Architecture

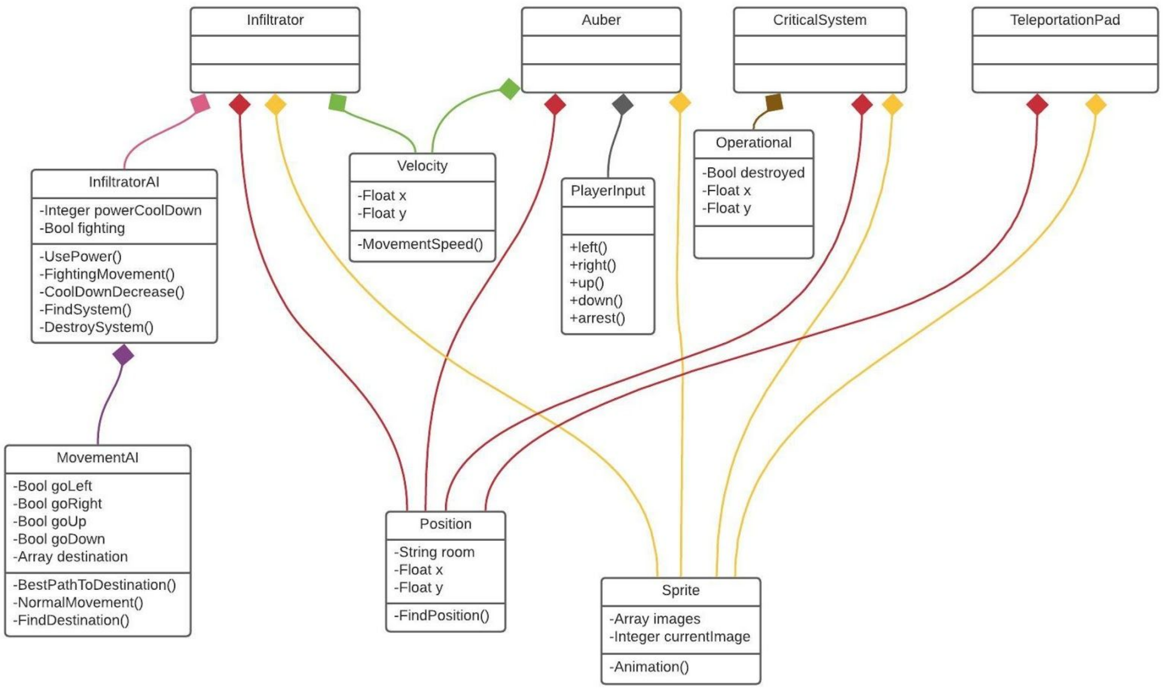
Team name: Endeavour (Team 28)

Team Members: Sarah Berry, Finley Brown, Yupeng Di, William Griffiths, Kurtas Joksas, Fraser Masson

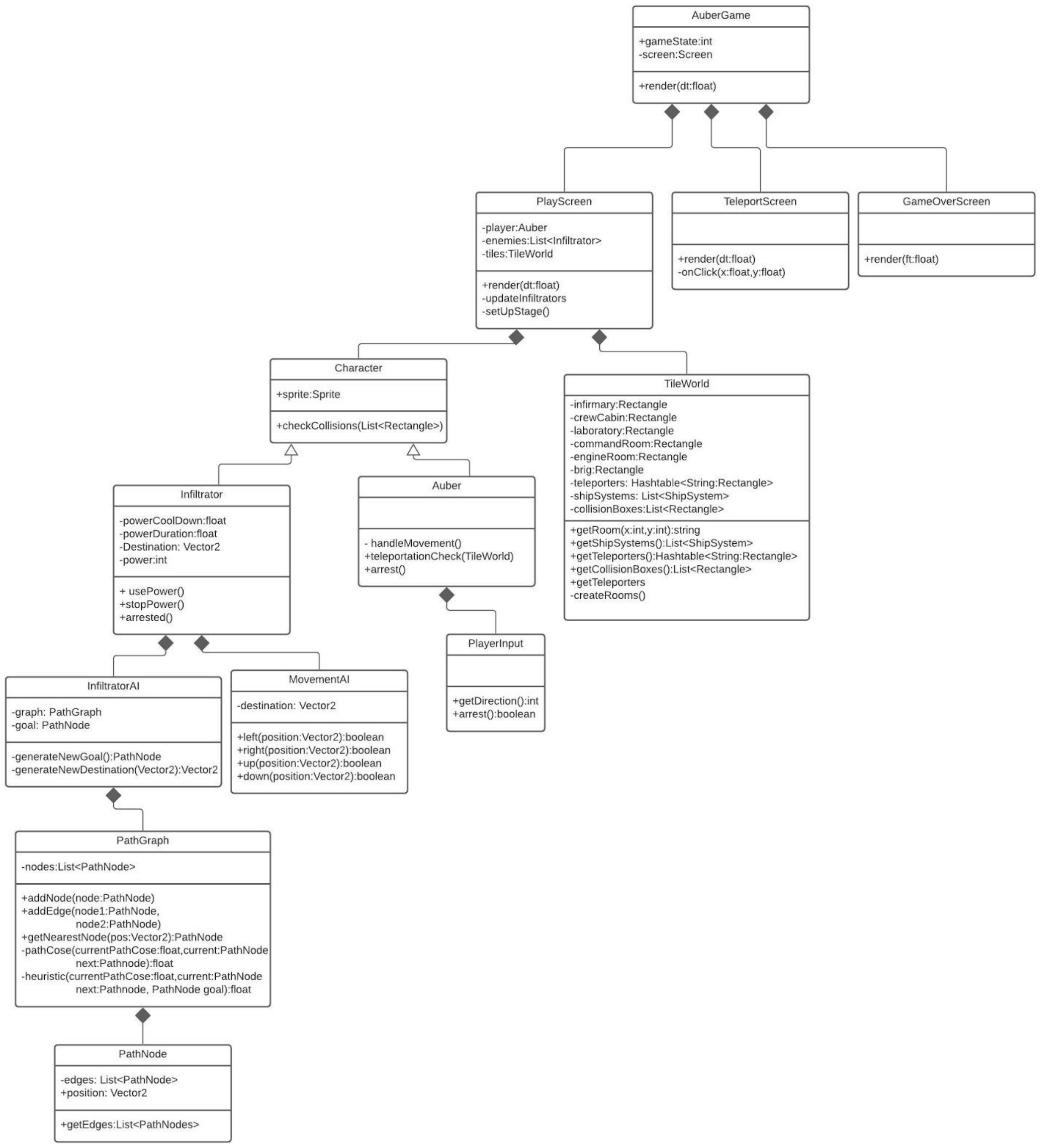
3a: Representation of architecture

We used lucidchart.com to create our class diagrams.Lucidchart itself is a software that can create all sorts of diagrams, from flow charts to class diagrams and more. We used lucidchart as using the organisation code of UML our diagrams were hard to follow, having connections grouping together making it hard to read. Lucidchart on the other hand makes easy to follow class diagrams that can be formatted to our specifications.
The language we based our class diagrams on was Java as it was the required language to code in.

<u>Abstract representation of architecture</u>



Concrete representation of the architecture



3b: Justification of architecture

Our abstract representation was based heavily on the Entity-Component-Systems pattern, favouring composition over inheritance. We chose this specific system as many current game development companies use this system as it is efficient, using less processing power, and is easier for the coder to implement. This is shown through the classes that use multiple of the same sub classes (e.g. velocity, sprite and position) however also have separate sub classes making them different. The sub classes then allow extra classes, for example a civilian class, to be implemented easier. The civilian classes can be composed of these predefined classes whilst being able to also use its own bespoke subclasses.

The abstract representation gave us a good idea of which classes needed to exist and how they would interact with each other. However when it came to implementation we found that we gravitated towards a more object oriented approach. This was due to the groups familiarity with it from our prior experiences using it. This meant that the final code uses a combination of the two ways of approaching classes, causing a difference with the concrete representation.

The concrete representation still builds on the foundation of the original abstract representation. Classes such as MovementAI, InfiltratorAI, PlayerInput etc. still have a key role in the concrete representation staying relatively the same but with more information and showing more specific methods (e.g. PathGraph, PathNode in infiltratorAI, GetDirection in PlayerInput). However there are some changes from the abstract representation, clearly shown via the connections, as now some classes inherit from a parent class. These classes are Infiltrator and Auber, as in implementation we found that it was easier to approach these two elements with an overarching parent class called Character.

The software we used in the development (ide intellij, game engine libgdx, language java) affected the concrete representation. Multiple variables were created that are specific to the game engine such as Rectangle variables like infirmary, crew cabin etc. making a rectangle of coords. This allows for collisions to be easier to code. New classes were created that did not exist in the abstract class as it is easier to program with these new classes. These classes are more specific with their variables and methods, (e.g. parameters) shown in methods getShipSystems():List<shipSystems> and getRoom(int x, int y):String. These show the format the methods would be in and the type of inputs and outputs they would have. Variable types are also specific to the language and game engine, (e.g. Rectangle, Hashtable, List etc.). This gives a better understanding of how the variables and methods would work in the context of the language and engine.

Concrete Architecture and our Requirements

In our concrete representation, we show how the player will interact with the game using a keyboard as per the FR_MOVEMENT and NFR_MOVEMENT_RESPONSE requirements. The player's input will be detected in the PlayerInput class and handled in Auber class in the method handleMovement() to move the auber on the screen.

For FR_ARREST (Auber can arrest enemies by holding down the space button), the detection of the space bar press is in PlayerInput class with the arrest() method. It then arrests the infiltrator using the arrest method in Auber class and the arrested method in Infiltrator.

For our game we decided to use a tile map which had MapObjects. This means we could find the locations of the teleporters in order to fulfil the FR_TELEPORT_PADS requirement. The location is found in TileWorld class and stored in a hashtable, teleporters, where the key is the room name and the value being the location. This hashtable can then be checked against the Auber location in teleportationCheck(), in the Auber class, where it sees if Auber is standing on a teleportation pad. The TeleportationScreen is then shown and allows the user to pick a destination. The auber is then teleported to the destination using the hash table coordinates,

In addition we designed the implementation of FR_INFILTRATORS_AI which dictates how the infiltrator will find and destroy a system. A graph PathGraph is created to allow the infiltrator to traverse the contained PathNodes which represent either a system or a doorway to another room. This lets the infiltrator find a path from their current location to another room and a corresponding operational system.

The type of power the infiltrator has is kept in the power attribute of the Infiltrator class. We could have implemented separate classes, one for infiltrator and one for every type of power, however to avoid inheritance we decided to store it in a variable. When usePower() is called, the corresponding infiltrator power is called to effect. For FR_INVISIBILITY and FR_SHAPESHIFT the infiltrator texture is changed. For FR_HALLUCINATIONS a hallucination is caused on PlayScreen and for FR_SPEED_BOOST the speed of the infiltrator is increased in MovementAI.

In order to implement the requirement of not allowing the infiltrators to repeat their power till a cooldown period has ended in FR_SPECIAL_ABILITIES, the Infiltrator class has a powerCoolDown attribute which holds the time since the last time the power was used. This is updated using the delta time in render from PlayScreen class.

We implemented a GameOverScreen class which is displayed from AuberGame when the game state is 2 or 3 (win or lose, 1 being what the screen is whilst playing). This displays to the player if they have won or lost (FR_GAME_OVER).

The location of the 6 unique rooms as per the FR_UNIQUE_ROOMS requirements are defined in TileWorld as Rectangles. These rectangles can be used to check what room an object is in. For example it can be used to check if Auber is in the infirmary to heal as per the FR_INFIRMARY requirement.

References
Mark Jordan, 2018, Entities, Components and Systems,
https://medium.com/ingeniouslysimple/entities-components-and-systems-89c31464240d