[width=' + self.__width + 'pt]{' + \
352                     self.__path_to_graphics + '}\n'
353         # Only add caption, if it exists
354         if self.get_title() != '':
355             temporary_latex_code += '\\caption{' + self.get_title() + '}\n'
356         temporary_latex_code += '\\end{figure}\n'
```

Page 4 of 6

File - E:\LatexCode\CodeGenerator\source\testreportgen.py

```
        return temporary_latex_code


class Text(ReportBuildingBlock):

    def __init__(self, title, text):
        """
        :param title: Caption string of the text
        :param text: String containing actual text
        """
        ReportBuildingBlock.__init__(self, title)
        assert isinstance(text, str)
        self.__text = text

    def convert_to_tex(self):
        """
        Generates latex code of the current text
        :return: Latex code string needed to build the text
        """
        temporary_latex_code = ''
        # Only add caption, if it exists
        if self.get_title() != '':
            temporary_latex_code += '\\textbf{' + self.get_title() + '}\n\\newline\n'
        # Only add text, if it exists
        if self.__text != '':
            temporary_latex_code += '\\textmd{' + self.__text + '}\n\\newline\n'
        return temporary_latex_code


class Table(ReportBuildingBlock):

    def __init__(self, title, table_of_content, table_of_color):
        """
        :param title: Caption string of the table
        :param table_of_content: 2D Array containing the strings to print into the table.
        :param table_of_color: 2D Array containing the strings to color the table.
        """
        ReportBuildingBlock.__init__(self, title)
        assert isinstance(table_of_content, np.ndarray)
        assert isinstance(table_of_color, np.ndarray)
        for list_of_color in table_of_color:
            for color in list_of_color:
                assert isinstance(color, Colors)
        assert table_of_content.shape == table_of_color.shape
        self.__table_of_content = table_of_content
        self.__table_of_color = table_of_color

    def convert_to_tex(self):
        """
        Generates latex code of the current table
        :return: Latex code string needed to build the table
        """
        temporary_latex_code = '\\begin{table}[h]\n\\centering\n'
        # Only add caption, if it exists
        if self.get_title() != '':
            temporary_latex_code += '\\caption{' + self.get_title() + '}\n'
        # add table format
        temporary_latex_code += '\\begin{tabular}{|' + self.__table_of_content.shape[1] * 'l|' + '}\n\\hline\n'
        # create flipped version of table. These are required for the following loop to write the corresponding code
        table_of_content = np.rot90(np.fliplr(self.__table_of_content))
        table_of_color = np.rot90(np.fliplr(self.__table_of_color))
        for y in range(len(self.__table_of_content)):
            for x in range(len(self.__table_of_content[y])):
                temporary_latex_code += '\\textcolor{' + str.lower(table_of_color[x, y].name) + '}{' + table_of_content
[x, y] + '}'
                if x < len(self.__table_of_content[y]) - 1:
                    temporary_latex_code += '&'
            temporary_latex_code += '\\\\ \\hline\n'
        temporary_latex_code += '\\end{tabular}\n\\end{table}\n'
        return temporary_latex_code


# Small Test
"""
text1 = Text('Text Title', 'This ist some Text')
text2 = Text('Text Title', '')
text3 = Text('', 'This ist some Text')
pic1 = Graphics('PNG Title', 'python.png', '100')
pic2 = Graphics('PGF Title', 'vgrafic.pgf', '100')
table1 = Table('Table Title', np.array([['0', '1'], ['2', '3']]),
               np.array([[Colors.BLACK, Colors.BLACK], [Colors.RED, Colors.BLACK]]))
test1 = Test(OK, 'Test Title', np.array([text1, pic2, pic1, table1]))
test2 = Test(SKIPPED, 'Test Title', np.array([text2]))
test3 = Test(FAILED, 'Test Title', np.array([text1, pic2, pic1, table1]))
test2.add_graphics('', 'vgrafic.pgf', '15')
test2.add_graphics('', 'python.png', '15')
subsection1 = Subsection('Subsection Title', np.array([test1, test1]))
subsection2 = Subsection('Subsection Title', np.array([test2, test2]))
subsection3 = Subsection('Subsection Title', np.array([test3, test3]))
```

Page 5 of 6

File - E:\LatexCode\CodeGenerator\source\testreportgen.py

```
445 section1 = Section('Section Title', np.array([subsection1, subsection2, subsection1]))
446 section2 = Section('Section Title', np.array([subsection2, subsection2, subsection3]))
447 section3 = Section('Section Title', np.array([subsection1, subsection1]))
448 document1 = Document('Document Title', np.array([section1, section2, section3]), np.array([text2]))
449 document1.generate_report()
450 """
```

Page 6 of 6

A.3 test.py

File - E:\LatexCode\CodeGenerator\source\test.py

```
1 import numpy as np
2 import testreportgen as rg
3
4 """
5 Small test of "testreportgen"
6 This test creates a latex file using reportgen.
7 text1 contains both bold and normal text.
8 text2 just bold and text3 just normal text.
9 test1 and test2 will have graphics with captions, while Test 2 won't have captions.
10 test1 is OK, test2 is Skipped, test3 is Failed.
11 Subsection1 should be OK. Subsection2 should be Skipped. Subsection 3 should be Failed.
12 Section1 and section2 should be Failed. Section 3 should be OK.
13 """
14
15 text1 = rg.Text('Text 1 Title', 'This ist some Text 1')
16 text2 = rg.Text('Text 2 Title', '')
17 text3 = rg.Text('', 'This ist some Text 3')
18 pic1 = rg.Graphics('PNG Title', 'python.png', '100')
19 pic2 = rg.Graphics('PGF Title', 'vgrafic.pgf', '100')
20 table1 = rg.Table('Table 1 Title', np.array([['0', '1'], ['2', '3']]),
21               np.array([[rg.Colors.BLACK, rg.Colors.BLACK], [rg.Colors.RED, rg.Colors.BLACK]]))
22 test1 = rg.Test(rg.OK, 'Test 1 Title', np.array([text1, pic2, pic1, table1]))
23 test2 = rg.Test(rg.SKIPPED, 'Test 2 Title', np.array([text3, text2]))
24 test3 = rg.Test(rg.FAILED, 'Test 3 Title', np.array([text1, pic2, pic1, table1]))
25 test2.add_graphics('', 'vgrafic.pgf', '15')
26 test2.add_graphics('', 'python.png', '15')
27 subsection1 = rg.Subsection('Subsection Title', np.array([test1, test1]))
28 subsection2 = rg.Subsection('Subsection Title', np.array([test2, test2]))
29 subsection3 = rg.Subsection('Subsection Title', np.array([test3, test3]))
30 section1 = rg.Section('Section Title', np.array([subsection1, subsection2, subsection1]))
31 section2 = rg.Section('Section Title', np.array([subsection2, subsection2, subsection3]))
32 section3 = rg.Section('Section Title', np.array([subsection1, subsection1]))
33 document1 = rg.Document('Document Title', np.array([section1, section2, section3]), np.array([text2]))
34 document1.generate_report()
35
```

Page 1 of 1

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original