

## Architecture

### Class diagram

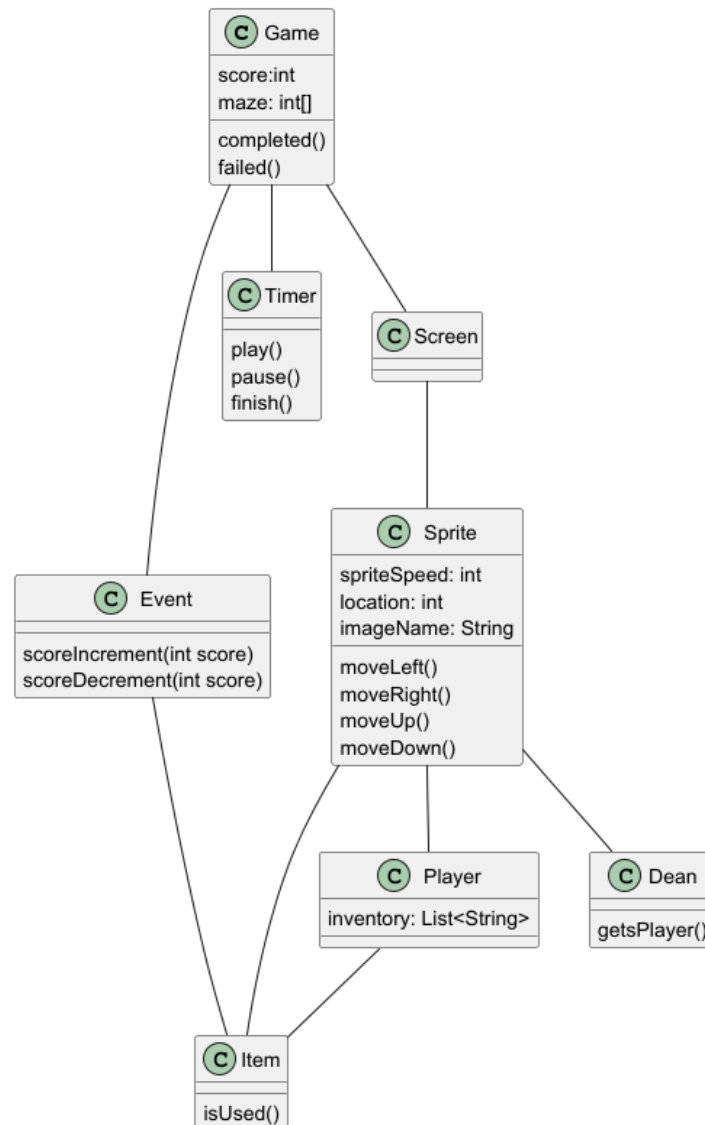


Fig 3a: the final class diagram

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:](#)

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

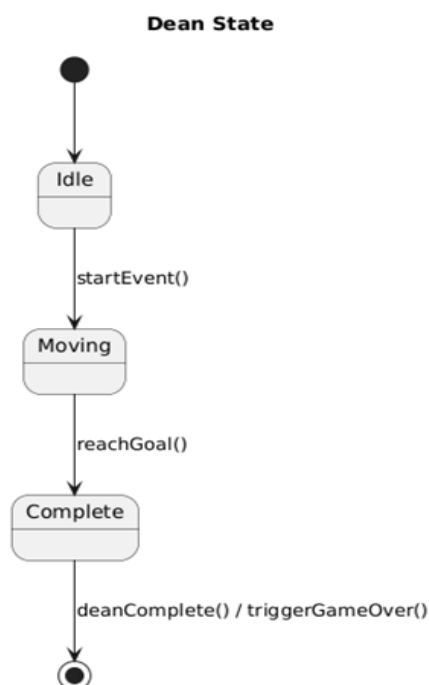
reflection, we realised that there wouldn't be much difference between the PositiveEvent, NegativeEvent and HiddenEvent classes, therefore we removed these classes, and made **Event** a regular class, and included a method to increase the score, and another to decrease the score.

### State diagrams

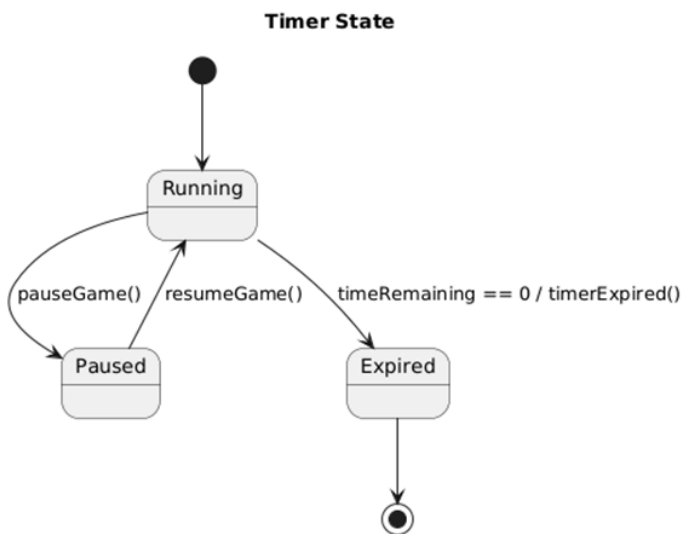
Given the event driven nature of our class architecture a state diagram of the game was created to reflect this. This was done as the overall game is determined by discrete events that trigger state transitions as opposed to executing in a linear fashion this system aligns more so with event driven systems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

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 movement pressed, released and the detection of collision with a wall

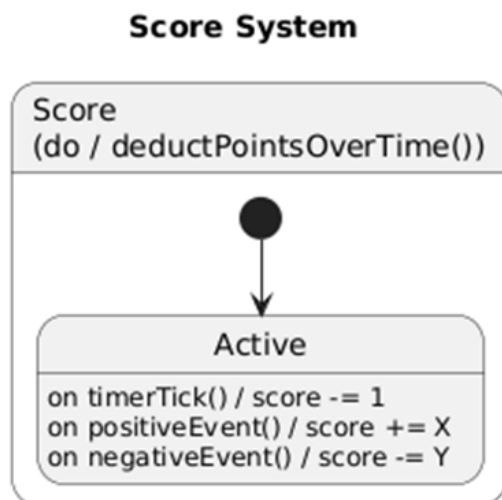
The Enemy/Dean shows the movements and the events caused by the main enemy of the game which follows the player after an event occurs this substate responds to start event and reach goal moving through idle moving and complete states triggering a gameover when complete



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



The Score substate updates its internal state in response to events that happen around it such as timer tick or the triggering of positive and negative events. This is one of the few states that is not self contained as it relies on the timer primarily and the positive and negative events triggered by player interacting with the environment.



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, game over, 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.