Subsystem Interface Design

for

Edumon

Version 1.0 approved

Prepared by Group1:

Beh Chee Kwang Nicholas (U1824125F)
Chen Xueyao (U1922640G)
Chong Jing Hong (U1922300B)
Goh Hong Xiang, Bryan (U1920609E)
Huang Shaohang (U1921245D)
Lim Jun Wei (U1922756C)
Muhammad Al-Muhazerin (U1921191L)
Quah Dian Wei (U1920106C)
Shauna Tan Li-Ting (U1840754E)
Swa Ju Xiang (U1822040G)

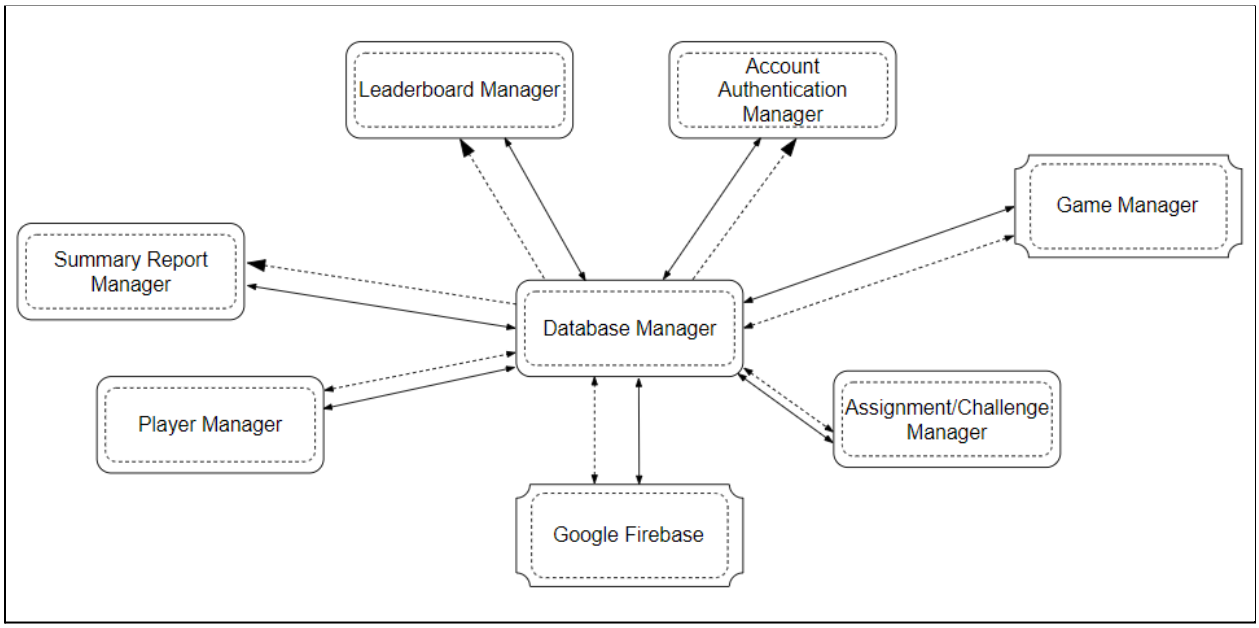
18/10/2021

Table of Contents

Revision History

Name	Date	Reason for change	Version
Everyone	13/09/2021	Initial draft	0.1
Everyone	17/10/2021	For submission	1.0

Subsystem Architectures

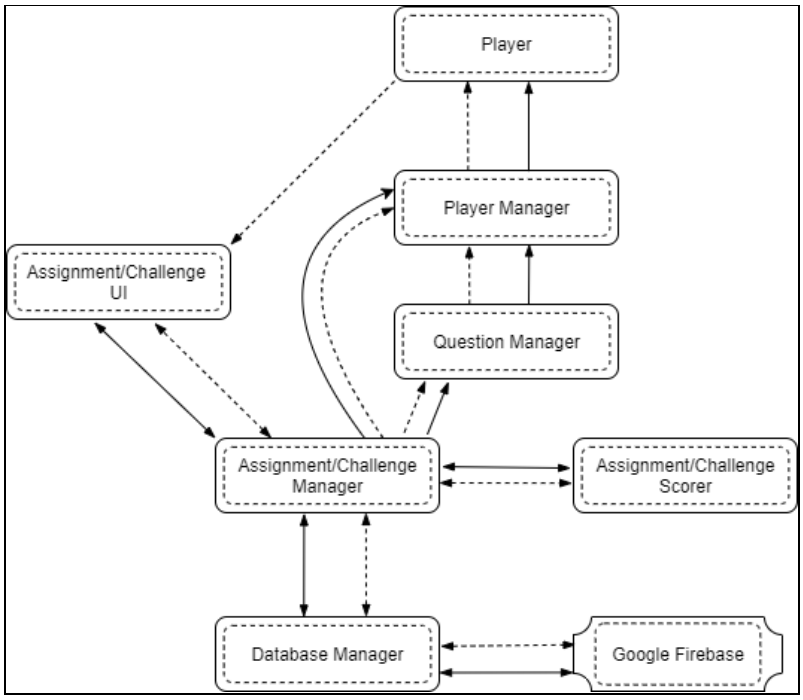


Functions:

- authManager(): void
  - This function calls the authentication manager
- leaderboardManager(): void
  - This function calls the leaderboard manager
- reportManager(): void
  - This function calls the report manager
- playerManager(): int
  - This function calls the player manager
- questionManager(): void
  - This function calls the question manager
- gameManager(): void
  - This function calls the game manager
- assignmentChallengeManager(): void
  - This function calls the assignment / challenge manager

The above shows the managerial interactions, mainly from the main database of the application to the respective managers, which uses a Call-and-Return architecture. The entire architect of the application can be depicted with the diagram above. Encompassing each manager are their own architects that if depicted in the diagram above would take up much space, hence a more detailed version can be found below.

Subsystem Interface Design (Assignment/Challenge)



Assignment/Challenge Interface
This interface supports the use cases of creating challenges and assignments, viewing challenges and assignments, as well as completing challenges and assignments.

Functions:

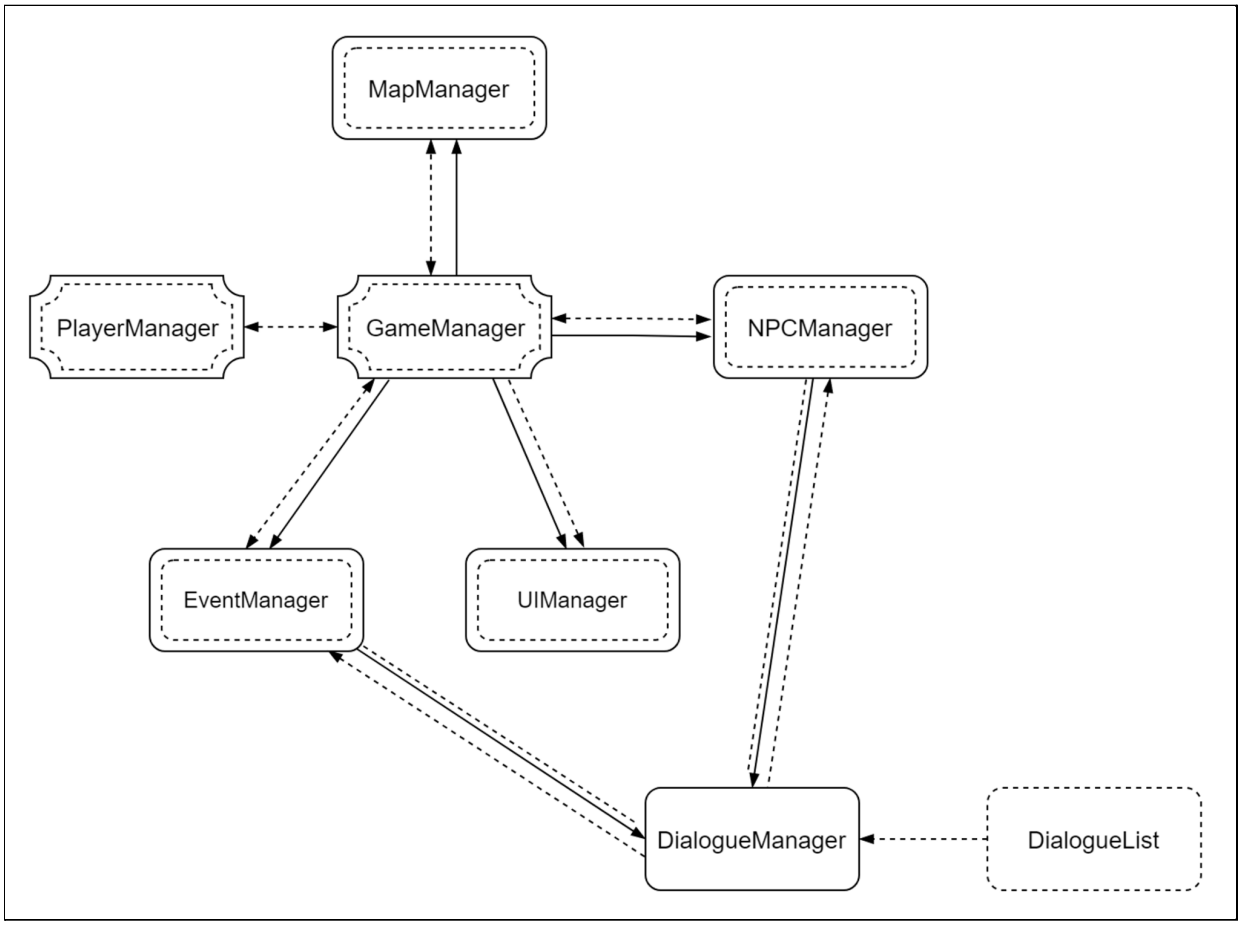
- createChallenges(int studentID): void
    - This function allows players to create challenges for other players.
- displayChallenges(int studentID): void
    - This function shows the challenges a player has.
- submitChallenges(int studentID, int challengeID): void
    - This function submits the students' answers for a particular challenge.
- getChallengeScore(int studentID, int challengeID): int
    - This function gets the students' score for a particular challenge.
- createAssignment(): void
    - This function allows teachers to create challenges for their players.
- displayAssignment(int studentID): void
    - This function shows the challenges a player has.
- submitAssignment(int studentID, int assignmentID): void
    - This function submits the students' answers for a particular assignment.
- getAssignmentScore(int studentID, int assignmentID): int
    - This function gets the students' score for a particular assignment.

For the Assignment/Challenge subsystem, we decided to implement an Event-Based architecture. The Assignment/Challenge Manager is at the center of this architecture, giving and receiving control and data to other components of the architecture, whenever a new assignment or challenge is created.

A Player accesses the Assignment/Challenge UI to create assignments or challenges. The UI will pass the control and data to the Assignment/Challenge Manager, which will store the assignment or challenge in Google Firebase through the Database Manager. The Assignment/Challenge Manager will then forward the assignment or challenge to the relevant players through the Question Manager and Player Manager.

To complete assignments or challenges, Players will access the Assignment/Challenge UI. Once done, the UI will forward the answers to the Assignment/Challenge Manager, which will pass them to the Assignment/Challenge Scorer for scoring. The results of the scoring will be pushed to Google Firebase through the Assignment/Challenge Manager and Database Manager, and the Players will be informed of their scores through the Player Manager.

Subsystem Interface Design (World)



World Interface

This interface supports the use cases of exploring the world and moving between different areas in the world and between different maps

Functions:

- enterGym(): void
  - This function allows the player to enter the gym in the map.
- exitGym(): void
  - This function allows the player to exit the gym in the map.
- wildGrassEvent(): void
  - This function allows the player to interact with wild grass in the map.
- retrieveDialogue(string type, string name): string
  - This function retrieves the dialogue for the particular event type and name (if applicable) from the database.
- displayMaps(): void

    - This function allows the player to view all the different maps, and shows which are unlocked and which are locked.

- moveMap(int mapNum): void
    - This function moves the player to the chosen map based on the map number.

- moveToNextMap(): void
    - This function moves the player to the next map (if applicable).

- generateNPCs(int num): List of NPC objects
    - This function generates a number (num) of Non-playable Characters (NPCs) for that map.

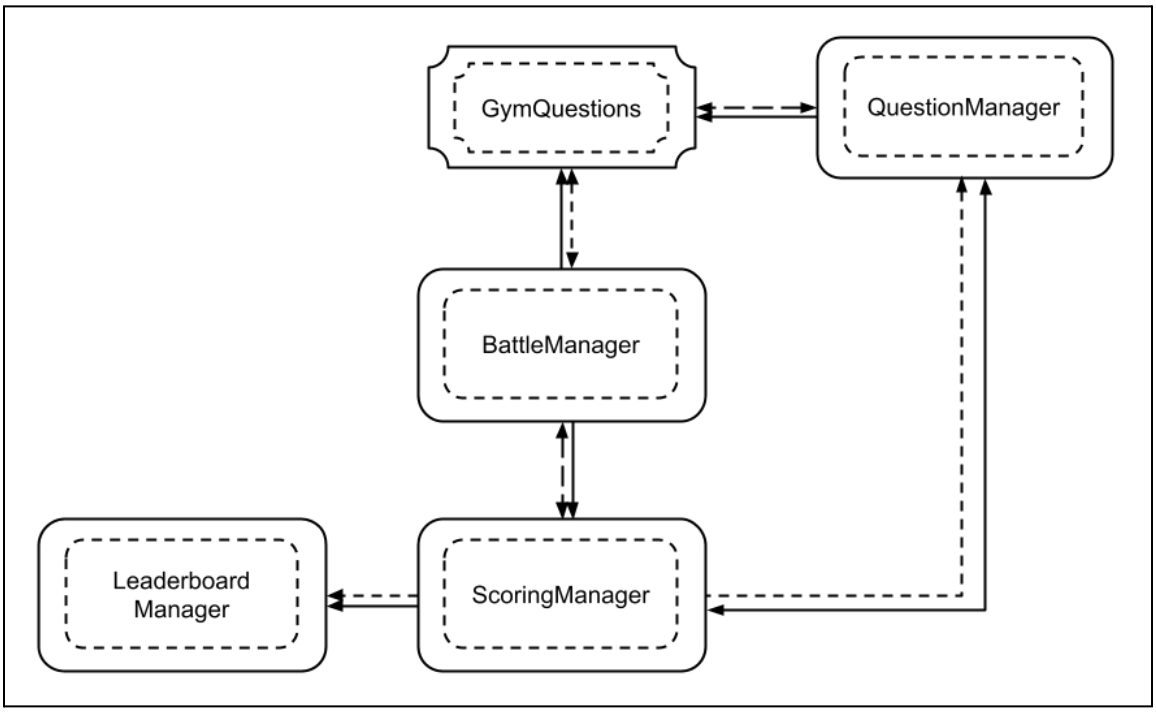
For the World subsystem, we chose a Call-and-Return architecture centered around the GameManager program and with multiple subroutines such as EventManager, MapManager, etc. In this subsystem, GameManger is the main program/subroutine. It controls most of the program and sends data to the UIManager for display updating.

GameManager will decide how many Non-playable characters (NPCs) to spawn, and pass this information to the NPCManager. The NPCManager will then decide on which NPCs to spawn, and send this information to DialogueManager which will gain control and obtain the dialogue for this NPC from DialogueList. The same applies for EventManager, which handles the events that occur when the player interacts with the world. EventManager will send information on the event that was triggered to DialogueManager, which will obtain the dialogue for the event.

PlayerManager handles the movement of the player and player data, and sends and receives information to and from GameManager. MapManager handles the movement between different maps. Lastly, the UIManager handles the display of the game to the user.

As seen in the diagram, the main program with subroutine architecture allows us to pass data and control as and when it is needed, allowing us to use modular design to handle complexity and to allow us to easily modify the program as and when modules need to be changed, or updated.

Subsystem Interface Design (Gym)



Gym Battle Interface

This interface supports the use cases of students performing battles in gyms and the update of gym questions and student's leaderboard ranking.

Functions:

- addQuestion(Question question): void
  - This function allows teachers to add gym battle questions.
- deleteQuestion(Question question): void
  - This function allows teachers to delete gym battle questions.
- startBattle(int difficulty): void
  - This function creates a gym battle instance containing questions of the specified difficulty.
- showScore(): void
  - This function calculates and displays the students' gym battle results
- update(): void
  - This function calls sortRankings and updatePersonalScore (please see "Leaderboard Interface" below.

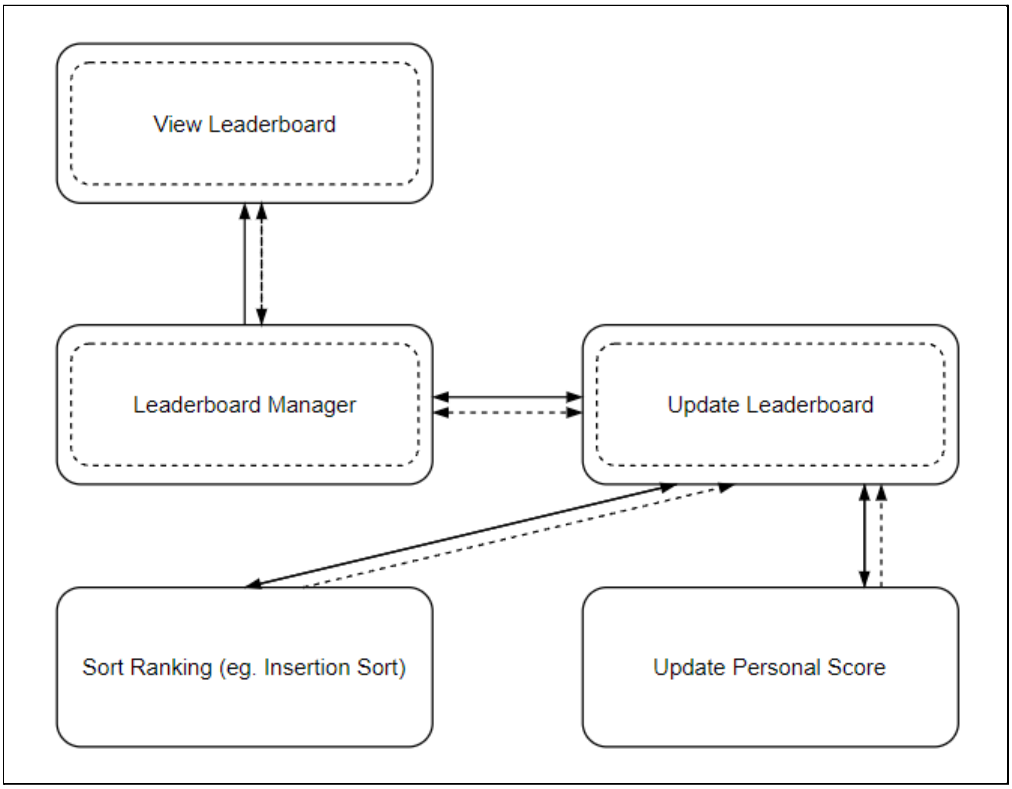
For the Gym Battle subsystem, we chose a Call-and-Return architecture centered around the BattleManager program and with multiple subroutines such as ScoringManager and QuestionManager.

The advantages of using this architecture for the Gym Battle subsystem is that the modular design can handle complex instructions more easily. In addition, this architecture also has a high level of modifiability and modifications to this subsystem will not impact other subsystems much. The trade-off of using this architecture design is that each individual module may be more complex to design.

In this subsystem, BattleManager is the “main” program and first reads data from GymQuestions, which is an active data component that takes in user input from Teachers through QuestionManager.. The relevant data is forwarded to BattleManager, which creates and displays the battle.

Once the battle is completed, ScoringManager will calculate and display the student’s score. The score will be sent back to QuestionManager as feedback. The score will also be sent to the Leaderboard Manager for updating.

Subsystem Interface Design (Leaderboard)



Leaderboard Interface
This interface supports the update and viewage of the leaderboard

Functions:

- updatePersonalScore(Player player, int score): void
  - This function allows the system to assign the score to the unique individuals to the database
- sortRankings(Player player, int score, Leaderboard leaderboard): Leaderboard
  - This function adds the new player's score into the leaderboard does a sort algorithm
- update(): void
  - This function calls sortRankings and updatePersonalScore
- view(): Leaderboard
  - This function console writes the ranks of the students / returns the leaderboard

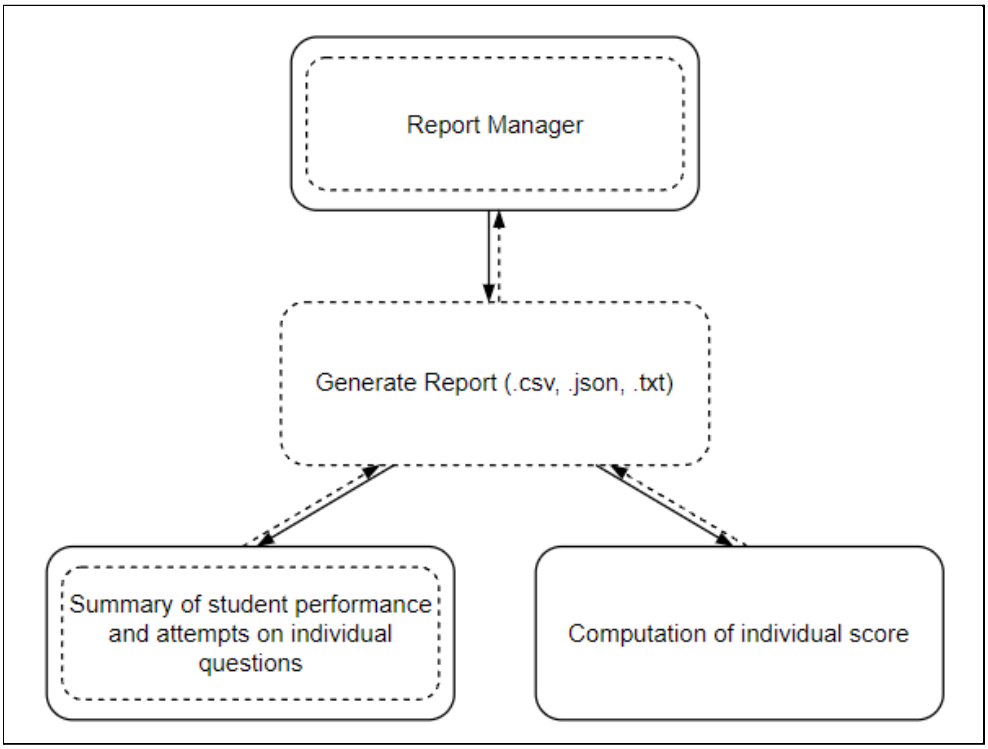
For the Leaderboard subsystem, we chose a Call-and-Return architecture centered around the *Leaderboard Manager* program and with multiple subroutines such as *View Leaderboard*, *Update Leaderboard* etc. In this subsystem, the *Leaderboard Manager* handles the control and calls *View Leaderboard* or *Update Leaderboard* according to the situation.

The general flow of the leaderboard is as follows. When there is an update, eg. a new player has completed the assignment / challenge, *Leaderboard Manager* will be called and *Leaderboard Manager* will in turn call *Update Leaderboard*.

Since there is a new input due to a new player completing the assignment or challenge and obtaining a score, the *Update Leaderboard* will call *Update Personal Score* to assign the score to the unique individual (eg. player456 → 80 / 100). Within the leaderboard there is also a ranking system where the players will be ranked according to their scores. Hence *Sort Ranking* would be called to sort out the ranking with insertion sort. With the updated player's score and updated rankings, the two sub-subsystem will then call back *Update Leaderboard* to return the new values.

Update Leaderboard will then call *Leaderboard Manager* to return the new ranking and player scoring. Leaderboard Manager will then call *View Leaderboard* and pass the new ranking / player scoring over for viewing.

Subsystem Interface Design (Report)



Report Interface
This interface supports the generation of reports.

Functions:

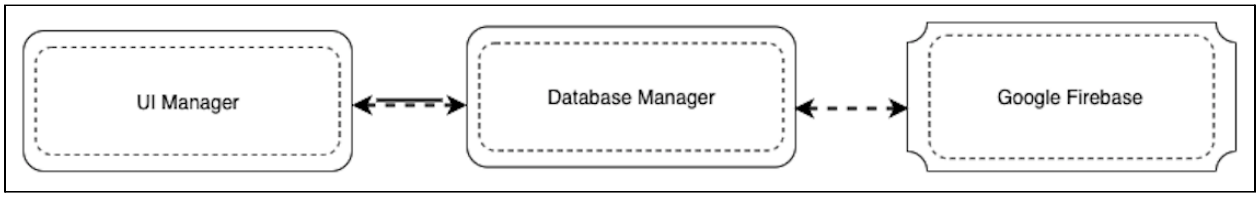
- computeScore(Player player, int score): int
  - This function calculates the total score for the player for that individual assignment / challenge
- getSummary(Player player): String
  - This function console writes the player's performance and returns the player's performance
- generate: void
  - This function generates a csv folder and list the performance / overall score of that individual student

For the *Report Manager* subsystem, we chose a Call-and-Return architecture centered around the *Report Manager* program and with one subroutine *Generate Report*. In this subsystem, the *Report Manager* handles the control and calls *Generate Report*.

When the teacher for example wants to understand how the player did for the assignment in specifics, the teacher can choose to generate a report. This will call the *Report Manager* and the *Report Manager* will call *Generate Report*. So the report will consist of two main components, the overall score of the student for that assignment and the specific summary of the students performance (eg. individual answers to individual questions).

Generate Report will call a sub-subsystem to pull out the student's score from the database and the score will be returned to the *Generate Report*. *Generate Report* can also call another sub-subsystem to fetch the performance of the student for that assignment and the performance will be returned to the *Generate Report*. *Generate Report* will then compile the information and return it back to the *Report Manager* as for example a csv file.

Subsystem Interface Design (Login/Register)



Login/Register Interface
This interface supports the registration and logging in of users.

Functions:

- login(String username, String password): Boolean
    - This function checks if the user has an account with the specified username and password. If yes, returns True. Else, returns False.

- register(String username, String password): Boolean
    - This function checks if the username has already been registered by another user. If not, create a record with the specified username and password in our database and return True. Else, if the username has already been used, return False.

- forgetPassword(String username): String

    - This function will check if the username is present in the database. If yes, send a link to the username's associated email address for the user to reset his/her password.

For the Login/Register subsystem, we chose a Call-and-Return architecture centered around the Database Manager program. In this subsystem, the Database Manager is the main program/subroutine as it acts as the bridge between the UIManager and the Google Firebase. The Database Manager receives information and control from the UI Manager and passes the information on to the Google Firebase. This exchange of information flows in the opposite way as well, as the Database Manager retrieves information from the Google Firebase and reverts back to UI Manager along with the control for updating of display.

The Database Manager will determine whether a player is able to login or register an account based on the information provided by the UI Manager and checking the information against the Google Firebase.

The UI Manager handles the input of the player, sends and receives information to and from the Database Manager. In addition, the UI Manager displays the updated information upon receival from the Database Manager.

The Google Firebase acts as a storage medium to hold all the login information of the users. For a player to register, the information must be cross-checked against the Google Firebase to ensure the same particulars had not been used. For a player to login, the information

must be cross-checked against the Google Firebase as well for access into the user's account.