Architecture

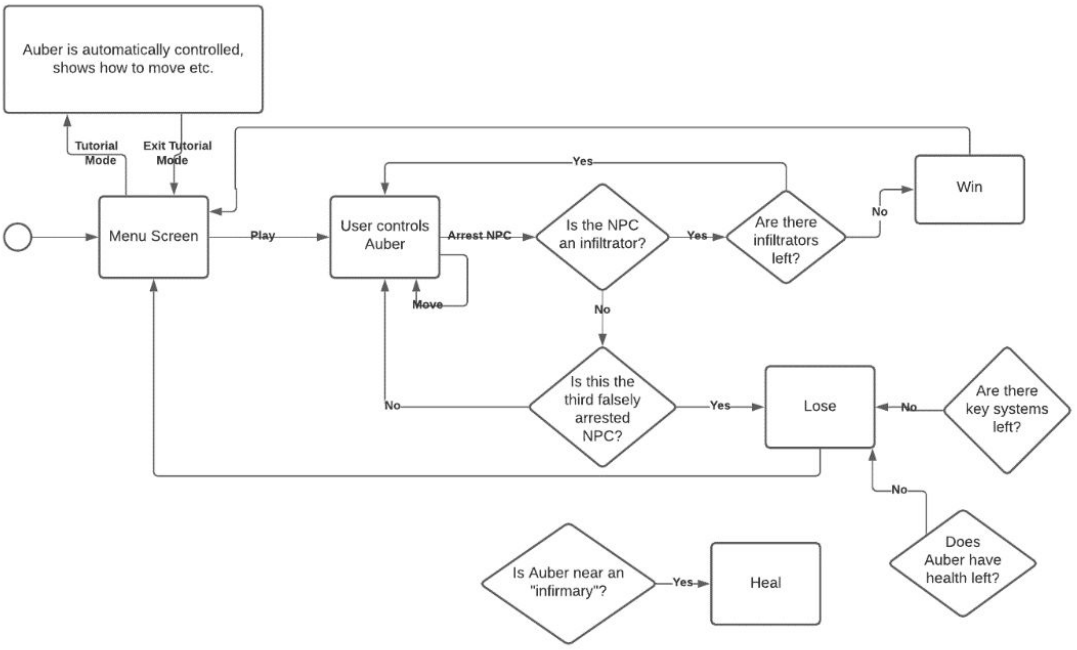
TEAM 32

Team Members:

Ethan Higgins, Lakhan Mankani, Clara Hohenlohe,
David Thomson, George Burke, Adam Gurdikyan

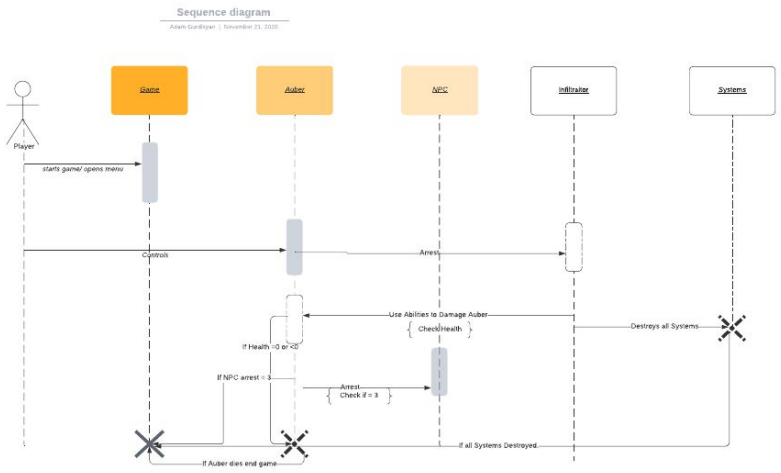
Representation of Architecture

State Diagram:



For this state diagram the different boxes represent each state that the game can be in. The arrows represent different state transitions, whilst the diamonds represent the different control structures and finally the text represent the events that occur to trigger the transition.

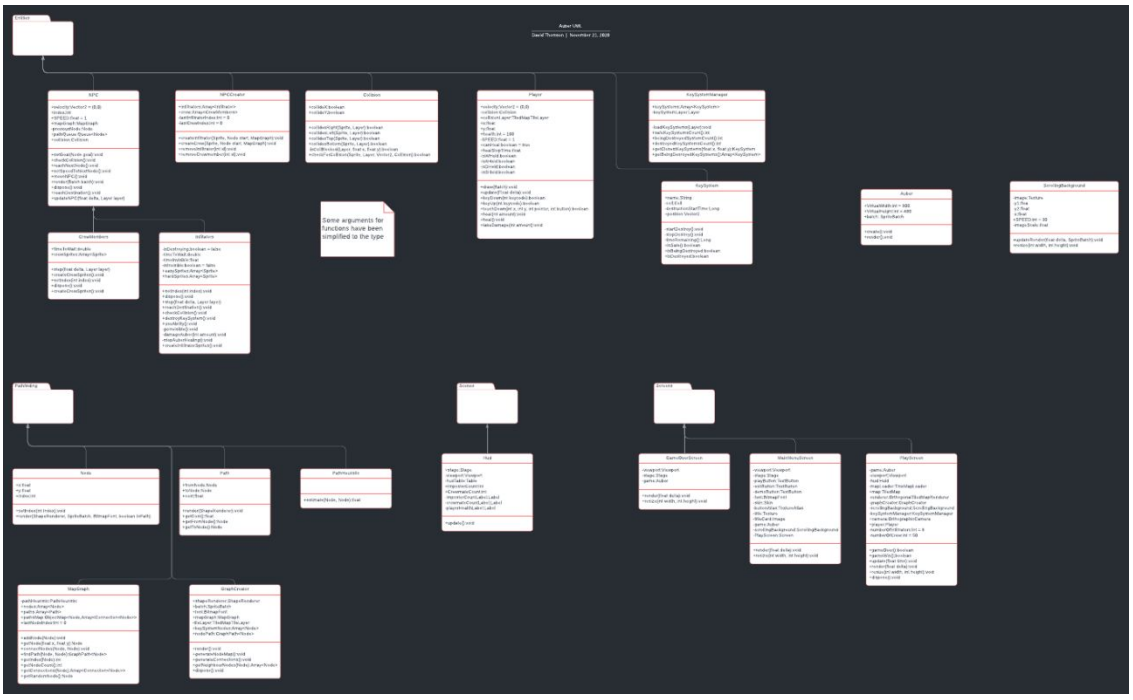
Sequence Diagram:



For this sequence diagram the vertical dotted lines represent an objects lifeline, the rectangles with the text inside them represent the object themselves. The arrows from each object with the text written on them are messages that one object gives another. The rectangles along the dotted line

represent the object's activation. Finally, the crosses and the end of an object's lifeline represent the termination of that specific lifeline.

UML Class diagram:



In this diagram a class is represented by a box which has three compartments: name, attributes and operations. The top compartment of the box is responsible for the name of the class; the middle compartment is responsible for the class attributes; the bottom compartment of the class is used to represent the operations of the class. The "+" signifies that the class attribute is public. The "-" signifies that the attribute is private. The default class is public. The arrows represent inheritance, the arrow will point to a more general class. The white writing is just for reference.

Justification of Architecture

Our group decided on having two abstract diagrams and one concrete diagram to structure our game. The two abstract diagrams are a state diagram, which describes the different states that the game can be in, and a sequence diagram to show how the different objects interact with each other within the game. This gave us the foundations we needed to help build our concrete structure as well as helping us stay within the given requirements.

Firstly, our group decided to use a state diagram as our first abstract diagram as it has an easy and straightforward way of developing the understanding of the different states that the game can be in.

The state diagram allowed us to stay within some requirements as well as laid the foundations upon which different classes and attributes would be needed in our concrete architecture. For example, we built the state diagram in such a way that you could not be able to reach the end game state without going through the "arrest an NPC" transition function. This therefore made it obvious that we then needed to implement an arrest attribute as a part of the player class, which then aided us in

building the concrete structure. Which then meant that we created the touchdown method in the player class. This follows the (ID: UR_ARREST) requirement that states game shall allow you to arrest NPCs.

Moreover, we added a state transition from the first state to allow the player the choice of whether to play the demo or not. We knew we had to add this to the “MainMenuScreen” class as this was made clear in the state diagram by the “Menu Screen” state having a transition to the “tutorial mode”. This led to the creation of the “demoButton” attribute within the “MainMenuScreen” class, which ended up fulfilling (ID:FR_DEMO) a requirement that expected the system to allow a short simulation of how the game is to work to play. Overall, the state diagram provides a simplistic view and approach for the creation of the concrete diagram. It is straightforward and easy to follow and shows us exactly what part of the system we need to be focusing on which makes it a great and successful choice as an abstract diagram.

Secondly, our group decided to use a sequence diagram. We decided to use a sequence diagram as it illustrates how objects and components interact with each other in order to complete the processes of the game.

Sequence diagrams provided us with a straightforward way that we can visualise the ways that different entities should be able to interact with each other. For example, the sequence diagram clearly illustrates the different scenarios that could occur for the game to end. This part of the sequence diagram also assisted us in building the different attributes required for the game to have in order to be able to end, which helped us create parts of our concrete UML class diagram, such as the “takeDamage” method in the player class, or the “isDestroyed” method in the key systems class. This helped us satisfy the “FR_ENDGAME” requirements, which states: the system shall allow the game to end and not to be played forever, as the game would end if Auber takes too much damage or if enough key systems are destroyed.

Another example showing why the sequence diagram was a good choice is how it shows different messages that different entities send each other when they are interacting with each other. The “damageAuber” method stems from the “use abilities to damage Auber” message shown on the sequence diagram, this example clearly shows the utility of the sequence diagram, by demonstrating how a simple message and line can be turned into a crucial part of the game. It also helps the system fulfil the functional requirement to allow Auber to heal (ID:FR_HEAL). Therefore, we believe that our use of a sequence diagram is fully justified.

Finally, for our concrete diagram of choice was a UML class diagram. A UML class diagram can be used to model the structure and the behaviour of a system, it also makes it easier to give a lot more detail about the classes, attribute and methods, than other diagrams, like a sequence diagram.

This can be seen just by looking at the two diagrams, the UML class diagram gives a vast amount of detail about the menu screen, whereas the sequence diagram barely represents it through a message line. This shows how the UML class diagram gives us more detail however is harder to read and understand. The UML class diagram gave us an overview of the structure of the code, so we could see what classes were needed and how to organise them, for example when creating the infiltrator and crew member classes it was important that they both inherit from the NPC class, as they both have the same code in some aspects, like movements, which means we didn’t have to repeat our code, which lead to a better structured and more organised code. This level of detail would be much harder to implement without the UML diagram, which in turn improved the quality

of the game, as a better organised code meant we had a better organised project overall giving us more time to make sure any requirements were dealt with to the highest standard.