Architecture

Cohort 1, Group 9

Chris Sewell,
Fedor Kurochkin,
Matt Durham,
Max Peterson,
Oladapo Olaniran,
Wojtek Tomaszewski,
Yuqi Fu.

This document serves to describe the product's different architectural structures, inner components, the relationships between them and the justification for choosing them. The document will focus on the completed architectural diagrams for the current version of the game, with evidence of the iterative history of the project lifecycle found on the supporting Design Iterations webpages.
The diagrams in this document have first been stormed or sketched using the tools Miro, draw.io and starUML before being properly implemented via UML syntax using plantUML.

System Structure
Throughout the architectural design process the team followed the 5 steps of software development outlined by Sommerville (2016), iteratively identifying and updating components based on characteristics to meet requirements. (see requirements page).

After reviewing the different architectural styles described by Sommerville (2016) it was clear that a monolithic closed-layered structure with an embedded ECS system was best suited for the project due to it being most applicable to small-scale applications where scalability and distribution isn't a priority, with maintainability being convenient due to layer isolation and game logic centralised in the business layer.
(Richards and Ford, 2020; University of York, 2025).
The following elements outline the characteristics, design decisions and principles that informed this architecture structure.

Architectural Characteristics (non-functional requirements)
Following steps 1-3 of the Software Development Life Cycle, the team conceptualised initial components based on the most applicable non-functional requirements.
These characteristics derived from use cases and team storming sessions were then grouped with corresponding user and constraint requirements into a table that would serve as a reference throughout the project lifecycle (see Characteristics Table page).

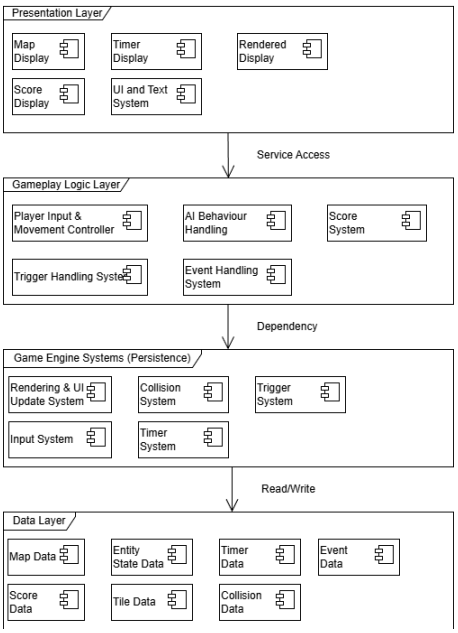
Design Decisions
The team agreed on architectural design decisions based on user and system requirements whilst ensuring these designs were compatible with the chosen Java based game engine: libGDX (satisfying CR_ENGINE).
The team chose this OOP game engine due to the on-going library support, tile integration and compatibility with the monolithic layered style architecture.
The team opted for an Entity-Component-System (ECS) structure to complement the monolithic style by embedding it within the business layer (game logic) to represent class-component relationships and ensure extensible game logic.

The business layer (ECS structure) focused on game logic and component systems, integrating movement, events, scoring and the timer (ensuring UR_EVENTS, UR_SCORE, FR_SCORE_CALC, FR_TIMER were adequately met).
A traditional AABB and 'Trigger' based collision system were implemented for movement and event handling for simplicity and performance (NFR_PERFORMANCE, UR_EVENTS and CR_CHEAT).
To mitigate design trade offs, we limited communication between layers to prevent the sinkhole anit-pattern problems due to too many layer bypasses ensuring systems interact indirectly via shared components.
We embedded event status tracking within entities to maintain inter package traceability for score and event logic. (UR_SCORE, UR_EVENTS).

Design Principles

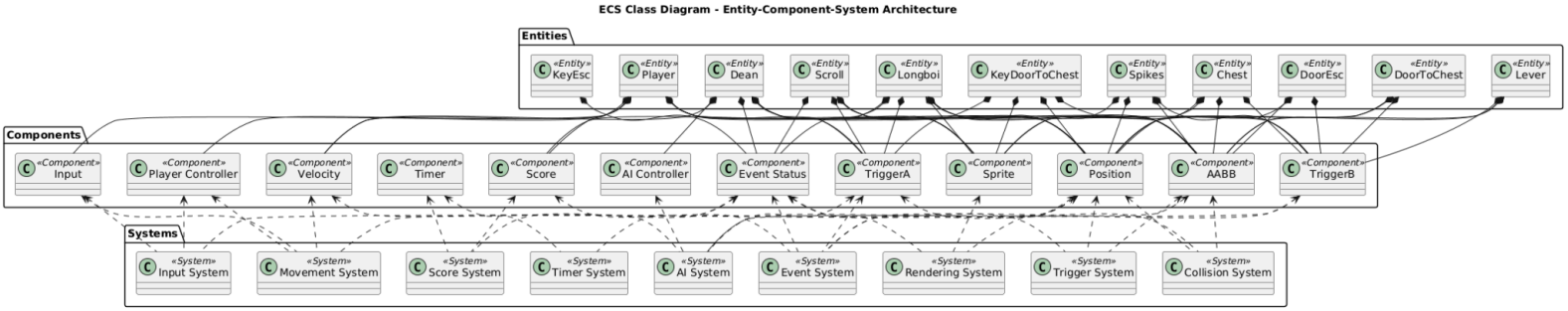
The design favours low coupling and high cohesion since each ECS system handles one responsibility.
Separation of concerns due to layer isolation: presentation (UI), game logic and data management on separate layers.
Relative modularity from the ECS architecture allows new entities and event logic to be added without altering existing systems.
Information hiding: systems interact via shared components not direct access maintaining low coupling.
Together these principles ensure modularity through effective cohesion, coupling and connascence between all classes and components.

Figure 1: Monolithic closed-layered architecture diagram.



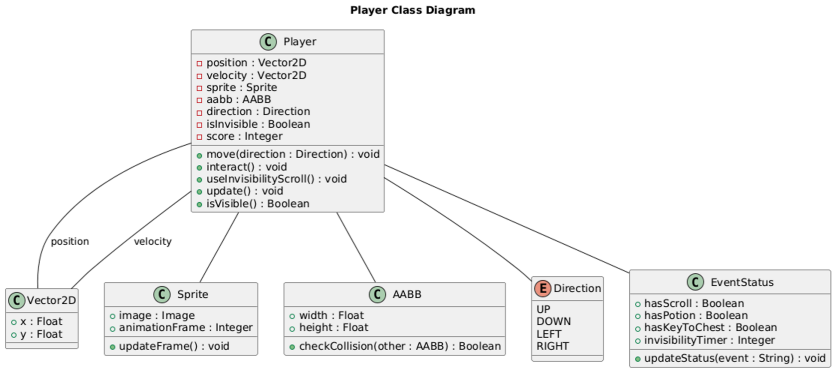
This diagram illustrates the overall architectural structure of the Escape from Uni game, composed of four layers: Presentation (UI and display), Business (game logic), Persistence (system handling) and Data. (see Design Iterations page for structure evolution)
The ECS operates inside the Business layer which requests system handling via the Persistence layer, while the Presentation layer manages the UI and the Data layer stores assets and entity information.
A monolithic closed-layered style was selected for simplicity and maintainability (NFR_RESILIENCE). This structure is optimised for small-scale projects which suits the team size and timeframe (CR_TIMEFRAME).

Figure 2: Entity-Component-System architecture diagram



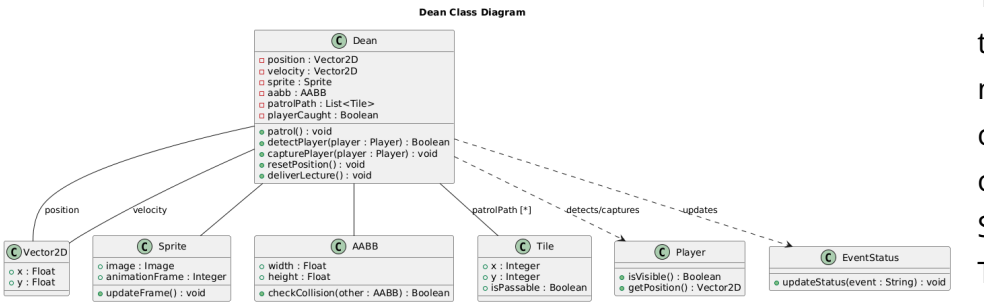
This diagram illustrates the ECS structures inter-package relationships via components to form modular and reusable game logic..
The diagram shows the dependencies and association/ container relationships encompassing: Player, Dean, Longboi, Scroll and Potion entities that inherit from the Entity superclass and share components such as Sprite, AABB and Vector2D etc. EventStaus records triggered items, visibility and score and passes that data to systems like Trigger, Collision and Event to manage activation consistently across entities.
Systems like Movement, Collision and Rendering execute on shared components rather than directly on entities themselves, reducing unnecessary code duplication and improving modularity and reusability satisfying UR_EVENTS, NFR_RESILIENCE and NFR_RELIABILITY.
(see Design Iterations page for the storming, sketches and design evolution).

Figure 3: Player class diagram



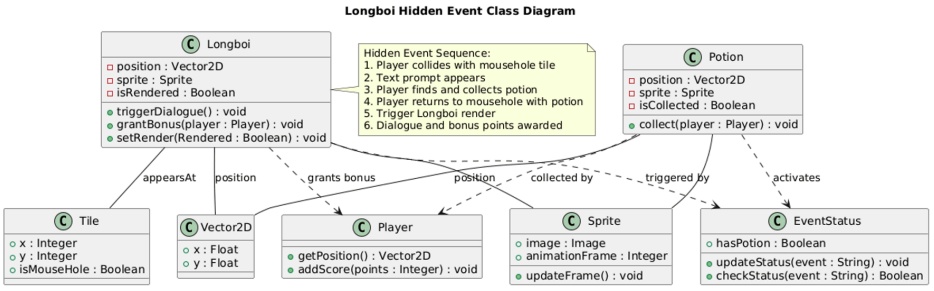
While ECS defines class-component connectivity at a high level, these diagrams focus on specific behaviour and dependencies of classes. The player class integrates Sprite, AABB, Vector2D and EventStatus components to manage position, rendering and invisibility (detectable) state.
Collision and interaction triggers (TriggerA and TriggerB) enable movement, input handling and event activation while tracking score and visibility. This design satisfies FR_MOVEMENT, FR_INVARIENTS and FR_MAP by ensuring consistent player rendering, input position response and collision feedback.

Figure 4: Dean class diagram (negative event trigger)



This diagram illustrates how the Dean class functions as a negative event while sharing components with other classes such as Vector2D, Sprite, AABB, and EventState. The Dean extends base Entity functionality with patrol, detection, capture and reset behaviour, interacting with the Player and EventStatus classes to trigger its event.
These methods form its patrol and capture logic: player invisibility checks, reset logic and score penalties upon capture, demonstrating class reuse and dependency management within the ECS structure thus satisfying: FR_SCORE_CALC, FR_TIMER, UR_EVENTS.

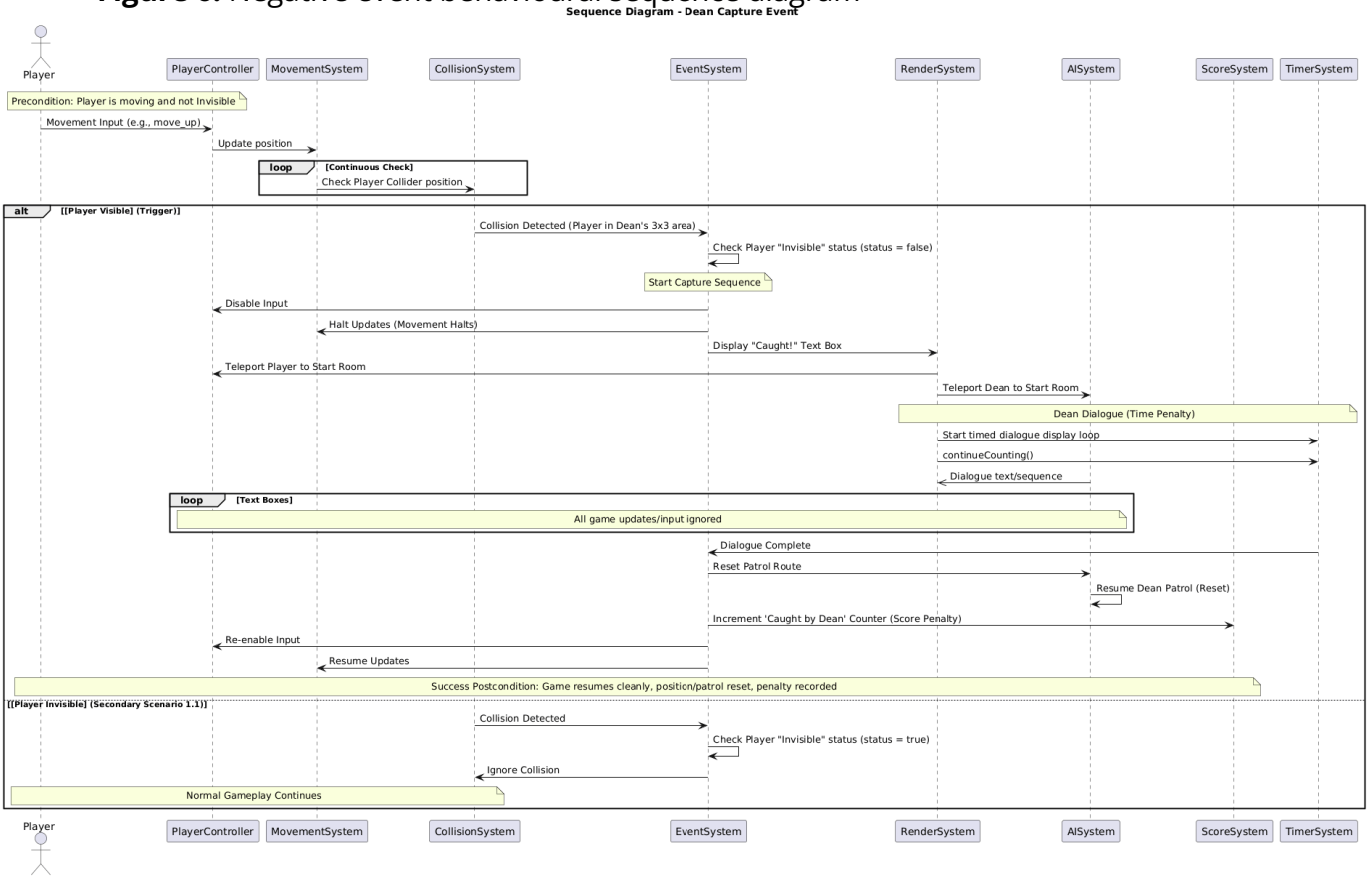
Figure 5: Longboi class diagram (Hidden event)



This diagram illustrates how the hidden event links class associations between Player, Potion and Longboi. Collisions with Potion (TriggerA) activate EventStatus.hasPotion and returning to the MouseHole tile triggers the rendering of Longboi, awarding bonus points. This logic satisfies UR_EVENTS, FR_SCORE_CALC, FR_EVENTS by using trigger systems and shared components to coordinate a multi-step event hidden from the player. (see Design Iterations page for a Unified class diagram of the 3 classes illustrated above)

The diagram below illustrates the full negative event process triggered when the player is detected and captured by the patrolling Dean. The sequence includes branching edge case parallel event logic handled without conflict. Initially adapted from earlier use cases this diagram evolved iteratively to update event logic following re-evaluation of requirements and testing feedback.
The diagram demonstrates detection, input halting, score modifiers, resetting event logic and ensuring reliable failState handling smoothly all within the ECS framework satisfying UR_EVENTS, FR_SCORE_CALC, FR_TIMER, NFR_RESILIENCE.

Figure 6: Negative event behavioural sequence diagram



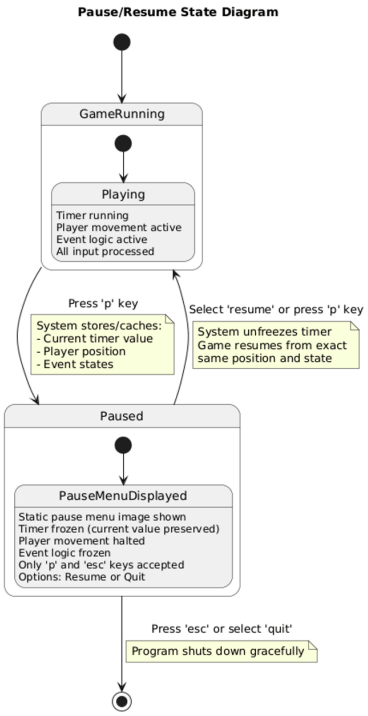


Figure 7: Pause behavioural state diagram

A state diagram was an intuitive choice for illustrating the pause game logic behaviour before and after the player inputs the pause key (P).

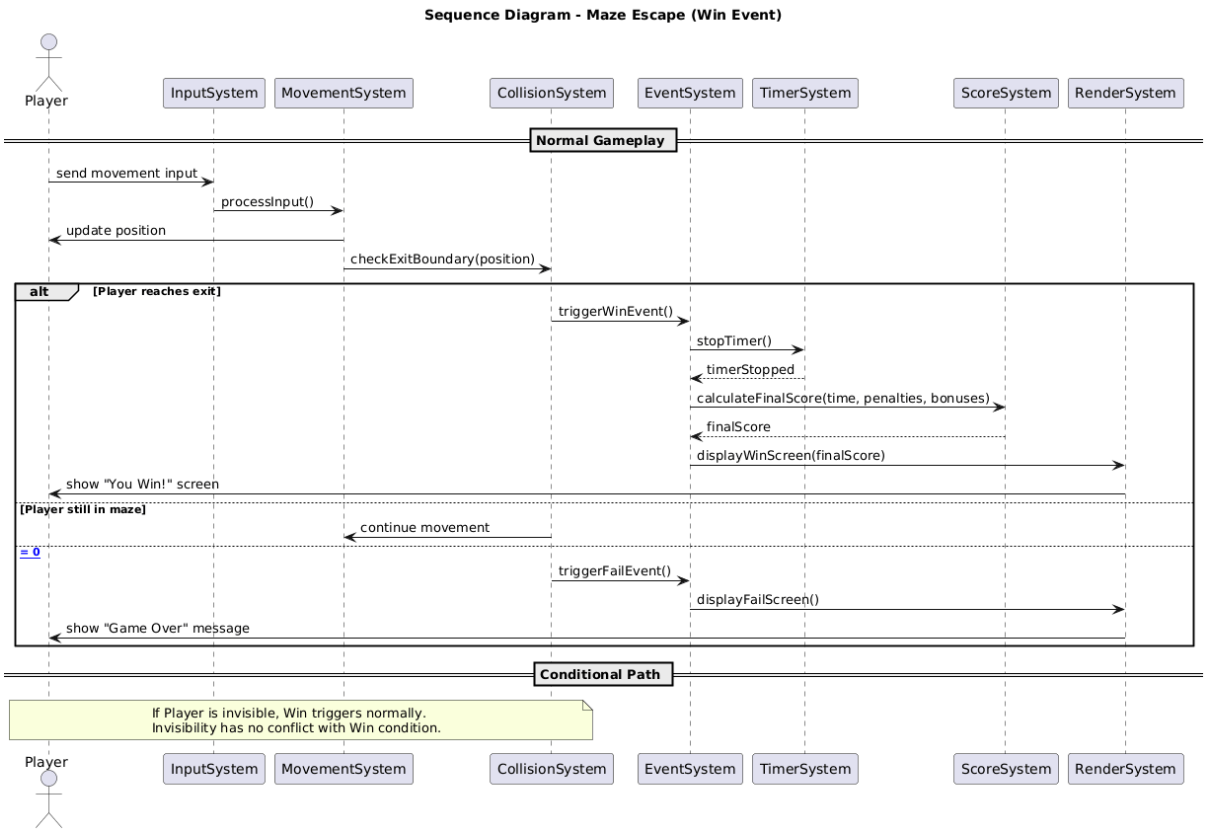
In accordance with the user requirements, the game must include a pause feature that halts all game logic including the timer, ignoring all player input with exception to menu keys such as P (resume) and Esc (quit).

The pause key triggers a static image screen representing a menu displaying the pause/resume and quit keys along with simple directional key instructions and a brief description of how to escape the maze.

Upon resume all game logic including events and timer is unfrozen successfully addressing: UR_PAUSE, FR_RESUME, FR_TIMER, UR_HOW, UR_EASE, NFR_INSTRUCTIONS, NFR_UI, NFR_RELIABILITY (quit key) and FR_RESET.

(see Design Iterations page for resume state evolution and player movement state diagram)

Figure 8: Maze Escape sequence diagram



This diagram illustrates a complete successful maze escape run with event conflict handling. It was adapted from initial use cases and updated following programming team feedback and requirements re-evaluation. The sequence shows the player triggering the win event only when crossing the exit boundary while the timer is greater than zero.
If the timer reaches zero the failEvent is triggered instead displaying the failure screen (see Design Iterations menu screens) skipping score calculation.
Triggering the win event halts the timer and all other game logic, calculates the player score based on time remaining and any event bonuses or penalties thus satisfying: UR_ESCAPE, UR_SCORE, FR_SCORE_CALC, FR_INVARIENTS, FR_RESET and CR_CHEAT.

This document, supported by the Design Iterations evidence, presents the complete architectural lifecycle of ‘Escape from Uni’ following the 5 stages of software development (Sommerville, 2016).
The final design integrates a closed-layered architecture with an embedded Entity-Component-System to achieve maintainability, modularity and scalability while meeting core user and system requirements. Each UML diagram was iteratively developed alongside the requirements and characteristics referencing systems, demonstrating traceability between system, behaviour and structure of the chosen architecture.

References

Sommerville, I. (2016) *Software Engineering.* 10th edn. Harlow: Pearson Education Limited.

Richards, M. and Ford, N. (2020) *Fundamentals of Software Architecture.* Sebastopol, CA: O'Reilly Media.

University of York (2025) *ENG1: Software Architecture Lecture Slides.* Department of Computer Science, University of York

PlantUML (2025) *UML Diagramming Tool.* Available at: https://plantuml.com (Accessed: 9 November 2025).

draw.io (2025) *Diagramming and Visual Design Tool.* Available at: https://www.draw.io (Accessed: 9 November 2025).