Institute of Mathematics and Informatics

Web Scraping and Monte Carlo Simulations for Analytical Forecasting

Author
Lorand Heidrich
Computer Science BSc

Supervisor
Adam Kovacs
Teaching Assistant

Eger, 2024

Acknowledgments

I would like to express my gratitude to Dr. Eric Grono and Mr. Garrett Weinzierl for their mentorship and support throughout my journey into the domain of Monte Carlo simulation and analytical forecasting. Their expertise, patience, and guidance have not only paved the way for new avenues of exploration but have also instilled within me an appreciation for the intersection of computer science and predictive modeling.

Their influence has played an instrumental role in shaping my academic and professional trajectory. I am indebted to them for their impact on my endeavors.

Thank you both for your generosity in sharing your time and knowledge, and your patience in addressing my myriad of questions throughout my studies.

Contents

Chapter 1

Introduction

In today's world, data has become of paramount importance, profoundly influencing our lives and shaping decision making processes. The acquisition, processing, and interpretation of data is fundamental across multiple domains. [1] Recognized as the cornerstone of contemporary insights, data serves as the basis of deriving valuable insights, and making informed projections, thereby guiding strategic planning and allowing for suitable preparation in the face of uncertainty. However, utilizing the full potential of acquired information effectively in a complex, multi-variable dynamic environment can be a challenging task [2].

This thesis approaches data collection and forecasting from a sports analytical perspective, aiming to derive statistical insights and formulate projections regarding future performance. It endeavors to utilize a combination of web scarping techniques [3] and Monte Carlo simulation [4] for analytical forecasting. Through the integration of these techniques, this research aims to explore a comprehensive methodology for data acquisition and predictive modeling.

1.1 Contextual Background

The National Basketball Association (NBA) [5] is well known for its worldwide prominence and dedicated fan base. Its enduring popularity has resulted in a multitude of analytical data relating to historic games. This abundance of statistical data, along with a widespread general awareness of the sport and my personal enthusiasm for it, positions historic NBA games an ideal domain for exploring predictive modeling based on data obtained through web scraping.

1.2 Motivation

The incentive for this research is derived from a keen interest in the technical intricacies of web scraping and probabilistic elegance of Monte Carlo simulations. The application of these techniques transcends the domain of sports analytics, with uses in finance [6], physics [4], and beyond [7].

1.3 Objectives

The primary objective of this thesis is two-fold. Initially, to employ web scarping techniques to gather comprehensive historical NBA game data from the early 1990s. Subsequently, to utilize said data to simulate a general probabilistic outcome for selected historic NBA games.

Specifically, the research aims to:

- Develop a multi-approach web scraping pipeline to gather comprehensive historical data for a given NBA season and team.
- Manage and store the acquired data.
- Implement a multi-epoch Monte Carlo simulation to model potential game outcomes based on the attained data through modeling offensive possessions.
- Evaluate the predictive accuracy and reliability of the proposed methodology through empirical testing and validation against actual historic game results.

Through these objectives, this thesis undertakes to promote a deeper understanding of web scraping and predictive modeling within sports analytical forecasting.

Chapter 2

Methodology

2.1 Web Scraping Techniques

2.2 Monte Carlo Simulation

Chapter 3

Requirements

3.1 Requirements List

The system shall be constructed to uniquely fulfill both the requirements of a computer science thesis, and the domain of data acquisition and simulation based projections. It should therefore result in an intuitive end-user experience, leveraging the web scraping and Monte Carlo simulation methodologies explored in this thesis.

The client interface should allow users to interact with the business logic[1], thereby accessing the database through built-in functions. It should also allow for the utilization of web scraping and Monte Carlo methodologies. The Use Case diagram depicted below outlines the basic functionality described by the Functional - (see table 3.1), Non-Functional - (see table 3.2), and Platform Requirements (see table 3.3) outlined in this chapter.

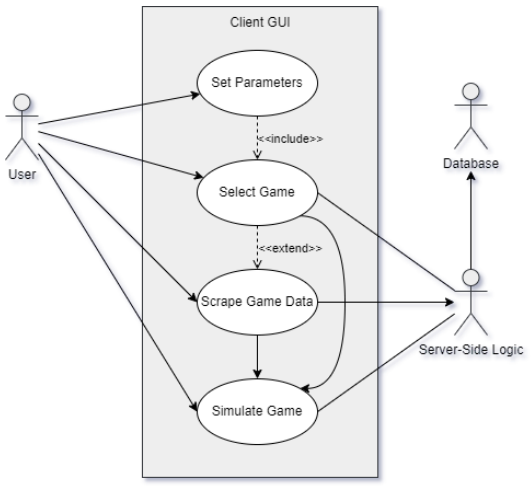


Figure 3.1: Use Case Diagram

[1]The term refers to the collection of algorithms responsible for allocating and processing data through communication with the database in order to serve the user interface, while maintaining its independence from both. For further information, please see [8].

3.1.1 Functional Requirements

ID	Name	Description
R1	Database	The system must allow users to select either the default database or utilize their own, based on a URI connection string.
R2	Game Parameters	It should provide users with a method to set the season, home- and away team.
R3	Epochs	The system must enable users to specify the number of epochs for the Monte Carlo simulation.
R4	Game Data	Historic game data should be displayed based on these settings for user review.
R5	Select Game	Users must be able to select an exact game to simulate from the displayed list of historic games.
R6	Missing Game	The system must recognize if the selected historic game is not in the database.
R7	Scrape Method	It should provide users with options for scraping the missing data trough different web scraping methods.
R8	Proxies	Scraping options should include the ability to use proxies.
R9	Proxy List	Users should have the ability to utilize their own proxy lists.
R10	Forced Scrape	The system must allow users the option to scrape game data even when it is deemed unnecessary by the algorithm.
R11	Validation	The system must ensure that data is not duplicated in the database.
R12	Simulation	The system must execute Monte Carlo simulations based on the selected game parameters.
R13	Graphs	It should visualize simulation results with graphs, including a probability density graph and a violin graph.
R14	Metrics	The system must return basic metrics such as the number of wins for each team and the mode of scores.
R15	Comparison	Users should be able to compare simulation results with original game data.

Table 3.1: List of Functional Requirements

3.1.2 Non-Functional Requirements

ID	Name	Description
NR1	Anonymity	The system must take steps to attempt anonymity throughout the web scraping process.
NR2	Validation	It should validate user input parameters, throwing errors when incorrectly set.
NR3	Errors	Users should be notified of errors during the application's operation.
NR4	Logging	It must utilize a logging system to allow for easier debugging.
NR5	Intuitive	The system must have an easy-to-use and intuitive interface.
NR6	Requests	The client side of the system must communicate with the server-side logic using HTTP to attain services as a responses.
NR7	SQL	The server-side logic should interact with the database using SQL queries.
NR8	Database	The system must be able to utilize separate MySQL database servers.
NR9	Testing	It should undergo thorough testing and validation to ensure accuracy, reliability, and robustness.
NR10	Regulation	The system must comply with relevant legal and regulatory requirements.

Table 3.2: List of Non-Functional Requirements

3.1.3 Platform Requirements

ID	Component	Requirement
PR1	Client	The application should be compatible with Windows 10 (or later) operating systems.
PR2	Client	The operating system is required to have .Net Framework 4.0.3 (or later).
PR3	Host	Server environment must be capable of running a Python application with a Flask framework.
PR4	Requirements	Back-end application requirements are available at: `https://github.com/lesheidrich/WebScraping_and_MCSim/blob/master/requirements.txt`.
PR5	Database	The database server must be compatible with either XAMPP or MySQL.

Table 3.3: List of Platform Requirements

3.1.4 Use Cases

- **Game Selection**: The user selects parameters such as the desired season, home- and away team, to initialize game selection, then chooses the desired match from the returned table.
- **Validation**: After accidentally setting a team to play against themselves, the user receives an error message alerting them of the mistake.
- **Scraping**: Following the game selection process, the system determines the game data is not in the database, then proceeds to utilize web scraping techniques to gather the data from online sources.
- **Simulation**: A user parameterizes the number of epochs for the Monte Carlo simulation and initializes game selection. The system utilizes the acquired historical data to run simulations, generating probabilistic outcomes for the historic NBA game.
- **Comparison**: Users compare the results of the Monte Carlo simulation with the original game data, assessing the accuracy of the model. The system further provides probability density- and violin graphs to further facilitate result analysis.
- **Error Handling**: When the user tries to initialize game selection, the database is down. The host service returns an error, notifying the user of the access issue. The user escalates the error, and upon its resolution normal system operations resume.

Chapter 4

Architecture

4.1 Design Concepts

At its core, the application relies heavily on a three principal layer [9, p. 19] concept, commonly found in systems utilizing a database and presentation layer. Fowler refers to these as presentation logic, domain logic, and data source. The presentation logic facilitates user interaction with the system, the data source handles data transactions and houses application information, while the domain logic's algorithms are responsible for data modification and layer interaction.

This is in line with the Gang-of-Four's [1] Model-View-Controller (MVC) [10, p. 529] design pattern. The View receives user-initiated interactions along with their parameters, and presents the application's data. It is capable of connecting directly to the model, while operating in conjunction with the Controller. The Controller interacts with both components as it processes their data and coordinates operations. The Model houses and manages the application data.

As discussed by Ahlan, A. R., Ahrnud, M. B., and Arshad, Y. [12], there have been several uses and variations of thin client applications since the 1970s. As a generalization, the *view* in its capacity as the client receives application data and logic based services from a host system. This application adheres to the this concept quite strictly, with the client acting as an intermediary between the user and the host, taking use parameters and displaying host response results. The host service, operating as the back-end, encompasses the previously discussed Model, Controller and all other components of the application.

[1] The Gang of Four [11] (GOF) are a group of four writers, all computer science professionals and entrepreneurs. Their literature and courses focus on professional development in the domain of computer science.

4.2 Components

Following the MVC design pattern's component structure, the application's presentation logic is allocated to the *view* package. The database and simple business logic allowing for record management is stored in the *model*. Acting as an intermediary between the two, the *controller* package orchestrates the flow of information along with its processing. The *webscraper* and *simulator* packages also tie into the *controller*, offering web scraping, and Monte Carlo simulation logic respectively, while further decoupling the application's components and allowing for better organization and maintainability through separation of concerns [2].

The application is organized in a manner, that all components embody system packages allowing improved readability and usability. The following subsections discuss each package's purpose and functionality.

4.2.1 *controller*

Purpose: The package is responsible for processing component interaction, and serves as the core logic of the system.

Functionality: Serving as the system's host, the *controller* is responsible for handling user prompts sent by the client in order to formulate an appropriate response. In order to fulfill this function it utilizes its connection to each component of the application, accessing their functionality as needed.

One of its responsibilities is accumulating data through the application of various web scraping methods, which are stored for future use. It's web scraper control functionality enables it to utilize the *webscraper* package to appropriate preset website data into memory, then reallocate it to the database using a combination of its own control processes along with the business logic of the *model* package. The web scraping sequence diagram illustrates the process (see 4.4).

When running a simulation for a parameterized game, the *controller* restructures relevant data for the selected historic games from the *model* into player, roster, and team objects usable by the *simulator*. The simulator returns the results to the host, which are forwarded back to the client. The Monte Carlo simulation sequence diagram shows further details (see 4.3).

Interaction: In its capacity as the main communications hub of the application, the *controller* interacts with every component in the system. The package's component

[2]Separation of concerns (SoC) is a software development design principle promoting segregation of source code elements by functionality in order to improve readability, organization and modification [13].

diagram provides a high level overview of basic component interaction (see4.1).

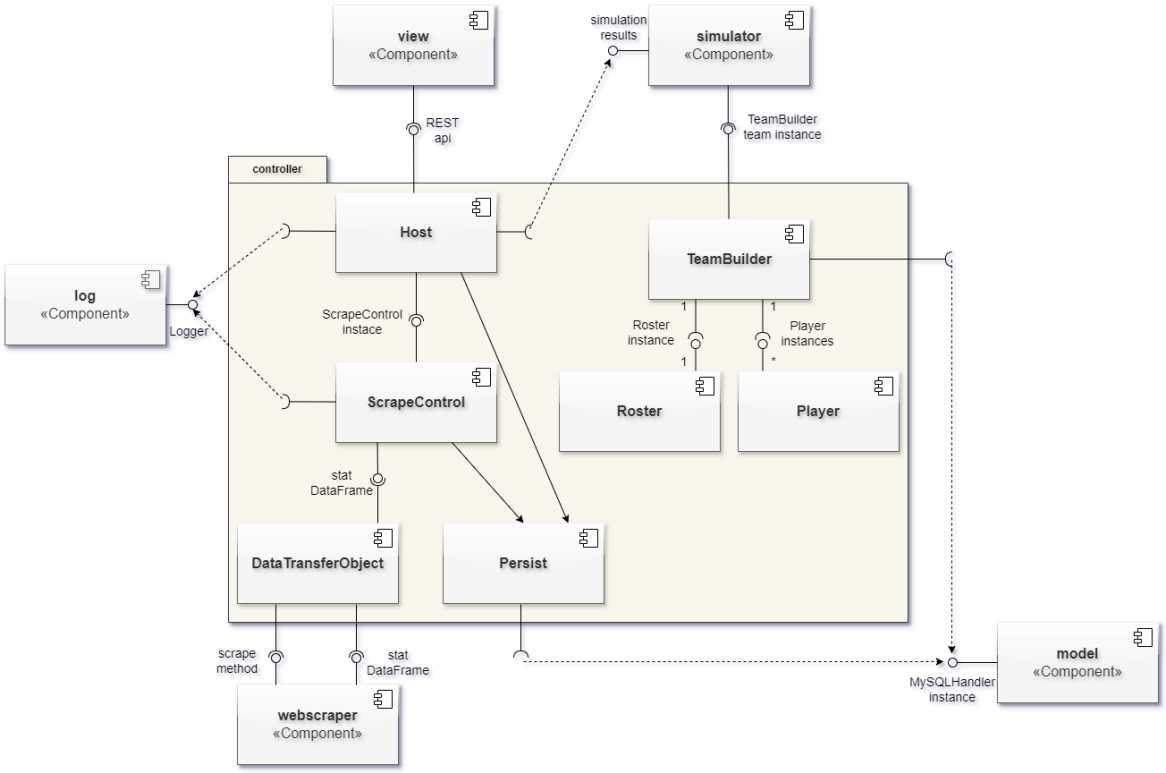


Figure 4.1: *controller* UML Component Diagram

4.2.2 *log*

Purpose: The component provides logging functionality for back-end operations.

Functionality: The *log* package contains logic allowing components to access system logs to document runtime errors and operations, useful for debugging potential issues. Log management functionality is also provided by the package, ensuring proper settings, size limitations and functionality.

Interaction: The *log* interacts with the *controller* and *webscraper* packages.

4.2.3 *model*

Purpose: The package ensures successful data management services within the application, through the storage and manipulation of data records and structures.

Functionality: The business logic enables system interaction with the database, encapsulating data access and administration services. Data security is enforced by minimizing vulnerabilities and prevents data corruption trough validation logic. The *model* services the system through the *controller* package's data manipulation and retrieval components.

The package also contains the NBA team enums [3] employed by the back-end logic. These ensure data validation and encapsulate all occurrences of team names in the system and its dataset, streamlining their application throughout processes.

Interaction: The primary business logic elements of the *model* communicate exclusively with the *controller*, enhancing system modularity (see 4.1). Due to their earlier described functionality, NBA team enums are also widely employed throughout the *simulation* package.

4.2.4 *simulation*

Purpose: The package's primary objective encompasses the repetitive simulation of a basketball game between selected teams at a specific point in time, with the end goal of returning the outcomes as graphical representations, comparable with the original historic game's outcome.

Functionality: The core functionality of the application revolves around implementing Monte Carlo simulations over a preset range of epochs to determine the probabilistic outcome of a historic NBA game.

The *simulator* package receives the relevant compiled data from the *controller*, initializing the creation of the simulation. Upon successful completion of the simulation, probability density and violin graphs are returned to the *controller* along with minimal game statistic like total win percentage and mode of scores reached per team. A detailed description of the process is illustrated in the Monte Carlo simulation's sequence diagram (see 4.3).

Interaction: The *simulator* relies on the *model*'s NBA team enum during operation, along with the *controller* for providing the necessary historic game data for simulation's successful operations. It further communicates with the *controller* in its capacity as the host. The *simulation* component diagram offers an overview of component interaction (see 4.2).

[3] In their capacity as a distinct object type, enumerations (enums) offer value binding across a group of encapsulated constants tied together through enumeration. Each value can act as a key identifier during instantiation, making enums a powerful structure for housing validated data [15].

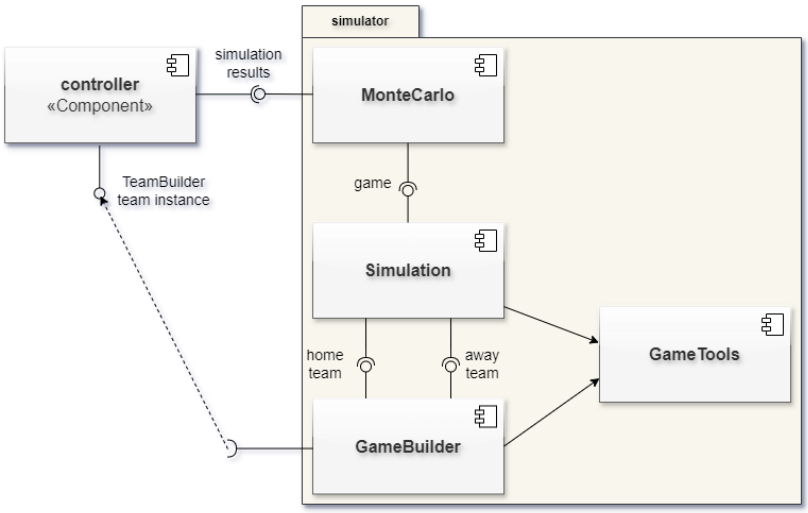


Figure 4.2: *simulation* UML Component Diagram

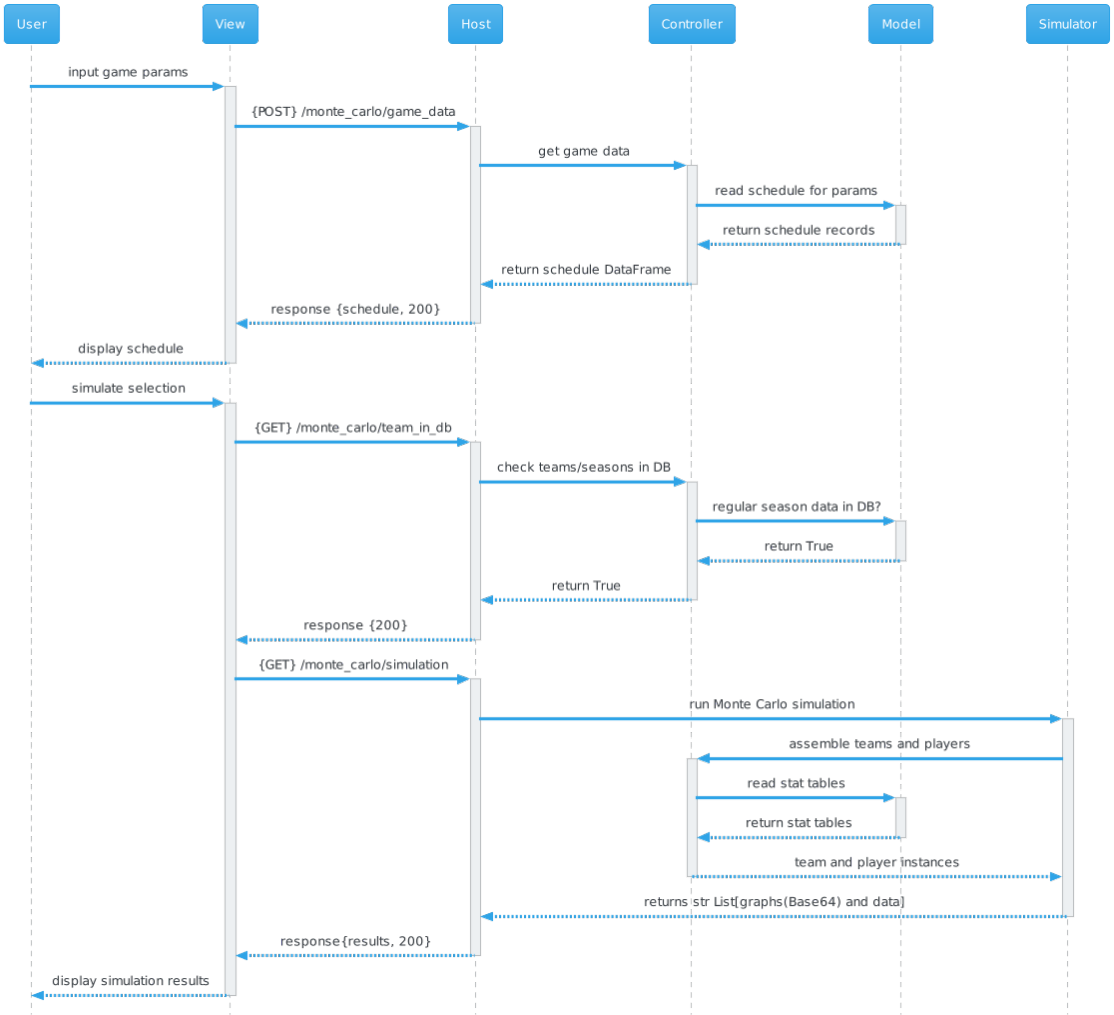


Figure 4.3: Monte Carlo Simulation UML Sequence Diagram

4.2.5 *test*

Purpose: The component is responsible for providing comprehensive back-end testing services.

Functionality: The test package constitutes a wide spectrum of tests focusing on the back-end of the system. It provides full-scale uni tests organized by module. Integration testing and linting is also included. The Testing and Validation chapter (see 6) contains a detailed breakdown of the package's functionality and implementation.

Interaction: Each unit test interacts with their respective components on the back-end. Integration tests interact with multiple packages following their functionality, in their attempt to check full system compatibility.

4.2.6 *view*

Purpose: To allow for user interaction with the system through communication with the host along with its application logic and data.

Functionality: The view package operates as a thin client, taking user input through a graphic user interface (GUI) [14]. User data is cached on the client side for the sake of user convenience, decreasing the input required to achieve functionality. Host responses, along with operational errors are displayed for user viewing. Errors are not logged on the client side of the application.

Interaction: The component interacts with the user and the host module of the *controller* package.

4.2.7 *webscraper*

Purpose: The package completes data gathering services from the amalgamation of preset websites and parameters representing selected NBA games, in order to supply historic statistical game data for the application.

Functionality: The acquisition process extracts data from a combination of preset URLs, set to match parameterized game data originating from the user. It includes functionality to interact with each URL in a manner defined by the user's chosen scraping method. Collected information is preprocessed and transformed before being committed to memory in a preset reusable format, easily accessed by the *controller* as it looks to the *webscraper* to acquire new information. Web scraping requests are generally not repeated, as the system is built to house already accessed data, thereby

minimizing dependency on online sources to bare necessity. The component's sequence diagram (see 4.4) illustrates the process during runtime.

Interaction: The *webscraper* package primarily communicates with the *controller* in order to receive data acquisition requests and parameters. Its preprocessed results are also returned to the *controller*. The package's component diagram (see 4.5) illustrates the *webscraper*'s interactions. Logging is utilized throughout the process.

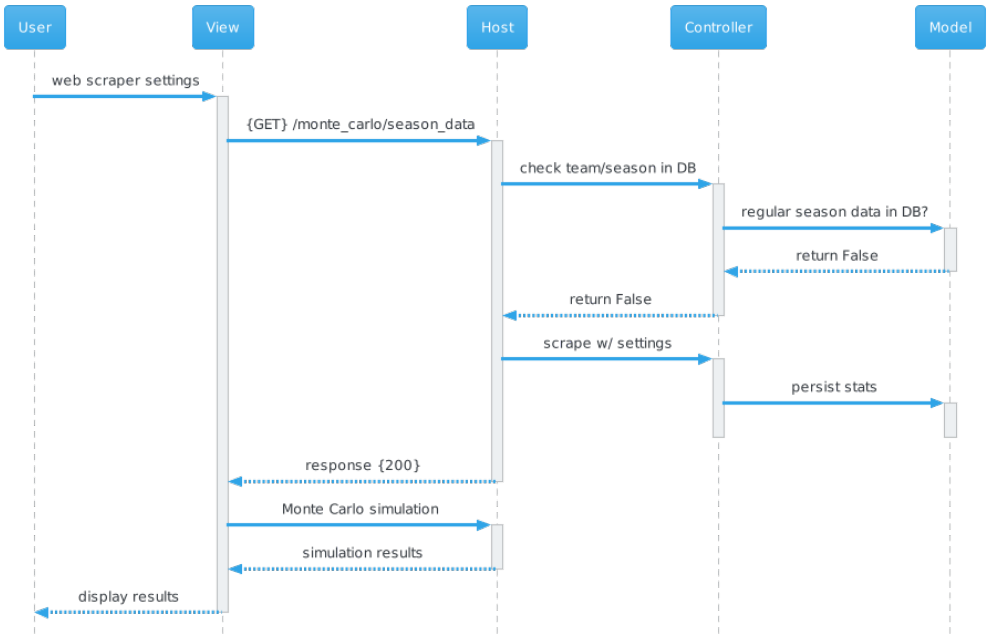


Figure 4.4: Web Scraping UML Sequence Diagram

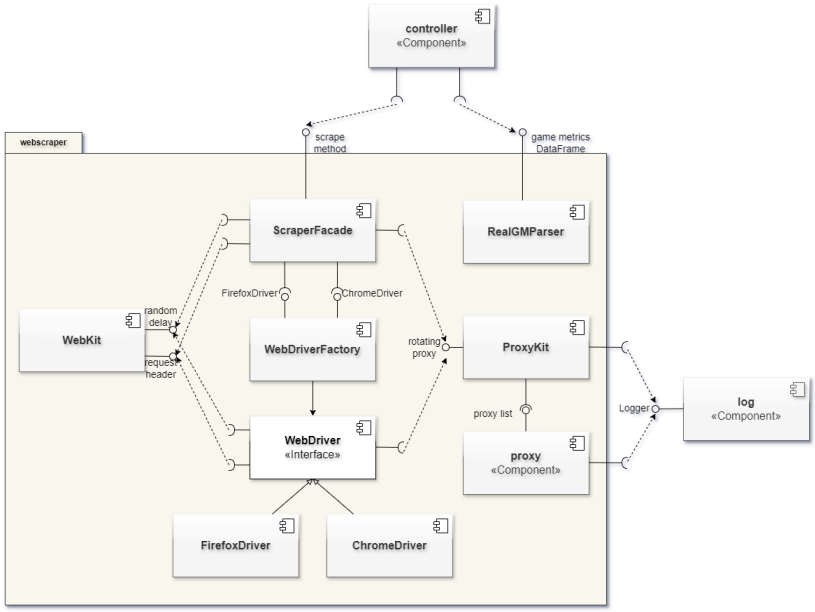


Figure 4.5: *webscraper* UML Component Diagram

4.3 Database Architecture

The database consists of two main table types: utility tables and tables containing relevant game metrics. The utility tables are static, and assist in the application's game related logic, while metrics tables house data scraped from the web to be utilized in the simulation process.

4.3.1 Utility Tables

***in_playoffs*:** This is a boolean table pairing each NBA team (columns) to a season of game-play (records). Teams that made the playoffs receive a 1, while teams that failed to make the playoffs in the given season are left NULL.

***schedule*:** The table houses historic NBA game dates and outcomes, along with the game type (regular or playoff game). Records range from the commencement of the 1990-1991 season until the end of the 2022-2023.

4.3.2 Metrics Tables

There are three types of metrics tables:

- **Individual game tables** contain game data for each individual NBA game. Multiple player records are available for each date, with stats portraying the players performance for the single game. Separate tables exist for regular season and playoff games.
- **Player tables** are unique to each player of each team. They represent the player's averages for the given season. Due to the differences in performance, separate tables exist for home and away games, but also for regular season and playoff games, making a total of four player tables.
- **Team tables** house averages pertaining to the each team's performance in the given season, therefore each team is unique to each season. Once again, due to the high fluctuation between regular season and playoff metrics, tables are separated into these two categories.

To ensure all information remains unique, yet no new data is erroneously deemed as a duplicate, all cells of a record are expected to be unique in each table. This constraint ensures data validation, as a player will not be able to appear twice in relation to a game date with all the same metrics, or in the same season. Teams undergo the same validation process, being unique for each season.

Table metrics are presented in more detail during the 5.4 *simulation* and 5.6 *web-scraper* sections of the Implementation chapter.

4.4 Technologies and Frameworks

4.4.1 .NET

The .NET framework was built by Microsoft for developing Windows-based applications. Today, with the integration of cross-platform and open source frameworks, the .NET Core supports multiple operating systems and is also open source. It supports multiple programming languages, including C#. [36]

The view component of the application is based on the .Net framework and written in C#, due to its ability to model a range of services. It's WinForms (see 4.4.15) framework and RestSharp package allow for fast and secure development, as well as a seamless user exprience.

4.4.2 Beautiful Soup 4

Beautiful Soup [17] is a popular python package, designed to extracts data from HTML [4] and XML [5] documents sourced from the web. By leveraging attributes unique to the selected markup language, it facilitates tag and text content based parsing in order to precisely identify and extract the desired data. Developed for web scarping purposes, it is often used alongside packages responsible for making content requests to websites.

This was also the case in this project. Beautiful Soup was a great asset during the allocation and extraction of acquired web contents, which could then be passed on for further formatting and storage.

4.4.3 Fake User Agent

Throughout the extraction of data from websites it is crucial to consider the exchange of information between the client making the request, and the website's host server responding to the request. During this process, the host receives client information in the header, allowing it to discern information about the requesting client, like its operating system and browser type (see subsection 4.4.5 for information on HTTP requests and their headers).

Operations requiring repeated content requests to a single host, such as web scraping, can therefore be identified as outside the scope of regular usage requirements provided by the website. This in turn may lead to limitations of service, disrupting web scraping-based system operations.

[4]HyperText Markup Language (HTML) is a markup language used to house online content and set its layout, thereby creating the basic structure of web pages [16].

[5]Extensible Markup Language (XML) is a markup language for data transfer and storage. Rather than offer preset tags, users can organize content subjectively through the application of a key value pair data structure [18].

Fake user agent applications are designed to solve this problem. Python's *fake-useragent* library allows for the creation of random user agent headers, which can be assigned to requests [20]. Users have the option to select from a wide range of preset options, which can also be filtered to only mimic specific browser technologies and operating systems. While this method only changes a portion of the request data, when used alongside other methodologies promoting anonymity, it can prove to be a powerful tool.

The package was employed in the project for this ability to generate random user agents at runtime, thereby contributing to the application's anonymity while web scraping.

4.4.4 Flask

Flask is a Python based web framework created to facilitate the development of web applications. The framework is known for its versatility, and aims to provide a minimalistic and flexible approach in comparison with other web application frameworks [26].

Developers can choose their server platforms and environments, as Flask's WSGI (Web Server Gateway Interface) specifications enable it to integrate with a multitude of web servers, such as Apache and Nginx [25].

Flask also contains a built-in development server, which provides application testing and debugging features. It also supports other extensions and libraries, granting integration capabilities to developers to handle session management or database connectivity. The framework focuses on the provision of necessary tools instead of enforcing constraints, thereby making Flask a popular web framework [25].

The framework was employed in the application due in part to its ease-of-use and flexibility. The host required a lightweight solution compatible with Python. A robust strict framework such as Django would have hindered the timely development of the project, and Flask's accessibility made it a great candidate.

4.4.5 HTTP

Hypertext Transfer Protocol (HTTP) is a method of communication employed to facilitate data exchange between servers hosting web-based resources, and clients such as web browsers or web-applications [19].

HTTP is unidirectional, operating on a request-response basis. When the client, such as a web browser requests services in the form of online content from a server, it does so through an HTTP request. The server processes the request, and sends an appropriate HTTP response. Data can be transmitted in multiple formats. Common

examples include JSON [6] , HTML, and XML [23].

The client's request contains information such as the request line, message body, and headers [21]. The request line contains the destination resource, the message body stores parameters used by request methods, and the headers contain information about the client's choice of operating system and web browser. The response is comprised of a status code, headers and message body, which houses the requested data [23].

HTTP offers methods which can apply parameterized actions to the selected resource. Some examples include:

- The GET method presents the specified resource.
- POST submits data parameters to the server which are required for further functionality.
- PUT is used to update a selected resource on the server.
- The DELETE method removes the parameterized resource from the server.

Status codes are utilized to indicate the outcome of processes initiated by the client's HTTP request. They are categorized into groups, covering categories such as informational responses of success as well as errors. Some common examples of success codes include:

- 200: The requested transaction was completed successfully and the requested content is returned.
- 204: The server processed the request, and is not returning any content.
- 400: A client error obstructed the server's attempt to complete the request.
- 404: The server does not contain the requested resource.
- 500: Indicates the occurrence of an internal server error, obstructing the fulfillment of the client request.

Due to security concerns, a more secure version of HTTP Secure (HTTPS) [24] was created. It employs encryption mechanisms to ensure transaction security, thereby thwarting potential eavesdropping and tampering attempts.

[6] JavaScript Object Notation (JSON) is a popular lightweigth data structure which utilizes key-value pair functionality. Due to it's simplicity and compatibility with many programming languages it is a common choice for online information exchange [22].

4.4.6 Matplotlib

Matplotlib is an extensive community-maintained Python library utilized in the creation of static, animated, and interactive visualizations [27]. Matplotlib offers a wide range of functionalities, making it a versatile tool for data visualization tasks. It's Pyplot module allows for the creation of customizable and embeddable graphs and diagrams.

Ultimately the plotting capabilities of the package's Pyplot module led to its utilization in the system.

4.4.7 MySQL

MySQL is a free open source relational database management system (RDBMS). It is widely used due to it's speed, reliability and scalability. Structured Query Language (SQL) is utilized to communicate with the database for data management and user access modifications. [28]

The decision to use MySQL as the application's database management system was reached due to several factors. The application required a relational model for easier storage of acquired statistical data, which in turn allowed easier processing upon extraction from the database. Its reliability and complementary open source nature, coupled with its widespread adoption and accessibility solidified the decision to utilize it in the application.

4.4.8 Pandas

The pandas library is an open source project developed for Python by Wes McKinney and Chang She during their time at AQR Capital Management. The developers sought to attain the tabular functionality of DataFrames [7] in the R programming language for their flexibility and functionality in working with financial data. [29]

Pandas remains an open source library to this day, and has become very popular for its versatility and functionality in data analysis endeavors. It has been employed in this project due to this high level of tabular functionality and data accessibility, along with its efficiency in parsing tabular data into DataFrames.

4.4.9 Python

Python is a high level open source object-oriented programming language, with features such as dynamic typing, dynamic binding, and built in data structures. It is an

[7] DataFrames are two-dimensional tabular data structures in pandas, housing data accessible by rows and columns. Visually, DataFrames resemble spreadsheets, while structurally they key-value pair data structures. [29]

interpreted language with a modular structure, and an easily readable syntax. [30]

The programming language was utilized for the back-end due to its extensive complementary open source library. There availability of multiple web scraping packages, along with powerful data analytics libraries such as pandas or matplotlib allow for convenient and fast paced development.

4.4.10 Requests

The Python requests library is a simple package allowing for communication through HTTP requests. [31] It is utilized in the application for making requests to specified websites in order to retrieve their HTML content in the response. Requests allows for parameterized customization of proxies, timeout settings, and request methods, many of which are utilized in the web scraping process.

4.4.11 REST API

Representational State Transfer (REST) refers to a set of principles acting as guidelines in the development of APIs for web services. These stipulate the use of HTTP as a communication protocol, however data encoding is left up to the developer, with options including JSON, HTML, and XML. Requests should be independent, not relying on each-other's success. Caching of data is acceptable, along with the sending of executable code to the client where needed. REST further requires a standard method of sending data and a layered organizational approach. [33]

In the application, REST is utilized to facilitate communication between the client and host. HTTP requests are sent to the host in order to receive responses in the form of JSON.

4.4.12 RestSharp

RestSharp is a tool utilized in .NET development, offering synchronous and asynchronous communication to remote resources using HTTP. It allows for easier managing of diverse request and response types while interacting with APIs by handling serialization and deserialization of message bodies to JSON and XML. [32]

The package is utilized on the client side of the application, due to these capabilities. Its GET and POST methods handle interactions with the host's API. Response JSONs are deserialized to access response content.

4.4.13 Selenium

Selenium is an open source test automation tool created for web applications. It enables developers to single out and interact with user interface (UI) elements thereby

simulating user interaction. Selenium's WebDriver tool allows for the utilization of web browsers to interact with online sources, enabling programmatic clicking of buttons, mouse movements, and traversing web pages. Selenium supports multiple programming languages, including Python. [34]

While driver instantiation already allows for the use of well known browsers, such as Mozilla's Firefox and Google's Chrome, the open source community has further contributed packages such as Undetected ChromeDriver. This Python library aims to provide the same driver functionality, while attempting to be less detectable by the host server. [35]

This project utilizes Python's Selenium package for web scraping purposes. Instantiating the driver allows for content retrieval in a less detectable, albeit slower manner.

4.4.14 TestStack.White

White is a test automation framework used to test Windows desktop applications. It is based on the .NET framework and does not require the use of scripting languages. Test code can be written in any .NET supported language. [38]

The framework is implemented in the project for UI automation testing purposes. The client's WinForms-based desktop application is tested for returning appropriate responses and basic functionality for quality assurance purposes.

4.4.15 WinForms

Windows Forms (WinForms) is a .NET graphical user interface (GUI) framework for building desktop applications. It provides developers with controls for dragging and dropping elements to rapidly create interactive user interfaces. [39]

The application's client interface is built with WinForms in the C# programming language. WinForms allowed for timely and precise development process, creating a visually appealing Windows desktop application.

4.4.16 XAMPP

Cross platform Apache MariaDB PHP Pearl (XAMPP) is a complementary open source web server solution stack meant for development environment utilization. Originally developed by Apache Friends, the application is widely used for testing and development purposes. It's available for multiple operating systems and makes the conversion process to a live server seamless, as it utilizes the same tech stack utilized by most production environments. [37]

MariaDB, the open source database server used by XAMPP, is a fork of the the original MySQL. It was created with the intention that the project remain free and open source. [40] As a fork of MySQL, it shares a high level of compatibility with it, and has been included in the project for this purpose. While XAMPP employs MariaDB, the project's business logic utilizes MySQL related code and Python libraries. This compatibility allows the application to access both MySQL servers and XAMPP's MariaDB-based service.

XAMPP is utilized as a development environment in the project, housing the application's built in database. It was chosen for its robust functionality and online user interface, which made testing during the creation and implementation of the model a more user-friendly experience.

Chapter 5

Implementation

In keeping with the structural approach of the Architecture chapter, the project's implementation is presented by package. Each section will delve deeper into the modular and functional breakdown of the project, along with their application. As other components require its presence, the *model* package is introduced first, along with an introduction of the business logic.

5.1 *model*

The *model* package contains two classes: *Teams*, and *MySQLHandler*. They represent the business logic, responsible for encapsulation of the database's operations for data manipulation and contain the enum ensuring team names are handled appropriately throughout the application. As figure 5.1's class diagram shows, the package mainly services the *controller*, with the *simulator* also utilizing team enums for smoother operations.

5.1.1 *Teams* Enum

The Teams enumeration represents each NBA team in the database in all forms of their use throughout the application. These are:

- **Full Name**: represents both the city and team name (e.g.: Chicago Bulls).
- **Short Name**: short form of the team's city (e.g.: CHI).
- **Link Name**: both the city and team's name along with their unique id number as used in RealGM website URLs (e.g.: Chicago-Bulls/4).
- **City Name**: only the city the team plays for (e.g.: Chicago).

The class contains logic to create enums from each of these name forms, along with the logic for displaying their converted values.

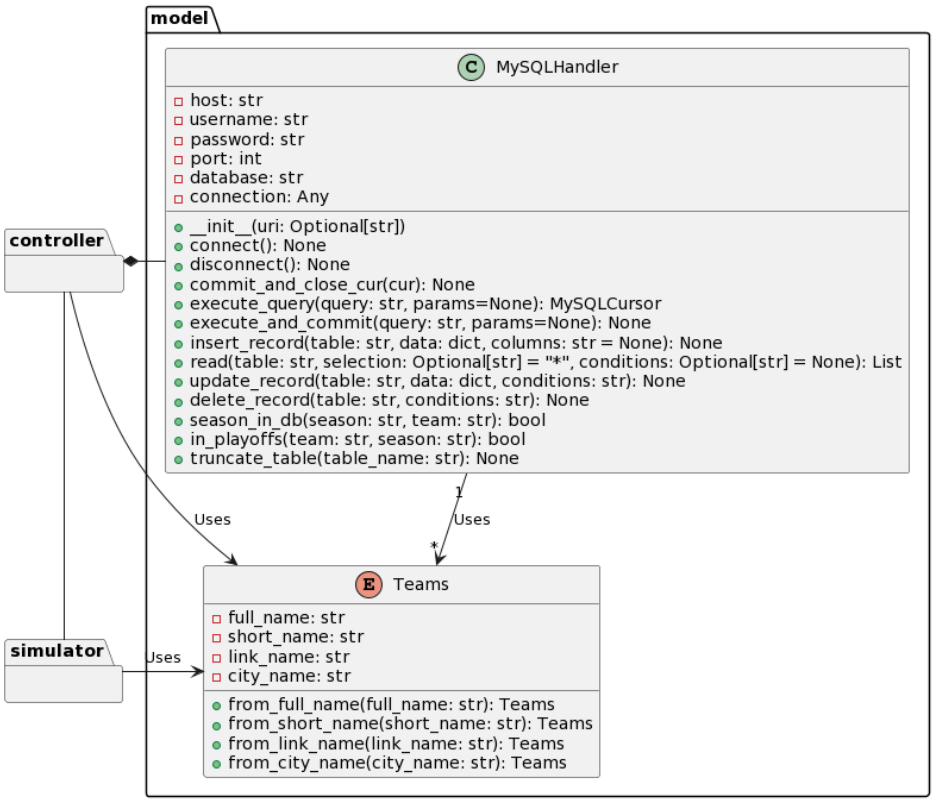


Figure 5.1: *model* UML Class Diagram

5.1.2 *MySQLHandler*

The class facilitates communication with a MySQL or MariaDB database, initialized through the parameterized URL string. The connection string is parsed to allocate the correct sub-strings of the parameter to each class attribute (such as host, username, password, and port). Python's *mysql.connector* library (see [44] for details) then proceeds to establish a connection to the database, allowing for use of the provided data management methods.

This initialization method allows the application to establish a connection at will with the user's database of choice.

Data management functionality is customizable and secure, due to cursor parameterization with placeholders for dynamically inserting values into SQL queries. This decreases the likelihood of injection attacks, which would otherwise modify the code with malicious intent, or access unauthorized information [45].

Data modification proceeds through a query execution function, which initializes a cursor and attempts to execute the parameterized query string. Changes are rolled back upon encountering potential connection errors. Code Extract 5.1 illustrates the source code of the method. Upon successful completion of the operation, a separate function handles the commit to the database, and closes the cursor.

Code Extract 5.1: Cursor Execution of Query

```
1     def execute_query(self, query: str, params=None) -> mysql
          .connector.connection.MySQLCursor:
2         cursor = self.connection.cursor()
3         try:
4             if params:
5                 cursor.execute(query, params)
6             else:
7                 cursor.execute(query)
8             return cursor
9         except mysql.connector.Error as e:
10            self.connection.rollback()
11            raise ConnectionError(f"Error executing query: {e
                  }") from e
```

While the client side of the application does not require update, delete, and truncate methods, the functionality has been included in the data access layer to enable comprehensive data management from the back-end.

5.2 *controller*

The package is responsible for the implementation of three high level functionalities: the API host service, web scraping, and constructing *TeamBuilder* NBA team instances for the simulator. Each of these controls manage a designated section of the back-end logic's operations.

As discussed in the Architecture chapter (see 4), the *Host* amalgamates final operations of both web scraping -, and Monte Carlo simulation services. Web scraping operations are coordinated by the *controller*'s *ScrapeControl* class, while NBA team instantiation is handled by the *TeamBuilder*. Team instances are provided for the *simulator* package, which in turn services the *Host*. The following sub-sections presents each functionality in detail.

5.2.1 Host Service

The *host_service.py* module houses the *Host* class containing the Flask server handling requests pertaining to the functionality provided by the above described *webscraper* and *simulator* packages. Server instances handle these requests through the following endpoints and their respective class methods:

- **/monte_carlo/game_data**: Its get_game_data() method reads the game schedule from the database based on the provided arguments, returning a JSON

of the records to the client.

- **/monte_carlo/team_in_db**: The get_teams_in_db() method checks if the selected and previous season's data is present in the database for both teams, notifying the client of its findings.
- **/monte_carlo/season_data**: Its get_season_data() method initiates a scrape of missing game data. In case of failure, the method re-initiates the process for a second time, logging the attempt. The client is notified of the operation's success or failure via a REST response. In case of failure, the method removes records from the players table, thereby ensuring future checks will see it is still missing from the database.
- **/monte_carlo/simulation**: The get_monte_carlo_sim() initiates the parameterized Monte Carlo simulation, returning its results to the client. The JSON contains the probability density plot and violin plots as base64-encoded strings of the plot image files created by matplotlib. Basic game statistics are also included, such as modes of the score arrays, and win percentage.

Flask instances are initialized with a timeout of 500 seconds, just over 8 min. allowing adequate time for slightly longer simulations (see line 5 in Code Extract 5.2).

Utilizing matplotlib to create graphs during server runtime was challenging at first. When the server instance was running alongside the WinForms GUI, threading conflicts would randomly occur. This was because matplotlib's default back-end GUIs are not guaranteed to be thread-safe, therefore WinForms would occasionally attempt to utilize a resource already allocated to the plotting function. The solution was to set matplotlib to the Anti-Grain Geometry (Agg) plotting library (see line 3 in Code Extract 5.2), which does not require the use of said resources. [41]

Code Extract 5.2: Flask Instance Initialization

```
1     def __init__(self):
2         # non-interactive rendering env
3         matplotlib.use('Agg')
4         self.app = Flask(__name__)
5         self.app.config['TIMEOUT'] = 500
6         self.log = Logger(log_file="application_log.log",
7                           name="FLASK HOST",
8                           log_level="INFO")
```

5.2.2 Web Scraping

The *controller* package's web scraping functionality is encapsulated in two modules: *control_service* and *dto_service*. Each module's classes contribute specialized logic to the data gathering process. Together, the modules orchestrate the web scraping process, handle the transformation of data in memory, and manage persistence operations through interactions with the business logic. Figure 5.2 illustrates the package's class diagram pertaining to web scraping and data allocation logic.

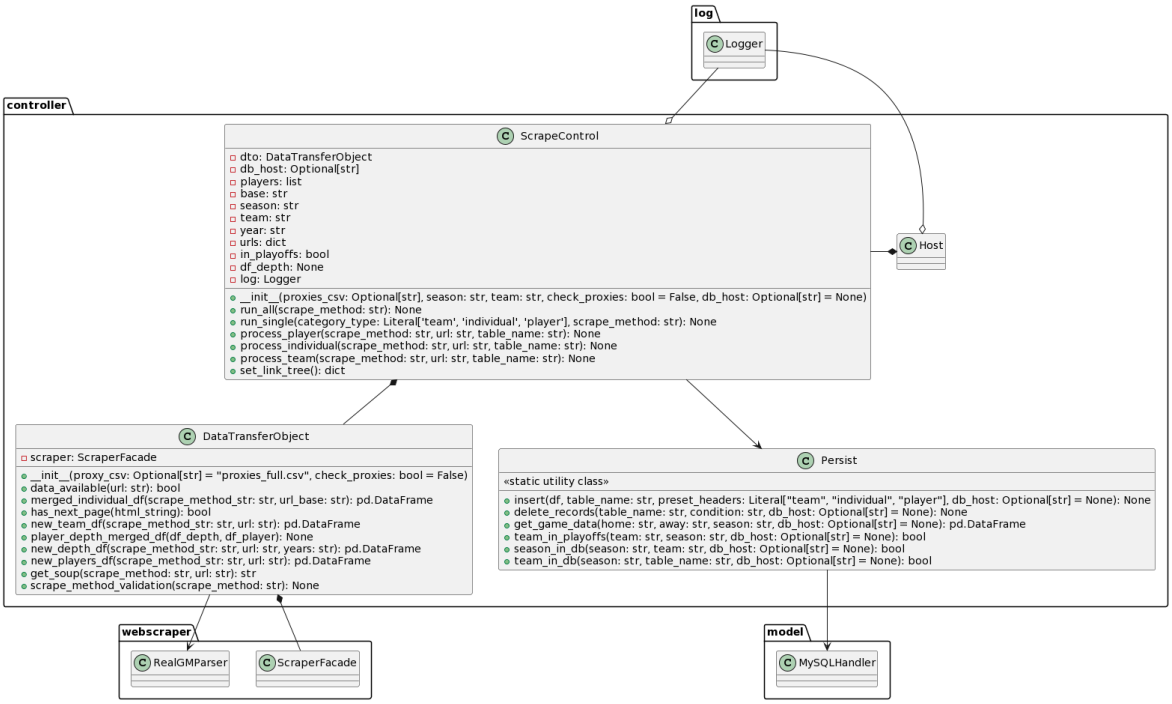


Figure 5.2: The *controller*'s Web Scraping Services UML Class Diagram

The *control_service* module houses the *ScrapeControl* class. During initialization it:

- Instantiates a *DataTransferObject* (see section 5.2.2) utilizing the specified proxy settings.
- Creates a dictionary of key-value pairs comprised of the parameterized URLs for each web page it intends to scrape data from.
- Assesses if the team is in the playoffs, storing a boolean from the result of the function call to *Persist.team_in_playoffs*.
- Initializes a logger instance, and sets the necessary attributes for further operations.

The class is comprised of basic logic for scraping each table type, stored in the database: player, individual game, and team statistics. These are utilized by two main operating functions: *run_single* and *run_all*. This is because the application is designed to allow for simple use through a client interface, but also complex back-end use for a more detailed control of the process as required by analysts. The *run_all* method completes a full scrape for the chosen team and season, while the *run_single* method takes parameters allowing for further specification of the table type to acquire and populate (player, individual game, or team statistics).

The dictionary data structure housing the URLs allows for a controlled and detailed iteration, utilizing the assessment of the team's participation in playoffs for the given season set up during the instance's initialization. While iterating over the URLs, the appropriate scrape methods are utilized to attain raw HTML data as text, parse it to a Pandas DataFrame, and finally persist the data to the database. This functionality is further discussed in section 5.6 titled *webscraper*.

The *dto_service* module contains the *DataTransferObject* and *Persist* classes. *DataTransferObject* instances facilitate the retrieval of data through *ScraperFacade* instances (see section 5.6). Users have the option to choose from a variety of scrape methods, with or without the utilization of proxies. The class further provides logic for handling data sourced from each pre-determined URL, flipping through pages where they are available, and storing them in Pandas DataFrames.

Beyond utilizing services encapsulated by the *webscraper* package, the class aims to alter and store data in memory for easier processing and better accessibility. Utilizing Pandas DataFrames as a data transfer object [1] allows the class to compartmentalize data in structures that are easy to handle, debug, view and transfer to database tables.

With a collection of static utility methods, the *Persist* class is responsible for handling data management operations within the application's database. Trough interaction with the *MySQLHandler*'s business logic, it facilitates the insertion, deletion, and retrieval of data.

5.2.3 Team Instantiation

The *dto_sim* module contains three classes responsible for constructing NBA team instances: *TeamBuilder*, *Roster*, and *Player*. *TeamBuilder* instances create and house all data pertaining to teams participating in the match, with each team (home, and away) being a separate *TeamBuilder* instance. Data is restructured and housed in *Player* instances, which in turn are allocated to the appropriate position in the team's *Roster*.

[1] Data transfer objects (DTO) facilitate data transfer between processes, where the only behavior allowed to the data structure is the storage of data [42].

Once completed, team instances are utilized by the *simulator* package's *GameBuilder* class in the construction of individual game simulations. Figure 5.3 illustrates the class diagram relevant to the three classes responsible for team instantiation.

Figure 5.3: The *controller*'s TeamBuilder UML Class Diagram

Roster instances are only meant to function as iterable containers. They are comprised of five empty lists upon initialization, each representing a playable position. These are:

- PG - Point Guard: coordinates offense.
- SG - Shooting Guard: driving forward and scoring.
- PF - Power Forward: role is a mix between forward a center.
- SF - Small Forward: distance and inside scoring.
- C - Center: under basket game.

Each position list houses the *Player* instances assigned to the respective position. The *Roster* itself is also iterable.

Player objects store game statistics pertaining to the individual player. Even though I attempted to keep sports analytics to a minimum during the project, the list of attributes grew quite extensive. The majority of the statistics are derived from information already stored in the database. The exceptions are game metrics utilized as weights for decision making by the *simulator*. Highlights of these metrics and the logic responsible for their calculation:

- **adjusted_points**:
  Adjusted points are an attempt to numerically categorize a player's true effectiveness via their affect on the team's offense, due to the player's direct contributions through points scored, rebounds, assists, etc. The are multiple approaches for this calculation. This project utilizes the model put forth by Jim Lackritz[2] and Ira Horowitz[3], available at the San Diego State University Fowler College Sports Business Program [43].

$$\frac{3 \times \text{Total Points}}{4} + \frac{2.383 \times \text{Assists}}{4} + \frac{0.588 \times \text{Off. Rebounds}}{4} + \frac{0.530 \times \text{Steals}}{4}$$

(Model 5.1: Lackritz's Adjusted Points Equation)

- **player_w**:
  A player's weight is the metric allowing the Monte Carlo simulation to make random weighted decisions about a players participation in the specified game. It is calculated by averaging the player's adjusted points percentage and playtime weight.

- **playtime_w**:
  A players playtime weight is my arbitrary representation of a weight representing the player's probability to be on the court for any given possession. It utilizes the player's depth in the official roster for the season, and the player's average minutes played during the season divided by the number of minutes per basketball game (48). Second string players probability is then decreased to a quarter of its original value, why third string players are decreased even more, to 2% of their original playtime. All other players are not considered during the simulation.

  This is representative of a players playtime in a real life scenario, where third-string players will generally not see a lot of playtime unless the match itself is a statistical outlier. This can mean either injury, or the team performing so well or so poorly, that novice players can be introduced without detrimentally affecting the outcome of the match.

[2]Jim Lackritz is the co-founder of the Sports Business MBA program at San Diego State University's Fowler College [43].

[3]Ira Horowitz is an emeritus professor of statistics at the University of Florida [43].

This approach is a simplified representation of basketball analytics. More professional methods exist for calculating player utilization, however they are outside the scope of this project as it endeavors to approach the subject matter from a computer science perspective at the BSc level.

- **totalPpercent**:
  A player's total point percentage is calculated by averaging the percentage of their success rate from free throws, three pointers and field goals.

TeamBuilder instances are initialized, with the following operations:

- Validating the game date to ensure it falls within the required season.
- Creation of a Pandas DataFrame for each table category in the database:
  - Individual games per player for specific game date.
  - Player table of seasonal averages.
  - Team table containing seasonal averages pertaining to the team.
- The creation, population, and validation of a *Roster* instance.
- Populating the necessary attributes for their functionality.

It is important to note, that the three initialized DataFrames are filtered specifically to the parameterized data unique to each game. This decreases the demand on memory. Team statistics will therefore only contain one record for the instance's team in the season. Player statistics remain unchanged. The individual games table, however, is restructured, as player instance data is calculated from these.

The team's individual games table consists of one calendar year's data leading up to the game date. Code Extract 5.3 illustrates the source code responsible for compiling the DataFrame.

Code Extract 5.3: Initializing Individual Games DataFrame

```
1 def get_individual_data(self, team_short:str, game_date:str,
2                          game_type:str) -> pd.DataFrame:
3         condition = (f'team="{team_short}" AND '
4                 f'date >= "{int(game_date[:4]) - 1}{game_date
                    [4:]}" '
5                 f'AND date < "{game_date}" ORDER BY date')
6         table = f"individual_games_{game_type}"
7         return self.get_data(condition, table)
```

The DataFrame is then modified, repeating records pertaining to the last 2-3 games (those in the 98.5% time span threshold of the game date) five times. Games falling within the 80% threshold of the game date are repeated three times, while games in the more distant past only appear once in the re-constructed DataFrame. As all player data is reconstructed from stats in this DataFrame, more recent games will carry a greater weight in the player's performance. This is intended to represent the psychological affect an individual's recent performance has on their current output.

The *build_roster* method constructs *Player* instances from the assembled DataFrames, populating them in their respective positions in the team's *Roster*. This is done by iterating over each player in the players DataFrame. A boolean test ensures the player is assigned to a position (novice players that don't see playtime are not always assigned to positions). This is a preliminary filter to once again decrease the strain on memory. Second, the logic ensures the player is present in the individual games DataFrame. This ensures the player has seen playtime in the past 365 days. Players that do not meet this criteria are overlooked during the team construction process, as their chances of seeing playtime would be minuscule in reality as well. This further decreases demand on memory and computing time. All players who meet the requirements of the test logic are instantiated utilizing the individual games DataFrame filtered to the player's stats, and additional stats from the player DataFrame. Code Extract 5.4 illustrates the testing and appending functionality of the method.

Code Extract 5.4: Code Snippet of Roster Construction

```
for _, row in pl_df.iterrows():
    test = row["position"] and row["player"] in self.
        individual_df['player'].unique()
    if test:
        position = getattr(self.roster, row["position"])
        position.append(Player(self.individual_df
            .loc[self.individual_df['player'] ==
                row["player"], :], row["player"], row["GP"],
                row["depth"], row["MPG"]))
```

The finished *TeamBuilder* instance is then easily utilized by the simulation's *GameBuilder*. The encapsulated data allows for fast access in a logical and intuitive structure.

5.3 *log*

The logging system of the application is designed to be flexible, allowing for easy integration into different components of the application. Each component uses the same log, sometimes simultaneously, therefore the logger allows for the creation of

access instances. Each is identified separately by module or functionality allowing for easier debugging of runtime errors.

The *logger* module houses a single *Logger* class, which provides this functionality to the *webscraper* and *controller* packages of the application. Other package functionality is logged by the *controller*. Figure 5.4 depicts the *Logger*'s class diagram. The package also contains the application's logs.

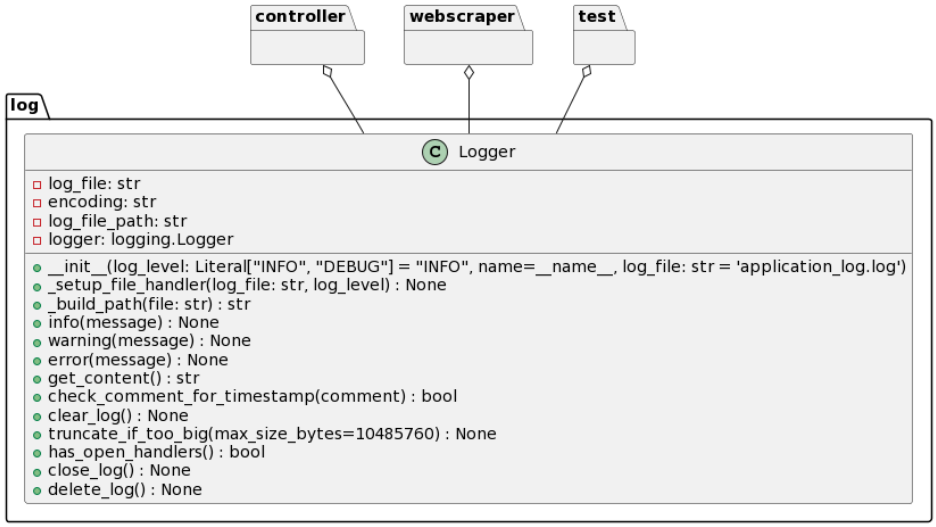


Figure 5.4: Logger UML Class Diagram

By default *Logger* instances receive a log level of INFO and are referred to the *application_log.log*, however parameters exist to alter these settings upon initialization.

Instances complete the initialization process by ensuring the log size has not exceeded the allotted 10 MB threshold. Upon passing this benchmark, truncation is ensured with the next instantiation.

The service allows the application to log messages of varying severity levels through the use of the *info*, *warning*, and *error* methods. Messages logged under the base log level are omitted from the log file. This functionality allows different components to access the log for different purposes, and allow better control of what gets logged during debugging, testing, or runtime.

The logger also supports advanced operations, useful during testing and debugging, such as retrieving log file contents, patterns checking log contents, and managing the file itself.

5.4 *simulation*

The package is responsible for setting up and running the Monte Carlo simulation, returning its findings for the parameterized epochs to the *controller* package's host

service. The package consists of four classes: *GameBuilder*, *Simulation*, *MonteCarlo*, and the *GameTools* static utility class.

The package's simulation constructor classes, the *GameBuilder* and *Simulation* classes, both utilize the static utility methods the *GameTools* class in their logic. The enum service provided by the *model* package is also used throughout the *simulator* package, as depicted in Figure 5.5 displaying the *simulator*'s class diagram.

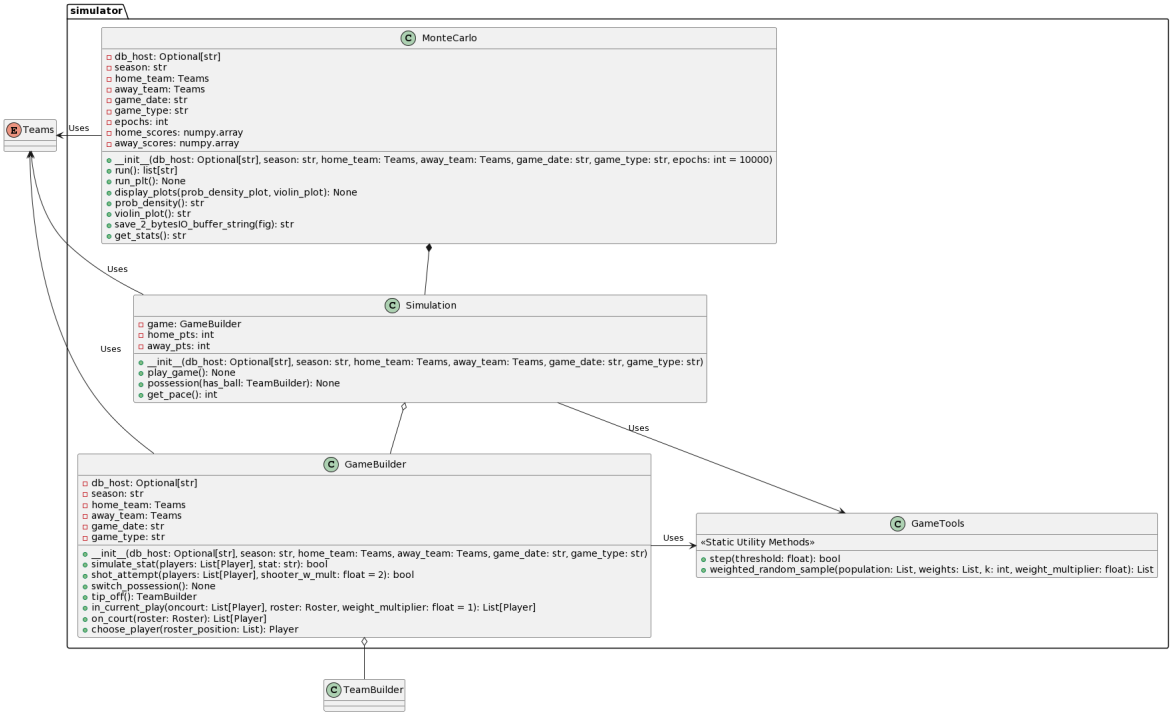


Figure 5.5: *simulator* UML Class Diagram

5.4.1 *GameTools*

The class contains static methods that provide non-deterministic results for the simulation process. This is done by introducing randomness through thresholds and weights in decision making. These methods are static for convenience upon use.

The Step Method: This method makes decisions based on one parameterized metric, returning a boolean value of success or failure. A random float is generated between 0 and 1, and if it does not exceed the threshold (which is a float with the same boundaries) then the method returns a successful attempt. The threshold is utilized as an upper boundary, because higher statistical averages would otherwise lead to a smaller chance of success, requiring the negation of the results. Code Extract 5.5 illustrates the method.

Code Extract 5.5: Step Method

```
1     @staticmethod
2     def step(threshold: float) -> bool:
3         if random.uniform(0, 1) < threshold:
4             return True
5         return False
```

Player Sampling: This method randomly selects players from a parameterized pool of options, applying weights to the decision making process. The effect of the weights on the selection process can be increased through the weight multiplier (which exponentiates the value) parameter. The required sample size is represented by k. Code Extract 5.6 illustrates the method.

Code Extract 5.6: Weighted Random Player Sampoling Method

```
1     @staticmethod
2     def weighted_random_sample(population: List, weights:
         List, k: int, weight_multiplier: float) -> List:
3         result = set()
4
5         while len(result) < k:
6             w = [math.pow(_, weight_multiplier) for _ in
                 weights]
7             i = random.choices(range(len(population)),
                 weights=w)[0]
8             if population[i] not in result:
9                 result.add(population[i])
10        return list(result)
```

5.4.2 *GameBuilder*

The class utilizes *TeamBuilder* instances to create the home and away teams parameterized during initialization. Its methods are focused on performing game related interactions, which assist the *Simulation* class in completing the Monte Carlo simulation.

Proceeding in chronological order relating to game-flow, the *tip_off* method simulates a tip off, deciding which team gets to attack first. The method selects the strongest rebounder (the player with the highest amount of offensive and defensive rebounds) from each team, then adds a home-court advantage of half the rebound delta between the home and away player's score. It then utilizes the *GameTools*' step method to

determine if the home team won the tip off. The team instance with possession of the ball is returned, and starts the flow of the game by filling the *has_ball* attribute.

Each possession randomly samples the players on the court, however not all players on the court participate in active game-play at all times. Therefore, the players on court are once again randomly sampled to decide which of them will actively participate in that single possession. The player sampling method is utilized to make these random selections. Each player's playtime_w attribute serves as their respective weight in determining their likelihood of court time and active play time once on court. A combination of the *on_court*, *choose_player*, and *in_current_play* methods provide this functionality.

The *simualte_stat* and *shot_attempt* methods randomly decide if their respective events come to pass by utilizing the step method. The first is meant to simulate metrics such as turn overs (when a player looses the ball for any reason), among other, and allows for further development of the simulation through its utilization on additional metrics to improve results.

The *shot_attempt* method is specifically meant to deal with scoring attempts. It first decides who, of the players actively participating in the play will attempt to score the point. Then their shot metrics are reviewed to decide what type of shot they will attempt (3 point or regular 2 point). Code Extract **??** shows the method.

Code Extract 5.7: Weighted Random Player Sampoling Method

```
1     def shot_attempt(self, players: List[Player],
          shooter_w_mult: float = 2) -> bool:
2         # who takes shot
3         w = [p.player_w + (p.FGA + p.threePA) / 200 for p in
              players]
4         shooter = GameTools.weighted_random_sample(players, w
              , 1, shooter_w_mult)[0]
5         # type of shot attempted
6         shot_points_w = [shooter.FGA, shooter.threePA]
7         points = GameTools.weighted_random_sample([2, 3],
              shot_points_w, 1, 1)[0]
8         if points == 2:
9             return GameTools.step(shooter.FGpercent/100) *
                  points
10        return GameTools.step(shooter.threePpercent/100) *
              points
```

Finally, the *switch_possession* method awards the ball to the opposing team, allowing them an offensive attempt in turn. Further functionality is added to the simulation through the utilization of these methods by the *Simulation* class.

5.4.3 *Simulation*

The class constructs a simulation instance which will play a single basketball game. The parameters and data reconstruct a specific historical match. Upon initialization, the class utilizes the *GameBuilder* class to construct the game scenario, before proceeding with the simulation and returning the final score.

The pace[4] of the game is determined by the combination of each teams relevant metric, providing the required number of possession iterations to conclude the match.

The *possession* method applies the relevant logic encapsulated by the *GameBuilder* instance to the parameterized *TeamBuilder* object. After deciding which players are on court, and which of these are actively involved in play, the possibility of a turnover is reviewed. If the ball is turned over, the opposing team continues with their possession, otherwise a shot attempt is made. If the shot is unsuccessful, the rebound metrics are simulated to determine if the offensive team will attack again (this does not count as a new possession in the iteration), otherwise the ball is turned over to the opposing team.

After the allotted number of iterations have concluded, the simulation can present the home and away scores representing the outcome of the single match.

5.4.4 *MonteCarlo*

Monte Carlo simulation instances are initialized with parameters specifying a database connection, relevant game data, which then populates the necessary attributes required for operations, and the desired number of epochs.

The class provides two separate run methods, allowing for both client and back-end utilization. Client-based use is constructed specifically for the purpose of presenting this project for academic review, and is therefore limited by a connection time which consequently limits the amount of epochs. Back-end use is meant for higher epoch counts, placing less demand on the system. This is utilized to test the application's results with higher epoch counts, and determine how realistic the outcomes are.

Both methods create a *Simulation* instance utilized to play one basketball game per epoch iteration. Results are tracked and returned as matplotlib plots, one probability density, and one violin. The back-end method *run_plt* displays the plots upon completion of the simulation, while the *run* method, intended for client use, returns a list of the plots as strings, along with the basic statistics of the simulation such as the team's win percentage per the total number of epochs and the mode of their results. Code Extract 5.8 illustrates how the method utilized to convert matplotlib graphs to base64.

[4]In a basketball game, the pace metric indicates the average or expected amount of possessions the team will have during the expected 48 minutes of play [46].

Code Extract 5.8: Plot Conversion to Base64

```
    def save_2_bytesIO_buffer_string(self, fig) -> str:
        buffer = BytesIO()
        fig.savefig(buffer, format='png')
        plt.close(fig)
        buffer.seek(0)
        return base64.b64encode(buffer.getvalue()).decode('
            utf-8')
```

5.5 *view*

The package is a WinForms .NET framework application, and it acts as the client side of the system. Users can utilize the Windows-based program to communicate with the back-end's Python host via REST API. C#'s RestSharp package provides the capability to handle HTTP requests and deserialize the returning responses into the appropriate objects for utilization.

The view is comprised of three windows: *FindGame*, *ScrapePopUp*, and *MonteCarlo*. Each of these handles the communication necessary for their operation. The client operates only as a display, and an interface for the user. All operations are performed by back-end logic, and the results sent to the client as a JSON. Parameters are cached on the client side, decreasing the necessity of taking the same input parameters from the user. This information is also passed on to subsequent windows allowing for a smoother user experience.

The client only allows for simple, minimal client control of the application. Database operations are purposely limited to the back-end logic. The *view* is truly a minimalist use interface intended to display the application's capabilities for the sake of presenting this project.

The *FindGame* window takes initial parameters for setting up the game, such as home and away teams, season, and epoch count. When the *Get Games* button is pressed, it requests all historic games via the */monte_carlo/game_data* resource endpoint, displaying them in a table via DataGridView [47].

The selected row automatically parameterizes the */monte_carlo/team_in_db* endpoint, which notifies the client of the required data's presence, or lack thereof, in the database. If required, the *ScrapePopUp* window appears, allowing the user to specify their choice of scrape settings to acquire the data with the */monte_carlo/season_data* endpoint.

If the user wishes, web scraping can be forced, even if the data already exists by pressing the *Forced Scrape* button. In this case all data is scraped again, however as

all tables in the database require unique records, the data will only be added to the database if it is truly missing from a table.

The application does not include a VPN, therefore users are encouraged to use a third party service. Proxies are included with the application, and if turned on can provide an additional layer of security, however these are free proxy addresses that have not been thoroughly reviewed. They therefore cannot guarantee anonymity. Furthermore, the use of the built in proxies greatly decreases application speed, as the system will likely need a long time to locate one that is operational, due to the high demand for these services. Users do have the option to provide their own proxy list. If they choose to do so, they must provide it as a CSV table, with the first row being header, first column containing IP addresses, and second row containing port numbers.

Once it is established that the database contains the necessary information (either by means of a successful scrape, or by confirmation from the host), the final window is activated. This *MonteCarlo* window receives all previous parameters from the previous windows upon initialization, and commences the simulation without the need for user interaction via the */monte_carlo/simulation* endpoint, displaying both graphs along with original and simulated game data for the user.

Message boxes notify the user of any potential errors throughout the application process, giving brief descriptions. All errors are logged on the server-side of the application allowing for further debugging.

5.6 *webscraper*

The *webscraper* package provides a variety of web scraping services, which are utilized and operated by the *controller*. It provides both data acquisition and data parsing services through its *ScraperFacade* and *RealGMParser* classes. The facade class connects either directly or transitively to all other classes in the package. The *RealGMParser* and *ProxyHandler* class also make use of the *Logger*. Figure 5.6 illustrates the package's class diagram.

This section reviews the *webscraper*'s implementation broken down by functionality in order to give an overview of how they work together with the facade class that represents the package's functionality.

5.6.1 Utility Classes

The *WebKit* class is a collection of static methods that are generally used by scrape services throughout the package. They attempt to mimic human use and introduce randomness to decrease the chance of being blocked by the service provider. Services include:

- Creating new random headers for HTTP requests, mimicking varying operating systems and browser technologies.
- Introducing random with preset intervals for delays between requests.
- In case of Selenium utilization, providing mouse movements to specific elements within the HTML.

The *ProxyHandler* class creates and returns a list of proxies, with the added capability to check each proxy before appending them to the list. While this functionality can be useful, it doesn't necessarily guarantee access to the proxy upon later use, as free proxies have a high demand online. The class also provides a default proxy list, however parameterization allows for the use of one's own list. Proxy lists must be CSVs, with the first row being the header, first column the IP addresses, and second the port numbers.

The *ProxyKit* offers proxy services to chosen web scraping methodologies in the facade through it's proxy list attribute. The class contains methods to check proxy and IP address formats, and package them for utilization by the scraping methods (e.g.: Requests library's request method requires a dictionary format specifying the string separately for HTTP and HTTPS). The class also houses the *apply_rotating_proxy* method, responsible for applying single proxies to the web scraping process. Each scrape attempt is assigned a new proxy. Addresses are popped from the list and attempt a connection. This process continues until a successful connection is made, after which the working proxy address is appended to the back of the list for future use.

5.6.2 Selenium WebDrivers

The Selenium service uses an abstract WebDriver acting as a base class, defining common virtual and regular methods for all implementing child classes. Each child class operates a separate browser technology, offering the same functionality. Methods pertaining to WebDriver setup are virtual, allowing each WebDriver instance to implement its browser specific setup options, providing, among other things, proxy functionality for later use. Regular methods provide web scraping capabilities and quit the driver.

The *WebDriverFactory* is a factory class with static methods responsible for creating instances of the sub-classes, such as FirefoxDrivers and ChromeDrivers. This method is based on the Factory Method described in the GOF book Head First Design Patterns [10, p. 134], and Abstract Factory Pattern [10, p. 156] which allows utilizing systems to decide which sub-class they want to create. The static factory methods allow further decoupling, by allocating sub-class instantiation to a separate static factory instead of utilizing the WebDriver for this task.

The code contains the setup for Undetected ChromeDriver, however it is not implemented (it's commented out), as the library has caused several bugs during testing, including issues with the numpy library and failing to delete instances after use. The code segments remain in the project because they are an interesting addition that merit further study and testing.

5.6.3 The Facade

The *ScraperFacade* utilizes the Facade Design Pattern from the GOF's Head First Design Patterns, wherein a complex component architecture provides a single class which encompasses the package's functionality, and is consequently able to act as an access point for all utilizing services within the application [10, p. 264].

The facade therefore provides full data acquisition services to the *controller*. It accesses the factory method to create Selenium driver instances, or creates HTTP content requests with Python's Requests library. Services provided by the *WebKit* are automatically included in both. It also provides the necessary access methods to utilize proxies during web scraping.

5.6.4 Parsing Service

The *RealGMParser* is treated as a standalone component within the package. The class consists of static utility methods, built for parsing and handling content acquired form the RealGM website's URLs. The class makes use of the BeautifulSoup4 and Pandas Python libraries to parse the scraped data into DataFrames.

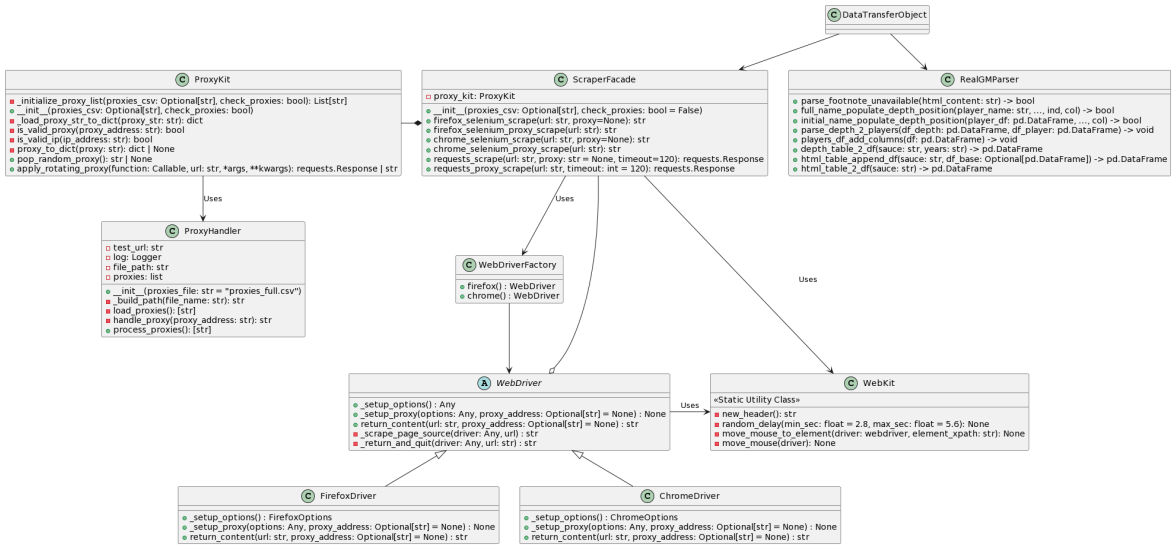


Figure 5.6: *webscraper* UML Class Diagram

Chapter 6

Testing

The application contains over one-hundred tests, responsible for reviewing operational integrity at all levels. The majority of the tests are unit tests, however both integration and system tests are included.

Tests are run separately for the front and back-end of the application. Front end testing is discussed in the System Testing section. Back-end testing consists of a combination of unit and integration tests, both utilizing Python's built-in *unittest* framework, which provide a suite of testing methods specifically designed for this purpose [48].

It was not my original intention to have such a comprehensive test suite, however due to the complexity of the project, as the application grew, it became necessary to run frequent tests. The *regression_test* module was therefore created, which includes all unit and integration tests, along with a full linting of the back-end. Tests can be run form their respective modules as well, either by activating the full module test, or just running the individual test pertaining to the single component, by pressing the green triangle beside the name of the class or method.

Warning: Implementing the full test suite involves live web scraping methods, therefore the use of a VPN service is strongly suggested. Furthermore, the proxy addresses in the default proxy list have not been individually tested to ensure they provide complete anonymity.

6.1 Unit Testing

The *unit* package is responsible for unit testing the back-end and is housed within the *test* package. Each test module is responsible for unit testing its namesake within the back-end. Test cases focus on specific method and class functionality, are aptly named, and thoroughly commented.

Test modules and their responsibilities include:

- **db_handler_test**: tests cover various functionalities of the *MySQLHandler* class.
- **dto_service_test**: test cases for ensuring *Persist* class functionality within the *controller*.
- **dto_sim_test**: test cases validate the functionality of the *TeamBuilder* class.
- **game_service_test**: test methods for ensuring the *GameBuilder* class operates as expected.
- **logger_test**: covers various aspects of the *Logger* class.
- **parse_service_test**: test cases validate the functionality of the methods in the *RealGMParser* class.
- **proxy_handler_test**: tests cover the functionality of the *ProxyHandler* class, ensuring that methods related to proxy processing function correctly.
- **selenium_service_test**: tests all classes pertaining to Selenium service: *FirefoxDriver*, *ChromeDriver*, and *WebDriverFactory*.
- **teams_test**: tests the *Teams* enumeration class and its functionality.
- **webscraper_test**: test cases for the *ScraperFacade* class using `www.icanhazip.com`, a website which returns the requesting IP address.
- **webscraper_utilities_test**: contains methods for testing the functionality of the *WebKit* and *ProxyKit* classes.

6.2 Integration Testing

The package contains two modules: *control_service_test* and *monte_carlo_test*. Each conducts a full test of the respective functionality provided by the application.

The *control_service_test* runs a full scrape of the Chicago Bulls' 1991-1992 season. Results are persisted to the test database, before being checked from there, allowing for a comprehensive test of the system's operation. Singular scrape and persist methodology is also tested. In this case, each table type (player, individual game, and team) are scraped, persisted and checked individually. Tests are automatically truncated, leaving the test database empty for future use.

The *monte_carlo_test* runs a simulation for both regular season and playoff games, ensuring proper functionality of Monte Carlo simulation services.

6.3 System Testing

The full systems test is completed through the Client interface, utilizing .NET's TestStack.White package. As discussed in the Technologies and Frameworks section, this package allows interface testing, and supports WinForms, among others.

The test suite first checks window parameterization functionality by applying a combination of erroneous choices and ensuring a message box return and error. The incorrect parameters tested include:

- Epoch count is not filled.
- Epoch count is filled with non-numeric value.
- Season drop-down menu is left unselected.
- Home and Away teams are the same value.
- Home team drop-down menu is left unselected.
- Away team drop-down menu is left unselected.

Finally, selected seasons are tested by iterating over all teams and performing a Monte Carlo simulation on the parameters. This is done by creating two array: *nbaTeams* and *seasons*, and iterating through the combination of teams where they are not equal for each season's iteration. Errors are logged during the testing, and the culmination of the process automatically opens the log for user viewing. The log is not truncated automatically. This allows users to view previous results, but also holds them accountable for file management.

The test does not include every game of the season, only the first game of each team combination. Even so, runtime is in excess of an hour due to the amount of combinations the suite needs to iterate over.

Due to the expanse of data utilized during operations, this approach became highly effective for bug hunting. One issue brought to light by this form of testing was something I didn't think was necessary to account for. The system ran into errors with some of the games at the beginning of the season. Random player selection was unable to pull from certain positions in the team's roster. This was because all players of the given position had no prior metrics with the team, and they were therefore excluded from the roster. This in turn meant the sampling function was attempting to take from an empty list of players, which threw an error. I did not assume such a scenario would come up, as having an entire position full of freshly drafted players, and traded players would be detrimental to performance in any sport. However, this method of testing identified six such instances in the 1991-1992 season alone.

Chapter 7

Manual

7.1 Installation

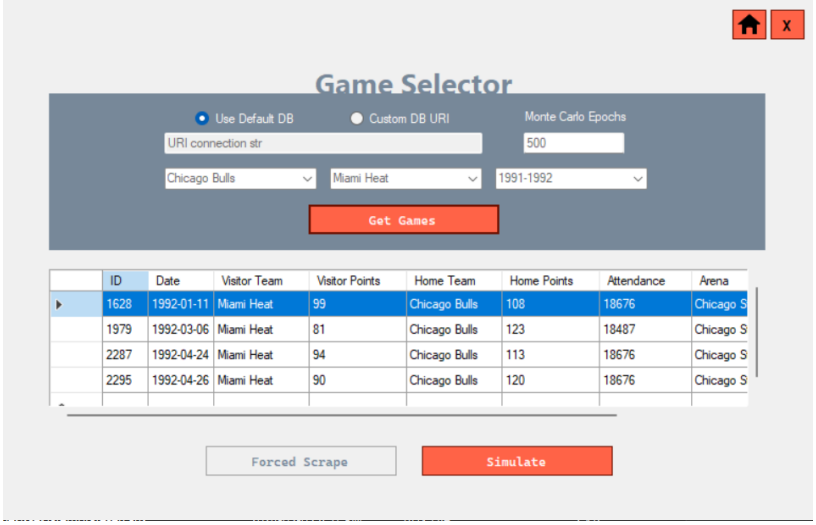
1. Installation prerequisites:
   - An internet connection.
   - Client side:
     - Operating system must be Microsoft Windows 10 or later.
     - .NET framework 4.8 or later.
   - Server side:
     - Python 3.11 or later must be installed.
     - A VPN service is suggested.
     - XAMPP 3.3.0 or later (or any other database service that works with MySQL and MariaDB).
2. Retrieve the source code from `https://github.com/lesheidrich/WebScraping_and_MCSim`
3. From /model/data/ upload *nba.sql* and *nba_test.sql* to your database. Depending on your database provider you may need to break the *individual_games_regular* table into smaller chunks, due to row limitations.
4. Install requirements with the command by navigating to the project folder or using the absolute path with the command: *pip install -r requirements.txt*
5. Fill out *project_secrets.py* with the necessary access data.
6. Ensure your database is running, then start the back end service by running *main.py*.

7. The client application found in /view should be downloaded to, and run from the client system. /view/NBA_Sim_View/bin/Debug/NBA_Sim_View.exe will commence the client application.

7.2 User Guide

Users are greeted with the FindGames Window (see Figure 7.1), where they can initialize the simulation parameters by:

- Choosing either the default database, or providing their own
- Setting the epoch count.
- Selecting Home and Away teams, and season.



FindGames Window

After pressing the *Get Games* button, a table appears, where all the parameterized games for the season are listed. After selecting the row of the desired game in the table, users can proceed by pressing the *Simulate* button.

Figure 7.1: The FindGames Window

ScrapePopUp Window

If the game metrics are not yet in the database, or if the user selected the *Forced Scrape* button, the ScrapePopUp Window appears (see Figure 7.2), which allows the selection of a web scraping method. The top of the window shows the team that is missing from the database. Users can utilize their own proxy list by pressing the *Custom Proxy List* button, and opt to pre-check them by clicking on *Check Proxies* as well. Clicking *Scrape* initializes the process, which automatically opens the next window once complete.

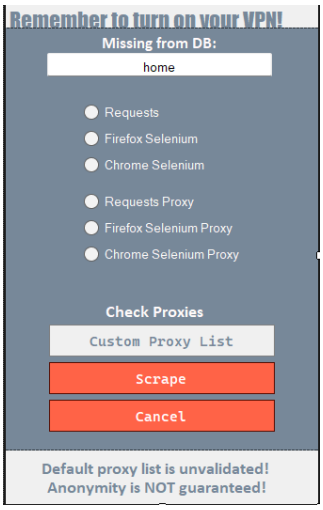


Figure 7.2: The ScrapePopUp Window

MonteCarlo Window

Once all the necessary data is available, the system proceeds with the simulation. The user will need to wait at the previous window until this process is complete. Wait times are determined by the epoch count given in the first window. Historical game results can be seen in the top right corner (see Figure 7.3).

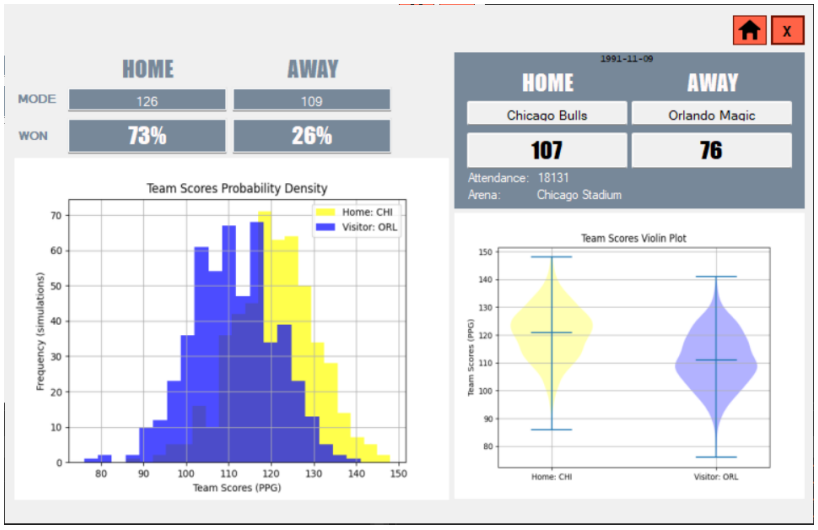


Figure 7.3: The Monte Carlo Simulation Results Window

The top left corner shows the percentage of games won by each team and the mode of their results. Both the probability density (left) and violin plots (right) are based on the final outcomes of the game per each epoch, displaying the frequency of each outcomes occurrence. Home scores are shown in yellow, away are blue.

Users can navigate back to the initial FindGames window at any time by pressing the home button on the top right-hand side, and initiate a new simulation by changing the parameters.

Chapter 8

Results

8.1 Web Scraping

The web scraping functionality of the project was concluded with a high success rate. This was in part due to the relaxed security on the server side of the RealGM website. Even though the site's robots.txt disallows scraping of all tabular data, even simple HTTP content requests have been successful at attaining a response.

The measured scraping process spanning all teams for fifteen seasons contained the following features:

- Alternating random request headers with the *fake_useragent* library.
- Request timeout of two minutes.
- A free VPN service was run to mask the system's IP address, however proxies were not utilized to decrease runtime.
- Python's Requests library was used to make HTTP requests to the service provider.
- A random wait time ranging from two to seven seconds was introduced after each requests.
- In case of failure, a backup scrape was implemented with Selenium's Firefox WebDriver.

The web scraping process ran for over twelve hours. During the process, HTTP requests made with the Requests library failed four times. All scrape attempts executed with Selenium's Firefox were successful.

In comparison with other scraping technologies, the applied method of a quick HTTP request followed by browser imitation in case of potential issues show promise. It combines the speed and effectiveness of the Requests library, while still providing the higher success probability of a Selenium based scrape as a backup. This also ensures

timely execution, as using Selenium on its own would drastically increase processing time. Requests based web scraping attempts only approached the two minute cutoff in case of connection errors, while Selenium based scrapes generally performed around the three minute mark.

This approach is not novel, the combination has been discussed over multiple research papers, forums and blogs. Ajay Bale et al give a thorough evaluation of the web scraping methods in their 2022 paper titled *Web Scraping Approaches and their Performance on Modern Websites* [49].

Parsing the results with a combination of BeautifulSoup4 and Pandas into DataFrames gave a high level of control to the application's methods, allowed for fast development, and easier debugging.

8.2 Monte Carlo Simulation

I dove into the Monte Carlo simulation with modest expectations. Initially, the methodology was included in the application as a means of familiarizing myself with its intricacies, thereby laying the groundwork for potential future projects utilizing it.

Aware of my limitations in understanding the complexities of basketball metrics, along with how individual statistical markers affect each-other, I expected limitations pertaining to the simulation's outcome. Furthermore, the time required to thoroughly research and assemble a proper database containing all the necessary data to make realistic predictions would take longer than the time allotted for researching, building and writing this thesis.

The basketball game's simulation process is also missing details, such as fouls and injuries. Delving deeper into the simulation would have necessitated an exponential increase in sports-related research and would have required specific metrics, which were unavailable. To give an example, free online statistics measure personal fouls per player, but it is much harder to find the proportion of offensive to defensive fouls, and each introduce new intricacies to the game flow.

Consequently, my expectations towards the simulation remained tempered. I assumed a high probability of randomness for outcomes, which would only resemble history to a minor degree. To my surprise, in many cases the predicted outcome matched the range of the historic game's outcomes. Simulation scores always resembled a higher percentage of historic game scores, however this was to be expected as shot attempts were calculated with the inclusion of free throw percentages. As free throws have a much higher rate of success than field goals, this is completely understandable. Results would usually become less correct when historic game scores were close, within a 5 point range. Table 8.1 highlights a few of the simulated games and their results, broken down into an epoch range of one, five, and ten thousand.

Date	Home	Away	Actual	Epochs	Win %	Mode
1992-01-11	Chicago	Miami	108 - 99	1 000 5 000 10 000	71% 69% 71%	118 - 110 118 - 106 116 - 107
1992-04-10	Boston	Milwaukee	109 - 100	1 000 5 000 10 000	59% 60% 61%	110 - 109 110 - 107 114 - 108
1992-01-12	LA Lakers	Orlando	112 - 99	1 000 5 000 10 000	55% 53% 54%	106 - 100 108 - 104 107 - 103
1992-02-02	New York	Golden State	113 - 120	1 000 5 000 10 000	61% 56% 55%	114 - 120 114 - 118 114 - 115
1992-02-26	Denver	Miami	98 - 105	1 000 5 000 10 000	71% 69% 70%	102 - 116 104 - 115 102 - 110
1992-01-15	LA Lakers	Charlotte	95 - 93	1 000 5 000 10 000	55% 55% 56%	109 - 108 112 - 108 110 - 106
1992-03-01	Chicago	Portland	111 - 91	1 000 5 000 10 000	57% 60% 59%	121 - 118 120 - 118 120 - 116

Table 8.1: Monte Carlo Simulation Results

Table 8.1 gives a good general sense of potential game outcomes, however by plotting each epoch, additional insights can be gained. Both the probability density and violin graphs visible in Figures 8.1, 8.2, and 8.3 (simulating one, five, and ten thousand epochs) illustrate the frequency of each score's occurrence, which shows the range of all likely outcomes instead of focusing only on the most probable score values. This has the added benefit of displaying potential trends that fall short of the mode's value.

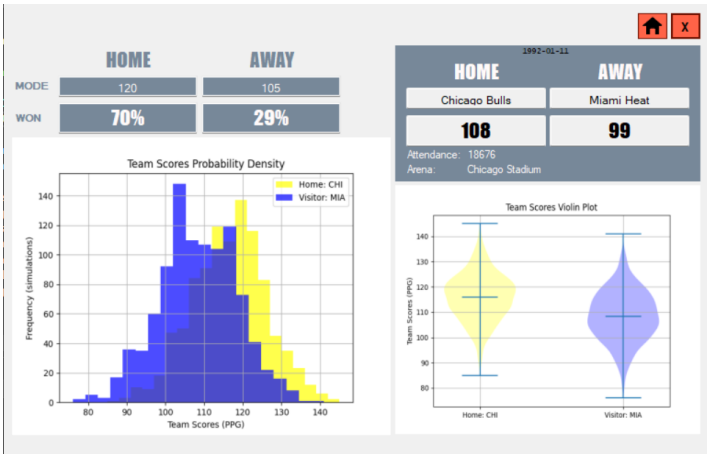


Figure 8.1: Simulation Results per 1000 Epochs

As seen in Figures 8.1, 8.2, and 8.3, an increase in epoch count leads to the convergence of score values around the mode of all scores, and consequently decreases the occurrence of other high frequency score occurrences. Model testing, therefore, does not surpass the ten thousand epoch mark, as further increases would be unlikely to yield better results.

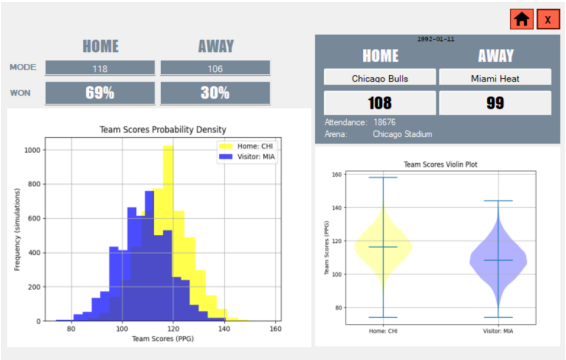


Figure 8.2: Simulation Results per 5000 Epochs

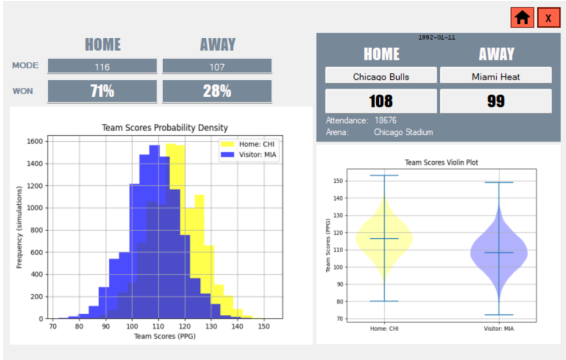


Figure 8.3: Simulation Results per 10000 Epochs

Overall the Monte Carlo simulation surpassed my expectations, performing quite admirably despite the inherent limitations. It is interesting to note that the model's lightweight build not only didn't decrease performance as expected, but also decreased the required amount of epochs to realize an acceptable outcome. As seen in the above images, sometimes even one thousand epochs result in a successful simulation. While this model is rudimentary in comparison to Monte Carlo simulation capabilities, it helped gain insight and lay the foundations for future research and refined my understanding of simulation methodologies.

Chapter 9

Conclusion

9.1 Limitations and Future Work

Throughout the course of the research, numerous insights and challenges have surfaced, shedding light on possible alternate approaches, limitations, and areas of improvement. As with any ambitious project, time management and realistic goals are key factors of success. While I would have liked to delve deeper into data analysis, validation, and certain technical aspects of the simulation, I found it necessary to curb my enthusiasm and maintain focus on the project's core objectives.

Reflecting on the project's structure, I would make several adjustments. Rather than developing a client application which is served by a Flask host, I would opt to run the program directly from the Python console, showcasing results utilizing a reporting tool such as Microsoft PowerBI. While the initial approach held merit form a computer science perspective, refraining from it would allow for the enhancement of data validation practices, along with the opportunity to delve further into basketball analytics, thereby refining the simulation's operations.

I would further update the project's architecture by decoupling the simulation process' dependencies into a single control sequence, and removing data transfer related logic from the *controller*.

Data validation is one of my biggest takeaways as I look towards future projects of this caliber. Web scraping from multiple sources for cross referencing data before its integration would have been a valuable addition to the project. While this represents a significant demand increase on both time and resources, the assurance of valid data guaranteeing no gaps or deficiencies is a worthwhile investment.

I am eager to explore further applications of Monte Carlo simulations, not just in sports analytics, but also regarding finance, particularly in investments. Integrating deep learning techniques into repetitive high data volume parts of this methodology also present exciting research opportunities for enhanced analysis and prediction.

9.2 Conclusion

In this thesis, I undertook to explore a combination of web scraping techniques and Monte Carlo simulations as they apply to analytical forecasting through utilizing the methods to predicting historic basketball game outcomes in the 1990s. By developing a web scraping tool, I acquired vast amounts of statistical data pertaining to the NBA games from free online sources, which enabled the creation of a robust dataset used for analysis and simulation.

The implemented Monte Carlo simulations provided insight into the potential outcomes of basketball games, the success of which was analyzed based on the historic outcomes of the matches. Despite the inherent challenges and limitations, such as the complexity of the metrics, the simulation results showed a promising correlation with historical game data.

The comparison of simulation results based on various epoch counts highlighted the importance of occurring frequency and the convergence of score values around a most likely outcome. By plotting probability density and violin graphs for epoch counts and frequency of scores, I also demonstrated the occurrence of alternative high-frequency score values, and their decrease in comparison to increasing epoch counts.

I am grateful to have had this opportunity to study both web scraping and Monte Carlo simulation methodologies, and look forward to future projects relating to both. This thesis has served as a foundational study for me in the application of web scraping and Monte Carlo simulations in the domain of predictive modeling.

Bibliography

[1] Medium, *The Power of Data: Understanding Its Impact and Applications Across Various Domains*, Jonathan Mondaut, 2023, https://medium.com/@jonathanmondaut/the-power-of-data-understanding-its-impact-and-applications-across-various-domains-6b3c2b2f1ca3, [Retrieved 2 March 2024]

[2] Northeastern University, College of Science, *Why it's so hard to make accurate predictions*, Jason Kornwitz, 2017, https://cos.northeastern.edu/news/hard-make-accurate-predictions/, [Retrieved 27 February 2024]

[3] Moaiad Ahmad Khder, *Web Scraping or Web Crawling: State of Art, Techniques, Approaches and Application*, International Journal of Advance Soft Computing and Applications, Vol. 13, No. 3, 2021, Print ISSN: 2710-1274, Online ISSN: 2074-8523, Al-Zaytoonah University of Jordan

[4] Adekitan Aderibigbe, *A Term Paper on Monte Carlo Analysis/Simulation*, Department of Electrical and Electronic Engineering, Faculty of Technology, University of Ibadan, 2014.

[5] Wikipedia, *National Basketball Association*, 2024, https://en.wikipedia.org/wiki/National_Basketball_Association, [Retrieved 27 February 2024]

[6] Don L. McLeish: *Monte Carlo Simulation and Finance*, Hoboken, New Jersey, USA, John Wiley & Sons, Inc., 2005.

[7] Paul Steffen: *Statistical Modeling of Event Probabilities Subject to Sports Bets: Theory and Applications to Soccer, Tennis, and Basketball*, Statistics [math.ST], Université de Bordeaux, 2022. English. NNT: 2022BORD0210. tel-03891393.

[8] Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, Kelli A. Houston, *Object-Oriented Analysis and Design with Applications*, Massachusetts, USA, Addison-Wesley, 2007.

[9] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, Randy Stafford, *Patterns of Enterprise Application Architecture*, USA, Addison-Wesley Professional, 2002.

[10] Eric Freeman, Elisabeth Freeman, Bert Bates, Kathy Sierra, *Head First Design Patterns*, O'Reilly, 2004.

[11] Spring Framework Guru, *National Basketball Association*, 2024, `https://springframework.guru/gang-of-four-design-patterns/`, [Retrieved 5 March 2024]

[12] Abdul Rahman bin Ahlan, Murni bt Mahmud, Yusri bin Arshad, *Conceptual Architecture Design and Configuration of Thin Client System For Schools in Malaysia: A Pilot Project*, Department of Information System, Kulliyyah of Information and Communication Technology, Kuala Lumpur, Malaysia, 2010.

[13] Chris Reade, *Elements of Functional Programming*, Boston, USA, Addison-Wesley Longman, 1989.

[14] Wikipedia, *Graphical user interface*, 2024, `https://en.wikipedia.org/wiki/Graphical_user_interface`, [Retrieved 5 March 2024]

[15] Microsoft Learn, *Enumeration types (C# reference)*, Bill Wagner, 2023, `https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum` [Retrieved 6 March 2024]

[16] MDN Web Docs, *HTML: HyperText Markup Language*, 2024, `https://developer.mozilla.org/en-US/docs/Web/HTML`, [Retrieved 6 March 2024]

[17] Beautiful Soup 4.12.0 documentation, *Beautiful Soup Documentation*, 2004-2023 Leonard Richardson, `https://www.crummy.com/software/BeautifulSoup/bs4/doc/#`, [Retrieved 6 March 2024]

[18] MDN Web Docs, *XML: Extensible Markup Language*, 2024, `https://developer.mozilla.org/en-US/docs/Web/XML`, [Retrieved 6 March 2024]

[19] MDN Web Docs, *HTTP*, 2024 `https://developer.mozilla.org/en-US/docs/Web/HTTP`, [Retrieved 6 March 2024]

[20] PyPI Python Package Index, *fake-useragent*, 2023, `https://pypi.org/project/fake-useragent/#description`, [Retrieved 6 March 2024]

[21] MDN Web Docs, *HTTP headers*, 2024, `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers`, [Retrieved 6 March 2024]

[22] MDN Web Docs, *JSON*, 2024, https://developer.mozilla.org/en-US/docs/Glossary/JSON, [Retrieved 7 March 2024]

[23] Wikipedia, *HTTP*, 2024, https://en.wikipedia.org/wiki/HTTP, [Retrieved 7 March 2024]

[24] Wikipedia, *HTTPS*, 2024, https://en.wikipedia.org/wiki/HTTPS, [Retrieved 7 March 2024]

[25] Pallets Projects, *Flask*, https://flask.palletsprojects.com/en/3.0.x/, [Retrieved 7 March 2024]

[26] Read the Docs, *Flask*, 2024, https://readthedocs.org/projects/flask/, [Retrieved 7 March 2024]

[27] Matplotlib, *Matplotlib: Visualization with Python*, 2023, https://matplotlib.org/, [Retrieved 7 March 2024]

[28] Wikipedia, *MySQL*, 2024, https://en.wikipedia.org/wiki/MySQL, [Retrieved 7 March 2024]

[29] Wikipedia, *pandas (software)*, 2024, https://en.wikipedia.org/wiki/Pandas_(software), [Retrieved 8 March 2024]

[30] Python, *What is Python? Executive Summary*, https://www.python.org/doc/essays/blurb/, [Retrieved 8 March 2024]

[31] Read the Docs, *Requests: HTTP for Humans*, https://requests.readthedocs.io/en/latest/, [Retrieved 8 March 2024]

[32] RestSharp, *Recommended usage*, Peter Breen, 2023, https://restsharp.dev/intro.html, [Retrieved 8 March 2024]

[33] Red Hat, *What is a REST API?*, 2020, https://www.redhat.com/en/topics/api/what-is-a-rest-api, [Retrieved 8 March 2024]

[34] Harvard Scholar, *Selenium Documentation Release 1.0*, 2012, https://scholar.harvard.edu/files/tcheng2/files/selenium_documentation_0.pdf, [Retrieved 8 March 2024]

[35] Github, *undetected-chromedriver*, 2024, https://github.com/ultrafunkamsterdam/undetected-chromedriver, [Retrieved 8 March 2024]

[36] Microsoft Learn, *Introduction to NET*, 2024, https://learn.microsoft.com/en-us/dotnet/core/introduction, [Retrieved 8 March 2024]

[37] Wikipedia, *XAMPP*, 2024, https://en.wikipedia.org/wiki/XAMPP, [Retrieved 8 March 2024]

[38] Read the Docs, *TestStack.White*, https://teststackwhite.readthedocs.io/en/latest/, [Retrieved 8 March 2024]

[39] Microsoft Learn, *Desktop Guide (Windows Forms .NET)*, 2023, https://learn.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-8.0, [Retrieved 8 March 2024]

[40] MariaDB Foundation, *About MariaDB Server*, https://mariadb.org/about/, [Retrieved 9 March 2024]

[41] matplotlib, *The builtin backends*, https://matplotlib.org/stable/users/explain/figure/backends.html, [Retrieved 10 March 2024]

[42] Wikipedia, *Data transfer object*, 2024, https://en.wikipedia.org/wiki/Data_transfer_object, [Retrieved 14 March 2024]

[43] San Diego State University Fowler College of Business, *What is an NBA Players True Offensive Value?*, 2020, https://business.sdsu.edu/news/2020/12/what-is-an-nba-players-true-offensive-value, [Retrieved 15 March 2024]

[44] MySQL Developer Zone, *MySQL Connector/Python Developer Guide*, 2024, https://dev.mysql.com/doc/connector-python/en/, [Retrieved 16 March 2024]

[45] OWASP, *SQL Injection*, https://owasp.org/www-community/attacks/SQL_Injection, [Retrieved 17 March 2024]

[46] Basketball Reference, *Glossary*, https://www.basketball-reference.com/about/glossary.html, [Retrieved 18 March 2024]

[47] Microsoft Learn, *DataGridView Class*, https://learn.microsoft.com/en-us/dotnet/api/system.windows.forms.datagridview?view=windowsdesktop-8.0, [Retrieved 18 March 2024]

[48] Python, *unittest — Unit testing framework*, https://docs.python.org/3/library/unittest.html, [Retrieved 19 March 2024]

[49] Ajay Sudhir Bale, Naveen Ghorpade, S. Kamalesh, Rohith S., Rohith R., Rohan B. S. , *Web Scraping Approaches and their Performance on Modern Websites*, Third International Conference on Electronics and Sustainable Communication Systems (ICESC 2022), Coimbatore, India, September 2022, DOI: 10.1109/ICESC54411.2022.9885689

[50] Scrapy, *Scrapy 2.11 documentation*, 2024, https://docs.scrapy.org/en/latest/, [Retrieved 20 March 2024]

DECLARATION

I, the undersigned **Lorand Heidrich**, hereby declare under penalty of perjury, that the submitted thesis called, **Web Scraping and Monte Carlo Simulations for Analytical Forecasting** is my own intellectual work. Pieces from other authors including printed and online sources are cited appropriately in my work.

I hereby declare that the printed and the online versions are identical as far as form, structure and content are concerned.

I hereby understand that, after the defence the electronic version of my thesis will be put into the library's archive where it will be freely available.

Place and Date:

Eger, April 5, 2024

……..

Signature