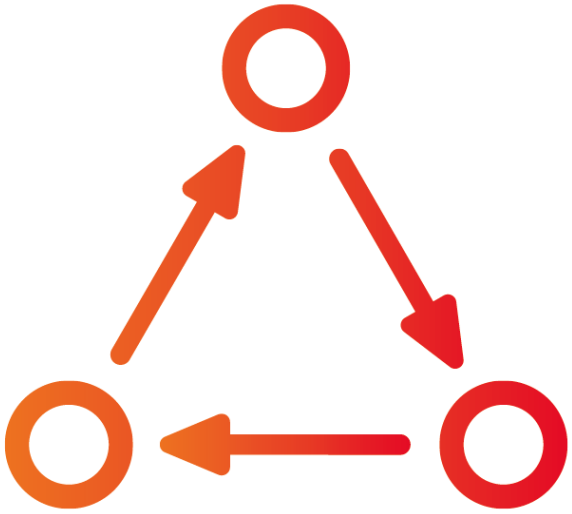
# Tactalyse-2

# Design Document



**Sangrok Lee (S3279480)**

**Bianca Raețchi (S4755138)**

**Matteo Gennaro Giordano (S5494974)**

**Mikko Brandon (S3781356)**

2023

RUG

# Contents



# Glossary

| Term | Definition |
|---|---|
| (The) service | An abstraction of an application, particularly the PDF generator API or the graph generator API. |
| (The) system | An abstraction of multiple services combined, that together form a greater application. |
| Graphs/Plots | Radar charts and line plots containing player statistics. |
| Coaches | Football coaches of professional football clubs. |
| Players | The football players the coaches request Tactalyse's service for. |
| (The) client | Tactalyse's owner (or Tactalyse's clients if specified) |

# 1. Introduction

Tactalyse is an organization that aims to provide data reports about football players. The football players' match data is from customers and the data report contains statistics and graphs based on these football players' match data. In this project, our main task is to receive data from the frontend service, create related graphs and plots based on the received data, and create data reports with the target football player's basic information and the graphs.
In this document, the architecture and design of our project are described. We provide overviews that range from high-level to low-level details. Since the client wants to develop and maintain the project further, most of our design choices are based on the client's familiarity.

# 2. Architectural Requirements

This section describes the architectural requirements the client imposed on us. These will not be functional requirements as described in the requirements documents, but rather requirements that significantly affect the architectural design of the project.

- **Python:** The client wants us to write a backend for the app in Python. The existing code was written as such, so there is some familiarity with the language within Tactalyse. Since the code is going to be maintained by the company after we deliver it to them, it is important that the coding staff understands how it works. Writing the code in Python facilitates this.
- **Excel data:** To create the PDF reports, we need to process football data into a graphic format. The data comes from a service named Wyscout, which provides it in Excel sheets. Due to financial constraints, it is not possible for the client to gain access to an API that would obtain these files directly from Wyscout. As such, our design needs to allow for the input of Excel sheets, which will then be processed further. Since our project begins by extracting target player's information from this excel data, this architectural requirement is a very important part of our project.

# 3. Technology Stack

This section describes the technologies we used to best fit the client's needs, and our motivation for their selection.

## 3.1 Programming Language

As explained in section Architectural Requirements, since Python was the programming language that our clients wanted from us for future development and maintenance, our main programming language is Python.

## 3.2 Framework

Although the client suggested we work with Flask for our backend framework, the client gave us freedom to choose a framework and they wanted us to have a rational reason for our choice. After the research about possible frameworks, we chose to use Flask, as well as to motivate it further.

When it comes to Python frameworks, the most popular choices are Flask and Django. To illustrate our decision-making process, we will discuss the similarities and differences between the two frameworks.

Firstly, since the project needs to be developed and maintained by Tactalyse in the future, we believe it is important that the framework we use should be **intuitive** and **easy to understand**. It allows for greater maintainability by employees of Tactalyse in the future.

Flask is a microframework, meaning it has minimal built-in functionality. Django is a full-stack web framework. Its philosophy is “batteries included”; it has as much functionality as deemed required for most web applications. Although this allows for more variety “out the box”, it makes it harder to comprehend all functionality. As such, it is easier for someone unfamiliar with the framework to wrap their head around Flask than it would be for Django.

As mentioned, Django has more built-in functionality. However, for the purposes of our project, a lot of the built-in functionality is not required. Furthermore, if the client wishes to develop the code further to include more technologies and functionality in the future, it is not the case that it would be impossible to implement in Flask. In fact, due to Django’s rigid nature, it might be easier in some cases to maintain new technologies in Flask. Therefore, we believe Flask to be the right choice regarding its fit to the currently planned **required functionality**.

Finally, the company’s employees have some **existing experience** using Flask. If we were to use Django for our project, it would require more effort on Tactalyse’s part. We want to keep future effort and development time to a minimum. With this in mind, Flask would be the logical choice.

All in all, we believe **Flask** to be the right choice for our project’s web framework.

## 3.3 Data

Our software takes data from two sources: a data file containing information about a football league’s players, and a data file containing match information about a specific player in that league. Both of these files are uploaded in .xlsx format, i.e. Excel worksheets. As mentioned in the architectural requirements, the files originate from a service named Wyscout. Clients of Tactalyse share these files with Tactalyse employees when requesting a PDF, after which an employee sends the files to our backend. The input data format was not negotiable. As such, we stuck with Excel worksheets. In figure 1, we provide a brief diagram of this input data. (The diagram does not provide all parameters in the actual data, but provides relatively important stats in each data file.)


Figure 1: Brief diagram of the input data

The data files from Wyscout are constantly updated. This makes it so that it is unlikely for two uploaded data files to be the same in terms of the data they contain. As such, the use of a database for data preprocessing was deemed unnecessary. In discussion with the client, we found that if a database were to be used, it would be to provide read-only access to previously uploaded data for our client. However, due to the fact that it would not directly contribute to the PDF generation process, we decided to leave the potential use of a database for the future.

## 3.4 Libraries

In this section, we describe the reasons why we chose the following libraries for our project. Some libraries are used by the original project, but there are also new libraries.

### 3.4.1 Excel Data Processing

For processing the input football data Excel files, we chose to use the Pandas library. A big reason we chose Pandas was that the existing code used Pandas to process raw data. As we described in the introduction section, the client wanted continuous code maintenance even after

we completed the project. Therefore, we chose Pandas, which the client is already familiar with. Moreover, Pandas provides faster data processing speed than Excel. Since one of the most important requirements of clients is to speed up the time to create data reports, data processing speed is a very important part.
There is another option for data preprocessing: Polars. Although Polars is optimized for processing large amounts of data with fast speed, our project does not need to preprocess large data which means that the efficiency of using Polars instead of Pandas is not that high. Therefore, We believe that Pandas does not differ much in performance from Polars and can meet the client's further maintenance requirements.

### 3.4.2 Graphs

We had two options for generating graphs/plots : Seaborn and Matplotlib. We decided to use Seaborn for generating line, bar, scatter plots, due to the advice from the client that Seaborn has easy-to-use functions to implement simple graphs. However, we also used Matplotlib, because Seaborn does not have a feature for generating radar charts. To further motivate our choice, we detail differences between the libraries:

- **Seaborn**

Seaborn is a Python data visualization library based on matplotlib. Seaborn allows a concise but limited approach to quickly visualizing datasets with nicer-looking style defaults than Matplotlib. Therefore, it is easy to write concise code and generate more plots in less time. Since we need to work on increasing the efficiency of our code and reducing the time it takes to produce results, we adopted seaborn as our main graph generation library.

- **Matplotlib**

Matplotlib is a base library for Seaborn, which means we need to use this library for the Seaborn functionalities. Furthermore, Seaborn has no standard option for radar charts, which we need to include in our document per the requirements. As such, we will use matplotlib to create these plots.

### 3.4.3 PDF Generation

For reasons similar to Pandas, since the existing code generates data reports with the pyFPDF library, we were urged to use this library in our code. To evaluate whether there were better options, we considered two alternatives: PDFlib and TCPDF. However, both of these libraries are slower than pyFPDF. Since this was the biggest concern for the client when it came to the PDF generation process, we decided to stick with pyFPDF. The library has all the features we need in our project, including image support, which is the most important part for the data reports.
The original pyFPDF release that is out as of the development of our application is quite slow. Generating a two page PDF with just two PNGs included took roughly 27 seconds. Since the client’s main requirement was increased speed of the application, we decided to use fpdf2, a fork of the original pyFPDF release. It uses a byte array buffer, which speeds up the output of a pdf file significantly. Specifics on why this is faster can be found here.

# 4. Architectural Overview

This section describes the architectural decisions we made to guide the coding process, and the overall structure of the application. For better comprehension, we present the following conceptual diagram of the data flow:

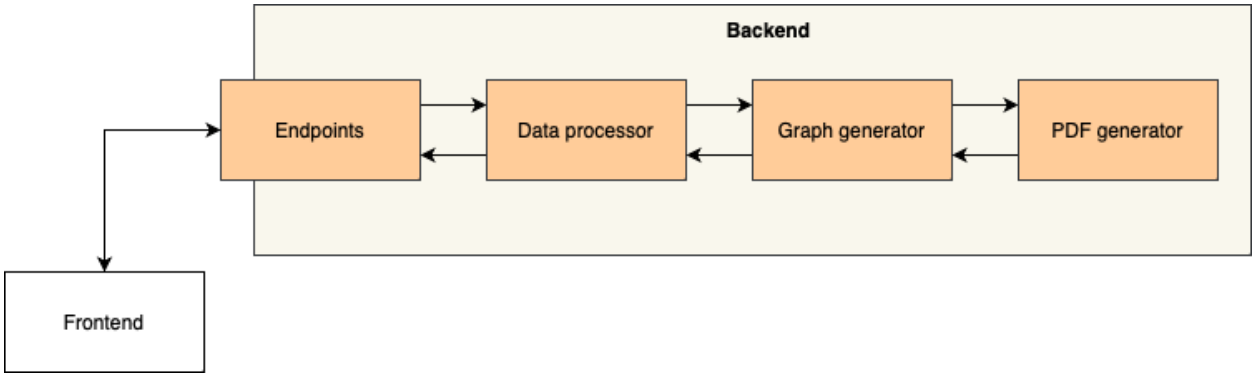



Figure 2: conceptual diagram

Some labels in this diagram need clarification.

- **Frontend**: Divided into two parts, one is web browser for data report, the other part is bot service for graph/plots generation.
  Web browser which is auth by Tactalyse employees. The employees use this for generating and accepting data reports. For this, the employees send Excel data and the name of the target player.

  Bot which is used by discord users. When users type in input in the discord server, graph API gets the input and generates related plots. The generated plots are sent to the discord bot and also twitter bot.
- **Data processor**: Using pandas to process raw data from the Frontend.
- **Graph generator**: generating graphs/plots based on the processed data.
- **PDF generator**: generating data reports by properly arranging the generated graphs/plots from Graph generator.

## 4.1 Design Patterns

**REST API**
We decided to create an API to allow access to our services from any other component in the greater system. It makes our services more flexible than if we were to deliver them as Python scripts that would be plugged into the system wherever necessary. As such, we adhered to REST API design, facilitated by Flask. This entails the use of HTTP verbs and response codes.

**MVC**

To guide the coding process and ensure best practices, we decided to use the Model-View-Controller (MVC) pattern. We deemed it appropriate considering the functionality and structure of our software system.

- **Model**

Per definition, the Model contains all core functionality and data access code. In our services, this entails data extraction/manipulation, graph generation, and PDF generation.

- **Controller**

The controller serves as an interface between the model and the view. The view signals the controller that something has changed. The controller makes the appropriate call to the model's functions according to the information received from the view. The model processes the passed information, and returns it to the controller. More on this can be found in the Module View and Execution View sections.

- **View**

The view is the user interface of the application. It is updated through the controller. Neither of our APIs contain view components. It could be argued that the PDF and graph generator modules could act as the "View", since they do generate some sort of visual data depending on what is signaled from the controller, in the form of graphs and PDFs. However, this visual data is not displayed or even really changed based on commands; they are generated and manipulated as data objects. As such, we still view these modules as Model components. The frontend website, Tactalyse's Discord server, and Tactalyse's Twitter page act as the Views for our applications.

**Object Oriented Design Patterns**

We employed the **abstract factory** design pattern in the graph generator module. We believed it to be a good fit due to the fact that we plan on allowing for many different kinds of plots to be defined. To have a good overview of which plots can be instantiated, and to keep the specifics of the instantiation of plot objects under the hood, we believed having a common interface for each plot super-class would be beneficial.

# 4.2 API Specification

This section details the functionality of our APIs, and their endpoints.

**Maturity level:** The APIs function at maturity level 3. They make use of HTTP (level 0), they use uniform resource identifiers (level 1), they use HTTP verbs (level 2), and they use content negotiation in the form of allowed content types (level 3).

## 4.2.1 PDF Generator

In this subsection, we provide the parameters used in the endpoints and endpoints specification.

**Parameters:**

- **league-file**

    - .xlsx format data file containing data about players in the league the file represents
    - In query
- **player-file**
    - .xlsx format data file containing match data of the player the file represents
    - In query
- **compare-file**
    - .xlsx format data file containing match data of the player to compare to that the file represents
    - In query
- **player-name**
    - String representing the name of the player whose match data is contained in **player-file**
    - In query
- **league-name**
    - String representing the name of the football league that the player whose match data is contained in **player-file** plays in
    - In query
- **compare-name**
    - String representing the name of the player whose match data is contained in **compare-file**
    - In query
- **start-date**
    - String representing the starting date of Tactalyse's services for the specified player in YYYY-MM-DD format
    - In query
- **end-date**
    - String representing the ending date of Tactalyse's services for the specified player in YYYY-MM-DD format
    - In query
- **player-image**
    - Image to be used for the main player in the introductory page of the report
    - In query
- **compare-image**
    - Image to be used for the player to compare to in the introductory page of the report
    - In query

### Endpoints

Here, each endpoint is listed with its HTTP methods. Required parameters are written in bold.

- POST /pdf
    - Retrieves a PDF based on passed parameters

- Parameters: **league-file, player-file, player-name,** league-name, compare-name, compare-file, start-date, end-date, player-image, compare-image
- Responses:
  - ➢ 200
    - Case: PDF report successfully created.
    - Body: PDF report in byte format.
    - Content-Type: application/pdf
  - ➢ 400
    - Case: Bad request (files did not match what was expected)
  - ➢ 5XX
    - Case: Server error

### 4.2.2 Graph Generator

In this subsection, we provide the parameters used in the endpoints and endpoints specification.

**Parameters:**

- **graph-type**
  - String representing the type of graph that should be randomized for the random endpoint. Currently supports ‘line’ and ‘radar’.
  - In query
- **league**
  - String representing the name of the league to use for radar graphs. The league must exist in the set of names of the local league files.
  - In query
- **player**
  - String representing the name of the main player to graph stats for. For line graphs, the name must exist in the set of names of the local player files. For radar graphs, the name must exist in the file of the passed league.
  - In query
- **compare**
  - String representing the name of the player to compare to in the generated graph. For line graphs, the name must exist in the set of names of the local player files. For radar graphs, the name must exist in the file of the passed league.
  - In query
- **stat**
  - String representing the stat to graph for line graphs. The stat name must exist in the set of stats contained in the player files.
- **start-date**
  - String representing the starting date of Tactalyse’s services for the specified player in YYYY-MM-DD format
  - In query
- **end-date**

    - String representing the ending date of Tactalyse's services for the specified player in YYYY-MM-DD format
    - In query

### Endpoints

Here, each endpoint is listed with its HTTP methods. Required parameters are written in bold.

- POST /graph
    - Retrieves a randomized graph. Allows for passing of additional parameters for anything the user does not want randomized.
    - Parameters: graph-type, league, player, start-date, end-date
    - Responses:
        - ➢ 200
            - Case: Graph successfully created.
            - Body: Random graph in byte format.
            - Content-Type: image/png
        - ➢ 400
            - Case: Bad request (invalid JSON payload)
        - ➢ 5XX
            - Case: Server error
- POST /graph/line
    - Retrieves a line graph based on passed parameters. Parameters that are not passed are either omitted, or randomized.
    - Parameters: player, compare, stat, start-date, end-date
    - Responses:
        - ➢ 200
            - Case: Graph successfully created.
            - Body: Line graph in byte format.
            - Content-Type: image/png
        - ➢ 400
            - Case: Bad request (invalid JSON payload)
        - ➢ 5XX
            - Case: Server error
- POST /graph/radar
    - Retrieves a radar graph based on passed parameters. Parameters that are not passed are either omitted, or randomized.
    - Parameters: league, player, compare
    - Responses:
        - ➢ 200
            - Case: Graph successfully created.
            - Body: Radar graph in byte format.
            - Content-Type: image/png
        - ➢ 400
            - Case: Bad request (invalid JSON payload)

➢ 5XX
- Case: Server error

## 4.3 Conceptual View

This section gives a conceptual overview of how our services connect to the client's systems. As detailed in the requirements document, we were involved in the development of two greater systems. Development of these systems had been divided over two teams: Tactalyse-1, and us, Tactalyse-2. Tactalyse-1 was responsible for the PDF report system's entire frontend, as well as backend functionality to do with the Twitter and Discord bots. We were responsible for the backend of the PDF report generator, and the graph generator used by the bots. As such, our services can be seen as an extension of Tactalyse-1's frontend and bot services. The component diagrams show high-level designs of the system, with each team's components divided through a border with the team's name. Arrows roughly indicate dependencies between components.

### 4.3.1 PDF Report System

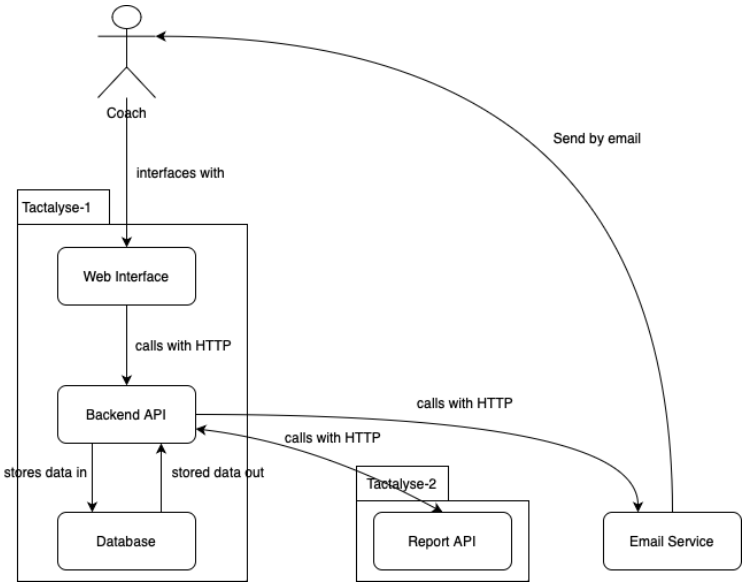



Figure 3: Diagram of the PDF report system architecture

### 4.3.2 Discord/Twitter Bot System

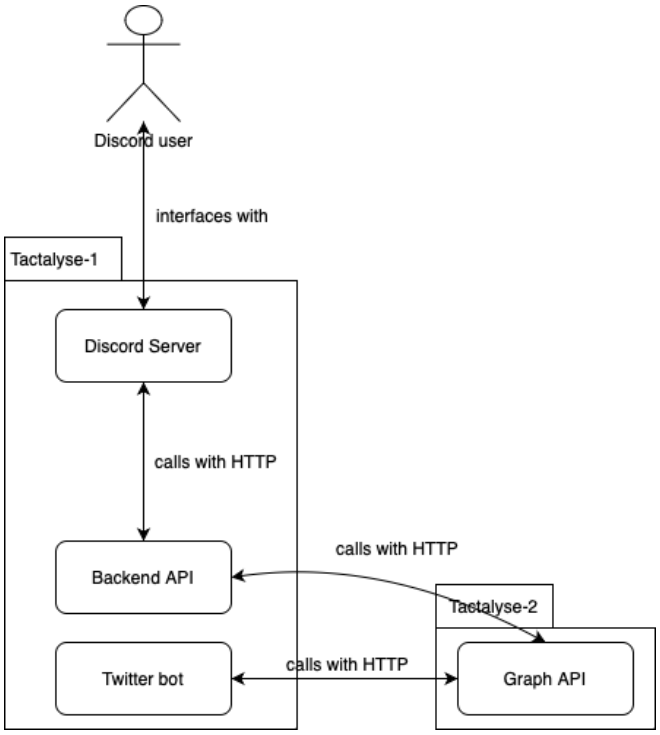


Figure 4: Diagram of the bot system architecture

## 4.4 Module View

This section gives an overview of the module design of our project.
To adhere to the MVC pattern, we have decoupled the API endpoint in the controller from the core functionality of the application. This is visible through the separation of the `controller` module, which only contains the endpoint, and `service` classes that make calls to functions from other modules.
The model is represented through three modules: `data`, `graph_generator` and, for the PDF generator, `pdf_generator`. `data` contains functionality related to extracting and processing data from Excel files. `graph_generator` contains functionality related to converting the processed data into relevant graphs. `pdf_generator` contains functionality related to generating a PDF report from input. `graph_generator` will need data in a dataframe structure in order to generate the graphs. This data will be provided by `controller` after passing the received excel files through `data` function calls. In the case of the PDF generator service, `pdf_generator` will need graphs in an image format in order to generate the reports. These will be provided by the `controller` module after generating them through `graph_generator` function calls. As such, `data`, `pdf_generator` and `graph_generator` are fully decoupled in both APIs. `controller` is the only module with dependencies on each of the other modules. The frontend/bots query the APIs, and gain access to their functionality that way. The frontend acts as the View component, and the bots contain both additional Controller, and View components.

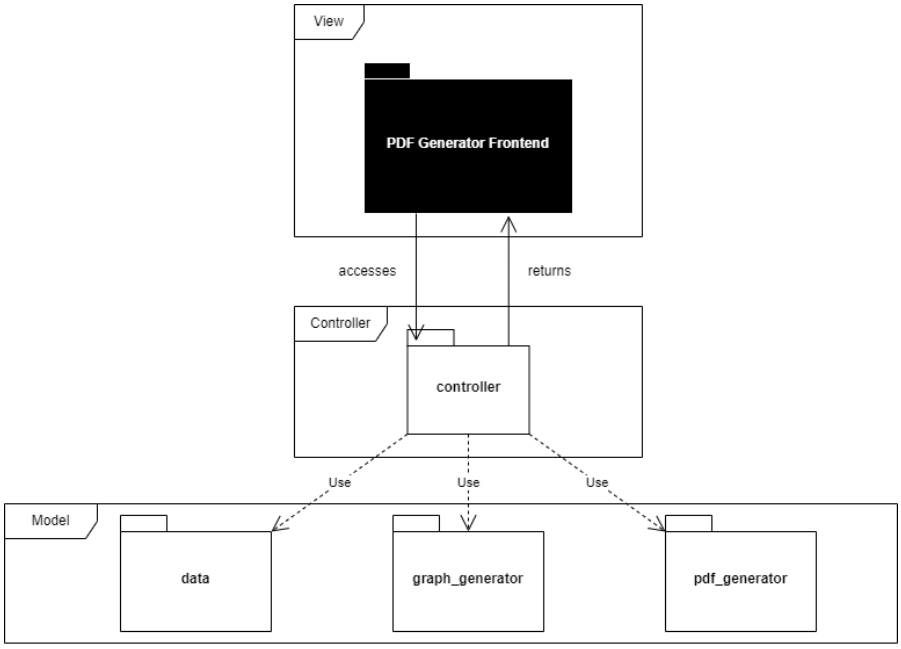



Figure 5: Module diagram showing module dependencies for the PDF generator API

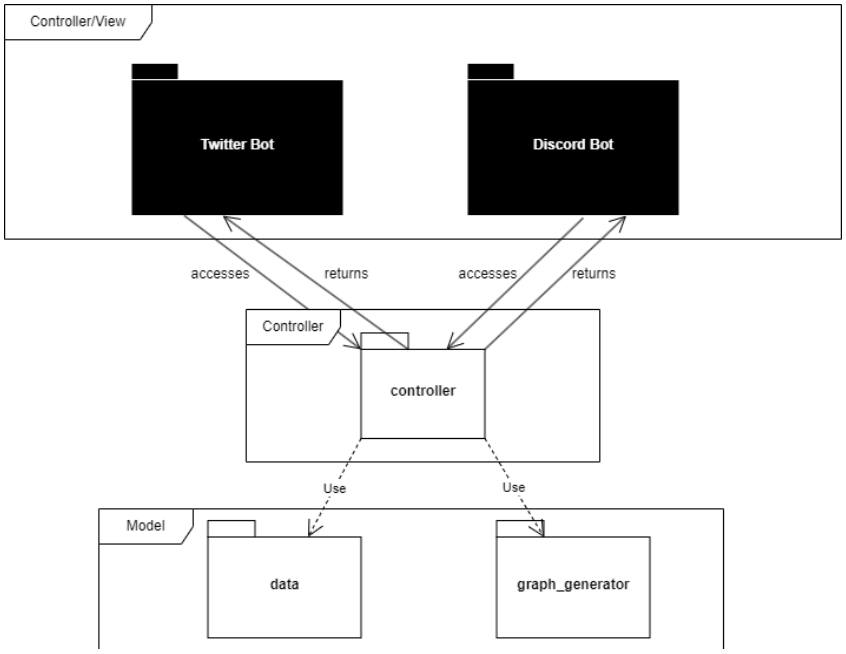



Figure 6: Module diagram showing module dependencies for the graph generator API

## 4.5 Execution View

In this section, we describe the sequential process of our project. The following diagrams show what each use case sequence in our programs look like, with every function call included. Aside from the diagram for the pdf generator, all diagrams were generated using the AppMap extension for IntelliJ to save time. The diagrams are also available for download in the appendix, as they are quite large and hard to read in this report's format.

- Use Case 1: Generate Football Data Report

This use case concerns the generation of a football data report, using the /pdf POST endpoint, as specified in section 4.2.

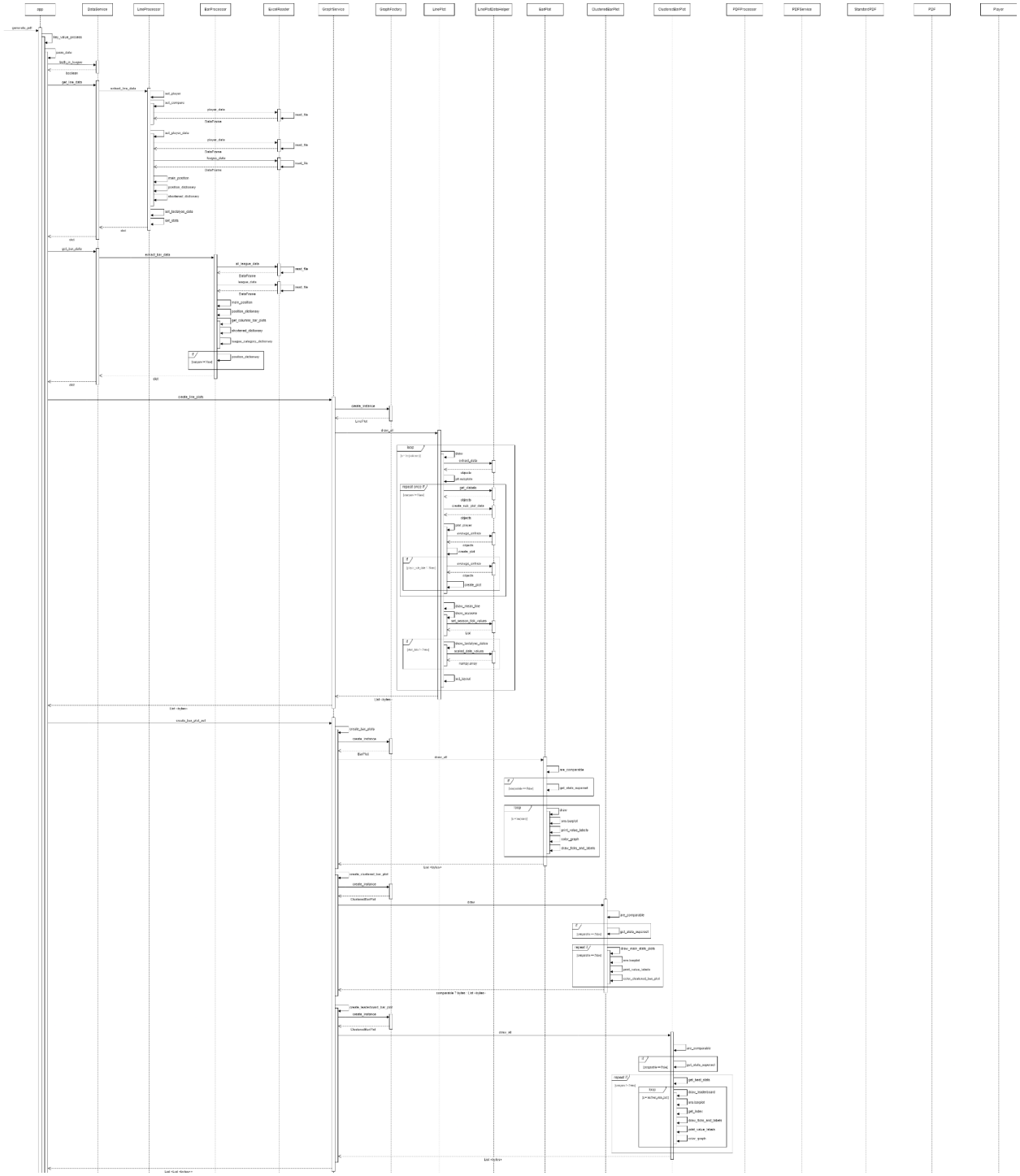


Figure 7: First half of the sequence diagram for generating a single player football player report with the PDF generator API. This diagram includes data processing and graph generation.

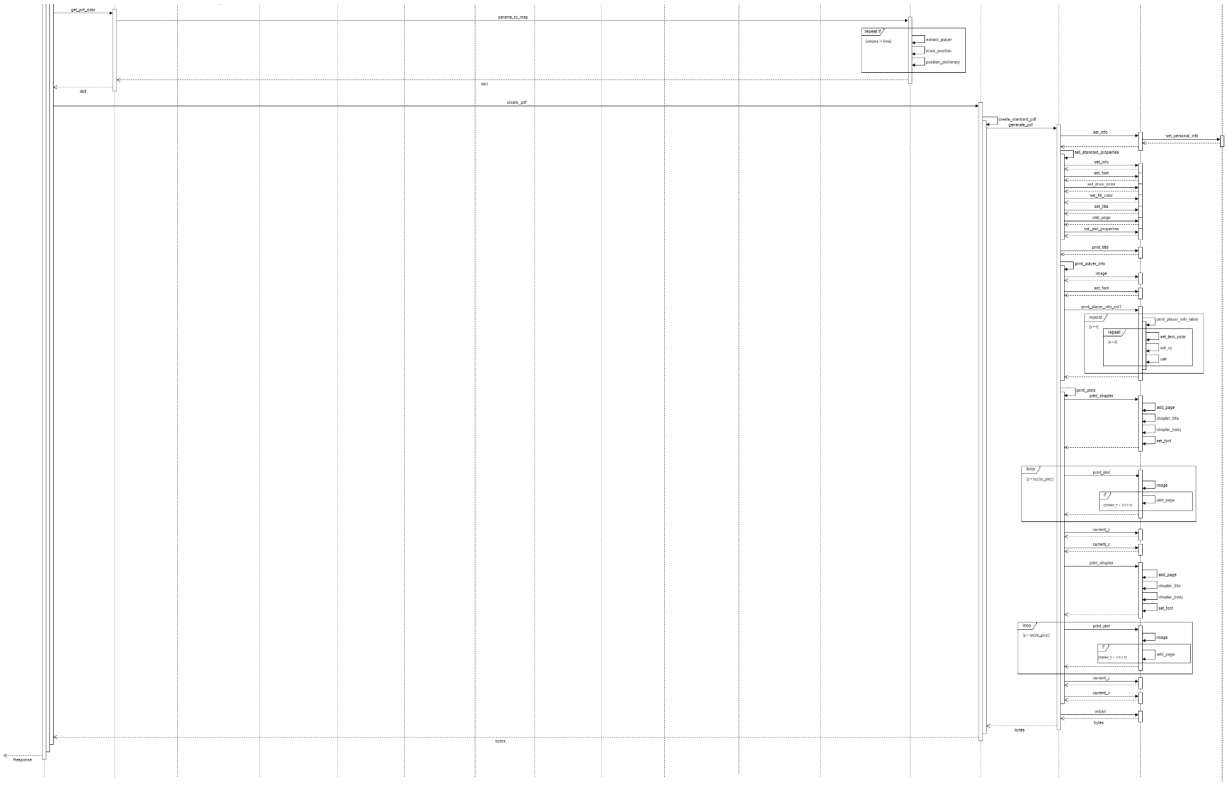


Figure 8: Second half of the sequence diagram for generating a single player football player report with the PDF generator API. This diagram includes PDF generation.

- Use Case 2: Generate Line Graph

This use case concerns the generation of a line graph out of player match Excel data files, using the /graph/line POST endpoint, as specified in section 4.2.

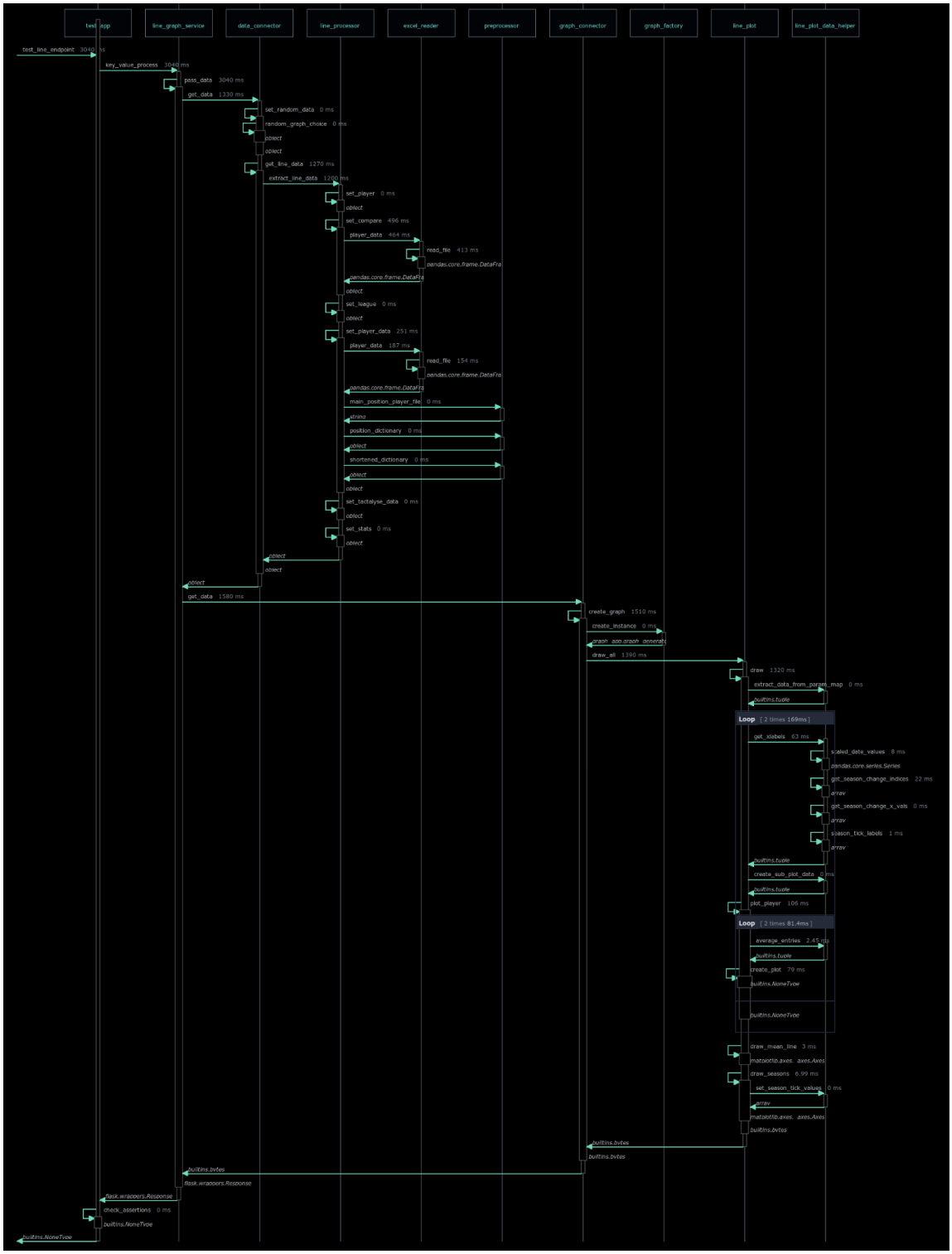



Figure 9: Sequence diagram for generating a line graph with the graph generator API

- Use Case 3: Generate Radar Graph

This use case concerns the generation of a radar graph out of football league Excel data files, using the /graph/radar POST endpoint, as specified in section 4.2.

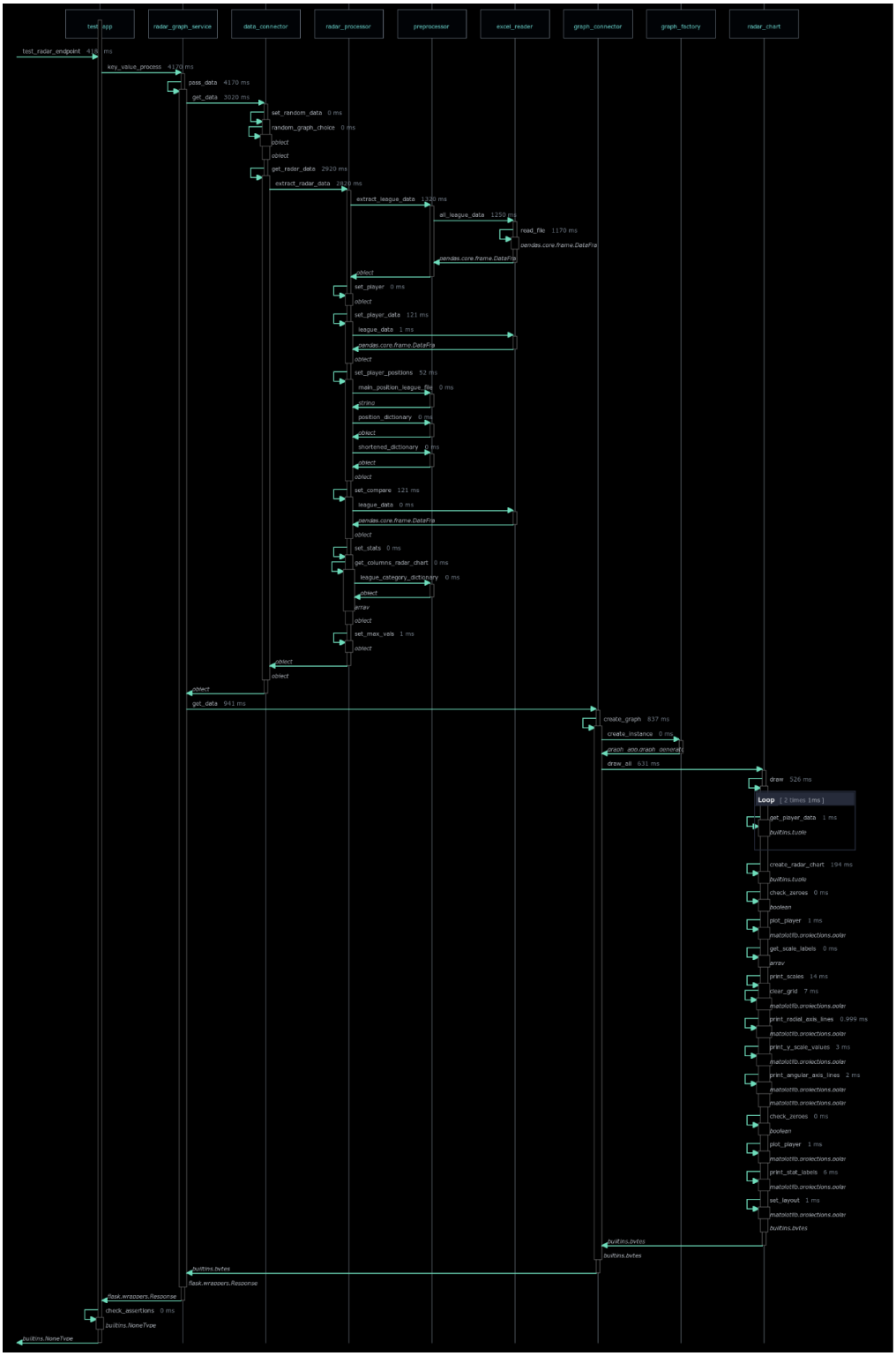



Figure 10: Sequence diagram for generating a radar graph with the graph generator API

- Use Case 4: Generate Random Graph

This use case concerns the generation of a random graph out of football data Excel files, using the /graph POST endpoint, as specified in section 4.2. This particular flow generated a randomized line graph.

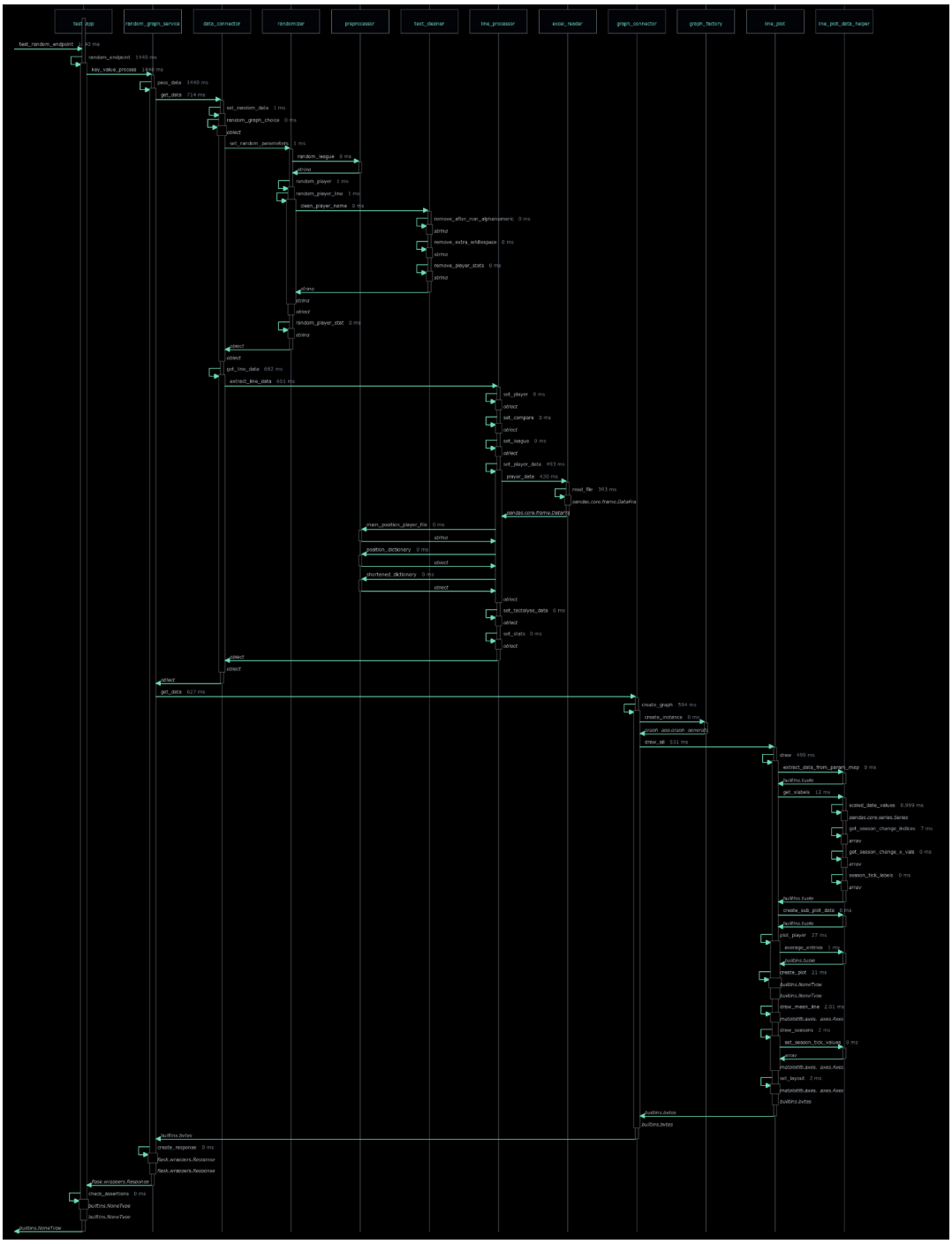


Figure 11: Sequence diagram for generating a random graph with the graph generator API

# 4.6 Code Allocation View

In this section, we provide an overview of the code base in class diagrams.
`__temp` indicates a private variable, `_temp` a protected variable. Field variables are defined as *`type`* `(__)name`. Abstract classes and functions are indicated by the name being italicized. For classes (indicated with capital letters), it may be assumed that all private field variables have getters. Different colors indicate different modules or modules within other modules. Dashed lines with arrows indicate dependencies, while solid lines with arrows indicate generalization/specialization, according to the UML syntax.
For better readability, we have sectioned the same diagram without dependency arrows, and placed it in Appendix C.

**PDF Generator**

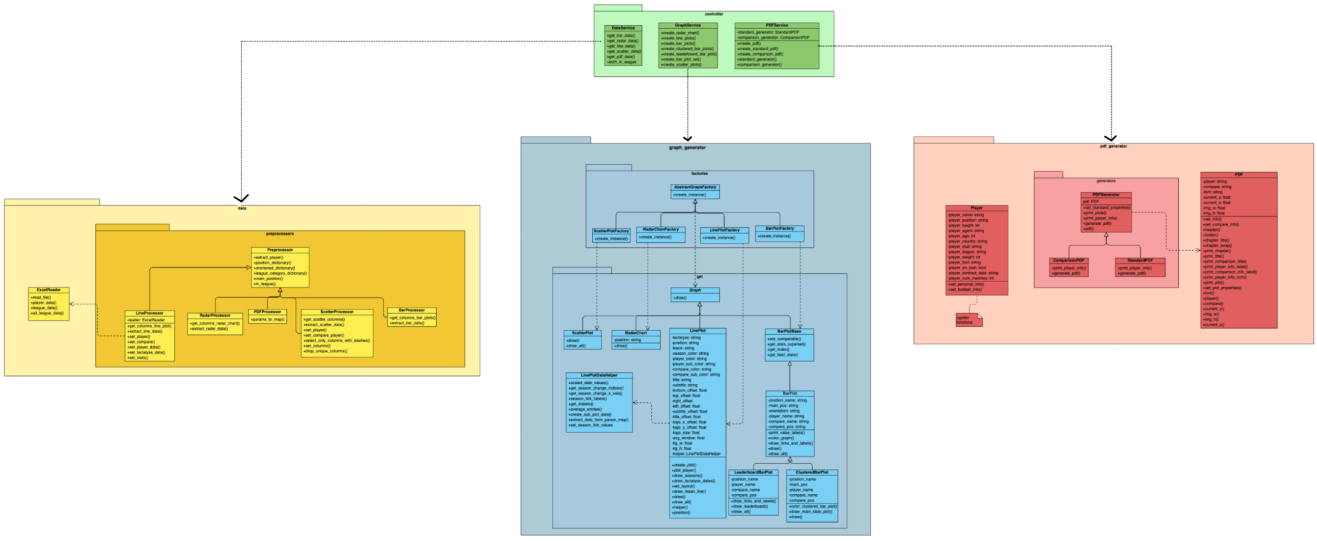


Figure 12: Class diagram showcasing the PDF generator code allocation

**Graph Generator**

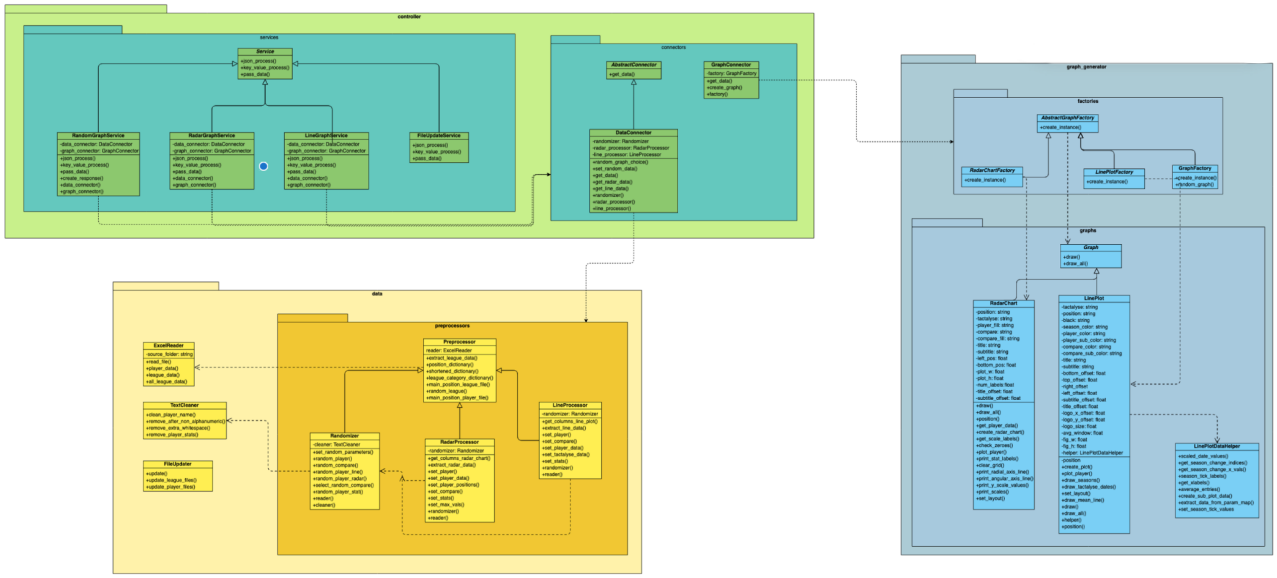


Figure 12: Class diagram showcasing the graph generator code allocation

# Appendix

## A. Change Log

- 17.March.2023 **Sangrok**, **Mikko**, **Bianca**: Add sketch of the architecture.
- 20.March.2023:
    - **Sangrok**: Add sequence diagram for Execution View
    - **Mikko, Bianca**: Add UML diagram for Module View
- 21.March.2023 **Sangrok**: Add Introduction section of the document.
- 23.March.2023:
    - **Mikko**: Stakeholders (pasted), significant requirements, technology stack (language, framework)
    - **Sangrok**: Add Libraries section(Pandas, pyFPDF)
- 24.March.2023:
    - **Sangrok**: Add contents in pyFPDF, Add Seaborn section, Execution View.
    - **Mikko**: Data, Design Patterns, API Specification, text for Conceptual View and Module View.
- 28.March.2023: **Sangrok**: Modify and add some contents in Libraries and Execution View.
- 31.March.2023: **Sangrok**: Add table of contents and Glossary, Fix Introduction, Technology Requirements, Technology Stack based on the feedback.
- 01.April.2023:
    - **Mikko**: Modify conceptual and module diagrams, Code Allocation View text.
    - **Sangrok**: Add high-level diagram of our project, Fix Execution View by breaking down the original diagram.
- 02.April.2023: **Sangrok**: Fixed typos in the diagram(Backend ↔ PDF generator).
- 02.April.2023: **Mikko**: Went over Sangrok's changes.
- 03.April.2023: **Mikko**: Added class diagram, final changes to document.
- 05.May.2023:**Sangrok**: Modify introduction section.
- 06.May.2023:**Sangrok**: Delete Stakeholders section, add the importance note at the excel data part in Architectural Requirements section, modify Technology Stack section(Programming language, Framework).
- 07.May.2023:**Sangrok**: Add data diagram, Modify Libraries section by giving alternatives and chosen reasons, add new conceptual diagram in the Architectural Overview section.
- 05.Jun.2023: **Sangrok**: Went over overall design doc's texts, modify text in API Specification
- 06.Jun.2023:
    - **Mikko**: Added API specification for graph API
    - **Sangrok**: Added fixed conceptual diagram and fixed texts, added Bot conceptual diagram and texts.

- 07.Jun.2023: **Sangrok**: Went over Technology Stack section and added missing texts, added more information in the conceptual diagram in the beginning of Architectural Overview section.
- 08.Jun.2023:
    - **Sangrok**: Add numbering in the sections, Modify overall design doc texts based on the TA's feedback.
    - **Mikko**: Added sequence diagrams
    - **Matteo**: Added class diagrams
- 9 June 2023: **Bianca, Sangrok, Mikko, Matteo**: Finished the document

## B. Sequence Diagrams

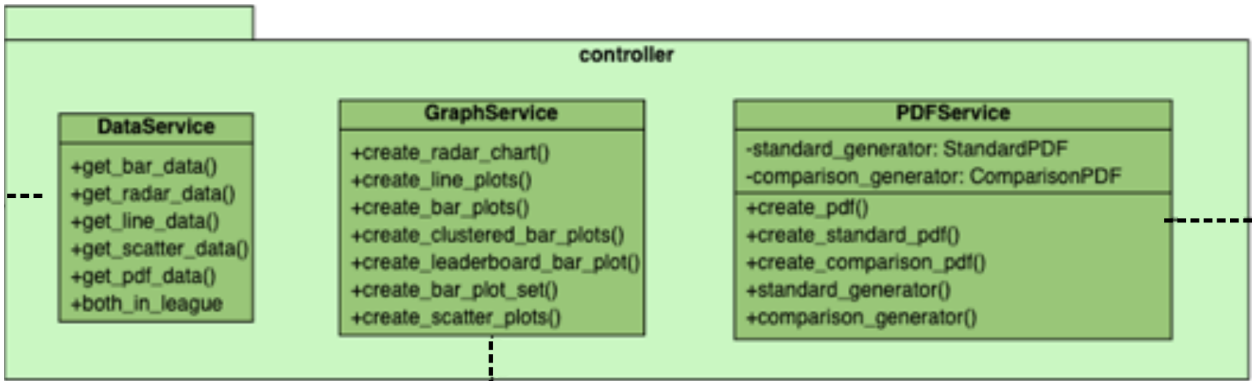
Due to the size of the diagrams, it may be hard to read them in this document. However, due to the page limit, we could not split them up and make them bigger. If better readability is desired, here are download links for the images:

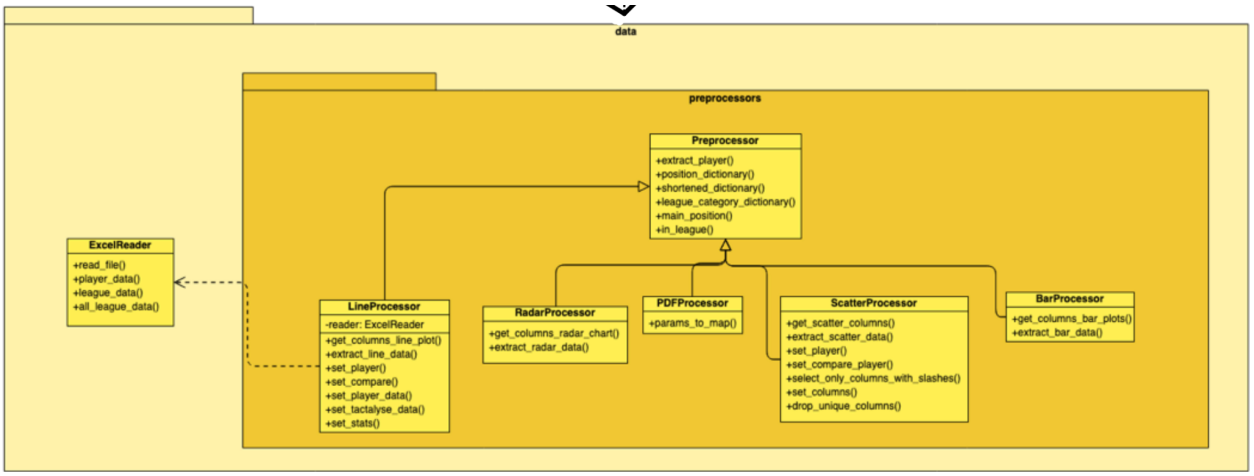
- line_sequence.png
- PDF Sequence.png
- radar_sequence.png
- random_line_sequence.png

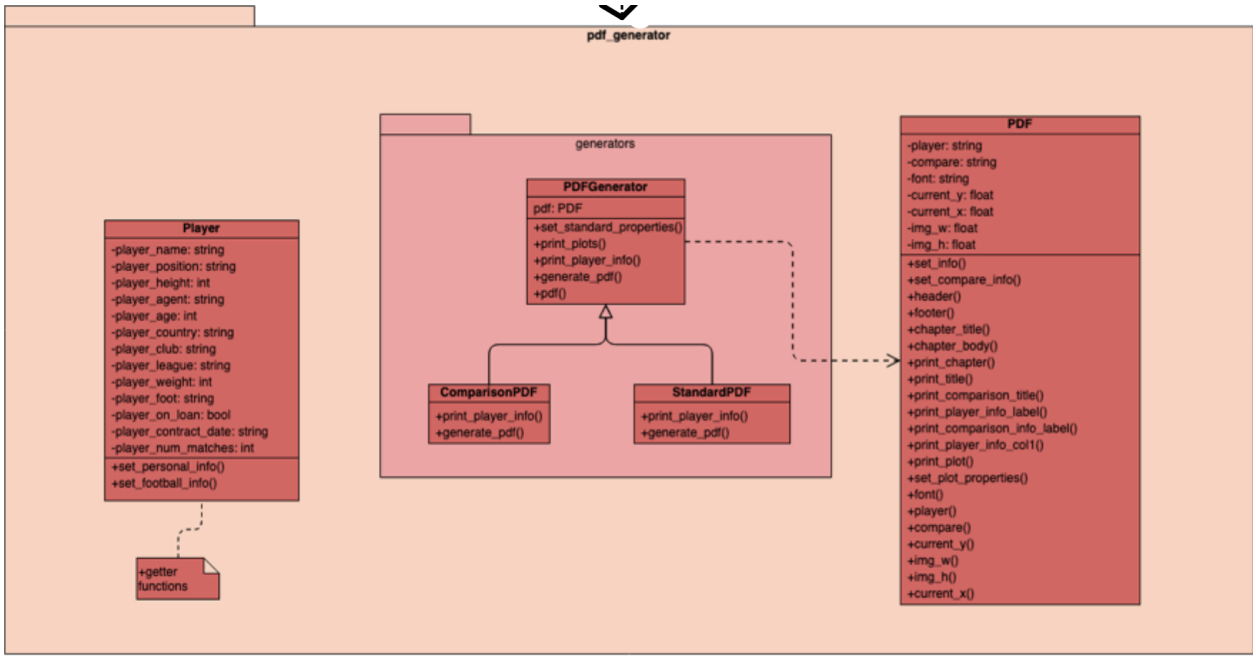
## C. Class Diagram

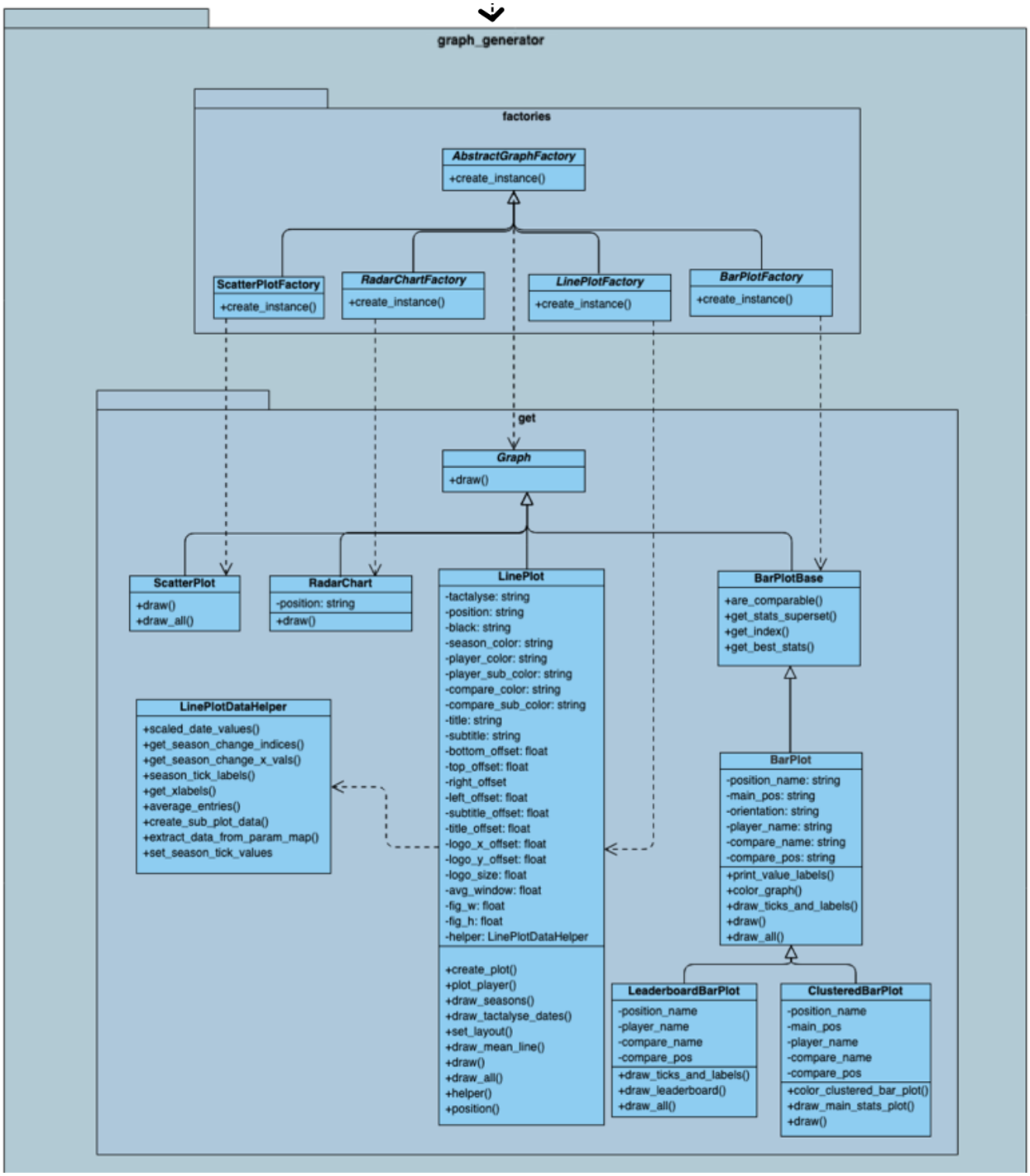
The explanation for the diagrams is the same as detailed in section 4.6: Code Allocation View.

**PDF Generator**

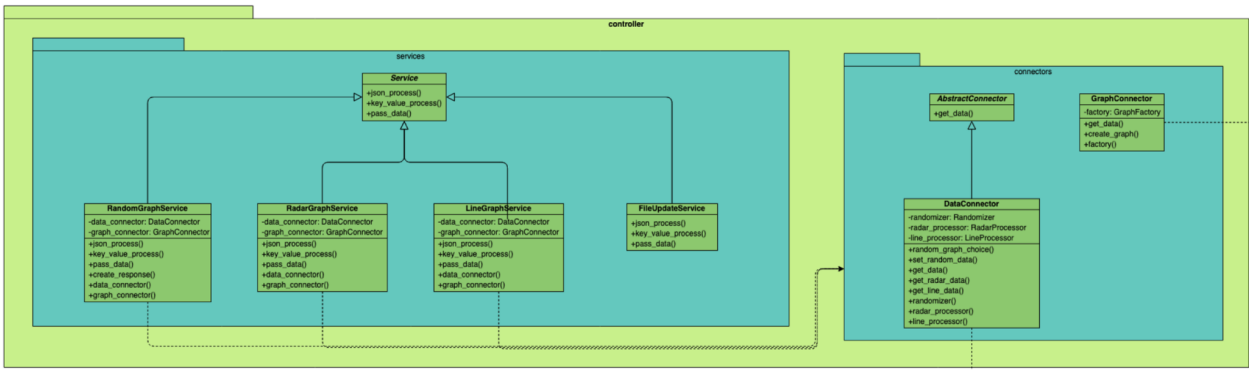


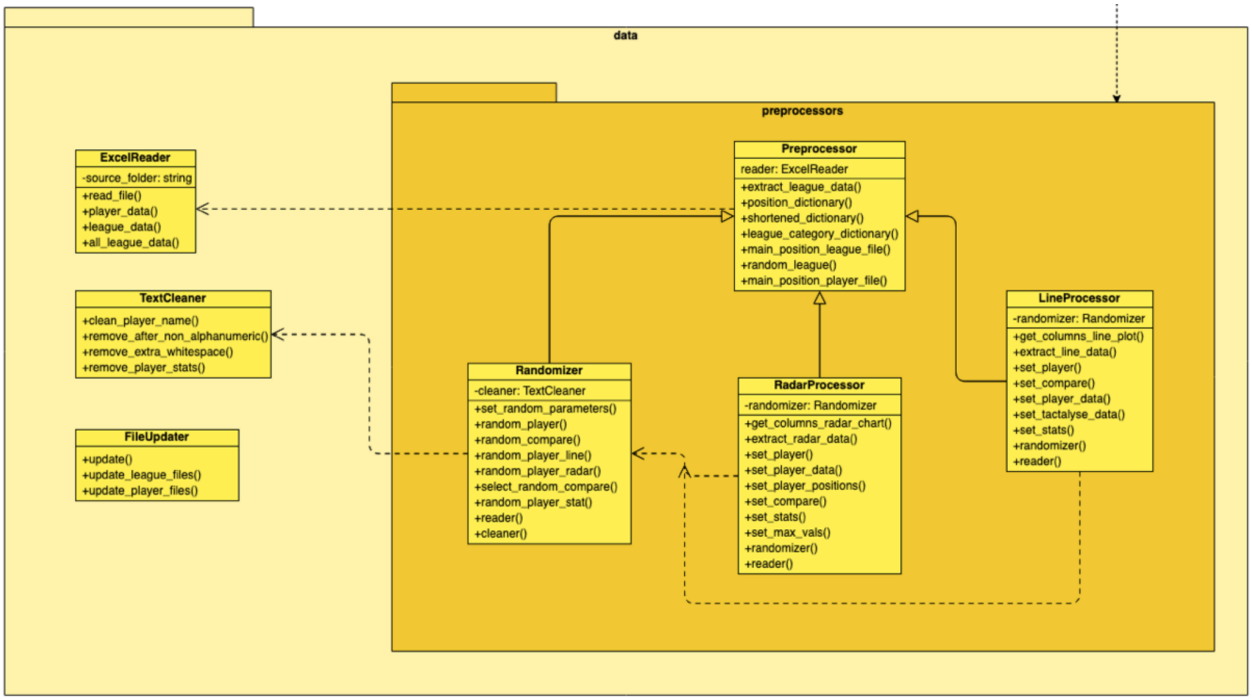

data
preprocessors
Preprocessor
+extract_player()
+position_dictionary()
+shortened_dictionary()
+league_category_dictionary()
+main_position()
+in_league()
ExcelReader
+read_file()
+player_data()
+league_data()
+all_league_data()
LineProcessor
-reader: ExcelReader
+get_columns_line_plot()
+extract_line_data()
+set_player()
+set_compare()
+set_player_data()
+set_tactalyse_data()
+set_stats()
RadarProcessor
+get_columns_radar_chart()
+extract_radar_data()
PDFProcessor
+params_to_map()
ScatterProcessor
+get_scatter_columns()
+extract_scatter_data()
+set_player()
+set_compare_player()
+select_only_columns_with_slashes()
+set_columns()
+drop_unique_columns()
BarProcessor
+get_columns_bar_plots()
+extract_bar_data()

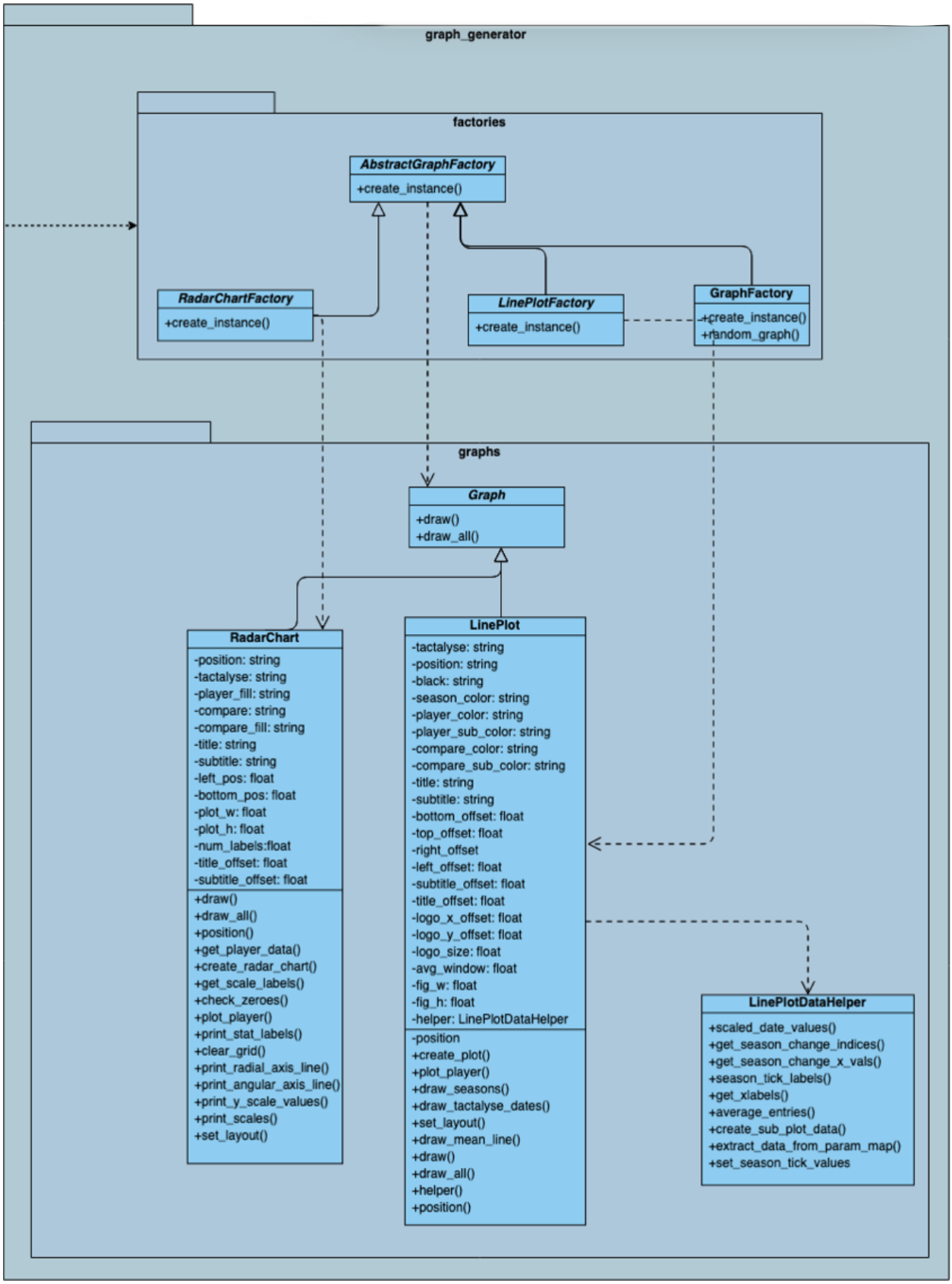

pdf_generator
generators
PDFGenerator
pdf: PDF
+set_standard_properties()
+print_plots()
+print_player_info()
+generate_pdf()
+pdf()
ComparisonPDF
+print_player_info()
+generate_pdf()
StandardPDF
+print_player_info()
+generate_pdf()
Player
-player_name: string
-player_position: string
-player_height: int
-player_agent: string
-player_age: int
-player_country: string
-player_club: string
-player_league: string
-player_weight: int
-player_foot: string
-player_on_loan: bool
-player_contract_date: string
-player_num_matches: int
+set_personal_info()
+set_football_info()
+getter functions
PDF
-player: string
-compare: string
-font: string
-current_y: float
-current_x: float
-img_w: float
-img_h: float
+set_info()
+set_compare_info()
+header()
+footer()
+chapter_title()
+chapter_body()
+print_chapter()
+print_title()
+print_comparison_title()
+print_player_info_label()
+print_comparison_info_label()
+print_player_info_col1()
+print_plot()
+set_plot_properties()
+font()
+player()
+compare()
+current_y()
+img_w()
+img_h()
+current_x()


graph_generator
factories
AbstractGraphFactory
+create_instance()
ScatterPlotFactory
+create_instance()
RadarChartFactory
+create_instance()
LinePlotFactory
+create_instance()
BarPlotFactory
+create_instance()
get
Graph
+draw()
ScatterPlot
+draw()
+draw_all()
RadarChart
-position: string
+draw()
LinePlot
-tactalyse: string
-position: string
-black: string
-season_color: string
-player_color: string
-player_sub_color: string
-compare_color: string
-compare_sub_color: string
-title: string
-subtitle: string
-bottom_offset: float
-top_offset: float
-right_offset
-left_offset: float
-subtitle_offset: float
-title_offset: float
-logo_x_offset: float
-logo_y_offset: float
-logo_size: float
-avg_window: float
-fig_w: float
-fig_h: float
-helper: LinePlotDataHelper
+create_plot()
+plot_player()
+draw_seasons()
+draw_tactalyse_dates()
+set_layout()
+draw_mean_line()
+draw()
+draw_all()
+helper()
+position()
BarPlotBase
+are_comparable()
+get_stats_superset()
+get_index()
+get_best_stats()
LinePlotDataHelper
+scaled_date_values()
+get_season_change_indices()
+get_season_change_x_vals()
+season_tick_labels()
+get_xlabels()
+average_entries()
+create_sub_plot_data()
+extract_data_from_param_map()
+set_season_tick_values
BarPlot
-position_name: string
-main_pos: string
-orientation: string
-player_name: string
-compare_name: string
-compare_pos: string
+print_value_labels()
+color_graph()
+draw_ticks_and_labels()
+draw()
+draw_all()
LeaderboardBarPlot
-position_name
-player_name
-compare_name
-compare_pos
+draw_ticks_and_labels()
+draw_leaderboard()
+draw_all()
ClusteredBarPlot
-position_name
-main_pos
-player_name
-compare_name
-compare_pos
+color_clustered_bar_plot()
+draw_main_stats_plot()
+draw()

## Graph Generator




graph_generator
factories
AbstractGraphFactory
+create_instance()
RadarChartFactory
+create_instance()
LinePlotFactory
+create_instance()
GraphFactory
+create_instance()
+random_graph()
graphs
Graph
+draw()
+draw_all()
RadarChart
-position: string
-tactalyse: string
-player_fill: string
-compare: string
-compare_fill: string
-title: string
-subtitle: string
-left_pos: float
-bottom_pos: float
-plot_w: float
-plot_h: float
-num_labels:float
-title_offset: float
-subtitle_offset: float
+draw()
+draw_all()
+position()
+get_player_data()
+create_radar_chart()
+get_scale_labels()
+check_zeroes()
+plot_player()
+print_stat_labels()
+clear_grid()
+print_radial_axis_line()
+print_angular_axis_line()
+print_y_scale_values()
+print_scales()
+set_layout()
LinePlot
-tactalyse: string
-position: string
-black: string
-season_color: string
-player_color: string
-player_sub_color: string
-compare_color: string
-compare_sub_color: string
-title: string
-subtitle: string
-bottom_offset: float
-top_offset: float
-right_offset
-left_offset: float
-subtitle_offset: float
-title_offset: float
-logo_x_offset: float
-logo_y_offset: float
-logo_size: float
-avg_window: float
-fig_w: float
-fig_h: float
-helper: LinePlotDataHelper
-position
+create_plot()
+plot_player()
+draw_seasons()
+draw_tactalyse_dates()
+set_layout()
+draw_mean_line()
+draw()
+draw_all()
+helper()
+position()
LinePlotDataHelper
+scaled_date_values()
+get_season_change_indices()
+get_season_change_x_vals()
+season_tick_labels()
+get_xlabels()
+average_entries()
+create_sub_plot_data()
+extract_data_from_param_map()
+set_season_tick_values