Architecture

C3 Group 6
Team WHNI

Aaratrika Das
Arkoday Roychowdhury
El Foster
Hanna Pieniazek
Lewis Keat
Matthew Baghomian
Matthew Ford
Subham Magar

Class diagram

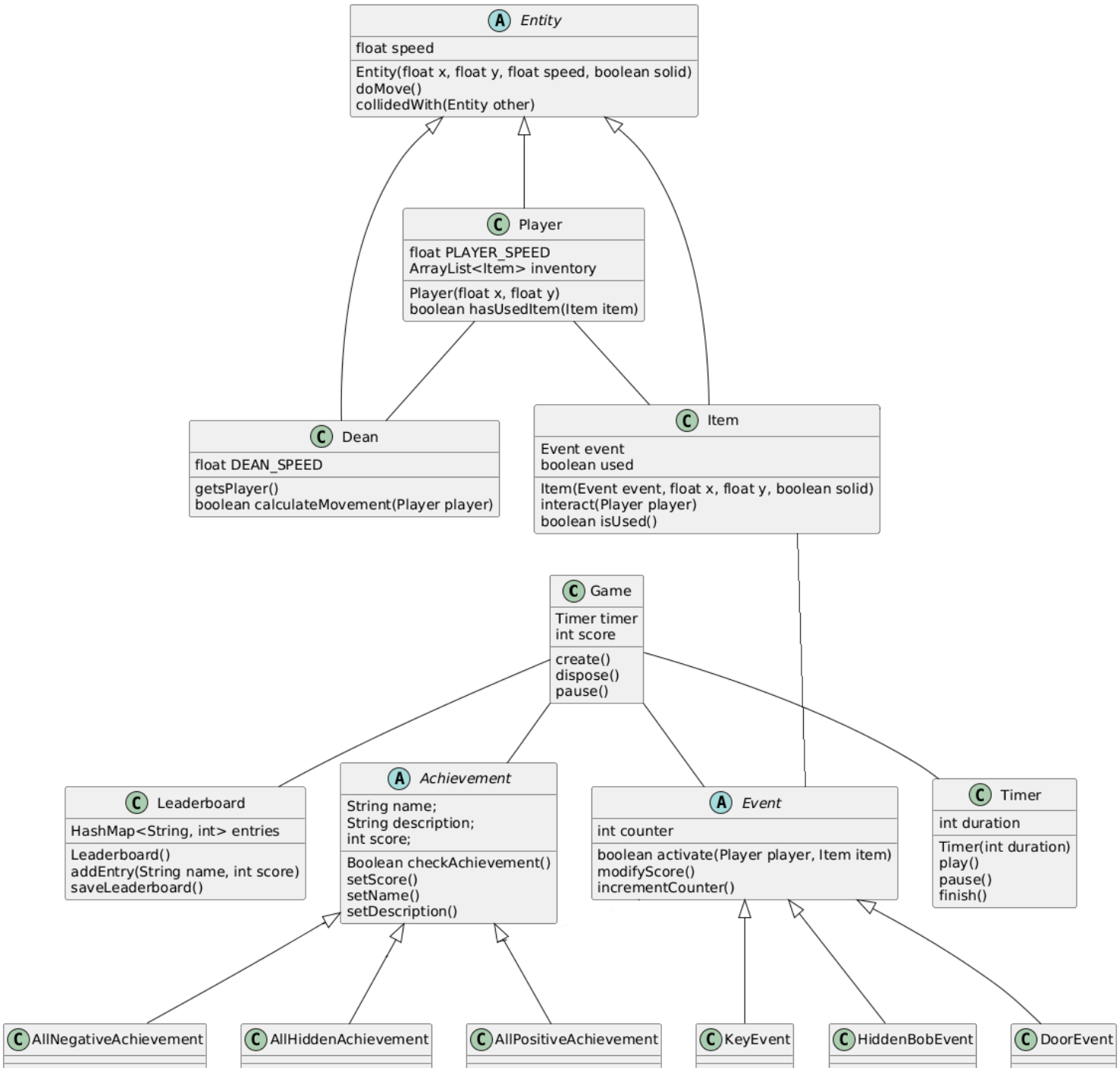


Fig 3a: the final class diagram

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 a timer and a score to represent how much time the player has left and how well the player is doing respectively. This is needed in order to fulfil UR_TIME and UR_SCORING.

In Fig 3a there is a link between the **Game** and **Timer** class, as 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 **Entity** class is abstract as the **Dean**, **Player** and **Item** classes inherit from the entity class. This was done as these classes share similar functionality and so having them inherit the same class helps to reduce code duplication. 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 the method modifyScore(), which is used as a score modifier, 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. The **Event** class is abstract as it allows for each event in the game to inherit from it which allows common features to be shared between events.
The **Achievement** class is used by the **Game** class in order to store achievements in the game and to update the achievements the player gets once they reach a goal. This was done in order to fulfill UR_ACHIEVEMENT. The **Achievement** class was made abstract so that the different achievements could inherit from the **Achievement** so that functionality could be shared in order to reduce code duplication.

The **Leaderboard** class is used by the **Game** class in order to store all the scores that players have made on the game. This was achieved by using a hashmap containing the name given by the player and the score achieved by that player. This was done in order to fulfil UR_LEADERBOARD.

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 **Entity** class. Thus, the Tile class felt redundant.

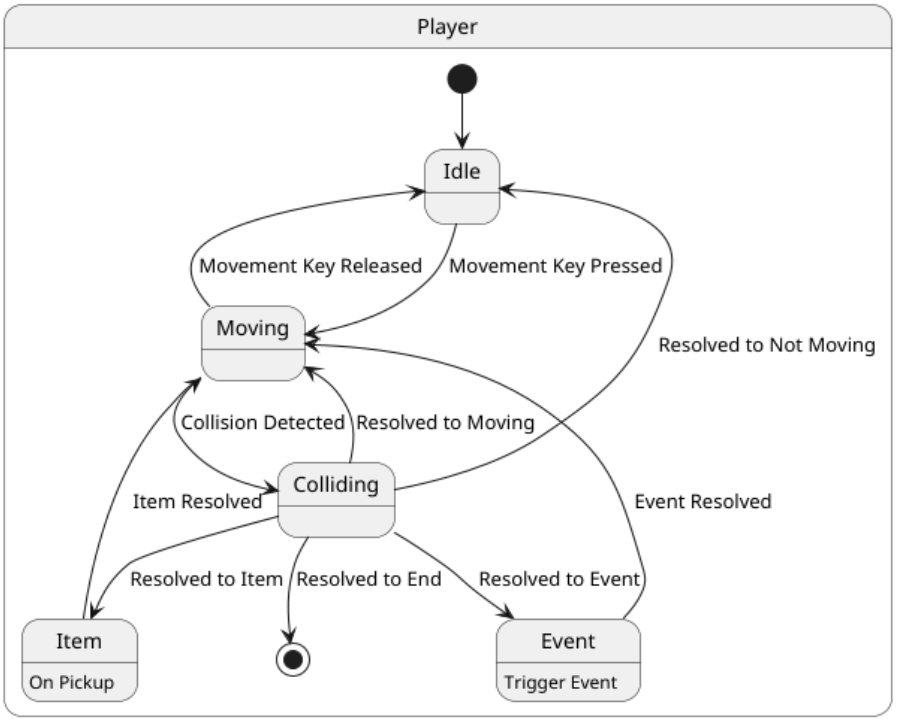
We considered the **Entity** class and thought instead of having all the different assets (such as the Dean and Player) containing their own methods affecting the visual sprite, it would be best to put all that information in the **Entity** class.
Another idea we had initially was having **Event** as 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 alter the score.

State diagrams

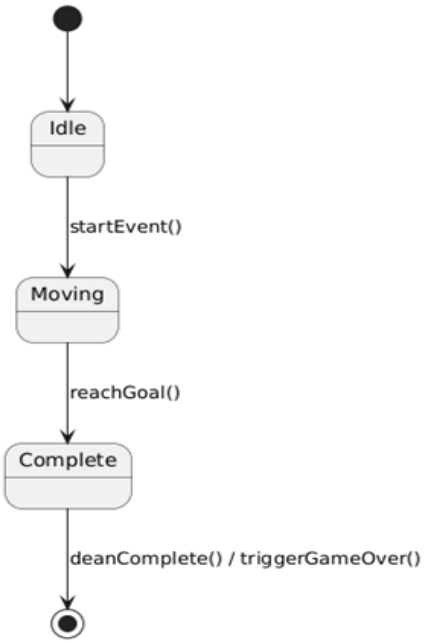
Given the event-driven nature of our class architecture, a state diagram of the game was created to reflect this. This was done as the overall game is determined by discrete events that trigger state transitions, as opposed to executing in a linear fashion. Meaning this system aligns more so with event 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.**

Player State

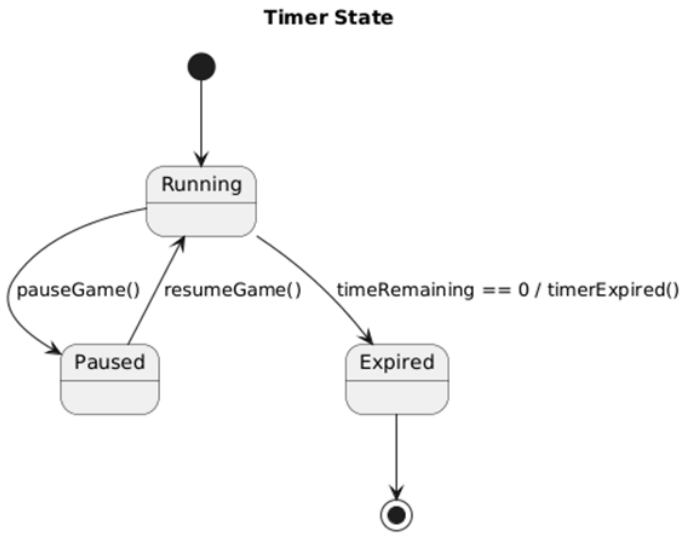


Dean State

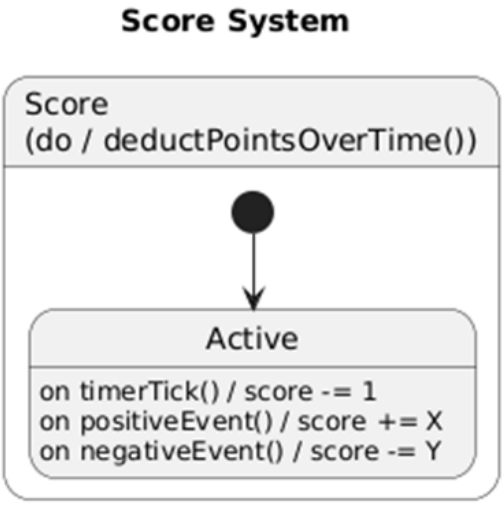


The **Enemy**/**Dean** shows the behaviour and the events caused by the main enemy of the game which begins to follow the **Player** after an event occurs. This substate responds to **startEvent()** and **reachGoal()** triggers moving through **idle**,**moving** and **Complete** states triggering a **gameOver()**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

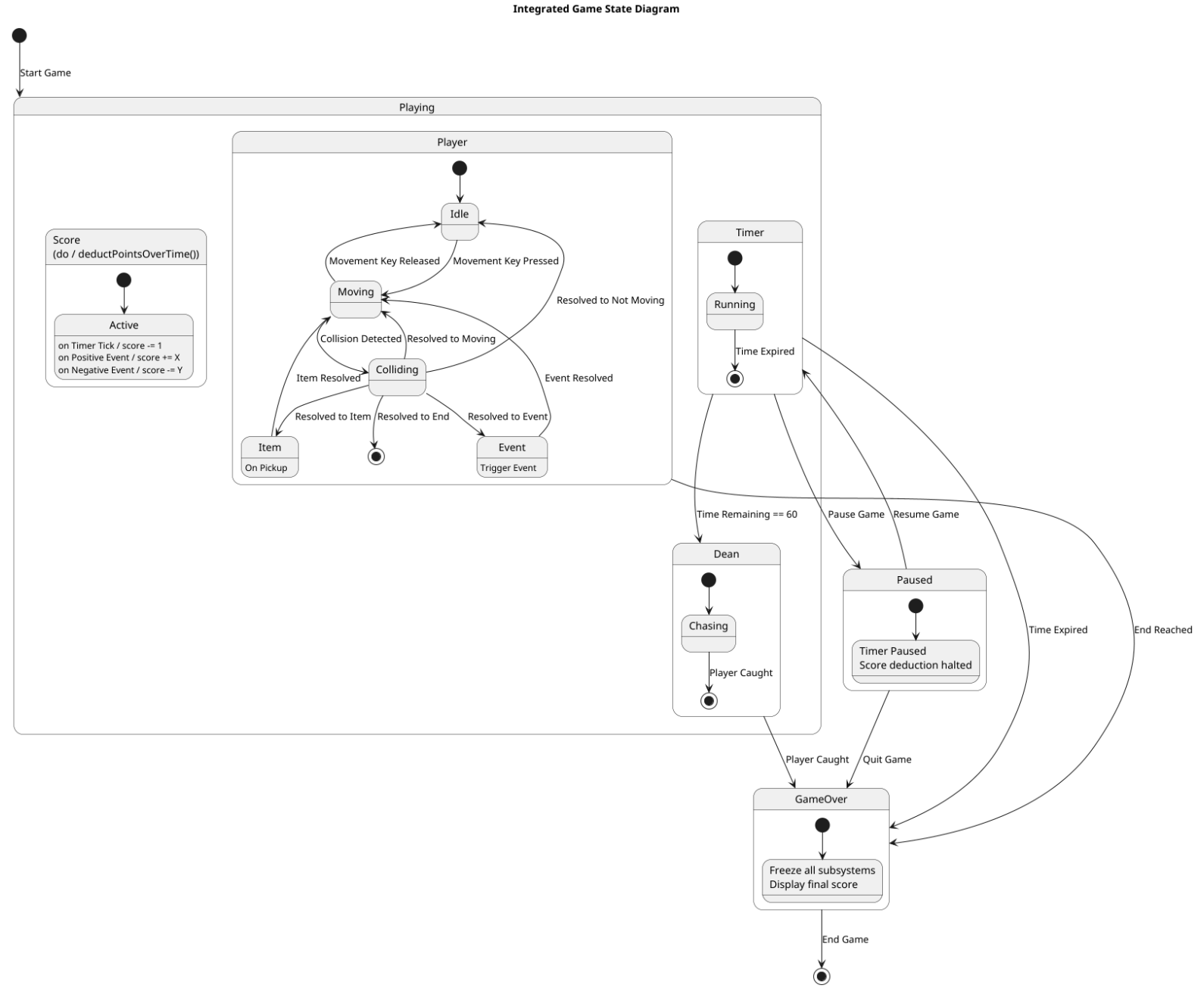
The timer substate reacts to **pauseGame**, **resumeGame** and the timer reaching 0 transitioning between **Running, Paused** and **Expired** when the **player** interacts with the pause menu buttons this substate satisfies the FR_PAUSEING and the UR_PAUSE requirements which is also shown in the larger 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.