Roadmap: TruCol
A decentralised collaboration protocol for test-driven development

September 5, 2022

1 Introduction

This document presents the roadmap for the TruCol company as well as the estimated costs of the accompanying human labour and other posts. The objective of this document is to provide some basic insight into the order of magnitude of the costs of developing a sustainable variant of the TruCol company such that it is able to generate returns for its potential investors.

Section 2 specifies the assumptions, section 3 describes how the costs are estimated, section 4 presents the projected basic planning and briefly details what each activity in the Gantt chart represents. Section 7 provides some nuance on the accuracy of the estimates, and section 8 summarises the body of information in this document.

2 Assumptions

2.1 Decentralisation Developer Wages

The hourly wage of the developers working on decentralised technology is based on a mixture of $\pm$ 3 junior developers working at €100.000,- per year, and 2 senior developers working at €200.000,- per year. This yields an average developer cost of

$$\frac{3 \cdot 50 + 2 \cdot 100}{5} = \frac{350}{5} = €70,- \tag{1}$$

The datapoints used to come to this estimate are the promoted starting wages for Junior Developers/Engineers at Optiver in Amsterdam, Think-cell in Berlin, and a third Zurich company, which all ranged between 80 to 120k at the time of inspection (Around March 2021). No proper datapoint is used to estimate the salary of the senior developers. Previous experience in co-working with senior developers led to an estimate that their hourly contributions are at least twice as valuable as that of a junior developer. Another indicator for the doubling in wage between junior and senior dev may be the hear-say high demand in solidity/decentralisation developers.

The 70,- hourly wage is interpreted as €75, − per hour to be on the conservative side of estimates.

2.2 Website/Platform Wages

The website+API+GUI development is estimated at €40 per hour. This estimate is based on a reduced hourly wage of the junior decentralised technology developers (from €50, − to €40, −). Some of the development costs for these activities may be performed at a lower hourly cost price, this platform development work also contains UX design. And excellent UX design is quite costly, hence the average hourly wage for this estimate is kept at €40, −.

2.3 Business wages

The hourly wages for the business development side of our company is estimated at €35, − per hour. This estimate is based on a reduced hourly wage of the junior platform developers (from €40, − to €35, −).

2.4 Activity durations

The estimates for the durations of the activities for both decentralised technology development as well as ecosystem development are extrapolations of our experience in developing in these disciplines. The business development activities durations are based roughly on estimating what those activities entail and how long it would take to complete them.

3   Cost Model Description

The total costs are computed based on two factors.

- The cumulative amount of human labour hours that are planned to be executed, multiplied with their respective hourly wage costs as specified in section 2.

- A combination of bounty subsidisation and buffer costs of €100.000, − are estimated to generate wide-spread adoption of the TruCol protocol.

4   Results

Figure 1 contains the Gantt chart that is generated to plan the development of the TruCol company. One can observe that several of the development-activities can be performed in parallel, these are accordingly stacked vertically. Dependencies of outputs of activities imply a "stairway" pattern in the Gantt chart.
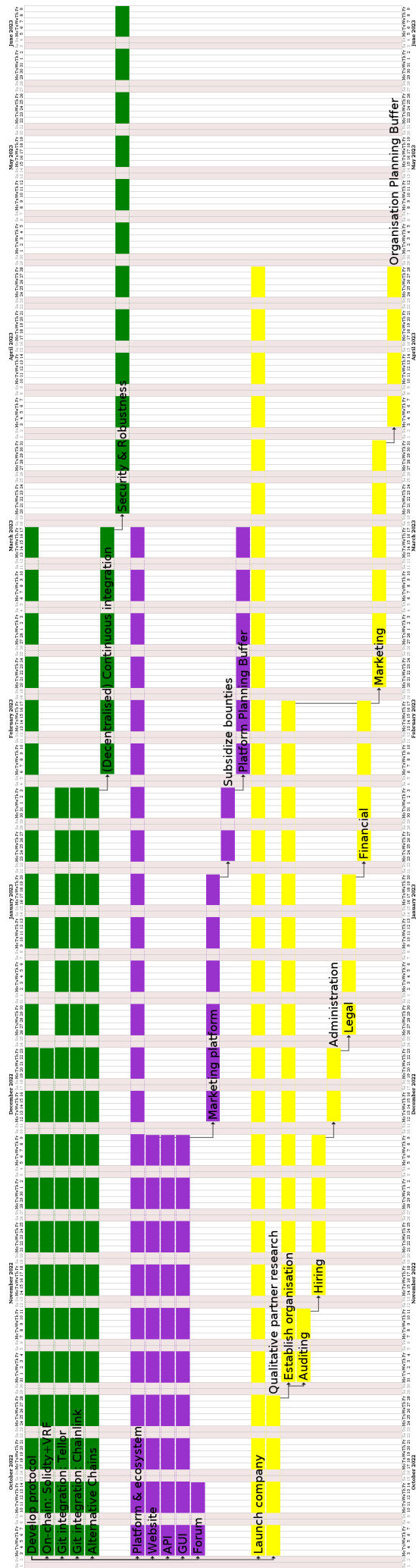
Figure 1: Gantt chart that is generated to plan the development of the TruCol company. (Source code in appendices, and on github.com/trucol/Roadmap).

The description of the activities can be given as:

4.1   Decentralised Technology Development

- **Develop protocol** - The programming work and documentation work that is required to render the TruCol protocol to a mature and robust state.

- **On-chain: Solidity+VRF** - Finalisation of the Solidity to Solidity implementation of the TruCol protocol implementation that leverages Chainlinks Verifable Random function (VRF).

- **Git integration: Tellor** - Providing a lower-cost option to the users whilst allowing the user to apply the TruCol protocol in practically any programming language using Tellor oracles that query Git repository content and (build) status.

- **Git integration: Chainlink** - Same as the Tellor option, except using Chainlinks oracles.

- **Alternative Chains** - Implementing the TruCol protocol in alternative chains to facilitate easy use for the users whilst possibly lowering costs and/or modulating the desired levels of decentralisation

- **(Decentralised) Continuous Integration** - Realizing a mature implementation in which the oracles can verify the build status (whether the tests in the smart contract actually passed or not) in a robust fashion. Ideally implementing support for decentralised CIs.

- **Security & Robustness** - A security audit of the TruCol protocol implementations.

4.2   Platform Development

- **Platform & Ecosystem** - Development of the online platform that provides a convenient place for users to use-, discuss- and learn about the TruCol protocol and its various implementations.

- **Website** - Completion of the company website.

- **API** - Application programming interface that allows users to submit contracts using the command line interface (CLI).

- **GUI** - Graphical user interface, that makes it easy and intuitive for new users to start using the TruCol protocol for their applications.

- **Forum** - Environment a-la stack-overflow that cultivates a knowledge base around the use of the TruCol protocol.

- **Marketing platform** - Development of the approach to realise wide-spread adoption of the TruCol ecosystem.

- **Subsidize bounties** - Subsidisation of bounties to attract new users to the platform.

- **Platform Planning Buffer** - A buffer accounting for unknown unknowns/unexpected delays.

4.3   Business Development

- **Launch company** - The administrative and non-technical aspects of growing the TruCol company.

- **Qualitative partner research** - An analysis to identify relevant partners in the growth of our company.

- **Establish Organisation** - organisational aspects of growing the company, with "Auditing, Hiring, Administration, Legal & Financial tasks as its respective subset".

- **Marketing** - Development of the approach to realise wide-spread adoption of the TruCol protocol whilst realising a steady stream of new customers.

- **Organisation Planning Buffer** - A buffer accounting for unknown unknowns/unexpected delays.

5 Cost estimates

After multiplying the amount of labour hours with their respective hourly labour costs, a total estimated labour costs of €470.400, − is generated. This is composed of:

- Decentralised Tech Development: €252.000, −

- Platform Development: €112.000, −

- Business Development €106.400, −

An additional €100.000, − are included for bounty subsidisation and as a buffer, yielding a total expected cost to create a healthy company of roughly €470.000+€100.000 =€570.000, − which is rounded to approximately .6 Meuro.

6 Sensitivity Analysis

7 Discussion

The hourly wage estimates for the senior decentralised technology development, platform development and business estimates could be refined by using databases of competitive salaries for the respective tasks/work. Furthermore, a refinement of the duration estimates is proposed at the moment the tasks are started, as we expect more relevant information will be available at those times. In addition, external resources should be addressed to check whether any (critical) components are omitted in this Gantt planning.

8 Conclusion

This document presents the planning to develop the healthy TruCol company within the timeframe of roughly 9 months, and documents the assumptions and methods used to generate this planning. The main tasks in this planning are composed of decentralised technology development, TruCol platform development and business development. A total labour cost of roughly €470k is estimated, and another €100k is estimated for bounty subsidisation and as a buffer.

At the end of this planning, the TruCol company is expected to sustainably operate, generating ROI. Our aim is to gradually increase the cultivation of the diversification potential of the TruCol protocol at this point, whilst starting to develop a strategy to develop our own in-house automation/AI-engine based on the dataset that we continuously grow.

References

A Appendix src/__main__.py

```python
"""Entry point for this project, runs the project code and exports
    ↪ data if
export commands are given to the cli command that invokes this script
    ↪ ."""


from src.arg_parser import parse_cli_args
from src.export_data.export_data import export_data

# Project code imports.
from src.Gantt import Gantt

# Export data import.


# Parse command line interface arguments to determine what this
    ↪ script does.
args = parse_cli_args()

# create gantt chart
gantt = Gantt("src/export_data/Diagrams/Dynamic_diagrams/gantt.uml")

# Run data export code if any argument is given.
if not all(
    arg is None for arg in [args.l, args.dd, args.sd, args.c2l, args.
        ↪ ec2l]
):
    export_data(args)
```

B Appendix src/arg_parser.py

```python
"""This is the main code of this project nr, and it manages running
    ↪ the code
and outputting the results to LaTex."""
import argparse


def parse_cli_args():
    """Parses the command line arguments to determine what this code
        ↪ should
    do."""
    # Instantiate the parser
    parser = argparse.ArgumentParser(description="Optional app
        ↪ description")

    # Include argument parsing for data exporting code.
    # Compile LaTex
    parser.add_argument(
        "--l",
        action="store_true",
        help="Boolean indicating if code compiles LaTex",
    )

    # Generate, compile and export Dynamic PlantUML diagrams to LaTex
        ↪ .
    parser.add_argument(
        "--dd",
        action="store_true",
        help=(
            "A boolean indicating if code generated diagrams are
                ↪ compiled"
            + " and exported."
        ),
    )
    # Generate, compile and export Static PlantUML diagrams to LaTex.
    parser.add_argument(
        "--sd",
        action="store_true",
        help=(
            "A boolean indicating if static diagrams are compiled and
                ↪ "
            + " exported."
        ),
    )

    # Export the project code to LaTex.
    parser.add_argument(
        "--c2l",
        action="store_true",
        help="A boolean indicating if project code is exported to
            ↪ LaTex.",
    )

    # Export the exporting code, and the project code to LaTex.
    parser.add_argument(
        "--ec2l",
        action="store_true",
        help=(
            "A boolean indicating if code that exports code is
                ↪ exported"
            + " to LaTex."
```

```
53            ),
54        )
55        # Load the arguments that are given.
56        args = parser.parse_args()
57        return args
```

C   Appendix src/export_data/Hardcoded_data.py

```python
"""Specify hardcoded output data."""


# pylint: disable=R0902
# pylint: disable=R0903
class Hardcoded_data:
    """Contains a list of parameters that are used in this project
       ↪ that
    combines Python code with a LaTex document."""

    def __init__(self):

        # Specify code configuration details
        # TODO: include as optional arguments.
        self.await_compilation = True
        self.verbose = True
        self.gantt_extension = ".uml"
        self.diagram_extension = ".png"

        # Filenames.
        self.main_latex_filename = "main.tex"
        self.export_data_dirname = "export_data"
        self.diagram_dir = "Diagrams"
        self.plant_uml_java_filename = "plantuml.jar"

        # Appendix manager filenames
        self.export_code_appendices_filename = "
           ↪ export_code_appendices.tex"
        self.export_code_appendices_filename_from_root = (
            "export_code_appendices_from_root.tex"
        )
        self.project_code_appendices_filename = "
           ↪ project_code_appendices.tex"
        self.project_code_appendices_filename_from_root = (
            "project_code_appendices_from_root.tex"
        )
        self.automatic_appendices_manager_filenames = [
            self.export_code_appendices_filename,
            self.export_code_appendices_filename_from_root,
            self.project_code_appendices_filename,
            self.project_code_appendices_filename_from_root,
        ]

        self.manual_appendices_filename = "manual_appendices.tex"
        self.manual_appendices_filename_from_root = (
            "manual_appendices_from_root.tex"
        )
        self.manual_appendices_manager_filenames = [
            self.manual_appendices_filename,
            self.manual_appendices_filename_from_root,
        ]
        self.appendix_dir_from_root = "latex/Appendices/"

        # Folder names.
        self.dynamic_diagram_dir = "Dynamic_diagrams"
        self.static_diagram_dir = "Static_diagrams"

        # Specify paths relative to root.
        self.path_to_export_data_from_root = f"src/{self.
           ↪ export_data_dirname}"
```

```python
57         self.jar_path_relative_from_root = (
58             f"{self.path_to_export_data_from_root}"
59             + f"/{self.plant_uml_java_filename}"
60         )
61         self.diagram_output_dir_relative_to_root = (
62             f"latex/Images/{self.diagram_dir}"
63         )
64
65         # Path related variables
66         self.append_export_code_to_latex = True
67         self.path_to_dynamic_gantts = (
68             f"{self.path_to_export_data_from_root}/"
69             + f"{self.diagram_dir}/{self.dynamic_diagram_dir}"
70         )
71         self.path_to_static_gantts = (
72             f"{self.path_to_export_data_from_root}/"
73             + f"{self.diagram_dir}/{self.static_diagram_dir}"
74         )
```

D Appendix src/export_data/Plot_to_tex.py

```python
"""File used to export plots to a latex directory.
Call this from another file, for project 11, question 3b:
from Plot_to_tex import Plot_to_tex as plt_tex
multiple_y_series = np.zeros((nrOfDataSeries,nrOfDataPoints), dtype=
    ↪ int);
# actually fill with data
lineLabels = [] # add a label for each dataseries
plt_tex.plotMultipleLines(plt_tex,single_x_series,multiple_y_series,"
    ↪ x-axis
 label [units]","y-axis label [units]",lineLabels,"3b",4,11)
4b=filename
4 = position of legend, e.g. top right.
For a single line, use:
plt_tex.plotSingleLine(plt_tex,range(0, len(dataseries)),dataseries,"
    ↪ x-axis
label [units]","y-axis label [units]",lineLabel,"3b",4,11)"""

import os

import matplotlib.pyplot as plt
import numpy as np
from matplotlib import lines

# You can also plot a table directly into latex, see
    ↪ example_create_a_table(..)
# Then put it in latex with for example:
# \begin{table}[H]
#     \centering
#     \caption{Results some computation.}\label{tab:some_computation}
#     \begin{tabular}{|c|c|} % remember to update this
# % to show all columns of table
#         \hline
#         \input{latex/project3/tables/q2.txt}
#     \end{tabular}
# \end{table}


class Plot_to_tex:
    """Object used to output plots to latex directory of project."""

    def __init__(self):
        self.script_dir = self.get_script_dir()

    # plot graph (legendPosition = integer 1 to 4)
    def plotSingleLine(
        self,
        x_path,
        y_series,
        x_axis_label,
        y_axis_label,
        label,
        filename,
        legendPosition,
        project_name,
    ):
        """

        :param x_path: param y_series:
        :param x_axis_label: param y_axis_label:
        :param label: param filename:
```

```python
        :param legendPosition: param project_name:
        :param y_series: param y_axis_label:
        :param filename: param project_name:
        :param y_axis_label: param project_name:
        :param project_name:

        """
        # pylint: disable=R0913
        # TODO: reduce 9/5 arguments to at most 5/5 arguments.
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.plot(x_path, y_series, c="b", ls="-", label=label,
            ↪ fillstyle="none")
        plt.legend(loc=legendPosition)
        plt.xlabel(x_axis_label)
        plt.ylabel(y_axis_label)
        plt.savefig(
            os.path.dirname(__file__)
            + f"/../../../latex/{project_name}"
            + "/Images/"
            + filename
            + ".png"
        )

    #       plt.show();

    # plot graphs
    def plotMultipleLines(
        self,
        x,
        y_series,
        x_label,
        y_label,
        label,
        filename,
        legendPosition,
        project_name,
    ):
        """

        :param x: param y_series:
        :param x_label: param y_label:
        :param label: param filename:
        :param legendPosition: param project_name:
        :param y_series: param y_label:
        :param filename: param project_name:
        :param y_label: param project_name:
        :param project_name:

        """
        # pylint: disable=R0913
        # TODO: reduce 9/5 arguments to at most 5/5 arguments.
        fig = plt.figure()
        ax = fig.add_subplot(111)

        # generate colours
        cmap = self.get_cmap(len(y_series[:, 0]))

        # generate line types
        lineTypes = self.generateLineTypes(y_series)

        for i in range(0, len(y_series)):
```

```python
118             # overwrite linetypes to single type
119             lineTypes[i] = "-"
120             ax.plot(
121                 x,
122                 y_series[i, :],
123                 ls=lineTypes[i],
124                 label=label[i],
125                 fillstyle="none",
126                 c=cmap(i),
127             )
128             # color
129
130         # configure plot layout
131         plt.legend(loc=legendPosition)
132         plt.xlabel(x_label)
133         plt.ylabel(y_label)
134         plt.savefig(
135             os.path.dirname(__file__)
136             + f"/../../../latex/{project_name}"
137             + "/Images/"
138             + filename
139             + ".png"
140         )
141
142     # Generate random line colours
143     # Source: https://stackoverflow.com/questions/14720331/
144     # how-to-generate-random-colors-in-matplotlib
145     def get_cmap(self, n, name="hsv"):
146         """Returns a function that maps each index in 0, 1, ..., n-1
                ↪ to a
147         distinct RGB color; the keyword argument name must be a
                ↪ standard mpl
148         colormap name.
149
150         :param n: param name:  (Default value = "hsv")
151         :param name: Default value = "hsv")
152         """
153         return plt.cm.get_cmap(name, n)
154
155     def generateLineTypes(self, y_series):
156         """
157
158         :param y_series:
159
160         """
161         # generate varying linetypes
162         typeOfLines = list(lines.lineStyles.keys())
163
164         while len(y_series) > len(typeOfLines):
165             typeOfLines.append("-.")
166
167         # remove void lines
168         for i in range(0, len(y_series)):
169             if typeOfLines[i] == "None":
170                 typeOfLines[i] = "-"
171             if typeOfLines[i] == "":
172                 typeOfLines[i] = ":"
173             if typeOfLines[i] == " ":
174                 typeOfLines[i] = "--"
175         return typeOfLines
176
```

```python
        # Create a table with: table_matrix = np.zeros((4,4),dtype=object
        ↪ ) and pass
        # it to this object
        def put_table_in_tex(self, table_matrix, filename, project_name):
            """

            :param table_matrix: param filename:
            :param project_name: param filename:
            :param filename:

            """
            cols = np.shape(table_matrix)[1]
            some_format = "%s"
            for _ in range(1, cols):
                some_format = some_format + " & %s"
            some_format = some_format + ""
            # TODO: Change to something else to save as txt.
            np.savetxt(
                os.path.dirname(__file__)
                + f"/../../../latex/{project_name}"
                + "/tables/"
                + filename
                + ".txt",
                table_matrix,
                delimiter=" & ",
                fmt=format,
                newline="  \\\\ \\hline \n",
            )

        # replace this with your own table creation and then pass it to
        # put_table_in_tex(..)
        def example_create_a_table(self):
            """Example on how to create a latex table from Python."""
            project_name = "1"
            table_name = "example_table_name"
            rows = 2
            columns = 4
            table_matrix = np.zeros((rows, columns), dtype=object)
            table_matrix[:, :] = ""   # replace the standard zeros with
                ↪ empty cell
            print(table_matrix)
            for column in range(0, columns):
                for row in range(0, rows):
                    table_matrix[row, column] = row + column
            table_matrix[1, 0] = "example"
            table_matrix[0, 1] = "grid sizes"

            self.put_table_in_tex(table_matrix, table_name, project_name)

        def get_script_dir(self):
            """returns the directory of this script regardless of from
                ↪ which level
            the code is executed."""
            return os.path.dirname(__file__)

        def export_plot(self, some_plt, filename):
            """

            :param plt:
            :param filename:

            """
```

```python
            self.create_target_dir_if_not_exists("latex/Images/", "graphs
                ↪ ")
            some_plt.savefig(
                "latex/Images/" + "graphs/" + filename + ".png", dpi=200
            )

    def create_target_dir_if_not_exists(self, path, new_dir_name):
        """

        :param path:
        :param new_dir_name:

        """
        if os.path.exists(path):
            if not os.path.exists(f"{path}/{new_dir_name}"):
                os.makedirs(f"{path}/{new_dir_name}")
        else:
            raise Exception(f"Error, path={path} did not exist.")


if __name__ == "__main__":
    main = Plot_to_tex()
    main.example_create_a_table()
```

E   Appendix src/export_data/create_dynamic_diagrams.py

```python
1  """Generates .uml PlantUML files based on Python code.
2
3  Typically used for Gantt charts.
4  """
5  from src.export_data.plantuml_compile import (
6      compile_diagrams_in_dir_relative_to_root,
7  )
8  from src.export_data.plantuml_generate import
   ↪ generate_all_dynamic_diagrams
9  from src.export_data.plantuml_to_tex import export_diagrams_to_latex
10
11
12 def create_dynamic_diagrams(args, hd):
13     """Generates the dynamic diagrams."""
14     # Generate PlantUML diagrams dynamically (using code).
15     if args.dd:
16         generate_all_dynamic_diagrams(
17             f"{hd.path_to_export_data_from_root}/Diagrams/
               ↪ Dynamic_diagrams"
18         )
19
20         # Compile dynamically generated PlantUML diagrams to images.
21         compile_diagrams_in_dir_relative_to_root(
22             hd.await_compilation,
23             hd.gantt_extension,
24             hd.jar_path_relative_from_root,
25             hd.path_to_dynamic_gantts,
26             hd.verbose,
27         )
28
29         # Export dynamic PlantUML text files to LaTex.
30         export_diagrams_to_latex(
31             hd.path_to_dynamic_gantts,
32             hd.gantt_extension,
33             hd.diagram_output_dir_relative_to_root,
34         )
35
36         # Export dynamic PlantUML diagram images to LaTex.
37         export_diagrams_to_latex(
38             hd.path_to_dynamic_gantts,
39             hd.diagram_extension,
40             hd.diagram_output_dir_relative_to_root,
41         )
```

F Appendix src/export_data/create_static_diagrams.py

```python
1  """This file creates the diagrams that are written directly in plain
       ↪ .uml
2  files."""
3  from src.export_data.plantuml_compile import (
4      compile_diagrams_in_dir_relative_to_root,
5  )
6  from src.export_data.plantuml_to_tex import export_diagrams_to_latex
7
8
9  def create_static_diagrams(args, hd):
10     """Generates .png images of the static diagram .uml files.
11
12     :param args:
13     :param hd:
14     """
15     # PlantUML
16     if args.sd:
17         # Compile statically generated PlantUML diagrams to images.
18         compile_diagrams_in_dir_relative_to_root(
19             hd.await_compilation,
20             hd.gantt_extension,
21             hd.jar_path_relative_from_root,
22             hd.path_to_static_gantts,
23             hd.verbose,
24         )
25
26         # Export static PlantUML text files to LaTex.
27         export_diagrams_to_latex(
28             hd.path_to_static_gantts,
29             hd.gantt_extension,
30             hd.diagram_output_dir_relative_to_root,
31         )
32
33         # Export static PlantUML diagram images to LaTex.
34         export_diagrams_to_latex(
35             hd.path_to_static_gantts,
36             hd.diagram_extension,
37             hd.diagram_output_dir_relative_to_root,
38         )
```

G   Appendix src/export_data/export_data.py

```python
"""File used to export data to latex."""
from src.export_data.create_dynamic_diagrams import
    ↪ create_dynamic_diagrams
from src.export_data.create_static_diagrams import
    ↪ create_static_diagrams
from src.export_data.Hardcoded_data import Hardcoded_data
from src.export_data.latex_compile import compile_latex
from src.export_data.latex_export_code import export_code_to_latex


def export_data(args):
    """Parses the Python arguments that specify what should be
       ↪ compiled.

    :param args:
    """

    hd = Hardcoded_data()

    # Generating PlantUML diagrams
    create_dynamic_diagrams(args, hd)
    create_static_diagrams(args, hd)

    # Plotting graphs using Python code, and export them to latex.
    # Generate plots.
    # Export plots to LaTex.

    # Export code to LaTex.
    if args.c2l:
        # TODO: verify whether the latex/{project_name}/Appendices
            ↪ folder
        # exists before exporting.
        # TODO: verify whether the latex/{project_name}/Images folder
            ↪  exists
        # before exporting.
        export_code_to_latex(hd, False)
    elif args.ec2l:
        # TODO: verify whether the latex/{project_name}/Appendices
            ↪ folder
        # exists before exporting.
        # TODO: verify whether the latex/{project_name}/Images folder
            ↪  exists
        # before exporting.
        export_code_to_latex(hd, True)

    # Compile the accompanying LaTex report.
    if args.l:
        compile_latex(True, True)
        print("")
    print("\n\nDone exporting data.")
```

H   Appendix src/export_data/helper_bash_commands.py

```python
"""Code used to execute Bash commands from Python."""
import subprocess  # nosec

from src.export_data.plantuml_compile import \
    get_output_of_bash_command


def run_a_bash_command(await_compilation, bash_command, verbose):
    """Runs a bash commands from Python.

    :param await_compilation:
    :param bash_command:
    :param verbose:
    """
    if await_compilation:
        if verbose:
            subprocess.call(
                bash_command,
                shell=True,  # nosec
                stdout=subprocess.PIPE,
                stderr=subprocess.STDOUT,
            )
        else:
            # pylint: disable=E1129
            # pylint: disable=R1732
            subprocess.call(
                bash_command,
                shell=True,  # nosec
                stderr=subprocess.DEVNULL,
                stdout=subprocess.DEVNULL,
            )
    else:
        if verbose:
            # pylint: disable=R1732
            subprocess.Popen(
                bash_command,
                shell=True,  # nosec
                stdout=subprocess.PIPE,
                stderr=subprocess.STDOUT,
            )
        else:
            with subprocess.Popen(
                bash_command,
                shell=True,  # nosec
                stderr=subprocess.DEVNULL,
                stdout=subprocess.DEVNULL,
            ) as process:
                get_output_of_bash_command(process)
```

I Appendix src/export_data/helper_dir_file_edit.py

```python
"""A helper file that is used for directory and file editing."""
import glob
import os
import shutil


def file_contains(filepath, substring):
    """Returns True if a file exists. None otherwise.

    :param filepath:
    :param substring:
    """
    with open(filepath, encoding="utf-8") as f:
        return substring in f.read()


def get_dir_filelist_based_on_extension(dir_relative_to_root,
    extension):
    """

    :param dir_relative_to_root: A relative directory as
    seen from the root dir of this project.
    :param extension: File extension that is used/searched in this
        function.

    """
    selected_filenames = []
    # TODO: assert directory exists
    for filename in os.listdir(dir_relative_to_root):
        if filename.endswith(extension):
            selected_filenames.append(filename)
    return selected_filenames


def create_dir_relative_to_root_if_not_exists(dir_relative_to_root):
    """

        :param dir_relative_to_root: A relative directory as
    seen from the root dir of this project.

    """
    if not os.path.exists(dir_relative_to_root):
        os.makedirs(dir_relative_to_root)


def dir_relative_to_root_exists(dir_relative_to_root):
    """

        :param dir_relative_to_root: A relative directory as
    seen from the root dir of this project.

    """
    if not os.path.exists(dir_relative_to_root):
        return False
    if os.path.exists(dir_relative_to_root):
        return True
    raise Exception(
        f"Directory relative to root: {dir_relative_to_root}"
        + " did not exist, nor did it exist."
    )
```

```python
59
60
61  def get_all_files_in_dir_and_child_dirs(extension, path,
    ↪ excluded_files=None):
62      """Returns a list of the relative paths to all files within the
        ↪ some path
63      that match the given file extension. Also includes files in child
64      directories.
65
66      :param extension: File extension that is used/searched in this
        ↪ function.
67      The file extension of the file that is sought in the appendix
        ↪ line. Either
68      ".py" or ".pdf".
69      :param path: Absolute filepath in which files are being sought.
70      :param excluded_files: Default value = None) Files that will not
        ↪ be
71      included even if they are found.
72      """
73      filepaths = []
74      for r, _, f in os.walk(path):
75          for file in f:
76              if file.endswith(extension):
77                  if (excluded_files is None) or (
78                      (excluded_files is not None)
79                      and (file not in excluded_files)
80                  ):
81                      filepaths.append(r + "/" + file)
82      return filepaths
83
84
85  def get_filepaths_in_dir(extension, path, excluded_files=None):
86      """Returns a list of the relative paths to all files within the
        ↪ some path
87      that match the given file extension. Does not include files in
88      child_directories.
89
90      :param extension: File extension that is used/searched in this
        ↪ function.
91      The file extension of the file that is sought in the appendix
        ↪ line.
92      Either ".py" or ".pdf".
93      :param path: Absolute filepath in which files are being sought.
94      :param excluded_files: Default value = None) Files that will not
        ↪ be
95      included even if they are found.
96      """
97      filepaths = []
98      current_path = os.getcwd()
99      os.chdir(path)
100     for file in glob.glob(f"*.{extension}"):
101         print(file)
102         if (excluded_files is None) or (
103             (excluded_files is not None) and (file not in
                ↪ excluded_files)
104         ):
105             # Append normalised filepath e.g. collapses b/src/../d to
                ↪ b/d.
106             filepaths.append(os.path.normpath(f"{path}/{file}"))
107     os.chdir(current_path)
108     return filepaths
109
```

```python
110
111  def sort_filepaths_by_filename(filepaths):
112      """
113
114      :param filepaths:
115
116      """
117      # filepaths.sort(key = lambda x: x.split()[1])
118      filepaths.sort(key=lambda x: x[x.rfind("/") + 1 :])  # noqa: E203
119      for filepath in filepaths:
120          print(f"{filepath}")
121      return filepaths
122
123
124  def get_filename_from_dir(path):
125      """Returns a filename from an absolute path to a file.
126
127      :param path: path to a file of which the name is queried.
128      """
129      return path[path.rfind("/") + 1 :]  # noqa: E203
130
131
132  def delete_file_if_exists(filepath):
133      """
134
135      :param filepath:
136
137      """
138      try:
139          os.remove(filepath)
140      except OSError:
141          pass
142
143
144  def convert_filepath_to_filepath_from_root(filepath,
     ↪ normalised_root_path):
145      """
146
147      :param filepath:
148      :param normalised_root_path:
149
150      """
151      normalised_filepath = os.path.normpath(filepath)
152      filepath_relative_from_root = normalised_filepath[
153          len(normalised_root_path) :  # noqa: E203
154      ]
155      return filepath_relative_from_root
156
157
158  def append_lines_to_file(filepath, lines):
159      """
160
161      :param filepath:
162      :param lines:
163
164      """
165      with open(filepath, "a", encoding="utf-8") as the_file:
166          for line in lines:
167              the_file.write(f"{line}\n")
168
169
170  def append_line_to_file(filepath, line):
```

```python
        """

        :param filepath:
        :param line:

        """
        with open(filepath, "a", encoding="utf-8") as the_file:
            the_file.write(f"{line}\n")
            the_file.close()


def remove_all_auto_generated_appendices(hd):
        """

        :param hd:

        """

        # TODO: move identifier into hardcoded.
        all_appendix_files = get_all_files_in_dir_and_child_dirs(
            ".tex", hd.appendix_dir_from_root, excluded_files=None
        )
        for file in all_appendix_files:
            if "Auto_generated" in file:
                delete_file_if_exists(file)


def delete_dir_relative_to_root_if_not_exists(dir_relative_to_root):
        """

            :param dir_relative_to_root: A relative directory as
        seen from the root dir of this project.

        """
        if os.path.exists(dir_relative_to_root):
            # Remove directory and its content.
            shutil.rmtree(dir_relative_to_root)
```

J   Appendix src/export_data/helper_tex_editing.py

```python
"""Helper that modifies LaTex file to allow automatically including
    ↪ Pyathon
code as appendices."""
import os

from src.export_data.helper_dir_file_edit import (
    append_line_to_file,
    append_lines_to_file,
    convert_filepath_to_filepath_from_root,
    delete_file_if_exists,
    get_filename_from_dir,
)


def code_filepath_to_tex_appendix_filename(
    filename, _, is_project_code, is_export_code
):
    """

    :param filename:
    :param from_root:
    :param is_project_code:
    :param is_export_code:

    """

    # TODO: Include assert to verify filename ends at .py.
    # TODO: Include assert to verify filename doesn't end at .py
        ↪ anymore.
    filename_without_extension = os.path.splitext(filename)[0]

    # Create appendix filename identifier segment
    verify_input_code_type(is_export_code, is_project_code)
    if is_project_code:
        identifier = "Auto_generated_project_code_appendix_"
    elif is_export_code:
        identifier = "Auto_generated_export_code_appendix_"

    appendix_filename = f"{identifier}{filename_without_extension}"
    return appendix_filename


def verify_input_code_type(is_export_code, is_project_code):
    """

    :param is_export_code:
    :param is_project_code:

    """
    # Create appendix filename identifier segment
    if is_project_code and is_export_code:
        raise Exception(
            "Error, a file can't be both project code, and export
                ↪ code at"
            + " same time."
        )
    if not is_project_code and not is_export_code:
        raise Exception(
            "Error, don't know what to do with files that are neither
                ↪ project"
```

```python
57              + " code , nor export code ."
58          )
59
60
61  def tex_appendix_filename_to_inclusion_command(appendix_filename,
    ↪ from_root):
62      """
63
64      :param appendix_filename:
65      :param from_root:
66
67      """
68      # Create full appendix filename.
69      if from_root:
70          # Generate latex inclusion command for latex compilation from
              ↪  root dir.
71          appendix_inclusion_command = (
72              f"\\input{{latex/Appendices/{appendix_filename}.tex}} \\
                  ↪ newpage"
73          )
74          # \input{latex/Appendices/Auto_generated_py_App8.tex} \
              ↪ newpage
75      else:
76          # \input{Appendices/Auto_generated_py_App8.tex} \newpage
77          appendix_inclusion_command = (
78              f"\\input{{Appendices/{appendix_filename}.tex}} \\newpage
                  ↪ "
79          )
80      return appendix_inclusion_command
81
82
83  def create_appendix_filecontent(
84      latex_object_name, filename, filepath_from_root, from_root
85  ):
86      """
87
88      :param latex_object_name:
89      :param filename:
90      :param filepath_from_root:
91      :param from_root:
92
93      """
94      # Latex titles should escape underscores.
95      filepath_from_root_without_underscores = filepath_from_root.
          ↪ replace(
96          "_", r"\_"
97      )
98      lines = []
99      lines.append(
100         rf"\{latex_object_name}{{Appendix "
101         + rf"\{filepath_from_root_without_underscores}}}"
102         + rf"\label{{app:{filename}}}"
103     )
104     if from_root:
105         lines.append(rf"\pythonexternal{{latex/..{filepath_from_root
              ↪ }}}")
106     else:
107         lines.append(rf"\pythonexternal{{latex/..{filepath_from_root
              ↪ }}}")
108     return lines
109
110
```

```python
111  def create_appendix_manager_files(hd):
112      """
113
114      :param hd:
115
116      """
117      # Verify target directory exists.
118      if not os.path.exists(hd.appendix_dir_from_root):
119          raise Exception(
120              "Error, the Appendices directory was not found at:"
121              + f"{hd.appendix_dir_from_root}"
122          )
123
124      # Delete appendix manager files.
125      list(
126          map(
127              lambda x: delete_file_if_exists(f"{hd.
                  ↪ appendix_dir_from_root}{x}"),
128              hd.automatic_appendices_manager_filenames,
129          )
130      )
131
132      # Create new appendix_manager_files
133      list(
134          map(
135              # pylint: disable=R1732
136              lambda x: open(
137                  f"{hd.appendix_dir_from_root}{x}", "a", encoding="utf
                      ↪ -8"
138              ),
139              hd.automatic_appendices_manager_filenames,
140          )
141      )
142
143      # Ensure manual appendix_manager_files are created.
144      list(
145          map(
146              # pylint: disable=R1732
147              lambda x: open(
148                  f"{hd.appendix_dir_from_root}{x}", "a", encoding="utf
                      ↪ -8"
149              ),
150              hd.manual_appendices_manager_filenames,
151          )
152      )
153
154
155  # pylint: disable=R0913
156  def create_appendix_file(
157      hd,
158      filename,
159      filepath_from_root,
160      latex_object_name,
161      is_export_code,
162      is_project_code,
163  ):
164      """
165
166      :param hd:
167      :param filename:
168      :param filepath_from_root:
169      :param latex_object_name:
```

```python
170         :param is_export_code:
171         :param is_project_code:
172
173         """
174         verify_input_code_type(is_export_code, is_project_code)
175         filename_without_extension = os.path.splitext(filename)[0]
176         if is_project_code:
177             # Create the appendix for the case the latex is compiled from
                    ↪ root.
178             appendix_filepath = (
179                 f"{hd.appendix_dir_from_root}/Auto_generated_proj"
180                 + f"ect_code_appendix_{filename_without_extension}.tex"
181             )
182
183             # Append latex_filepath to appendix manager.
184             # append_lines_to_file(
185             #     f"{hd.appendix_dir_from_root}{hd.
                    ↪ project_code_appendices_filename}",
186             #     [tex_appendix_filepath_to_inclusion_command(
                    ↪ appendix_filepath)],
187             # )
188
189             # Get Appendix .tex content.
190             appendix_lines_from_root = create_appendix_filecontent(
191                 latex_object_name, filename, filepath_from_root, True
192             )
193
194             # Write appendix to .tex file.
195             append_lines_to_file(appendix_filepath,
                    ↪ appendix_lines_from_root)
196         elif is_export_code:
197             # Create the appendix for the case the latex is compiled from
                    ↪ root.
198             appendix_filepath = (
199                 f"{hd.appendix_dir_from_root}/Auto_generated_export"
200                 + f"_code_appendix_{filename_without_extension}.tex"
201             )
202
203             # Append latex_filepath to appendix manager.
204             # append_lines_to_file(
205             #     f"{hd.appendix_dir_from_root}{hd.
                    ↪ export_code_appendices_filename}",
206             #     [tex_appendix_filepath_to_inclusion_command(
                    ↪ appendix_filepath)],
207             # )
208
209             # Get Appendix .tex content.
210             appendix_lines_from_root = create_appendix_filecontent(
211                 latex_object_name, filename, filepath_from_root, True
212             )
213
214             # Write appendix to .tex file.
215             append_lines_to_file(appendix_filepath,
                    ↪ appendix_lines_from_root)
216     # TODO: verify files exist
217
218
219 def export_python_project_code(
220     hd, normalised_root_dir, python_project_code_filepaths
221 ):
222     """
223
```

```python
    :param hd:
    :param normalised_root_dir:
    :param python_project_code_filepaths:

    """
    is_project_code = True
    is_export_code = False
    from_root = False
    for filepath in python_project_code_filepaths:
        create_appendices(
            hd,
            filepath,
            normalised_root_dir,
            from_root,
            is_export_code,
            is_project_code,
        )
        create_appendices(
            hd,
            filepath,
            normalised_root_dir,
            True,
            is_export_code,
            is_project_code,
        )


def export_python_export_code(
    hd, normalised_root_dir, python_export_code_filepaths
):
    """

    :param hd:
    :param normalised_root_dir:
    :param python_export_code_filepaths:

    """
    is_project_code = False
    is_export_code = True
    from_root = False
    for filepath in python_export_code_filepaths:
        create_appendices(
            hd,
            filepath,
            normalised_root_dir,
            from_root,
            is_export_code,
            is_project_code,
        )
        create_appendices(
            hd,
            filepath,
            normalised_root_dir,
            True,
            is_export_code,
            is_project_code,
        )


# pylint: disable=R0913
def create_appendices(
    hd,
```

```python
        filepath,
        normalised_root_dir,
        from_root,
        is_export_code,
        is_project_code,
):
    """

    :param hd:
    :param filepath:
    :param normalised_root_dir:
    :param from_root:
    :param is_export_code:
    :param is_project_code:

    """
    # Get the filepath of a python file from the root dir of this
    #     ↪ project.
    filepath_from_root = convert_filepath_to_filepath_from_root(
        filepath, normalised_root_dir
    )
    print(f"from_root={from_root},filepath_from_root={
        ↪ filepath_from_root}")

    # Get the filename of a python filepath
    filename = get_filename_from_dir(filepath)

    # Get the filename for a latex appendix from a python filename.
    appendix_filename = code_filepath_to_tex_appendix_filename(
        filename, from_root, is_project_code, is_export_code
    )

    # Command to include the appendix in the appendices manager.
    appendix_inclusion_command =
        ↪ tex_appendix_filename_to_inclusion_command(
        appendix_filename, from_root
    )
    # if from_root:
    #     print(f'tex_appendix_filename_to_inclusion_command=
    # + {appendix_inclusion_command}')
    #     exit()

    append_appendix_to_appendix_managers(
        appendix_inclusion_command,
        from_root,
        hd,
        is_export_code,
        is_project_code,
    )

    # Create the appendix .tex file.
    # TODO: move "section" to hardcoded.
    if from_root:  # Appendix only contains files readable from root.
        create_appendix_file(
            hd,
            filename,
            filepath_from_root,
            "section",
            is_export_code,
            is_project_code,
        )
```

```python
345
346  def append_appendix_to_appendix_managers(
347      appendix_inclusion_command, from_root, hd, is_export_code,
         ↪ is_project_code
348  ):
349      """
350
351      :param appendix_inclusion_command:
352      :param from_root:
353      :param hd:
354      :param is_export_code:
355      :param is_project_code:
356
357      """
358      # Append the appendix .tex file to the appendix manager.
359      if is_project_code:
360          if from_root:
361              # print('from_root={from_root}Append to:
362              # +f"{hd.project_code_appendices_filename_from_root}')
363              append_line_to_file(
364                  f"{hd.appendix_dir_from_root}"
365                  + f"{hd.project_code_appendices_filename_from_root}",
366                  appendix_inclusion_command,
367              )
368          else:
369              # print(f'from_root={from_root}Append to:"
370              # +f"{hd.project_code_appendices_filename}')
371              append_line_to_file(
372                  f"{hd.appendix_dir_from_root}"
373                  + f"{hd.project_code_appendices_filename}",
374                  appendix_inclusion_command,
375              )
376
377      if is_export_code:
378          if from_root:
379              append_line_to_file(
380                  f"{hd.appendix_dir_from_root}"
381                  + f"{hd.export_code_appendices_filename_from_root}",
382                  appendix_inclusion_command,
383              )
384          else:
385              append_line_to_file(
386                  f"{hd.appendix_dir_from_root}"
387                  + f"{hd.export_code_appendices_filename}",
388                  appendix_inclusion_command,
389              )
```

K   Appendix src/export_data/helper_tex_reading.py

```python
"""Helper script to parse Latex files, to support including LaTex
appendices."""
from src.export_data.helper_dir_file_edit import file_contains


def verify_latex_supports_auto_generated_appendices(
    path_to_main_latex_file):
    """Ensures the Latex file supports including automatically
        appending code
    as appendices.

    :param path_to_main_latex_file:
    """
    # TODO: change verification to complete tex block(s) for
        appendices.
    # TODO: Also verify related boolean and if statement creations.
    determining_overleaf_home_line = (
        r"\def\overleafhome{/tmp}% change as appropriate"
    )
    begin_apendices_line = "\\begin{appendices}"
    print(f"determining_overleaf_home_line={
        determining_overleaf_home_line}")
    print(f"begin_apendices_line={begin_apendices_line}")

    if not file_contains(
        path_to_main_latex_file, determining_overleaf_home_line
    ):
        raise Exception(
            f"Error, {path_to_main_latex_file} does not contain:\n\n"
            + f"{determining_overleaf_home_line}\n\n so this Python
                code "
            + "cannot export the code as latex appendices."
        )
    if not file_contains(
        path_to_main_latex_file, determining_overleaf_home_line
    ):
        raise Exception(
            f"Error, {path_to_main_latex_file} does not contain:\n\n"
            + f"{begin_apendices_line}\n\n so this Python code cannot
                export"
            "the code as latex appendices."
        )
```

L Appendix src/export_data/latex_compile.py

```python
"""Compiles the latex report using the compile script."""


from src.export_data.helper_bash_commands import run_a_bash_command


def compile_latex(await_compilation, verbose):
    """Compiles the LaTex report of this project using its compile
        ↪ script.

    :param await_compilation: Make python wait until the PlantUML
        ↪ compilation
    is completed.
    :param project_name: The name of the project that is being
        ↪ executed/ran.
    :param verbose: True, ensures compilation output is printed to
        ↪ terminal,
    False means compilation is silent.

    Returns:
        Nothing.

    Raises:
        Nothing.
    """

    # Ensure compile script is runnable.
    bash_make_compile_script_runnable_command = (
        "chmod +x latex/compile_script.sh"
    )
    run_a_bash_command(
        await_compilation, bash_make_compile_script_runnable_command,
            ↪    verbose
    )

    # Run latex compilation script to compile latex project.
    bash_compilation_command = "latex/compile_script.sh"
    run_a_bash_command(await_compilation, bash_compilation_command,
        ↪ verbose)
```

```python
"""Exports code to latex appendices."""
import os

from src.export_data.helper_dir_file_edit import (
    get_all_files_in_dir_and_child_dirs,
    get_filepaths_in_dir,
    remove_all_auto_generated_appendices,
    sort_filepaths_by_filename,
)
from src.export_data.helper_tex_editing import (
    create_appendix_manager_files,
    export_python_export_code,
    export_python_project_code,
)
from src.export_data.helper_tex_reading import (
    verify_latex_supports_auto_generated_appendices,
)


def export_code_to_latex(hd, include_export_code):
    """This function exports the python files and compiled pdfs of
        ↪ jupiter
    notebooks into the latex of the same project number. First it
        ↪ scans which
    appendices (without code, without notebooks) are already manually
        ↪  included
    in the main latex code. Next, all appendices that contain the
        ↪ python code
    are either found or created in the following order: First, the
        ↪ __main__.py
    file is included, followed by the main.py file, followed by all
        ↪ python code
    files in alphabetic order. After this, all the pdfs of the
        ↪ compiled
    notebooks are added in alphabetic order of filename. This order
        ↪ of
    appendices is overwritten in the main tex file.

    :param main_latex_filename: Name of the main latex document of
        ↪ this project
    number.
    :param project_name: The name of the project that is being
        ↪ executed/ran.
    The number  indicating which project this code pertains to.
    """
    script_dir = get_script_dir()
    latex_dir = script_dir + "/../../latex/"
    path_to_main_latex_file = f"{latex_dir}{hd.main_latex_filename}"
    root_dir = script_dir + "/../../"
    normalised_root_dir = os.path.normpath(root_dir)
    src_dir = script_dir + "/../"

    # Verify the latex file supports auto-generated python appendices
        ↪ .
    verify_latex_supports_auto_generated_appendices(
        ↪ path_to_main_latex_file)

    # Get paths to files containing project python code.
    python_project_code_filepaths = get_filepaths_in_dir(
        "py", src_dir, ["__init__.py"]
```

```python
49          )

51          get_compiled_notebook_paths(script_dir)
52          print(f"python_project_code_filepaths={
              ↪ python_project_code_filepaths}")

54          # Get paths to the files containing the latex export code
55          if include_export_code:
56              python_export_code_filepaths = get_filepaths_in_dir(
57                  "py", script_dir, ["__init__.py"]
58              )

60          remove_all_auto_generated_appendices(hd)

62          # Create appendix file # ensure they are also deleted at the
              ↪ start of every
63          # run.
64          create_appendix_manager_files(hd)

66          # TODO: Sort main files.
67          export_python_project_code(
68              hd,
69              normalised_root_dir,
70              sort_filepaths_by_filename(python_project_code_filepaths),
71          )
72          if include_export_code:
73              export_python_export_code(
74                  hd,
75                  normalised_root_dir,
76                  sort_filepaths_by_filename(python_export_code_filepaths),
77              )


80  def get_compiled_notebook_paths(script_dir):
81      """Returns the list of jupiter notebook filepaths that were
          ↪ compiled
82      successfully and that are included in the same dias this script (
          ↪ the src
83      directory).

85      :param script_dir: absolute path of this file.
86      """
87      notebook_filepaths = get_all_files_in_dir_and_child_dirs(
88          ".ipynb", script_dir
89      )
90      compiled_notebook_filepaths = []

92      # check if the jupyter notebooks were compiled
93      for notebook_filepath in notebook_filepaths:

95          # swap file extension
96          notebook_filepath = notebook_filepath.replace(".ipynb", ".pdf
              ↪ ")

98          # check if file exists
99          if os.path.isfile(notebook_filepath):
100             compiled_notebook_filepaths.append(notebook_filepath)
101     return compiled_notebook_filepaths


104 def get_script_dir():
```

```
105    """returns the directory of this script regardless of from which
           ↪ level the
106    code is executed."""
107    return os.path.dirname(__file__)
```

```python
"""This script automatically compiles the text files representing a
  ↪ PlantUML.

# diagram into an actual figure.

# To compile locally manually:
# pip install plantuml
# export  PLANTUML_LIMIT_SIZE=8192
# java -jar plantuml.jar -verbose sequenceDiagram.txt
"""

import os
import subprocess  # nosec
from os.path import abspath

from src.export_data.helper_dir_file_edit import (
    get_dir_filelist_based_on_extension,
)
from src.export_data.plantuml_get_package import got_java_file


def compile_diagrams_in_dir_relative_to_root(
    await_compilation,
    extension,
    jar_path_relative_from_root,
    input_dir_relative_to_root,
    verbose,
):
    """Loops through the files in a directory and exports them to the
      ↪ latex.

    /Images directory.

    Args:
    :param await_compilation: Make python wait until the PlantUML
      ↪ compilation
    is completed. param extension: The filetype of the text file that
      ↪ is
    converted to image.
    :param jar_path_relative_from_root: The path as seen from root
      ↪ towards the
    PlantUML .jar file that compiles .uml files to .png files.
    :param verbose: True, ensures compilation output is printed to
      ↪ terminal,
    False means compilation is silent.
    :param extension: The file extension that is used/searched in
      ↪ this
    function.
    :param input_dir_relative_to_root: The directory as seen from
      ↪ root
    containing files that are modified in this function.

    Returns:
        Nothing

    Raises:
        Nothing
    """
    # Verify the PlantUML .jar file is gotten.
    got_java_file(jar_path_relative_from_root)
```

```python
53
54      diagram_text_filenames = get_dir_filelist_based_on_extension(
55          input_dir_relative_to_root, extension
56      )
57
58      for diagram_text_filename in diagram_text_filenames:
59          diagram_text_filepath_relative_from_root = (
60              f"{input_dir_relative_to_root}/{diagram_text_filename}"
61          )
62
63          execute_diagram_compilation_command(
64              await_compilation,
65              jar_path_relative_from_root,
66              diagram_text_filepath_relative_from_root,
67              verbose,
68          )
69
70
71  def execute_diagram_compilation_command(
72      await_compilation,
73      jar_path_relative_from_root,
74      relative_filepath_from_root,
75      verbose,
76  ):
77      """Compiles a .uml/text file containing a PlantUML diagram to a .
            ↪ png image
78      using the PlantUML .jar file.
79
80      Args:
81      :param await_compilation: Make python wait until the PlantUML
            ↪ compilation
82      is completed. param extension: The filetype of the text file that
            ↪  is
83      converted to image.
84      :param jar_path_relative_from_root: The path as seen from root
            ↪ towards the
85      PlantUML .jar file that compiles .uml files to .png files.
86      :param verbose: True, ensures compilation output is printed to
            ↪ terminal,
87      False means compilation is silent.
88      :param input_dir_relative_to_root: The directory as seen from
            ↪ root
89      containing files that are modified in this function.
90      Returns:
91          Nothing
92
93      Raises:
94          Nothing
95      """
96      # Verify the files required for compilation exist, and convert
            ↪ the paths
97      # into absolute filepaths.
98      (
99          abs_diagram_filepath,
100         abs_jar_path,
101     ) = assert_diagram_compilation_requirements(
102         jar_path_relative_from_root,
103         relative_filepath_from_root,
104     )
105
106     # Generate command to compile the PlantUML diagram locally.
107     print(
```

```python
        f"abs_jar_path={abs_jar_path},"
        + f" abs_diagram_filepath={abs_diagram_filepath}\n\n"
    )
    bash_diagram_compilation_command = (
        f"java -jar {abs_jar_path} -verbose {abs_diagram_filepath}"
    )
    print(
        f"bash_diagram_compilation_command={
            bash_diagram_compilation_command}"
    )
    # Generate global variable specifying max image width in pixels,
        in the
    # shell that compiles.
    os.environ["PLANTUML_LIMIT_SIZE"] = "16192"

    # Perform PlantUML compilation locally.
    if await_compilation:
        if verbose:
            subprocess.call(
                bash_diagram_compilation_command, shell=True  # nosec
                    B602
            )
        else:
            subprocess.call(
                bash_diagram_compilation_command,
                shell=True,  # nosec
                stderr=subprocess.DEVNULL,
                stdout=subprocess.DEVNULL,
            )  # nosec
    else:
        if verbose:
            with subprocess.Popen(
                bash_diagram_compilation_command,
                shell=True,  # nosec
                stdout=subprocess.PIPE,
                stderr=subprocess.STDOUT,
            ) as process:
                get_output_of_bash_command(process)
        else:
            with subprocess.Popen(
                bash_diagram_compilation_command,
                shell=True,  # nosec
                stderr=subprocess.DEVNULL,
                stdout=subprocess.DEVNULL,
            ) as process:
                get_output_of_bash_command(process)


def get_output_of_bash_command(process, verbose=True):
    """Returns the output of a bash command."""
    stdout, stderr = process.communicate()
    result = stdout.decode("utf-8")
    if verbose:
        print(f"result={result}")
        print(f"stderr={stderr}")
    return result


def assert_diagram_compilation_requirements(
    jar_path_relative_from_root,
    relative_filepath_from_root,
):
```

```python
167        """Asserts that the PlantUML .jar file used for compilation
           ↪ exists, and
168        that the diagram file with the .uml content for the diagram
           ↪ exists. Throws
169        an error if either of two is missing.

170
171        :param relative_filepath_from_root: Relative filepath as seen
           ↪ from root of
172        file that is used in this function.
173        :param output_dir_from_root: Relative directory as seen from root
           ↪ , to which
174        files are outputted.
175        :param jar_path_relative_from_root: The path as seen from root
           ↪ towards the
176        PlantUML .jar file that compiles .uml files to .png files.

177
178        Returns:
179            Nothing

180
181        Raises:
182            Exception if PlantUML .jar file used to compile the .uml to .
               ↪ png files
183            is missing.
184            Exception if the file with the .uml content is missing.
185        """
186        abs_diagram_filepath = abspath(relative_filepath_from_root)
187        abs_jar_path = abspath(jar_path_relative_from_root)
188        if os.path.isfile(abs_diagram_filepath):
189            if os.path.isfile(abs_jar_path):
190                return abs_diagram_filepath, abs_jar_path
191            raise Exception(
192                f"The input diagram file:{abs_diagram_filepath} doesn't
                   ↪ exist."
193            )
194        raise Exception(f"The input jar file:{abs_jar_path} does not
           ↪ exist.")
```

O Appendix src/export_data/plantuml_generate.py

```python
1  """This script generates PlantUML diagrams and outputs them as .uml
   ↪ files."""
2
3  import os
4  from os.path import abspath
5
6  from src.export_data.helper_dir_file_edit import (
7      create_dir_relative_to_root_if_not_exists,
8      dir_relative_to_root_exists,
9  )
10
11
12 def generate_all_dynamic_diagrams(output_dir_relative_to_root):
13     """Manages the generation of all the diagrams created in this
          ↪ file.
14
15     Args:
16     :param output_dir_relative_to_root: Relative path as seen from
          ↪ the root dir
17     of this project, to which modified files are outputted.
18
19     Returns:
20         Nothing
21
22     Raises:
23     """
24     # Create a example Gantt output file.
25     filename_one, lines_one = create_trivial_gantt("trivial_gantt.uml
          ↪ ")
26     output_diagram_text_file(
27         filename_one, lines_one, output_dir_relative_to_root
28     )
29
30     # Create another example Gantt output file.
31     filename_two, lines_two = create_trivial_gantt("
          ↪ another_trivial_gantt.uml")
32     output_diagram_text_file(
33         filename_two,
34         lines_two,
35         output_dir_relative_to_root,
36     )
37
38
39 def output_diagram_text_file(filename, lines,
       ↪ output_dir_relative_to_root):
40     """Gets the filename and lines of an PlantUML diagram, and writes
          ↪  these to
41     a file at the relative output path.
42
43     Args:
44     :param filename: The filename of the PlantUML Gantt file that is
          ↪ being
45     created.
46     :param lines: The lines of the Gantt chart PlantUML code that is
          ↪ being
47     written to file.
48     :param output_dir_relative_to_root: Relative path as seen from
          ↪ the root
49     dir of this project, to which modified files are outputted.
50
```

```python
    Returns:
        Nothing

    Raises:
        Exception if input file does not exist.
    """
    abs_filepath = abspath(f"{output_dir_relative_to_root}/{filename}
        ↪ ")

    # Ensure output directory is created.
    create_dir_relative_to_root_if_not_exists(
        ↪ output_dir_relative_to_root)
    if not dir_relative_to_root_exists(output_dir_relative_to_root):
        raise Exception(
            "Error, the output directory relative to root:"
            + f"{output_dir_relative_to_root} does not exist."
        )

    # Delete output file if it already exists.
    if os.path.exists(abs_filepath):
        os.remove(abs_filepath)

    # Write lines to file.
    with open(abs_filepath, "w", encoding="utf-8") as f:
        for line in lines:
            f.write(line)
        f.close()

    # Assert output file exists.
    if not os.path.isfile(abs_filepath):
        raise Exception(f"The input file:{abs_filepath} does not
            ↪ exist.")


def create_trivial_gantt(filename):
    """Creates a trivial Gantt chart.

    Args:
    :param filename: The filename of the PlantUML diagram file that
        ↪ is being
    created.

    Returns:
        The filename of the PlantUML diagram, and the lines of the
            ↪ uml content
        of the diagram

    Raises:
        Nothing
    """
    lines = []
    lines.append("@startuml\n")
    lines.append("[Prototype design] lasts 15 days\n")
    lines.append("[Test prototype] lasts 10 days\n")
    lines.append("\n")
    lines.append("Project starts 2020-07-01\n")
    lines.append("[Prototype design] starts 2020-07-01\n")
    lines.append("[Test prototype] starts 2020-07-16\n")
    lines.append("@enduml\n")
    return filename, lines
```

```python
108  def create_another_trivial_gantt(filename):
109      """Creates a trivial Gantt chart.
110
111      :param filename: The filename of the PlantUML Gantt file that is
          ↪ being
112      created.
113
114      Returns:
115          The filename of the PlantUML diagram, and the lines of the
              ↪ uml content
116          of the diagram
117
118      Raises:
119          Nothing
120      """
121      lines = []
122      lines.append("@startuml\n")
123      lines.append("[EXAMPLE SENTENCE] lasts 15 days\n")
124      lines.append("[Test prototype] lasts 10 days\n")
125      lines.append("\n")
126      lines.append("Project starts 2022-07-01\n")
127      lines.append("[Prototype design] starts 2022-07-01\n")
128      lines.append("[Test prototype] starts 2022-07-16\n")
129      lines.append("@enduml\n")
130
131      return filename, lines
```

P Appendix src/export_data/plantuml_get_package.py

```python
1  """Downloads the PlantUML package if it does not yet exist."""
2  import os
3  import subprocess  # nosec
4
5  import requests
6
7
8  def check_if_java_file_exists(relative_filepath):
9      """Safe check to see if file exists or not.
10
11     Args:
12     :param relative_filepath: Path as seen from root towards a file.
13
14     Returns:
15         True if a file exists.
16         False if a file does not exists.
17
18     Raises:
19         Nothing
20     """
21
22     return os.path.isfile(relative_filepath)
23
24
25 def got_java_file(relative_filepath):
26     """Asserts if PlantUML .jar file exists. Tries to download is one
        ↪    time if
27  it does not exist at the start of the function.
28
29     Args:
30     :param relative_filepath: Path as seen from root towards a file.
31
32     Returns:
33         True if a file exists.
34
35     Raises:
36         Exception if the PlantUML .jar file does not exist after
            ↪  downloading
37         it.
38     """
39     # Check if the jar file exists, curl it if not.
40     if not check_if_java_file_exists(relative_filepath):
41         # The java file is not found, curl it
42         request_file(
43             "https://sourceforge.net/projects/"
44             + "plantuml/files/plantuml.jar/download",
45             relative_filepath,
46         )
47     # Check if the jar file exists after curling it. Raise Exception
        ↪  if it is
48     # not found after curling.
49     if not check_if_java_file_exists(relative_filepath):
50         raise Exception(f"File:{relative_filepath} is not accessible"
            ↪  )
51     print("Got the PlangUML Java file.")
52     return True
53
54
55 def request_file(url, output_filepath):
56     """Downloads a file or file content.
```

```python
57
58        Args:
59        :param url: Url towards a file that will be downloaded.
60        :param relative_filepath: The path as seen from the root of this
              ↪ directory,
61        in which files are outputted.
62
63        Returns:
64            Nothing
65
66        Raises:
67            Nothing
68        """
69
70        # Request the file in the url
71        response = requests.get(url, timeout=20)  # seconds
72        with open(output_filepath, "wb", encoding="utf-8") as f:
73            f.write(response.content)
74
75
76    def run_bash_command(bashCommand):
77        """Unused method. TODO: verify it is unused and delete it.
78
79        :param bashCommand: A string containing a bash command that
80        can be executed.
81        """
82        # Verbose call.
83        # subprocess.Popen(bashCommand, shell=True)
84        # Silent call.
85        # subprocess.Popen(bashCommand, shell=True, stderr=subprocess.
              ↪ DEVNULL,
86        # stdout=subprocess.DEVNULL)
87
88        # Await completion:
89        # Verbose call.
90        subprocess.call(bashCommand, shell=True)  # nosec
91        # Silent call.
92        # subprocess.call(bashCommand, shell=True, stderr=subprocess.
              ↪ DEVNULL,
93        # stdout=subprocess.DEVNULL)
```

Q   Appendix src/export_data/plantuml_to_tex.py

```python
"""Exports the generated PlantUML diagrams to the latex Images
    ↪ directory."""
import os.path
import shutil

from src.export_data.helper_dir_file_edit import (
    create_dir_relative_to_root_if_not_exists,
    get_dir_filelist_based_on_extension,
)


def export_diagrams_to_latex(
    input_dir_relative_to_root, extension,
        ↪ output_dir_relative_to_root
):
    """Loops through the files in a directory and exports them to the
        ↪  latex.

    /Images directory.

    :param dir: The directory in which the Gantt charts are being
        ↪ searched.
    :param extension: The file extension that is used/searched in
        ↪ this
    function. The filetypes that are being exported.
    :param input_dir_relative_to_root: Relative path as seen from the
        ↪  root dir
    of this project, containing files that modified in this function.
    :param output_dir_relative_to_root: Relative path as seen from
        ↪ the root dir
    of this project, to which modified files are outputted.
    """
    diagram_filenames = get_dir_filelist_based_on_extension(
        input_dir_relative_to_root, extension
    )
    if len(diagram_filenames) > 0:
        # Ensure output directory is created.
        create_dir_relative_to_root_if_not_exists(
            ↪ output_dir_relative_to_root)

    for diagram_filename in diagram_filenames:
        diagram_filepath_relative_from_root = (
            f"{input_dir_relative_to_root}/{diagram_filename}"
        )

        export_gantt_to_latex(
            diagram_filepath_relative_from_root,
                ↪ output_dir_relative_to_root
        )


def export_gantt_to_latex(
    relative_filepath_from_root, output_dir_relative_to_root
):
    """Takes an input filepath and an output directory as input and
        ↪ copies the
    file towards the output directory.

    :param relative_filepath_from_root: param
        ↪ output_dir_relative_to_root:
```

```
50     :param output_dir_relative_to_root: Relative path as seen from
        ↪ the root dir
51     of this project, to which modified files are outputted.
52
53     Returns:
54         Nothing.
55
56     Raises:
57         Exception if the output directory does not exist.
58         Exception if the input file is not found.
59     """
60     if os.path.isfile(relative_filepath_from_root):
61         if os.path.isdir(output_dir_relative_to_root):
62             shutil.copy(
63                 relative_filepath_from_root,
                    ↪ output_dir_relative_to_root
64             )
65         else:
66             raise Exception(
67                 f"The output directory:{output_dir_relative_to_root}
                    ↪ does"
68                 + " not exist."
69             )
70     else:
71         raise Exception(
72             f"The input file:{relative_filepath_from_root} does not
                ↪ exist."
73         )
```