Architecture

Class diagram

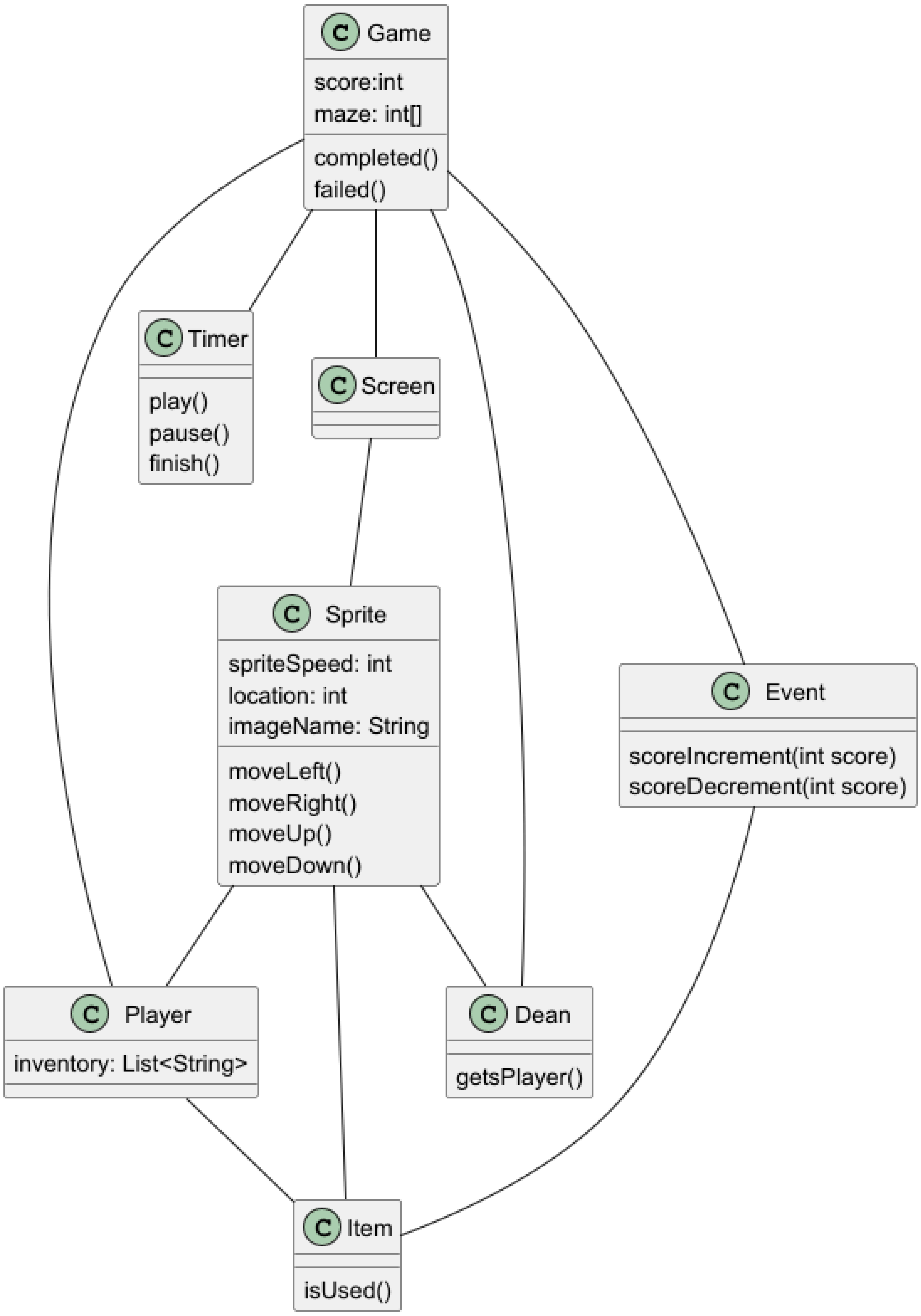

Fig 3a: the final class diagram

(Need to remove lines with Player-Game and Dean-Player)

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.

The design was influenced by pipeline architecture. The player will send inputs to the program, and these inputs will influence the different components of the game (such as the location of the player). This information can then be used to produce further data relating to other game components (e.g. the player landing on an event). This in turn then influences the graphical output.

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 architecture

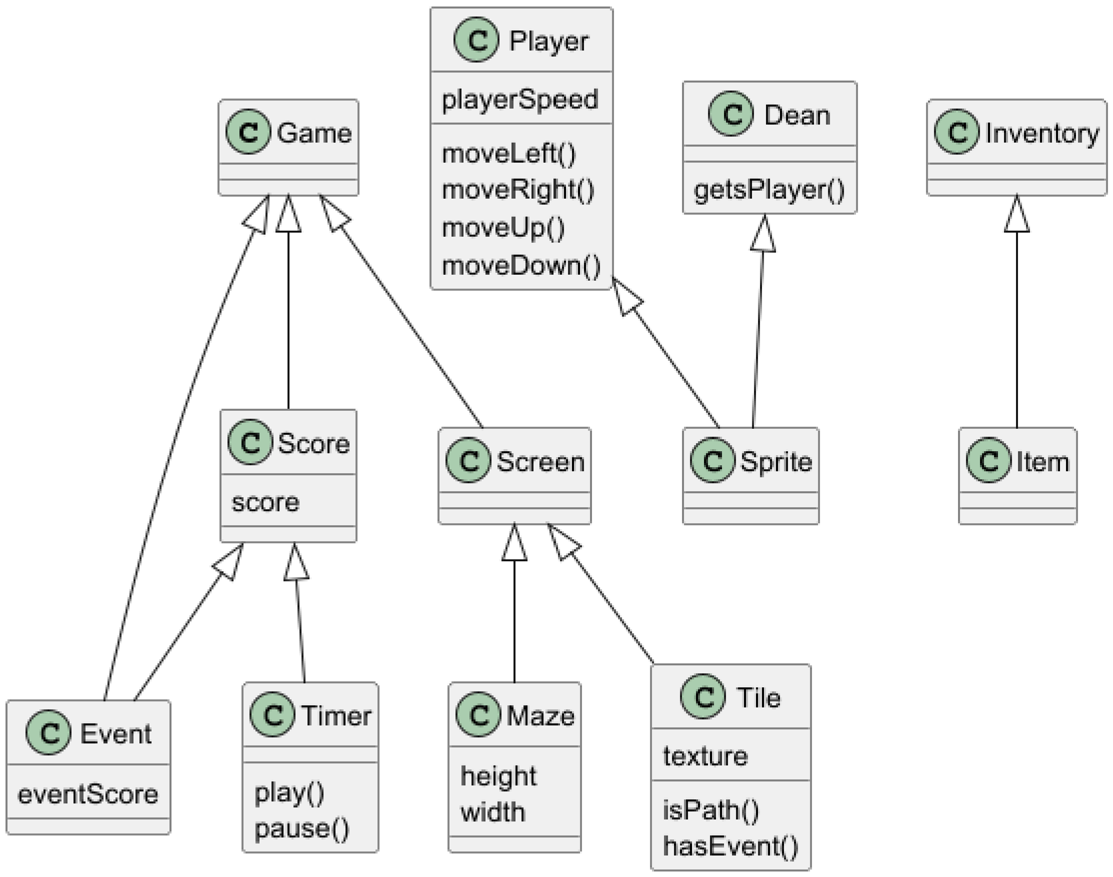

Fig 3b: an initial class diagram.

To initially come up with ideas on the architecture, we focused on what classes would be needed. Some of these can be seen in Fig 3b. 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, the Maze and Tile classes shown in Fig 3b 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.

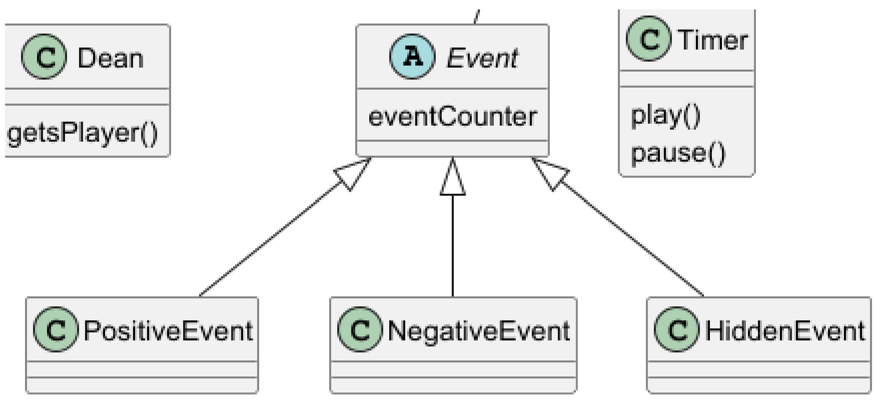

Fig 3c: part of a previous class diagram.

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**, shown in Fig 3c. 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 diagram (behavioural) - shows how the game changes with different events, how decisions influence game