Architecture

Initially by Cohort 3 Team 7 (Yetti)

Bak, Bartek
Burberry, Katelyn
Collins, Lucy
Ganley, Joe
Keegan, Josh
Kirkham, Katharine
Mayall, Daniel
Sawdon, Theo

Updated by Cohort 3 Team 3

Fares Almutairi
Taro Chandiwana
Lily Gaunt
Kostas Kanakaris
Jacob Last
Alex Ponomarenko
Giulia Sclano

Overview

The purpose of this document is to record design decisions we made relating to the high-level architectural structure of the game, explaining how we came up with them, how they changed over the course of the project, and to explain & justify why certain decisions were made.

With the structural architecture, we aimed for our classes to fulfil the OCP design principle, as this improved maintainability and helped us extend our code through methods such as inheritance. This also ties in with the DRY design principle, as it allowed us to expand on code rather than duplicate it. Additionally, whilst we wanted our structural architecture to allow for a thorough understanding, we also wanted to find a balance, so that we could provide enough detail, while still being a high-level representation (KISS design principle).

As stated above, the scope of this architecture will be fairly high-level, meaning we won't include small details such as specific events that happen in the maze game, as these are implementation details. However, it will be quite clear where these classes will need to go in the class hierarchy when these are developed.

Structural Diagrams

Class Diagram

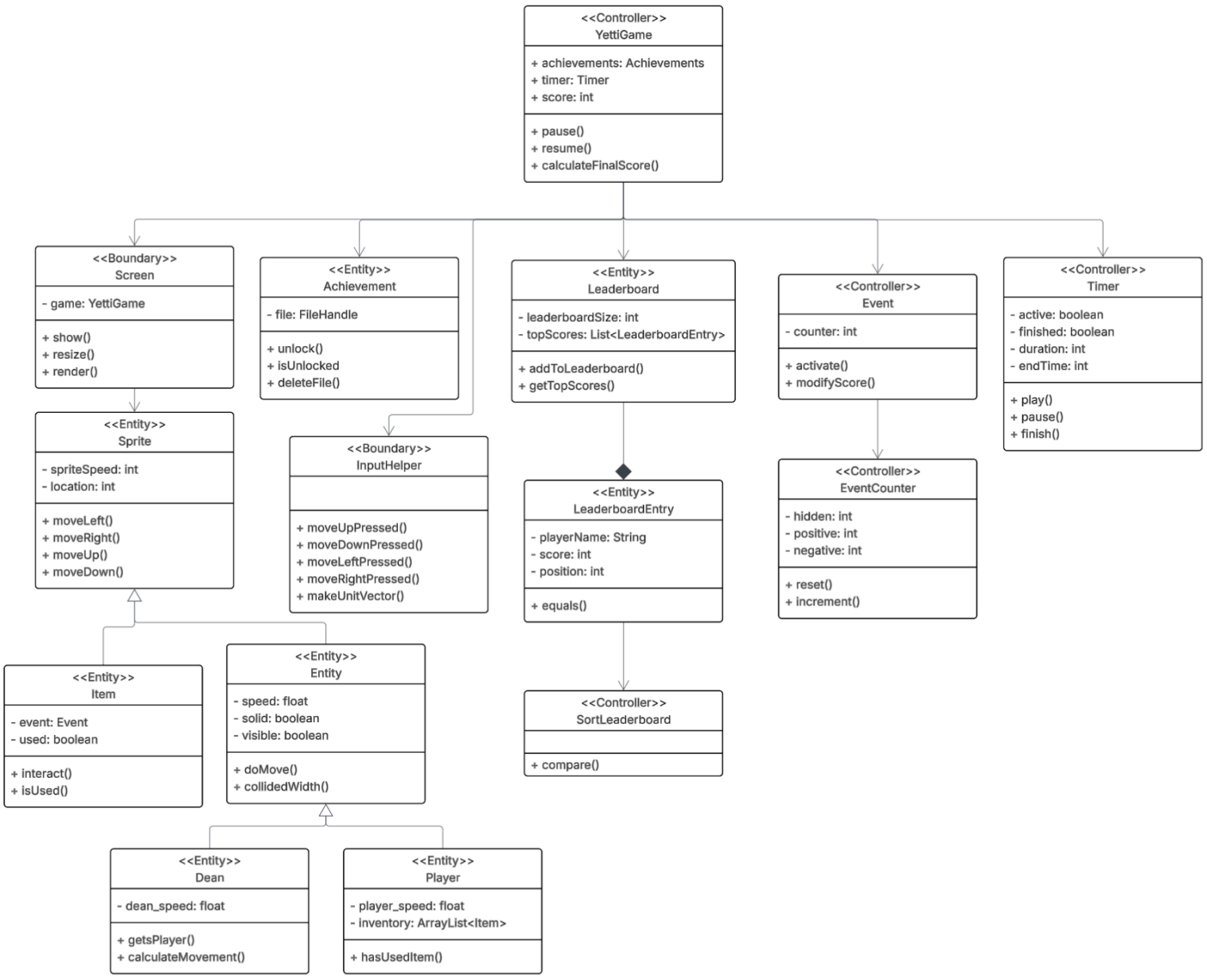


Fig 3a: the final class diagram

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 class diagram was designed using LucidChart.

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 **Achievement** class has a similar connection to **Game**, keeping track of the achievements that the user has unlocked (UR_ACHIEVMENTS).

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. 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.

The **Leaderboard** class is used by the **Game** class after the player completes the game. It is composed from **LeaderboardEntry** objects in order to create a leaderboard of top scores as described in UR_LEADERBOARD. **SortLeaderboard** is a comparator class used to sort the entries into descending order.

Design Process

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.

Design Patterns

Below, we have identified the design pattern types of the most significant classes, to increase the reusability of the system.

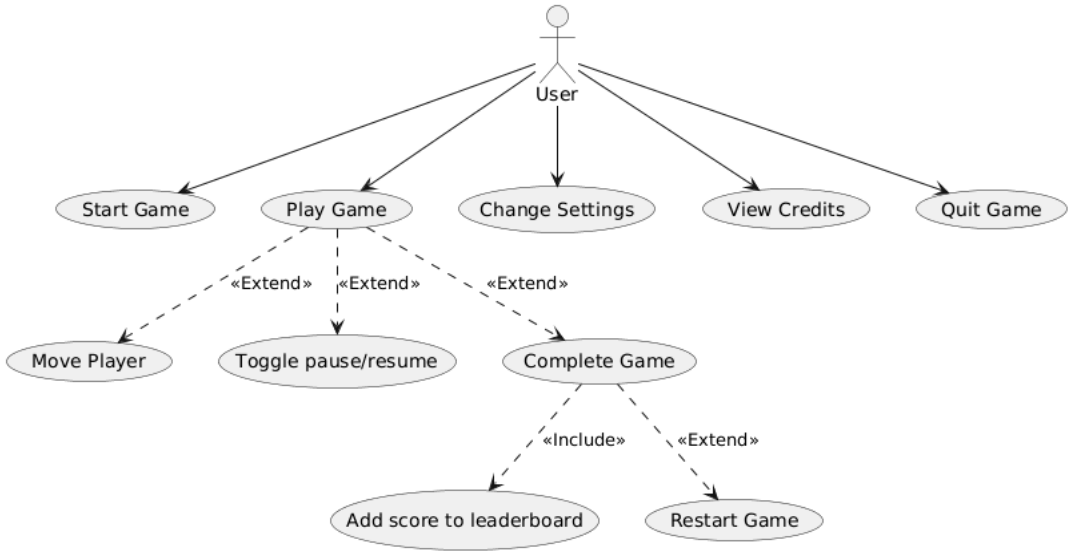
- YettiGame, Leaderboard and Timer fall under 'singleton' as there is only one instance of each.
- Event and Screen are 'factory' classes, as they provide an interface for the features objects should have.
- Screen is also a 'state' class, as switching between them can happen during runtime.
- Achievement is an 'observer' class, as they unlock in response to game events.
- Sprite is a 'prototype' as it is cloned to create similar objects.

Behavioural Diagrams

The UML behavioural diagrams were produced using plantUML, and model what the game does when the user interacts with it. These diagrams are fairly abstracted, focussing on what the game should do and how the objects could interact, rather than specific implementation details. The diagrams were very useful for validating the structural diagrams and making sure it supports all of the required behaviours. The user and functional requirements that each of the diagrams support are listed below each diagram.

UML Use Case Diagram

This diagram shows a high-level overview of what the user should be able to do when they're running the game.



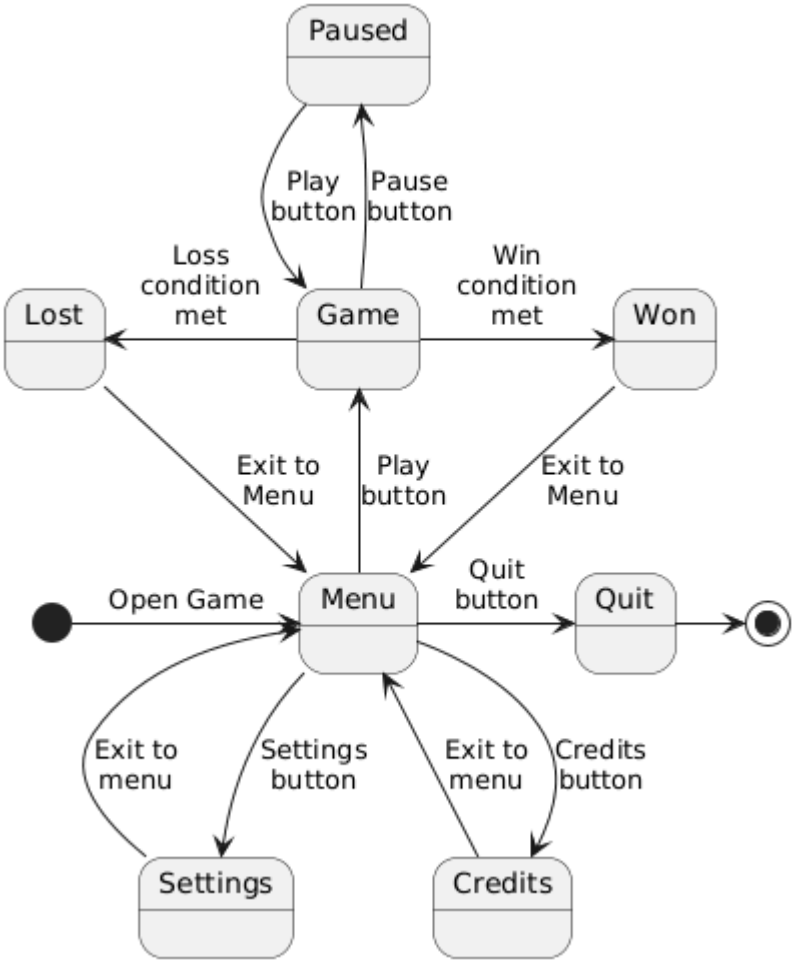
It's easy to understand from a non-technical point of view, making it a useful diagram to show our stakeholders to verify we will be meeting their requirements. We also compared it against our requirements document, concluding that it helps to support UR_SCORING, UR_PAUSE, UR_RESTART, UR_ENDING, UR_MAIN_MENU, UR_LEADERBOARD and the Functional Requirements that link to these User Requirements.

UML State Diagrams

As the structural architecture is event-driven, it makes logical sense to visualise these events by showing how they affect the state the game is in. We have produced two state diagrams in order to do this; one high-level one for the game as a whole, and one showing the state transitions as the game is being played.

Overall Game State Diagram

This shows an overall view of all the states the game can be in. It is based mainly on what 'screens' the game can be in from a user-perspective, so helps to verify that we have covered all functionality the user should be able to complete from each screen.



It also helps to verify some requirements, particularly UR_PAUSE, UR_RESTART (not explicitly but via 2 transitions from Lost/Won back to Game), UR_ENDING, UR_MAIN_MENU and the Functional Requirements that link to these User Requirements.

‘Playing’ State Diagram

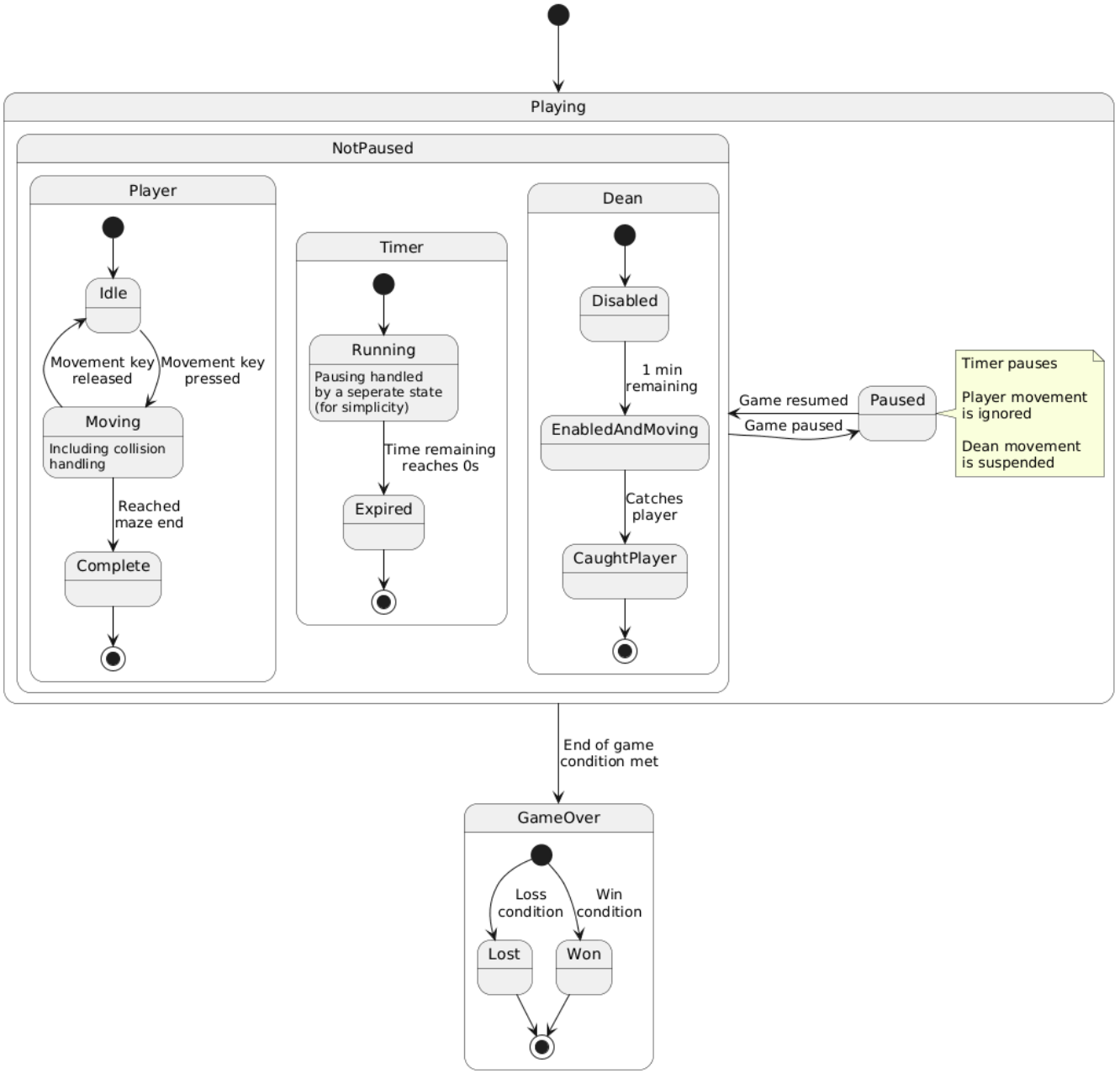
This ‘zooms in’ to the ‘Game’ state of the previous state diagram, and consists of substates for the Player, Timer and Dean.

This diagram was created to show the behaviour of the game itself and to show how the different actors interact with one another. This representation makes it clear how the various actors 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 transition between **Idle** and **Moving** being triggered by a discrete event.

The Playing substate reacts to the game being paused and resumed, and the Timer substate reacts to reaching 0s.



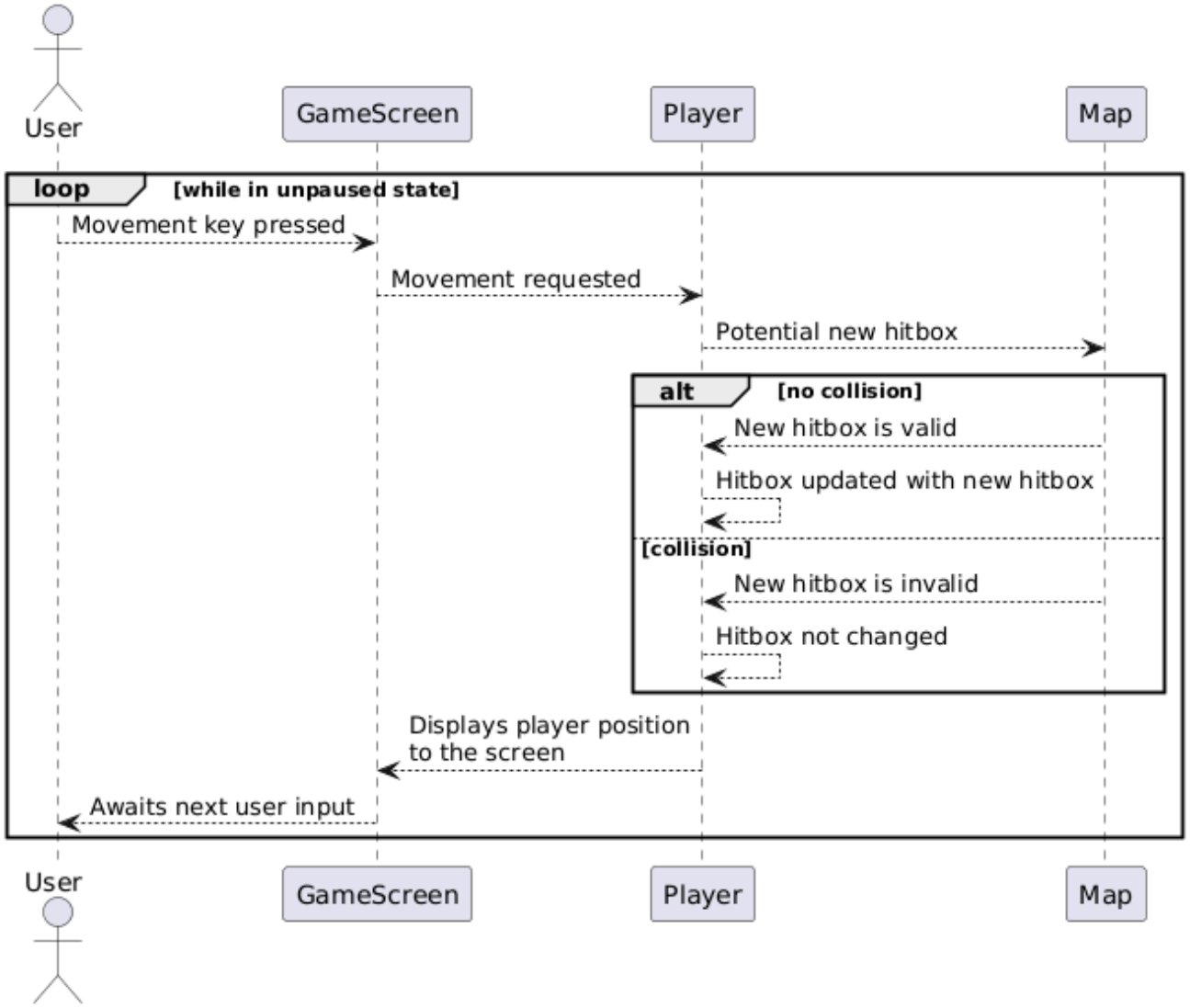
This state diagram shows the architecture supports a modular design making it easier to improve, change or rearrange certain aspects of the game. Each substate handles its own logic separately while still being able to affect and be affected by other substates within the system.

The requirements it helps to support are: UR_EVENTS (however, only partially as only one event is included), UR_PAUSE and UR_ENDING, alongside the Functional Requirements that link to these User Requirements.

UML Sequence Diagram

The purpose of this diagram is to expand on how collisions may be handled within the ‘Player → Moving’ State above.

It didn’t make sense to include this as a state, as in reality the collision handling works by working through a sequence of steps and communication between multiple actors, hence a sequence diagram is perfect to convey this.



This helps to verify what each actor should be responsible for. The main game should detect user input and trigger the Player actor to generate a new hitbox based on this. This is then checked by the Map to see if it is valid, and updated accordingly by the Player. It also helps to visually see what information each actor needs to perform their job, e.g. the Map doesn't need to know the old hitbox or which movement key was pressed, all it needs is the new candidate hitbox.