

# CCT College Dublin Continuous Assessment

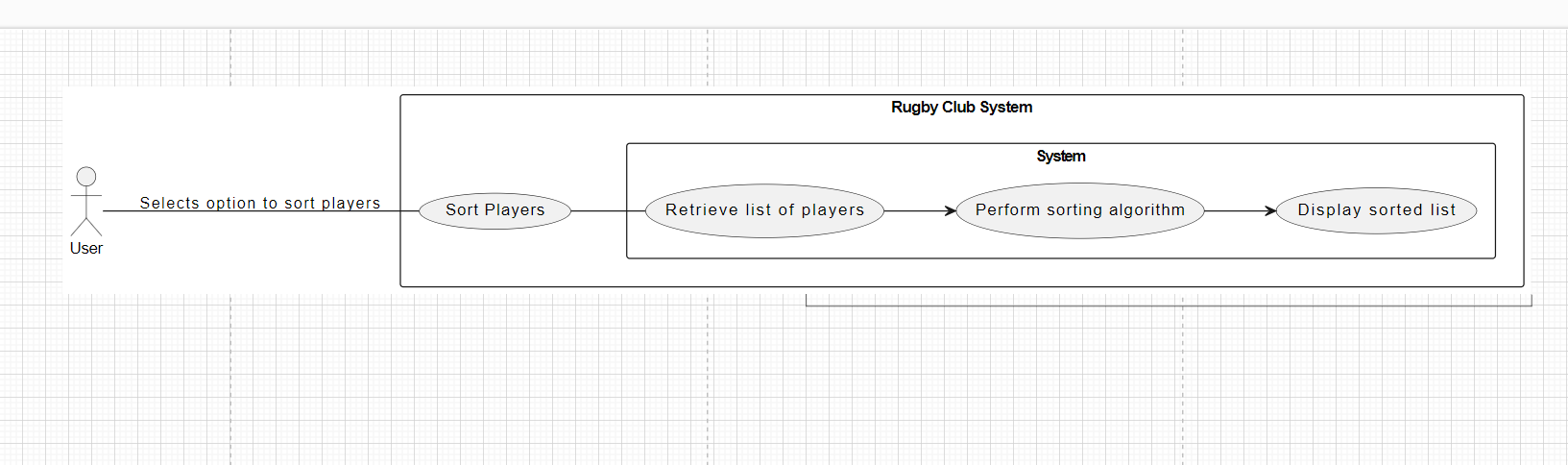
|  |  |  |  |
| --- | --- | --- | --- |
| **Program Title:** | *Higher Diploma in Science in Computing* | | |
| **Cohort:** | *Sept 23 FT* | | |
| **Module Title(s)**: | Software Development Fundamentals  Project Skills and Professionalism Algorithms & Constructs | | |
| **Assignment Type:** | *Individual* | Weighting(s): | *Software Development*  *Fundamentals - 55%*  *Project Skills and*  *Professionalism - 60%*  *Algorithms & Constructs - 60%* |
| **Assignment Title:** | *Rugby Club System* | | |
| **Lecturer(s)**: | *Kayoum Khbuli*  *James Garza*  *Ken Healy* | | |
| **Submitted By:** | *Mazan Javaid* | | |
| **Student No:** | *2023339* | | |
| **Issue Date:** | *15th April 2024* | | |
| **Submission**  **Deadline Date:** | *12th May 2024* | | |
| **Method of Submission:** | **Moodle** | | |
| **GitHub Repo**  **Link:** | [**Link to Repository**](https://github.com/mazan2023/H-DIP-CA2) | | |

**Software Development Fundamentals**

**Use Cases:**

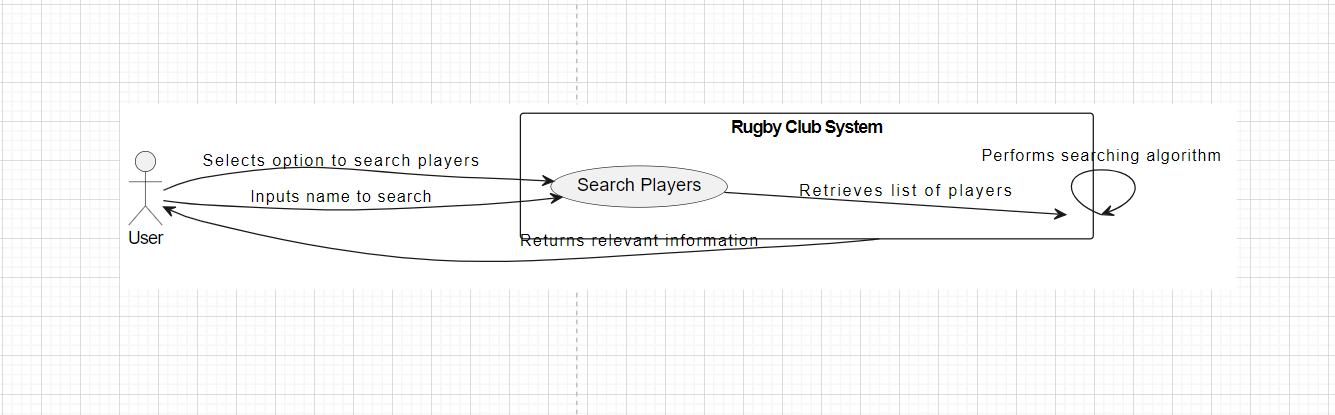
|  |  |
| --- | --- |
| **Use Case #1** | **Sort Players** |
| **Actor:** | User |
| **Description:** | It allows the user to sort the list of players alphabetically. |
| **Preconditions:** | The system has a list of players. |
| **Postconditions:** | The list of players is sorted alphabetically. |
| **Main Flow** | |
| 1. User selects the option to sort players. | |
| 1. System retrieves the list of players. | |
| 1. System performs sorting algorithm (e.g. Quicksort) recursively. | |
| 1. Sorted list is displayed on the screen. | |

**Diagram #1**

****

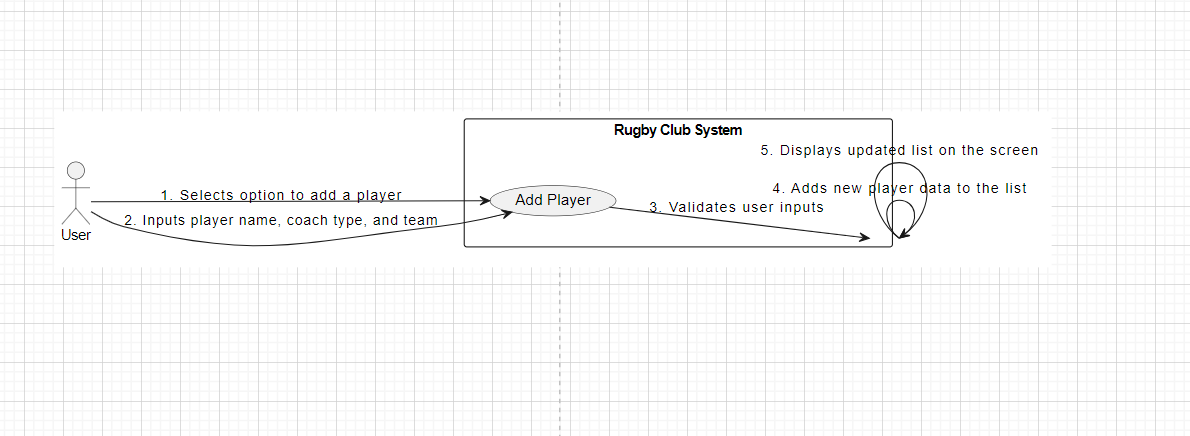
|  |  |
| --- | --- |
| **Use Case #2** | **Search Players** |
| **Actor:** | User |
| **Description:** | It enables the user to search for players by name and retrieve their coach type and team name. |
| **Preconditions:** | The system has a list of players. |
| **Postconditions:** | The relevant information (coach type and team name) of the searched player is displayed. |
| **Main Flow** | |
| 1. User selects the option to search players. | |
| 1. User inputs the name to search. | |
| 1. System retrieves the list of players. | |
| 1. System performs searching algorithm (e.g., Binary Search). | |
| 1. Relevant information is returned and displayed. | |

**Diagram #2**

****

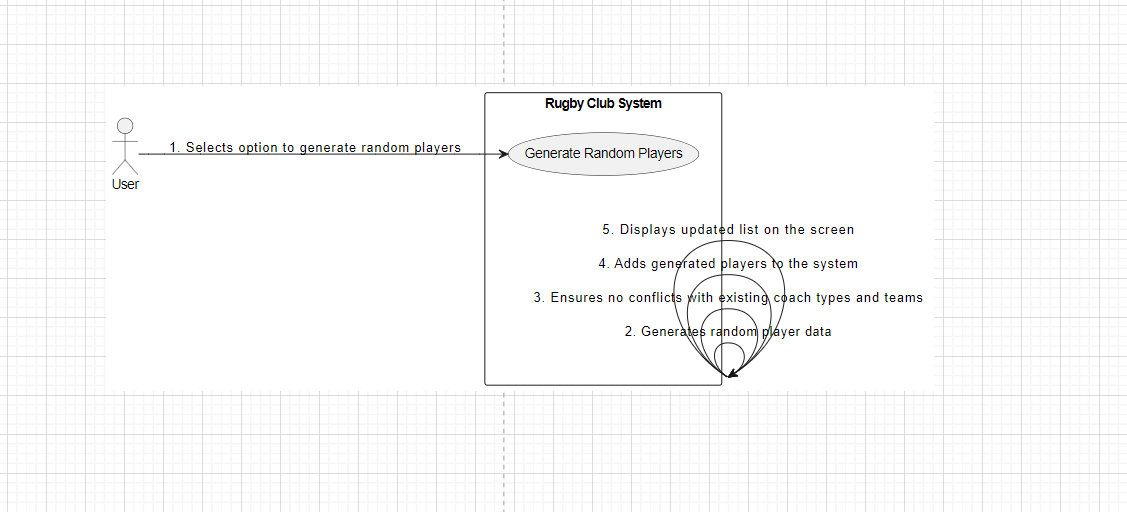
|  |  |
| --- | --- |
| **Use Case #3** | **Add Player** |
| **Actor:** | User |
| **Description:** | It allows the user to add a new player to the system by providing player name, coach type, and team. |
| **Preconditions:** | The system is running and has available coach types and teams. |
| **Postconditions:** | The new player is added to the system. |
| **Main Flow** | |
| 1. User selects the option to add a player. | |
| 1. User inputs player name, coach type, and team. | |
| 1. System validates user inputs. | |
| 1. New player data is added to the list. | |
| 1. Updated list is displayed on the screen. | |

**Diagram #3**



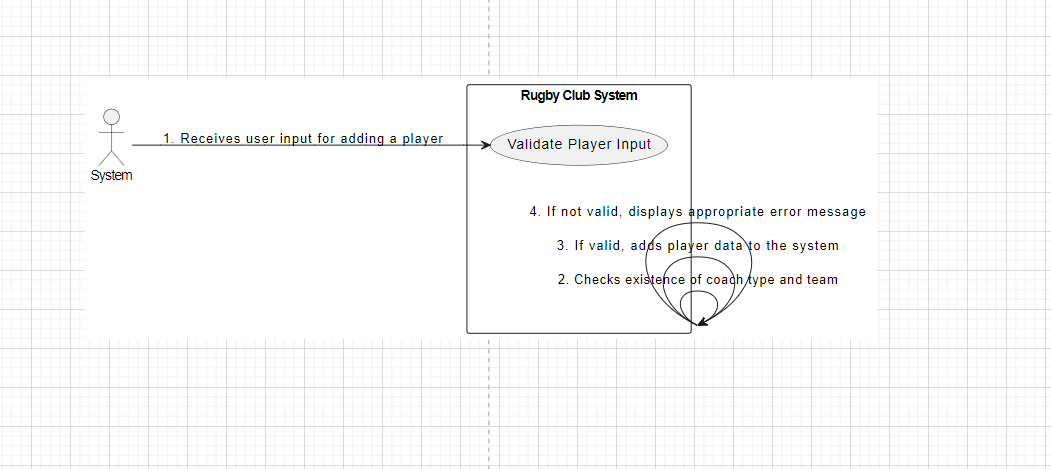
|  |  |
| --- | --- |
| **Use Case #4** | **Generate Random Players** |
| **Actor:** | User |
| **Description:** | It enables the user to generate random players with random coach types and teams. |
| **Preconditions:** | The system is running. |
| **Postconditions:** | Random players are added to the system without conflicting with existing coach types and teams. |
| **Main Flow** | |
| 1. User selects the option to generate random players. | |
| 1. The system generates random player data. | |
| 1. System ensures no conflicts with existing coach types and teams. | |
| 1. Generated players are added to the system. | |
| 1. Updated list is displayed on the screen. | |

**Diagram #4**



|  |  |
| --- | --- |
| **Use Case #5** | **Validate Player Input** |
| **Actor:** | System |
| **Description:** | It ensures that user input for adding a player is valid by checking the existence of coach type and team. |
| **Preconditions:** | User inputs player data for adding a player. |
| **Postconditions:** | User input is validated, and errors are handled appropriately. |
| **Main Flow** | |
| 1. System receives user input for adding a player. | |
| 1. System checks the existence of coach type and team. | |
| 1. If valid, player data is added to the system. If not, an appropriate error message is displayed. | |

**Diagram #5**

****

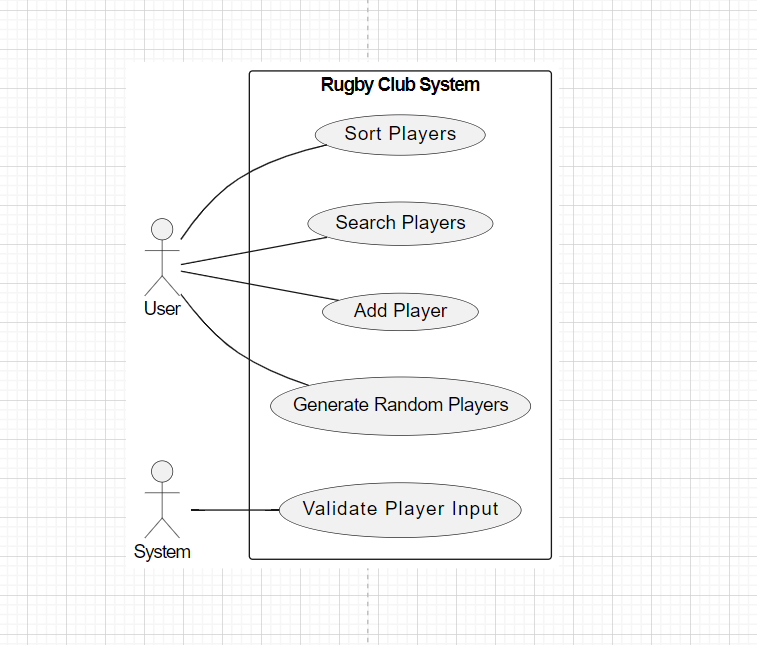
**UML Modeling Techniques:**

**1. Use Case Diagram:**

A Use Case Diagram provides a high-level view of the system's functionality and the interactions between actors (users) and the system.

**Use Case Diagram for Rugby Club System:**

**Diagram#6**

****

**Explanation:**

**Actors:**

**User (U):** Represents the user interacting with the system.

**System (S):** Represents the system itself.

**Use Cases:**

**Sort Players:** Allows the user to sort the list of players.

**Search Players:** Enables the user to search for players by name and retrieve their coach type and team name.

**Add Player:** Allows the user to add a new player to the system.

**Generate Random Players**: Enables the user to generate random players with random coach types and teams.

**Validate Player Input:** Ensures that user input for adding a player is valid by checking the existence of coach type and team.

**2. Activity Diagram:**

An Activity Diagram models the flow of activities or actions within a use case, showing the sequence of steps involved in performing a task.

**Activity Diagram for Rugby Club System:**

**Diagram#7**

**A diagram of a football game

Description automatically generated**

**Explanation:**

**Steps:**

* **User Selects Add Player Option:** The user selects the option to add a new player to the system.
* **User Inputs Player Name, Coach Type, and Team:** The user provides the necessary details for the new player, including name, coach type, and team.
* **System Validates User Inputs:** The system validates the user inputs to ensure they are valid.
* **Add Player Data to System:** If the inputs are valid, the system adds the new player data to the system.
* **Display Updated Player List on Screen**: The system displays the updated list of players on the screen.

These models provide a comprehensive view of the Rugby Club System's functionality and the flow of activities within specific use cases, helping to understand the system's behavior and interactions.

**Factors to consider while selecting Use Case and Activity Diagram as modeling techniques:**

|  |  |
| --- | --- |
| **Use Case Diagrams** | |
| **High-level System Understanding:** | It provides a high-level view of system functionality and user interactions. In the context of the Rugby Club System, this diagram effectively captures the core functionalities such as sorting players, searching players, adding players, generating random players, and validating player input, aligning well with the system's requirements. |
| **Actor-System Interactions:** | Use Case Diagrams clearly depict the interactions between actors (users) and the system. In the case of the Rugby Club System, actors like "User" and "System" are identified, and their interactions with system functionalities are illustrated. This representation is essential for understanding how users interact with the system to accomplish specific tasks. |

|  |  |
| --- | --- |
| **Activity Diagram** | |
| **Detailed Process Flow:** | Activity Diagrams provide a detailed view of the sequential steps involved in performing a specific task or use case. For example, in the "Add Player" use case, the Activity Diagram delineates each step from user input to system validation and data addition, providing a clear and detailed process flow. This level of detail is crucial for understanding the intricacies of system functionality. |
| **Visual Representation of Flow:** | Activity Diagrams offer a visual representation of the flow of activities within a use case, making it easier to comprehend complex processes. In the context of the Rugby Club System, activities such as user input validation and data addition are visually represented, aiding in understanding the system's behavior and logic. |

**Comparison**

|  |  |  |  |
| --- | --- | --- | --- |
| **Use Case Diagram** | **Sequence Diagram** | **Activity Diagram** | **State Diagram** |
| Provide a high-level overview of the system's functionality and user interactions, focusing on system capabilities and user goals. | Offer a detailed view of the interactions between objects (actors or components) in the system, showing the sequence of messages exchanged between them. | Represent the flow of activities or actions within a use case, showing the sequence of steps involved in performing a task. | Model the behavior of individual objects or components in the system, emphasizing their states and transitions between states. |
| Emphasize interactions between actors (users) and the system, illustrating how users interact with system functionalities to achieve specific goals. | Emphasize dynamic behavior and temporal ordering of interactions, suitable for modeling specific scenarios or user actions in detail. | Emphasize sequential logic and decision points, suitable for modeling detailed processes with multiple steps and decision branches. | Suitable for modeling systems with discrete states and state transitions, such as systems with complex state-based logic or finite state machines. |
| Suitable for capturing system requirements and core functionalities, making them ideal for representing features such as sorting players, searching players, adding players, etc., in the Rugby Club System. | Can become complex, especially in systems with numerous interactions or complex message flows, potentially leading to cluttered diagrams that are difficult to comprehend. | Provide a clear and visual representation of process flows, aiding in understanding the sequence of actions and decision points within a use case. | Can become complex, especially in systems with numerous states and transitions, potentially leading to diagrams that are difficult to interpret or maintain. |
| **Conclusion**  In conclusion, Activity and Use Case Diagrams were chosen for modeling the Rugby Club System due to their suitability for capturing high-level system functionality, user interactions, and detailed process flows. While Sequence and State Diagrams offer valuable insights into interactions and state-based behavior respectively, they may not be the most appropriate for modeling the core features and user interactions of the Rugby Club System | | | |

**Why Activity and Use Case Diagrams were Chosen:**

* **High-Level System Understanding:**

**Use Case Diagrams:** Provide a high-level overview of system functionality and user interactions, effectively capturing core features such as sorting players, searching players, adding players, etc.

**Activity Diagrams:** Offer a detailed representation of the sequential steps involved in performing specific tasks, aiding in understanding detailed process flows within a use case.

* **User Interaction Emphasis:**

Both Activity and Use Case Diagrams emphasize interactions between users and the system, highlighting how users interact with system functionalities to achieve specific goals such as adding players, searching players, etc.

* **Clarity and Visualization:**

Activity Diagrams provide a clear and visual representation of process flows, making it easier to understand the sequence of actions and decision points within a use case.

Use Case Diagrams offer a visual representation of system functionality and user interactions, aiding in the comprehension of high-level system behavior.

* **Comprehensiveness and Balance:**

The combination of Use Case and Activity Diagrams offers a comprehensive view of the system, providing both high-level overviews of system functionality and detailed process flows within specific use cases.

**Why Sequence and State Diagrams were Not Chosen:**

* **Complexity and Detail:**

Sequence Diagrams offer a detailed view of object interactions and message flows, which may introduce unnecessary complexity for capturing the core features and user interactions of the Rugby Club System.

State Diagrams model state-based behavior and transitions, which may not be directly applicable to representing the core functionalities and user interactions of the Rugby Club System, leading to diagrams that are overly detailed and complex.

* **Focus on Dynamic Behavior:**

Sequence Diagrams emphasize dynamic behavior and temporal ordering of interactions, which may not be the primary focus when modeling high-level system functionality and user interactions.

* **Limited Relevance:**

While Sequence and State Diagrams offer valuable insights into interactions and state-based behavior, they may not directly align with the core features and user interactions of the Rugby Club System, making them less suitable for modeling its functionality.

**Summary**

In summary, Activity and Use Case Diagrams were chosen as they offer a balanced approach to modeling the Rugby Club System, effectively capturing both high-level system functionality and detailed process flows within specific use cases, while Sequence and State Diagrams may introduce unnecessary complexity or may not directly align with the system's core features and user interactions.

**User Stories:**

**User Story 1:** As a coach, I want to be able to search for players by name so that I can view their coach type and team.

**Acceptance Criteria:**

* When I search for a player by name, the system should display their coach type and team name.
* If the player is not found, the system should display a message indicating that the player is not in the system.

**Acceptance Tests:**

* Given I am a coach searching for a player named "John Doe."
* When I search for the player by name
* Then the system should display "John Doe" with coach type and team name.
* Given I am a coach searching for a player named "Jane Smith."
* When I search for the player by name
* Then the system should display a message indicating that "Jane Smith" is not in the system.

**Unit Tests:**

* Test that the search functionality retrieves the correct player information.
* Test that the search functionality handles cases where the player is not found.

**User Story 2:** As a coach, I want to be able to add a new player to the system.

**Acceptance Criteria:**

* When I add a new player with valid information, the player should be added to the system.
* If I provide invalid information, the system should display an error message.

**Acceptance Tests:**

* Given I am a coach adding a new player with valid information.
* When I input the player's name, coach type, and team.
* Then the system should add the player to the system.
* Given I am a coach adding a new player with invalid information.
* When I input invalid information
* Then the system should display an error message.

**Unit Tests:**

* Test that the system correctly adds a new player with valid information.
* Test that the system displays an error message for invalid information.

**User Story 3:** As an administrator, I want to generate random players with random coach types and teams.

**Acceptance Criteria:**

* When I generate random players, they should have unique names and random coach types and teams.
* The generated players should be added to the system without conflicts with existing coach types and teams.

**Acceptance Tests:**

* Given I am an administrator generating random players.
* When I select the option to generate random players
* Then the system should add randomly generated players to the system.

**Unit Tests:**

* Test that the system generates random players with unique names.
* Test that the system adds the randomly generated players without conflicts with existing coach types and teams.

**User Story 4:** As a user, I want to be able to sort the list of players alphabetically.

**Acceptance Criteria:**

* When I choose to sort the list of players, the system should display the players in alphabetical order.
* The sorting algorithm should efficiently organize the player list.

**Acceptance Tests:**

* Given I am a user choosing to sort the list of players.
* When I select the option to sort
* Then the system should display the players in alphabetical order.

**Unit Tests:**

* Test that the sorting algorithm correctly sorts the player list alphabetically.
* Test the efficiency of the sorting algorithm for a large player list.

**User Story 5:** As a coach, I want to be able to validate player input before adding it to the system.

**Acceptance Criteria:**

* When I input player information, the system should validate the coach type and team.
* If I provide invalid coach type or team, the system should display an error message.

**Acceptance Tests:**

* Given I am a coach providing valid player information.
* When I input the player's name, coach type, and team.
* Then the system should validate the information and add the player to the system.
* Given I am a coach providing invalid player information.
* When I input invalid coach type or team
* Then the system should display an error message.

**Unit Tests:**

* Test that the system correctly validates player input with valid information.
* Test that the system displays an error message for invalid coach type or team.

**Algorithms & Constructs**

**Sorting Algorithm: Merge Sort**

Merge sort has been employed in the system to sort the list of people. Merge sort is chosen due to its stable performance across various scenarios. Its time complexity is consistently O (n log n) making it efficient for large datasets.

The merge sort algorithm divides the array into smaller sub-arrays, sorts them individually, and then merges them back together. This divide-and-conquer approach ensures that the sorting process is efficient and reliable. Additionally, merge sort is suitable for sorting objects with complex data structures, like the Person objects in this code, as it only requires comparisons between elements.

**Advantages**

* **Consistent Performance:** Merge sort performs consistently well even with large datasets, thanks to its time complexity of O (n log n).
* **Stability:** Merge sort is a stable sorting algorithm, meaning it preserves the relative order of equal elements. This property is crucial when sorting complex objects based on multiple criteria.
* **Suitability for Linked Lists:** Merge sort performs efficiently on linked lists due to its ability to handle sequential access patterns.

**Comparison**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Space Complexity** | **Stability** | **Suitability** | **Comparison** |
| **Merge Sort** | **O (n log n)**  In all cases | **O (n)**  For auxiliary array. | **Yes** | Large datasets, linked lists, and objects with complex structures. | Merge sort consistently performs well with its time complexity, making it suitable for general-purpose sorting. |
| **Quick Sort** | **O (n log n)** (average case)  **O (n^2)**  (worst case) | **O (log n)**  Due to recursive calls | **No** | Large datasets, especially when average case performance matters more than worst-case scenarios. | Quick sort can outperform merge sort in practice due to lower constant factors, but its worst-case behavior makes it less desirable for critical applications. |
| **Insertion Sort** | **O (n^2) (average & worst case)** | **O (1)** | **Yes** | Small datasets or nearly sorted lists. | Insertion sort is simple and efficient for small datasets but becomes inefficient for larger ones due to its quadratic time complexity. |

**Conclusion**

Merge sort offers consistent performance and stability, making it suitable for general use. Quick sort can outperform merge sort in practice but has a higher risk of worst-case behavior. Insertion sort is efficient for small datasets but impractical for larger ones due to its quadratic time complexity.

**Searching Algorithm: Linear Search**

Linear search is utilized in the program to find a person by name within the list of people. Although linear search is not the most efficient search algorithm for large datasets, it is suitable for this context because the list of people is expected to be relatively small.

**Advantages**

* **Simplicity:** Linear search is straightforward to implement and understand, making it suitable for small-scale applications.
* **Flexibility:** Linear search can be applied to both sorted and unsorted lists without any preprocessing, making it versatile.
* **Minimal Memory Usage:** Linear search requires minimal additional memory beyond the list being searched, making it memory efficient.

**Comparison**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Space Complexity** | **Suitability** | **Comparison** |
| **Linear Search** | **O (n)**  In worst case | **O (1)** | Small datasets or when the element to be found is likely to be near the beginning of the list. | Linear search is straightforward but inefficient for large datasets since it scans each element sequentially. |
| **Binary Search** | **O (log n)**  (For sorted Arrays) | **O (1)** | Sorted arrays or lists. | Binary search is highly efficient for large, sorted datasets, but it requires the array to be sorted beforehand, which may incur additional overhead. |
| **Hashing**  **(Hash Table)** | **O (1)** | **O (n)** | Retrieving elements quickly with known keys. | Hashing provides constant-time search in average cases but requires additional space for hashing functions and collision resolution. |

**Conclusion**

Linear search is simple but inefficient for large datasets. Binary search is highly efficient but requires sorted data. Hashing provides constant-time search but requires additional space. The choice depends on factors such as dataset size, structure, and search requirements.

**Project Skills and Professionalism**

**Problem Statement:**

The task at hand involves creating a software solution for managing a Rugby Club. This involves various functionalities such as reading data from files, sorting, searching, adding players, and generating random players.

**Key Challenges:**

|  |  |
| --- | --- |
| **Challenge** | **Description** |
| **Data Management:** | Efficiently managing data about club members, including their names, coach types, and team names. |
| **Sorting:** | Implementing a sorting algorithm to organize club members alphabetically by name. |
| **Searching:** | Developing a search algorithm to find club members by name. |
| **User Interaction:** | Providing a user-friendly interface for adding new players and interacting with the program. |
| **Scalability:** | Ensuring the program is scalable to handle potential future expansions of the rugby club. |

**Requirements:**

* Read data from file.
* Sort club members alphabetically by name.
* Search for club members by name.
* Add new players with coach types and team names.
* Generate random players with unique combinations of name, coach type, and team name.
* Provide a user-friendly interface for user interaction.

**Solutions and Rationale**

|  |  |  |
| --- | --- | --- |
| **Challenge** | **Solution** | **Rationale** |
| **Data Management** | Utilizing object-oriented programming principles to create a Person class to represent club members with attributes for name, coach type, and team name. | This approach organizes data efficiently and allows for easy manipulation and access. |
| **Sorting** | Implementing the Merge Sort algorithm for sorting club members alphabetically by name. | Merge Sort offers stable performance with a time complexity of O (n log n) making it suitable for sorting large datasets like the club members' list. |
| **Searching** | Implementing Linear Search for searching club members by name. | Linear Search is simple to implement and suitable for the relatively small dataset expected in this context. However, for larger datasets, other search algorithms like Binary Search could be considered for improved efficiency. |
| **User Interaction** | Providing a menu-driven interface allowing users to choose options like sorting, searching, adding players, and generating random players. | This approach offers clear and intuitive interaction, guiding users through available functionalities. |
| **Scalability** | Designing the program with modular and scalable architecture, separating concerns like data management, sorting, searching, and user interaction into distinct components. | This design allows for easier maintenance and future expansions of the program, accommodating potential growth in the rugby club. |

**Strengths:**

* Clear and modular design facilitates maintenance and scalability.
* Efficient sorting and searching algorithms ensure optimal performance.
* User-friendly interface enhances usability and accessibility.

**Weaknesses:**

* Linear search may become inefficient for larger datasets, requiring consideration of alternative search algorithms.
* The program's scalability may be limited by the chosen implementation choices, potentially requiring redesign for significant expansions.

**Alternative Approaches:**

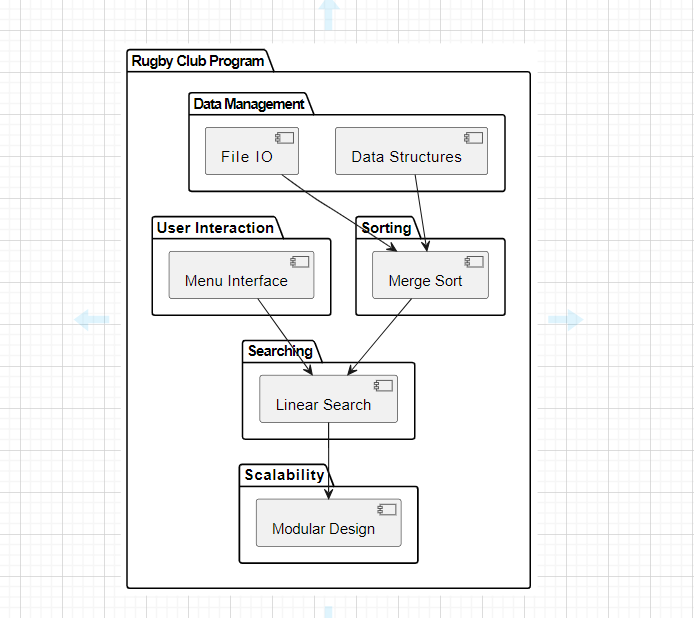
**Sorting Algorithm:** Instead of Merge Sort, Quick Sort could be considered for potentially better average-case performance. However, Quick Sort's worst-case behavior may pose a risk.

**Searching Algorithm:** Binary Search could be chosen over Linear Search for improved efficiency, especially for larger datasets. However, it requires the data to be sorted beforehand, adding complexity.

**User Interaction:** A graphical user interface (GUI) could be developed for enhanced user experience, though it may require additional development effort.

**Problem definition and mapping diagram**

A mapping diagram will visually represent the relationship and interactions between the different components of the Rugby Club program, including data management, sorting, searching, user interaction, and scalability.



* Data structures and file I/O operations feed into the sorting component, indicating that data from files is managed and then sorted using Merge Sort.
* Both sorting and the menu interface interact with the searching component, suggesting that sorted data can be searched through the menu interface.
* Finally, the Linear Search component interacts with the modular design aspect of scalability, indicating that the search functionality is designed to be modular and scalable.

**Risk Assessment**

|  |  |  |
| --- | --- | --- |
| **Risk** | **Description** | **Solution** |
| Data Integrity Risk | There is a risk of data corruption or loss during file I/O operations. | Implement error handling and data validation mechanisms to ensure data integrity.  For example, validate input data formats and handle file reading errors gracefully. |
| Performance Risk | Sorting and searching algorithms may not perform optimally for large datasets, impacting program performance. | Choose algorithms with reasonable time complexities and implement optimizations where possible.  For example, use Merge Sort for sorting as it has a time complexity of O (n log n), which is efficient for large datasets.  Additionally, consider implementing binary search for searching, which has a time complexity of O (log n) for sorted data. |
| User Error Risk | Users may input incorrect data or misuse the program, leading to unintended outcomes. | Provide clear instructions and error messages to guide users through interactions with the program. Implement input validation to prevent invalid data entry.  For example, validate user inputs for adding new players to ensure.  they are within acceptable ranges. |
| Scalability Risk | The program may not scale effectively to handle potential future expansions of the rugby club. | Design the program with modularity and scalability in mind.    Separate concerns into distinct components, such as data management, sorting, searching, and user interaction, to facilitate future modifications and expansions.  Additionally, use data structures and algorithms that can handle larger datasets efficiently. |

GitHub Link: https://github.com/mazan2023/H-DIP-CA2

Note: It is requested that currently this repository is private so that any chance of plagiarism can be avoided, and I have sent the invitation link to all the instructors to this repository as collaborators. I will make this repository public after 5 days of the submission for the ease of instructors.