# Architecture


Cohort 1, Team 3

Ari Kikezos

Arya Enkhnasan

Ben Green

Calum Wright

Lilac Graham

Skylar Garrett

This document functions as describing the product's architectural structure, core components, and the relationships between them, along with the logic behind the chosen design decisions. It focuses on the current version of the game architecture, including updated and newly introduced diagrams that reflect additional game events implemented during assessment 2. Evidence of earlier architectural exploration and iterative design can be found on the supporting Design Iterations webpages. (See the initial Design Iterations) Initial diagrams were stormed and sketched using tools such as Miro, draw.io, and StarUML before being formalised using UML syntax with PlantUML, with subsequent refinements captured within this document.

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 the initial Requirements).

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 the initial Characteristics Table page).

Design Decisions

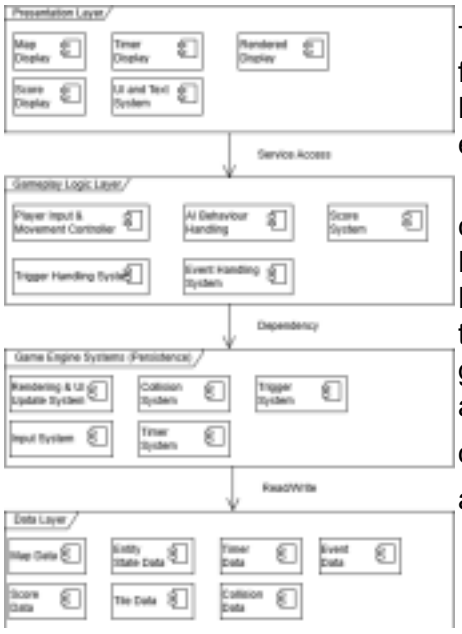
The team agreed on architectural design decisions based on user and system requirements whilst ensuring these designs were compatible with the chosen Java based game engine: libGDX (satisfying CR_ENGINE).
The team chose this OOP game engine due to the on-going library support, tile integration and compatibility 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.



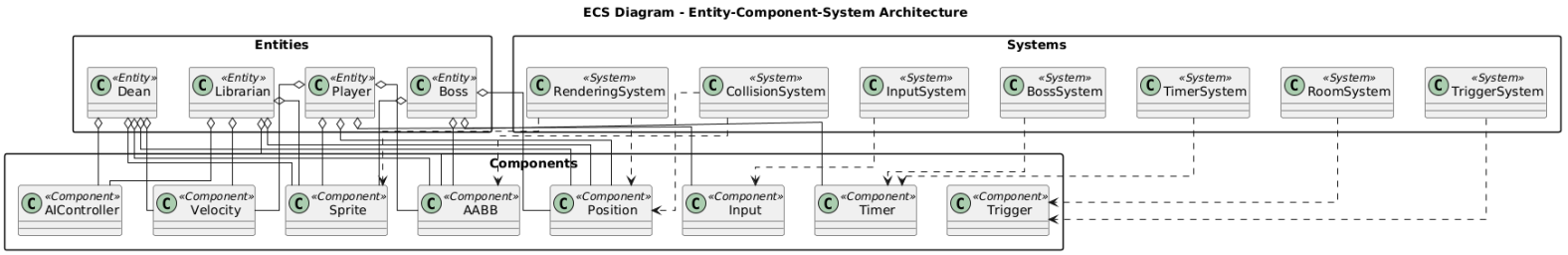

This diagram illustrates the overall architectural structure of Escape from Uni, composed of four layers: Presentation (UI and display), Business (game logic and ECS), Persistence (limited data storage, e.g., leaderboard), and Data (assets and entity information).

The ECS operates inside the Business layer, requesting and coordinating systems to manage entity behaviour, while the Presentation layer handles all UI, overlays, and visual feedback. The Persistence layer stores essential data such as the leaderboard, and the Data layer contains maps, sprites, and other assets required by the game. A monolithic closed-layered style was selected for simplicity and maintainability (NFR_RESILIENCE)

optimised for small-scale projects which suits the team size

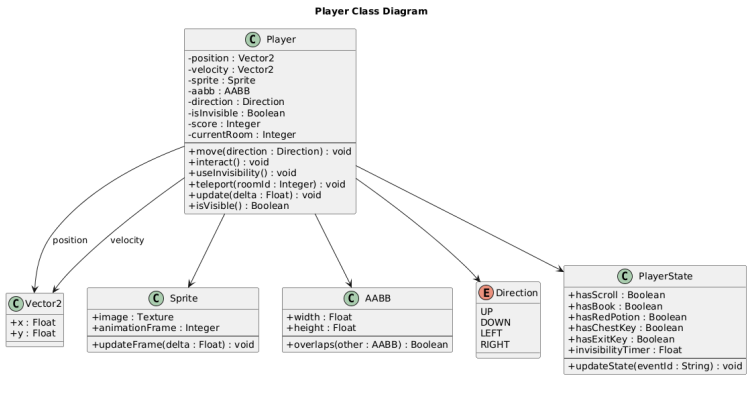
and timeframe (CR_TIMEFRAME).

**Figure 2**: Entity-Component-System architecture diagram



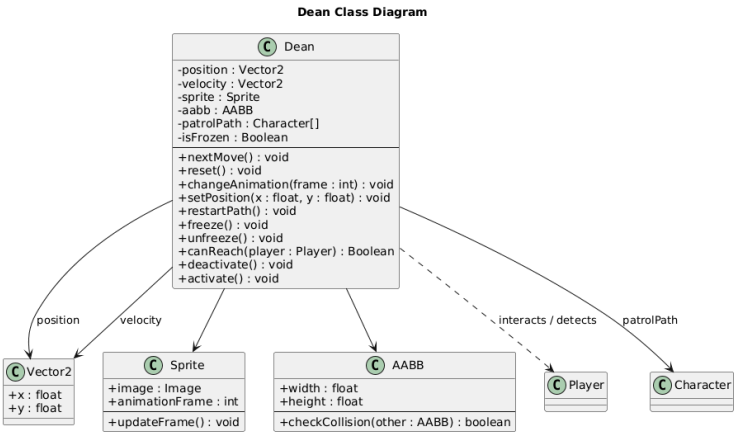

This diagram represents the Entity–Component–System (ECS) architecture, highlighting the inter-package relationships that enable modular and reusable game logic within the expanded game structure. Entities such as the Player, Dean, Librarian, Boss, Longboi, and various interactive objects are composed of shared components—such as Position, Velocity, Sprite, AABB, Input, AIController, Trigger, and Timer—rather than relying on inheritance. Components encapsulate state and behaviour data, while systems including InputSystem, CollisionSystem, TriggerSystem, RenderingSystem, TimerSystem, AchievementSystem, ToastSystem, RoomSystem, and BossSystem operate exclusively on these shared components. Player interactions, conditional events, room transitions, and boss encounters are coordinated through system-level logic, allowing multi-step gameplay behaviour to be implemented without embedding logic directly within entities. By decoupling entities from system behaviour and enabling systems to act on common components, this architecture reduces code duplication and maintains modularity, resilience, and reliability, satisfying the requirements UR_EVENTS, NFR_RESILIENCE, and NFR_RELIABILITY

**Figure 3:** Player class diagram



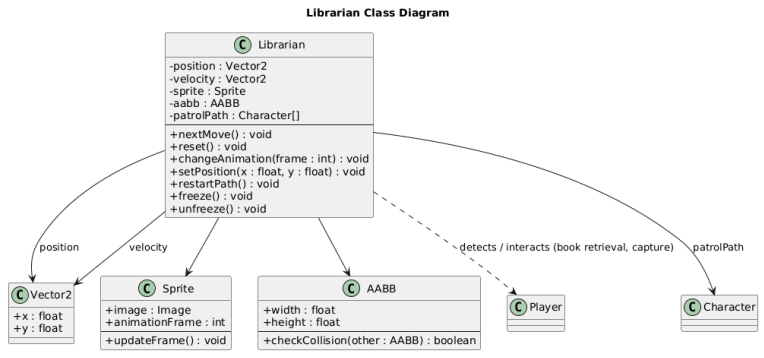

While ECS defines class–component connectivity at a high level, these diagrams focus on the Player's internal state representation and its interactions with external systems rather than architectural control flow. The Player class maintains position, speed, keys, potions, and invisibility state, using a Vector2 for coordinate tracking. Rendering and animation frames are handled externally, while collision detection and event evaluation are managed by dedicated game systems. Movement, input state, and status effects are represented within the Player class, whereas freezing, position resets on capture, and event-driven state transitions are coordinated externally at the system level. When the player is captured by the Dean or Librarian, the player's position is reset to the starting location or the prison and they are temporarily frozen, while inventory and other persistent state variables remain unchanged. This design satisfies FR_MOVEMENT, FR_INVARIANTS, and FR_MAP by ensuring consistent state updates, responsive input handling, and reliable interaction with game systems.

**Figure 4**: Dean class diagram (negative event trigger)



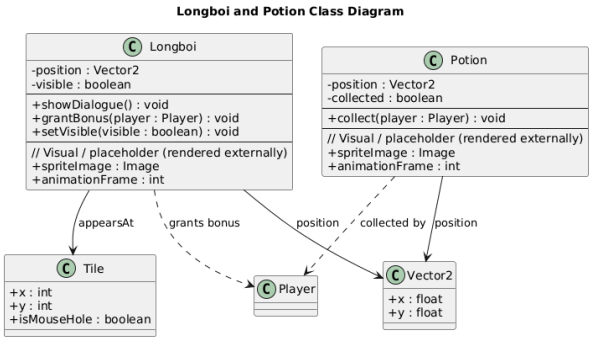

This diagram illustrates how the Dean class represents a negative event entity within the game, focusing on its internal state and interaction points with external systems rather than behavioural ownership. It uses shared components such as Vector2, Sprite, and AABB to represent position, movement, and collision boundaries. The Dean maintains a patrol path and supports state changes such as freezing and position resets, while interactions with the Player act as triggers for externally coordinated capture sequences. Capture handling, including timing, score penalties, and player state changes, is managed at the system level rather than encapsulated within the Dean class itself. This design preserves modularity, promotes component reuse, and supports reliable event-driven gameplay, satisfying FR_SCORE_CALC, FR_TIMER, and UR_EVENTS.

**Figure 5**: Librarian Class Diagram (negative event trigger)



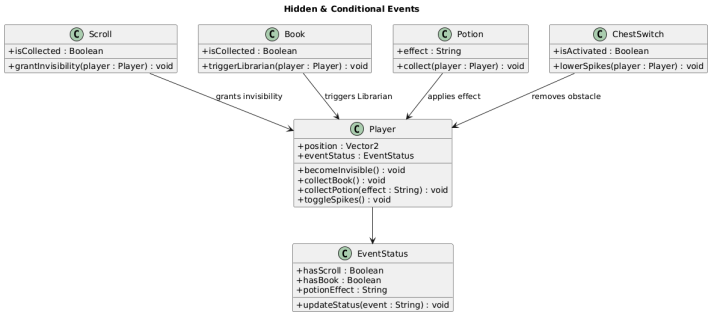

This diagram represents how the Librarian triggers differently from the Dean by being a conditionally activated negative event rather than continuously active. The Librarian is initially frozen and only becomes active after the player steals the book, reflecting its reactive role. In class operations, differences from Dean are illustrated, highlighting its conditional behavior while reusing shared methods like freeze, unfreeze, reset, and patrol path management. Capture events are coordinated externally by the game logic, maintaining modularity and supporting smooth event-driven gameplay, satisfying UR_EVENTS and FR_TIMER.

**Figure 6:** Longboi class diagram (Hidden event)



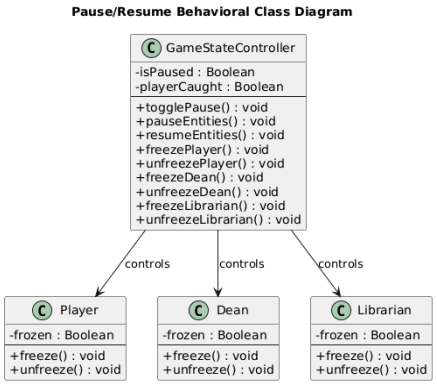

This diagram illustrates how the hidden event links class associations between Player, Potion, and Longboi. Collecting a Potion triggers the collect(player) method, updating the player's potion state and activating related game effects. When the Player interacts with Longboi, the grantBonus(player) method awards bonus points and optionally shows dialogue. Tile coordinates define where Longboi appears, while Vector2 tracks positions. Rendering and animation are handled externally. This logic satisfies UR_EVENTS, FR_SCORE_CALC, and FR_EVENTS by using interaction methods and external systems to coordinate a multi-step event hidden from the player.

**Figure 7**: Hidden & Conditional Events



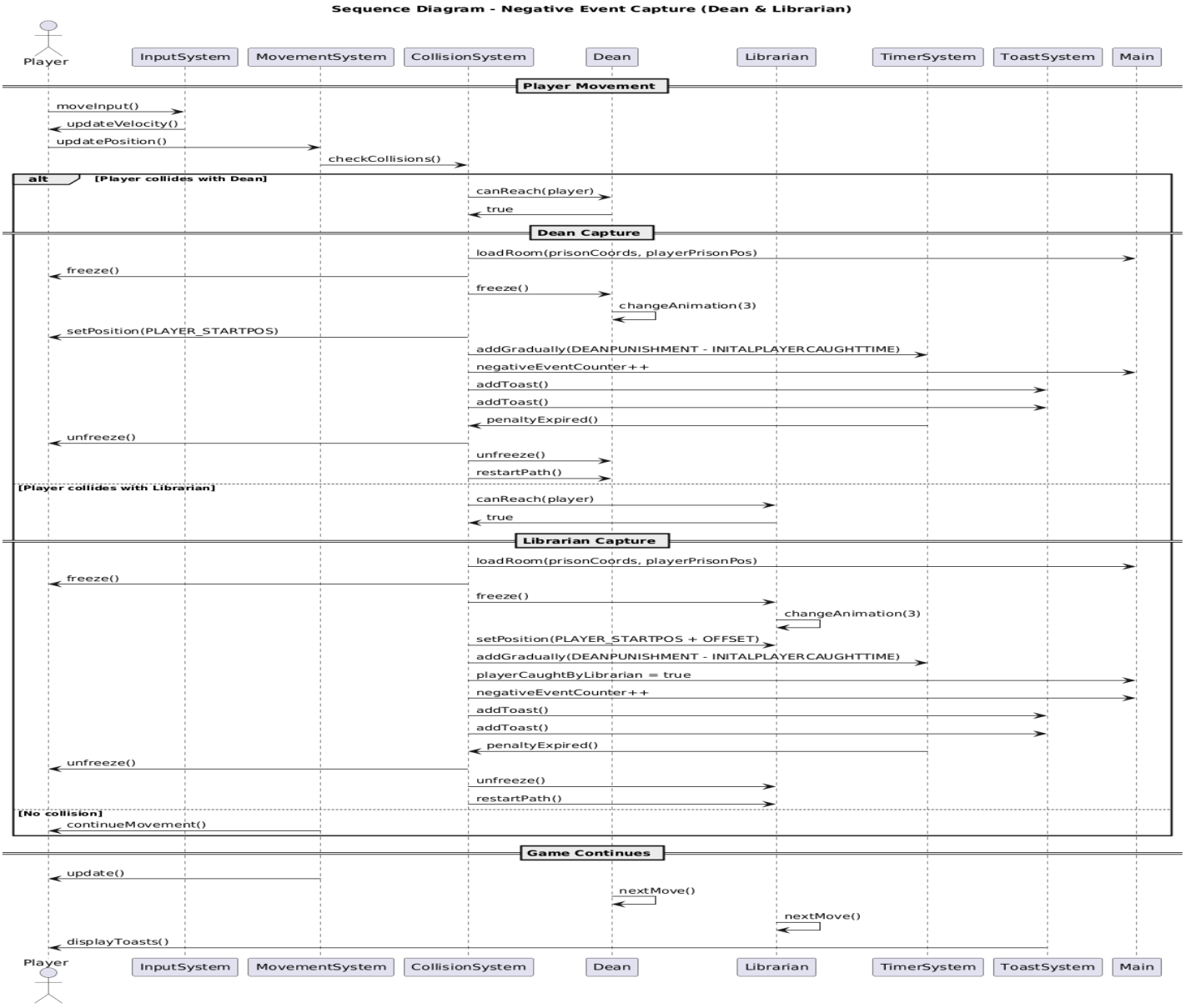

This diagram illustrates the interactions between the player and conditional or visible events in the game, such as Scrolls, Books, or Potions. It shows how player actions trigger specific event logic, update EventStatus, and produce gameplay effects, visual feedback, or achievements. For example, the Scroll grants temporary invisibility, Potions apply effects like speed, slowness, or darkness, and the Book triggers the Librarian's reactive behavior. The diagram highlights dependencies between the player and event components, demonstrating how triggers, collisions, and conditional checks coordinate event activation. By keeping these conditional events separate from hidden multi-step events, it ensures consistent handling of game state, player progression, and feedback, satisfying UR_EVENTS, FR_SCORE_CALC, and FR_EVENTS.

**Figure 8:** Pause/Resume Class diagram



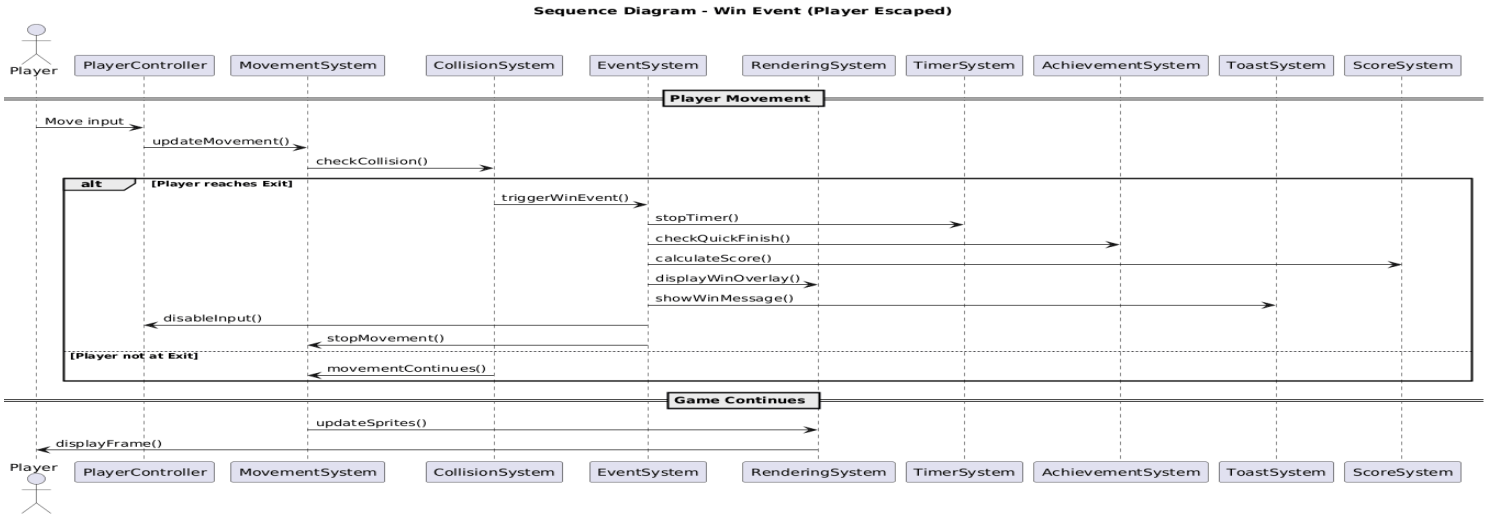

This diagram illustrates how the game implements pause and resume behavior using class-based control. The GameStateController manages the paused state of the game, coordinating the freezing and unfreezing of core entities—Player, Dean, and Librarian. When the player triggers the pause action, all relevant game logic, input handling, and entity updates are temporarily halted. Resuming the game restores normal behavior for all entities and systems. This design ensures that pause/resume functionality is consistently applied across the game, supporting smooth transitions while maintaining reliability and adherence to game rules, addressing requirements: UR_PAUSE, FR_RESUME, FR_TIMER, UR_HOW, UR_EASE, NFR_UI, NFR_RELIABILITY, and FR_RESET.

**Figure 9:** Negative event behavioural sequence diagram




This diagram illustrates what happens when the player is caught by either the Dean or the Librarian. Once detected, the capturing entity freezes the player, teleports them to the prison room if applicable, and applies a time penalty while displaying toast messages describing the event. Captured entities (Dean or Librarian) may also change animation and have their position reset as part of the capture sequence. After the penalty expires, the player and the capturing entity are unfrozen, and patrols are resumed. The diagram also shows how collisions, timers, and negative event counters are coordinated, ensuring consistent and reliable game state management through the ECS systems.

**Figure 10:** Maze Escape sequence diagram




This diagram illustrates a complete successful game run leading to a win event, including hidden event and achievement handling. It reflects the updated game logic where the player triggers the win condition by reaching the exit while holding the exit key. Upon triggering the win event, the timer is stopped, all active game logic is paused, and the player's score is calculated based on the remaining time and modifiers from hidden events or triggered achievements. The diagram also highlights conditional flows, ensuring that win events are correctly applied even if special conditions like invisibility are active. This updated sequence satisfies **UR_ESCAPE**, **UR_SCORE**, **FR_SCORE_CALC**, **FR_INVARIANTS**, **FR_RESET**, and **CR_CHEAT**, while clearly showing how ECS components and systems interact to manage game state, player progression, and feedback display.

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).