Architecture

Cohort 3 Team 4

Kiran Kang

Hannah Rooke

Ben Slater

Abualhassan Alrady

Cassie Dalrymple

Charles MacLeod

Dash Ratcliffe

Harley Donger

Class diagram

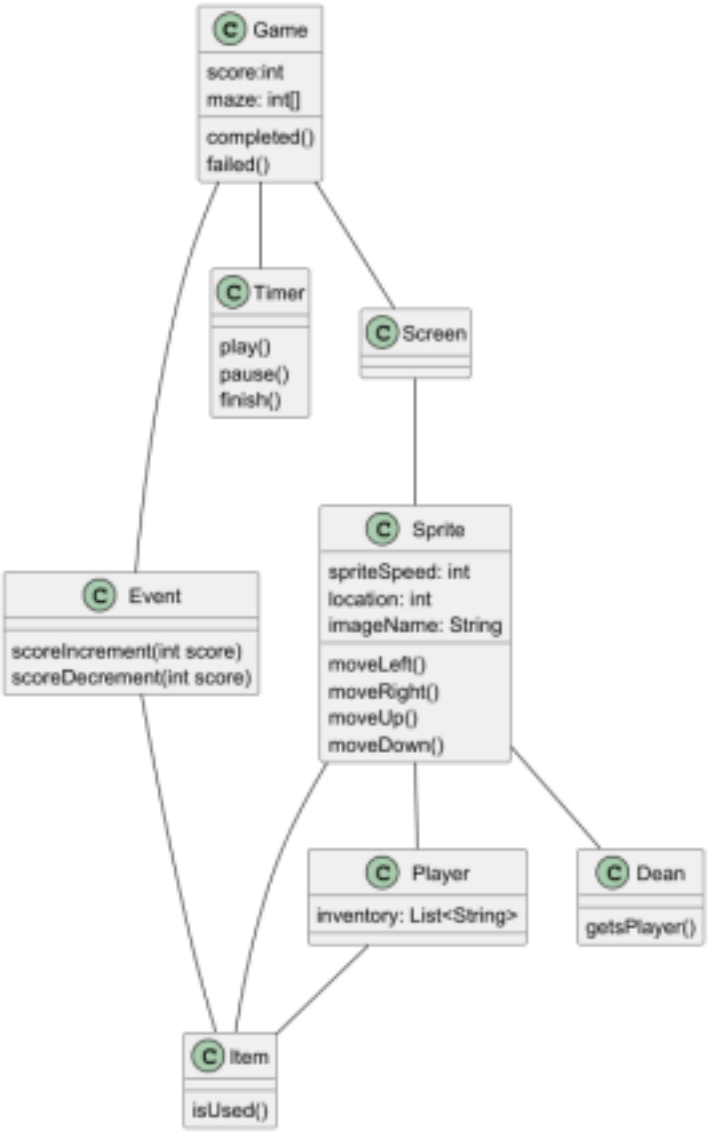


Fig 3a: the final class diagram

The above class diagram is a structural representation of the architecture of the game design. This was chosen as the game is coded using an object-oriented paradigm. A class diagram allows the relationships between the classes to be shown in a clear, brief overview of the implementation. This is a very abstract diagram, ignoring many details of implementation, allowing the main aspects of the architecture to be presented and understood with clarity.
Note: class diagrams were created in UML with PlantUML.

Fig 3a shows the main classes implemented. The **Game** class is used as a main class to bring all the different components together. It contains an array of integers to represent the maze, with different numbers representing different types of tiles. To fulfil FR_MAP_CREATION, the array will already be filled to represent a pre-set maze, and will remain the same throughout the whole game.
In Fig 3a there is a link between the **Game** and **Timer** class. The **Timer** class keeps track of the time since the user started the game and is needed to make sure the user reaches the end of the maze within 5 minutes, else the user loses (in order to satisfy UR_TIME). Also, the time remaining can be displayed, fulfilling part of UR_UI. The **Game** class uses the **Timer** class, as the score variable is impacted by the **Timer**, necessary for FR_SCORING. The **Timer** class also includes the methods pause() and play(), which are

needed to implement the functionality as given by FR_PAUSING and UR_PAUSE.

The **Screen** class brings together all the visual components of the game; this was necessary as requirements UR_UI and FR_GAME_CAMERA state the importance of user interface, and the **Screen** class makes it easier to maintain a consistent art style (which links to FR_MAP_STYLE). Furthermore, the **Screen** class uses a variety of sprites from the **Sprite** class, including the player and the dean. The **Sprite** class contains information from the library, as well as variables to store the location and the sprite speed. It also contains methods which take the user's inputs, stores this information, and then affects the output. In Fig 3a, it only includes methods to move the player, however more methods can be added for further functionality.

The **Sprite** class takes information from the **Dean** and **Player** classes. In Fig 3a, there is little information provided in these classes, however it felt necessary to show these classes as there is high potential to add more functionality to them later on. One thing to note is that the **Player** class contains a list of objects from the **Item** class. An object from **Item** has a method to determine whether it has been used or not. An **Item** can be used for different events, and the **Player** class has an inventory to store these items. This idea was chosen to make the game more engaging during the events.

The **Event** class is used by the **Game** class when the player lands on a special tile in the maze. Events are necessary in the game in order to accomplish UR_EVENTS. The **Event** class contains 2 methods, scoreIncrement() and scoreDecrement(), which are used as score modifiers, to satisfy FR_SCORING alongside the timer. In addition, **Event** class uses the **Item** class, so that certain items can be used to influence events.

Process of designing class architecture
Link to previous class diagrams: Yetti - Architecture

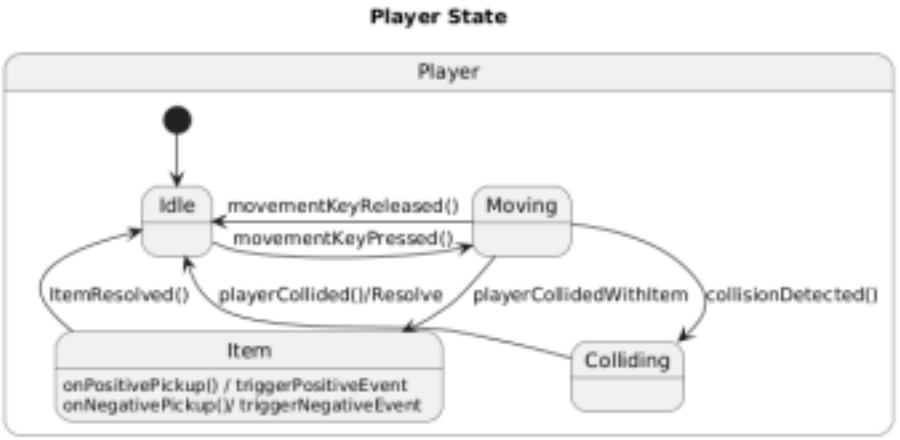
To initially come up with ideas on the architecture, we focused on what classes would be needed. Many of the classes remained in the final version of the class diagram, however we decided to remove some to make the overall structure simpler. For example, we had a Maze and Tile class which were removed, as we thought it may be best to store the maze array in the **Game** class. After having thought about how the array would work, we realised that integers could be used to represent the different tiles, and the graphics of the tiles could be represented by the **Sprite** class. Thus, the Tile class felt redundant.

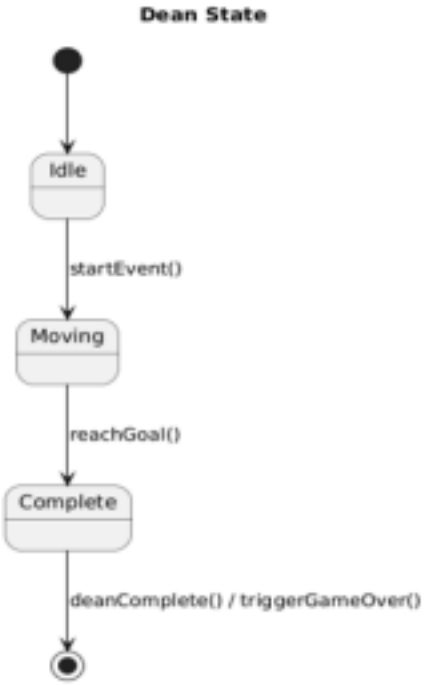
We considered the **Sprite** class and thought instead of having all the different assets (such as the Dean and Player) containing their own methods affecting the visual sprite, it would be best to put all that information in the **Sprite** class.
Another idea we had initially was having **Event** as an abstract class, and creating three classes for positive, negative and hidden events which inherited from **Event**. However, upon reflection, we realised that there wouldn't be much difference between the PositiveEvent, NegativeEvent and HiddenEvent classes, therefore we removed these classes, and made **Event** a regular class, and included a method to increase the score, and another to decrease the score.

State diagrams

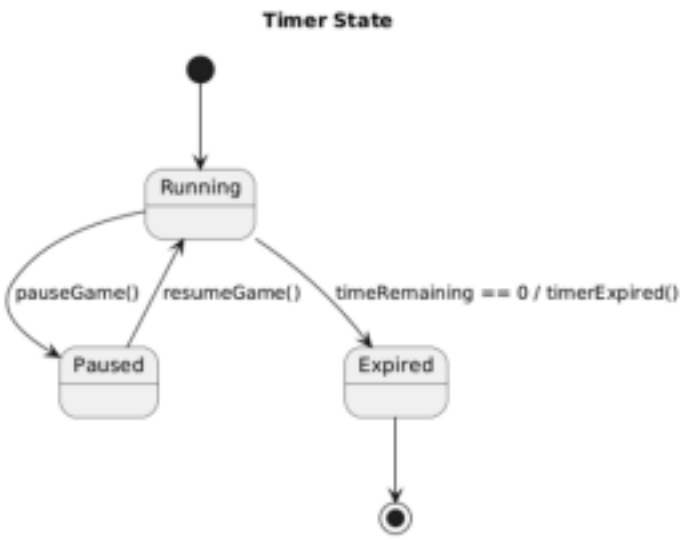
Given the event driven nature of our class architecture a state diagram of the game was created to reflect this. This was done as the overall game is determined by discrete events that trigger state transitions, as opposed to executing in a linear fashion. Meaning this system aligns more so with event driven systems as opposed to other types or architecture. These diagrams were created to show the behaviour of the game itself and to show how the different classes interact with one another such as the **player dean** and **score** substates. This representation makes it clear how the classes operate independently while still contributing to the overall system behaviour, the diagrams are not just a visual depiction of the game logic but also work as an architectural model that shows how the event driven system supports modularity and responsiveness.

Each substate within the overall **Playing** state acts as both an event producer and an event user. This can be seen in the **player** substate with the transitions between **Idle, Moving** and **Colliding** are all events that are triggered by discrete events such as **movementKeyPressed()**, **movementKeyReleased()** and **collisionDetected()**. These events determine the players behaviour and influence other parts of the system. The player state also encompasses the **items** used within the game which are one of the ways to manipulate the **Score.**

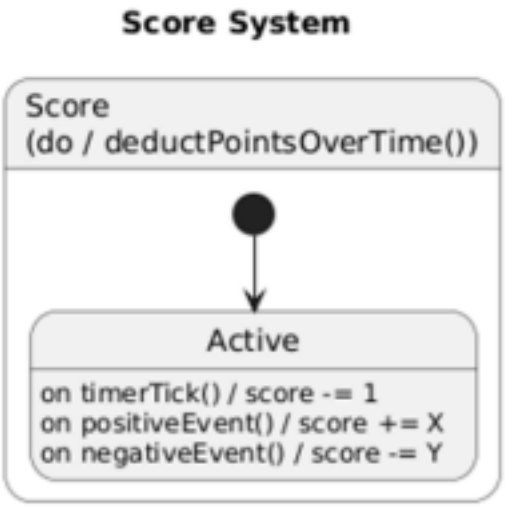




The **Enemy**/**Dean** shows the behaviour and the events caused by the main enemy of the game which begins to follow the **Player** after an event occurs. This substate responds to **startEvent()** and **reachGoal()** triggers moving through **idle**,**moving** and **Complete** states triggering a **gameOver()** when complete

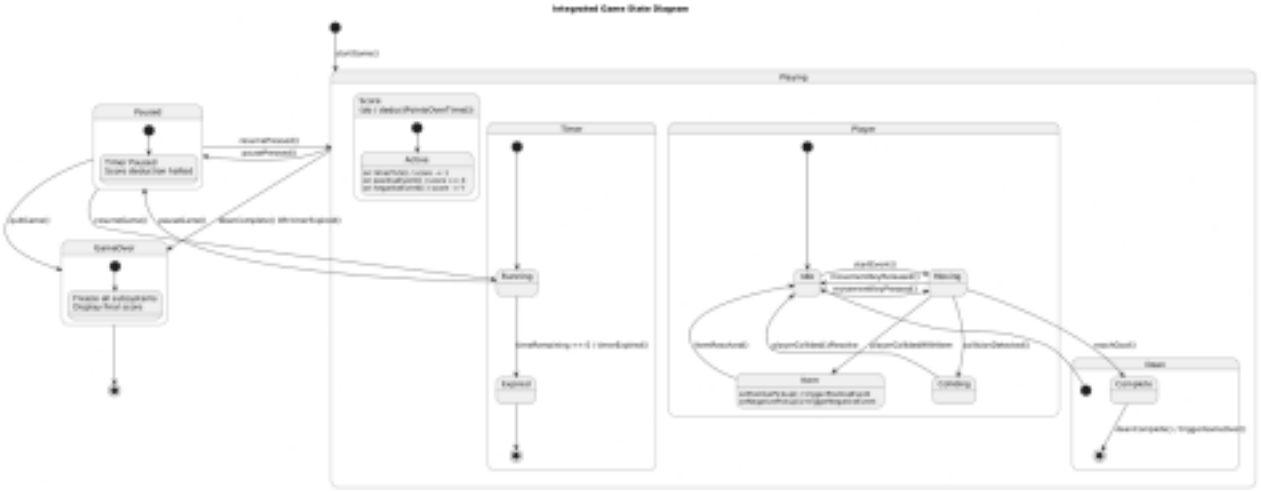


The timer substate reacts to **pauseGame**, **resumeGame** and the timer reaching 0 transitioning between **Running, Paused** and **Expired** when the **player** interacts with the pause menu buttons this substate satisfies the FR_PAUSEING and the UR_PAUSE requirements which is also shown in the larger state diagram as its own substate as well as satisfying NFR_GAME_TIME which ensures the game must last only five minutes with the expired state.



The **Score** substate updates its internal state in response to events such as **timerTick()** or the triggering of positive and negative events. This is one of the few states that is not self contained as it relies primarily on inputs from the **Timer** and the **Player**. The Timer affects score every tick making a constant change every second/tick whereas the inputs from Player require interactions with the environment this connected design highlights the nature of the event driven architecture allowing these subsystems to work together to create the score subsystem.

These subsystems all produce events that are used by the parent playing state in order to trigger higher level transitions such as **playing**, **paused**/**playing**, **gameOver**, which shows event propagation across multiple levels.



This state diagram shows the architecture supports a modular design making it easier to improve, change or rearrange certain aspects of the game as each substate handles its own logic separately while still being able to affect and be affected by other substates within the system. With the overall diagram you can see that the states meet the FR_WIN_SCREEN and FR_LOSS_SCREEN requirements which are triggered by the game over substrate outside the current playing states.

EDITS:

MJ_CDIAGRAM

Final class diagram:

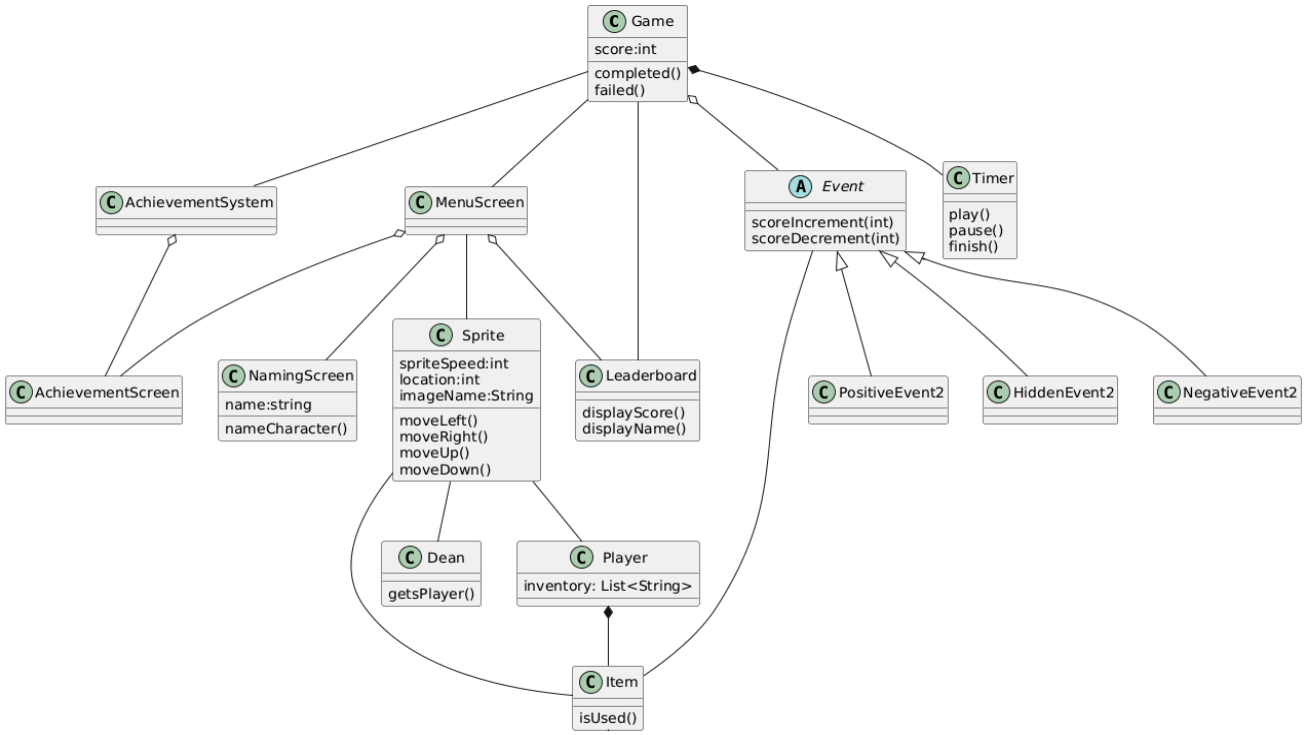


Fig 3b: Final class diagram for Assessment 2.

Since all classes from Assessment 1 have been explained and described, only the new classes that have been added for Assessment 2 will be explained.

The new updated class diagram is created using plantUML so it will continue to follow the OOA approach using OOP. This class diagram will help understand the relationships between classes showing association composition and inheritance. Likewise, this diagram is abstract so it ignores high details of implementation so architecture can be understood.

In Assessment 2, it is required to have more events (‘*UR_EVENTS’, ‘FR_POSITIVE_EVENTS’, ‘FR_NEGATIVE_EVENTS’,’FR_HIDDEN_EVENTS’*). To support the expansion of these event requirements the **Event** class was refactored into an abstract class and then was extended using inheritance so new specialised event classes such as **PositiveEvent2,NegativeEvent2** and **HiddenEvent2** inherit from **Event**. All of these events are tracked in the Event class to monitor how many events of each type have been triggered. Using inheritance will avoid duplication, improves maintainability and supports future event expansion without redesigning the whole architecture system.

In Fig 3b, there is a link between the **Event** class and the AchievementsSystem class since if specific events are triggered, this could lead to meeting certain conditions to unlock the achievement. The AchievementSystem class is then responsible for tracking the player’s actions and unlocking achievements based on the predefined conditions which then fulfills the *FR_ACHIEVEMENT_TRACKING* and *UR_ACHIEVEMENTS* requirement. In addition to this, the **AchievementScreen** class provides the user what achievements have been acquired including the name and description of that achievement, this provides clear feedback and progression information.

To fulfill the ‘*UR_LEADERBOARD’* AND *‘FR_LEADERBOARD_STORAGE’*, the Leaderboard class was introduced. This class is responsible for storing and displaying the top five highest

scores along with the corresponding player names at the end of the game by using the method displayScores() along with displayName. The reason why the Leaderboard is its own class is that the system can rely on managing the score data and present it through the *FR_WIN_SCREEN* user interface ensuring traceability between the requirement and implemented behaviour.

Lastly the class **NamingScreen** links with the **Leaderboard** class. Before the game starts, a screen should pop up asking the player to enter their name, which fulfills the '*UR_PLAYER_NAME'* requirement so it can then be used to display the name to the corresponding score on the leaderboard interface. This ensures that player scores can be uniquely identified and correctly associated, this supports the UR and FR requirements for the leaderboard.

MJ_EVOLVE

Website link to the process of architecture including date, images and explanation:

How the Architecture evolved during Assessment 2:

Following the inherited architecture from Assessment 1, the final class design has been evolved through three processes where the third process was the final design. The class diagram was evolved incrementally rather than from scratch. This was done to keep the architecture consistent while allowing the system to support additional requirements/features that were mentioned in the product brief.

The design process began by reviewing the inherited architecture diagram from Assessment 1 and comparing this with the new updated project requirements in the change report. Our focus was to add new classes that related to the new requirements that has been added by extending the inherited architecture without breaking existing design decisions.

The first Architecture class design introduced a few new core classes to support Assessment 2's functionality for implementation while also preserving the original structure. These new additions include NamingScreen,Leaderboard,AchievementScreen and AchievementSystem. These classes were added to fulfill the new user and functional requirements related to the player, score and achievements. The NamingScreen allows the player to enter a name before the game is started, which is needed for the leaderboard when the player wins. The AchievementSystem tracks the player's actions to see if it meets a certain condition and the AchievementScreen provides an interface showing what achievements have been unlocked/locked and its name and description for it. During this stage these classes were added only using simple associations to show how they are integrated into the existing architecture system.

The second process continued to extend this architecture to support the expanded gameplay mechanics. New positive,negative and hidden events were added to reflect the increased variety of events that were required by the product brief in Assessment 2. In addition to this, a Coins event was discussed and was suggested by the implementation team who decided they would like to add coins in the game to motivate players to try to achieve the best score when competing on the leaderboard so it supports continuous scoring and player feedback during gameplay.

The final class diagram evolved the design by introducing inheritance and containment relationships that were not shown in Assessment 1's final diagram. Inheritance was used to represent specialisation for different events and item types. Similarly, containment was used to show ownership and lifecycle dependencies between core system components. These changes were important because it would help guide the implementation team on what type of

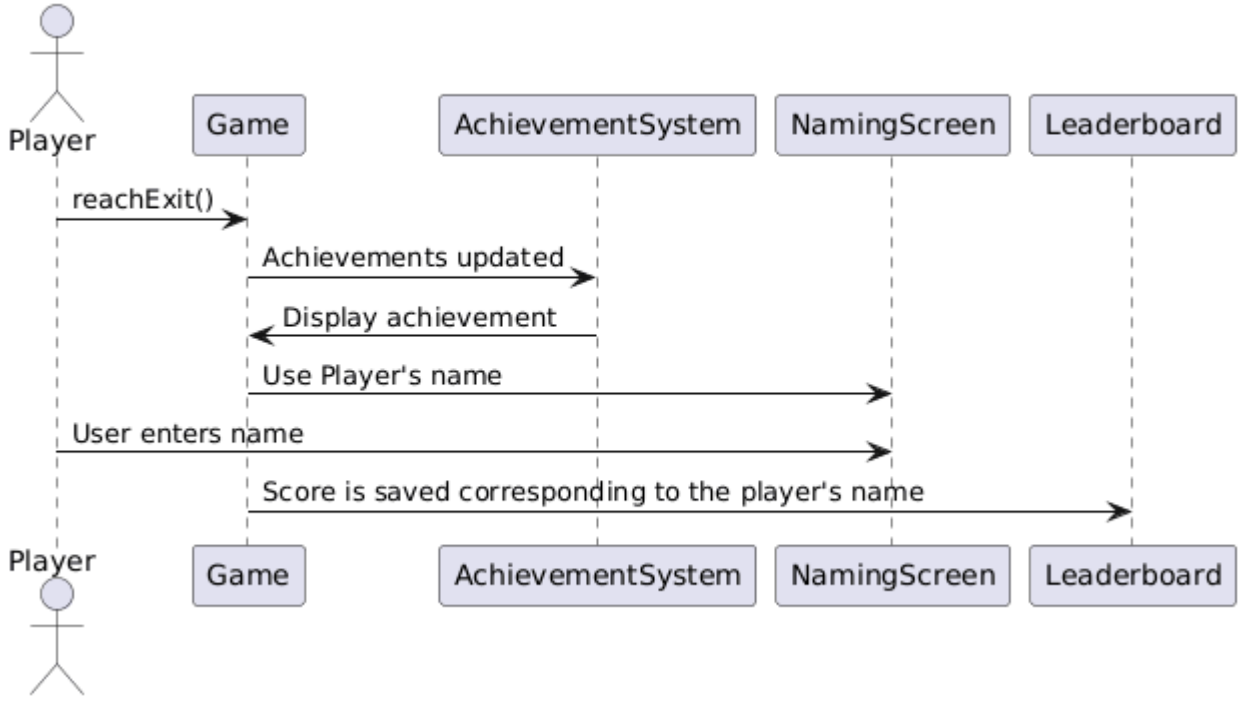
classes they'll need to use, improve traceability between requirements and architecture and support software maintenance. So, this final process represents a more mature and expressive architecture model while remaining consistent with the OOA and OOP principles that were set in Assessment 1.

MJ_SEQUENCE

The previous report on the player and Dean states were too simple and vague for the people reading the report to understand what is truly going on in the game

Note: The explanation highlighted in yellow, is the explanation added on from the previous group's explanation about their states.

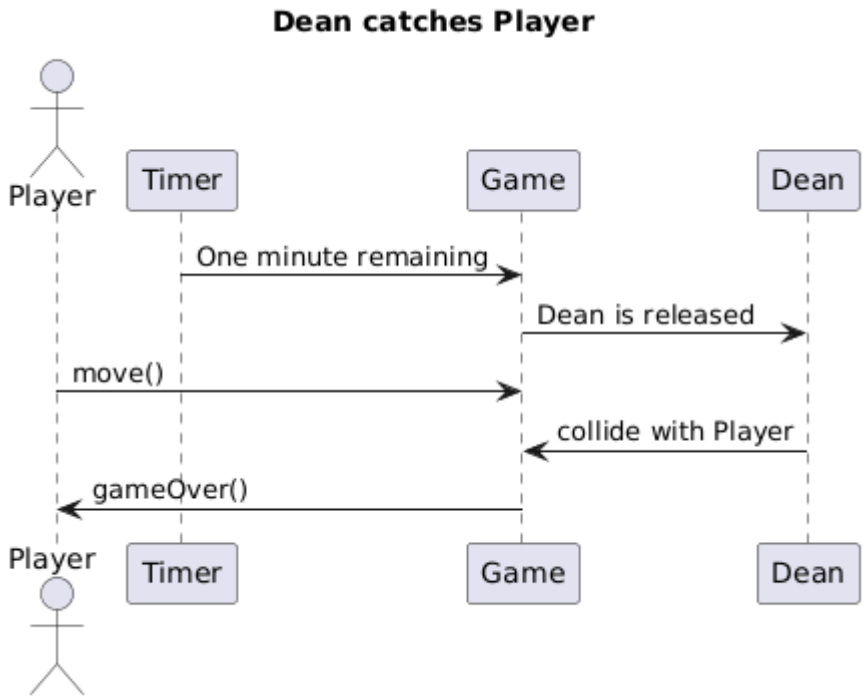
Player state:



To add onto the previous explanation of the Player state earlier in Assessment 1, this will be explained through the new sequence diagram. When the user enters the name, it will then be stored so the player's score will be able to correspond to the score which will be later used to associate the player's final score with the leaderboard.
During gameplay, the player's movement triggers events that interact with other systems in the architecture. This could lead to achievements being unlocked, score changes or collecting coins. The player's actions are monitored by the AchievementSystem which updates when a specific achievement is unlocked due to if specific conditions are met.This achievement is then visible to see in the Achievement Screen and displayed in the game for a short time to alert the player they have unlocked it.

When the player successfully reaches the exit, the game ends and the final score is evaluated. The leaderboard updates if the player reaches one of the top 5 best scores with their corresponding name they entered at the start of the game. It is then saved to the leaderboard. This sequence diagram shows how the Player state coordinates with input handling,event triggering, achievement tracking and leaderboard updates.

Dean state



To add onto the previous explanation of the Dean state earlier in Assessment 1, this will be explained through the new sequence diagram. The Dean remains mostly inactive during the game and it is only spawned when the timer has one minute remaining.
When it is spawned, the Dean instantly transitions into a chasing state and begins to follow the Player. During this, the system constantly checks if the player collided with the Dean. If the collision occurs, the gameover condition is triggered and the player will lose the game so the score would not be saved to the leaderboard. This sequence diagram demonstrates how the Dean is used as a final phase of gameplay and reinforces the Dean's role as a high risk threat rather than just a standard event.