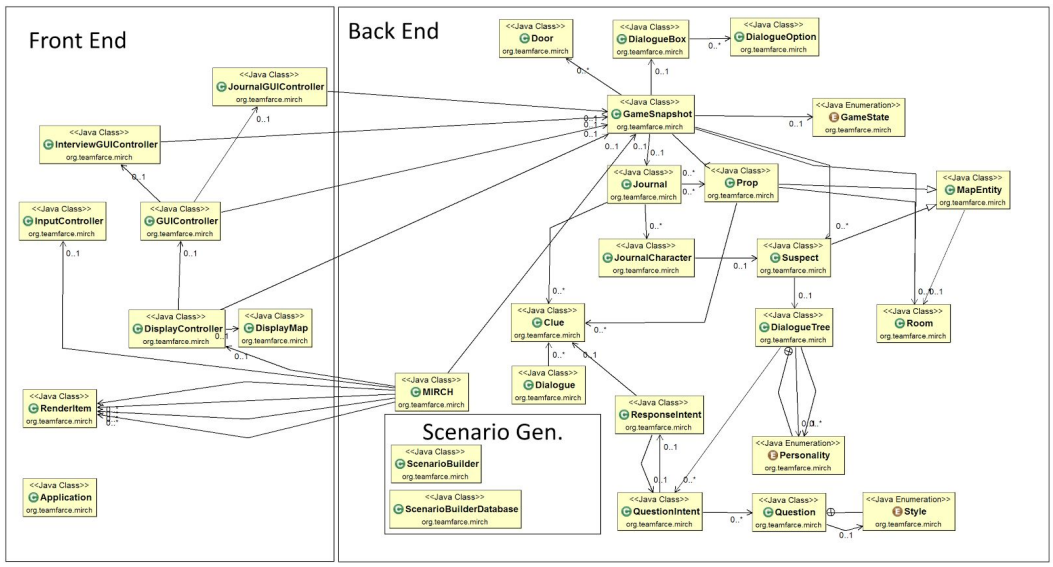
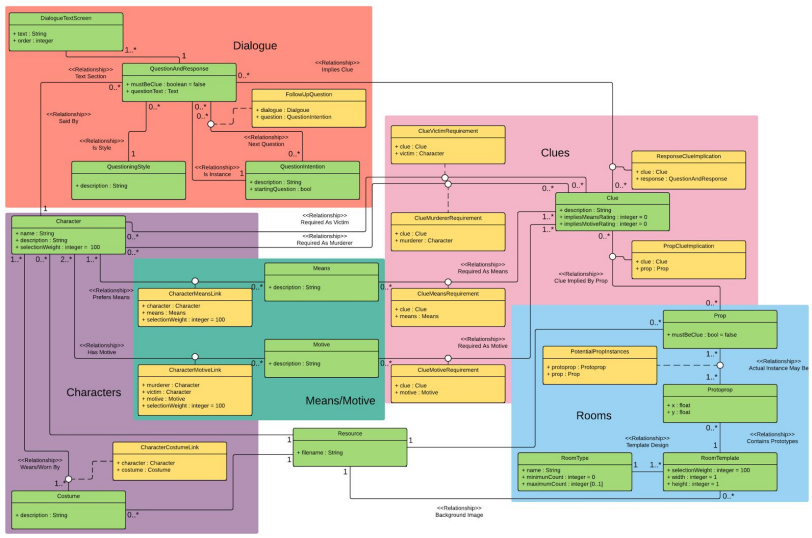
Architecture Report

We constructed an architecture diagram to represent the concrete architecture of the project in UML. The code is split into a two part modularised structure, consisting of the front end and the back end - these are described in more detail below. The team felt that UML was appropriate as the language used to design the architecture as it allows for the visualisation of the complete program with an increased level of detail, improving its traceability and easing our understanding of the connectivity of different modules. By following this model of split architecture design the programming team were able to parallelize their development processes, ensuring a quicker achievement of our end goals. *(Due to the large size of the diagrams they may be hard to view in this document, however full scale diagrams are available to view from the website.)*

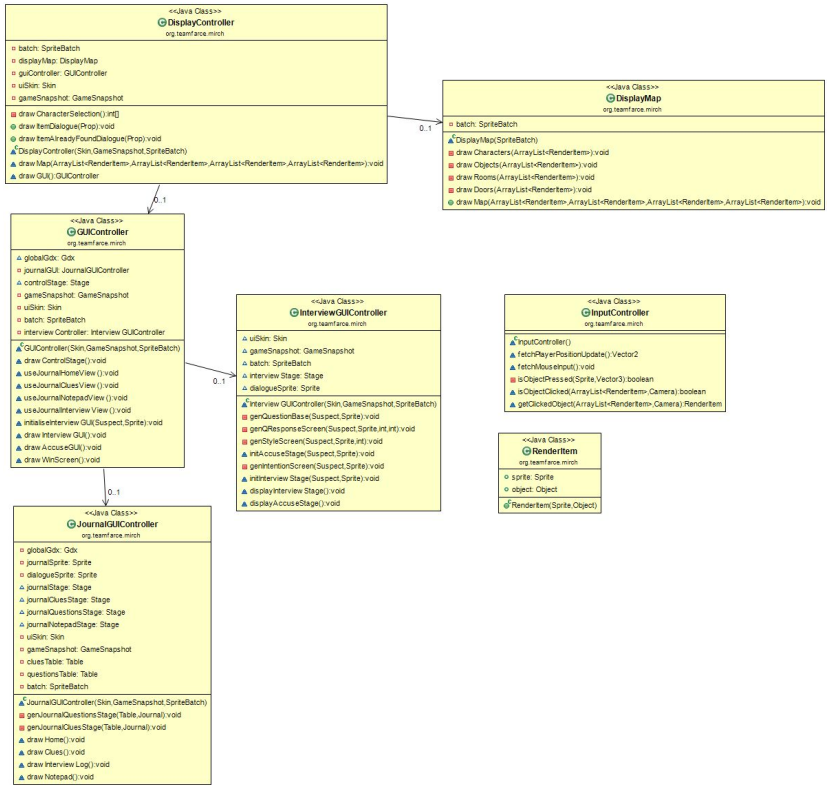
Overall UML Diagram: Link at the end of the document



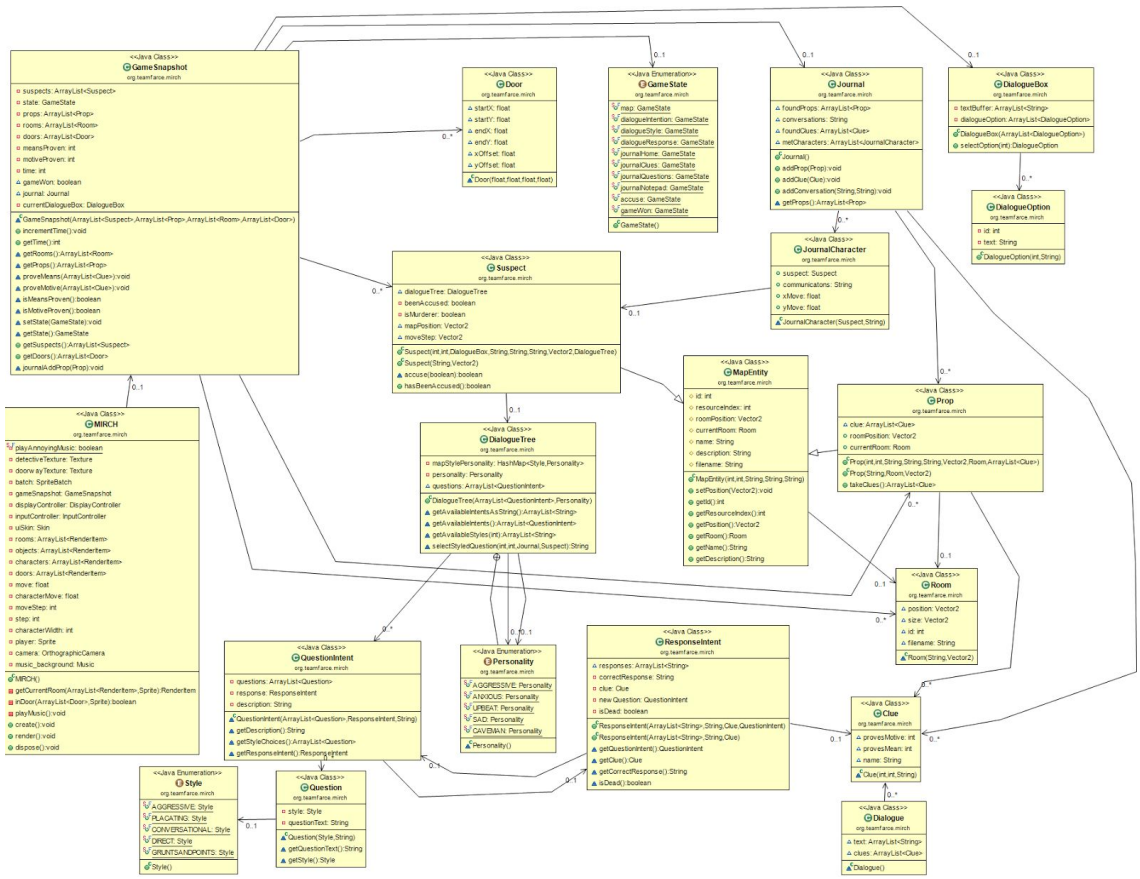
Scenario Builder Diagram: Link at the end of the document



Detailed Front End Diagram: Link at the end of the document



Detailed Back End Diagram: Link at the end of the document



Justification of Concrete Architecture

Development from Abstract Architecture to Concrete Architecture

Our concrete architecture has been built to the same ideals as our Abstract Architecture, by simply extending functional blocks to form correctly modularized connected Classes that still belong in two distinct areas, and can hence be developed in parallel.

When designing our concrete architecture, we first took the base blocks of our abstract architecture, and then focused on areas where the data and functionality inside each abstract block could be encapsulated further - building new classes and enumerations for each stage of encapsulation. Through this method, we were able to ensure that we continued to follow our Abstract Architecture whilst generating a strong Concrete Architecture to implement our program with.

Systematic Justification of Concrete Architecture

The Front End

The front end performs the following processes:

- The DisplayController class displays the GUI and the map, along with all the characters and props.
- The InputController class collects inputs from the user, converts them to meaningful commands and passes these to the backend.

Very little has changed in the front end architecture design, as we have maintained the Abstract Designs separation of DisplayController and InputController through the use of two classes. The task of game rendering and input translating from the abstract architecture for the front end was factored into multiple classes (DisplayController, GUIController, InputController, etc.) for the purpose of data encapsulation to allow different team members to work on the game rendering unit concurrently.

The Back End

The back end performs the following processes:

- Game Initialisation
    - Game Initialisation is carried out in the Create function of the MIRCH class. This function calls the ScenarioGenerator to generate a GameSnapshot, and then begins the GameLogic loop.
- Game Logic
    - Game Logic is controlled in the MIRCH classes Render function, this function runs in a continuous loop and has multiple purposes:
        - Collect inputs from the InputController and processes these inputs

        - Collect inputs from the InputController and processes these inputs
        - Calculates player movements
        - Calculates character movements
        - Calculates characters final score
        - Updates the game state where required
        - Outputs the necessary screen to the display through a call to the DisplayController function
- GameState Storage
    - The GameState Enum provides a simple method for storing the current state of the game. This enables the gameLogic loop to carry out relevant calculations depending on the current gameState.
    - The GameSnapshot class stores all integral data about the program, and provides means to update and access this data - this data includes:
        - Suspect Objects
        - Prop Objects
        - Clue Objects
        - Room Objects
    - Each data Object stored in the GameSnapshot class (i.e. Suspect Objects) also includes a number of functions relevant to that Class, for instance the Suspect object stores the DialogueTree of that Suspect. This allows for strong data encapsulation at every level of the program, making the program easier to implement, maintain and understand.
- DialogueTrees
    - The Dialogue Tree classes provide an intelligent method for storage and access of an individual Suspects available Dialogue, allowing the player to question the subject and receive different responses depending on both the style of their questioning and the Suspects character traits.
- ScenarioGeneration
    - The ScenarioGeneration class makes use of a ScenarioGenerationDatabase class to load a sql-lite database which stores all generation data into memory, and then makes use of this data to generate a new random scenario each time the game is run. This new scenario is stored in a GameSnapshot object, which is returned to the Create function in the MIRCH class for use. Managing the scenario generation in such a way allows for the separation of abstract game items such as Props and Rooms from the Class structure of the program, making it easy to build a detailed range of scenarios using the database structure, without having to worry about the programmatic implementation. This method gives twofold advantages:
        - A range of different databases can easily be created with different props, rooms and characters, allowing the scenarios of the game to be easily extensible.
            - For instance, in order to generate a OpenDay version of the game, only a simpler database file must be used - meeting User Requirement 15.

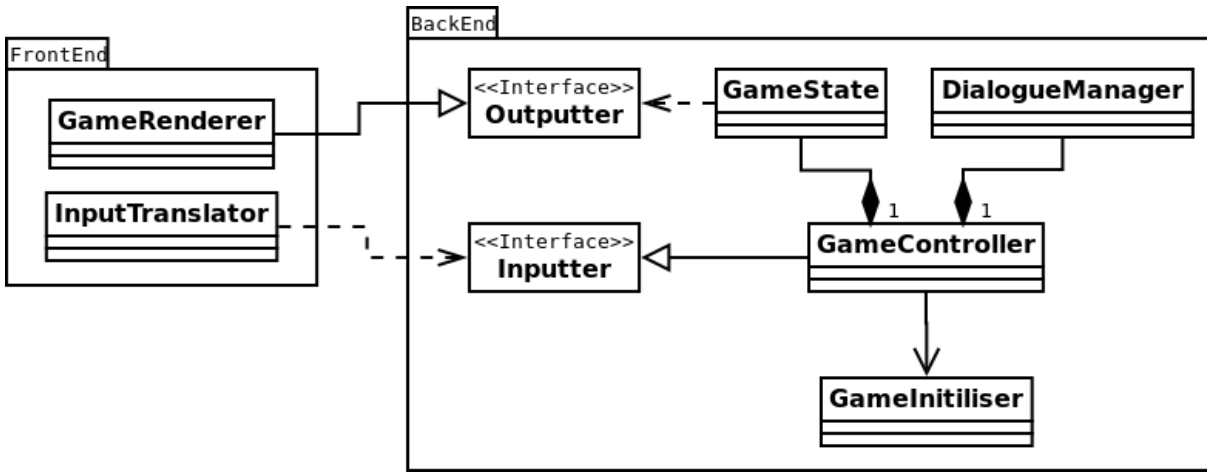
- Some team-members are able to focus on developing scenarios for generation, whilst others can focus on the generation programming, allowing for parallelisation of the development process.

Collecting the game logic in a separate back end unit allows the potential implementation of multiple front ends making requirements for visual features independent of the requirements for the game logic (core requirements), therefore making any future requirement changes easier to accommodate.

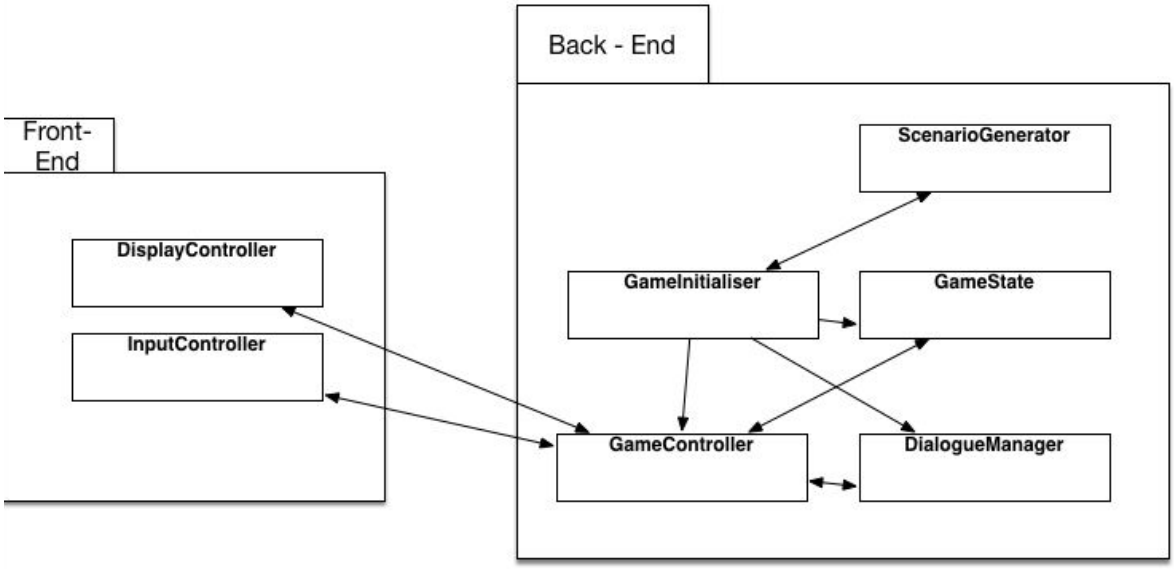
Necessary Changes to the Original Abstract Architecture and Associated Justification

During the development of our program from design to implementation, we adjusted our initial abstract architecture to remove un-necessary complications and therefore simplify the development process.

Initial Abstract Architecture



Updated Abstract Architecture



As you can see from the updated architecture diagram, the basic modularised structure has remained the same however as we are using the same language for both modules, the input and output interfaces were deemed unnecessary, and this functionality was therefore merged into the GameController unit. By maintaining a two-part modularized structure, we were still able to make use of our planned parallel development process - ensuring that two teams were able to efficiently implement and test desired functionality in the two modules separately. As well as benefiting our programming team, the maintaining of this split functionality will also benefit any future team that may work on our code; an additional benefit of this architecture is that we are able to implement and test multiple front ends with the same back end - allowing for different rendering options and control schemes to be simply implemented - making our program easily extensible and adjustable depending on any change of requirements.

Updated Requirements Link:
https://teamfarce.github.io/MIRCH/designdocs/submit_2/updated/Req2.pdf
Abstract Architecture Link:
https://teamfarce.github.io/MIRCH/designdocs/submit_1/Arch1.pdf
Class Diagram Links:
https://teamfarce.github.io/MIRCH/designdocs/submit_2/new/Class_diagram.png
https://teamfarce.github.io/MIRCH/designdocs/submit_2/new/back_end.png
https://teamfarce.github.io/MIRCH/designdocs/submit_2/new/Front_end.png
https://teamfarce.github.io/MIRCH/designdocs/submit_2/new/DataStoreDiagram.png