# Subsystem Interface Design

## for

## Edumon

**Version 1.1 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)**

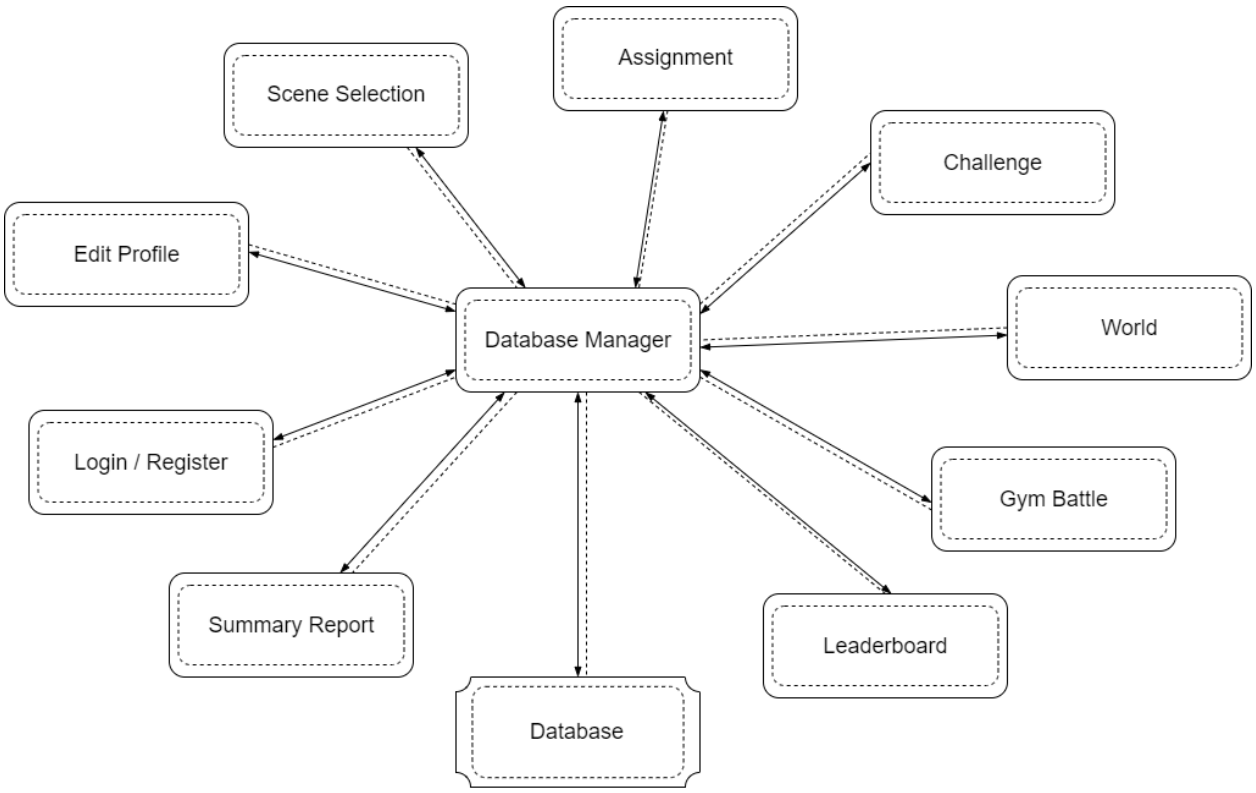
**14/11/2021**

# Table of Contents

# Revision History

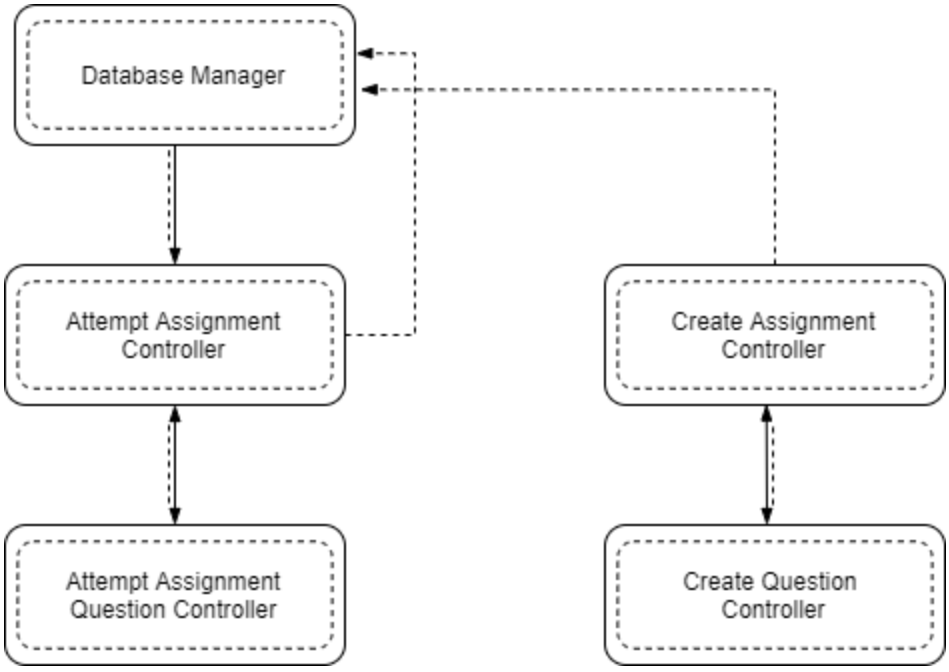
| Name | Date | Reason for change | Version |
|---|---|---|---|
| Everyone | 13/09/2021 | Initial draft | 0.1 |
| Everyone | 17/10/2021 | For submission | 1.0 |
| Jing Hong, Bryan, Shauna | 14/11/2021 | For final submission | 1.1 |

# Subsystem Architectures




The diagram above depicts the overall subsystem that makes up our game. The whole system implements a Call-and-Return architecture. Each individual subsystem has their own subsystems as well, mainly different managers and controllers to enable the flow of the game.
Database Manager manages the different tables of different subsystems, when needed, the subsystem can either POST or GET data from their own respective tables or even from other tables as well.

# Subsystem Interface Design (Assignment)




**Assignment Interface**
This interface supports the use cases of setting assignments as well as completing assignments.

**Functions:**

- Submit_Assignment(string dd, string MM, string yyyy, string HH, string mm, string ss): IEnumerator
    - This function allows teachers to submit the assignments they have created to the database

- Create_Question(string _question, string _option1, string _option2, string _option3, string _option4, string _option5, string _answer): IEnumerator
    - This function allows teachers to create the questions in the assignments.

- Retrieve_Assignment_Data(string _assignmentId): IEnumerator
    - This function retrieves the assignment from the database.

- Display_Question(): void
    - This function displays the questions of the assignment.

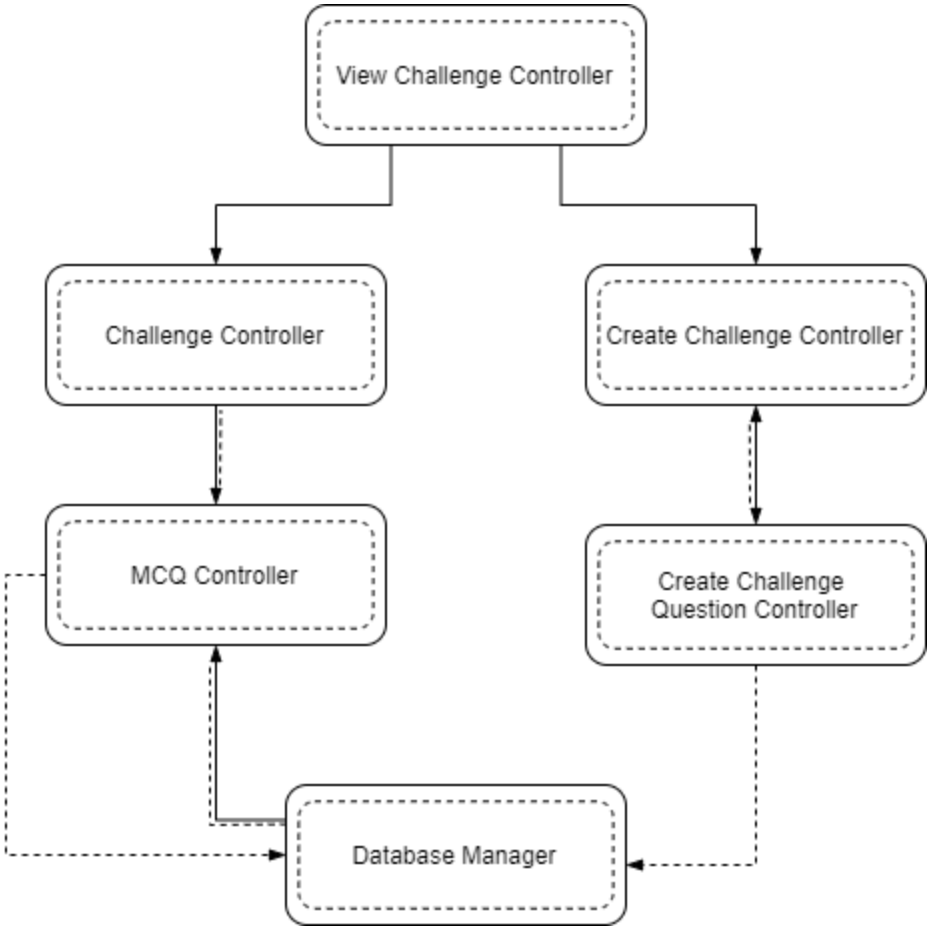
- Submit_Assignment(): IEnumerator
    - This function submits the assignments to the database after students have completed them.

For the Assignment subsystem, we decided to implement a Call-and-Return architecture.

When setting assignments, the Create Assignment Controller allows teachers to set the amount of questions they desire in their assignments. Afterwards, the control and the question number input is passed over to the Create Question Controller, where the Create_Question function is used to create questions for the assignment. After all questions have been created, the questions and controls are passed back to the Create Assignment Controller and the Submit_Assignment function is used to post the assignment into the database.

When completing assignments, the Attempt Assignment Controller will get the desired assignment through the Retrieve_Assignment_Data function. The Attempt Assignment Controller will then pass its control and the assignment over to the Attempt Assignment Question Controller, where it uses the Display Question function to display the questions. When the students are ready to submit the assignments, the Submit_Assignment function is called, and the Attempt Assignment Controller posts the students' answer onto the database.

# Subsystem Interface Design (Challenge)




**Assignment Interface**
This interface supports the use cases of creating challenges as well as completing challenges.

**Functions:**

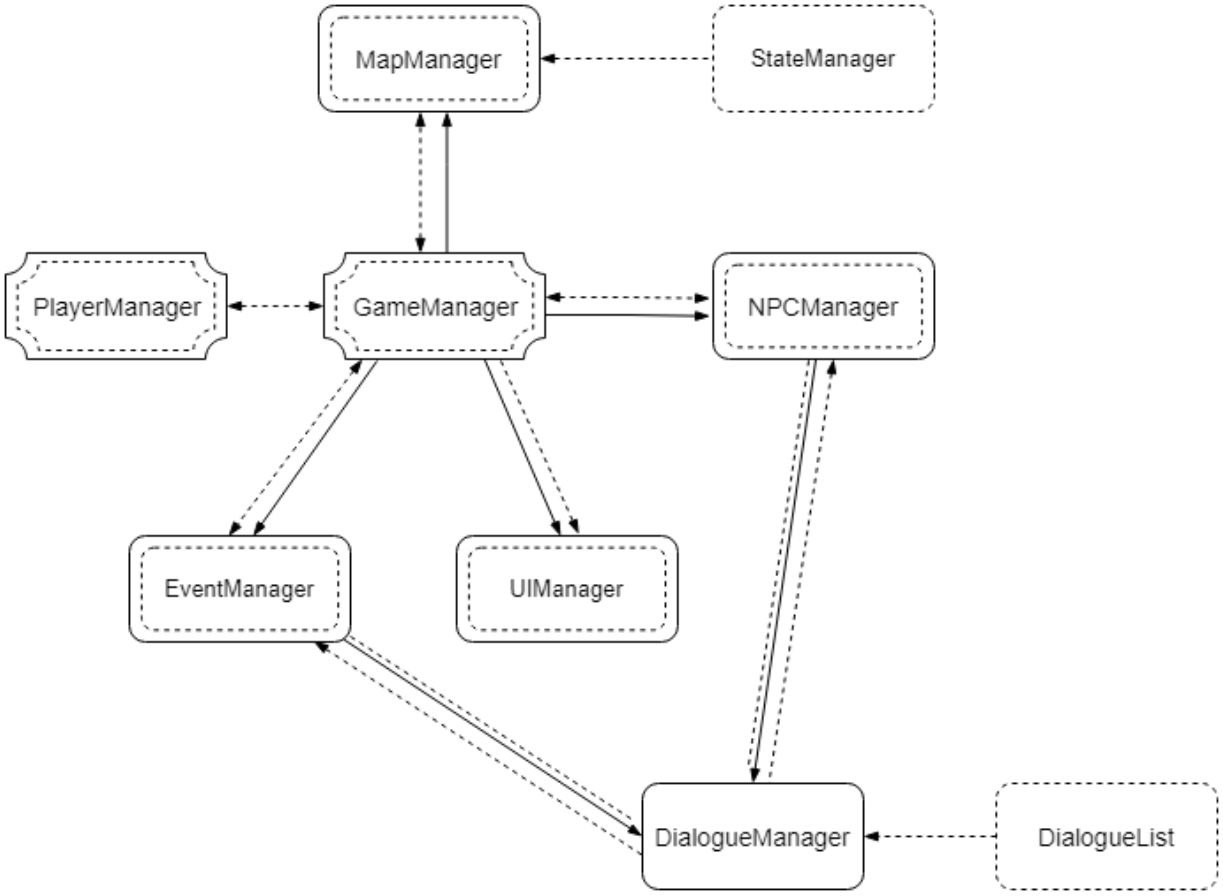
- GetChallengeRequest(string uri): IEnumerator
    - This function allows students to get the list of challenges available
- Create_Question(string _question, string _option1, string _option2, string _option3, string _option4, string _option5, string _answer): IEnumerator
    - This function allows students to create the questions in the challenges.
- DisplayQuestionFromId(string questionId): IEnumerator
    - This function displays the question of a challenge.
- SubmitMCQ(): IEnumerator
    - This function submits the challenge answers provided by students.
- Submit_Challenge(string opponentEmail): IEnumerator
    - This function submits the challenge created by students to the database.

For the Challenge subsystem, we decided to implement a Call-and-Return architecture.

When creating challenges, the View Challenge Controller will pass its control over to the Create Challenge Controller. The students input the number of questions they want in the challenge and the Create Challenge Controller will pass this information and control to the Create Challenge Question Controller. The Create_Question function is then used to create questions for the challenge. After all questions have been created, the questions and controls are passed back to the Create Challenge Controller and the Submit_Challenge function is used to post the challenge into the database.

When completing challenges, the View Challenge Controller will pass its control over to the Challenge Controller and it will display the available challenges using the GetChallengeRequest function. After selecting which challenge they would like to do, the Challenge Controller will pass over the control and the challenge ID to the MCQ Controller, where the questions will be displayed using the DisplayQuestionFromId function. Once the challenge is completed, the answers will be submitted to the database via the SubmitMCQ() function.

# 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:**

- 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.

- sceneManger.LoadScene(string): void
    - This function handles the transitions between maps that are within Unity's own build settings.

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. As only students are allowed into the game, Player Manager will restrict access for teachers.

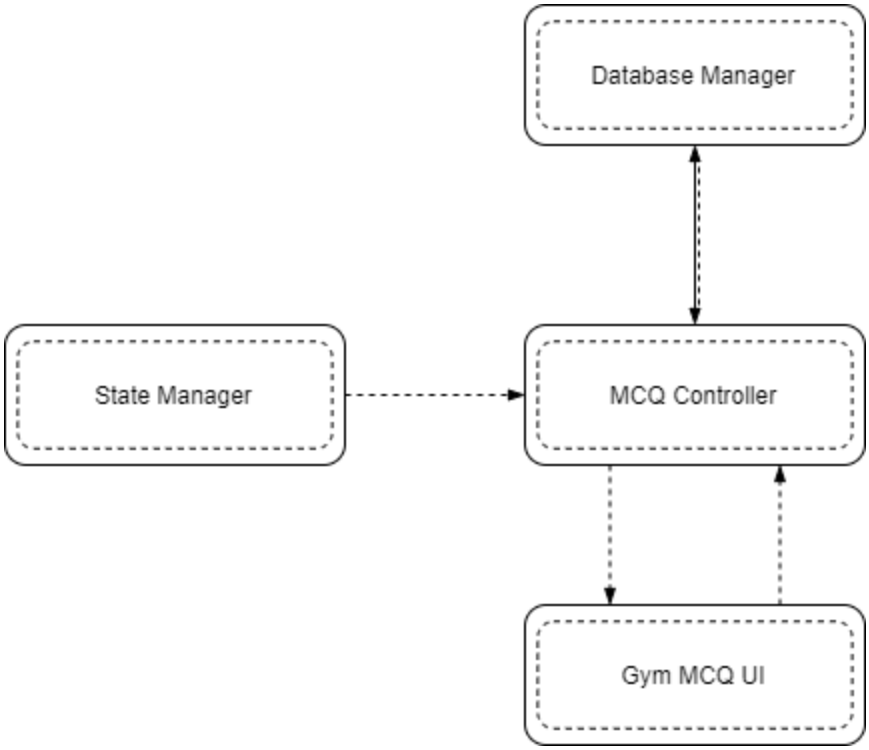
The number of Non-playable characters (NPCs) are fixed and are tasked as interaction points. The outcome of the interaction is decided by NPCManager. The NPCManager will then decide on which NPCs to spawn, and send this information to DialogueManager which will gain control and obtain the dialogue for 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.

The game is also designed to unlock features as the player progresses into the game and it is assumed that the player's knowledge on SDLC has improved as they progress. MapManager will read data stored in StateManager, specifically StateManager.user.unlocked_map, to see which maps are locked and which maps are unlocked for the player.

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 Battle)




**Gym Battle**

This interface supports the unlocking of other maps and improving individual personal knowledge on SDLC by allowing players to attempt gym questions.

**Note** State Manager ****

The State Manager manages the identity of individual gyms as different gyms have different question sets.

**Functions:**

- LoadQuestionsFromQuestionListCurrentIndex(): void
    - This function gets the respective set of questions for the respective gym from the database.
- SubmitMCQ(): void IEnumerator
    - This function submits the attempted MCQ to the database.
- CheckIfPass(string userEmail): void
    - This function checks if the player passed the gym quiz.
- UpdateStateManagerUser(string userEmail): void
    - This function stores the user's updated unlocked_map array in the database into StateManager.user.unlocked_map.
- Start(): void
    - This function calls on start() of scene.

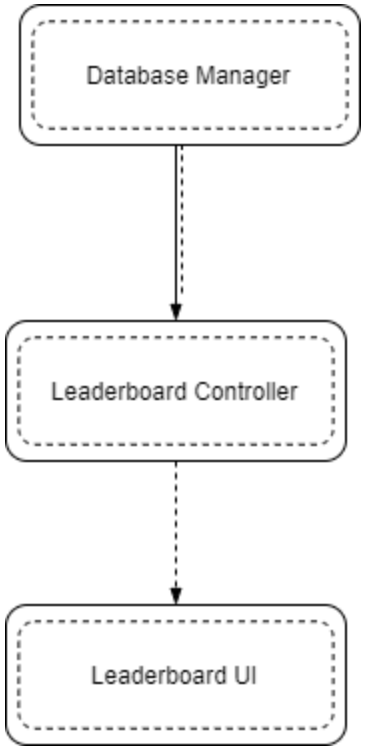
For the Gym Battle subsystem, we chose to implement a Call-and-Return architecture.

Upon entering the Gym Battle Scene, start() will be invoked to get the identity of the gym from the State Manager. The then identified identity will be passed on to the MCQ Controller which will then be used to get the respective question set of respective gyms from the database. Finally the data will be passed to Gym MCQ UI so that users can view the questions and attempt them accordingly.

The then attempted MCQs will be passed back to the MCQ Controller which will then be passed back to the database for ‘marking’.

The score of the player will then be retrieved from the database to check if they passed the gym quiz by calling CheckIfPass(userEmail) function and passing in this player’s email as the argument. If the player passed the quiz, UpdateStateManagerUser(userEmail) is called, passing in this player’s email as the argument, to update StateManager.user.unlocked_map to include the next map since it is now unlocked for the user.

# Subsystem Interface Design (Leaderboard)




**Leaderboard Interface**
This interface supports the update and viewing of the leaderboard.

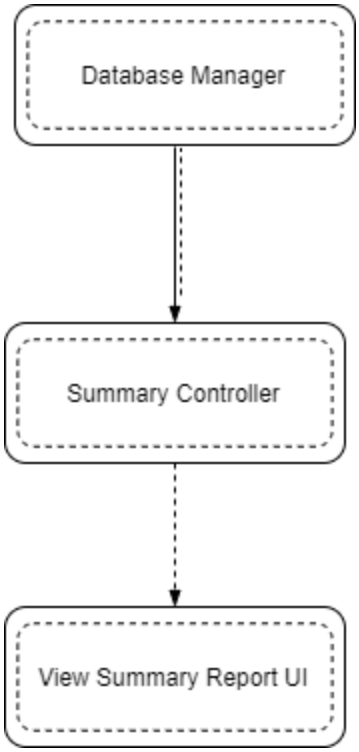
**Functions:**

- GetAccountRequest(string uri): IEnumerator
    - This function gets of the existing student accounts in the database

- GetAttemptRequest(string uri): IEnumerator
    - This function gets all of the attempts on quizzes, challenges and assignments made by students, tallies their scores, and sorts them in descending order.

For the Leaderboard subsystem, we chose a Batch Sequential Processing architecture. The Leaderboard Controller gets all of the existing student accounts in the database using the GetAccountRequest function. For each student account, it gets all of the attempts made using the GetAttemptRequest function. The function tallies all the points gained by each student and sorts the students based on their total points in descending order. This information is then passed to the Leaderboard UI, which will display the leaderboard information.

# Subsystem Interface Design (Summary Report)



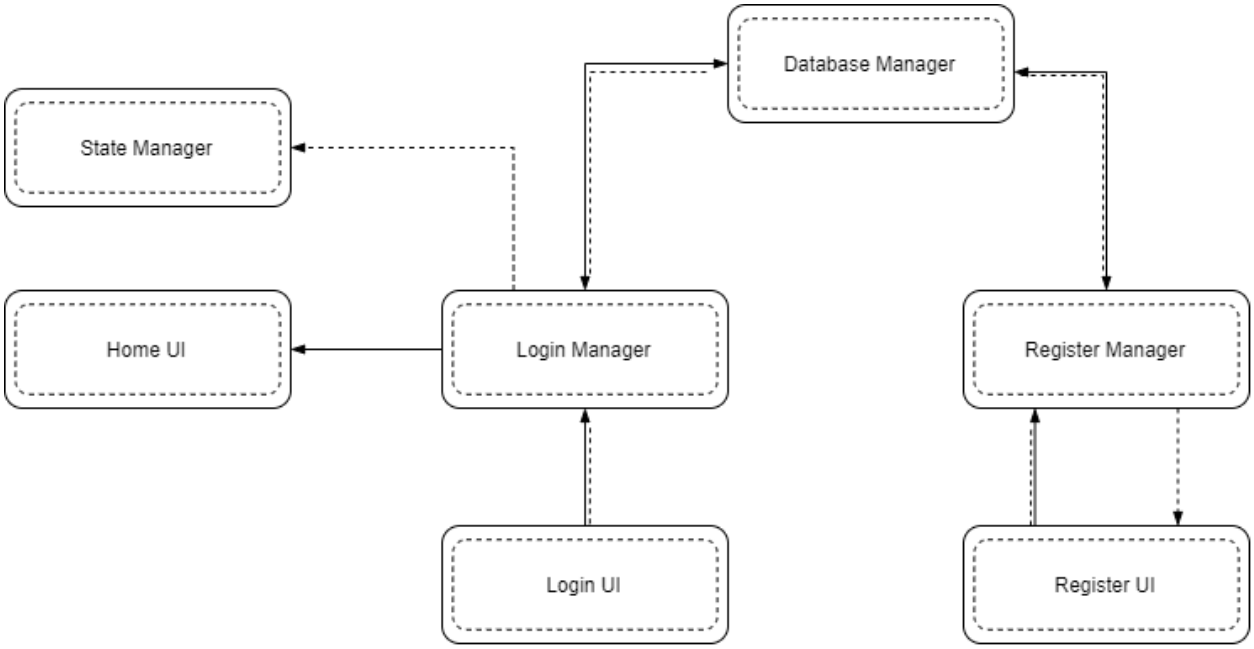

**Report Interface**
This interface supports the generation of summary reports, which includes the students' score for each assignment, challenge and gym quiz.

**Functions:**

- GetEmail(string username): IEnumerator
  - This function gets the email of the student from its username.
- GetAllAssignments(): IEnumerator
  - This function gets all existing assignments in the database.
- GetAllChallenges(): IEnumerator
  - This function gets all existing challenges in the database.
- GetReport( ): IEnumerator
  - This function generates the report, which includes the students' score for each assignment, challenge and gym quiz.

For the Leaderboard subsystem, we chose a Batch Sequential Processing architecture. The Summary Controller gets the desired student email in the database using the GetEmail function. It then gets the assignment and challenges that the student attempted using the GetAllAssignments and GetAllChallenges function. The controller then passes this information to the View Summary Report UI to be displayed.

# Subsystem Interface Design (Login/Register)




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

**Functions:**

- Login(string _url, string _email, string_password): IEnumerator
    - This function gets the user input of email and password and authenticates the login credentials with the database.

- Register(string _url, string _username, string _studentId, string _email, string _password, string _rePassword): IEnumerator
    - This function gets the user inputs required to create a new account and validates it with the database. If the inputs are valid, a new account is created.

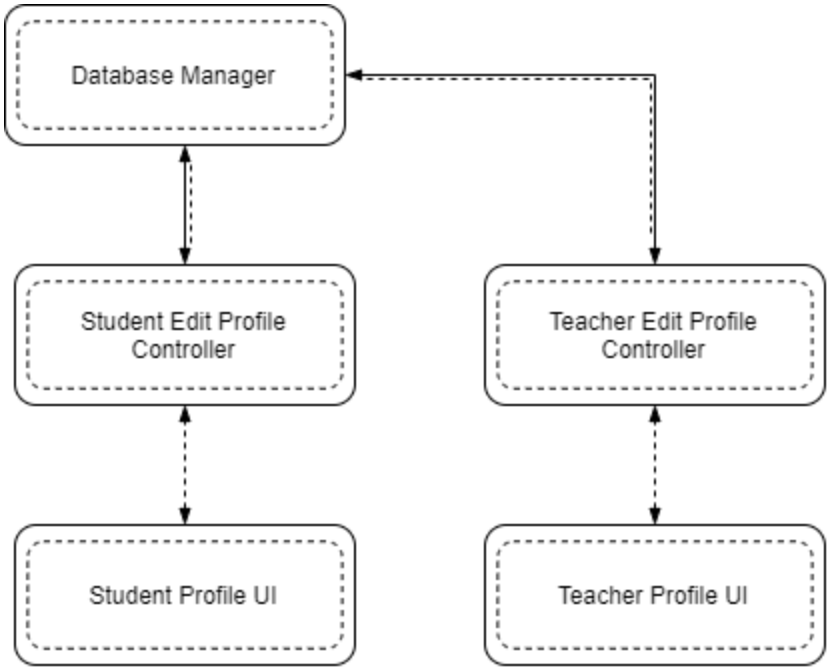
For the Login/Register subsystem, we chose a Call-and-Return architecture centered around the LoginManager and RegisterController programs.

For registration, the RegisterController gets the user inputs from the Register UI. The RegisterController uses the Register function to check with the database to see if those inputs are valid. If they are valid, a new account will be created and posted into the database. The database then returns control and data to the RegisterController and it passes data to the Register UI to display that a new account has been created.

For logging in, the LoginManager gets the user inputs from the Login UI. The LoginManager uses the Login function to check with the database to see any account matches with the inputs. If inputs are valid, the user will be logged in and the database returns control and data to the LoginManager. The LoginManager then redirects users to their respective Home

UIs and passes the user’s data to the State Manager.

# Subsystem Interface Design (Edit Profile)




**Edit Profile Interface**
This interface supports the editing of profiles by students and teachers.

**Functions:**

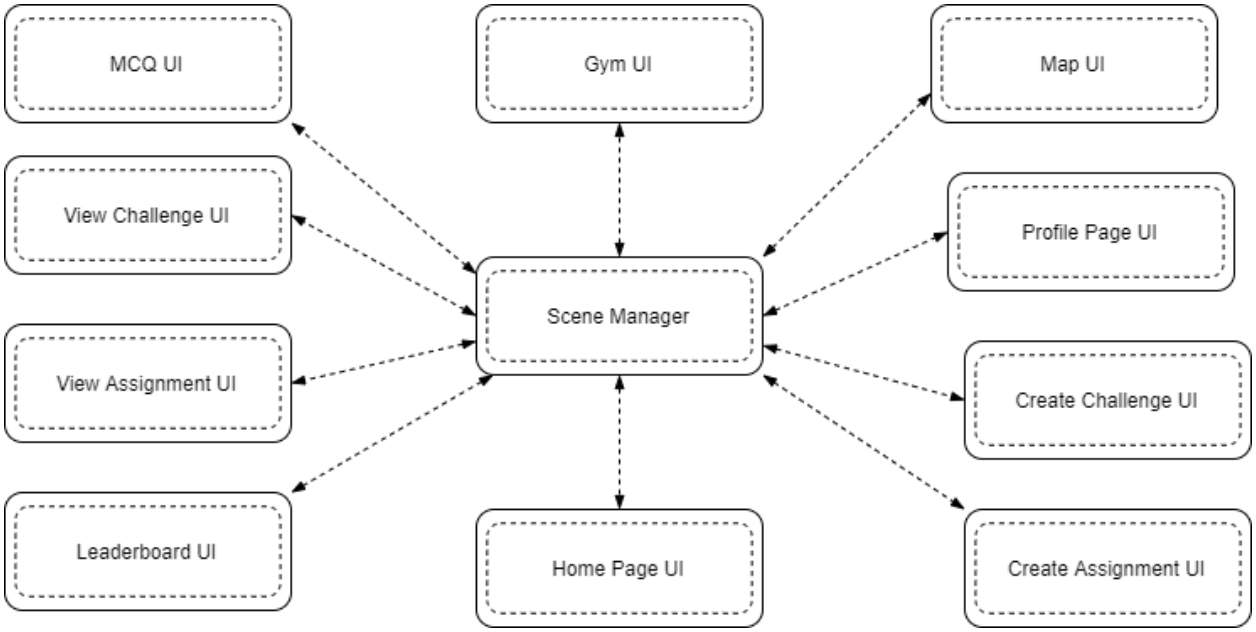
- EditTeacherProfile(string _username, string _email, string _password, string _rePassword): IEnumerator
    - This function gets the teacher's input of username, email, password and password again to change the profile credentials.

- EditStudentProfile(string _username, string _studentId, string _email, string _password, string _rePassword): IEnumerator
    - This function gets the student's input of username, student ID, email, password and password again to change the profile credentials.

For the Edit Profile subsystem, we chose a Call-and-Return architecture centered around the Student Edit Profile Controller and Teacher Edit Profile Controller programs.

Both students and teachers input the changes they want to make in their respective profile user interfaces. The changes are passed to the respective controllers and EditTeacherProfile and EditStudentProfile functions are executed. The functions check if the changes are valid, before posting the data to the database. The respective edit profile controllers then send a message to the UI to display that the changes have been made.

# Subsystem Interface Design (Scene Selection)




**Scene Selection Interface**
This interface supports the selection and displaying of different scenes.

**Functions:**

- LoadScene(string sceneName): unity
    - This function loads the desired scene.

For the Scene Selection subsystem, we chose a Call-and-Return architecture centered around the Scene Manager program.

Users pick their desired scenes to load from the current scene that they are in. The Scene Manager takes their input and loads the desired scene using the LoadScene function.