Architecture

Team 5

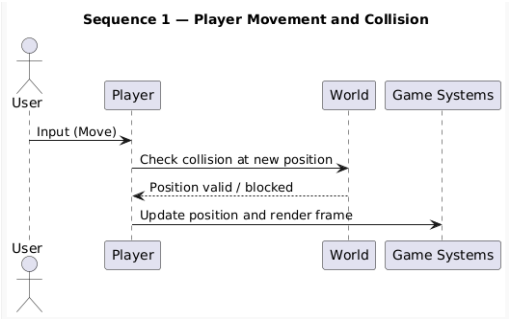
Lizzard Entertainment

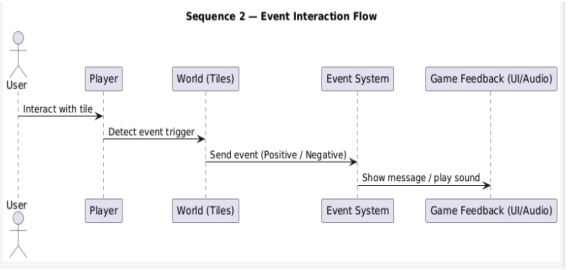
Gregory Binu
Lydia Eaton
DJ Fox
Daniel Maffei
Michael Papafilippou
Nathalie Rizzo

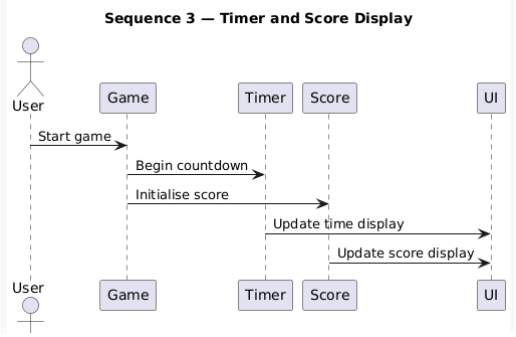
The architecture style we have chosen for this project is an Entity Component System (ECS) style [1]. This is because it is the most commonly used architecture style for games, and it is also very easy to implement thanks to libGDX (the engine being used for the project) coming with a framework called “ashley”, which makes it simple to create and assign modular components and entities. ECS also makes the code more flexible and modular, making it easier to change or add to. There are few dependencies between the components, and behaviour is separated from data.
This architecture works by having three main aspects:

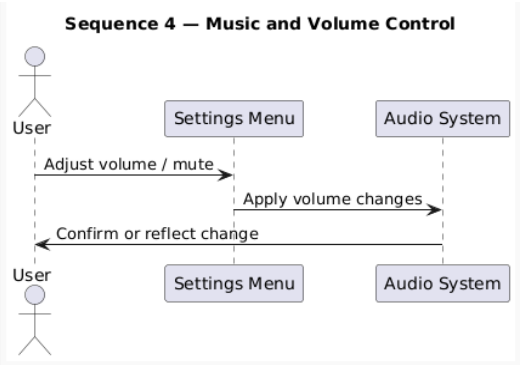
- Entities - game objects such as the Player, Items, Obstacles like the Duck or Central Hall that exist within the world.
- Components - small, reusable data classes (e.g. Position, Velocity, Input, Sprite, Score, Behaviour) which define what data an entity holds.
- Systems - logic modules (eg. InputMovement, Movement, Render, Event) that process all entities containing the relevant components each frame

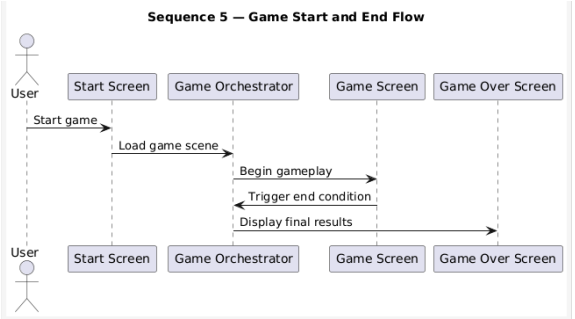
Sequence Diagrams:
To understand how users and subsystems would interact, we first created a set of sequence diagrams. These diagrams show the core gameplay flows before introducing detailed components or classes down the line. This helped us visualise interactions between actors and major game elements early in design.











Sequence 1: Represents the core gameplay loop showing how the user interacts with the player character and how that action moves through the game systems. It shows how movement, collision detection, and rendering interacts per frame.
Sequence 2: Outlines how the player interacts with the tiles that trigger positive, negative, or hidden events.
Sequence 3: Shows how the Timer and Score objects update during gameplay.
Sequence 4: Shows how the users interact with the Settings Menu to adjust or mute game audio.
Sequence 5: Shows the game's lifecycle from starting gameplay to reaching a win or loss state.

Candidate Responsibility Collaborator (CRC) Cards:
CRC Cards list a candidate and define the purpose of each candidate, their role stereotype, their responsibilities, and the other candidate it will interact with. We created these by writing on revision cards and laid them out on a table. This visually mapped out how each candidate would collaborate with others, making it easier to develop the prototype architecture. The process of creating these as a group in one of our meetings forced us to iteratively refine our architecture; we all contributed to creating them then talked about whether all of them were necessary and how they interacted with each other. The cards were clustered together into major components to establish clear boundaries and interactions:

- User Interface (UI) & Menus: Handles all visible menus, screens, and HUD elements.
- Audio: Manages all in-game sound components.
- World: Defines the structure and spatial logic of the game environment.
- Entities: Represents all active or passive objects and characters that exist within the World.
- Event System: Manages special occurrences, triggers, and resulting player notifications.
- Game Logic/Systems: Contains the core rules, mechanics, and various elements of gameplay.
- Game Orchestrator: A delegated controller that advances the main game loop and routes the signal between all other components.

Prototype Component Diagrams made with Plant UML [2]:
https://ksd540.github.io/lizzardentertainment/#prototype-component-diagrams

Using the responsibilities created on the CRC cards, various component diagrams were made and separated to create structures mentioned earlier. This was done so it could be cleanly mapped to the Entity Component System we were using as the behaviours shown in these diagrams were systematic and data-driven.

The High Level Overview showed the main controller with the Game Orchestrator coordinating UI, World, Entities, Systems, Events, Audio, which helped show our desired delegated control style. The UI & Menus diagrams show screen-level components and how the UI talks to the Orchestrator and Audio. The World and Entities diagram shows the separation of the World from Entities and helps show validation takes place based on collision and movement. The Game Logic & Event System diagram groups the various Systems and Events and shows their relations. The final Audio Integration diagram shows

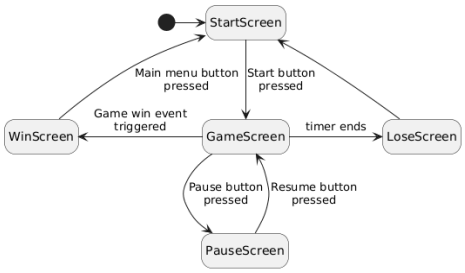
how the music and sound effects were going to be controlled through the events/systems and the game orchestrators.

As development progressed, the implemented code showed that most entities were better represented as tiles or objects, and making the Player the only entity served better for our restrictions and requirements. After multiple changes and iterations, we came up with the following architecture. This simplified the systems into clear modules and more closely matched the code:

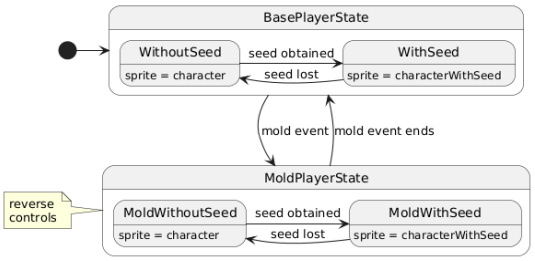
- Orchestrators (Game and Event) to manage control flow and communication.
- Systems (PlayerControl, Movement, Collision, Render).
- World (Map, Tile, Position, Collision) to hold map data.
- Screens/UI for user interaction.
- Other Objects (Score, Timer, Tile Sprites) for relevant gameplay states.

Finalised Behavioural Diagrams [3]**:**

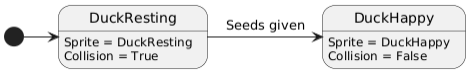
Game screens:



Player object:

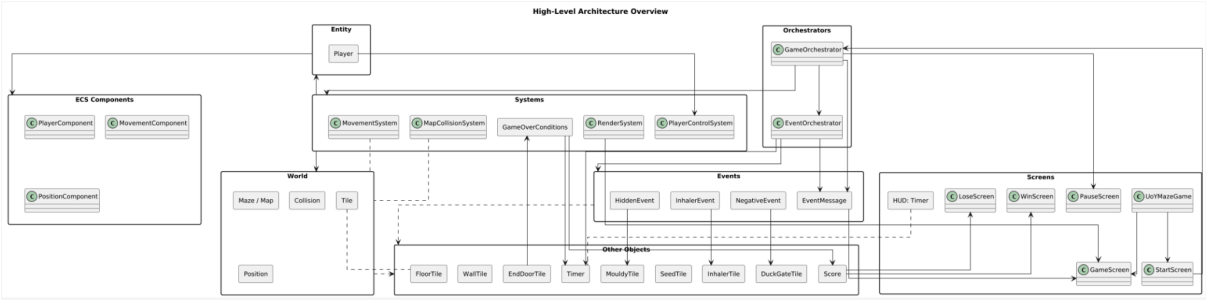


Duck tile:

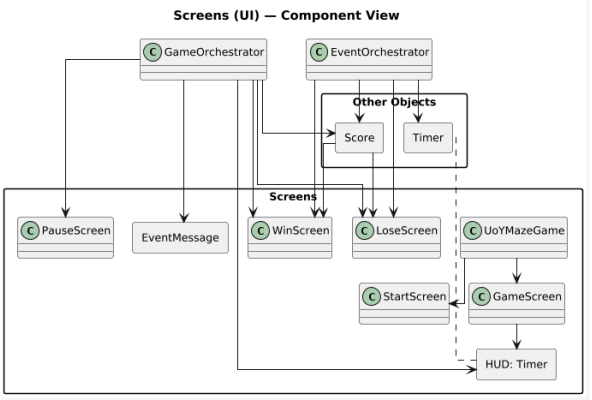


Finalised UML Diagrams [2]**:**

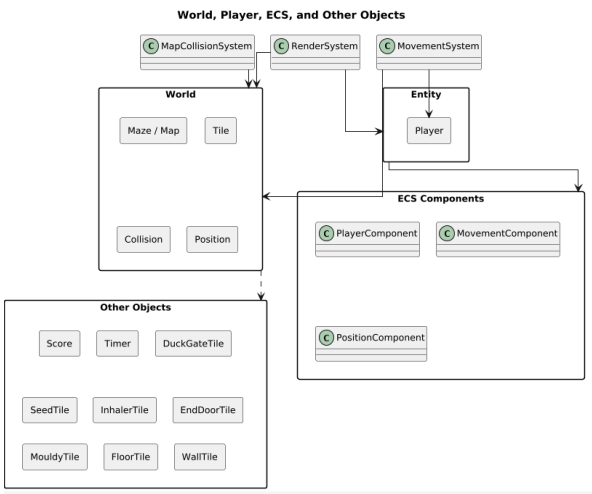
High Level Architecture Overview:



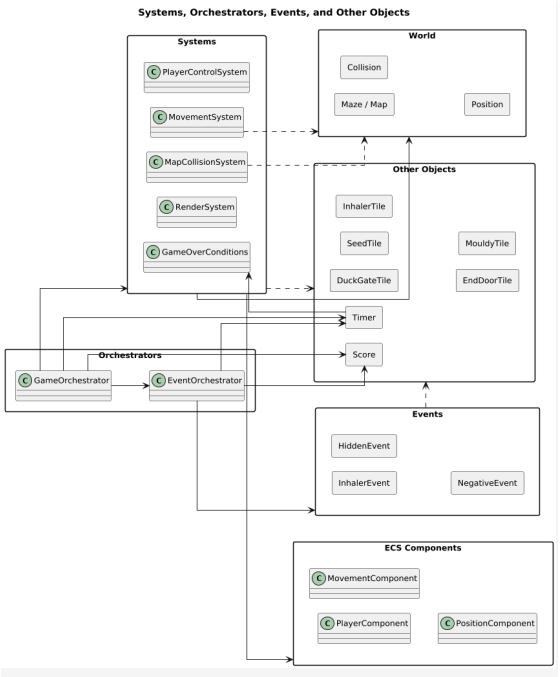
Screens (UI):



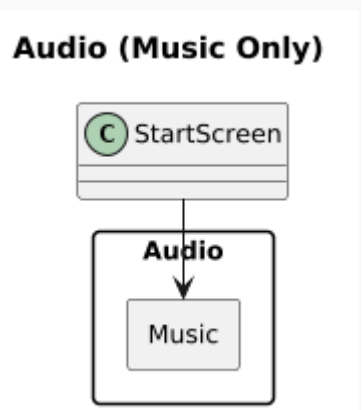
World, Player, ECS, and Other Objects:



Systems, Orchestrators, Events and Other Objects:



Audio:



Between the prototype diagrams and the updated diagrams there were multiple changes. The main changes were:

1. The Player is now the only entity.
2. Settings Screen and Janitor entity were not implemented.
3. EventOrchestrator was added to manage events.
4. The Score was changed to only now show in the Win/LoseScreen.
5. The Score is only calculated using data from the timer.
6. Sound Effects and Volume Control were not implemented.
7. Score and Timer are no longer part of ECS components and rather are modelled as objects.

Changes 1, 3, 4, and 7 were made in collaboration with the implementation team to make the code simpler and easier to maintain. Changes 2, 5, and 6 were made due to time constraints. The diagrams were also greatly simplified to make responsibilities direct and clear [4]. **Image with the methods and functions of the classes is shown here:** https://ksd540.github.io/lizzardentertainment/#methods-and-functions

Requirements Table Mapping:

Appendix:

- Appendix A = High Level Architecture Overview
- Appendix B = Screens (UI)
- Appendix C = World, Player, ECS, Other Objects
- Appendix D = Systems, Orchestrators, Events, Other Objects,
- Appendix E = Audio (Music Only) and UI Control

User Requirements:

Requirement ID	Related Diagram	Explanation
UR_CONTROL_CHARACTER	Appendix D	Shows **PlayerControlSystem to MovementSystem** to Position flow driven by the **GameOrchestrator**, hence implementing keyboard control of the Player.
UR_PAUSE_GAME	Appendix B	Includes **PauseScreen** and linkage to the **GameOrchestrator** state to pause/unpause the gameplay..
UR_SEE_TIMER	Appendix B	Shows **HUD: Timer** which reads the timer object.
UR_FINISH_GAME_SESSION	Appendix D	Shows **GameOverConditions** using **Timer** to end or complete within 5 mins.
UR_UNDERSTAND_OBJECTIVE	Appendix B	UI relays goals via the **StartScreen, GameScreen** and **EventMessage**.
UR_POSITIVE_EVENT	Appendix D	Events are packaged with **InhalerEvent** routed by **EventOrchestrator** to apply beneficial effects.
UR_NEGATIVE_EVENT	Appendix D	**NegativeEvent** processed via **EventOrchestrator**, hindering progress as the player is blocked via a gate..
UR_HIDDEN_EVENT	Appendix D	**HiddenEvent** routed by **EventOrchestrator** and inverts player control..
UR_MUSIC	Appendix E	Background Music loaded upon **StartScreen** activation.

Functional Requirements:

FR_CHARACTER_MOVEMENT	Appendix D	**PlayerControlSystem** converts input to intents and. **MovementSystem** updates Position. **MapCollisionSystem** also validates.
FR_PAUSE	Appendix B	**PauseScreen** and **GameOrchestrator** pauses the game and Timer.
FR_UNPAUSE	Appendix B	Unpauses via the **PauseScreen** returning control to **GameScreen** and **Timer** resumes.
FR_LOSE_GAME	Appendix D	On **EndDoor** tile, **LoseScreen**

		triggered by **GameOverConditions and** ends the game when **Timer** reaches 5 mins without completion.
FR_DISPLAY_OBJECTIVES	Appendix B	**StartScreen/GameScreen** with **EventMessage** displays objectives and scoring rules.
FR_OFFER_EVENT	Appendix D	**EventOrchestrator** runs positive/negative events on interactions, systems apply effects.
FR_HIDDEN_EVENT	Appendix D	**HiddenEvent** triggered from **MouldyTile** interaction and triggered via the **EventOrchestrator**.
FR_REPEAT_EVENT	Appendix D	The **EventOrchestrator** supports re-publishing: events can be re-triggered multiple times.
FR_MUSIC	Appendix E	Background Music loaded upon **StartScreen** activation.

Non-Functional Requirements:

NFR_INTUITIVE_INPUTS	Appendix B	**GameScreen** handles input and pairs with the **PlayerControlSystem** which makes control discoverable and consistent.
NFR_LEGIBLE_TEXT	Appendix B	The Objectives shown via **StartScreen/GameScreen** and **EventMessage** are made with readable UI text.
NFR_OBJECT_DESIGN	Appendix C	Other objects such as the **Door, Seed, Inhaler, EndDoorTile, MouldyTile** etc are all distinguished by sprites.

Unmet Requirements:

Requirement ID	Explanation?
UR_VOLUME FR_VOLUME	Time Constraints
FR_WIN_GAME	Time Constraints

References:

[1] “The Entity-Component-System Design Pattern” *Umlboard.com*, 2021. https://www.umlboard.com/design-patterns/entity-component-system.html
[2] “Component Diagram syntax and features,” *PlantUML.com*. https://plantuml.com/component-diagram
[3] “State Diagram syntax and features,” *PlantUML.com*. https://plantuml.com/state-diagram
[4] Barmpis, K (2025), *Software Architecture*, Department of Computer Science, University of York.