A logo for college computing

Description automatically generated

**CCT COLLEGE**

**COMPUTING IN SCIENCE**

**HIGHER DIPLOMA**

**WEB DEVELOPMENT, PROJECT SKILLS AND PROFESSIONALISM PROJECT**

**Eyüp Ensar Almaz**

**2023188**

**MR. KEN HEALY, MR. KAYOUM KHBULI AND MR. JAMES GARZA**

**DUBLIN 2024**

# 1. Software Development Fundamentals

## 1.1 Use Cases:

Use Case ID: RT001

Description: User Adds Player to Team

Dependencies: New Player Use Case

Precondition(s): User has a new player. User wants to add a new player to a team. User is on the "Add New Player" menu.

Primary Actor: User

Action:

1. User selects the "Add Player" option from the menu.
2. User enters the name of the player.
3. System presents a list of player types.
4. User selects the desired player type for the player.
5. System presents a list of teams.
6. User selects the team for the player.
7. New player is successfully added.
8. System displays a confirmation message.

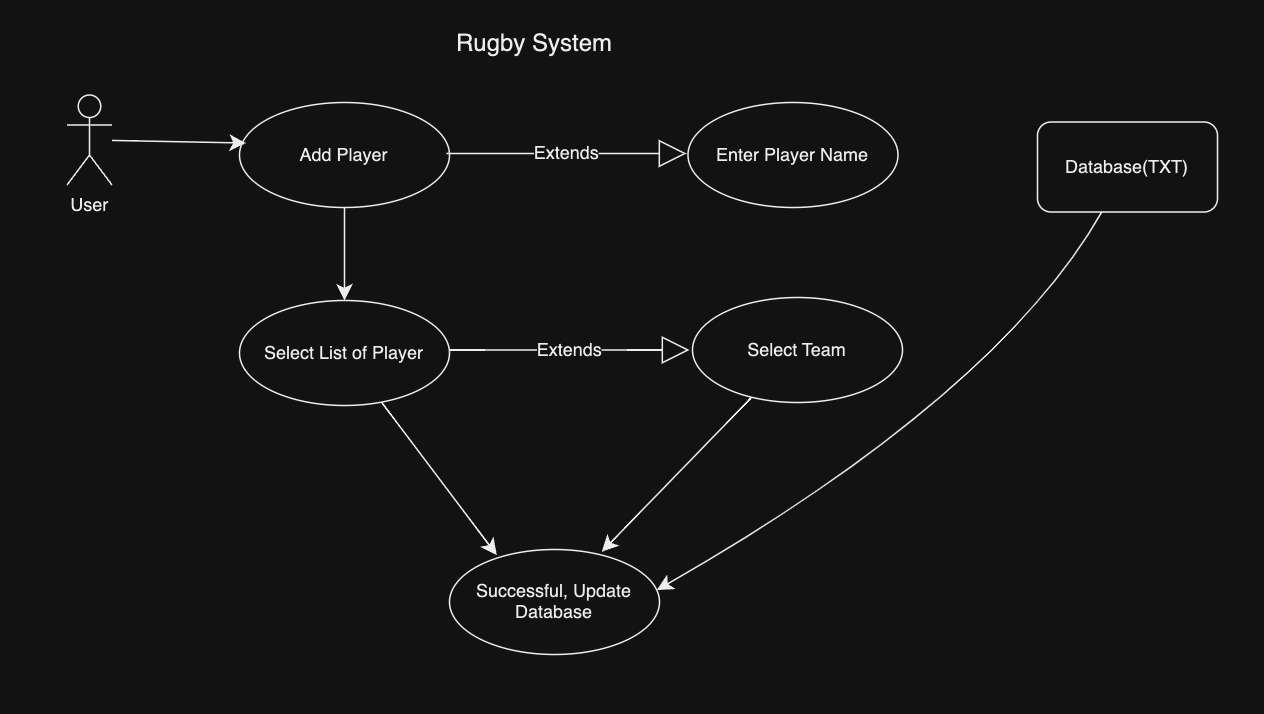
Postconditions: New player is added to the team, and the system displays an approval message.

Alternative Paths:

Error:

a. User enters an integer value other than the player's name.

b. System displays an error message.

c. User cancels the process.

Use Case ID: RT002

Description: User Sorting List of Players

Dependencies: None

Precondition(s): User wants to sort the list of players. The user is on the main menu.

Primary Actor: User

Action:

1. User selects "Sort" from the menu.
2. System displays the current list of players.
3. User chooses a sorting criterion (e.g., by name, by team, by coach).
4. System sorts the list based on the selected criterion.
5. System displays the sorted list to the user.

Postconditions: The list of players is sorted according to the selected criterion, and the sorted list is displayed.

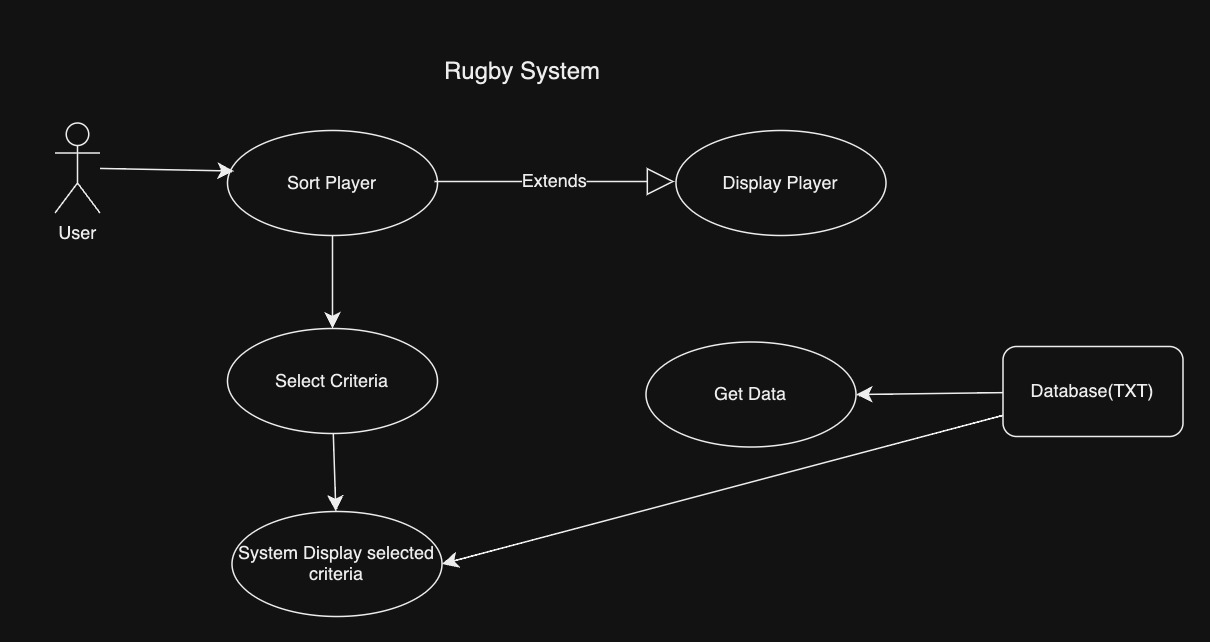
Alternative Paths:

Error:

a. User selects an invalid sorting criterion.

b. System displays an error message indicating invalid input.

c. User cancels the sorting process, and the list remains unchanged.



Use Case ID: RT003

Description: Generating Random Players with Coach Types and Teams

Dependencies: None

Precondition(s): User intends to generate random players with coach types and teams. The user is on the main menu.

Primary Actor: User

Action:

1. User selects "Generate Random Players" from the menu.
2. System prompts the user to specify the number of random players to generate.
3. User inputs the desired number of random players.
4. System generates random players with unique names, coach types, and teams that do not conflict with existing data.
5. System adds the randomly generated players to the list of players.
6. System displays the list of all players, including the newly generated ones, in the terminal.

Postconditions: Random players with unique coach types and teams are successfully generated and added to the list. The list of all players, including the newly generated ones, is displayed in the terminal.

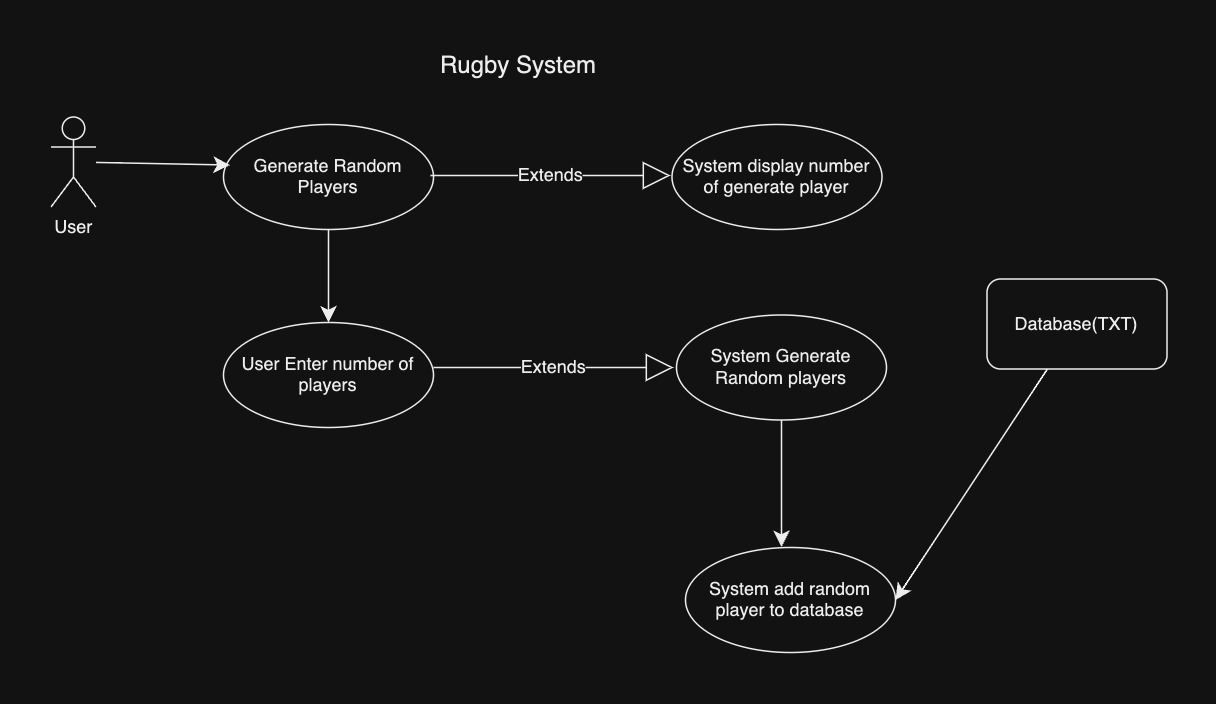
Alternative Paths:

Error:

a. User enters an invalid number of random players.

b. System displays an error message indicating invalid input.

c. User cancels the generation process, and no changes are made to the list of players.



Use Case ID: RT004

Description: Sorting a Dummy List of People

Dependencies: Club\_Form.txt file containing a list of random names

Precondition(s): User intends to sort the list of names gathered from the Club\_Form.txt file. The user is on the main menu.

Primary Actor: User

Action:

1. User selects "Sort List of People" from the menu.
2. System reads the list of names from the Club\_Form.txt file.
3. System initiates the recursive sorting algorithm (e.g., Merge Sort, Quick Sort) to sort the list of names in alphabetical order.
4. Once sorted, the system displays the first 20 records of the sorted list on the screen.

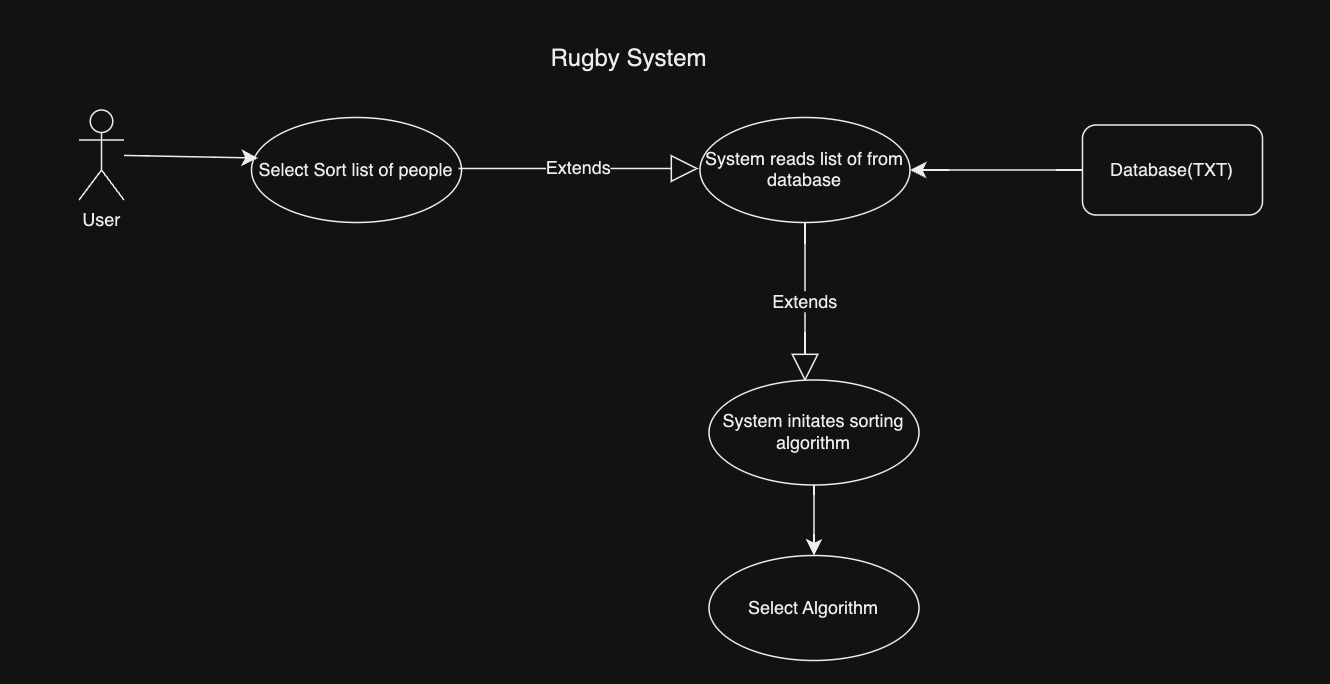
Postconditions: The list of names from the Club\_Form.txt file is successfully sorted in alphabetical order using a recursive sorting algorithm, and the first 20 records of the sorted list are displayed on the screen.

Alternative Paths:

Error:

a. If the Club\_Form.txt file is not found or empty, the system displays an error message indicating that the file is empty or missing.

b. If there is an issue with reading or accessing the file, the system displays an error message indicating the problem with file access.



Use Case ID: RT005

Description: Deleting a Person from a Team

Dependencies: None

Precondition(s): User intends to remove a person from a team. The user is on the main menu.

Primary Actor: User

Action:

1. User selects "Delete Person from Team" from the menu.
2. System displays a list of all players along with their associated teams.
3. User selects the person they want to remove from a team.
4. System prompts the user to confirm the deletion.
5. User confirms the deletion.
6. System removes the selected person from their team.
7. System updates the list of players to reflect the deletion.

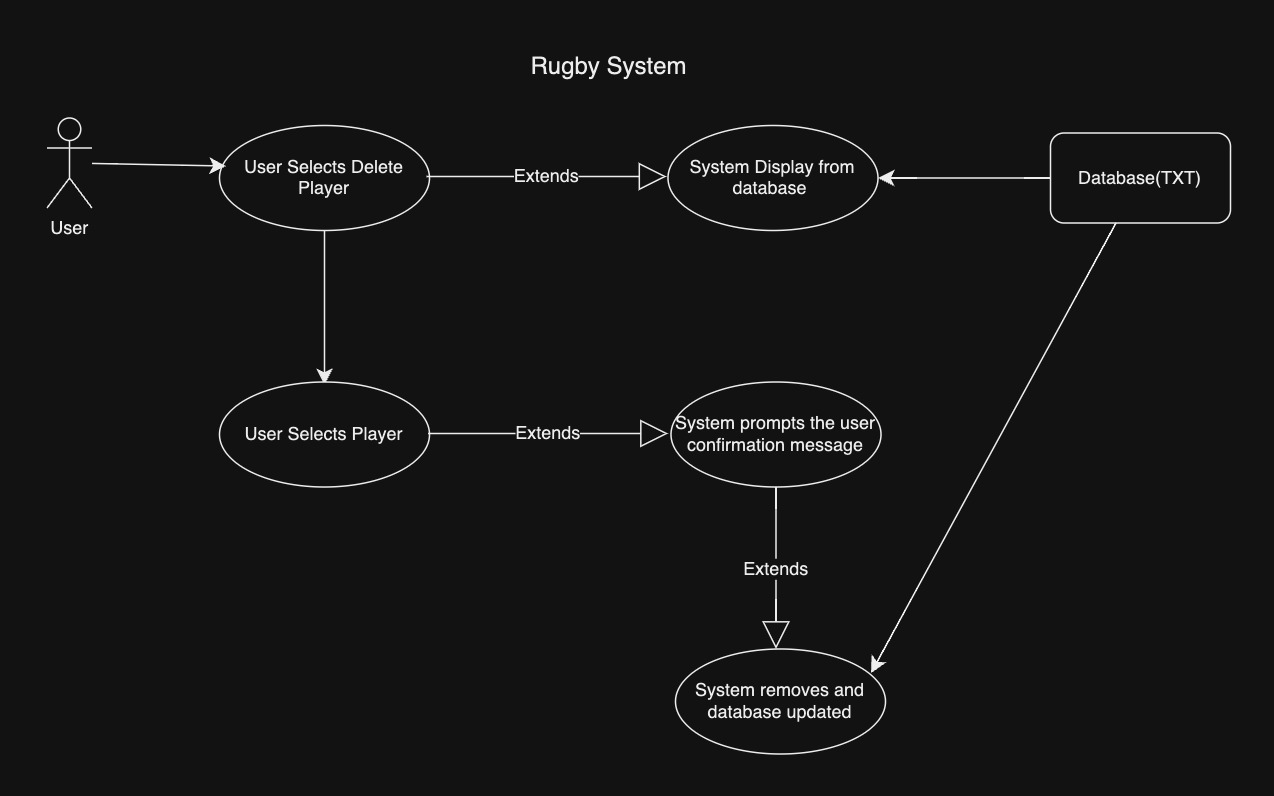
Postconditions: The selected person is successfully removed from their team, and the list of players is updated accordingly.

Alternative Paths:

Error:

a. If the user cancels the deletion process, no changes are made to the list of players.

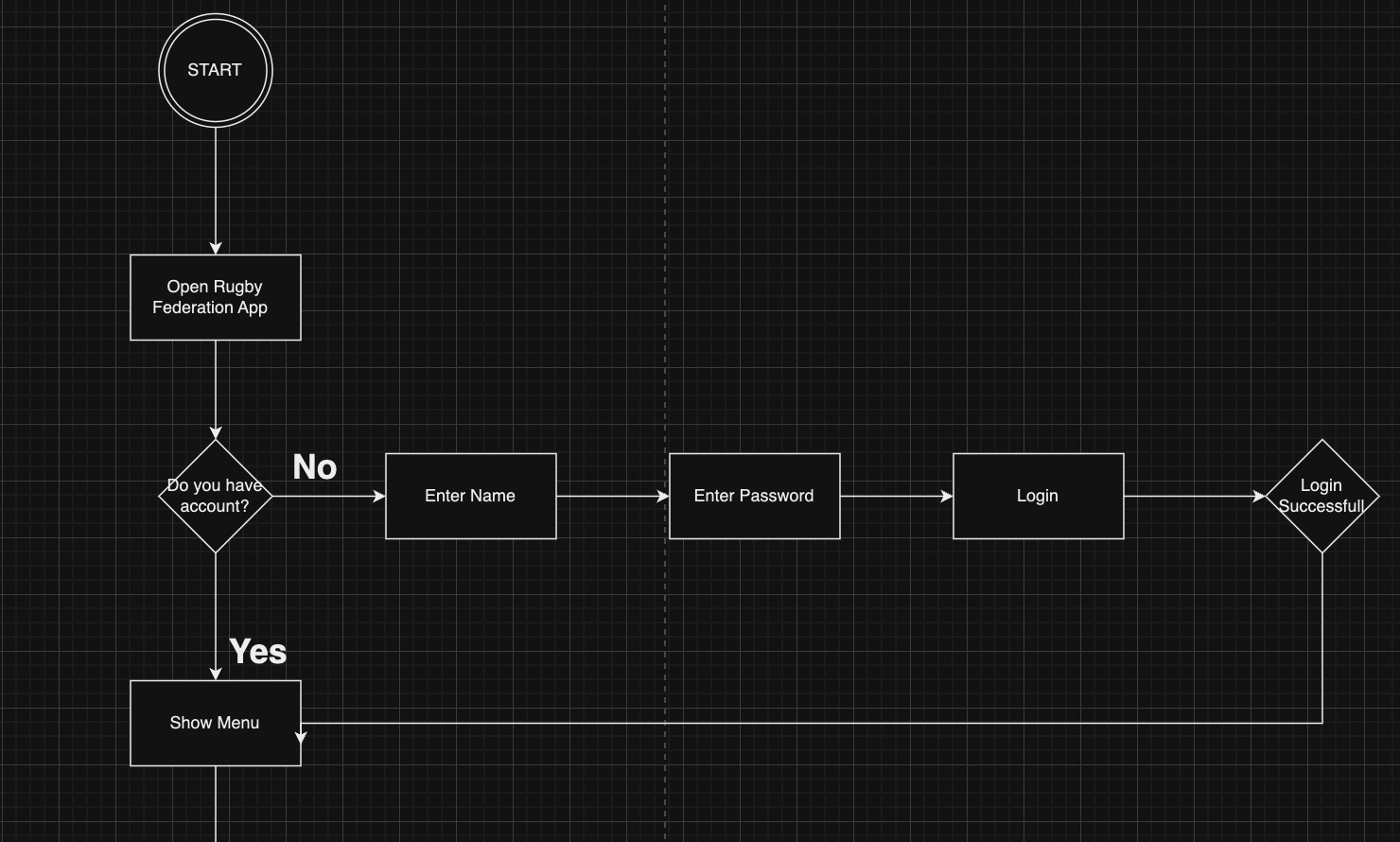
b. If there is an issue with accessing or updating the list of players, the system displays an error message indicating the problem.

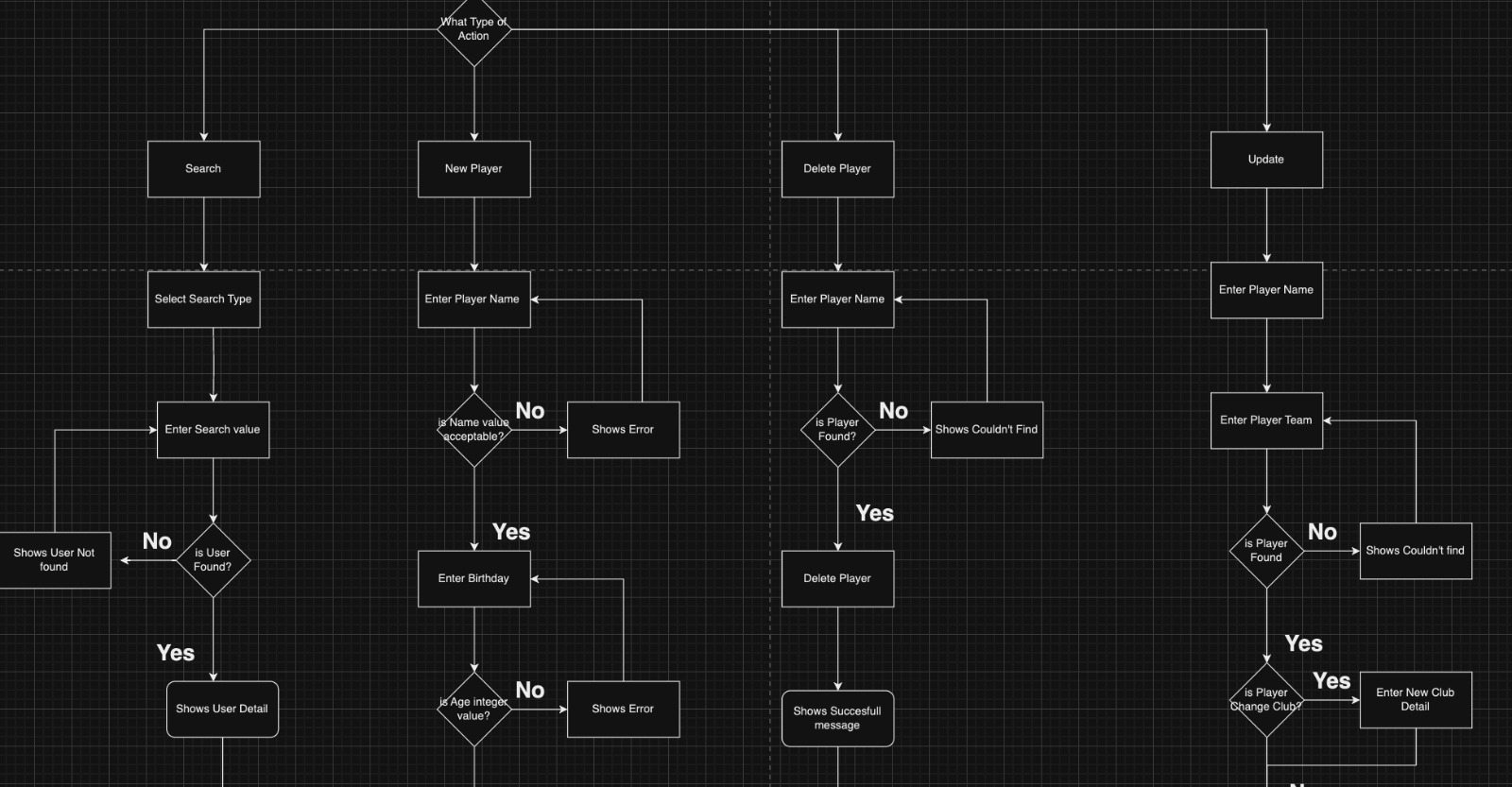


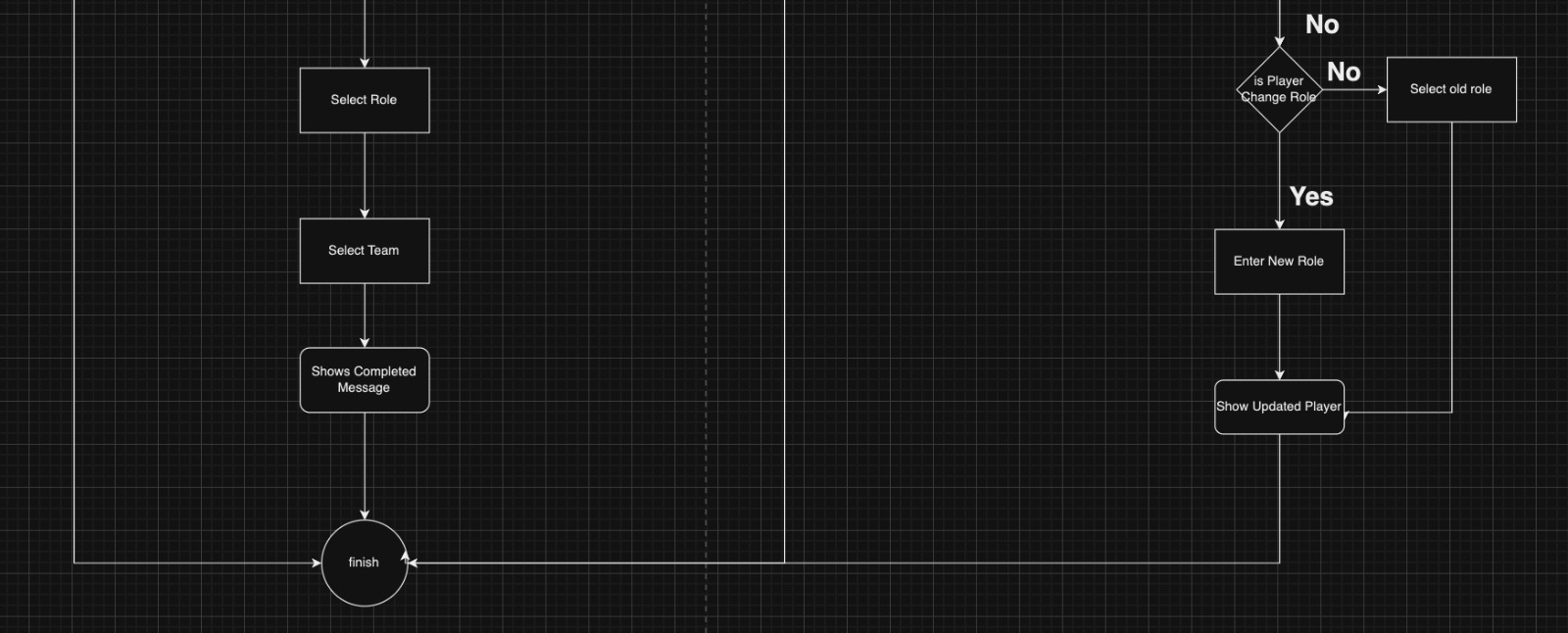
## 1.2 UML Modelling

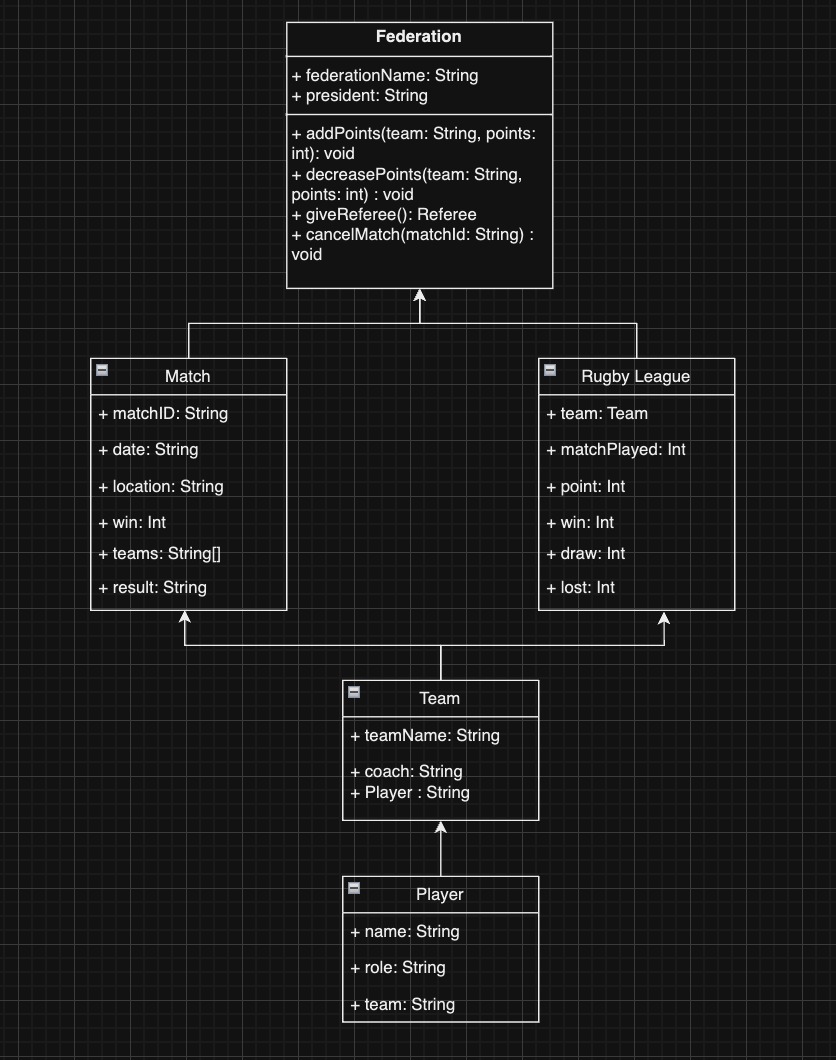
I designed a class diagram and an activity diagram. I've added Authentication, player updating, and player delete as new features. I've added a federation rating group for the rugby league federation.

**Activity Diagram:**

****





**Class Diagram**:

## 1.3 Activity And Class Diagram and Compare with Alternative Diagrams

**Why Activity Diagram and Class Diagram?**

I developed an Activity diagram because I want to show how a system dynamically looks and how system process flow and control flow works. Inter-process flow in rugby application, This diagram was chosen because it is sequential, branching, or simultaneous. The missing parts of activity diagrams compared to Sequence diagrams are messaging. We don’t need the messaging part because our system works like a decision point and action point. System has a login part for the federation if a login successful federation member has to make a decision. Federation members can add players, remove players , update play etc. Also the system is only a console application it shows us error or successful messaging. Another reason is that even if someone does not know the software system, everyone can easily understand our system workflow. My main target is easy to understand. Activity diagrams are generally easier to understand for stakeholders who may not be familiar with technical details or system internals.

**Activity Diagram**: Activity diagrams show the order or flow of operations in a system. They are similar to flowcharts, and show the outcomes of each action, and actions that happen after specific responses. While they can be either concurrent or sequential, all activity diagrams have a beginning and an end state. Widely used in business process modeling and algorithm design.

**Compare with Sequence Diagram**:

I'm comparing Sequence Diagrams because Sequence diagrams are more detailed and technical. Sequence Diagrams delve into the technical intricacies of the system, illustrating the precise sequencing of operations and communication patterns between objects or components. Sequence diagram mostly used for how systems show end use cases, user and system interactions, and system to system interactions. Commonly used for system design, software architecture and scenario analysis, this means if someone doesn’t know anything about a diagram they are never able to understand at first glance. And Sequence Diagram when you want to depict interactions and message flows between different objects or components in a system. **Activity Diagram is Well-suited for business process modeling, workflow modeling, and algorithm design. It was used in our system because we needed an algorithm modeling for the rugby system.**  Sequence diagrams are classified as interaction diagrams, and they visualize how objects collaborate, and focus on the order or time in which they happen. They show end use cases, user and system interactions, and system to system interactions.

**Class Diagram**: I developed a Class diagram because Java is an object oriented programming language. Class diagrams are designed based on OOP (Object Oriented Programming). The goal is to define the classes and the relationships between them in our software. For example Our system has a Player model and a Team model. We want to show some relation between the two model and a class diagram is a good option for us. If we want to extend our project or improve , class diagrams give accessibility to extend.

**Compare with Entity-Relationship Diagram:**

First reason is we don’t have a database for our system. Second reason is that Java is an object programming language. Class Diagrams may be more suitable for representing the system's static structure, including classes, attributes, methods, and relationships. ERD Diagram represents entities, attributes, and relationships in database design and data modeling. For a higher-level view of the system's design and architecture, a Class Diagram may be more appropriate. And for future Class diagrams are more extendable instead of ERD Diagram. **Class Diagram Extensible to incorporate software design patterns, interfaces, and architectural concepts. ERD Diagram Less extensible for representing non-database-related software design patterns.**

## 1.4 Unit Test

**Scenario, User Test 1**: Rugby Administrator log into the system to Add Player to Team

**Description**: As a Team administrator, I want to log into the system so I can add a new player to the team and that new player can join the team and play in a match.

**Acceptance Criteria**:

1. The administrator should be able to enter their credentials through a login interface provided by the system.
2. The system should verify the administrator's identity and enable access to administrative features after a successful login.
3. The system should indicate an error and ask the administrator to try again if the login credentials are not correct.
4. The administrator should have access to the feature of adding a player to a team after logging in.
5. To maintain security, the administrator should be automatically logged out of the system after a certain amount of inactivity.

**Acceptance Task:**

1. Make sure the administrator accesses the system and sees the login interface.
2. Check to see if the system uses the supplied credentials to authenticate the administrator's identity.
3. Check whether the feature that indicates an error when login credentials are entered incorrectly works.
4. Verify that, following a successful login, the administrator has access to the feature that allows them to add players to teams.
5. Verify that the administrator is automatically logged off of the system after a certain amount of inactivity.

**Unit Test**:

1. Verify that the administrator's identity can be verified by the Login class using the supplied credentials.
2. Check to see that the system appropriately reports an error for invalid login information.
3. Verify whether access to administrative features can be granted following a successful login.
4. If the administrator has been logged in for a long time, log out the administrator account.
5. Verify that the system can correctly add new players to the team.

**Scenario, User Test 2**: Search Player in League.

**Description**: As a User, I want to search for a Player.

**Acceptance Criteria**:

1. Users should be able to look for players using the search feature built into the system.
2. It should be possible to search players by name or relevant criteria.
3. The system should show the user the result found.
4. Search results should be related to what the user searched.

**Acceptance Task:**

1. Verify that users are able to type the name of the player or other pertinent parameters into the search field.
2. We must verify that the system returns the correct player searched.
3. The user must ensure that can access the details of the player's result.

**Unit Test**:

1. Verify that the player data can be retrieved from the Search menu using the specified search parameters.
2. Check to see if the system is correctly using the entered search criteria to filter players.

**Scenario, User Test 3**: Update Player Information.

**Description**: As a federation member, I want to update Player information.

**Acceptance Criteria**:

1. The federation member ought to be able to look up a player by name using the system.
2. The federation member should be able to edit Player information.
3. After editing player information our database(.txt) has to be updated and display member player information updated.
4. The system display Successful message after the update is completed.

**Acceptance Task:**

1. Make sure the player search feature is accessible to the federation member.
2. Check that the system shows the right player information for the searched player.
3. Check that the administrator may alter the player's information and save the changes.
4. Display confirmation message after updating.
5. Check txt document after the update is completed.

**Unit Test**:

1. Determine whether the Player data can be updated with new information.
2. Check that the update procedure properly alters the player's information in the database.
3. Testing the validation logic will guarantee that only valid updates are allowed.

**Scenario, User Test 4**: Generate Random Player

**Description**: I want to create random player profiles as a rugby club administrator, assigning them to teams and coach types in order to simulate player registration and load basic data into the system.

**Acceptance Criteria**:

1. It should be possible for the system to produce a predetermined amount of randomly generated player profiles.
2. Random information like name, age, position, and skill level should be included in every player profile that is generated.
3. Every player profile should have a random coach type assigned by the system, selected from a predetermined list of coach kinds.
4. The system ought to show a confirmation message after generation, signifying that the random player profiles were successfully created.

**Acceptance Task:**

1. Check to see if the system offers the ability to create player profiles at random.
2. Verify that every player profile that is generated has arbitrary information such name, age, position, and skill level.
3. Verify that a random coach type is allocated to each player profile from the predetermined list.
4. Verify that teams are formed at random from the available teams in the system by assigning participants to them.
5. Make sure that when the random player profiles are successfully generated, a confirmation message appears.

**Unit Test**:

1. Generate a random player profile and verify that the assigned team is one of the available teams in the system.
2. Generate random player profiles and verify that a confirmation message is displayed indicating the successful creation of profiles.

**Scenario, User Test 5**: Remove Player From Team

**Description**: As a Club administrator, I want to remove a player from the team.

**Acceptance Criteria**:

1. Users should be able to remove a player from a team via the system.
2. The system should provide functionality for users to remove a player from a team.
3. Users should be able to select the player they want to remove from the team.
4. After confirmation, the system should remove the selected player from the team and update the team's roster.
5. Upon successful removal, the system should display a confirmation message to the user.

**Acceptance Task:**

**1.** Check to see if the system has the ability to remove a player from a team.

**2.** Test that users can select the team from which they want to remove the player.

3. Check that users are presented with a list of players in the selected team.

4. Check if it is possible to update the team roster and remove the chosen member from the squad.

5.Confirm that a confirmation message is displayed upon successful removal of the player from the team.

**Unit Test**:

Simulate the removal of a player from a team and verify that no exceptions are thrown.

Verify that a confirmation message is displayed to the user after removing the player from the team.

# 2. Project Skills and Professionalism

**Problem Definition:** The Rugby Club program aims to develop a terminal-based application for managing a list of players, including functionalities for sorting, searching, adding, generating random players, and saving/loading data to/from a file.

**Key Challenges and Requirements:**

1. Sorting the list: The program should efficiently sort the list of players alphabetically.

2. Searching for players: Users should be able to search for players by name and display relevant information.

3. Adding new players: The application must allow users to input new player data, including name, coach type, and team, with appropriate validation.

4. Generating random players: There should be a mechanism to generate random player data for testing and initial population.

5. Saving/loading data: The program should be able to save the player data to a file and load it back when the program starts

**Challenges Faced in Designing the Rugby Club Program:**

1. Efficient Sorting Algorithm Selection: Choosing a sorting algorithm that balances efficiency and simplicity for sorting the player list alphabetically.

2. Effective Search Algorithm Implementation: Implementing a search algorithm that ensures fast and accurate retrieval of player information by name.

3. Input Validation: Designing a robust input validation mechanism to ensure the integrity of user-entered player data.

4. Random Player Generation: Developing a method to generate random player data that accurately reflects real-world player attributes.

5. Data Persistence: Implementing file I/O operations to save and load player data efficiently while maintaining data integrity.

**Solutions and Rationale:**

**1. Efficient Sorting Algorithm Selection:**

• Solution: Merge Sort Algorithm

• Rationale: Merge Sort offers stable sorting with a time complexity of O(n log n), ensuring efficient sorting even for large datasets. Its stability preserves the relative order of equal elements, crucial for maintaining data integrity. The recursive nature of Merge Sort simplifies implementation and maintenance, enhancing code readability and scalability.

**2. Effective Search Algorithm Implementation:**

•Solution: Binary Search Algorithm

Rationale: Binary Search is renowned for its efficiency in searching sorted arrays, with a time complexity of O(log n). By leveraging the sorted nature of the player list, Binary Search ensures rapid convergence to the target player, reducing search time significantly. Its versatility allows adaptation to both sorted and unsorted arrays, making it suitable for various search scenarios.

**3. Input Validation:**

• Solution: Regular Expressions (Regex) for Input Validation

• Rationale: Regular expressions provide a flexible and concise way to validate user inputs, ensuring that only valid player data is accepted. Regex patterns can be customized to match specific criteria, such as alphabetic names with whitespace. While regex may not cover all edge cases, it offers a balance between flexibility and efficiency for input validation.

**4. Random Player Generation:**

• Solution: Random Data Generation with Realistic Attributes

• Rationale: Implementing a method to generate random player data involves creating realistic player attributes, such as name, age, position, and team. By utilizing random number generators and predefined player attribute lists, the generated data closely resembles real-world player information. This approach ensures that the generated data is suitable for testing and initial population of the player list.

**5. Data Persistence:**

• Solution: File I/O Operations for Saving/Loading Player Data

• Rationale: Implementing file I/O operations allows the program to save player data to a file for future retrieval and persistence. By utilizing file handling libraries and appropriate data serialization techniques, player data can be stored and loaded efficiently while maintaining data integrity. This ensures seamless continuity between program sessions and enhances user experience.

**Strengths and Weaknesses of Implementation Choices:**

**1. Merge Sort Algorithm:**

• Strengths: • Stable sorting with consistent performance.

• Efficient for sorting large datasets.

• Recursive nature simplifies implementation and maintenance.

• Weaknesses: • Requires additional memory space for temporary arrays.

• Recursive implementation may lead to stack overflow errors for extremely large datasets.

**2. Binary Search Algorithm:**

• Strengths:

• Efficient for searching sorted arrays.

• Rapid convergence to the target element.

• Versatile for various search scenarios.

• Weaknesses:

• Requires sorted arrays for optimal performance.

**3. Regular Expressions for Input Validation:**

• Strengths:

• Flexible and concise validation mechanism.

• Allows customization of validation patterns.

• Weaknesses:

• May not cover all edge cases.

• Complexity may increase for complex validation patterns.

**4. Random Data Generation:**

• Strengths:

• Provides realistic player attributes for testing.

• Offers variability in generated data.

• Weaknesses:

• Difficulty in ensuring complete randomness.

• Requires predefined attribute lists for realistic data generation.

**5. File I/O Operations:**

• Strengths:

• Enables data persistence between program sessions.

• Efficient storage and retrieval of player data.

• Weaknesses:

• Potential for file corruption or data loss.

• Dependency on external file handling libraries.

**Alternative Approaches:**

**1. Sorting Algorithm:**

• Alternative: Quick Sort

• Rationale: Quick Sort offers comparable efficiency with an average time complexity of O(n log n). However, its instability and potential for worst-case time complexity degradation make it less suitable for sorting player data with stability requirements.

**2. Search Algorithm:**

• Alternative: Linear Search

• Rationale: Linear Search is simpler to implement and suitable for small datasets. However, its linear time complexity (O(n)) makes it inefficient for large datasets compared to Binary Search.

**3. Input Validation:**

• Alternative: Manual Input Validation

• Rationale: Manual validation allows for custom logic implementation but may increase code complexity and maintenance efforts compared to regex-based validation.

**4. Random Data Generation:**

• Alternative: Custom Data Generation Functions

• Rationale: Custom functions provide more control over data generation but may require extensive development efforts and lack randomness compared to random number generators. **5. File I/O Operations:**

• Alternative: Database Integration

• Rationale: Integrating a database management system offers scalability and data integrity benefits but may introduce complexity and dependencies.

**Implications of Choosing Different Approaches:**

**1. Efficiency vs. Complexity:**

• Choosing efficient algorithms like Merge Sort and Binary Search enhances program performance but introduces complexity in implementation and maintenance.

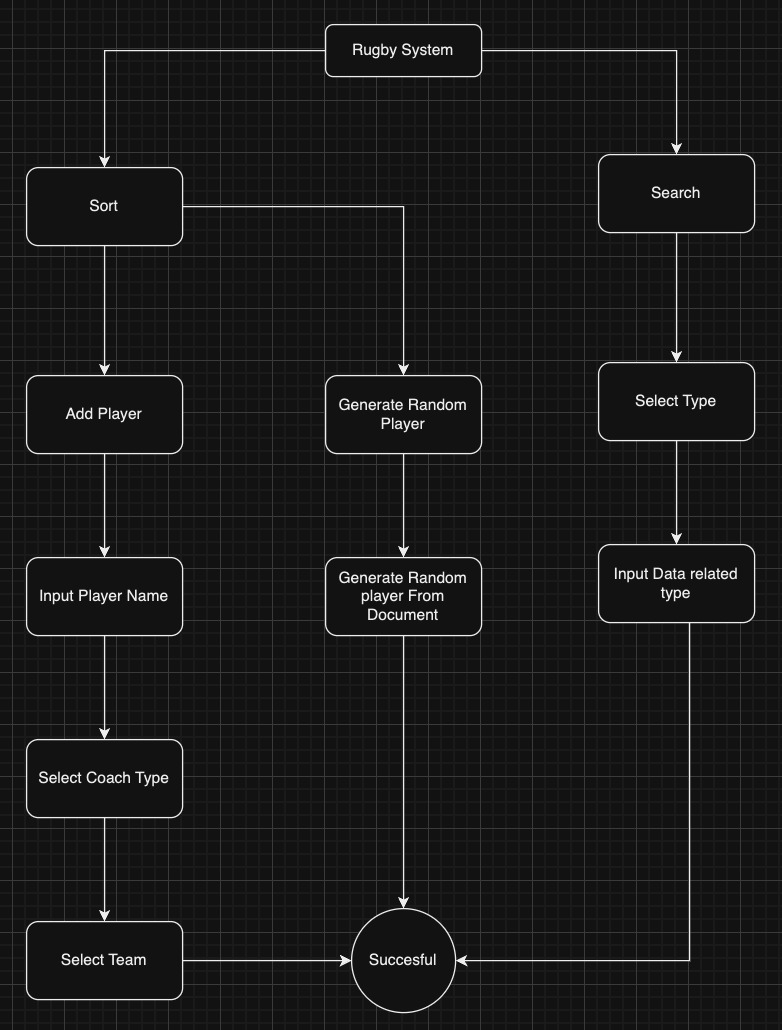
**2. Flexibility vs. Rigidity:**

• Regex-based input validation offers flexibility but may not cover all edge cases, while manual validation provides custom logic but may increase rigidity.

**3. Scalability vs. Resource Usage:**

• Merge Sort and Binary Search offer scalability but require additional memory space, whereas alternatives like Quick Sort and Linear Search may trade-off efficiency for resource usage.

## 2.1 Mapping Diagram

I've sorted the steps that the user can do with this diagram. I've determined which steps the user will follow. I've determined what steps to go back to if the user makes an error in the data entry, or what errors the system should make. I made a draft on the system requirements and how the project would proceed. First of all, I did research on the algorithm of the system and its working logic. Then I set the paths with a diagram. I investigated the errors in the algorithm and the solutions. When the algorithm didn't have any question marks, I moved to the encoding section and started the project.

# Algorithms and Constructs

**Conclusion**: The design and implementation of the Rugby Club program involve navigating various challenges related to sorting, searching, input validation, data generation, and persistence. By leveraging efficient algorithms, such as Merge Sort and Binary Search, and employing robust solutions for input validation, random data generation, and file I/O operations, the program can effectively manage player data while maintaining performance, integrity, and scalability. Understanding the strengths, weaknesses, and implications of different approaches allows for informed decision-making and ensures the successful development and deployment of the Rugby Club application in real-world scenarios.

**Why we have used Binary Search and Merge Sort:**

**Introduction**

Efficient data management lies at the heart of successful sports club operations, enabling streamlined processes, informed decision-making, and comprehensive performance analysis. The Rugby Club application serves as a central hub for managing player data, offering functionalities such as sorting, searching, adding, generating, and saving player information. Key to the effectiveness and performance of these functionalities are the choice of search and sort algorithms. This report explores the rationale behind leveraging Binary Search for people search tasks and Merge Sort for sorting player data, highlighting their superiority, advantages, and implications for the Rugby Club application. Efficiency and Performance: Unveiling the Power of Binary Search Binary Search stands out as a pinnacle of efficiency and effectiveness in searching specific elements within sorted datasets. Unlike its linear search counterparts, which traverse through each element sequentially, Binary Search harnesses the sorted nature of the dataset to swiftly converge on the target element. Through the ingenious strategy of dividing the search interval in half with each comparison, Binary Search drastically diminishes the search space, resulting in logarithmic time complexity (O(log n)). This efficiency is paramount in applications where realtime responsiveness and low latency are imperative requirements. One of the primary reasons behind the adoption of Binary Search is its prerequisite of a sorted array. While sorting incurs an initial time complexity of O(n log n), this investment pays dividends in subsequent search operations. Once the array is sorted, Binary Search operates optimally, capitalizing on the sorted nature of the dataset to execute efficient searches. This requirement aligns seamlessly with scenarios where datasets can be preprocessed or maintained in a sorted fashion, ensuring steadfast and consistent performance. Moreover, Binary Search's versatility extends beyond searching solely in sorted arrays; it can be adapted to unsorted arrays with minimal modifications. While its efficiency is maximized in sorted arrays, Binary Search remains a robust solution for various searching scenarios. Whether the dataset is sorted or unsorted, Binary Search offers a reliable and resilient approach to people search tasks, rendering it applicable across a spectrum of applications and domains. This adaptability is particularly advantageous for the Rugby Club application, which may encounter diverse datasets with varying degrees of organization and sorting.

**The Divide and Conquer Approach:**

Engine of Scalability and Consistency The intrinsic divide and conquer strategy of Binary Search underpins its efficacy and performance. By partitioning the search interval in half with each comparison, Binary Search adeptly narrows down the search space, leading to logarithmic time complexity. This approach ensures uniform performance, irrespective of the dataset's size, by swiftly homing in on the target element. Furthermore, the divide and conquer methodology endows Binary Search with inherent scalability and adaptability to diverse datasets. As the Rugby Club application scales to accommodate burgeoning player databases, Binary Search remains an unwavering and efficient solution for people search tasks, guaranteeing optimal performance and responsiveness.

**Ease of Implementation and Maintenance:**

Streamlining Development Processes Binary Search's simplicity and ease of implementation render it an attractive choice for developers and software engineers. Once the array is sorted, the algorithm involves straightforward steps of dividing the array and comparing the target element with the middle element. This simplicity reduces the likelihood of errors, facilitates debugging, and eases maintenance of codebases, thereby enhancing overall development efficiency. By streamlining development processes, Binary Search empowers developers to devote resources to enhancing other facets of the Rugby Club application, such as user experience, feature enhancements, and performance optimizations.

**Efficiency and Performance**: Unlocking the Potential of Merge Sort for Sorting Player Data Sorting player data alphabetically is a fundamental prerequisite for the Rugby Club application, facilitating easy access, organization, and analysis of player information. To accomplish this task efficiently, Merge Sort emerges as a quintessential and highly effective algorithm, offering stability, predictability, and scalability across datasets of varying sizes. Merge Sort's stability ensures that the relative order of equal elements remains preserved after sorting, thereby safeguarding the integrity of player data. In the context of sorting people, stability guarantees that if two people share the same name, their original order will endure postsorting. Furthermore, Merge Sort's predictable performance across diverse datasets makes it an ideal choice for sorting player data in the Rugby Club application. With a time complexity of O(n log n) in all scenarios, Merge Sort facilitates efficient sorting, even for extensive player databases. This efficiency is paramount for optimizing data management processes, curtailing sorting times, and enhancing overall application performance. The recursive nature of Merge Sort dovetails seamlessly with the structure of the algorithm, rendering it straightforward to implement and comprehend. The divide-and-conquer approach involves recursively partitioning the list into smaller sublists until each sublist contains only one element. Subsequently, these smaller sublists are merged in a sorted order until the entire list is sorted. This recursive nature simplifies the coding process, reduces error incidence, and enhances code readability and maintainability. Consequently, developers can focus on refining other aspects of the Rugby Club application while ensuring efficient sorting of player data.

**Conclusion**: Driving Efficiency and Performance in the Rugby Club Application In summation, the adoption of Binary Search for people search tasks and Merge Sort for sorting player data fortifies the Rugby Club application's efficiency, performance, and scalability. Binary Search's efficiency, versatility, and ease of implementation make it an ideal solution for searching specific elements within datasets, ensuring rapid and accurate retrieval of player information. Similarly, Merge Sort's stability, predictability, and recursive nature streamline the sorting process, guaranteeing consistent performance across datasets of varying sizes. As the Rugby Club application continues to evolve and scale, the efficacy of Binary Search and Merge Sort will remain central to its success. By harnessing these powerful algorithms, the application can optimize data management processes, improve decision-making capabilities, and deliver unparalleled value to coaches, administrators, players, and fans alike. As computational demands continue to grow, the strategic integration of Binary Search and Merge Sort ensures that the Rugby Club application remains at the forefront of sports management innovation, driving efficiency, performance, and success on and off the field.

**REFERENCE**:

Almaz, E. (2022). *What Is a Use Case & How To Write One | Wrike*. [online] www.wrike.com. Available at: https://www.wrike.com/blog/what-is-a-use-case/.

Almaz, E. (2023a). *Java algorithms: Implementing common search and sorting algorithms | Reintech media*. [online] reintech.io. Available at: https://reintech.io/blog/java-algorithms-implementing-common-search-and-sorting-algorithms.

Almaz, E. (2023b). *Merge Sort with Java*. [online] Java Challengers. Available at: https://javachallengers.com/merge-sort-with-java/#:~:text=It%20is%20an%20efficient%20sorting [Accessed 12 May 2024].

Almaz, E. (2024). *Mapping Diagrams*. [online] www.varsitytutors.com. Available at: https://www.varsitytutors.com/hotmath/hotmath\_help/topics/mapping-diagrams.

Eyup (2017). *Sorting Algorithms - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/sorting-algorithms/.

EyupAlmaz (2019). *Binary Search - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/binary-search/.