Design Iterations

Cohort 1, Group 9

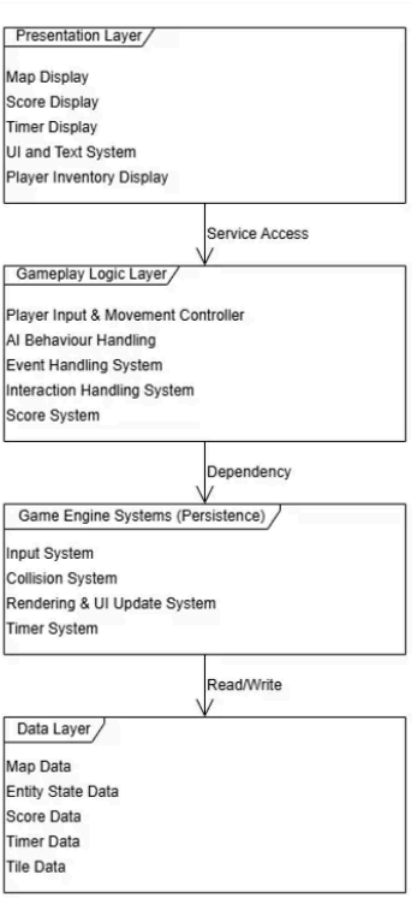
Chris Sewell,
Fedor Kurochkin,
Matt Durham,
Max Peterson,
Oladapo Olaniran,
Wojtek Tomaszewski,
Yuqi Fu.

This document serves as supporting evidence of iteration history for the Architecture deliverable.
Throughout the project the team closely followed the 5 steps of software development:

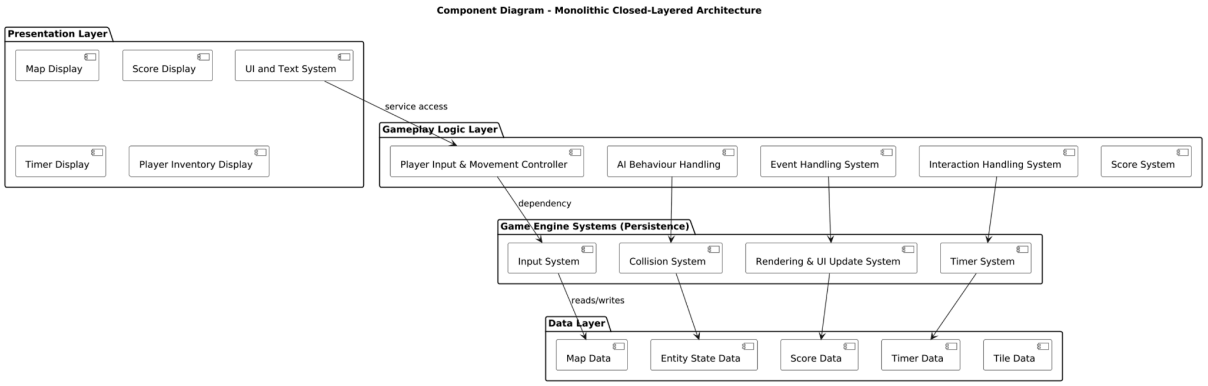
1) Identifying initial components
2) Assigning requirements to components identified
3) Analyse component roles and responsibilities
4) Analysis of component characteristics
5) Restructure components accordingly if necessary

However, before this process the architecture team reviewed the user requirements and product brief along with the relevant lecture material (reference to architecture lecture slides here) to decide on the most suitable architectural style.
It was clear that since the product is a 2D game developed in a Java environment (libGDX) that the ideal structure would be a monolithic layered architecture and so the team discussed how the different layers would be applied to our maze game.

For step 1of the development process, the team met regularly on discord to discuss what the product components should include based on the user requirements and client interview feedback.
Initial drafts and sketches were implemented in Miro (see below) to determine what classes and components the game would need and which layers they belong to.
Alongside this Initial storming, team members took inspiration from the requirements referencing table and constructed a table of characteristics to address step 1 and 2 directly by incorporating the requirements IDs into the table for a clear trackable representation that could be referred to throughout the project lifecycle.
(See Characteristics table on the website).

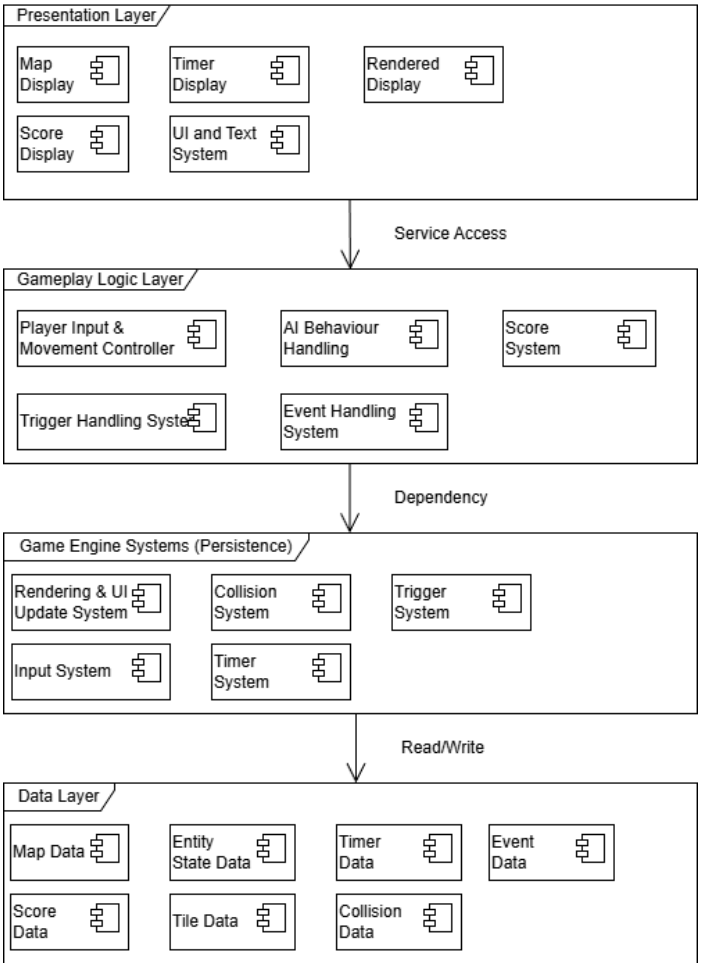


The monolithic layered structure was further refined into the UML based structures to the right and below after reviewing the characteristics table.



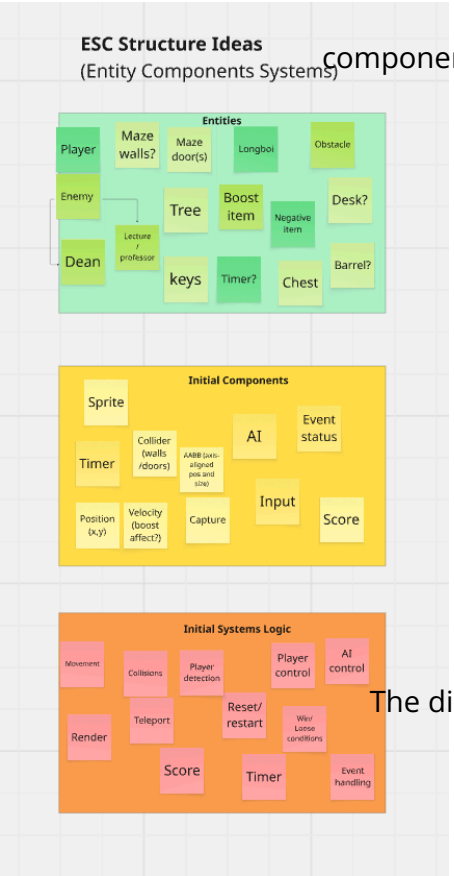
Later in the project lifecycle after use cases were updated and tested it was clear certain event systems and component relationships needed updating and the resulting component relationship structure below was settled on.

Final Iteration of the Monolithic closed-layered Architecture diagram.



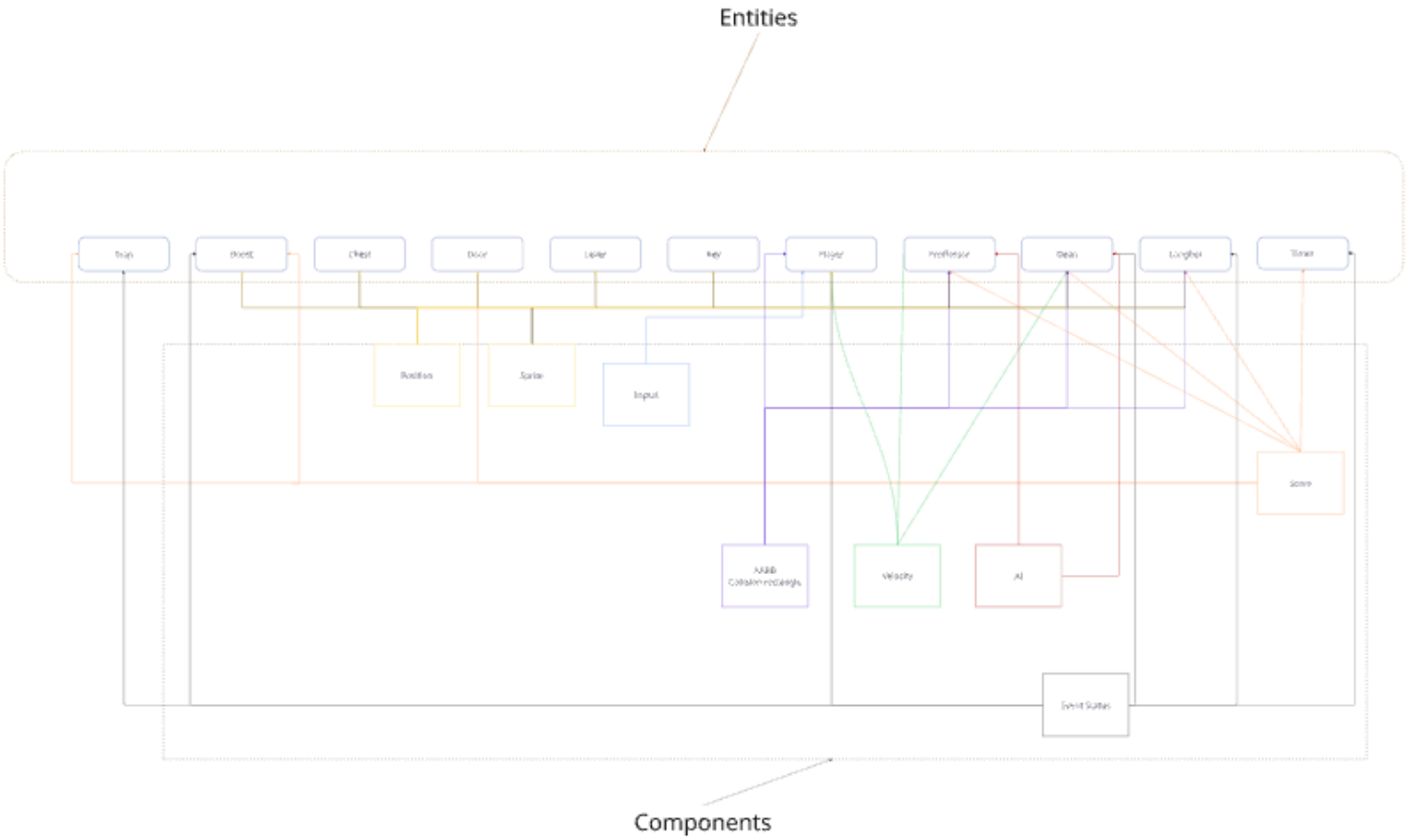
Alongside the layered structure hierarchy, the team simultaneously worked on a class-component-system structure to describe how the business layer (game logic) might behave.
The 5 steps were repeated and the characteristics were updated after new use cases were constructed and tested (see use case document on the website) and the initial components for the business layer were stormed below.

Entity Component System Class diagram iterations



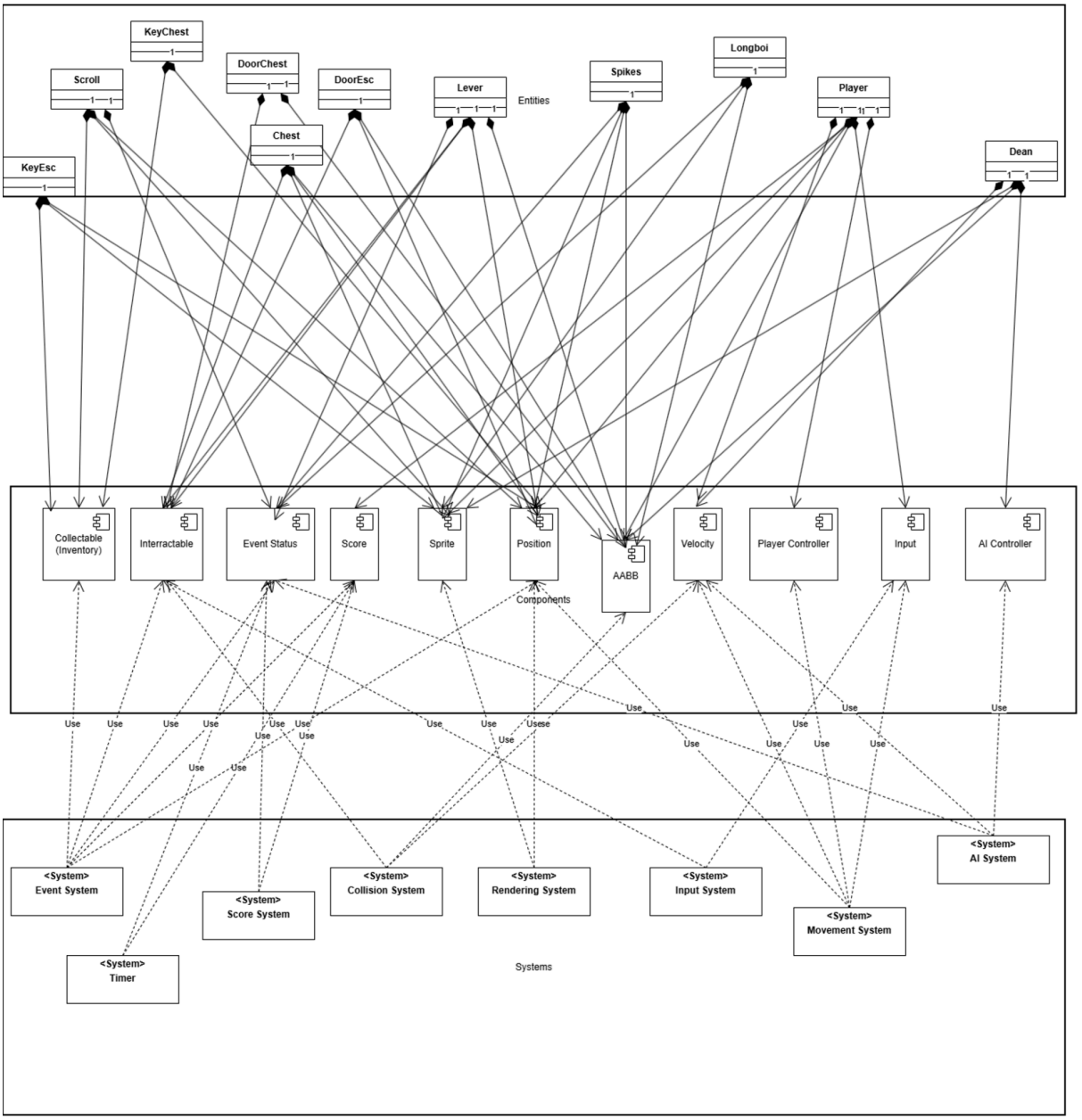
These sketches of the ECS diagram simply list the initial components needed and how they relate to each other within each package.

The diagram below is the first iteration, simply detailing entities and their dependencies/ ownership of components to be included.



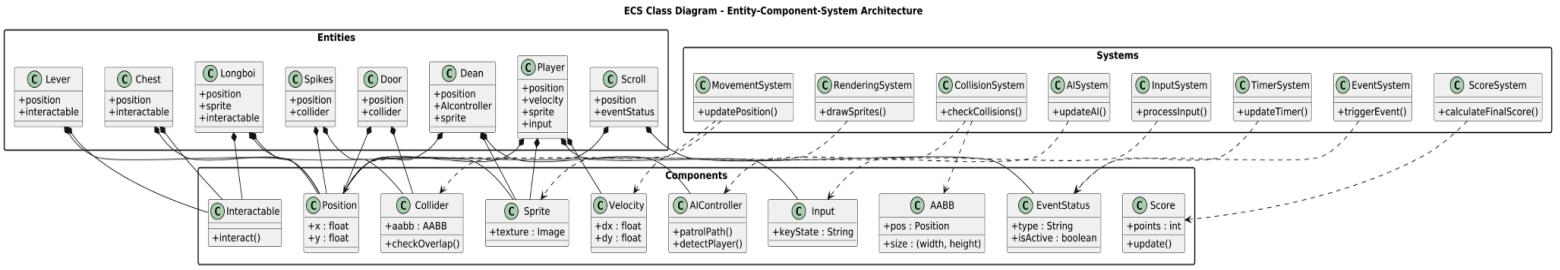
For iteration 2, the components and entities were refined after reviewing the use cases and feedback from the programming team, and new ones added to better suit the appropriate characteristics.
Systems were also added to describe how they work with each component forming a full diagrammatic representation of the game logic layer.

Iteration 2: ECS UML diagram made in Miro

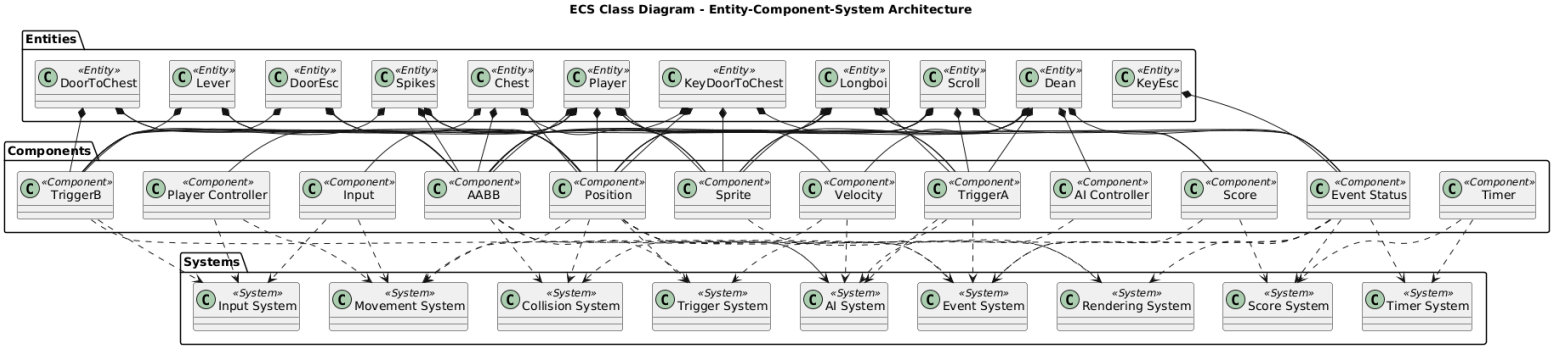


The final iterations were made after further updating the components following the 5 steps again following feedback from the developers and player testing.
These were tested each time in plantUML text until the diagram accurately represented the relationships between packages and components.

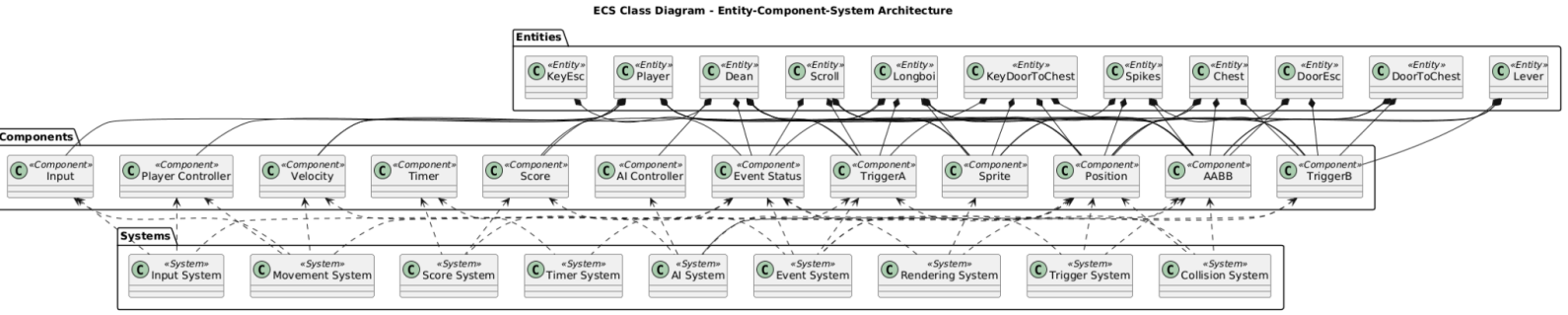
Iteration 3 with container relationship arrows



Iteration 4 with clearer package structure (Entities package above Components package above Systems package better representing the architecture hierarchy)

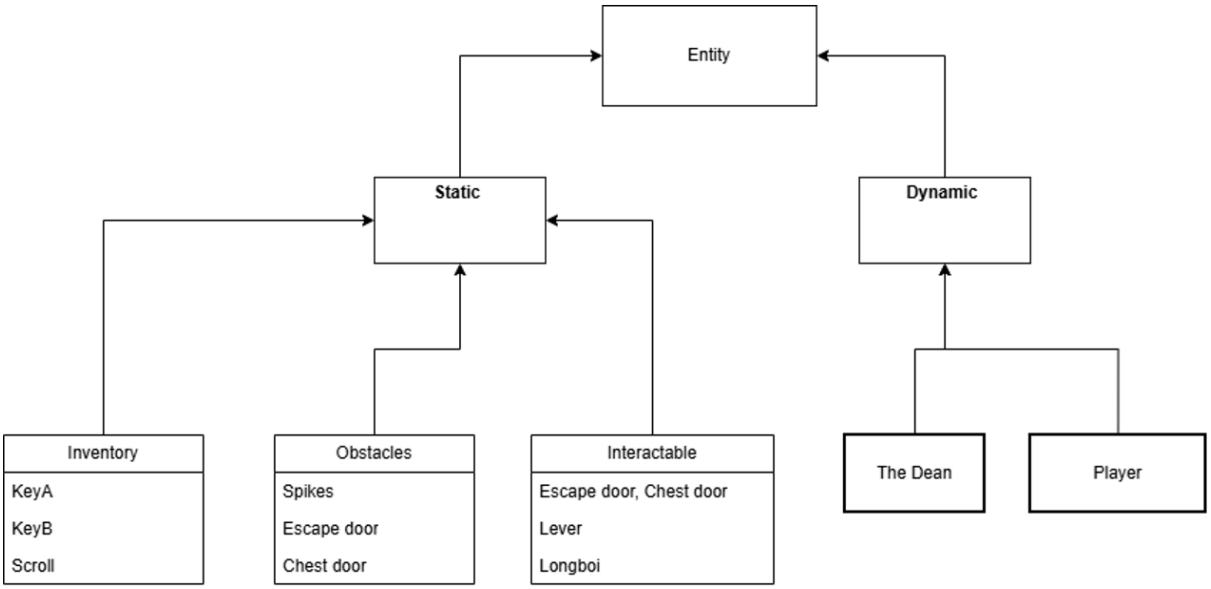


Final iteration for clearest representation of inter package relationships

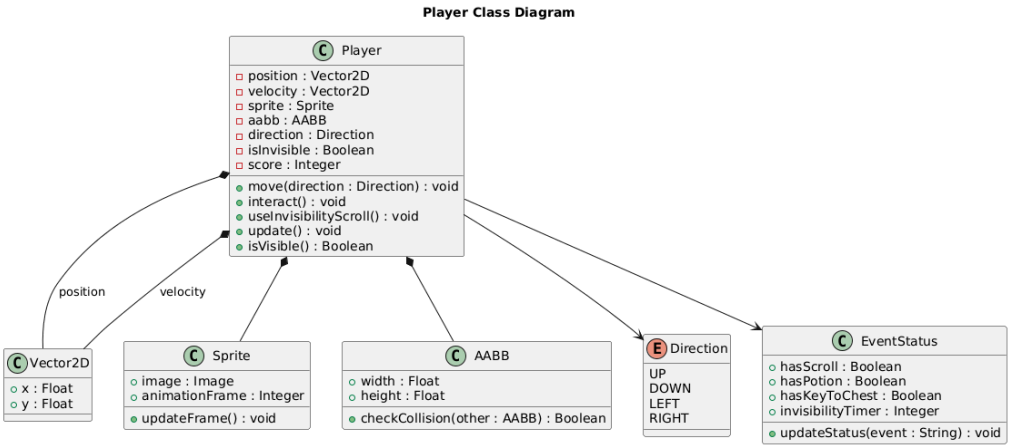


Class diagram iterations

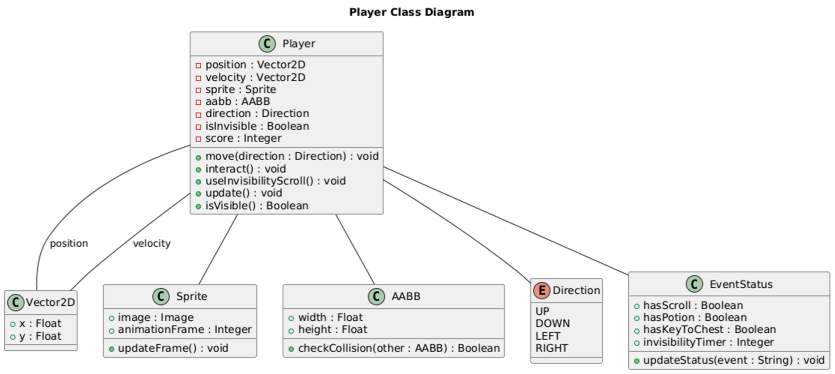
Simple draft for understanding class - component relationships before designing class diagrams.



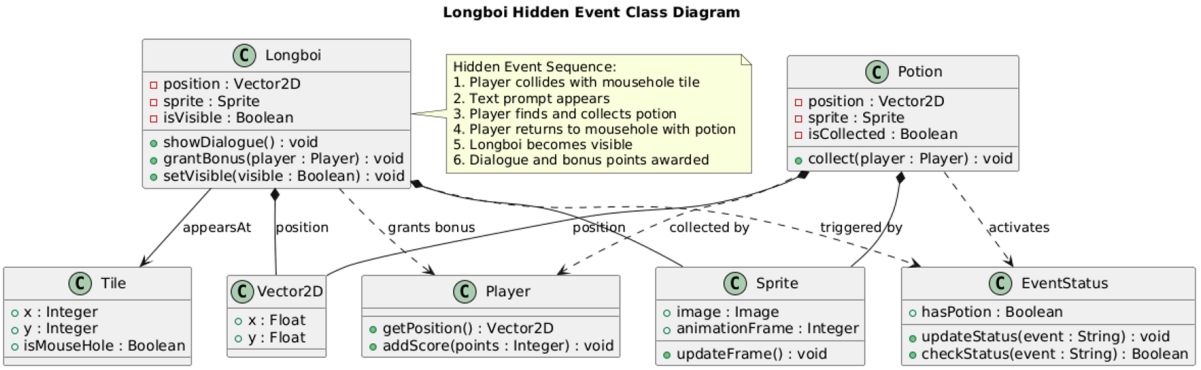
The i**nitial class diagram for the player** (below) was designed to include the new event systems after reviewing the new use cases (see use case webpage).



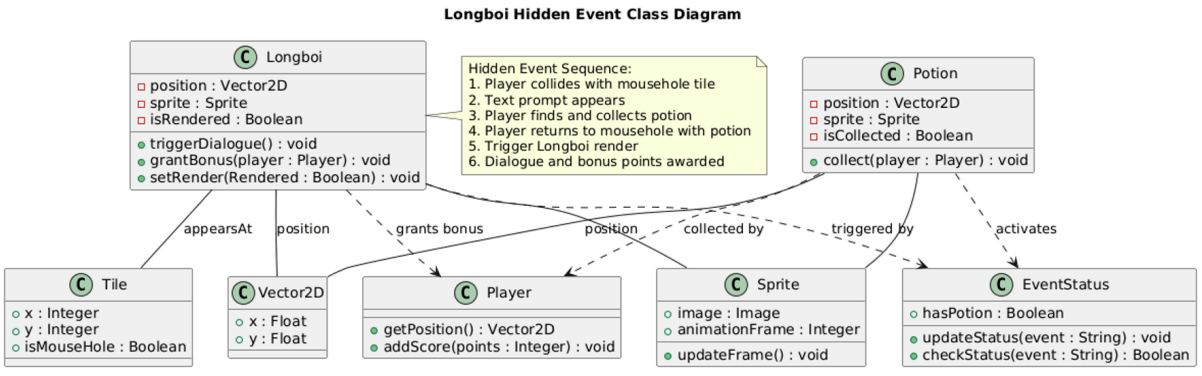
Final iteration of **Player class** to remove contains diamond arrows since the sprite and vectors are not dependent upon player class.
(Note this was applied to all three class diagrams).



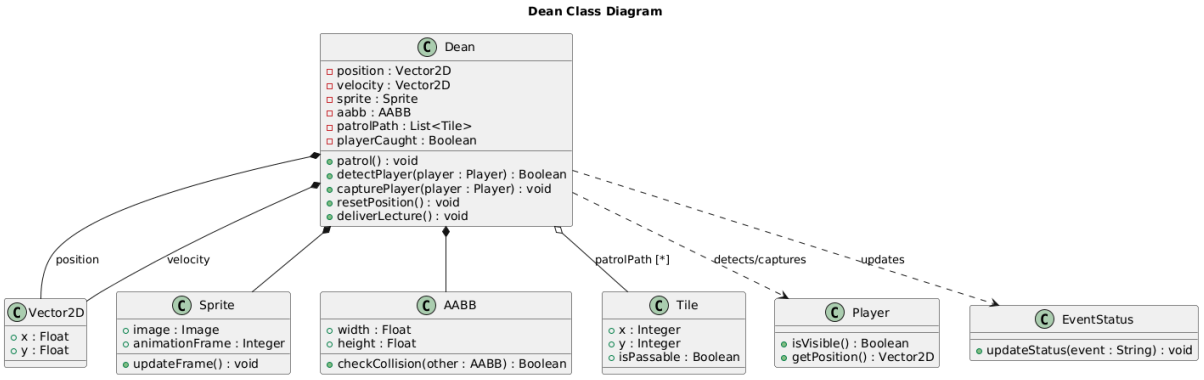
Initial iteration of **Longboi class** and hidden event system described using class relationships and dependencies (and triggers).



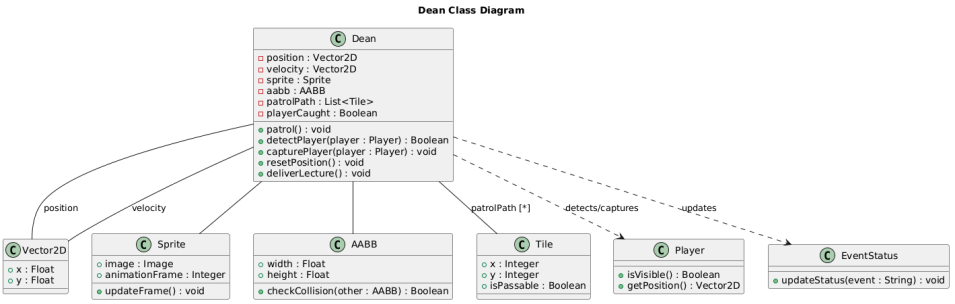
Final iteration of **Longboi class** and hidden event logic.
(See Trigger system tile map for more details on Trigger logic at the end of the document)



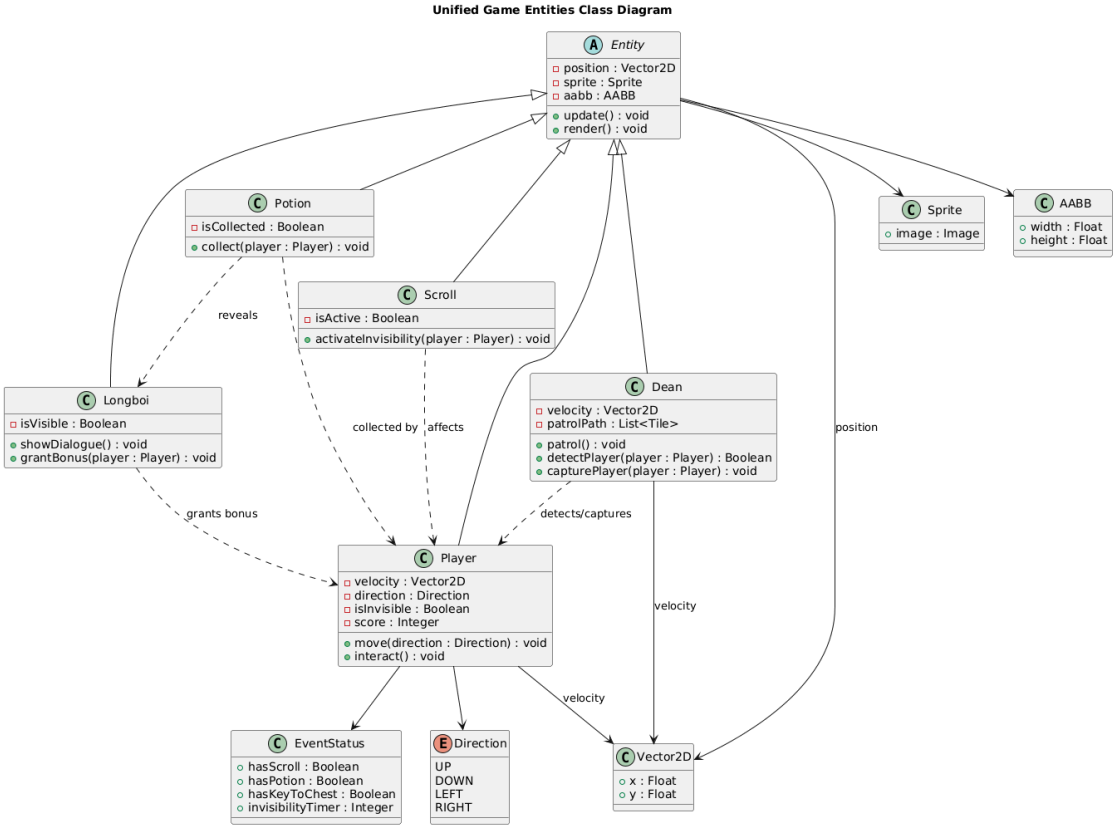
Initial iteration of Dean class and negative event logic.



Final iteration of **Dean class** and negative event logic



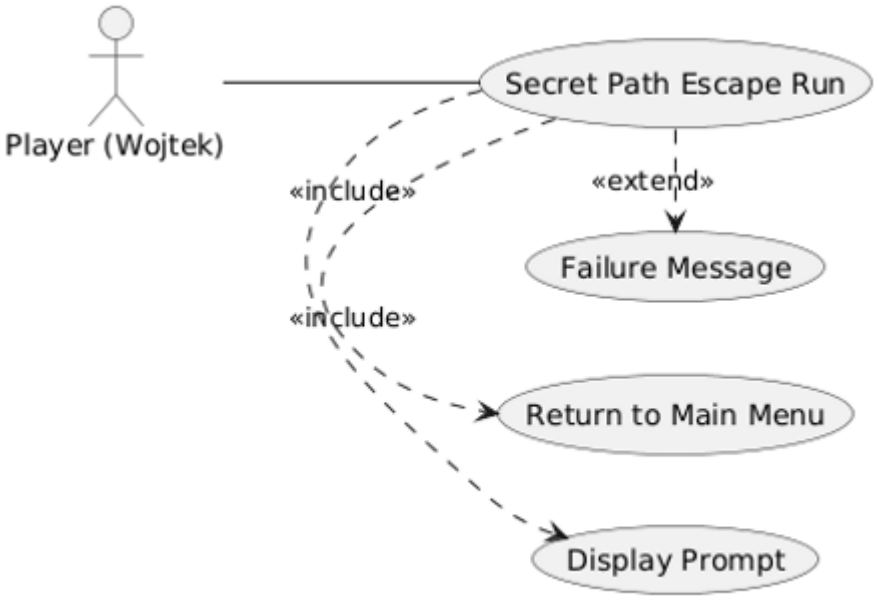
Unified class diagram describing inter-class relationships (event triggers and collision relationships)



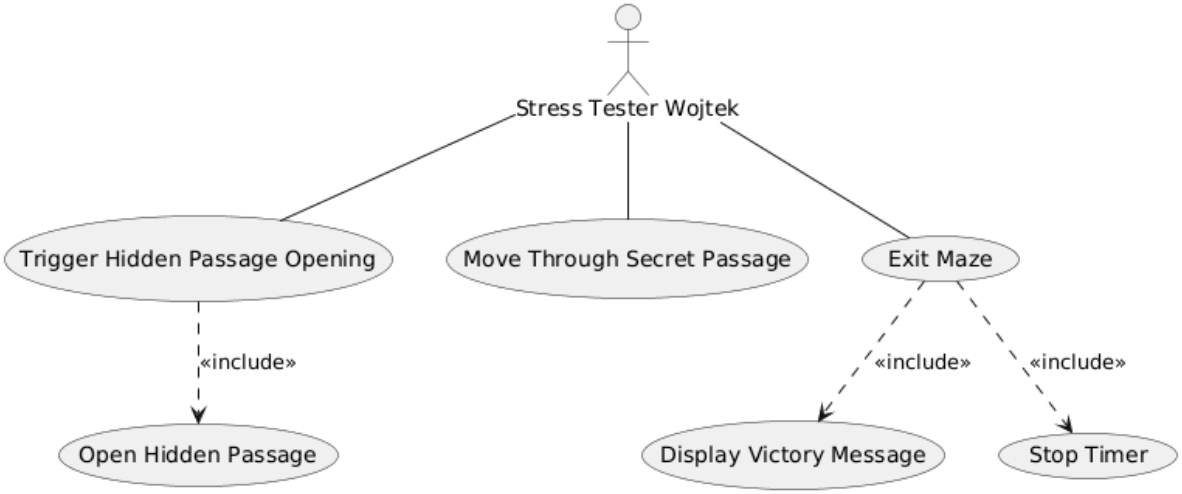
The above diagram illustrates how the classes described above behave together within the game architecture.
The diagram demonstrates how event key objects such as Scroll, Longboi and Potion use methods and associations to pass event logic between layers without requiring duplicate classes for each keeping coupling and modularity balanced whilst having appropriate package visibility for sensitive attributes like collision data and position (satisfying CR_FORBIDDEN, FR_EVENTS)

The following use case diagram iterations were tested in plantUML and refined to better represent the use case scenario descriptions.
(see Use Cases webpage)

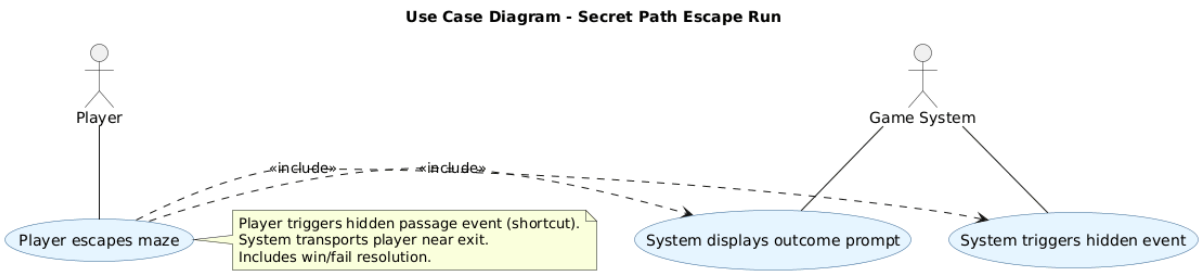
Use case iterations:
Scenario 1 (Escape via hidden event)
Iteration 1:
Simple initial escape use case diagram with just the prompt.



Iteration 2:
Improved by adding branches for using the hidden event to escape.

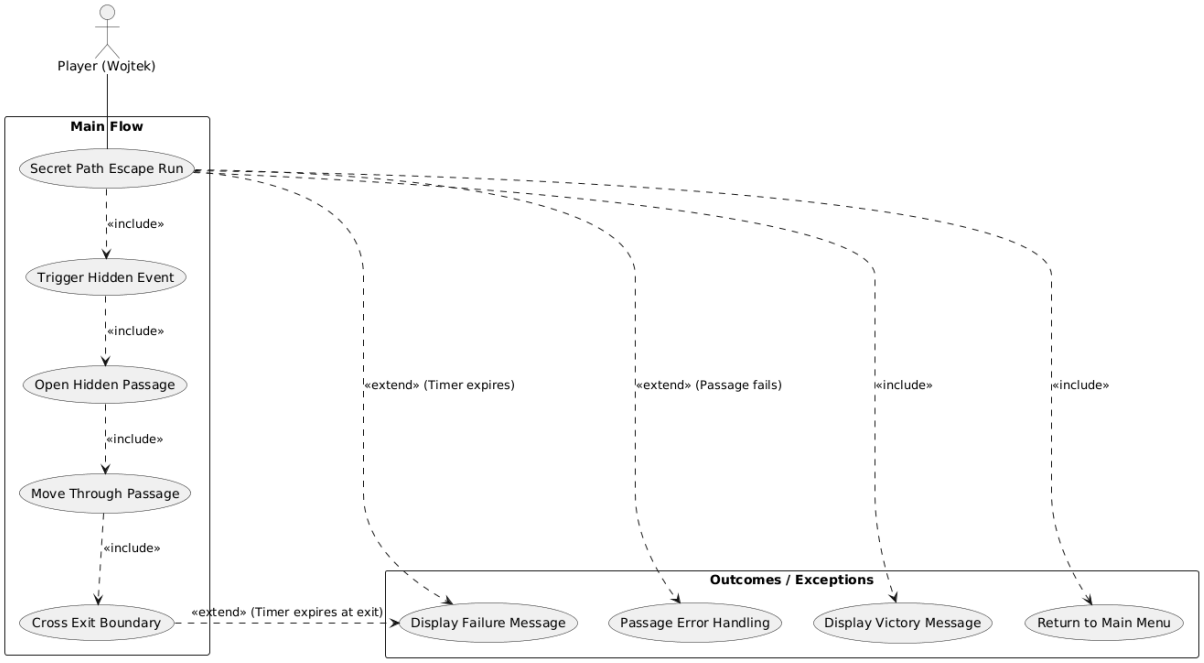


Iteration 3:
Another attempt to describe the player escape using UML, however over simplified and doesn't address edge cases.



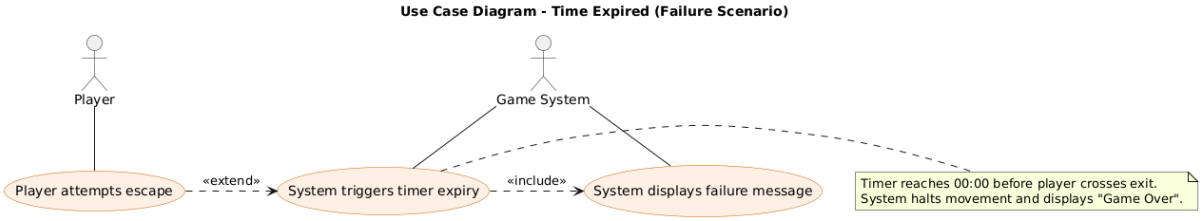
Iteration 4:

Handles errors and edge cases for the hidden passage event.



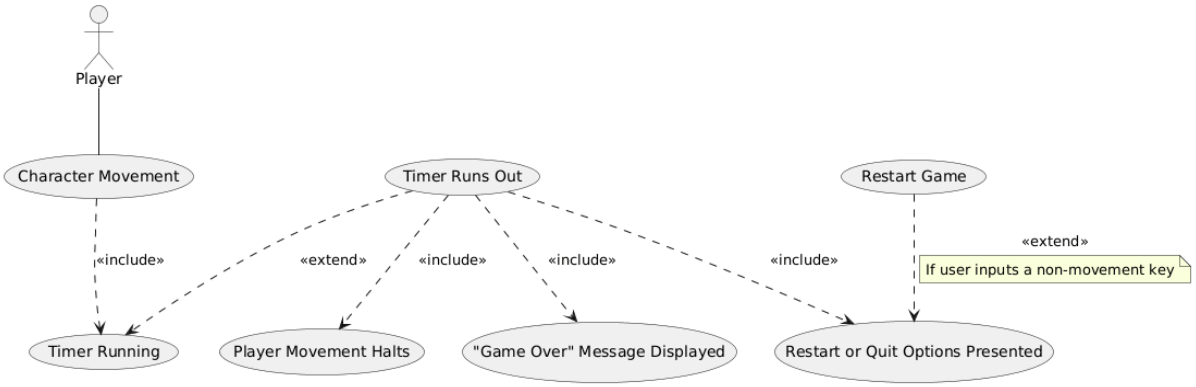
Scenario 2 (timer expires- fail state)

Iteration 1:

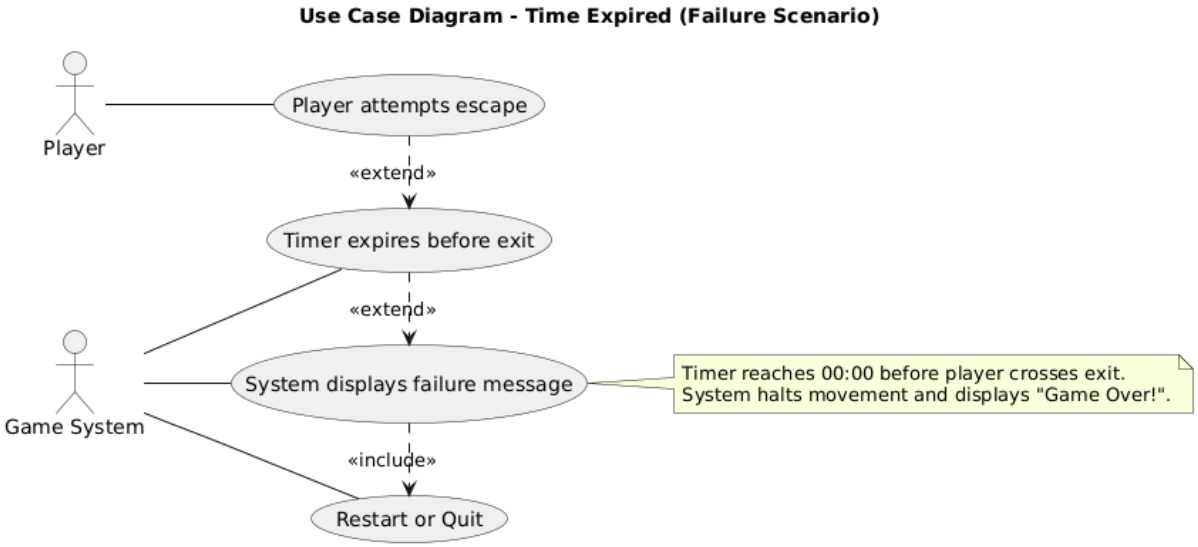


Iteration 2:

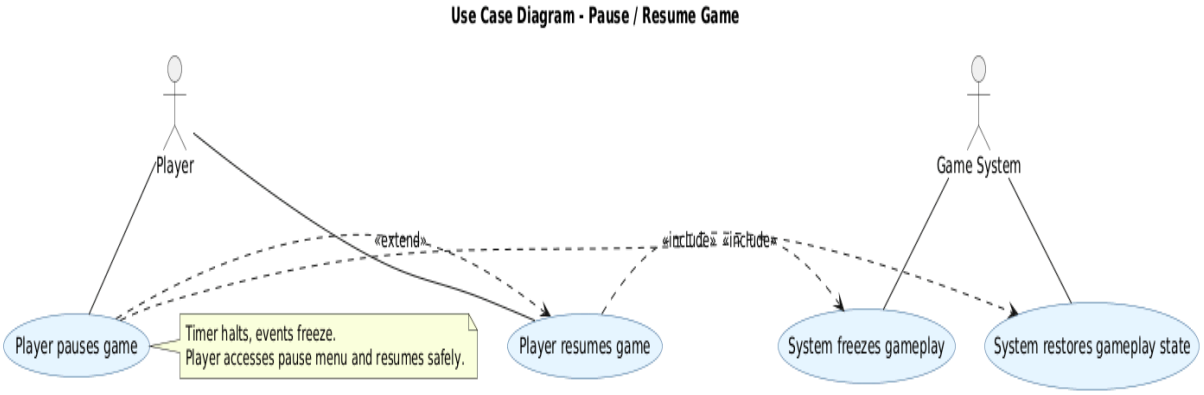
Describes the full process of the timer system when expired and how the player is prompted.



Iteration 3:
This iteration only improves on the previous one with the implementation of the game system as a separate ‘user’ to better represent the interaction between the player and system.



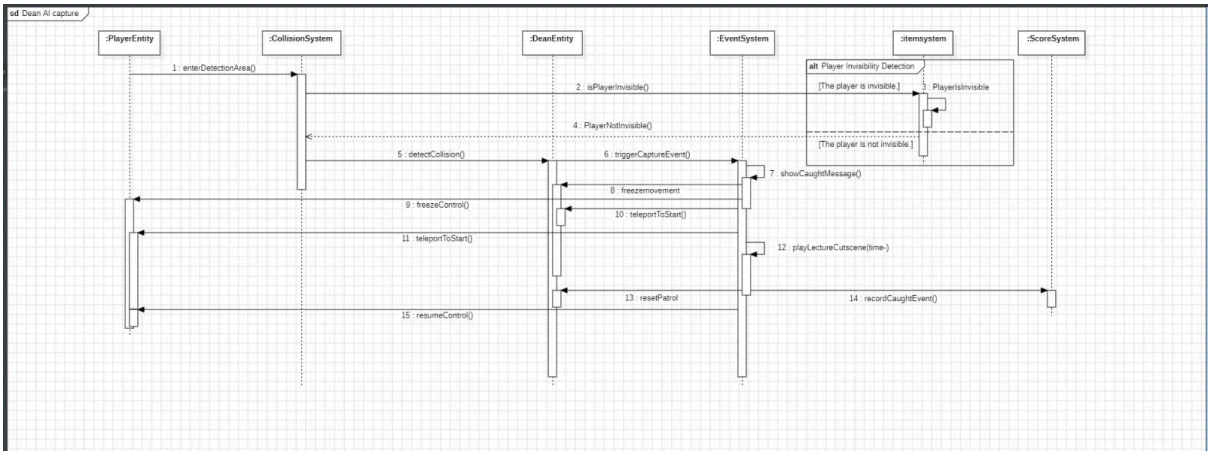
Scenario 3 (pause/resume game)
Iteration 1:
Describes the pause feature but the diagram is not clear and doesn’t describe system state after resuming.



Behavioural sequence diagram iterations for Dean capture event (negative event)

Dean AI capture (negative event) sequence.

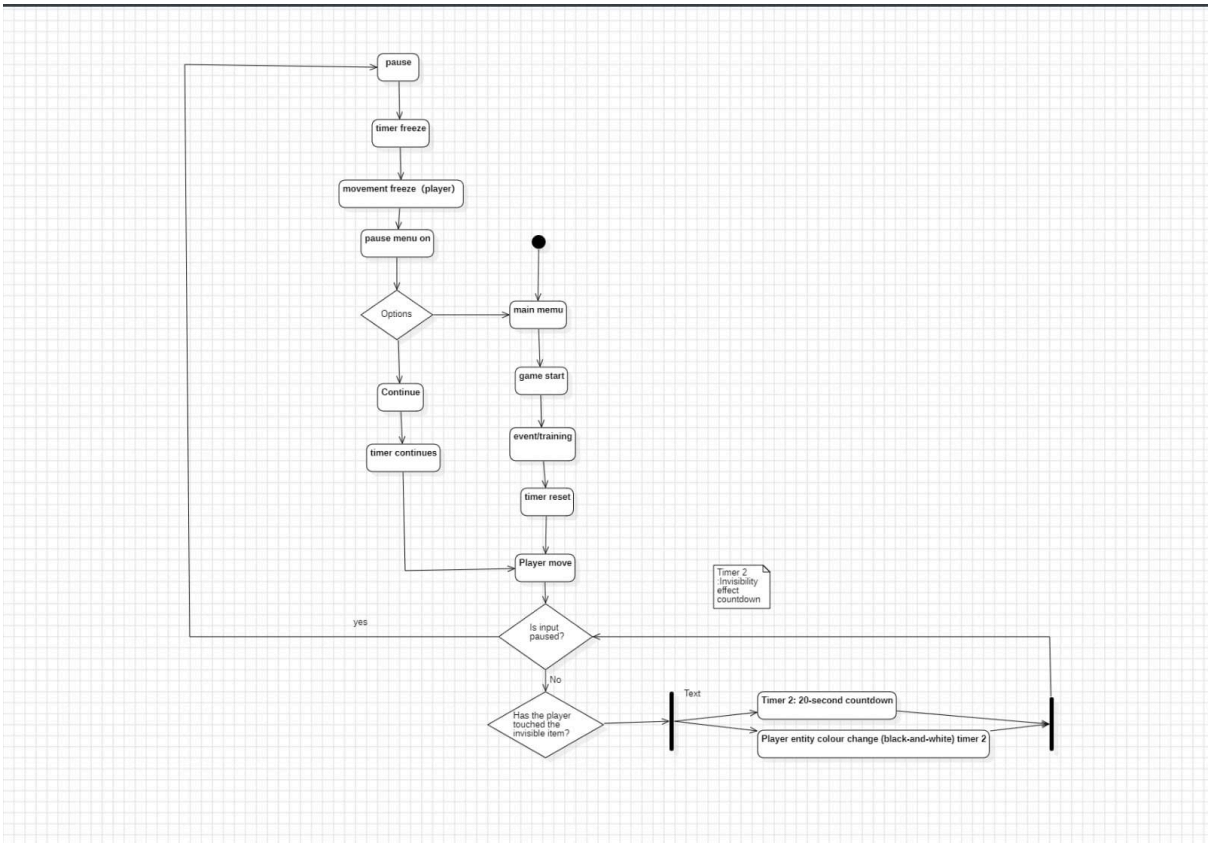
Iteration 1:



Behavioural state diagram for pause feature:

Iteration 1:

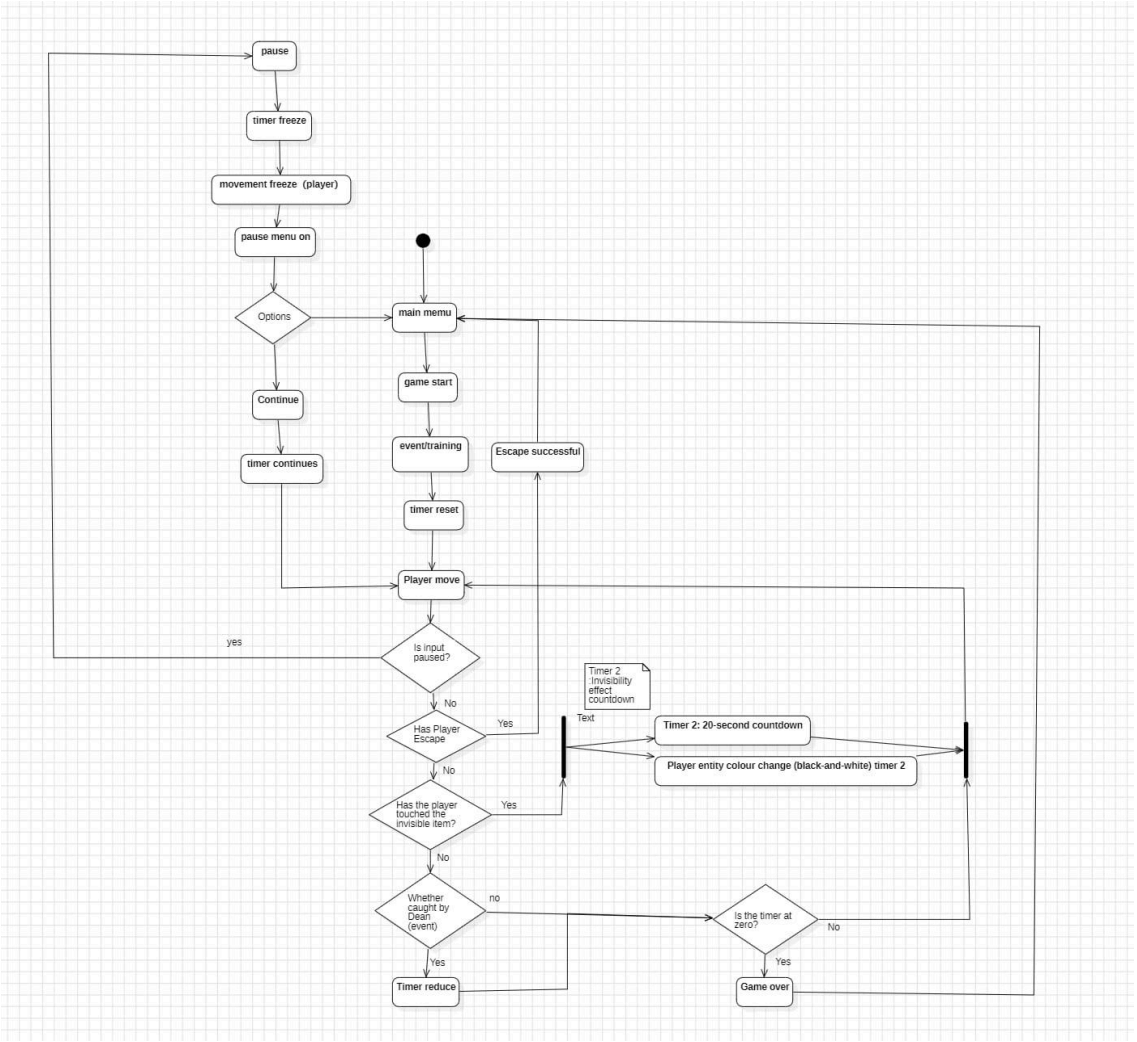
This iteration serves as an initial sketch (syntactically not UML complete)

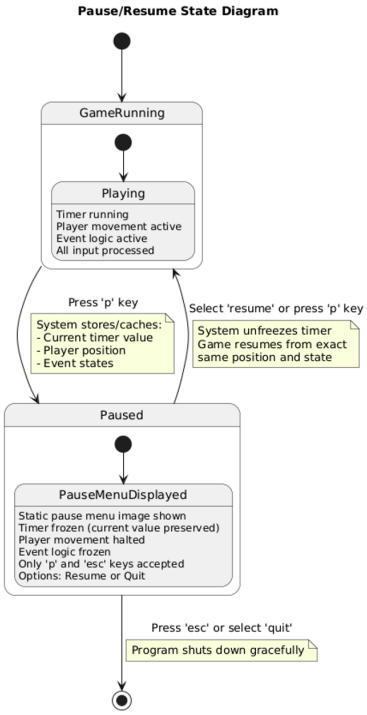


Iteration 2

This sketch describes the game state before and after the player has activated the pause feature.

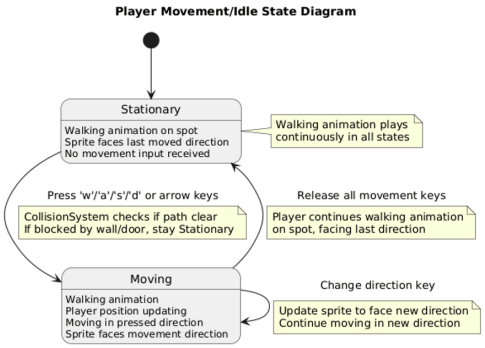
It handles the different components and systems affected by the pause including the timer and how the game state is expected to behave before and after.





Final iteration:

This version is more compact and in correct UML syntax.
This diagram clearly illustrates the pause resume feature along with the pause screen details including the game instructions satisfying: UR_PAUSE, UR_ACCESSIBILITY, FR_RESET(‘quit’ key), and NFR_UI.

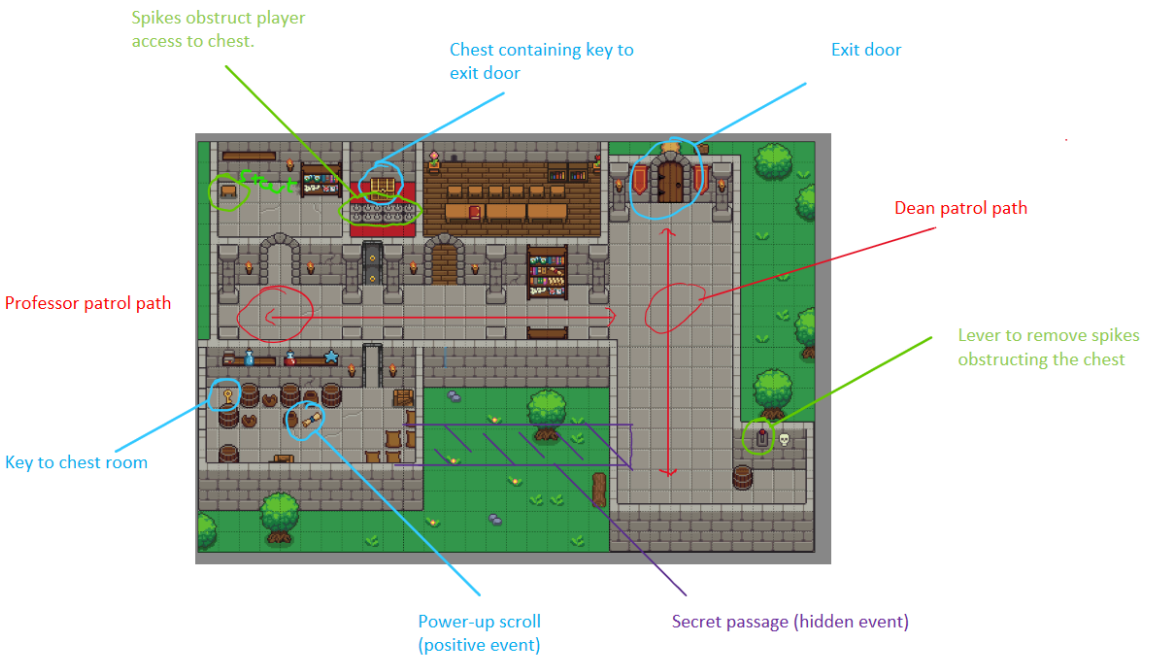


Player movement/idle state diagram

This diagram was adapted from the player and movement classes. It illustrates the different game states the player class and movement system can be in depending on player input commands. It shows the default state (walking on the spot animation) and the different directional variations of the sprite depending on the player input.

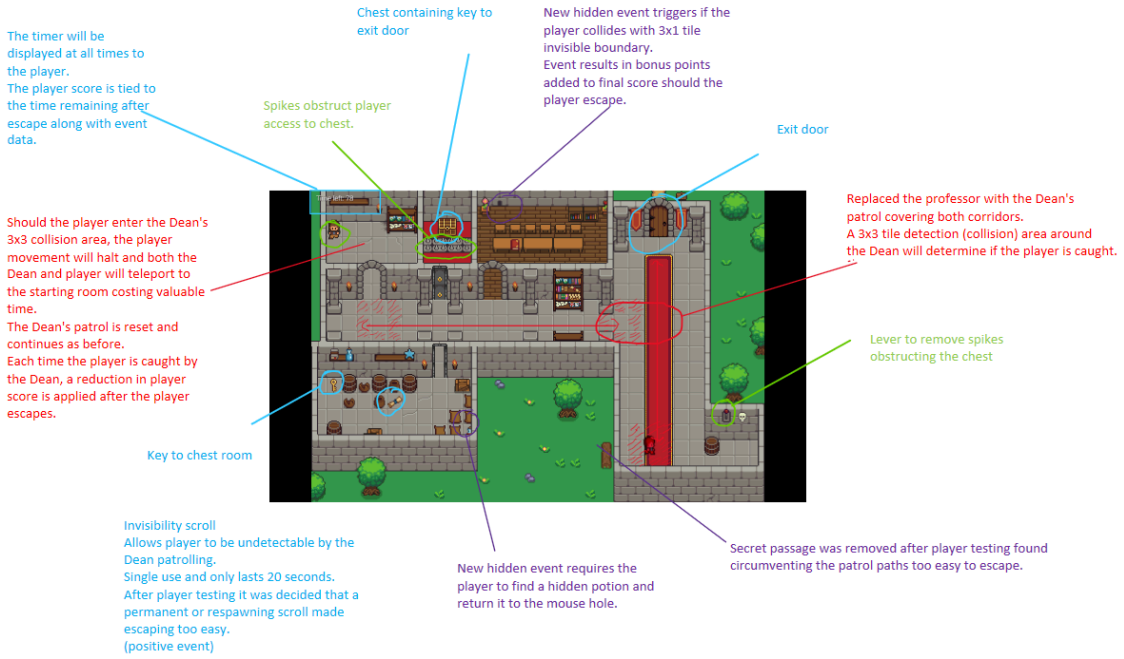
Map and game logic iterations:

Initial Map design after reviewing user, functional and non-functional requirements:

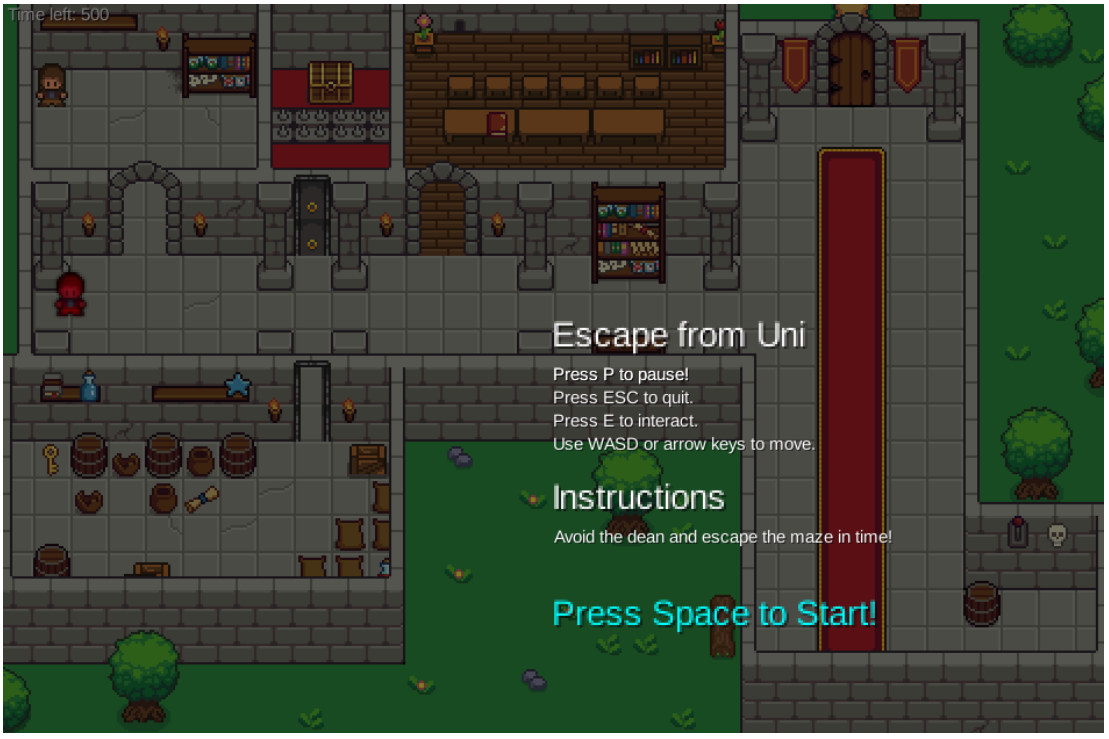


After user testing, feedback from the development team and reviewing the use case scenarios it was clear updates to the map design and event system implementation were needed.
The team returned to the requirements referencing system and Architectural systems table to decide how to refactor the map, escape process and implement new event systems.

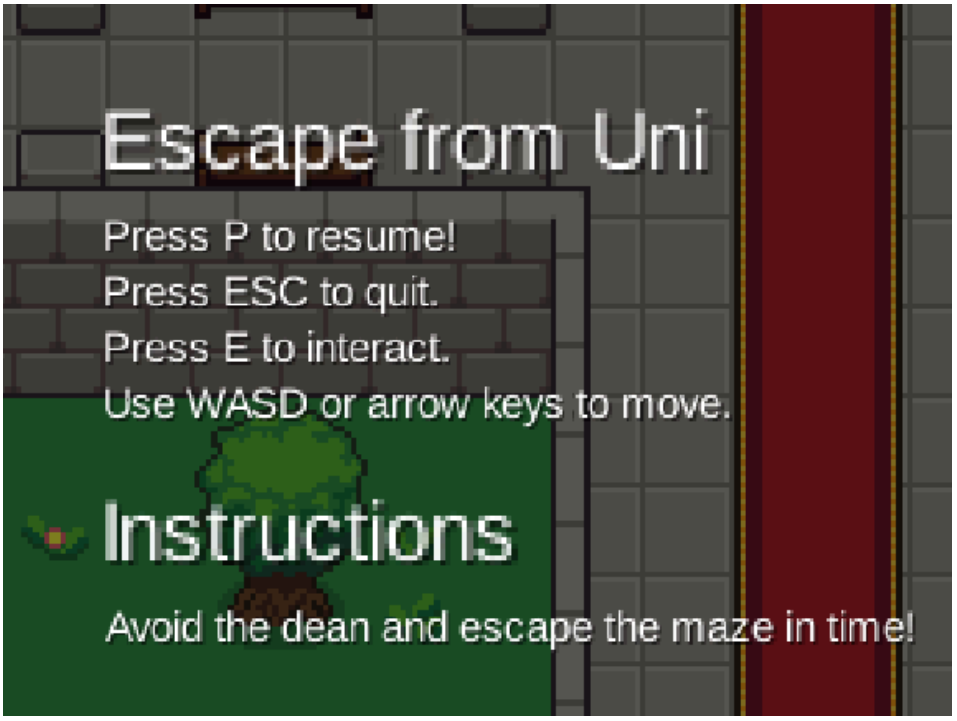
New use cases were tested and the following updated map and event systems were implemented:



Title / start screen menu with simple instructions displayed.



Pause screen containing simple instructions and resume/quit options.



Win-state screen with the player score displayed and option to quit (closes program gracefully)

Fail-state triggered when timer expires. The system effectively halts like pause, until the player quits. No score is displayed as it is predicated on time remaining upon escaping the maze.

Collision system tiles.

Trigger system tiles

(TriggerA activates a component/system when player collides with the blue trigger tile)
(TriggerB activates a component/system when player uses interaction key within the trigger tile)

See Class diagrams above for more details on Trigger System logic.