System Architecture

Introduction to the System Architecture

We plan to use a two part structure for our software architecture, with a front-end used to receive user inputs and display graphical outputs to the users display, and a back-end that processes player inputs and provides all calculations and procedural changes for the game. A modularised approach will then be taken inside each of these sections to ease variable management and organise our code - speeding development. Justification of this structure can be found in the ‘Justification of System Architecture’ section below.

The back-end of the game will work in a ‘step’ based fashion, with each step moving the game forward - moving AI players from room to room, displaying questions for interrogations/generating responses and updating the user's position and progress in the game. The front-end of the game will appear to work in real-time, randomly moving AI characters inside rooms, however will only step the game forwards (through the back-end) only when the player makes a move (i.e. moves room or interrogates an AI character etc.) This allows for a pseudo-real time effect for the game, allowing the user to believe the game is working in real time, whilst simplifying the programming of the game by actually creating a step-based simulation.

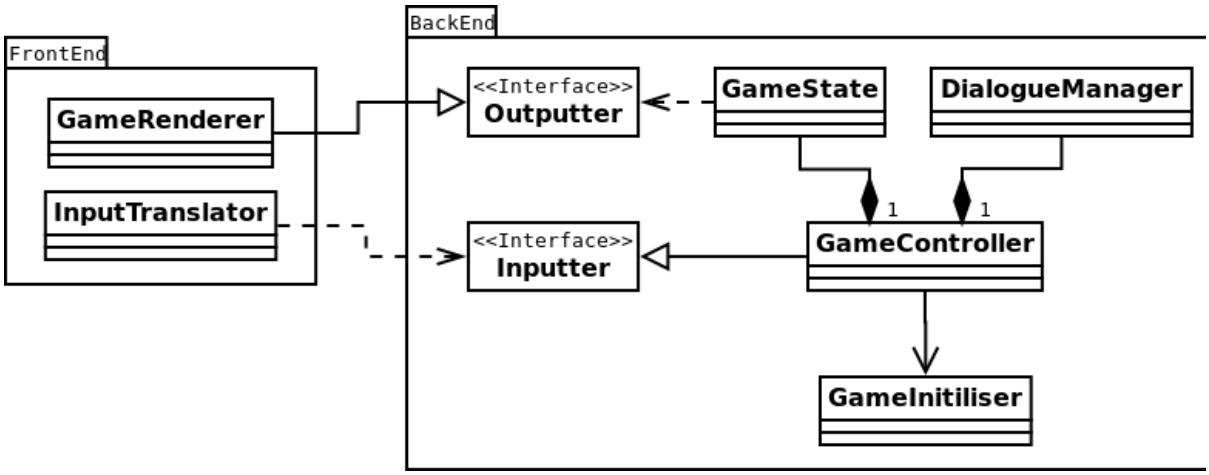
The Front End

The front end will collect inputs from the user, convert these to a sensible format and pass these inputs to the back end. It will also collect outputs generated by the backend and display these graphically onto the user's display. Finally the front-end will control the initialisation of the back-end, and also its termination on the closing of the visual interface.

The Back End

The back end will be passed information into an inputter, and will send information back via an outputter. It will process player inputs and provide relevant outputs; calculate the game setup including the murderer and clues; provide questions and answers for the interrogation of characters; calculate AI character movements; whether the end-game has been met and finally store details of the players in-game ‘notebook’.

To better understand and visualise the connection between the front-end and back-end, as well as the internal processes carried out by each, we have provided a simplified UML 2 Class Diagram (which excludes details of data passed and functions available); this diagram can be found below.

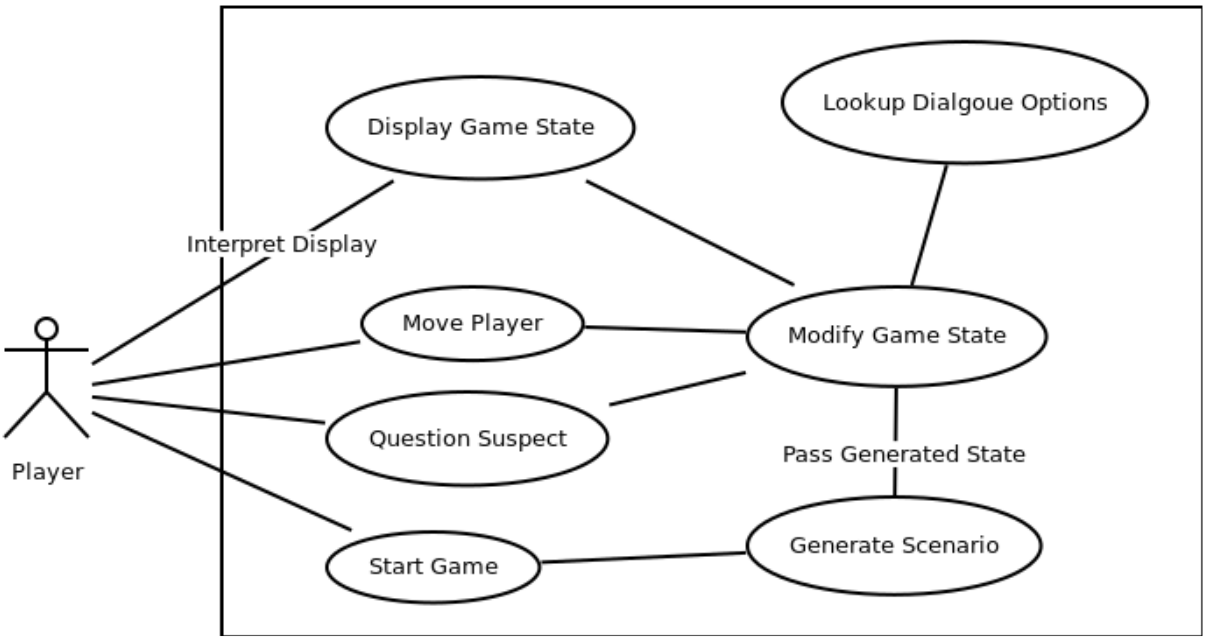


Justification of System Architecture

We have decided to use a split architecture with a modularised approach to build our game. The split architecture will consist of a frontend controlling inputs and drawing the game to the display, and a back-end controlling the game state. There are many advantages to using this two-part structure in our system architecture - as the two sections are structurally separate, they can be written in different programming languages if required and two teams can also work concurrently on the different sections, speeding development. This also means that multiple different front ends can be used with a single back end - for instance during development of the back-end a simple text based front-end could be quickly created to easily test the functionality of the back-end without the back-end developers having to wait for the required functionality to be built into the graphical front-end, once development of the graphical front-end has been completed, this can simply be ‘slotted in’ as replacement to the temporary text-based front end used during development. This ability to easily create different front-ends to work with the same back-end software also means that it will be easy to create different user interfaces for other environments such as computers, mobile phones and web browsers.

Inside each section of this architecture, we will use a modularised approach to our programming. This allows us to easily encapsulate code, allowing different developers to work concurrently on different parts of the program once each Classes interface has been planned out and detailed. This also allows for easier maintenance of our code base, as an modularised program is much easier to modify than a non-modularised program. Finally this approach will allow for the easy re-use and recycling of code in other sections of the program.

To better understand how components will work cohesively throughout the game from the user’s perspective we have also created a use case diagram (see below). This helps us to visualise how components will delegate tasks to other components, and also the components that should be used to accomplish certain tasks.



The front end contains an input translator and a game rendered.

The input translator manages data capture from the user, and converts this into a program readable data type. By modularising this functionality, different capture methods can easily be written for different styles of input device, such as the touch screens of mobile phones and tablets, web pages, and keyboards/mice for desktop applications, helping to ensure that the game is easily reconfigurable in the future if required.

The Game rendered collects data from the output interface of the back-end and manages the drawing of the game to the users display. Different front ends can be written for different styles of display if required, allowing us to easily develop for different devices and screen sizes (such as phones, computers and web browsers) if required. This allows us to easily match the 2D top down user interface outlined in our requirements document.

The back-end controls the game state, processing player inputs and providing procedural changes to the game.

The input manager ensures that all inputs are collected and managed by a single object, making input data easily accessible to all parts of the program. This also allows for easy validation of all input data, helping us to ensure that all input data is correct before use. By ensuring that all data is validated on input, it will be easier for a user to understand how the game works and to use it correctly - hence matching out non-functional requirement that the game must be usable by at least 80% of users without external help and without the manual.

The game controller controls the moving forward of game state. It manages the game and calls and initialises other objects in the back end (such as the game initialiser, and dialogue manager). This central controller is the key to a simple program structure, helping to ensure that all game data is passed logically around the program.

The game initialiser generates a new game, creating a new murderer, AI character suspects and generating and placing all relevant clues across the game map. This encapsulation of game initialisation allows us to easily generate and control different game

play options and styles, hence meeting our requirements to randomly generate rooms, clues and a murderer.

The dialogue manager allows for the interrogation of AI characters, providing questions for the player to ask and responses to these questions. By encapsulating this part of the game, we can easily extend the interrogation functionality - and easily meet our requirements to allow the user to question AI characters.

The game state object is used to store the game state - the player position, positions of AI characters, clue positions and details, room sizing and positions and all other game data. This method of encapsulating all global data in one place allows us to easily access all data required from any location in the program, simplifying the programming of the game.

Finally the output manager provides an external interface to the game state object, allowing the front-end to access all game information required to generate a visualisation of the game. By encapsulating this part of the project, it is possible for us to write separate output managers for different environments. For instance, if creating a web based game we will want the output manager to be accessed through a REST API, whereas if creating a desktop game we will wish it to be function based.