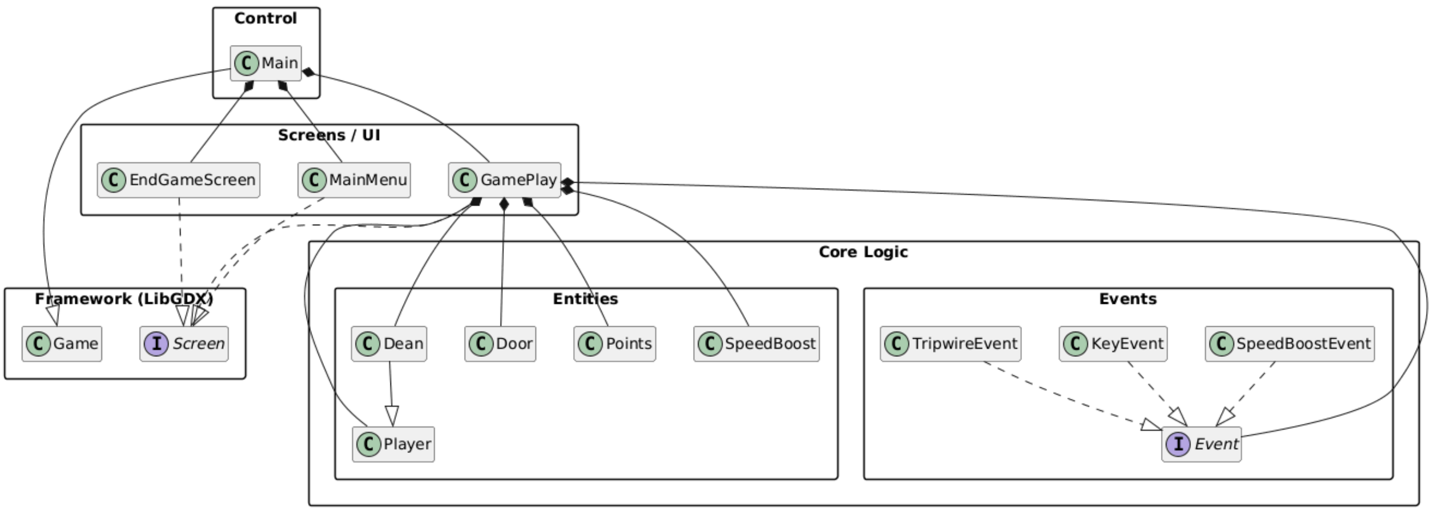
Architecture

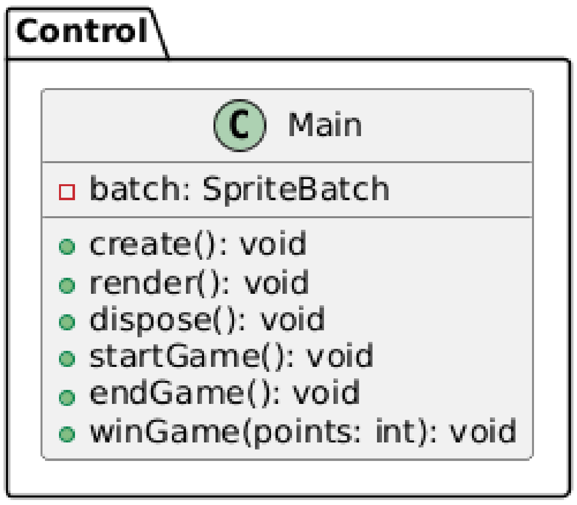
Cohort 2 Team 5

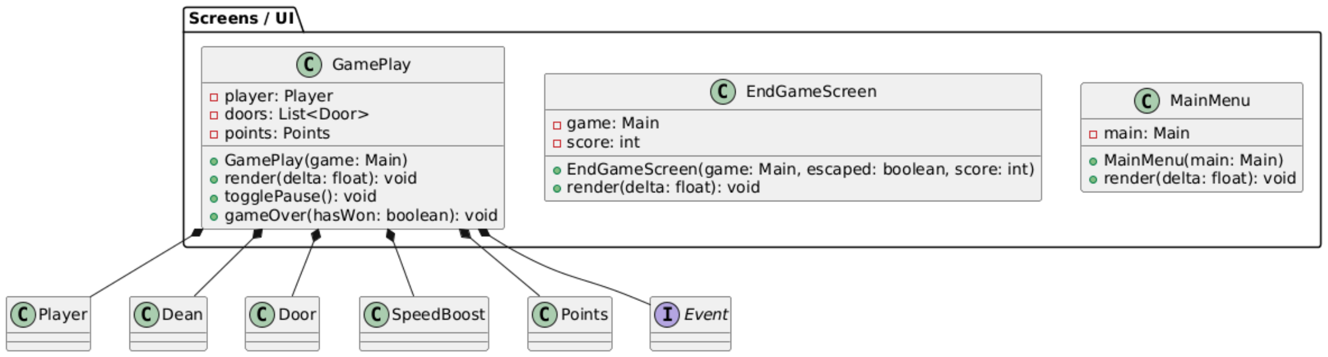
Harry Beaumont-Smith
Tom Nolan
Will Punt
Ruth Russell
Mimi Shorthouse
Lottie Silverton
Stanley Thompson

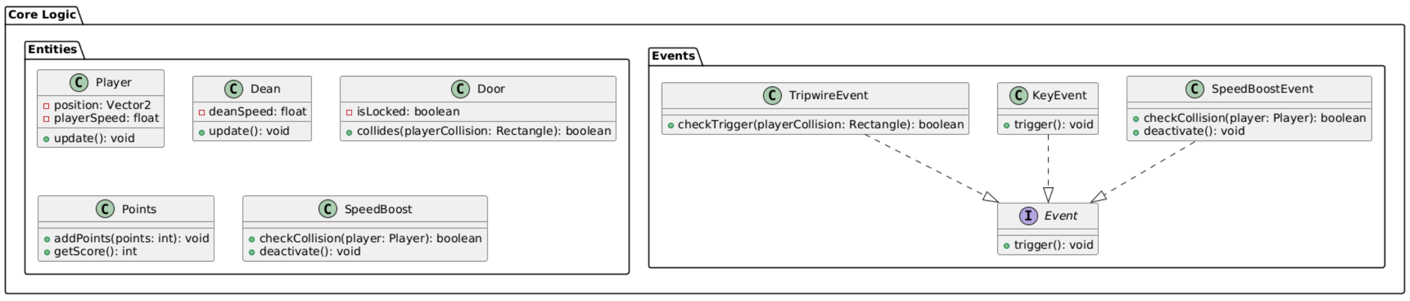
The contents of this document will contain the inner workings of our group's game including the various diagrams both class and user centered. The document provides an overview of how this game was structured as well as how the organisation of classes changed. The game was built using a monolithic object-orientated architecture with component-like qualities. While the entire codebase is deployed as a single application and meant to run at once, as indicated by the monolithic aspect of the structure, many of the classes were designed to be self-contained and modular. This in turn was achieved through the use of composition and through the use of interfaces where possible. Despite this some of the structures were built to extend others such as our Dean class which extends the Player, showing some dependencies between entities.

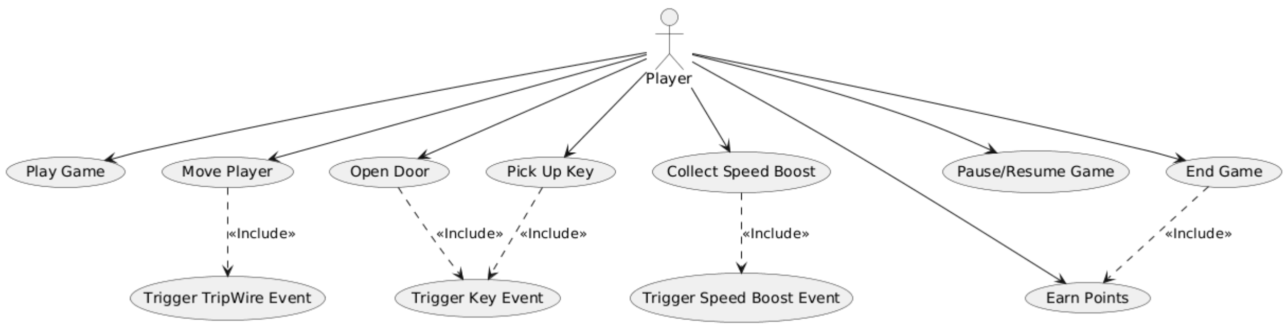
Moreover the structure was set up in layers in order to try and improve the cohesion between classes and in turn improve its modularity and maintainability. The layers involved a core game logic layer, which handled the mechanics and workings of entities, an Events layer, which handles events and a Screen layer which in turn handles gameplay screens. Finally, the design approach used is object-oriented, with a focus on modular, component-like entities that facilitate flexible interaction and future expansion where appropriate. As previously stated this document will look at visual representations of the structure, its justification as well as the tools used. Moreover it will also outline the process of development showing the intermittent steps as well as the traceability to the user requirements.











Above you can see the structure of our architecture which as mentioned in the introduction is a monolithic layered architecture which incorporates composition in order to improve modularity. The initial diagram is an overview of the entire structure whilst the succeeding diagrams are set up to show more detail about the classes and interfaces, such as their attributes and methods. Whilst the diagrams may not encompass all the methods or attributes it aims to show the methods which are most used in other classes as they are the most important. The UML elements used in the diagrams include classes and interfaces, as well as relationships between them such as inheritance, implementation and association (including composition). Then below the structure diagrams is a use case diagram outlining how the player interacts with the game. It shows how the core interactions the player has with the game such as moving, opening doors and the end game. The diagram also highlights how certain aspects of our code work together such as how certain events are triggered. The primary tool used in order to achieve this was UML with UML 2.0 standards.

The reason behind our choice of structure was to create modularity when it came to building the game, something we valued deeply. As a result the structure we chose aimed to maximise modularity allowing entities to be self contained and reusable where possible. The idea that entities are self contained can be seen as certain classes, such as Player, encapsulates their own state behaviour. As a result each class is responsible for its own data and behaviour, which in turn results in logic being localised. This results in an increase of cohesion, making classes easier to understand as well as test as they stand alone. Moreover it also reduces the chance of bugs as changes to one entity rarely ripple through unrelated classes. In addition most interactions use composition, which can be seen in the fact that Tripwire references a Door to trigger the event instead of inheriting from it. Composition reinforces the idea of self containment, reducing tight coupling between the two classes. Moreover it also keeps dependencies flexible whilst keeping them explicit. In addition it also supports reusability which makes it easier to adapt and extend. Although there are many benefits in using composition the architecture structure does also include some inheritance when it makes things easier. In some cases it is the most logical choice, this can be seen as the Dean class inherits from Player. Whilst we aimed for modularity the use of inheritance showed the flexibility of our structure. It shows we allow inheritance when it simplifies the design, without over complicating the other entities. Furthermore by using inheritance it also reduces the duplication of code.

In addition to building modularity our game structure uses layers in order to help maintain self containment. The layers as previously described in the introduction contained the core logic, the events as well as the screens used for the user interface. The natural layering set out by the team progressed over time and eventually led to the current structure. Although despite this it was something we had a priority for as it improved modularity. Each layer within the structure was set up to establish a clear responsibility. It is through keeping these responsibilities separate, each layer can be understood, tested, and maintained independently. Moreover it also meant that the modification of one layer did not inherently impact another. A key example would be when we implemented a screen layer further along the structure development, and the modification did not impact the code within the events layer. In addition the fact that it provided independent modification further supported flexibility and safe evolution of the system over time. All in all the structure achieved a layered structure within a monolithic codebase, giving clarity and maintainability.

Finally when picking out which structure to use, the team decided on using a monolithic design instead of an entity-composition-system design, despite its benefits and suitability for game development. The logical separation of entities and layers, allowed our game to behave modularly without the complexity of ECS. The higher cost of ECS, especially on a smaller project like ours, was not seen as visible, despite its fit for game development. In addition our classes were grouped by function which means despite the game running in one executable, technical partitioning was preserved. An example of this preservation is the pause screen, which only affects the screen layer. This in turn showed that modular, layered behaviour can exist within a monolithic deployment. All in all this was simpler and lower cost then implementing a full ECS system as previously mentioned.

The start of our architecture process began by first creating CRC cards in order to work out how we would go about creating the required classes for the game. This resulted in working out the responsibilities of the classes as well as their collaborators which initially gave us a brief outline of how the layer of the structure would work. Following on from the

CRC cards we then created our first implementation of a class diagram although this contained many classes we eventually did not use or adapted into something else. This can be seen in the snapshots provided in the website. The initial design of the game relied on a reduced number of layers and was heavily inheritance based, in which most entities stemmed from a single base class. As a result of the heavy reliance on inheritance there was minimal usage of composition, resulting in low modularity as well as flexibility. This resulted in a high connascence, as changes in the base class often required a change to the classes which inherited them. As a result it made modifications to the game far more difficult, resulting in the codebase being hard to maintain.

Over time however, the architecture of the game included additional layers on top of the existing main logical layer. This can be seen by the implementation of separate packages and closer grouping within the snapshots. A clear example of this, as seen in the website snapshots would be the addition of a screen layer, something which had been previously dealt with within the logic layer. In addition to the increase in layers, the architecture of the game also developed to use composition over inheritance where it was appropriate, something which can also be seen within the snapshots as well. This can be seen more specifically as entities such as Tripwire changed to interact with the door through composition rather than extending it. As a result this reduced coupling and improved modularity. This evolution resulted in a more flexible and maintainable architecture which in turn supports extension without introducing tightly coupled dependencies. In addition to the implementation of association such as composition, the move away from inheritance can also be seen in the snapshots through the fact that the Event abstract class was changed to an interface. The move to an interface increased the modularity and as a result improved the flexibility and maintainability of the code. Moreover it also enabled the possibility of expansion, something we were very mindful to consider and pursue.

To ensure our game met all the functionalities the user wanted we created requirements. The user requirements informed the non-functional requirements, which are the qualities a system must have, and the functional requirements, which define specific functions a system needs to perform. We used these to guide our design, to ensure that the user requirements were all met.

The functional requirements helped us determine what functions we needed to include within the system. For example, the FR_SCORE requirement says that the user should receive a score at the end of gameplay. This requirement led us to creating score as an object. Once the endGame had been triggered the game should lead to an endgame sequence which would display the score from the object. This also ties in with the requirement FR_STATE which lets the user know whether the player has completed the game or not. The functional requirements FR_PAUSE and FR_INSTRUCTIONS helped inform us of the different interface screens we'd need and how they'd interact. The requirements also meant we'd need to create a function that paused the game. We also had the very important requirement FR_TRIGGER_EVENTS which specified that the user should be able to interact with multiple types of events. For the different events we created different objects. To fulfill this requirement first we designed a basic event class as an abstract and then created the other types of event to extend the abstract class and build on that.

As you can see, from a few of these requirements, they helped us initially map and plan out the different aspects of the game, whilst also ensuring that we contained the functionalities the user initially wanted us to include. We also created non-functional requirements which enabled us to put constraints on the functional requirements and specify the qualities that a system should have. This enabled us to further identify our functions and adapt our architecture to become more suitable for the game we were designing.

For example the requirement NFR_Replayability meaning the game should be replayable. Due to this requirement we changed the design of the User Interfaces so that after finishing the game you can also restart the game.This meant that we had to adapt the start game feature so that it could be reused whenever the player finished the game too. There is also a requirement NFR_Usability which states that the way the user plays the game should be intuitive and understandable. This ties in to the functional requirement FR_PLAY which is that the user should be able to move the character around. Combining these means that we could determine how the user interacts and plays the game whilst adhering to both these requirements. Due to the non-functional requirement further specifying the functional requirement we could easily determine how the user moves with basic keys used to control the player through the maze.

To summarise, we used the non-functional requirements to help us specify and further influence the way we designed the structure of the game. Throughout iterations of the game we did alter the structure slightly but we ensured this was all done in agreement with the requirements we created. This enabled us to create a game that agreed with the initial user requirements and had all the functionality that was initially discussed.