Architecture

Cohort 1, Group 9

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

3. Architecture [22 marks]:
Give diagrammatic representations (**structural and behavioural diagrams**) of the architecture of the team's product, with a brief statement of the specific languages (for instance, relevant parts of **UML**) and the tool(s) used to create these representations. (**Gantt UML**) (first sketched in miro then draw.io then written in plantUML)

Include a systematic justification for this architecture and describe how it was initially designed and how it evolved over the course of the project. Provide evidence of the design process followed (e.g. interim versions of architectural diagrams, **CRC cards**) on your team's website and link to them from your report. Relate the architecture clearly to the requirements, using your requirements referencing for identification, and consistent naming of constructs to provide traceability.
(ID codes from the referencing system ie: UR_ESCAPE is the ID for the escape user requirement and when using a diagram to describe the escape classes or methods, use the relevant ID in the justification description).
(22 marks, ≤ 6 pages).

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 webpage.
The majority of 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 modularity and distribution isn't a priority, with maintainability being convenient due to layer isolation and game logic centralised in the business layer.
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 webpage).

## 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.
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 of the ECS structure focused on game logic and component systems, integrating movement, events, scoring and timer (UR_EVENTS, UR_SCORE, FR_SCORE_CALC, FR_TIMER were adequately met).
This embedded ECS structure allows for inner modularity and coupling between packages for extensible game logic and debugging.
The Team opted for a traditional AABB and 'Trigger' based collision system for simplicity and performance between components such as the different events and game logic. (see Design Iterations webpage for collision and Trigger system details).
The games systems such as movement, events, score and timer etc were chosen  the relevant requirement
The chosen Java based game engine was libGDX due to its ongoing support and extensive libraries and ease of asset and tile integration.
Design trade offs: limited communication between layers to prevent sinkhole anit-pattern problem due to too many bypasses between layers, thus we counter this by having components be the indirect link between systems and entities.

Design Principles
Low coupling, high cohesion since each ECS system handles one responsibility
Separation of concerns: layers isolate presentation (UI), game logic and data.
Modularity from open- closed layer principle and OOP nature: new entities and event logic can be added without altering existing systems. (extensibility)
Information hiding: Systems interact via shared components not direct access. (low coupling)

to optimise modularity via effective class-component Cohesion, Coupling and Connascence resulting in the finalised diagrams below.

The Team opted for a traditional AABB and 'Trigger' based collision system for simplicity and performance between components such as the different events and objects (see Design Iterations webpage for details).
These systems along with events, score and timer were chosen to satisfy the relevant requirements

**Introduction/ scope statement**

(introduction to the deliverable briefly outlining which structures and style of architecture the group has decided on and the steps to creating software:

1) System structure (monolithic closed layer)
2) Architecture characteristics (reference the table)
3) Architecture decisions (user and constraint requirements) Show understanding of design trade-offs and key decisions.
Write 5–7 bullet points summarising major design decisions.
Example:
Adopted **closed layered architecture** for maintainability and clarity.
Embedded **ECS pattern** for modular gameplay logic.
Chose **monolithic structure** due to team size and time constraints.
Implemented **AABB collision detection** for performance and simplicity.
Limited inter-layer communication to prevent **sinkhole anti-pattern**.
Chose **Java/LibGDX** as implementation language due to course requirement

4) Design principles (design decisions we made: campus structure, events and enemy interaction etc)
Link your design approach to software-engineering principles.

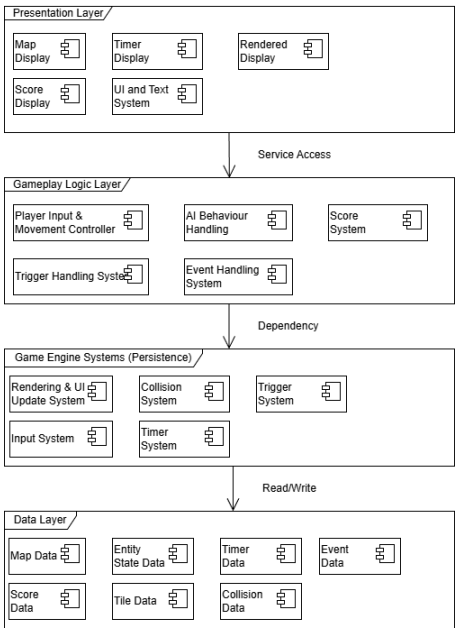
Write a short paragraph or 3–4 bullets summarising design principles.
**Design Principles:**

- **Low coupling, high cohesion:** Each ECS system performs one function independently.
- **Information hiding:** Systems interact through shared component data, not direct calls.
- **Separation of concerns:** Each layer handles distinct responsibilities.
- **Open–closed principle:** Systems can be extended with new behaviours without rewriting existing ones.

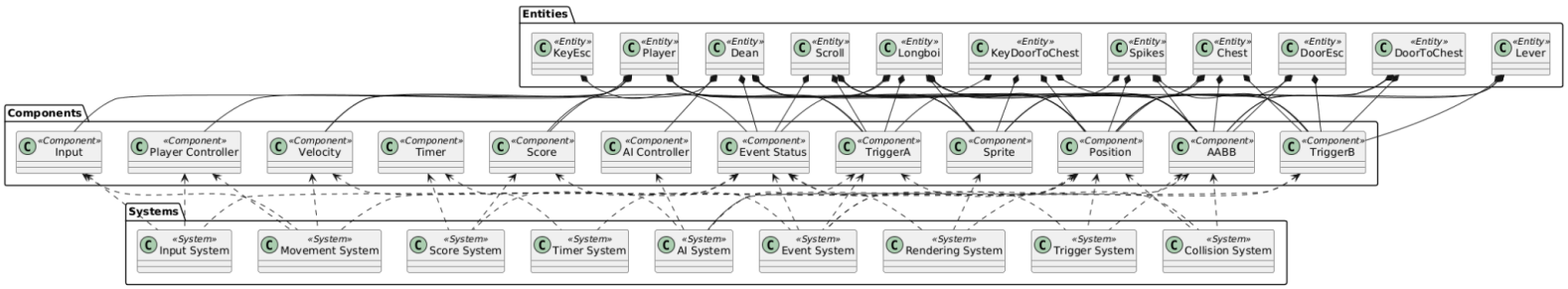
Example Scope statement/intro:

**Scope Statement:**

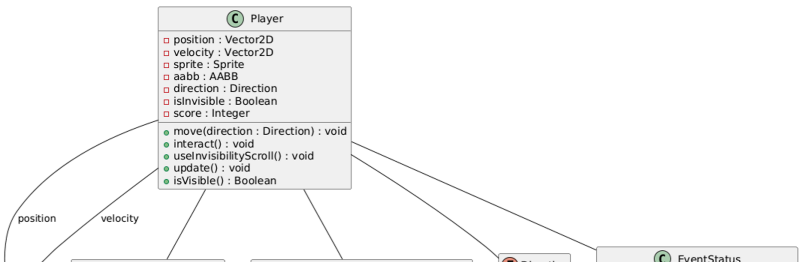
This deliverable models the architecture of a 2D maze-based game built within a **monolithic closed layered structure**. The system applies an **Entity–Component–System (ECS)** pattern for the game's core logic, including movement, collisions, scoring, and event handling. The architecture also defines separate **Presentation**, **Business Logic**, **Persistence**, and **Data** layers to ensure maintainability, modularity, and separation of concerns. Rendering of sprites and user interface elements is managed through the Presentation Layer, while game entities, components, and systems form the core of the Business Logic Layer.



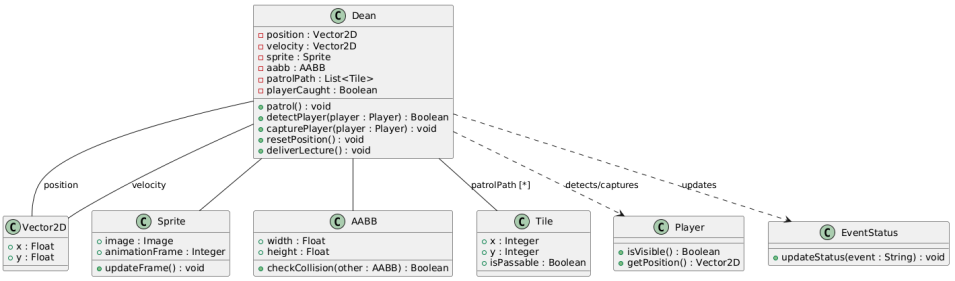
**ECS Class Diagram - Entity-Component-System Architecture**



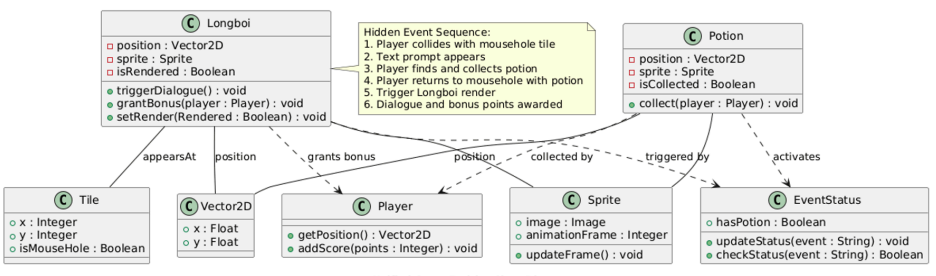

**Player Class Diagram**



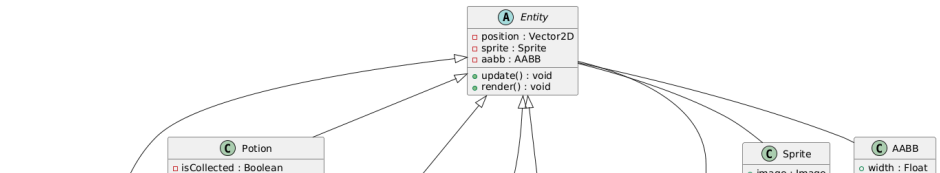

**Dean Class Diagram**

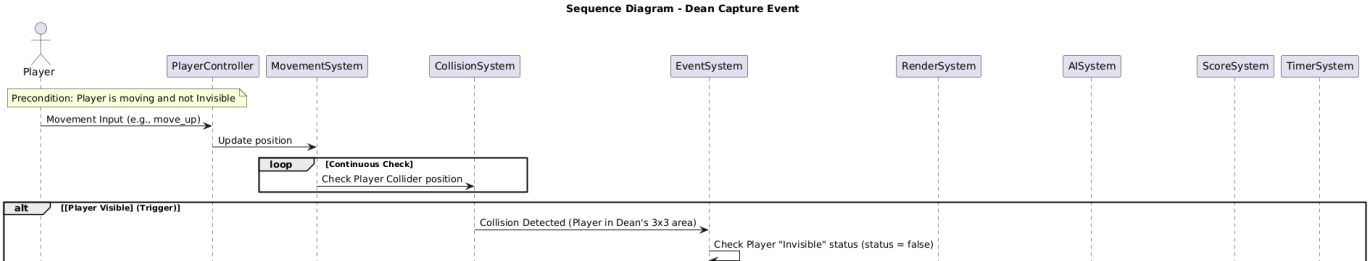


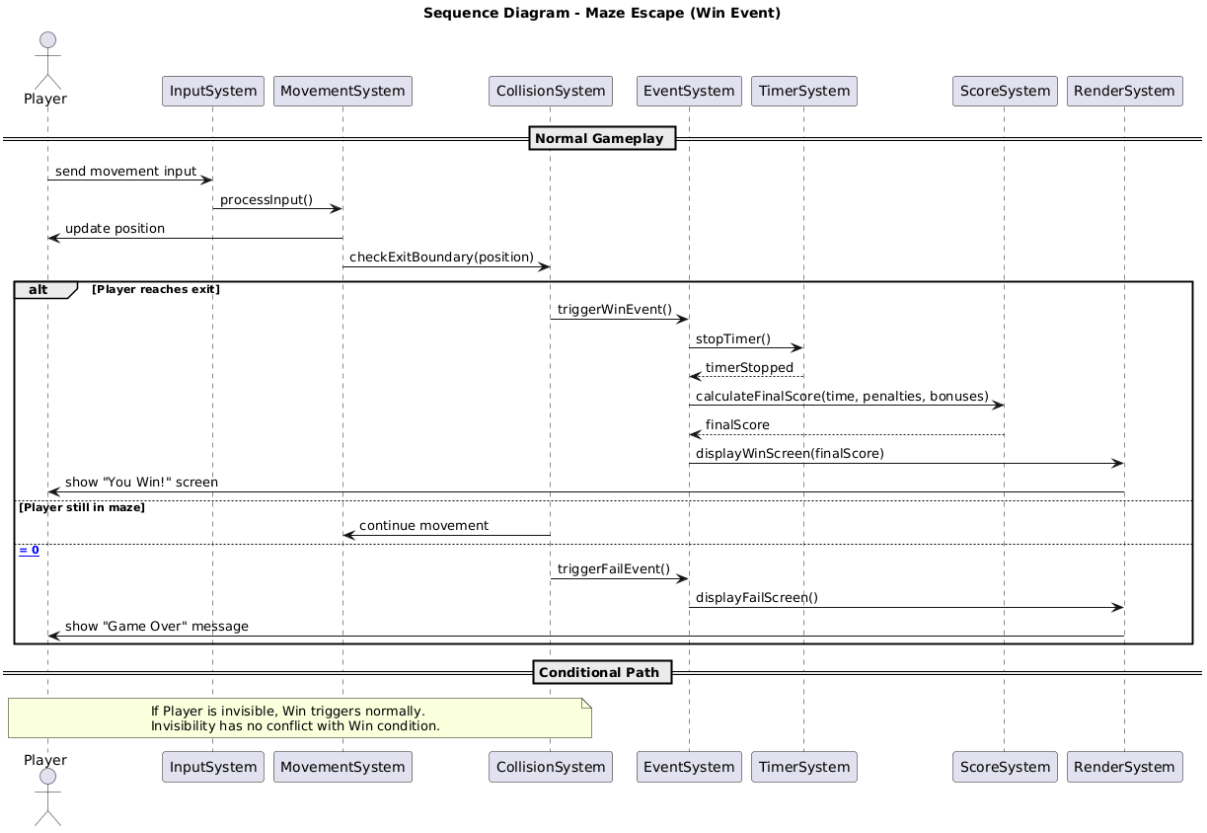

**Longboi Hidden Event Class Diagram**

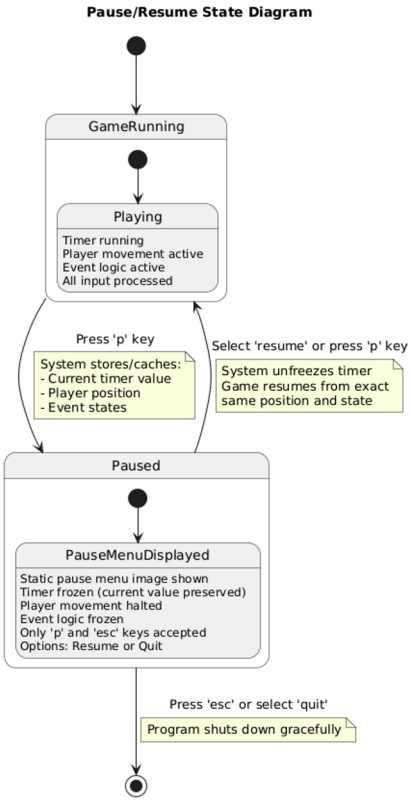


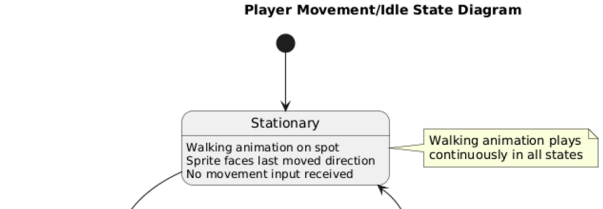

**Unified Game Entities Class Diagram**




Sequence Diagram - Dean Capture Event
Player
PlayerController
MovementSystem
CollisionSystem
EventSystem
RenderSystem
AISystem
ScoreSystem
TimerSystem
Precondition: Player is moving and not Invisible
Movement Input (e.g., move_up)
Update position
loop
[Continuous Check]
Check Player Collider position
alt
[[Player Visible] (Trigger)]
Collision Detected (Player in Dean's 3x3 area)
Check Player "Invisible" status (status = false)


Sequence Diagram - Maze Escape (Win Event)
Player
InputSystem
MovementSystem
CollisionSystem
EventSystem
TimerSystem
ScoreSystem
RenderSystem
Normal Gameplay
send movement input
processInput()
update position
checkExitBoundary(position)
alt
[Player reaches exit]
triggerWinEvent()
stopTimer()
timerStopped
calculateFinalScore(time, penalties, bonuses)
finalScore
displayWinScreen(finalScore)
show "You Win!" screen
[Player still in maze]
continue movement
= 0
triggerFailEvent()
displayFailScreen()
show "Game Over" message
Conditional Path
If Player is invisible, Win triggers normally.
Invisibility has no conflict with Win condition.


Pause/Resume State Diagram
GameRunning
Playing
Timer running
Player movement active
Event logic active
All input processed
Press 'p' key
System stores/caches:
- Current timer value
- Player position
- Event states
Select 'resume' or press 'p' key
System unfreezes timer
Game resumes from exact
same position and state
Paused
PauseMenuDisplayed
Static pause menu image shown
Timer frozen (current value preserved)
Player movement halted
Event logic frozen
Only 'p' and 'esc' keys accepted
Options: Resume or Quit
Press 'esc' or select 'quit'
Program shuts down gracefully


Player Movement/Idle State Diagram
Stationary
Walking animation on spot
Sprite faces last moved direction
No movement input received
Walking animation plays
continuously in all states

References

International Organization for Standardization (2017) *ISO/IEC/IEEE 12207:2017 – Systems and software engineering — Software life cycle processes*. Geneva: ISO.