We reviewed the most popular architectural styles, and narrowed down our search to microkernel, service-based and event-driven, simply because the other styles were not performant enough for our game. We discussed the microkernel architecture, and found it unsuitable because the inelasticity of the core system was unlikely to scale well – we might need to rework core elements of the game later on in development, resulting in the core system being changed which all the plug-in components depend on, forcing us to change those components as well. Then, we considered service-based and event-driven and concluded event-driven is the better option, because its benefits with elasticity and scalability would help us later on in development. Through implementing our requirements, event-driven made more sense over service-based to us because the flow of gameplay matched event-driven more, as the player would elicit events to the system which would handle them appropriately.

For the event-based architecture, we could decide between the asynchronous event-based and synchronous message-based. We went for the asynchronous option, because for message-based, the player would need to wait for the system to handle events, thereby interrupting gameplay. This would be detrimental to player experience, because they would have to wait for the system to handle events before resuming gameplay. Having decided on an architecture style, we can now begin the architecture development, in line with the User and system requirements.

Following the principles of responsibility-driven design (RDD), we started our architecture process by creating a designer story from our user and system requirements. We can use this designer story to provide focus to the design of our architecture, and to inform us on implementation choices in the future. Our designer story is as follows:

We are developing a single player game that allows a user to traverse and escape a maze, to be run as a portable application running locally on the host device.
Once the game starts, the maze will be loaded, and various positive and negative events will be placed in the maze. When these have been successfully loaded, the player will have 5 minutes to escape the maze, helped by the positive events, and hindered by the negative ones.
Positive events may give the player items that can be used to escape more quickly, or provide them with a faster route. Negative events will either slow the player down, or cause them to lose immediately, depending on the obstacle.
If after 5 minutes they have not escaped the maze, then the game declares that the “Dean” has caught them, and they lose. If they escape sooner, then the player wins.

Using this design story, we discussed the themes that are needed to meet the requirements of the program. These will allow us to define the components of the program, and begin to create objects to implement, in order to meet the requirements. The main themes are as follows:

- A portable, interactive User interface, made of different screens for different functions.

- A user friendly, university themed board layout which considers event locations and interactive items, as per the UR_THEME and NFR_THEME requirements.

- Events which provide interaction for the user, and a winning / losing condition, as per the UR_EVENT, UR_POSEVENT, UR_NEGEVENT and the UR_HIDEVENT requirements.

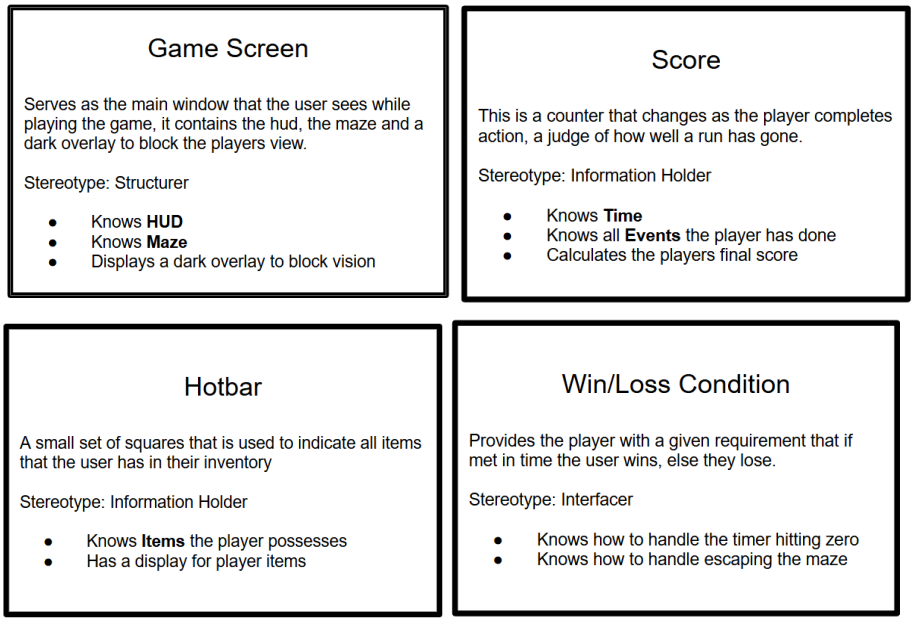
Having defined the key themes, we discussed candidate objects that can be used to implement the chosen themes. These were split into two categories: Domain objects, and Representative objects.

The Domain objects contain key concepts in the game, and consist of either entity or value Objects. We initially decided on the following Domain Objects:
Win/Loss condition, Positive Events, Negative Events, Hidden Events, Score, Interactables, Maze, Effects.

The other objects are representations of the software's view of the game, such as the screens that display the different concepts. We initially decided on the following Objects:
MenuScreen, GameScreen, SettingsScreen, PauseScreen, EndScreen, HUD, Hotbar.

Using these Candidate objects, we formed CRC cards, which state the purpose and role stereotypes of each Object. In our initial draft, we noted the purpose, and rudimentary responsibilities, of each card, found on our website https://matth3wk.github.io/images.html

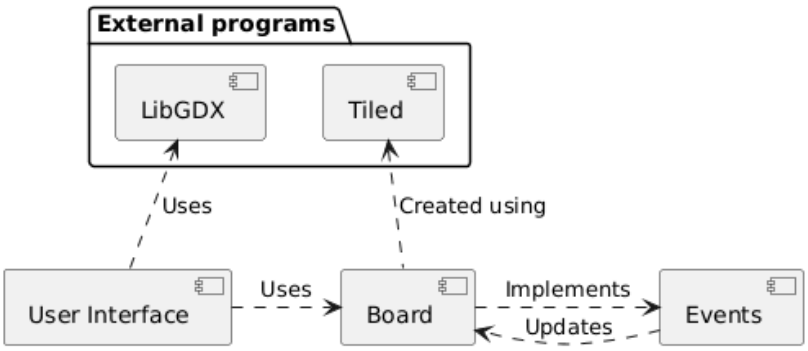
Following this, we then discussed the role stereotypes for each Candidate Object, and added them to the CRC cards, which we digitised. Some examples are shown below, and the rest can be found on our website:



Having completed each CRC card, we then grouped them into clusters of related cards. This highlighted some redundant character cards.

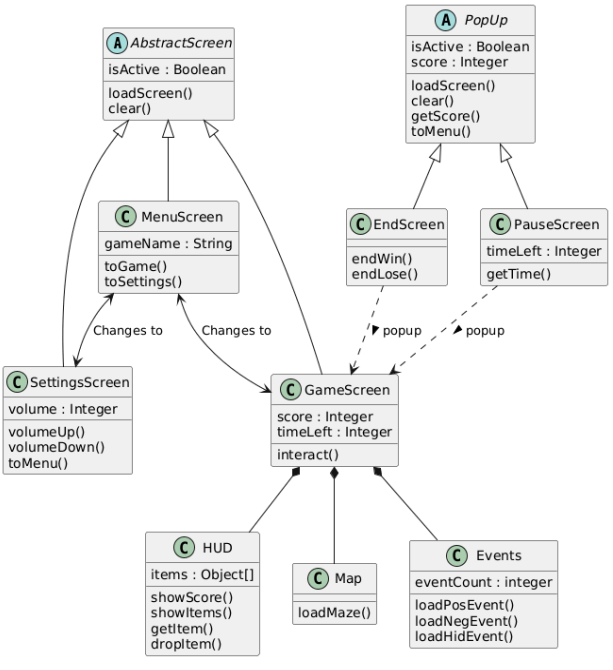
Once we were satisfied with the grouped CRC cards, we used them to start creating a diagrammatic model of our system. To create the diagrams, we used PlantUML, as it provides functionality and introductions for a wide range of diagrams, including Component diagrams, Class diagrams, and sequence diagrams. Furthermore, implementing these diagrams was made simple by the text-based input required to create the diagrams, and by the ease in copying and exporting the diagrams once generated. The UML code for all the following diagrams can be found on the website, here:
https://matth3wk.github.io/diagrams.html

We began by creating a component diagram, which provides a high level of abstraction from the actual program, but allows us to visualise how the key components and themes will interact with each other. Using the themes decided previously, the following component diagram was created:



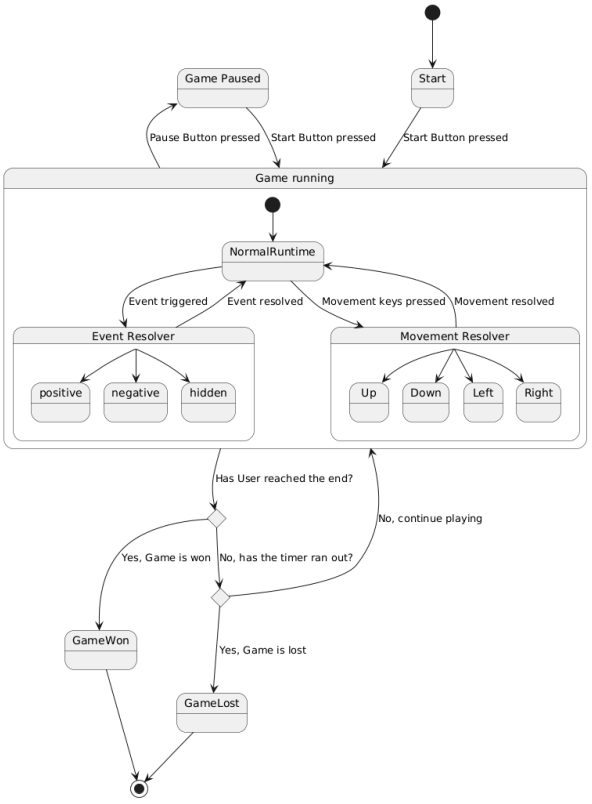
Using the component diagram, we then created class diagrams, which provides more detail than the component diagrams, and shows how the functionality described in the CRC cards will be implemented. We decided to include abstract classes for the screens in order to simplify the future implementation, as grouping classes with similar characteristics reduces the amount of repeated or redundant code.

Firstly, we created a basic diagram, containing the class names, and the relations between each class, shown in https://matth3wk.github.io/diagrams.html . From this, we added the attributes and methods, which provide the functionality of each class, while also adding complexity. We also completed all the relations, ensuring that each relation was correctly represented in the UML diagram, shown below:



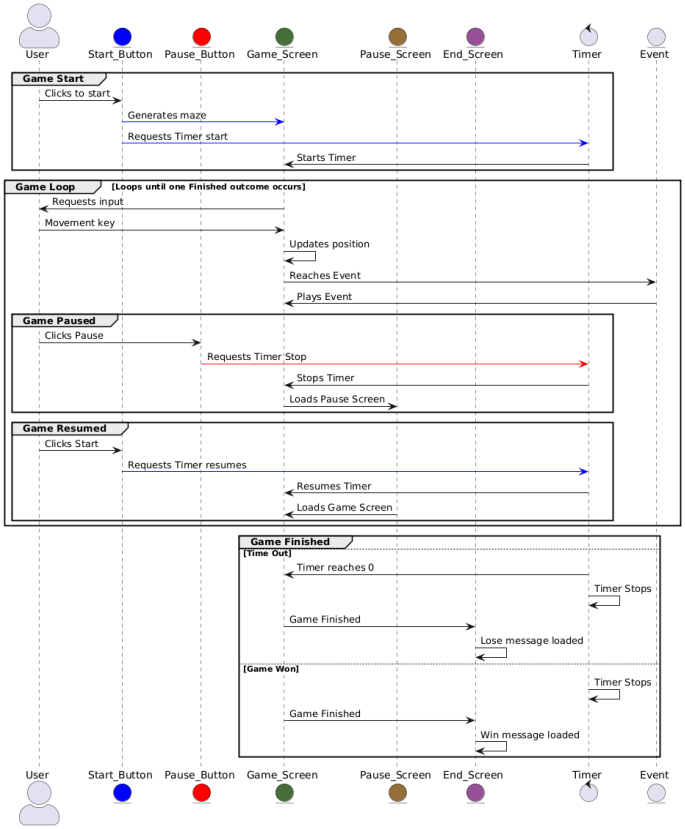
Now that the key structure of the system had been defined, we began to outline the behaviour of the program, through the use of state and sequence diagrams. This allowed us to understand the logical flow of the program and its components, while providing an abstraction of complexity, making it more simple to understand.

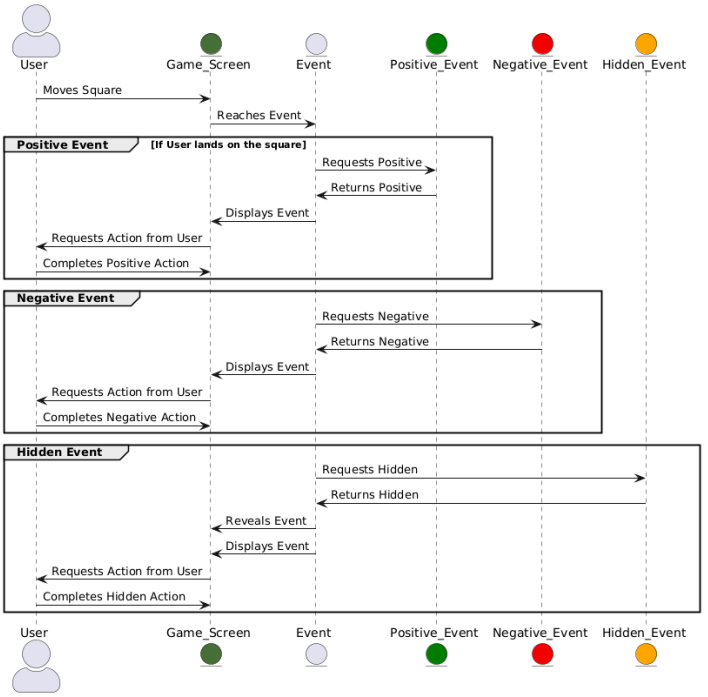
The state diagram is used to show the different states that the program may enter during runtime, and allows us to trace each possible path through the program. As shown below, the game begins in the start state, before moving into the game running state. This is in order to make it more user friendly, allowing the player to choose when they begin the game. The game then moves into the game running state, which the system will use the most throughout runtime. Within this state exists a substate called NormalRuntime, which represents the initial state of the loaded game, with the map, items and events loaded. This provides a base state which the other runtime states can return to after completion. Furthermore, there are Movement and Event resolver states, which the program will move into when there is input from the user, or an Event is triggered. As these require functionality beyond the base layout, it was decided to create separate states to ensure modularity and ease of understanding. Finally, there are several choice states, which run concurrently with the game running state, which determine the win/loss condition, as per requirement UR_END. These states determine whether the player has reached the end, and therefore won, or if the timer has run out, and therefore they have lost.



Next we created sequence diagrams, to show the logical flow of the program, and how different components are used during runtime. We began by creating a step by step sequence of the different interactions within the program, using the main components, mainly the screens, events, and the timer. We then grouped these sequences into phases,

with each phase corresponding to a state in the state diagram. However, we decided that the Events sequences were too oversimplified, so we created a separate sequence diagram showing how the Events are loaded and processed, shown on the right.





Using these diagrams, we can now begin to implement the program, using an iterative approach.