UNIVERSITY OF YORK

DEPARTMENT OF COMPUTER SCIENCE

ENG1 Group Assessment 1
Team 1

Auber

Arch1.pdf - Architecture

Group Members:

Jonathan Davies

Jamie Hewison

Harry Smith

Annie Sweeney

Zee Thompson

Mark Varnaliy

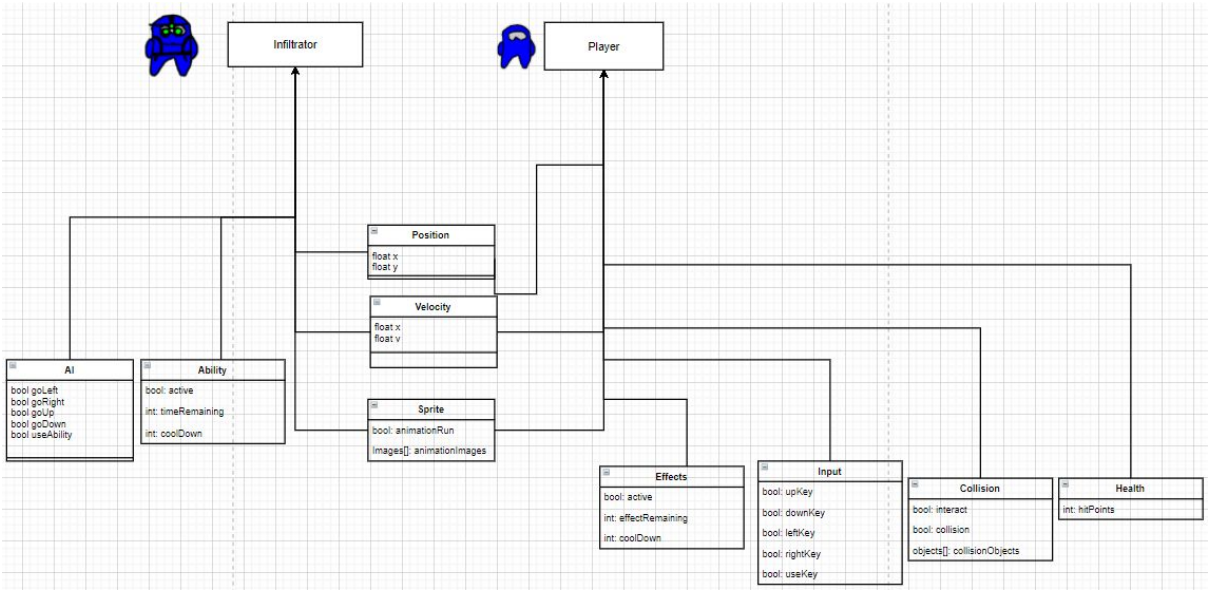
Abstract and Concrete Architecture Representation

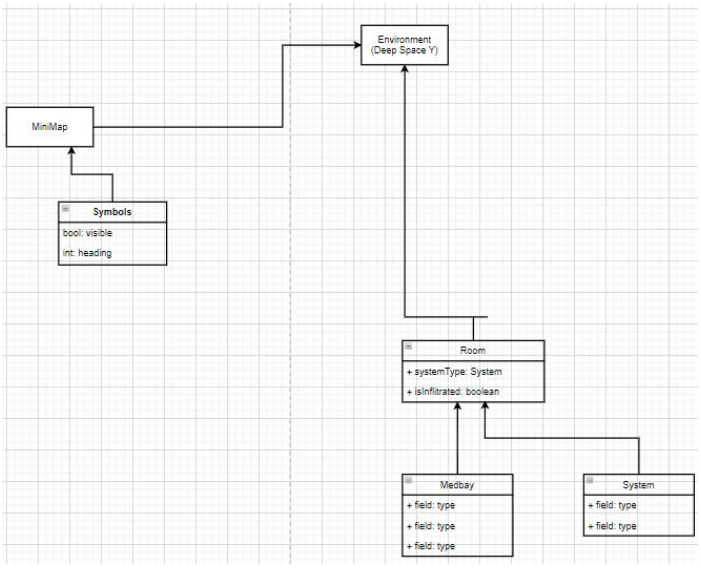
Abstract Representation:

To display the abstract representation of our architecture, we decided to use a UML diagram.

We used https://diagrams.net/ to create our UML diagram as it integrates well with Google Drive, allowing all of us to work on the UML diagram simultaneously.

Below is our UML diagram:

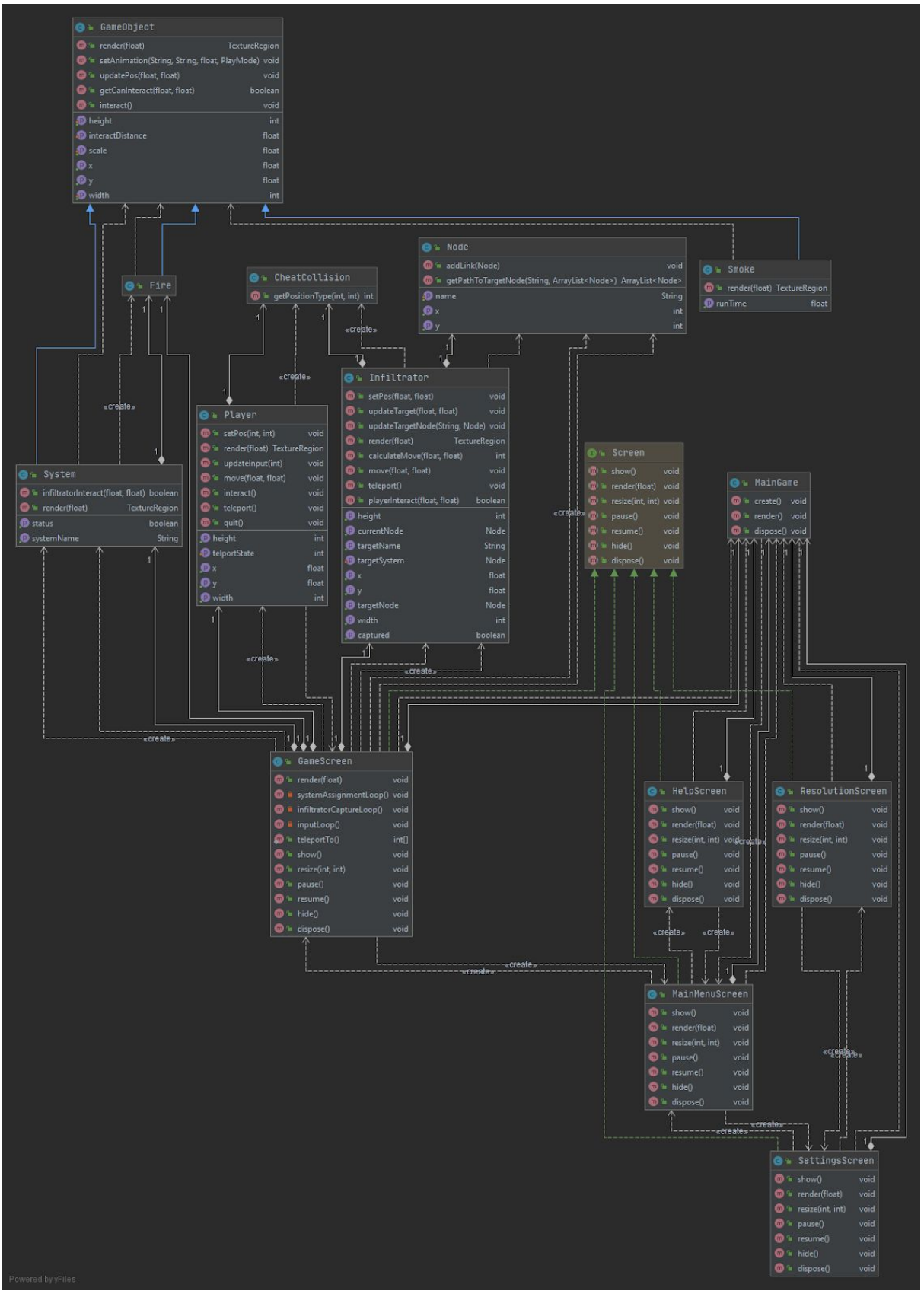




Most of the language used in our UML diagram is fairly self explanatory as it is standard terminology (bool, int, etc.). There are, however, a few things may require additional information. Under ‘sprite’, we have used an array images[] to hold all the images used to animate the movement of our characters. Similarly, under ‘collision’, we have an array objects[] which holds collisionObjects - essentially a list of objects that the sprite is able to collide with.

The final piece of terminology which may require additional explanation is ‘systemType: System’ under ‘Room’. This is using a data type named ‘systemType’ to represent what type of system is located in that specific room.

Concrete Representation



To give a concrete representation of the architecture of our software, we used a UML diagram generator to create a clear picture of our project. This generator converted our Java classes into the diagram shown above, therefore those who can program in Java should be able to intuitively understand the system architecture with assistance from the diagram. The tool we used comes as part of the UML plugin for IntelliJ, which is installed by default.

Systematic Justification of Abstract and Concrete Architecture

Looking at our abstract architecture, as shown by our UML diagram, we believe that our representation clearly shows how we intend to implement our game. Each class is clearly shown, along with the values that each class needs to hold in order to allow our game to run effectively. The UML diagram also shows how each class relates to each other, and how these are used by the sprites in the game.

Our concrete representation of the architecture is also a UML diagram, but created using the IntelliJ IDEA, which displays all the classes created and the relations between them. For example, ‘System’ is related to ‘GameObject’ by extending it, whereas ‘Player’ is related to‘CheatCollision’ as it is composed of it. It also describes the methods and attributes used in these classes, along with data types used by the attributes and returned by the methods (and whether these are public or private).

The concrete architecture builds from the abstract architecture in a number of ways. For example, with the player and infiltrator classes. Both share similar attributes: x and y coordinates, and methods to manipulate these coordinates (‘position’ and velocity’ in the abstract UML can be directly related to ‘move’ and ‘setpos’ in our concrete UML diagram). They also both use the ‘CheatCollision’ class specified as a ‘Collision’ method in the abstract UML diagram.

The ‘teleport’ function in the player and infiltrator in our concrete diagram were built off the ‘position’ function in our abstract diagram as well. The teleport function changes the x and y coordinates of the player, which the ‘position’ function uses in the abstract representation.

The ‘Sprite’ method in the abstract diagram that specifies some sort of animation function is also implemented in our concrete representation in both the player and infiltrator classes. It is implemented as a function named ‘render’ and uses the ‘TextureAtlas’ and ‘TextureRegion’ objects, imported from the libGDX library, instead of an array of texture images.
‘Render’ also builds off the ‘Input’ method in our abstract UML diagram. It checks the key pressed by the player, and calls the function ‘move’. The movement keys are implemented as ‘WSAD’ and the ‘use’ key as ‘T’ for teleporting.

However, it is clear that more detail was needed in some cases. For example, with the infiltrator, we needed to add more methods and objects to create an AI that worked as required. In our abstract UML diagram we wrote ‘AI’ to indicate what it should look like and a number of attributes, however we needed to create a class called ‘Node’ that would do the

main calculations and a number of methods (updateTarget, updateTargetNode, calculateMove) to implement this.

We used the ‘Room’ part of the abstract diagram as inspiration for our concrete architecture. Instead of having a tangible room object, the rooms are represented graphically in the map png file. The systems are also represented in the map graphically (for example life support in the medbay as a console with hospital beds), however, the ‘System’ is also a separate object that extends the ‘GameObject’ object (shown in the concrete representation). The medbay was not implemented, however if it was, it would have most likely extended the system object, with some additional methods to heal the player.

Clearly, our concrete architecture is built from the abstract architecture in a number of ways. Where it could not, it largely took inspiration from it and implemented it in the most accurate way possible.

The requirement FR_TELEPORTATION is implemented as a method of the Player and Infiltrator classes, in the case of the infiltrator, to be teleported to jail fulfilling FR_ARREST, and also makes use of a static function to get the location of teleporters from the GameScreen class. FR_MAP is implemented in the GameScreen class also, although the collision between player and map is derived from the CheatCollision class which is used by both Player and Infiltrator. FR_CONTROLS makes use of the inputLoop() method from the GameScreen class to pass inputs on to the Player updateInput() method to allow for responsive control of the character. FR_Enemies is implemented as the Infiltrator class, although abilities are yet to be added. FR_Enemies’ implementation is also supported by the Node class which assists some of the Infiltrators methods to allow a smart enemy which targets systems. NFR_WINDOW is handled throughout all of the screen classes, where the screen is scaled to fit the resolution of the monitor it is being displayed on. This is especially notable for GameScreen where an Orthographic Camera method keeps the player at the centre of the screen.