Roadmap: TruCol

A decentralised collaboration protocol for test-driven development

September 4, 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 ± 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.

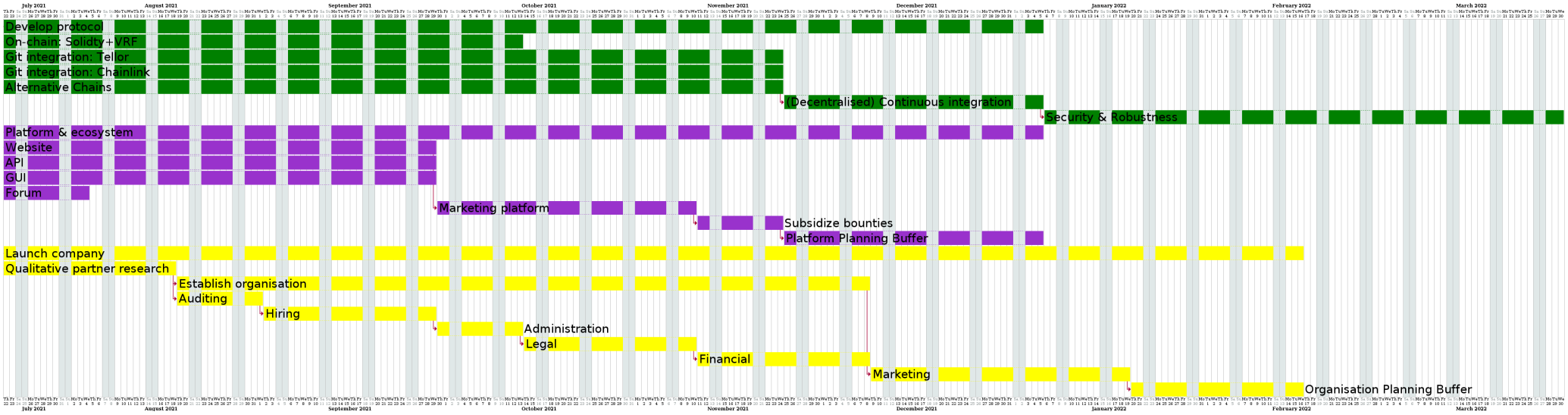

Figure 1: Gantt chart that is generated to plan the development of the TruCol company. (Source code in appendices, and on github.com/trucol/Roadmap).

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 ommited 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

```
"""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."""


# Project code imports.
from src.arg_parser import parse_cli_args
from src.export_data.export_data import export_data

# Export data import.


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

# 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}"
```

```
        self.jar_path_relative_from_root = (
            f"{self.path_to_export_data_from_root}"
            + f"/{self.plant_uml_java_filename}"
        )
        self.diagram_output_dir_relative_to_root = (
            f"latex/Images/{self.diagram_dir}"
        )

        # Path related variables
        self.append_export_code_to_latex = True
        self.path_to_dynamic_gantts = (
            f"{self.path_to_export_data_from_root}/"
            + f"{self.diagram_dir}/{self.dynamic_diagram_dir}"
        )
        self.path_to_static_gantts = (
            f"{self.path_to_export_data_from_root}/"
            + f"{self.diagram_dir}/{self.static_diagram_dir}"
        )
```

D Appendix src/export_data/Plot_to_tex.py

```
"""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:
```

```
        :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)):
```

```
            # overwrite linetypes to single type
            lineTypes[i] = "-"
            ax.plot(
                x,
                y_series[i, :],
                ls=lineTypes[i],
                label=label[i],
                fillstyle="none",
                c=cmap(i),
            )
            # color

        # configure plot layout
        plt.legend(loc=legendPosition)
        plt.xlabel(x_label)
        plt.ylabel(y_label)
        plt.savefig(
            os.path.dirname(__file__)
            + f"/../../../latex/{project_name}"
            + "/Images/"
            + filename
            + ".png"
        )

    # Generate random line colours
    # Source: https://stackoverflow.com/questions/14720331/
    # how-to-generate-random-colors-in-matplotlib
    def get_cmap(self, n, name="hsv"):
        """Returns a function that maps each index in 0, 1, ..., n-1
           ↪ to a
        distinct RGB color; the keyword argument name must be a
           ↪ standard mpl
        colormap name.

        :param n: param name:  (Default value = "hsv")
        :param name: Default value = "hsv")
        """
        return plt.cm.get_cmap(name, n)

    def generateLineTypes(self, y_series):
        """

        :param y_series:

        """
        # generate varying linetypes
        typeOfLines = list(lines.lineStyles.keys())

        while len(y_series) > len(typeOfLines):
            typeOfLines.append("-.")

        # remove void lines
        for i in range(0, len(y_series)):
            if typeOfLines[i] == "None":
                typeOfLines[i] = "-"
            if typeOfLines[i] == "":
                typeOfLines[i] = ":"
            if typeOfLines[i] == " ":
                typeOfLines[i] = "--"
        return typeOfLines

```

```
    # 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:

        """
```

```
236         self.create_target_dir_if_not_exists("latex/Images/", "graphs
               ↪ ")
237         some_plt.savefig(
238             "latex/Images/" + "graphs/" + filename + ".png", dpi=200
239         )
240 
241     def create_target_dir_if_not_exists(self, path, new_dir_name):
242         """
243 
244         :param path:
245         :param new_dir_name:
246 
247         """
248         if os.path.exists(path):
249             if not os.path.exists(f"{path}/{new_dir_name}"):
250                 os.makedirs(f"{path}/{new_dir_name}")
251         else:
252             raise Exception(f"Error, path={path} did not exist.")
253 
254 
255 if __name__ == "__main__":
256     main = Plot_to_tex()
257     main.example_create_a_table()
```

E Appendix src/export_data/create_dynamic_diagrams.py

```python
"""Generates .uml PlantUML files based on Python code.

Typically used for Gantt charts.
"""
from src.export_data.plantuml_compile import (
    compile_diagrams_in_dir_relative_to_root,
)
from src.export_data.plantuml_generate import
    ↪ generate_all_dynamic_diagrams
from src.export_data.plantuml_to_tex import export_diagrams_to_latex


def create_dynamic_diagrams(args, hd):
    """Generates the dynamic diagrams."""
    # Generate PlantUML diagrams dynamically (using code).
    if args.dd:
        generate_all_dynamic_diagrams(
            f"{hd.path_to_export_data_from_root}/Diagrams/
                ↪ Dynamic_diagrams"
        )

        # Compile dynamically generated PlantUML diagrams to images.
        compile_diagrams_in_dir_relative_to_root(
            hd.await_compilation,
            hd.gantt_extension,
            hd.jar_path_relative_from_root,
            hd.path_to_dynamic_gantts,
            hd.verbose,
        )

        # Export dynamic PlantUML text files to LaTex.
        export_diagrams_to_latex(
            hd.path_to_dynamic_gantts,
            hd.gantt_extension,
            hd.diagram_output_dir_relative_to_root,
        )

        # Export dynamic PlantUML diagram images to LaTex.
        export_diagrams_to_latex(
            hd.path_to_dynamic_gantts,
            hd.diagram_extension,
            hd.diagram_output_dir_relative_to_root,
        )
```

F Appendix src/export_data/create_static_diagrams.py

```python
"""This file creates the diagrams that are written directly in plain .uml
files."""
from src.export_data.plantuml_compile import (
    compile_diagrams_in_dir_relative_to_root,
)
from src.export_data.plantuml_to_tex import export_diagrams_to_latex


def create_static_diagrams(args, hd):
    """Generates .png images of the static diagram .uml files.

    :param args:
    :param hd:
    """
    # PlantUML
    if args.sd:
        # Compile statically generated PlantUML diagrams to images.
        compile_diagrams_in_dir_relative_to_root(
            hd.await_compilation,
            hd.gantt_extension,
            hd.jar_path_relative_from_root,
            hd.path_to_static_gantts,
            hd.verbose,
        )

        # Export static PlantUML text files to LaTex.
        export_diagrams_to_latex(
            hd.path_to_static_gantts,
            hd.gantt_extension,
            hd.diagram_output_dir_relative_to_root,
        )

        # Export static PlantUML diagram images to LaTex.
        export_diagrams_to_latex(
            hd.path_to_static_gantts,
            hd.diagram_extension,
            hd.diagram_output_dir_relative_to_root,
        )
```

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."
    )
```

```
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
```

```
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):
```

```
    """

    :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"
```

```
            + " code, nor export code."
        )


def tex_appendix_filename_to_inclusion_command(appendix_filename,
    ↪ from_root):
    """

    :param appendix_filename:
    :param from_root:

    """
    # Create full appendix filename.
    if from_root:
        # Generate latex inclusion command for latex compilation from
            ↪  root dir.
        appendix_inclusion_command = (
            f"\\input{{latex/Appendices/{appendix_filename}.tex}} \\
                ↪ newpage"
        )
        # \input{latex/Appendices/Auto_generated_py_App8.tex} \
            ↪ newpage
    else:
        # \input{Appendices/Auto_generated_py_App8.tex} \newpage
        appendix_inclusion_command = (
            f"\\input{{Appendices/{appendix_filename}.tex}} \\newpage
                ↪ "
        )
    return appendix_inclusion_command


def create_appendix_filecontent(
    latex_object_name, filename, filepath_from_root, from_root
):
    """

    :param latex_object_name:
    :param filename:
    :param filepath_from_root:
    :param from_root:

    """
    # Latex titles should escape underscores.
    filepath_from_root_without_underscores = filepath_from_root.
        ↪ replace(
        "_", r"\_"
    )
    lines = []
    lines.append(
        rf"\{latex_object_name}{{Appendix "
        + rf"\{filepath_from_root_without_underscores}}}"
        + rf"\label{{app:{filename}}}"
    )
    if from_root:
        lines.append(rf"\pythonexternal{{latex/..{filepath_from_root
            ↪ }}}")
    else:
        lines.append(rf"\pythonexternal{{latex/..{filepath_from_root
            ↪ }}}")
    return lines


```

```
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:
```

```
    :param is_export_code:
    :param is_project_code:

    """
    verify_input_code_type(is_export_code, is_project_code)
    filename_without_extension = os.path.splitext(filename)[0]
    if is_project_code:
        # Create the appendix for the case the latex is compiled from root.
        appendix_filepath = (
            f"{hd.appendix_dir_from_root}/Auto_generated_proj"
            + f"ect_code_appendix_{filename_without_extension}.tex"
        )

        # Append latex_filepath to appendix manager.
        # append_lines_to_file(
        #     f"{hd.appendix_dir_from_root}{hd.project_code_appendices_filename}",
        #     [tex_appendix_filepath_to_inclusion_command(appendix_filepath)],
        # )

        # Get Appendix .tex content.
        appendix_lines_from_root = create_appendix_filecontent(
            latex_object_name, filename, filepath_from_root, True
        )

        # Write appendix to .tex file.
        append_lines_to_file(appendix_filepath, appendix_lines_from_root)
    elif is_export_code:
        # Create the appendix for the case the latex is compiled from root.
        appendix_filepath = (
            f"{hd.appendix_dir_from_root}/Auto_generated_export"
            + f"_code_appendix_{filename_without_extension}.tex"
        )

        # Append latex_filepath to appendix manager.
        # append_lines_to_file(
        #     f"{hd.appendix_dir_from_root}{hd.export_code_appendices_filename}",
        #     [tex_appendix_filepath_to_inclusion_command(appendix_filepath)],
        # )

        # Get Appendix .tex content.
        appendix_lines_from_root = create_appendix_filecontent(
            latex_object_name, filename, filepath_from_root, True
        )

        # Write appendix to .tex file.
        append_lines_to_file(appendix_filepath, appendix_lines_from_root)
    # TODO: verify files exist


def export_python_project_code(
    hd, normalised_root_dir, python_project_code_filepaths
):
    """

```

```
    :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,
```

```
    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,
        )

```

```
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)
```

M Appendix src/export_data/latex_export_code.py

```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"]
```

```
    )

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

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

    remove_all_auto_generated_appendices(hd)

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

    # TODO: Sort main files.
    export_python_project_code(
        hd,
        normalised_root_dir,
        sort_filepaths_by_filename(python_project_code_filepaths),
    )
    if include_export_code:
        export_python_export_code(
            hd,
            normalised_root_dir,
            sort_filepaths_by_filename(python_export_code_filepaths),
        )


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

    :param script_dir: absolute path of this file.
    """
    notebook_filepaths = get_all_files_in_dir_and_child_dirs(
        ".ipynb", script_dir
    )
    compiled_notebook_filepaths = []

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

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

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


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__)
```

N Appendix src/export_data/plantuml_compile.py

```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)
```

```
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(
```

```
        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,
):
```

```
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
"""This script generates PlantUML diagrams and outputs them as .uml files."""

import os
from os.path import abspath

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


def generate_all_dynamic_diagrams(output_dir_relative_to_root):
    """Manages the generation of all the diagrams created in this file.

    Args:
    :param output_dir_relative_to_root: Relative path as seen from the root dir
    of this project, to which modified files are outputted.

    Returns:
        Nothing

    Raises:
    """
    # Create a example Gantt output file.
    filename_one, lines_one = create_trivial_gantt("trivial_gantt.uml")
    output_diagram_text_file(
        filename_one, lines_one, output_dir_relative_to_root
    )

    # Create another example Gantt output file.
    filename_two, lines_two = create_trivial_gantt("another_trivial_gantt.uml")
    output_diagram_text_file(
        filename_two,
        lines_two,
        output_dir_relative_to_root,
    )


def output_diagram_text_file(filename, lines, output_dir_relative_to_root):
    """Gets the filename and lines of an PlantUML diagram, and writes these to
    a file at the relative output path.

    Args:
    :param filename: The filename of the PlantUML Gantt file that is being
    created.
    :param lines: The lines of the Gantt chart PlantUML code that is being
    written to file.
    :param output_dir_relative_to_root: Relative path as seen from the root
    dir of this project, to which modified files are outputted.

```

```
    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
```

```
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
"""Downloads the PlantUML package if it does not yet exist."""
import os
import subprocess  # nosec

import requests


def check_if_java_file_exists(relative_filepath):
    """Safe check to see if file exists or not.

    Args:
    :param relative_filepath: Path as seen from root towards a file.

    Returns:
        True if a file exists.
        False if a file does not exists.

    Raises:
        Nothing
    """

    return os.path.isfile(relative_filepath)


def got_java_file(relative_filepath):
    """Asserts if PlantUML .jar file exists. Tries to download is one time if
    it does not exist at the start of the function.

    Args:
    :param relative_filepath: Path as seen from root towards a file.

    Returns:
        True if a file exists.

    Raises:
        Exception if the PlantUML .jar file does not exist after downloading
        it.
    """
    # Check if the jar file exists, curl it if not.
    if not check_if_java_file_exists(relative_filepath):
        # The java file is not found, curl it
        request_file(
            "https://sourceforge.net/projects/"
            + "plantuml/files/plantuml.jar/download",
            relative_filepath,
        )
    # Check if the jar file exists after curling it. Raise Exception if it is
    # not found after curling.
    if not check_if_java_file_exists(relative_filepath):
        raise Exception(f"File:{relative_filepath} is not accessible" )
    print("Got the PlangUML Java file.")
    return True


def request_file(url, output_filepath):
    """Downloads a file or file content.
```

```
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

```
"""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:
```

```
    :param output_dir_relative_to_root: Relative path as seen from the root dir
    of this project, to which modified files are outputted.

    Returns:
        Nothing.

    Raises:
        Exception if the output directory does not exist.
        Exception if the input file is not found.
    """
    if os.path.isfile(relative_filepath_from_root):
        if os.path.isdir(output_dir_relative_to_root):
            shutil.copy(
                relative_filepath_from_root, output_dir_relative_to_root
            )
        else:
            raise Exception(
                f"The output directory:{output_dir_relative_to_root} does"
                + " not exist."
            )
    else:
        raise Exception(
            f"The input file:{relative_filepath_from_root} does not exist."
        )
```