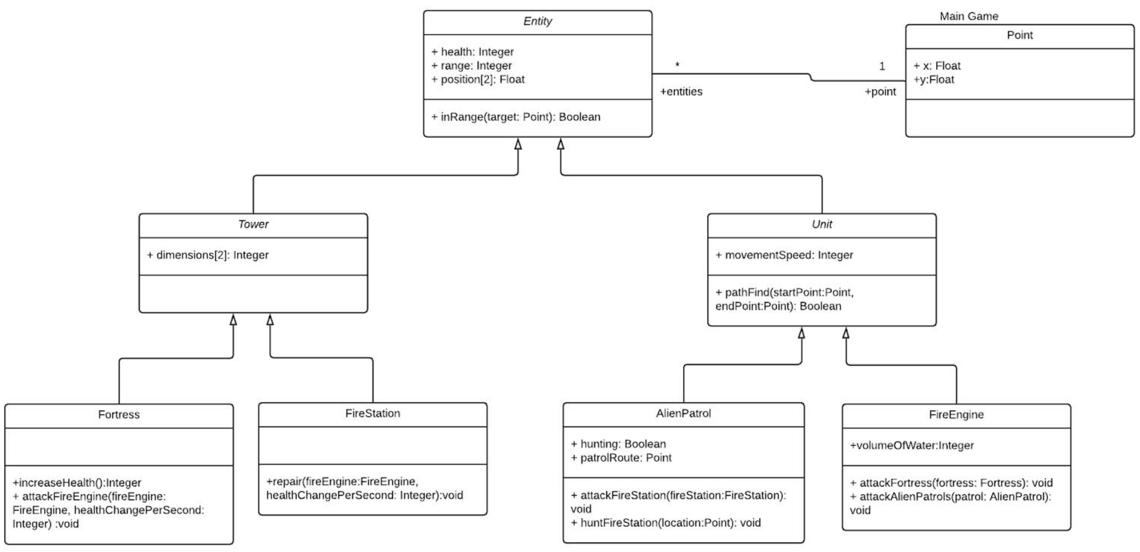
Module	SEPR
Year	2019/20
Assessment	1
Team	Dalai Java
Members	Jack Kershaw, Max Lloyd, James Hau, Yuqing Gong, William Marr, Peter Clark, Thomas Richardson.
Deliverable	Architecture

Architecture

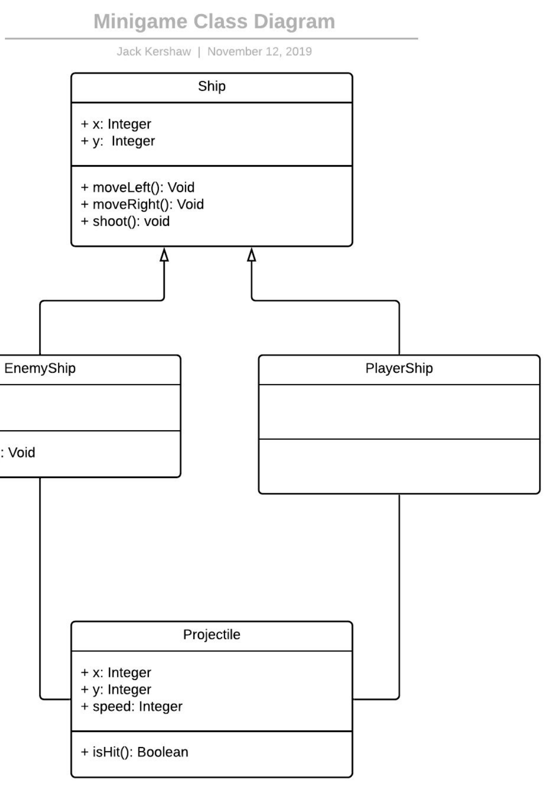
To ensure that we have a clear idea of the system we are creating and to bridge the gap between the gathered requirements and implementations, we have decided to create a prescriptive model of the system. This is an abstract representation of our final product, including both the main game and the embedded minigame, and will define how it should be implemented. We have created our basic architecture using a collaborative UML tool called LucidChart

.

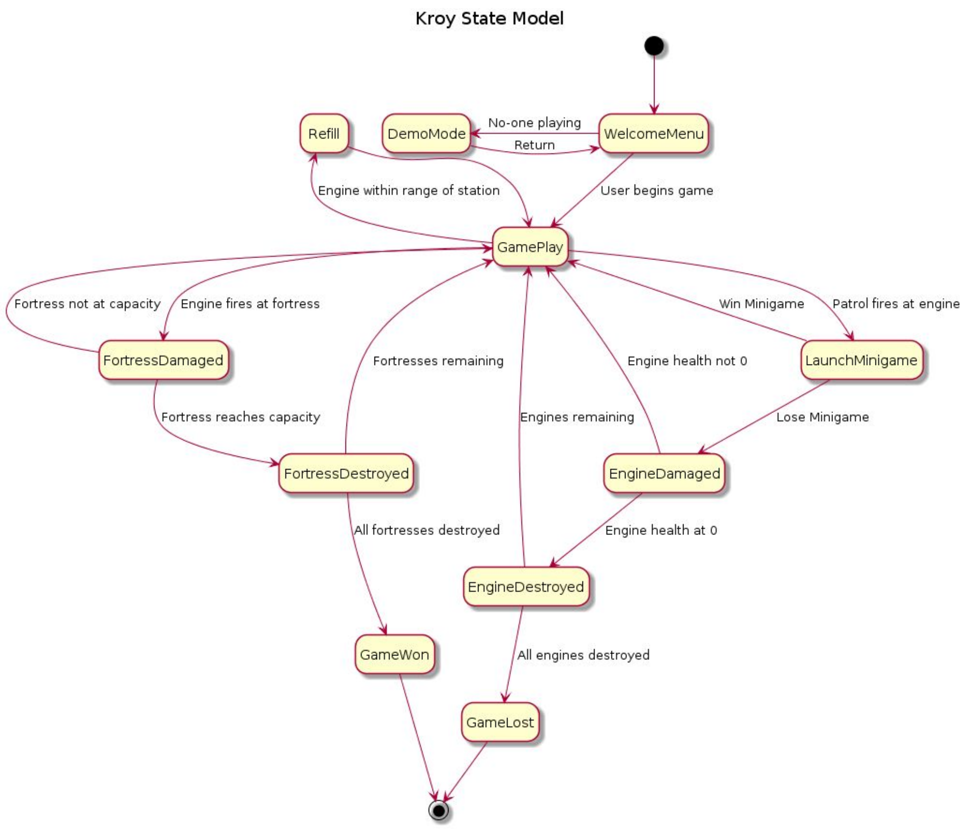
Main game UML

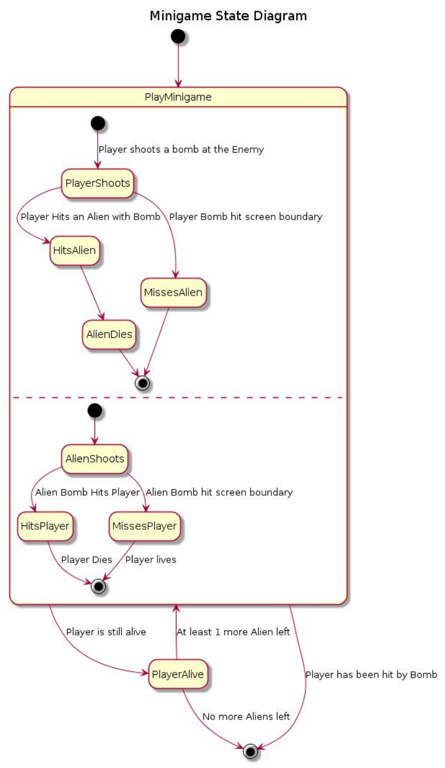

Main Game
Entity
+ health: Integer
+ range: Integer
+ position[2]: Float
+ inRange(target: Point): Boolean
*
+entities
1
+point
Point
+ x: Float
+y:Float
Tower
+ dimensions[2]: Integer
Unit
+ movementSpeed: Integer
+ pathFind(startPoint:Point, endPoint:Point): Boolean
Fortress
+increaseHealth():Integer
+ attackFireEngine(fireEngine: FireEngine, healthChangePerSecond: Integer) :void
FireStation
+repair(fireEngine:FireEngine, healthChangePerSecond: Integer):void
AlienPatrol
+ hunting: Boolean
+ patrolRoute: Point
+ attackFireStation(fireStation:FireStation): void
+ huntFireStation(location:Point): void
FireEngine
+volumeOfWater:Integer
+ attackFortress(fortress: Fortress): void
+ attackAlienPatrols(patrol: AlienPatrol): void

Minigame UML



We have also created a simple state diagram for both our main game and our mini-game using PlantUML to outline how the game would be played in general terms. This allows a visual representation of the logic that will need to be implemented in order to get a correctly functioning system. This diagram can be used during the implementation stage of the development cycle to check and confirm the flow of events leaving to certain parts of the game make sense and are fulfilling each of our requirements that we have laid out. This will make it far easier to tell if the development of the game is heading in the correct direction further focusing our development efforts in the correct direction. The arrows represent the possible valid changes of state that can be followed within the game. Any invalid moves are ignored as are not needed to understand the game flow. Within the minigame state diagram we used concurrency to show how both the aliens and the player can both be firing at each other at the same time and how also only one projectile from each ship can be shot at any one time.





Justification

In our class diagram, we have created a superclass called Entity. Fortresses, Fire Engines, Alien Patrols and Fire Stations are all subclasses of this class as all of them share some common features; each of them has some form of health (FR_ENGINE_SPEC_DAMAGE, FR_FORTRESS_SPEC_WATER; for the fortresses the ‘health’ variable will store the volume of water required to flood the fortress); each has a range (for the fortresses, fire stations and alien patrols this is the range at which they can attack (FR_ENGINE_SPEC_RANGE, FR_FORTRESS_SPEC_RANGE, for the fire station this is the range at which a fire engine can be refilled and repaired); and each has a position on the screen. We have related the Entity class to the ‘Point’ class which stores the x- and y-coordinates of the given entity. Each entity only has one point, however one point can have multiple entities as they may be crossing paths; therefore, the relationship is a many-to-one relationship. For each method that deals damage or heals a unit a parameter called healthChangePerSecond is called; for the Fortresses, Fire Engines and Patrols, this will be the damage delivered to the enemy (as required by FR_FORTRESS_SPEC_DAMAGE and FR_ENGINE_SPEC_DAMAGE), whilst for the Fire Stations this will be the health restored to the Fire Engines.

One of the subclasses of the class Entity is the class Tower. This class represents a functional structure that does not move, and has variable dimensions to store the size of each tower as each tower has a unique specification in terms of size. The Tower class has two subclasses; Fortress and Fire Station. The Fortresses are the entities which are ‘controlled’ by the aliens and inherit all the attributes from Method and Tower, however they also have an increaseHealth() method, which acts as a “leveling up” function within the game. This means that as the game progresses it also gets progressively harder to win. This is described within the product brief and solves the requirement, FR_FORTRESS_SPEC_HEALTH. The Fire Station also inherits all the previous methods, however it also has a repair method, which will take a Fire Engine and repair it using the previously defined healthChangePerSecond attribute. We have included this method as the customer requires the Fire Stations to be able to refill and repair Fire Engines (FR_ENGINE_MAINTENANCE). The Tower class exists as the Fortress and Fire Station both have an attribute in common which they can inherit from tower.

Another subclass of the class Entity is the class Unit. This inherits all the attributes of Entity, however also has a movementSpeed attribute defining the speed at which each unit can move. This is because the customer requires each Fire Engine to have a unique specification in terms of its speed (FR_ENGINE_SPEC_SPEED). Unit has two subclasses; alienPatrol and fireEngine. Alien Patrol has two new attributes; a Boolean called hunting and a Point called patrolRoute. This is required as the Alien Patrols will not be controlled by a player and so need some form of path finding. This can either be in the form of an algorithm or simply a predetermined route that the aliens are hard coded into. This solves the requirement FR_ET_FIND_OUT_ROUTE. The Fire engine also has a new attribute storing the volume of water it can hold. We have included this as the customer requires each Fire Engine to have a unique specification in terms of the amount of water it can hold (FR_ENGINE_SPEC_WATER).

An emphasis of ours was too minimise the number of classes that we will be having to implement to keep the game as simple as possible while still ensuring that the program is easy to maintain and understand.

The minigame that we will be implementing will be based off the game space invaders where the players ship is the fire engine shooting water and the aliens Ship are the ETs. Both forms of ship have the ability to move left or right but only the alien ships can move both up and down. For this reason only the methods moveLeft() and moveRight can be inherited from a Ship super class. Each ship, both player and enemy require a position of where they are on the board stored as two integer parameters x and y. This allows them to be rendered correctly on the display for the player to correctly view them as well as detect whether or not that the fired bombs have hit a target or a screen boundary.

State Diagram Justification

Our state diagram begins with a Welcome Menu, which will appear for a certain amount of time before switching to a Demo Mode, in which the game will run itself, when the player is idle, in order to satisfy the requirement UR_DEMO_MODE. When the user returns, the Welcome Menu will once again be shown. Once the user chooses to play the game, game play will then begin.

The state moves from the standard GamePlay state to the FortressDamaged state when the Fire Engine attacks the fortress. This satisfies the requirement FR_CONTROL_ENGINE_ATTACK by allowing the engine to attack the fortresses when within shooting range. If the fortress has reached the maximum water capacity it can hold then that fortress is destroyed (satisfying FR_FORTRESS_DESTROYED); and if all fortresses have been destroyed, then the game has been won (satisfying FR_GAME_WIN) and the user will be informed of this (satisfying FR_NOTICE_GAME_OVER). Else, the game returns to normal gameplay as the user must do more in order to win the game.

When an Alien Patrol fires at the Fire Engine, the state moves to the Mini Game state, where the embedded mini-game will be played (satisfying FR_MINI_GAME_BEGIN). If this game is won, the Engine will avoid any damage dealt by the patrol. However, if it is lost, the engine will be damaged (satisfying FR_MINI_GAME_LOSE). As with the fortresses, the state is then changed to EngineDestroyed if the engine's health is at 0 (satisfying FR_ENGINE_DESTROYED) and if all engines have been destroyed, the user has lost and is notified of this (satisfying both FR_GAME LOSE and FR_NOTICE_GAME_OVER), else the state returns to GamePlay as there are still engines the user can use to attack fortresses and potentially win the game.

The state can also be changed to the Refill state when the Fire Engine is in range of the station, allowing the Fire Engine to be refilled and repaired. This is an important feature of the game as stated in FR_ENGINE_MAINTENANCE. However, once the Fire Station has been destroyed the Refill state should no longer be reachable.

In terms of the Mini-game state diagram, there are two separate potential actions that could occur in the game; the player firing at the alien, or the alien firing at the player. These are being shown as though a concurrent state, as either or both could happen at any one time, meaning that it is impossible to simply create the events as a sequence. If the player fires at the alien then the state moves from to HitsAlien, and then to AlienDies. On the other hand, if the player's bomb hits the screen boundary, then the state is moved to MissesAlien. A similar set of states is used for when the player attacks the alien. After each of these events, we check to see whether the player is alive and if there are any aliens left; if either of these conditions are not met, the minigame ends. By creating the minigame in this way it makes it clear what the objectives of the game are, and allows us to successfully embed a minigame into our game.