UNIVERSITY OF YORK

DEPARTMENT OF COMPUTER SCIENCE

ENG 1 Team Assessment
Group 18

ARCHITECTURE

Group Members:

ROB ANDERSON

MORGAN ELKINS

ZIAD ISMAILI ALAOUI

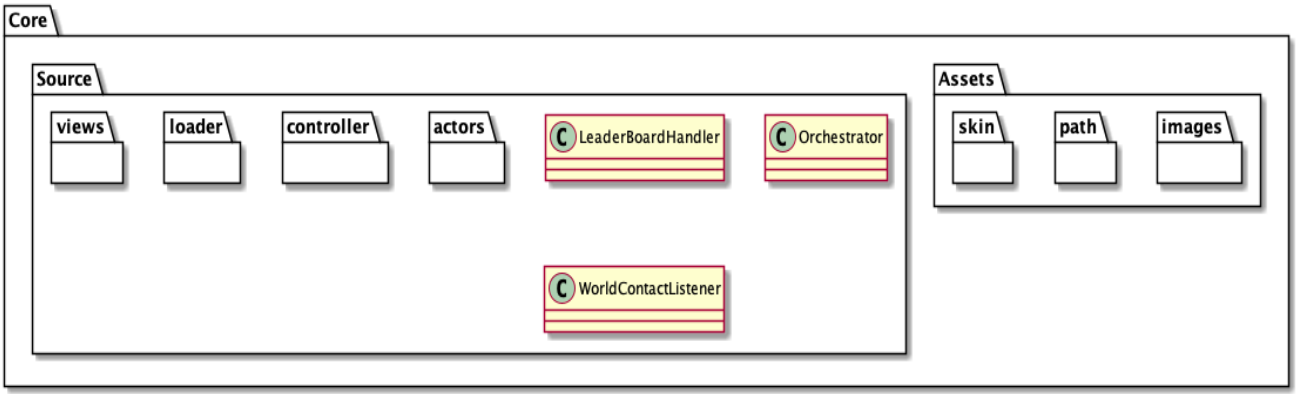
SAM KNIGHT

SCOTT REDSHAW

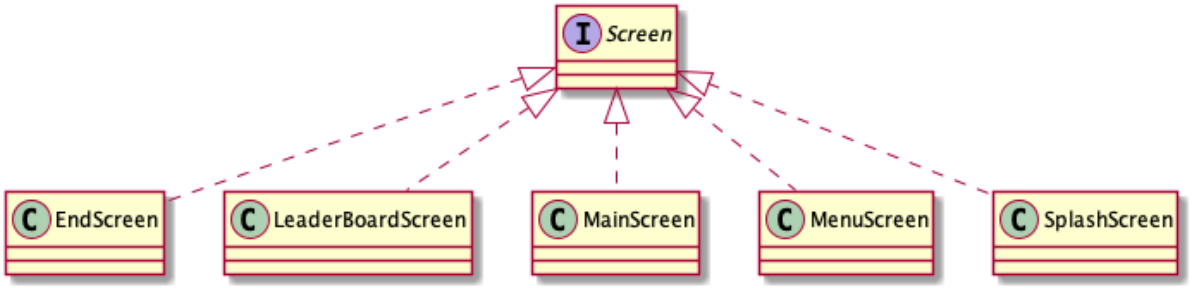
GEORGE TONNERO

3.a - Abstract Architecture Representation

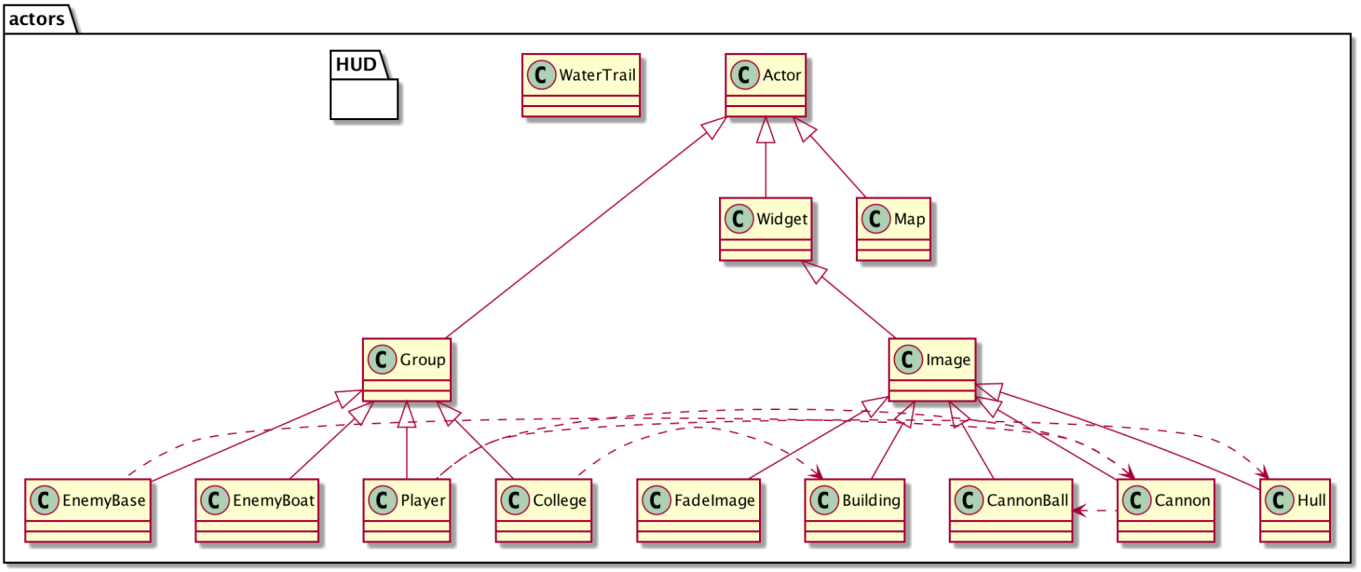
Core:



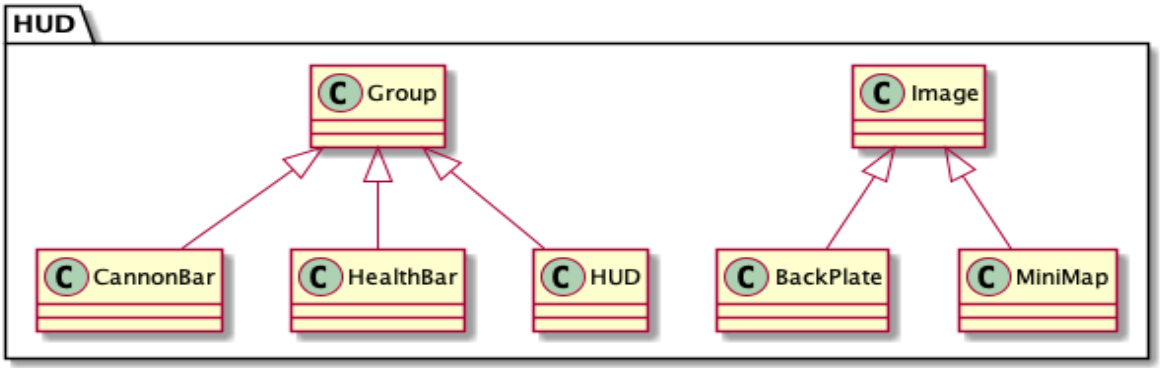
Views inheritance structure:



Actors inheritance/dependency structure:

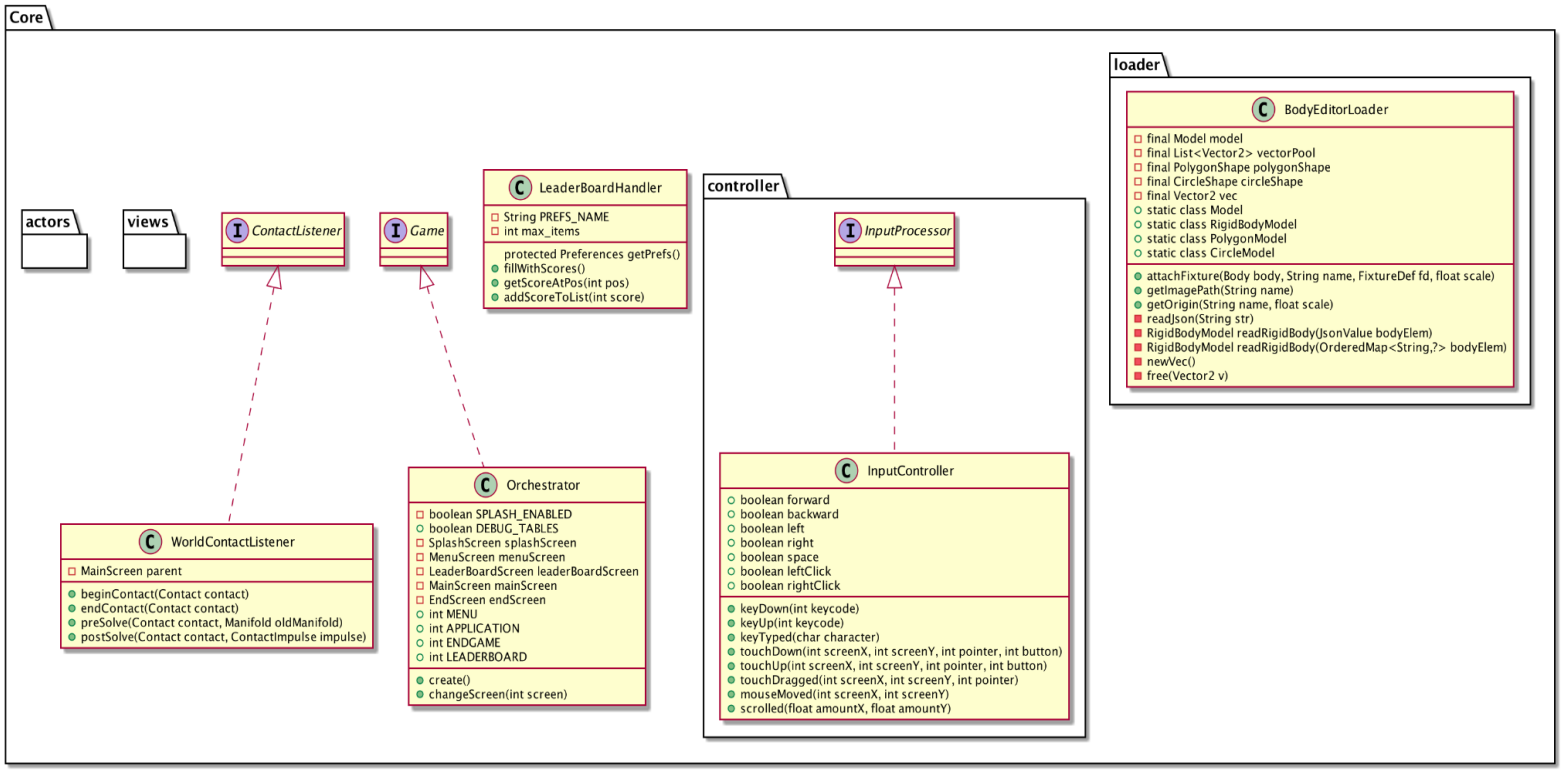


Hud inheritance structure (Group and Image inheritance shown above):

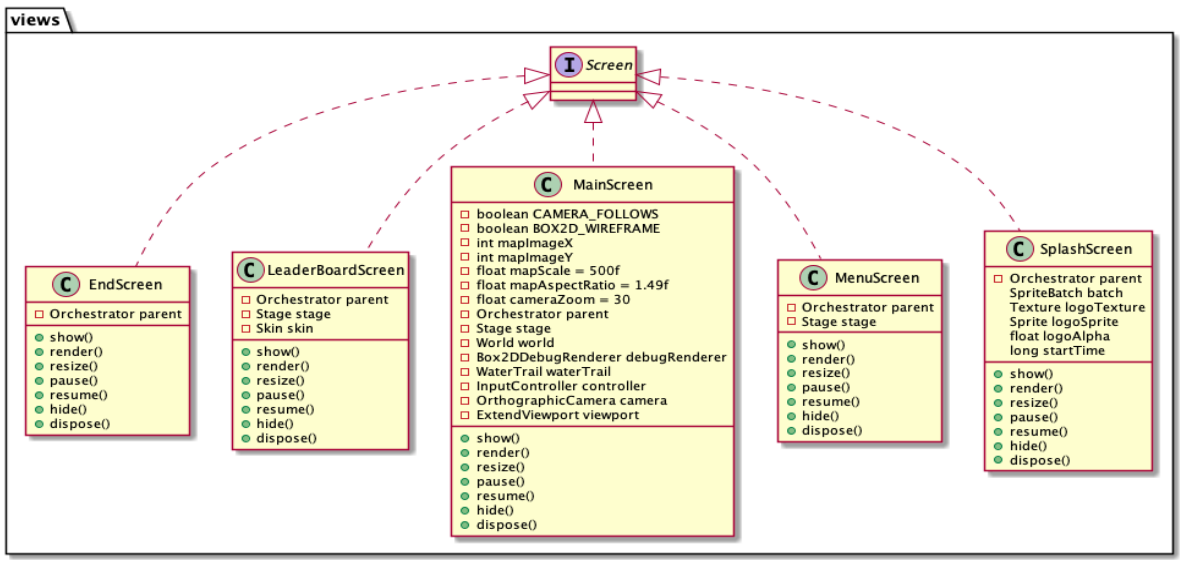


3.a - Concrete Architecture Representation

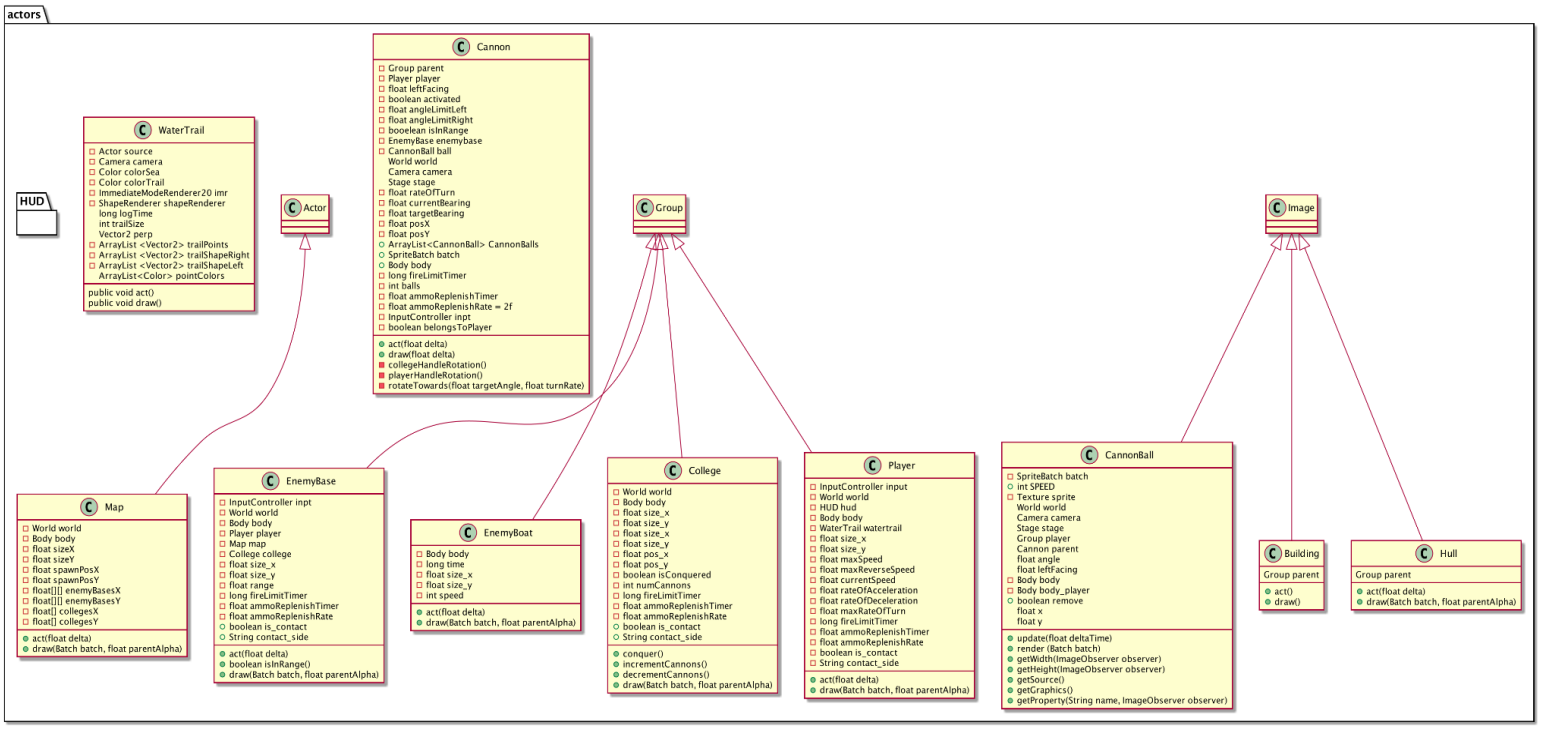
Core:

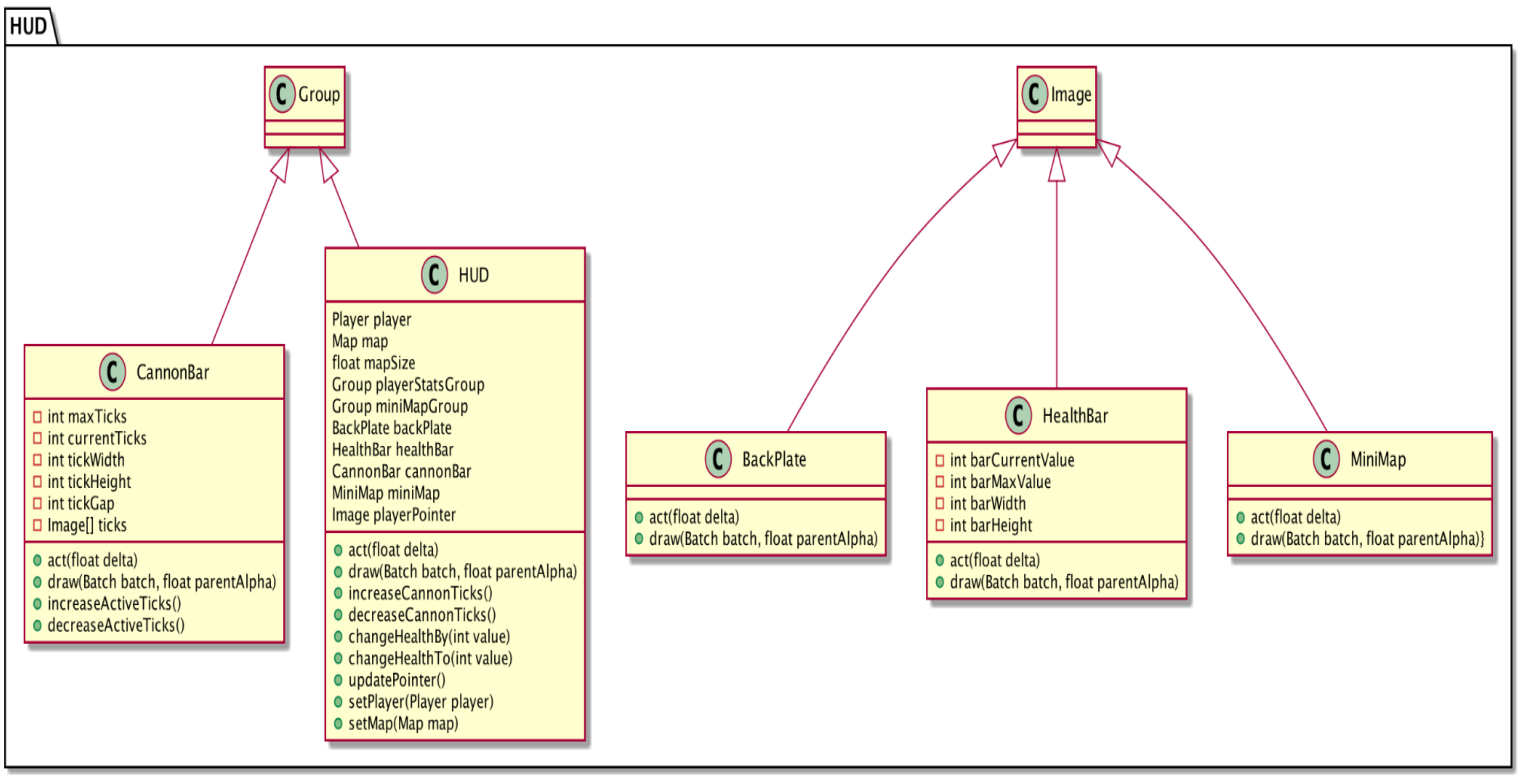


Views:



Actors:





To create the UML architecture representations PlantUML was used. This was chosen because it has all the necessary tools to create our required diagrams, such as class and component, as well as the code making it easy to implement and update the diagrams which also means that it's easier to share. There is also a

PlantUML plugin within our IDE of choice, IntelliJ which means we could write the diagrams where our code is, which makes it more convenient compared to other UML.

3.b - Abstract Architecture Justification

The architecture of our core package is 5 packages deep. It has an open architecture in that classes depend on others more than one package away from themselves. For example, MainScreen in the views class makes multiple calls to classes in the actors package, and actors make calls to items in the images package.

We chose the object oriented software pattern. The reason for choosing this over an entity-component-system was twofold; the members of our team were much more familiar with this pattern and we were working under time constraints; the requirements for the product were relatively simple, and so OOP did not suffer too much from complexity.

Nearly all of the game code is contained within the source package. Here we used a Model-view-controller architecture. The orchestrator is used by the screens in the views package to switch between themselves, and they all inherit from the Screen class. MainScreen is used to instantiate actor classes, and most of the game model/logic is stored within the actors themselves.

3.b - Concrete Architecture Justification

To start the game an instance of Orchestrator is created which inherits LibGdx's Game class. This has a variable for each of the screens in the views package and an integer for each. These all inherit the LibGdx Screen Class. On creation Orchestrator runs its create() function. This creates a sprite batch and a sprite of the game's logo. It draws this sprite to the screen using the render function. It decreases the alpha of the image as time elapsed increases, and then calls the orchestrator changeScreen function to change to a MenuScreen. This sequence helps fulfill the 'attractive graphics' requirement 011.

The menu screen creates a stage and a table as an actor in the show() method. The libgdx Stage Class is a 2D scene graph containing hierarchies of actors which handles the viewport and input events. The actors are objects added to the stage to allow for a list of progression actions to be applied, like movement and drawing, within the act() method. This allows game objects to be loaded in and rendered in a logical hierarchical fashion. It adds three buttons to the table - Play, Leaderboard and Quit - and adds changeListeners that change to a MainScreen, LeaderBoardScreen and EndScreen respectively on input. The style of the buttons is determined by a skin which is stored in the skin file, which again helps achieve requirement 011. It draws the stage to the screen with the render function.

If the leader board button is pressed then the changeListener calls the orchestrator function to change screen to the LeaderBoard screen. This fulfills the requirement FR_SCOREBOARD of having a leaderboard.

When the MainScreen is set it first creates an instance of the InputController and sets it as the input processor. It creates a new Box 2D world, and sets the contact listener of the world to the world contact listener. Box2D is a 2D physics engine and the world manages the state of entities**.** It creates two stages - one for the HUD and another for the game. It creates a new HUD object and new FadeImage object and adds them to the Hud stage. It creates a Map object, a player object and a water trail object and adds them to the game stage. It then creates a new college object for six colleges and a number of enemy bases for each college - one for the first and two for the rest. It adds these to the game stage. In the render function it runs the game logic for each component and then draws them to the screen.

The HUD is a display to show the user vital information about the boat and the user. The HUD contains five different classes: BackPlate, CannonBar, HealthBar, MiniMap and HUD. The BackPlate class that creates a backplate using a png to be able to add all the different components in a logical and organised way. The cannon bar is a bar that shows and calculates how many cannonballs to fire at that point in time. It uses time ticks to regain cannonballs after a certain amount of time and limitations on if the cannonballs can be fired even if there are no cannonballs left. The health bar is similar to the cannon bar as it keeps track of hits from enemy cannons and takes away the damage done by the enemy boats. Once the health is zero it creates a game over screen. The minimap is a png file with a pointer layered on top to indicate whereabouts the user is in relation to the map. Finally the HUD class gives the functionality to all the other classes. This will include the updating of the pointer, the increase and decrease of cannonballs, the change of health, overcoming changes in window size and instruction pop ups.

This fulfills requirement UR_MAP of having a minimap to see the lake and requirement NFR_MINIMAP of informing the user where they are on the minimap. It also fulfills requirement FR_SCORE of displaying the points on the screen, requirement NFR_MOVEMENT_TUTORIAL of showing the user the controls and contributes to the requirement of having attractive graphics.

The map class takes the world and size as parameters. It creates a static Box2D body. It loads the map using the loader class. It also stores the locations of the enemy bases and the colleges in arrays as ratios of the screen. It has getter functions which return the ratio multiplied by the screen size. This fulfills the requirement UR_MAP of having a map which contains map and ports.

The player is a group which means that it is a parent actor of multiple actors. It is composed of a Hull object and two Cannon objects. The hull simply creates a new texture of the ship's hull for the player. The cannon object handles the creation of the texture as well as its rotation, whether it can fire a cannonball and handling the created cannonballs. Cannonball objects are placed into an array within the cannon object to handle removal, and cannonballs create a new sprite to represent itself and handle its own movement. The player object handles its own movement input as well as its health, collisions and rendering. This fulfills requirement UR_MOVEMENT of allowing the user to move the ship, requirement FR_INPUT of the system taking input and adds to requirement FR_SCORE of allowing the user to gain points by defeating bases.

The college class creates a static Box2D body and adds an image texture which is stored in the assets folder. It has a health of 50 and has to be hit 5 times to be destroyed. The mechanic for this is once a collision has occurred (which is detected within the WorldContactListener) between it and a cannonball, the user data of the body is updated to represent that it has been hit. This is then read by the parent body to decrease its health and once it reaches 0 it puts itself into a destroyed state. The enemy base class is another group that creates a Box2D body. Currently the only actor it holds is a cannon, which has a parameter for determining whether it is a base or player's cannon. However, it was made a group in case there was a need to extend the base properties in future implementations. These have a health of 20 and have to be hit twice to be destroyed. There is also an enemy ship class, but due to the requirements this has no cannon functionality and just wanders around.

The fog in the game is a png that sits above the map. The map is a jpeg to reduce file size but clouds are png to allow transparency. This was a cheap and fast way to give the game a complex aesthetic and helped again fulfil the 'attractive graphics' requirement. The water trail class also adds to the fulfillment of this requirement by creating a water trail behind the player. This makes the game look more realistic and interactive.