Architecture

Cohort 2 Team 1

Ahmet Abdulhamit

Zoey Ahmed

Tomisin Bankole

Alanah Bell

Sasha Heer

Oscar Meadowcroft

Alric Thilak

Introduction and Tools:

The system architecture for *Escape from University* was modelled using the Unified Modelling Language (UML). This language allowed us to provide a formal description of the structural and behavioural aspects of the game. UML is standardised notation for representing classes, components and their interactions in object-oriented and entity-component-system designs.
The architecture shows how the game's core modules work together to deliver gameplay and their alignment with the user and system requirements.

Structural diagrams were created using PlantUML. This is a text based tool that uses source descriptions to automatically produce class and component diagrams. This allowed the team to iteratively improve architecture diagrams alongside implementation and ensure version control by using GitHub. During early discussions, we also used Lucidchart to make small component diagrams.

Behavioural aspects of the system were represented using UML sequence and state machine diagrams. These tools were used to model message flows between key objects during gameplay events. This includes player movement, collision handling and event triggering. These diagrams help to support traceability and they show runtime behaviour by connecting interactions with requirements.

All diagrams and interim versions were produced following an Agile/Scrum workflow. This ensures that architectural models evolve alongside each sprint. Our approach was an iterative, tool-supported approach, which allows the team to visualise dependencies and communicate design choices while ensuring that the implemented architecture matched the functional and non functional requirements.

Structural Architecture Diagrams:

The architecture follows a modular object-oriented design built on the LibGDX framework. There is a clear separation between presentation, game logic and entity management. This structure emphasises composition over inheritance and also utilises LibGDX's screen-based architecture for managing game states.

Layered Architecture:

This structure separates any concerns between presentation, game logic and framework responsibilities, allowing changes to user interface elements without disrupting core gameplay mechanics.

Layer	Components	Responsibilities
Presentation layer	MenuScreen, GameScreen, WinScreen, GameOverScreen, TutorialScreen	Handles all user-facing elements such as screen transitions and menu navigation. Converts inputs into game actions
Game Logic layer	Player, Dean, NPC, Locker, BusTicket, GameTimer	Contains the core game rules and entity behaviours. Manages player movement, enemy movement, item

		interactions and game timing
Framework layer	LibGDX Engine, SpriteBatch, OrthographicCamera, TiledMap	Provides the foundational services such as rendering and screen lifecycle control through LibGDX's framework

Class Relationships:

The core game entities follow a composition based design where GameScreen acts as the central coordinator. Each entity encapsulates its own state and behaviour whilst GameScreen manages their interactions and rendering order. The Player class provides character movement and direction, the Dean class implements pursuit AI while the interactive objects like Locker and BusTicker manage their own state transitions.

OOP UML Component Diagram

Our final component diagram represents the implemented architecture where GameScreen serves as the central coordinator, directly managing all game entities and systems. The architecture employs composition-based relationships rather than deep inheritance hierarchies with each entity encapsulating its own state and behaviour while GameScreen handles their interactions and rendering order. This design supports our core requirements: FR_MOVEMENT through direct input handling, FR_ANTAGONIST via the Dean's pursuit AI and the three event types (FR_POSITIVE_EVENT, FR_NEGATIVE_EVENT, FR_HIDDEN_EVENT) through specialised entity interactions. The final OOP component diagram for the game's architecture can be found in Appendix [9].

Behavioural Architecture Diagrams:

The behaviour architecture models the dynamic interactions between core components at runtime. These sequence diagrams illustrate message flows in response to player actions and automated system events using three primary gameplay interactions.

Use Case 1 - Player movement:

This sequence models how the system processes player movement and validates moves within the maze environment. UML sequence diagram in Appendix [1]

Actors and Components:

- Player: entity receiving movement commands
- GameScreen: central coordinator processing input and game state
- TiledMap: provides collision data through tile properties

Process Summary:

- When the player inputs movement, GameScreen.handleInput() processes the command, checks collision via isCellBlocked() and updates the player position if their move was valid. The camera follows the player and the rendering system updates the display.

Use Case 2 - Dean's Pursuit:

This sequence describes the AI logic controlling the Dean's automatic movement through the maze. The UML sequence diagram is in Appendix [2]

Actors and Components:

- GameScreen: triggers entity updates each frame
- Dean: AI-controlled entity pursuing the player
- Player: target entity providing position data

Process Summary:

- For each frame, GameScreen updates the Dean which calculates direction towards the player, checks for valid movement and updates the Dean's position accordingly. Collision detection occurs in GameScreen, triggering reset logic when the player is caught by the Dean.

Use Case 3 Player Searching Objects:

This sequence outlines system behaviour when the player interacts with game objects. UML sequence diagram in Appendix [3]

Actors and Components:

- Player: initiates interactions via the E key being pressed
- GameScreen: manages interaction flow and state changes
- Interactive Objects = BusTicker, Locker, NPC

Process Summary:

When the player presses E near interactive objects, GameScreen checks proximity and triggers appropriate responses: collecting items, activating boosts or displaying dialogue.

Justification and Design Rationale:

Why Object-Oriented Architecture with LibGDX Screens?

We selected a pure object-oriented architecture built on LibGDX's screen pattern because it provided the optimal balance between structure and framework integration. The screen-based architecture naturally fits the game's state transitions (menu -> game -> win/lose screens), while OOP design patterns made the codebase accessible to our team and maintainable throughout development.

Why UML and PlantUML?

UML diagrams helped standardise team communication and ensured shared understanding of system architecture. Using PlantUML allowed us to maintain diagrams in version controlled text format which allowed for easy updates and traceability throughout development. This workflow provided both clarity for design discussions as well as accurate documentation of the implemented system.

Support for Maintainability, Scalability and Testability

The screen-based OOP architecture improves maintainability by isolating different game states into separate classes. Each screen manages its own lifecycle and resources, preventing interference between game states. Scalability is supported through LibGDX's robust framework which handles asset management and rendering efficiently. Testability is enhanced by well-defined class responsibilities with clear interfaces which allow for targeted unit testing of individual game entities.

Trade-offs

The main trade-off in our architecture is the central role of GameScreen as a coordinator as this creates some coupling between systems. However, this is offset by the clarity and simplicity of having a clear central point for game logic. By keeping entities well-encapsulated and using composition, we maintained flexibility while leveraging LibGDX's proven architectural patterns.

Evolution/Iteration Evidence:

Our architecture evolved significantly from the initial concepts to the final implementation. All interim architectural versions, including early ECS diagrams, OOP transition plans and CRC cards are available on our project website at [LINK WEBSITE]

Initial ECS Exploration -> Final OOP Implementation:

Initially, the team explored the idea of using an Entity-Component System (ECS) architecture (as shown in Appendix [4] and [6]) and created component diagrams modelling entities as compositions of data components. However, after evaluating team expertise and project scope, we transitioned to the more familiar, traditional OOP approach using LibGDX's established patterns. This decision prioritised development efficiency and framework alignment over architectural innovation.

Screen Architecture Refinement:

Early designs used a monolithic game controller but we evolved to LibGDX's screen-based architecture after recognising the fact that it better fit our game state management. This change improved separation of concerns and simplified state transitions.

Entity Management Evolution:

Initially, we planned separate manager classes for collision, input and physics but later decided to consolidate these responsibilities into GameScreen after finding out that LibGDX's integrated approach reduced complexity without sacrificing any clarity.

Final OOP Architecture:

The final OOP component diagram for the game's architecture can be found in Appendix [9].

CRC Cards:

Class	**Responsibilities**	**Collaborators**
GameScreen	Process player input, update game entities, manage collisions, handle screen transitions, coordinate rendering	Player, Dean, BusTicker, Locker, GameTimer
Player	Handle movement direction, maintain position state, manage sprite rendering	GameScreen, SpriteBatch
Dean	Pursue the player using simple AI, update position each frame, reset when catching the player	GameScreen, Player
NPC	Display dialogue when the player interacts with it, manage dialogue visibility, render NPC sprite and message	GameScreen, Player, SpriteBatch

Locker	Provide speed boost when interacted with, manager boost duration timer, display interaction messagers, render locker sprite	GameScreen, Player, SpriteBatch
BusTicket	Manage collection state, render ticket in world when discoverable, render UI icon when collected, handle ticket discovery logic	GameScreen, SpriteBatch, OrthographicCamera
GameTimer	Count down game time from 300 seconds (5 minutes), format time display as mm:ss, update UI label display	GameScreen, uiStage, Skin
MenuScreen	Display main menu interface, handle menu navigation inputs, transition to tutorial/game scenes, manage win screen graphics	MyGame, GameScreen, SpriteBatch
WinScreen	Display victory message and final score, show score breakdown with penalties, handle return to menu navigation, render win screen graphics	MyGame, GameScreen, SpriteBatch
GameOverScreen	Display game over message, handle menu return navigation, render loss screen interface, manage screen resources	MyGame, GameScreen, SpriteBatch
TutorialScreen	Display tutorial instructions image, handle progression to main game, manage tutorial resources, render tutorial information	MyGame, GameScreen, SpriteBatch
MyGame	Manage screen lifecycle and transitions, create initial application context, coordinate between different screens, implement LibGDX game interface	All Screen classes, LibGDX framework

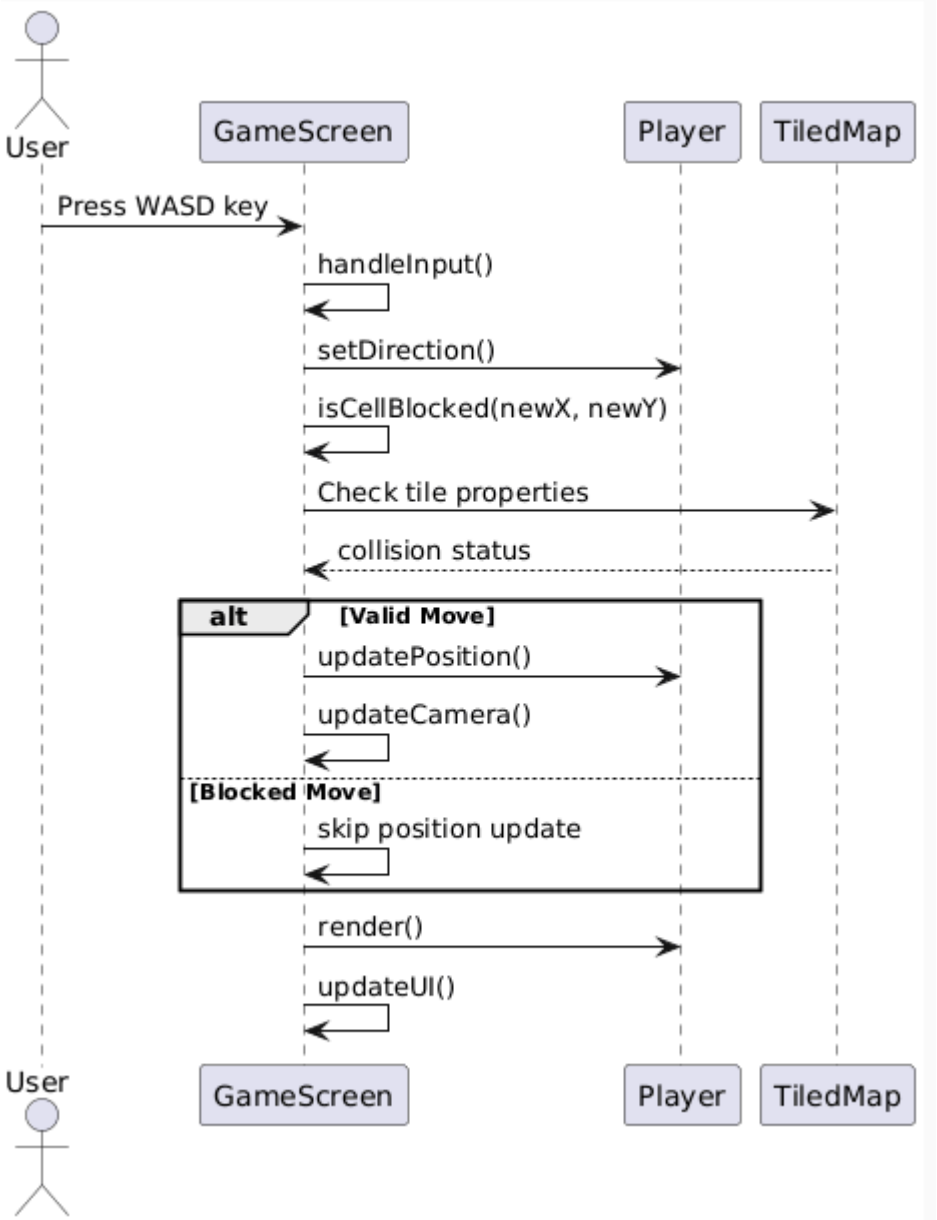
Traceability to Requirements:

Requirement ID	Requirement Description	Related Component(s)	Architectural Rationale
FR_MOVEMENT	The system shall allow keyboard inputs to control player movement	GameScreen.handleInput(), Player.setDirection(), Player.getPosition(), GameScreen.isCellBlocked()	Input processed in GameScreen.handleInput() which detects WASD keys, sets player direction via Player.setDirection(), validates movement through GameScreen.isCellBlocked() before updating player position
FR_ANTAGONIST	The system shall include an antagonist who follows the player	Dean.update(), Dean.getPosition(), Dean.resetPosition(),GameScreen.render(), Player.getPosition(), GameScreen.isCellBlocked()	Dean.update() calculates pursuit direction each frame using player position, moves with speed = 0.7f, resets player to start when caught via Dean.resetToStart()
FR_POSITIVE_EVENT	The system shall present a positive event in which the player finds a locker, opens it and gains a speed boost advantage	Locker.update(), Locker.isBoostActive(), GameScreen.handleInput() movement speed logic	Locker.update() detects player proximity and E key presses, activates temporary speed boost, GameScreen doubles movement speed

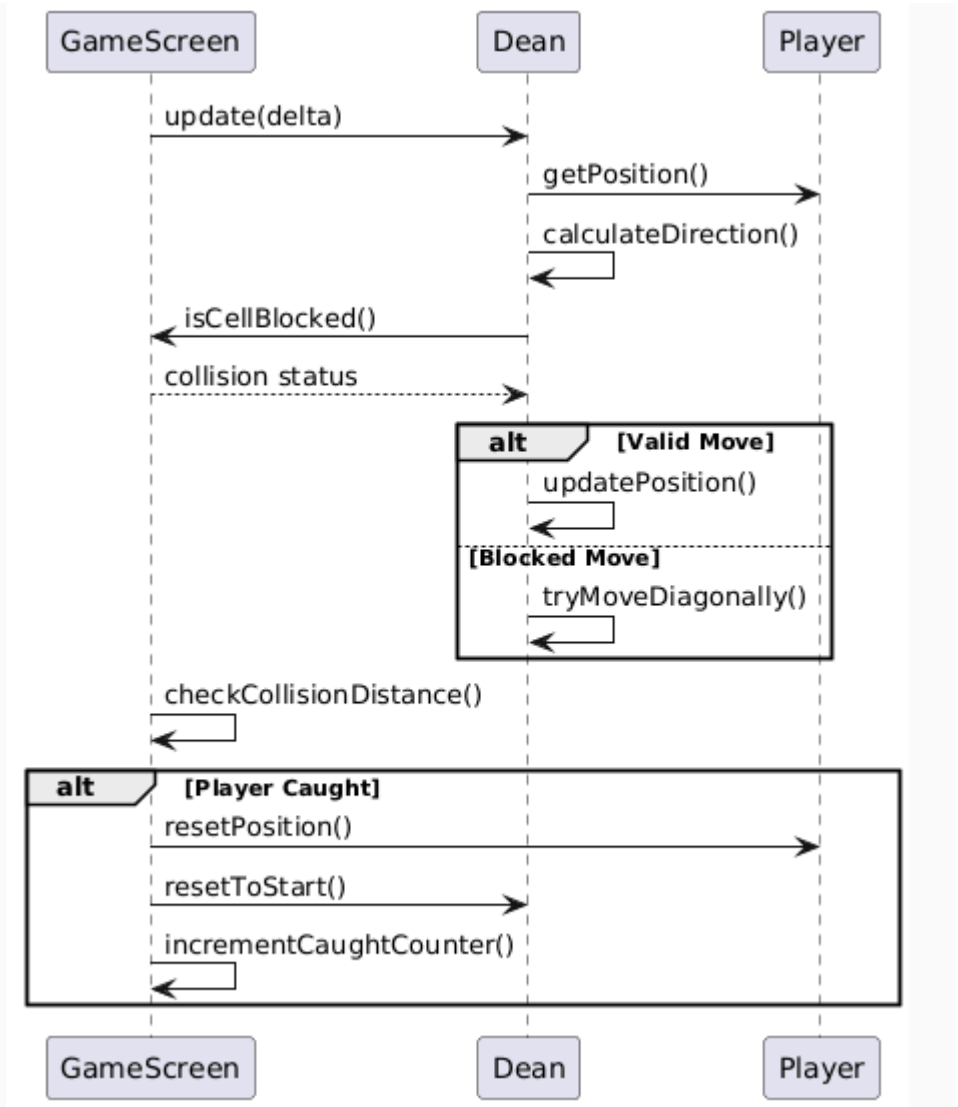
			when boost is active
FR_NEGATIVE_EVENT	The system shall present a negative event of the player not being to get on the bus without their bus pass	BusTicket.isCollected(), GameScreen.canEndGame , busInteractionArea Rectangle	Bus boarding is blocked until BusTicket.isCollected() returns true, GameScreen checks rectangle overlap before allowing game completion
FR_HIDDEN_EVENT	The system shall contain a hidden event of the player's bus pass being hidden in a bush which, until triggered, cannot be seen on the map	BusTicket.isCollected(), NPC.showMessage, BusTicket.render() conditional rendering	Bus ticket initially hidden, revealed via BusTicket.discover() when player gets close, NPC provides hint through dialogue message
FR_OFFLINE	The system and all its features shall not require any connection to a network	All asset loading via local files	All textures and maps are loaded from local filesystem without network dependencies
FR_END_SCORE	The system shall display the user's end score when they complete or fail the game	WinScreen, GameOverScreen, GameScreen.calculateFinal Score(), GameTimer.getTimeLeft()	Final score calculated from remaining time minus penalties, displayed when player wins the game
FR_USER_TIMER	The system shall display a timer to the user user which displays how long they have been playing for	GameTimer, timerLabel, uiStage	Countdown timer displays remaining time in mm:ss format through Scene2D UI label

Appendix

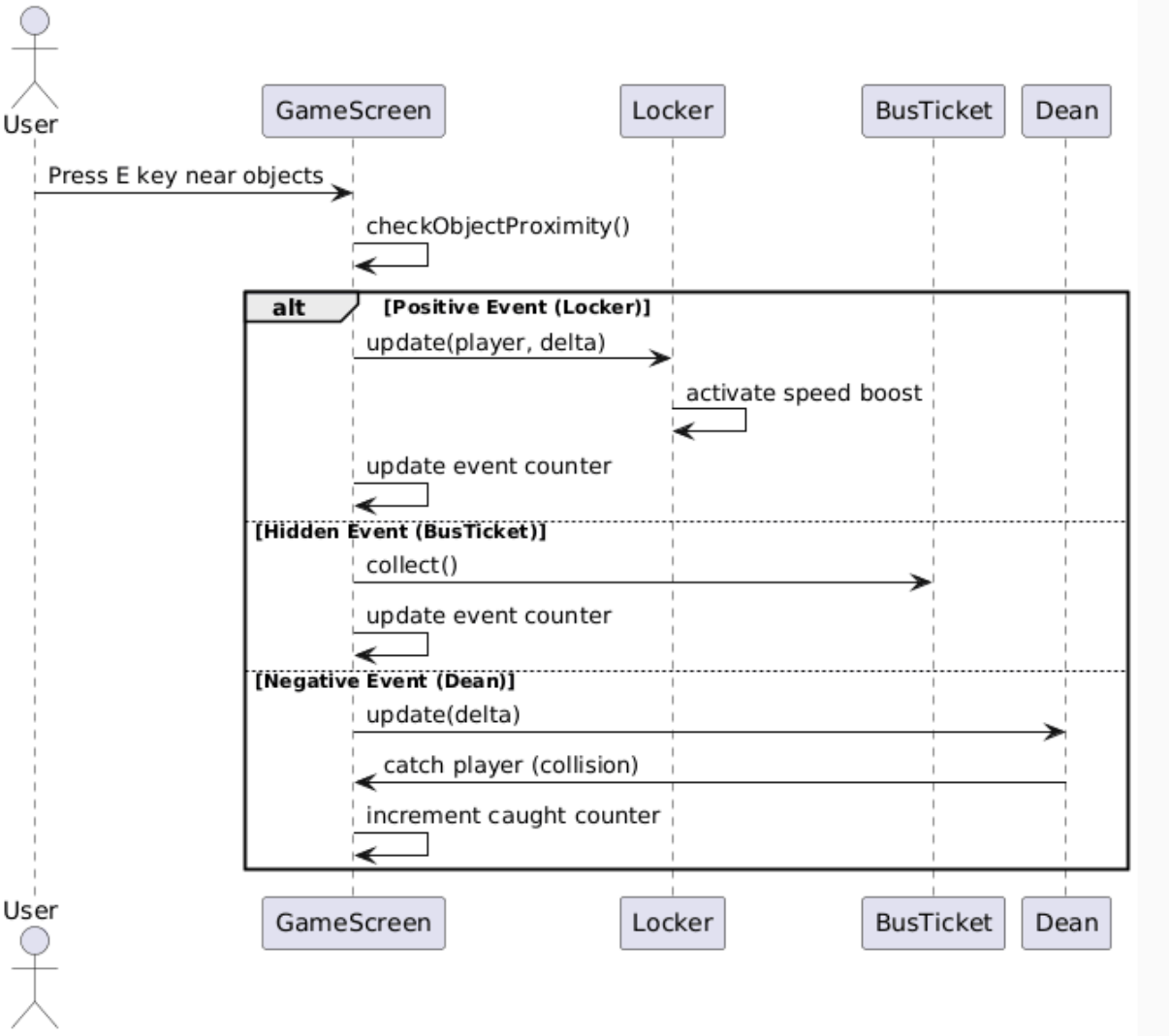
[1] - Use Case 1 Sequence Diagram



[2] - Use Case 2 Sequence Diagram


GameScreen
Dean
Player
update(delta)
getPosition()
calculateDirection()
isCellBlocked()
collision status
alt
[Valid Move]
updatePosition()
[Blocked Move]
tryMoveDiagonally()
checkCollisionDistance()
alt
[Player Caught]
resetPosition()
resetToStart()
incrementCaughtCounter()
GameScreen
Dean
Player

[3] - Use Case 3 Sequence Diagram



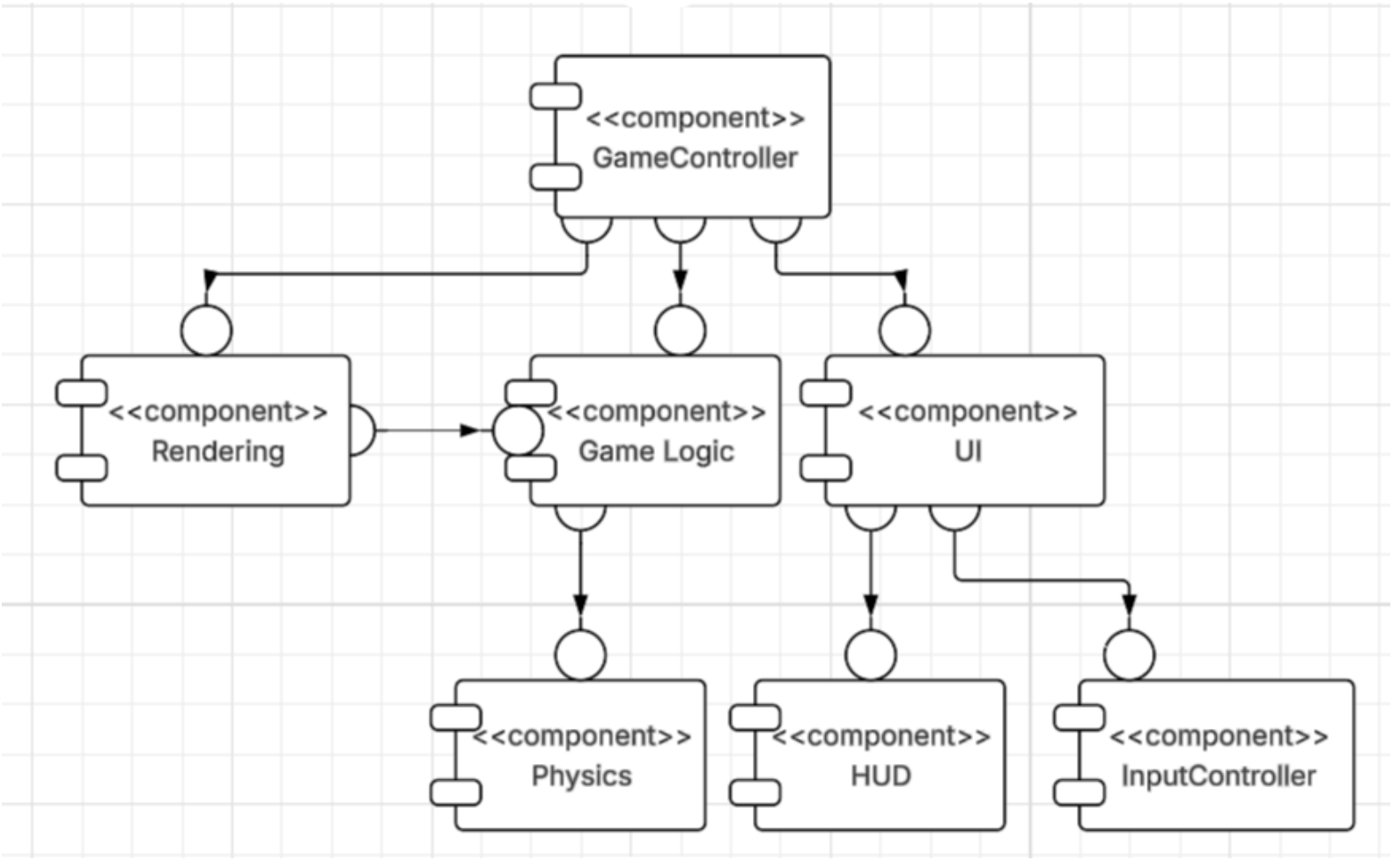
[4] - ECS Main Components

ECS Concept	**UML Representation**	**Example Components**
Entity	Containers shown as components or nodes	PlayerEntity, EnemyEntity, NPCEntity
Components	Data-holding modules	PositionComponent, Velocity Component, SpriteComponent
System	Behavioural logic units	MovementSystem, RenderSystem, InputSystem, CollideSystem
Engine	The central co-ordinator that runs the systems we design. A container with entities, components and systems inside. A database.	Delete, Create → enables it to create and delete systems, entities and components.

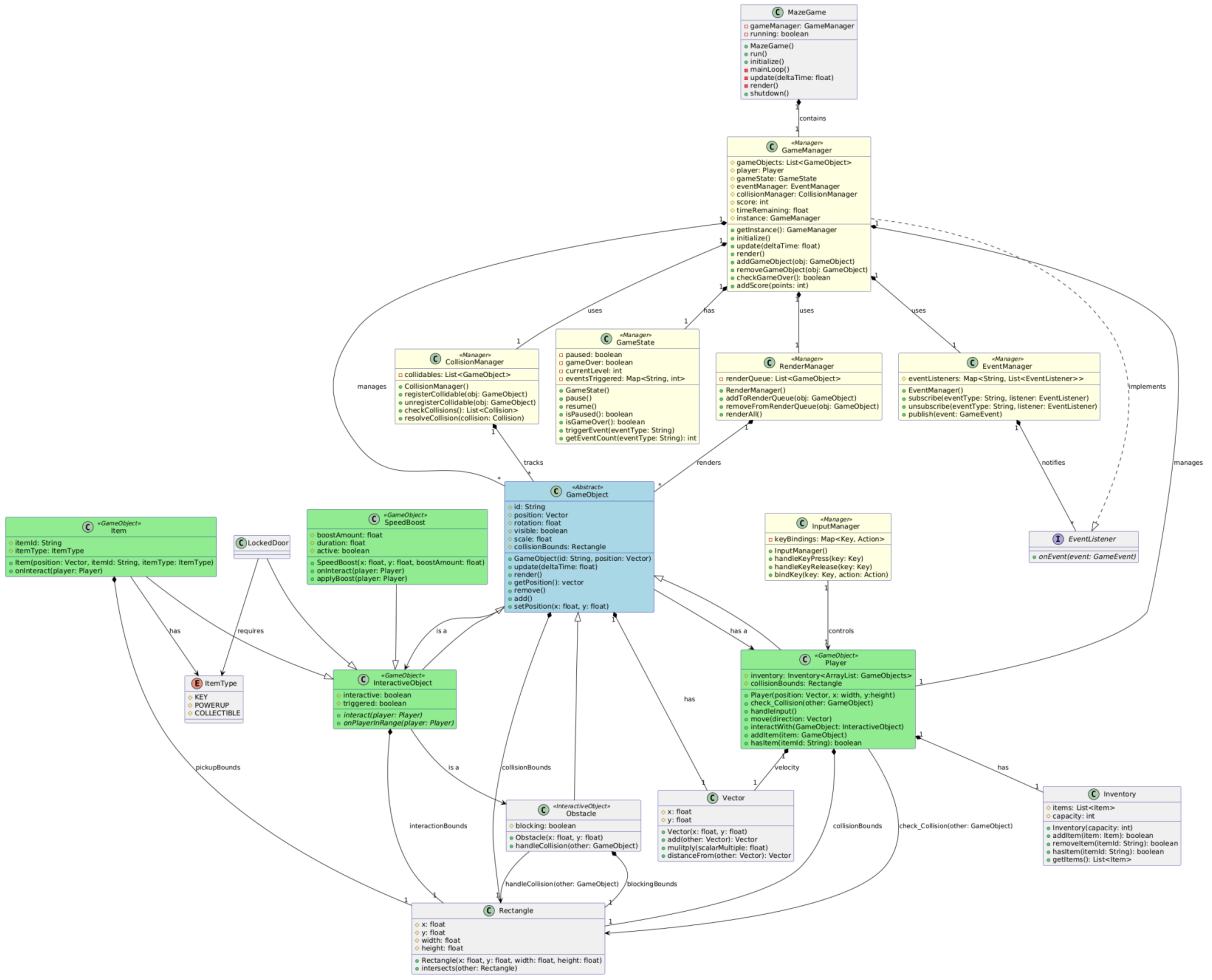
[5] - OOP Main Components

Component	Example Classes	Responsibility
GameController	MainGame, Game Screen	Entry point (where game starts running), game loop (cycle that keeps the game going) state transitions (screen switching)
UI / Input	HUD (heads-up display), Menu, InputHandler	User interface and input handling
GameObjects	Player, Enemy, Item, Maze	Core in-game entities each with their own class
Physics / Movement	CollisionManager, PhysicsEngine	Collision detection and motion logic
Rendering	Renderer, SpriteLoader	Drawing game objects using LigGDXs APIs
Persistence	Preferences	Settings menu

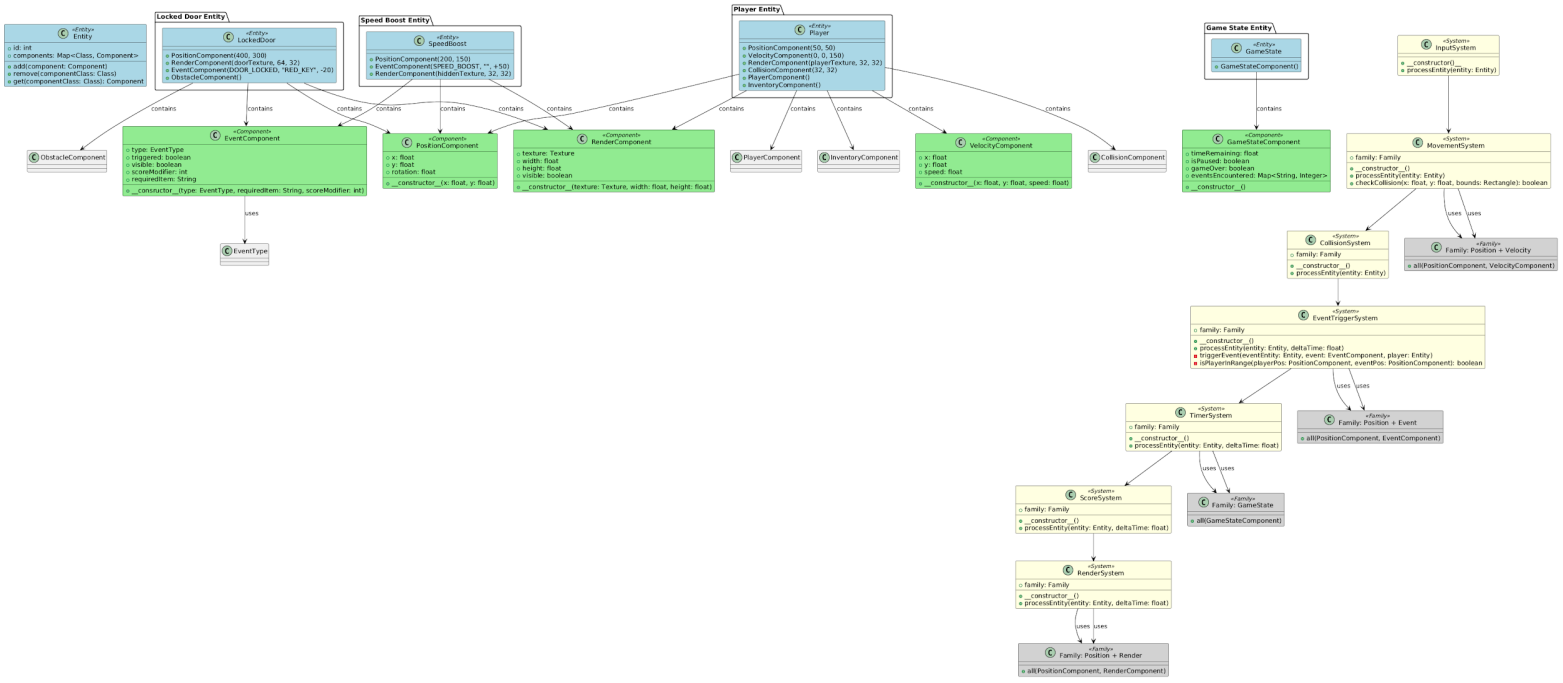
[6] - ECS UML Component Diagram Version 1



[7] - OOP Component Diagram Version 1



[8] - ECS Component Diagram Version 2



[9] - Final OOP Component Diagram (Version 3)