SEPR

SpaceKey Projects

Assessment 1

Architecture

JORDAN CHARLES

SAMUEL HUTCHINGS

CHLOE HODGSON

GOLNAR KAVIANI

TAMOUR ATLAF

JACK THOO-TINSLEY

2019/2020

Architecture

a) Abstract Representation

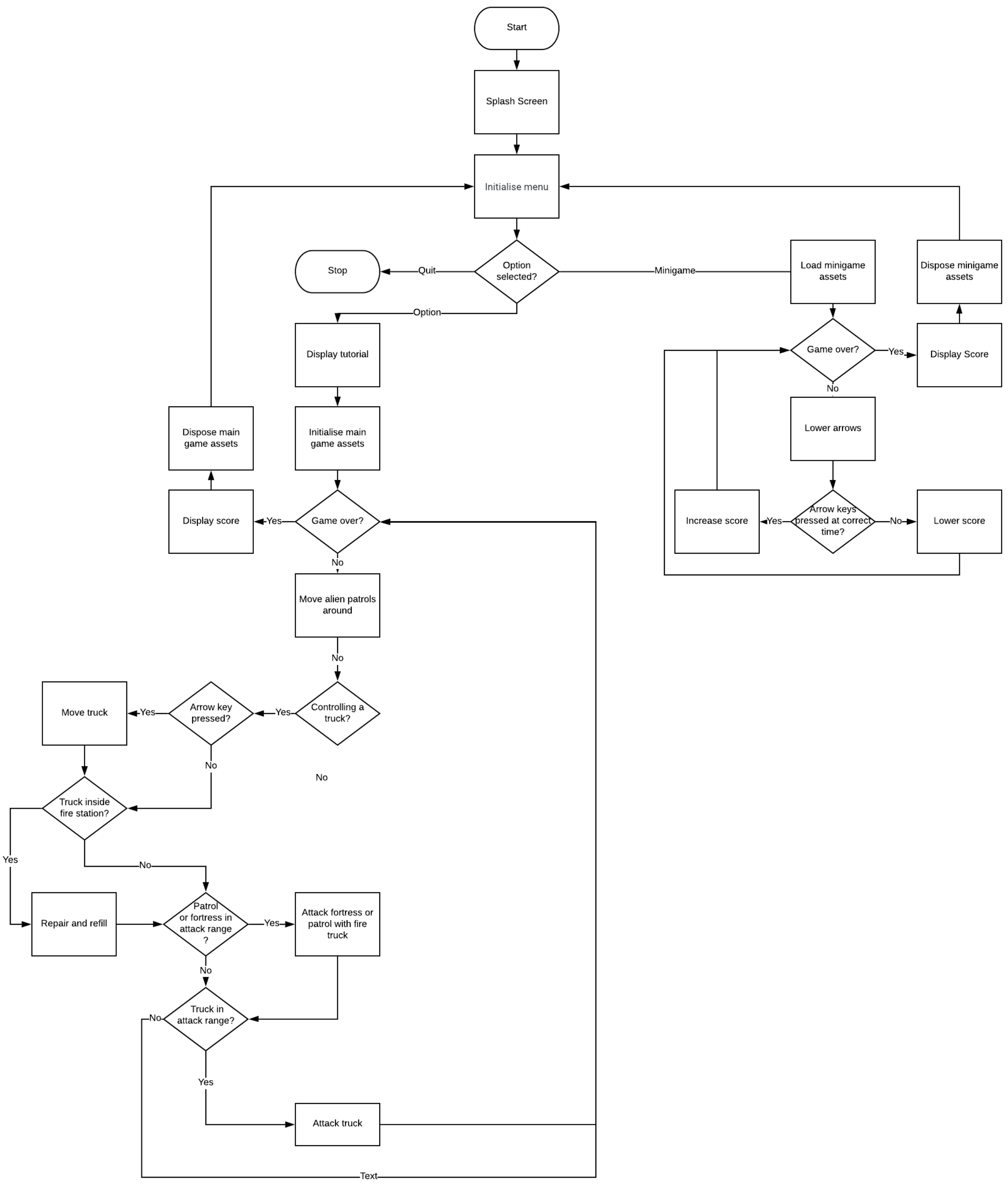
Our justification for planning and designing the system architecture stems from the aforementioned system requirements. Note that constraint CR_JAVA initially limits our device options to either Android or desktop systems (forgoing non-native libraries), and CR_DESKTOP & CR_CONTROLS determine it must be on PC (or Mac/Linux). We have decided to use the open source library libGDX. This game development library not only allows us to ensure our desktop game is made in Java, but its ease of use and prebuilt functionality will make user requirements such as UR_ENJOY and UR_AESTHETICS implementable easily due to its streamlined UI tools.

We have supplied both a flowchart describing the user flow, and a UML diagram for an abstract description of classes below to help us have a better understanding of the game and will give us an idea of how to program our different classes.

Flowchart

To have a better understanding of the game, we decided to create a flowchart simple and easy to track. Our flowchart gives us a general idea of the game, shows how different parts of the program will connect together, and how users interact with the game. There are some aspects of our game that we have omitted from our flowchart, like how often patrols should spawn or what happens if the station is destroyed. The flowchart is an abstract representation of the base game logic and we didn't need to go further in detail at this early stage. A general read through of the flowchart is enough to understand how the game works; parts of the flowchart relate to our requirements, such as how the minigame flow meets our UR_MINIGAME requirement and being able to understand the flowchart in and of itself meets our NFR_USABILITY requirement. Writing this flowchart first made it easier to write our UML class diagram, as understanding how the game needs to work helped us determine what classes we need to create for the game to function as described.

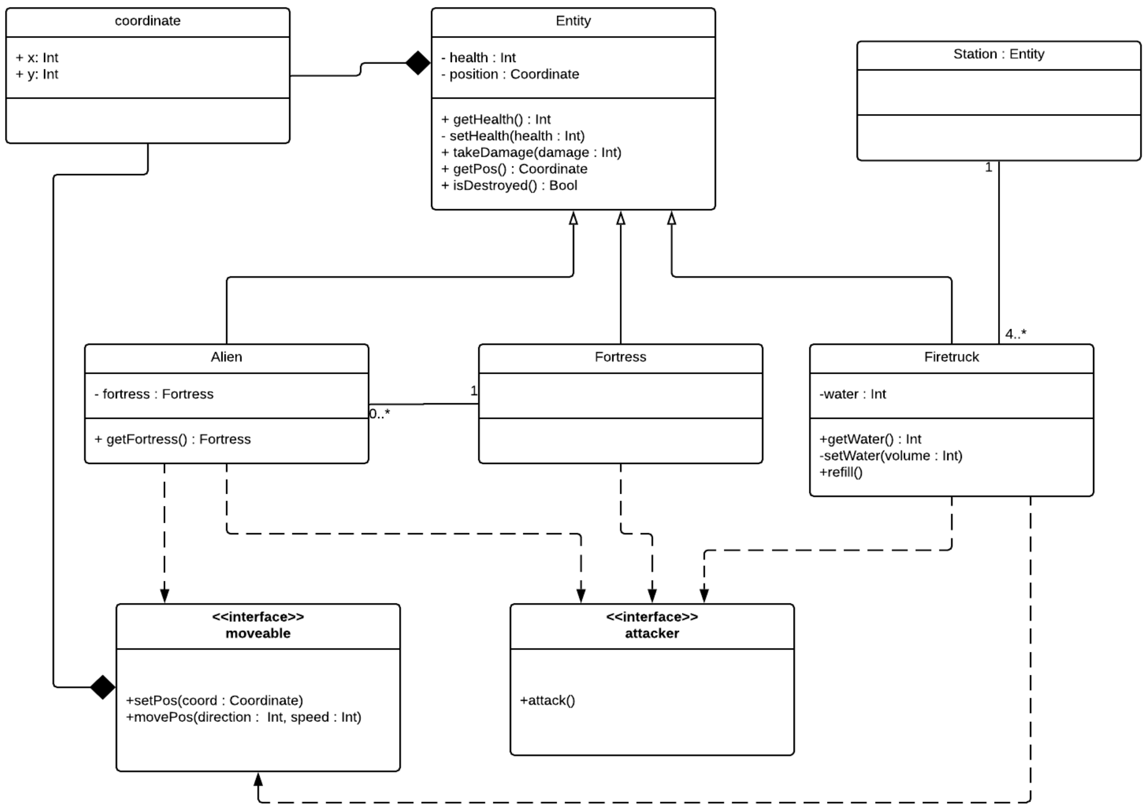
We used Lucidchart to make our diagram as it is easy to work with it and it is really good for group projects because we can share a project link and work on the chart simultaneously. While there were alternatives available (draw.io/StarUML), all we needed was a free piece of software that allowed us to create a very simple flowchart, without needing any advanced features. We were lucky in this aspect because Lucidchart also had collaboration tools we used to our advantage. It was a good choice to go with the same tool that Google, Adobe, and Netflix also use.


Start
Splash Screen
Initialise menu
Option selected?
Quit
Stop
Minigame
Option
Display tutorial
Initialise main game assets
Game over?
Yes
Display score
Dispose main game assets
No
Move alien patrols around
No
Controlling a truck?
Yes
Arrow key pressed?
Yes
Move truck
No
No
Truck inside fire station?
Yes
No
Repair and refill
Patrol or fortress in attack range ?
Yes
Attack fortress or patrol with fire truck
No
Truck in attack range?
No
Yes
Attack truck
Text
Load minigame assets
Game over?
Yes
Display Score
Dispose minigame assets
No
Lower arrows
Arrow keys pressed at correct time?
Yes
Increase score
No
Lower score

UML Class Diagram

The UML class diagram is an abstract representation of how our game objects relate and interact with each other. We created our classes by systematically going through the brief and our requirements, and deciding what unique classes we'd have to create to meet our needs. The first step was to figure out the names of every unique thing that might need to be created, such as the fortress, aliens, and the fire station. For some of these objects, we grouped them into what they might share in common given what was described in the brief, such as how the station, fortress, aliens, and firetrucks all have health and a position, but the station does not need to attack like the other classes do, and the fortress also does not need to move around. Once we figured these out, we created methods and attributes which described what we could do to each of these classes, and what information describes them. Finally, we added some associations, so we knew how exactly the classes related to each other.

We chose to go with Lucidchart again for the UML class diagram. Given we had already completed the flowchart, we were experienced in using the software. It is also fortunate that Lucidchart supported the symbols needed for creating UML class diagrams, including object composition and interfaces which we have included in the class design. It made sense to use what we know instead of perhaps going for another solution such as Platinuml, as we'd have to have learned how to use an entirely different type of software to complete a small, simple task.



b) **Systematic Justification**

In the below table, we have described the classes present in the class diagram. We have explained why we have created our classes in this way, what each class includes, and how they satisfy user requirements.

Entity	Entity is the base class that all our game objects inherit from. Because many of our classes share the same attributes and methods (like health, getPos()), it makes more sense to define these in a parent class and then encapsulate them there instead of having to redefine them in each child class. Because every entity has a health value, this helps meet our FR_ENEMY_HEALTH and UR_HEALTH requirements, which require the health status of our entities to our user. Because we have an isDestroyed() method, we can check if something is dead but still keep it alive in our game world (i.e we do not delete the entity). So if firetrucks or the fire station are destroyed, we can still leave their sprites in game but render them unusable.
Coordinate	Coordinate is a struct that contains and x and y value relating to the position of an object. Using object composition, we can define this class instead of using a size 2 array to represent positions, and use this throughout our game. This is also useful in making sure that every class that implements the usable interface will always accept a coordinate in its setPos() method, instead of possibly accepting the wrong data types.
Fortress	This class is a subclass of Entity. While it doesn't contain any unique methods or attributes, it is an entity that must be able to attack, so having it be its own unique class that inherits from the attacker interface means that when we implement the attack() method, we can have it unique to each fortress (instead of having a generic attack that all fortresses share) which is needed to meet FR_VARIATION. The multiplicity associated with Fortress and Alien represents that each fortress can spawn one or more alien patrols, which are "attached" to that parent fortress.
Firetruck	Firetruck is a subclass of Entity. It inherits from the attacker interface, but as it can move it also inherits from the movable interface. Uniquely, it has a water attribute that represents how much water the truck has, and methods to represent accessing its water level, and refilling it.
Alien	Alien is a subclass of Entity, which implements the movable and attacker interfaces. Instances of Alien are only spawned by fortresses, so there is a method and attribute available to get the Fortress object that created the Alien. This is good for if we want to include any additional functionality that we haven't deemed necessary in our requirements, such as aliens of a certain fortress homing in on any nearby fire trucks if they are attacking it.
Attacker	Multiple classes require different ways to attack. We could have created a parent class for them to inherit from, and making this class abstract would have ensured you couldn't have instantiated an object of this bass class. However, because we only need to define an attack() method, and no attributes, making this class an interface was better suited for our needs. Each class that implements attacker will have its own custom definition of the

	attack() method. Fortress in particular will require many different unique attacks at the implementation level in order to meet our FR_VARIATION requirement.
Moveable	Similar to Attacker, it was better suited to use an interface instead of an abstract class. When we implement the +movePos() method in the Alien class, movement may be more random or perhaps alien patrols will slightly home in onto the nearest fire truck. The implementation in Firetruck however will instead need us to ensure that when an arrow key is pressed (we use arrow keys because FR_CONTROLS defines we must use keyboard input), the fire truck that moves is only the fire truck that we are currently controlling.
Station	Station isn’t a defined class, but is instead an instance of entity. It doesn’t contain any unique attributes or methods, but fire trucks can only refill if their position is at the same position as the station. The multiplicity supplied represents that for one station, there are four or more fire trucks that may refill at that location (as per our project brief, we need a minimum of four fire trucks).