ARCHITECTURE

NP Studios – Team 8

Team Members
Lucy Ivatt, Jordan Spooner, Alasdair Pilmore-Bedford, Matthew Gilmore,
Bruno Davies, Cassandra Lillystone

Architecture Structure

Tools, Languages and Resources

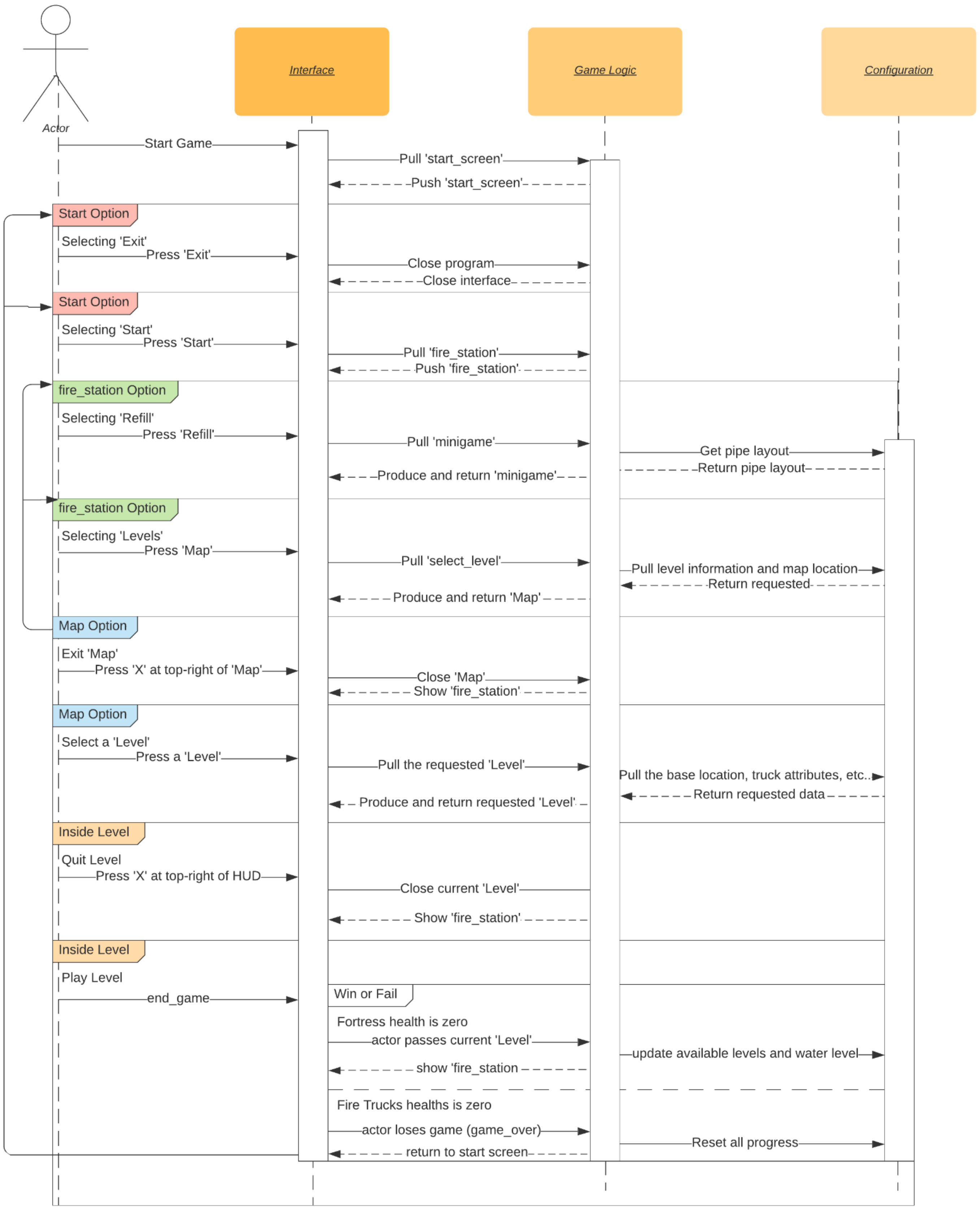
We used Unified Modelling Language (UML) to represent the proposed structure of our game. This is a graphical modelling language which is specifically designed for object-oriented languages. We chose UML because it seems the most relevant for this project as it's a very readable language and makes it easy for us as developers to understand the system we are creating. UML allows us to create an initial blueprint for our program by visualising classes and their relationships. By creating this, each member of the team can gain a reference point when we begin to program. It is a standard language, meaning there is a large amount of support available and is widely known by most.

The tool we decided to use is 'draw.io'. We chose this as our tool for creating UML diagrams as it is both simple and intuitive to use. As well as this, 'draw.io' can be integrated with Google Drive which was our primary storage tool we have used for our documentation. We thought this would be particularly useful as we could work on the UML diagrams collaboratively, allowing us all to contribute our ideas and keep them in a centralised document.

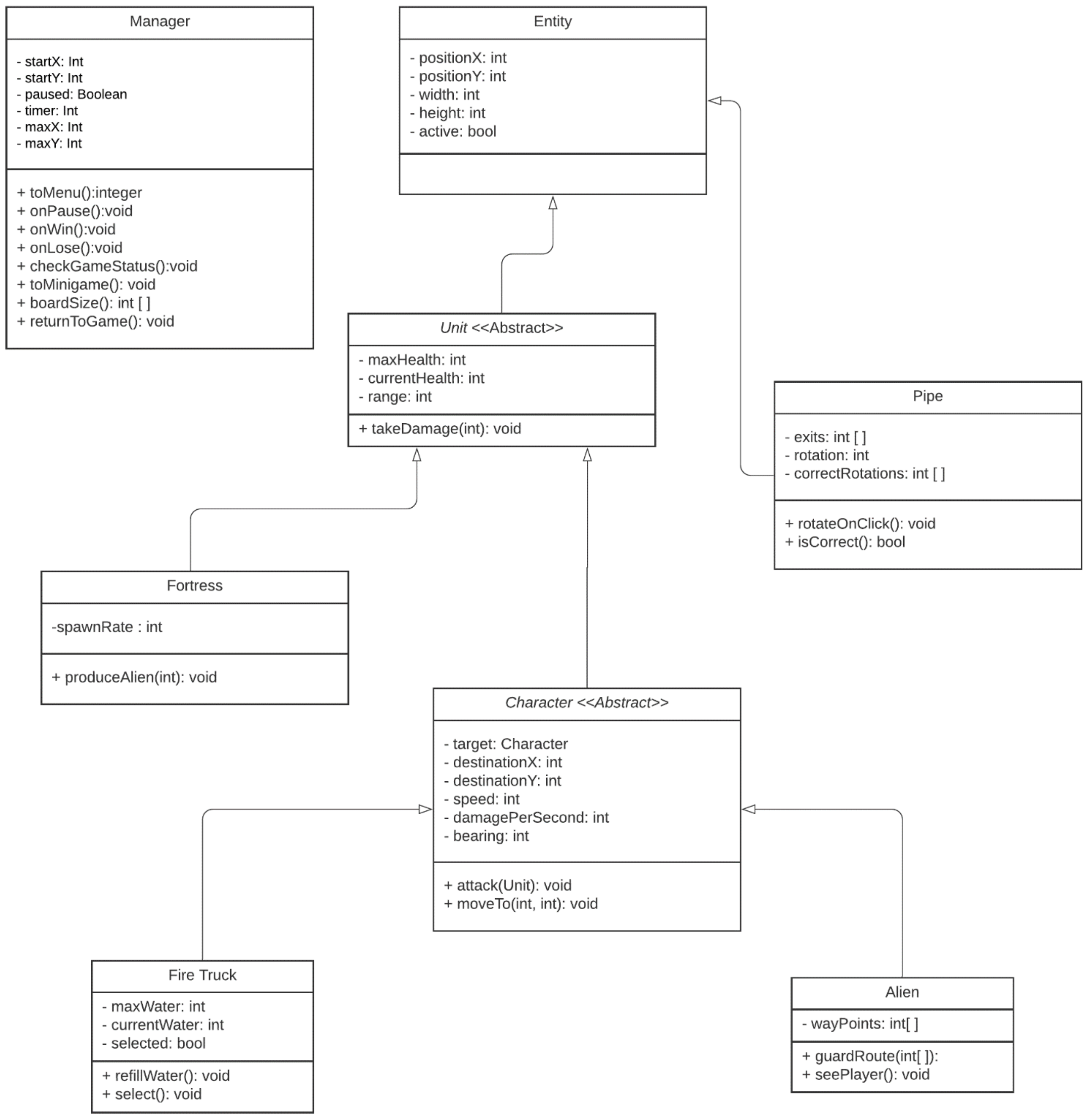
We decided to produce both a class diagram and a sequence diagram for this project. These two diagrams will provide both an overview of the program structure and how the objects within that structure will relate to and interact with each other. The sequence diagram also allows us to optimize our structure to only include the relevant classes as it will describe how each of the classes will work together. Any irrelevant classes or methods will be detected due to their lack of justified interactions in the diagram and can therefore be removed.

For the development of the UML Diagrams we used various resources to ensure our diagrams were correct. The most important and well-known resource we used was the Object Management Group's documentation [1]. They provide an up-to-date pdf containing information about UML such as abstract syntax and constraints. We referred to this document when unsure about the correct semantics and syntax. For example, we were unsure of the correct way to indicate an Array data type within our UML diagrams. Within the Object Management Group's documentation, we found the correct format (page 106) for them within UML and thus feel confident our UML diagram could be understood by others with an understanding of UML structure. For our sequence diagram we also reviewed the documentation to ensure our implementation was correct and conformed with the standards. (section 17.8)

Sequence Diagram



UML Class Diagram



Justification

Before creating a UML class diagram, we decided as a group to create a sequence diagram. The process of making this diagram was key to laying out the main 'flow' of the game and what the result of specific actions should be. While traditional sequence diagrams often use specific methods and classes, we used ours to bridge the gap from the ideas we had to what the class diagram should include. For example, in the sequence diagram we reference the minigame with pipes and the need to get the pipe 'layout'. This emphasised the fact that there should be a class to help assemble this information when the minigame is called. Then, after creating the 'end_game' case in a 'Level', we see again a class needed to help decide if it's a 'win' or 'lose' scenario. Because of this, we decided to use a 'Manager' class (in our UML Class diagram), which helps manage the game. This crucial class could have been disregarded since it seems obvious to the team what the game should do, but the sequence diagram forced us to formally consider how it should be outlined.

To ensure that our architecture would not become unusable we took to an abstract (general) approach. Abstraction allows for changes in implementation to occur without making this general structure obsolete. We had initially thought of specific game development tools, like rendering and game engines, however these could change when implementing. Through designing in an abstract and broader view the group grew to have a more aligned view on the project.

Abstract classes and inheritance will allow us to produce clean and efficient code by minimising code duplication and increasing reusability, helping us save time. The features will take both less time to implement and be easier to maintain throughout the project. For instance, if an edit or a fix needs to be made to a method which is inherited by multiple classes, the edit only needs to be made in the parent class.

Entity: This is a class used to define the position, size (width and height), and the Boolean state 'active'. The attributes position and size are needed to inform the game of which spaces are already taken up by entities, preventing the possibility of entities overlapping one another. The 'active' state specifies whether the object should be displayed on the screen, but at this stage of the project the rendering of objects is assumed to be done through the game engine and this implementation is left abstract. The active attribute is also set to false if a Unit (a class inheriting from Entity) is destroyed. This is a superclass, as everything in the game will need to have a position and size, therefore it will inherit these attributes rather than duplicating them. We decided that this class shouldn't be abstract as we would like to create instances of it for obstacles, such as buildings on the map.

Pipes: This class is used to create the pipe objects for the minigame. The pipes class inherits from entity but will have its own attributes which are 'rotation' and a list of integers called exits. Rotation defines the current rotation of the pipe and the list describes at what rotations water can exit the pipe (in other words the open ends). The correctRotations is an array that contains the correct rotations needed for each pipe to pass the game. The pipes have the method rotateOnClick() which is used to change rotation by 90 degrees to the right if clicked and the isCorrect() method returns true if the current rotation is included in the correctRotations array.

Unit: This is an abstract class used to define necessary overlapping attributes of Fortresses and Characters. It is an abstract class because there will be no instances from this class itself. The class contains health, maxHealth and range. All units require health so we can determine when a unit is destroyed. maxHealth is used in situations where the units may regain health either through level gimmicks or through health regenerating over time and range is needed to know where a unit is able to attack. Finally, the method takeDamage(int) reduces the health of the Unit.

Fortress: This is a class used to define fortresses, a unit responsible for spawning enemy units and are the ultimate objective for the player to destroy. The fortress needs a spawnRate attribute to determine how many aliens can be spawned over 10 seconds. The method produceAlien() spawns new instances of the class Alien within range of the Fortress's position defined by the attribute range inherited from Unit.

Character: This is an abstract class which outlines the main attributes and methods that the alien and Firetruck classes share. We decided that both the Alien and the Firetruck would have a speed attribute, a bearing, a dps(damage per second) attribute and an attackTarget(Unit object). The method moveTo() relocates the Character to the given x and y at the speed attribute. The attack() method uses the attackTarget attribute to get the health of the enemy target and reduce it by the dps amount during the attacking state. If there is no enemy to attack attackTarget is set to null.

FireTruck: This is a class needed to define the fire trucks in the game. This class needs attributes for water to determine how many attacks the fire truck can perform before it needs to refill. In the method refillWater(), currentWater is set to the value of maxWater. The method select() checks if the player has clicked on the Firetruck, if so that instance can be given a new destination or attackTarget() depending on where the player next clicks.

Alien: This is a class needed to define aliens in the game. We added the ability to set the aliens to follow a path to patrol the fortress. We named the attribute waypoints which are represented in the array data type. Through this data type we can have tuples of x and y coordinates that the aliens will follow in the order of the array. The method guardRoutine() checks if the Alien has reached the next waypoint, once this is true the method changes the next destinationX and destinationY location to the next element in the waypoint array. The method seePlayer() is used to check whether a firetruck is within the range of the Alien. If so the Aliens sets the fire truck as a target and will change destinationX and destinationY to the fire trucks current position.

Manager: This is a class used to define the current state of the game. The attributes startX and startY are used to define the position from where the player can return to the fire station and refill their truck. The Boolean paused is used to define whether the game is in pause mode and the timer attribute is used to measure the time since the level started. This will also stop counting when the Boolean paused is true. The manager also contains arrays which lists the active units in game. These arrays are used in the method checkGameStatus() to check if any units health are destroyed or should not be displayed and therefore can be set to inactive. This method will also activate the method onLose() if all FireTrucks have been set to inactive or onWin() if all enemy Fortresses are inactive. The method toMinigame() changes the screen to the minigame screen, sets the position of pipes on screen, and sets all units on screen temporarily to inactive while the minigame is being played. The method returnToGame() returns to the main game, either with max water level or not depending on if the user succeeds. toMenu() is needed to return the player to the main menu screen. The method boardsize() sets the parameters of the boards size which prevents entities from appearing or moving outside of this region. onPause() sets all units to inactive and transitions to the pause screen.

Other class Instances: Obstacles: These are created from an instance of the class Entity. Obstacles define impassable areas on the in-game map. An obstacle only requires the attributes and position and its state does not change during the level. This justifies our decision not to make Entity an abstract class as we need to create instances of it which will act as our obstacles for the game.

References

[1] Object Management Group, “OMG® Unified Modeling Language® Version 2.5.1,” 2017.