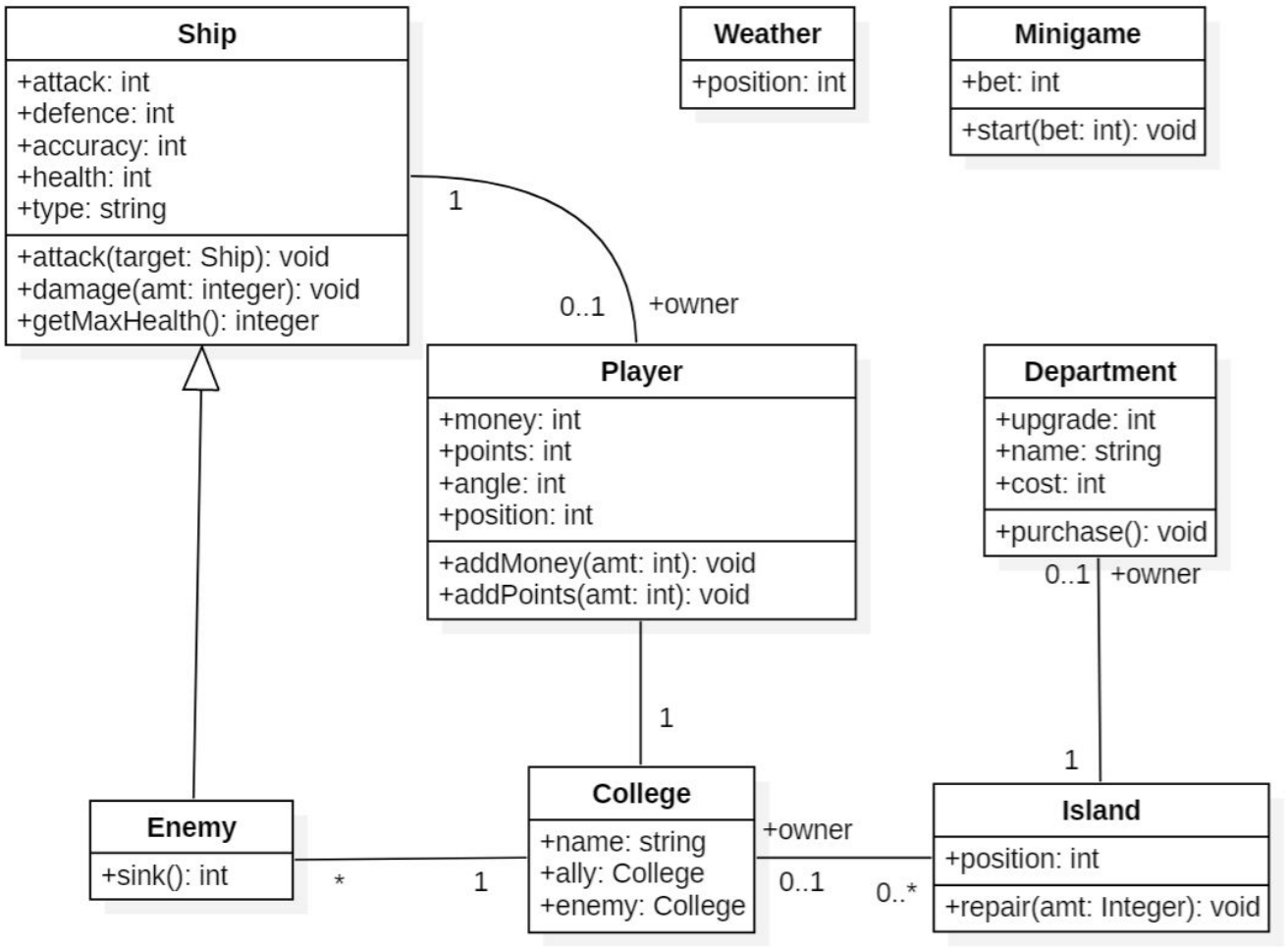
Architecture

Abstract Representation

We used StarUML for designing the UML class structure model as it adheres strictly to UML notation, is available for free and generates professional-looking UML diagrams. As for notation, we have mostly adhered to the standard UML notation with the possible minor exception of the College class. Because this class links to two other instances of itself we decided it would look messy to have a two-way link from a class to itself and so we instead decided to use it as a de-facto data type for two attributes ('ally' and 'enemy'), making the diagram much neater.

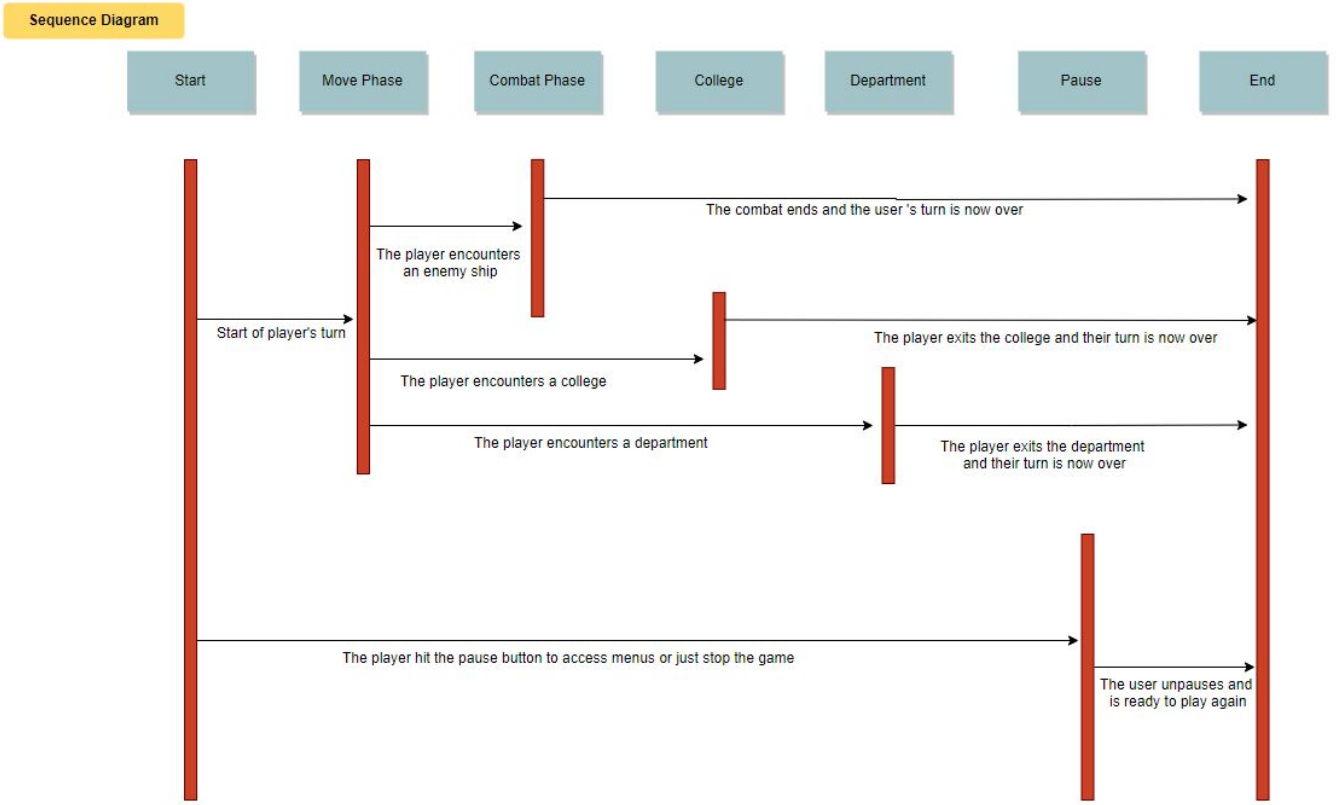


Class	Brief Description
Player	This class creates the player object that holds many key variables in the game such as money and points as well as the player's position on the map. It also has an associated ship object and college. Functions: • The *addMoney* function takes an integer and alters the player's money by that amount. It takes both positive and negative integers, allowing it to also be used to subtract from the players money in cases such as the player making a purchase from a shop or losing at the minigame. • The *addPoints* function will work similarly to the addMoney function, except it will alter the player's points attribute instead of their money.

Ship	This class allows multiple ships to be instantiated, each with their own unique stats and is the superclass of Enemy. The object stores information about the ship such as its stats (attack, defence and accuracy) as well as its current health and type, the latter of which determines its default stats and graphic representation in the game. Functions: ● The *attack* function allows one ship to attack another. It will calculate whether the attack hits or misses and how much damage should be dealt based on the first ship's stats and then call the second ship's damage function to deal the damage. ● The *damage* function will handle the dealing of damage to a ship, adjusting its health attribute accordingly. It will also detect whether the ship has sunk and if needed call the sink function. ● The *getMaxHealth* function calculates the ships maximum health based on its stats. This will be used for comparison to the health attribute and for repairing the ship back to full health.
Enemy	This is a subclass of ship and inherits it's methods. It can also take a College object as a modifier to the class to allow enemy ships to be from different colleges. Functions: ● The *sink* function will be used when the enemy ships health falls to zero or less. It will calculate how much money the player should receive as a reward and return it as an integer.
Island	This class creates the object Island, and holds the position (coordinates) of the island on the map. Functions: ● The *repair* function repairs the players ship by the amount specified by the parameter 'amt'.
Department	This class implements the Department instances of Islands and as such is associated with Island by a 1-1 relationship. It also has additional attributes such as 'upgrade' (the type of upgrade the department sells) and 'cost' (how much the upgrade costs to purchase) to facilitate the player buying upgrades for their ship. Functions: ● The *purchase* function handles the purchasing of the department's upgrade and its application to the players ship.
College	This class implements the college factions in the game. It is associated with any number of islands to allow for the capturing of other colleges and also stores the colleges ally and rival (both of type College).
Minigame	This class implements the minigame and as such has an attribute containing the amount the player has bet, to be used in calculating winnings. Functions: ● The *start* function is used to start an instance of the minigame. It takes the amount the player has bet in the parameter 'bet'.

Weather	This class implements the weather system of the game. The attribute 'position' will hold the current position of that instance of the weather system on the map.

Sequence Diagram



A brief description of each player's action options when they move one tile.

- Start - Any action that occurs when the player is stationary on a square
- Move Phase - Any action that concerns the moving of the player ship to a new grid position
- Combat Phase - Any action that concerns the user encountering an enemy ship
- College - Any action that concerns the user encountering a college
    - Minigame - Any action that concerns the minigame
- Department - Any action that concerns the user encountering a department
    - Upgrades - Any action performed at a department where the user upgrades their ship or views possibilities to
- Pause - Any action that concerns the user pausing the game
    - Minigame
- End - Any action that will lead to the ending of a player's movement.

Justification

Class	Justification
Player	This class allows us to meet the requirement **F6** (point system), as the method 'addPoints' allows the points attribute in the Player instance to be modified whenever necessary. This classes 'money' attribute also allow requirements **F7** and **F11** to be met via the 'addMoney' method.
Ship	This class allows us to meet requirement **F1** (ships as transportation) as the object has methods ('setMoving' and 'turn') that allow ship movement on the map. This also allows us to meet the functional requirement **F3** (sailing mode).
Enemy	This class allows the generation of enemies allowing us to meet functional requirement **F4** (combat mode).
Island	Island class along with the associated College class allows us to meet requirement **F8** (capturable colleges), as the associated College can be changed to that of the players. Island also allows the method 'repair' to be implemented, meeting requirement **F11.2** (plunder can be spent on healing)
Department	Department class allows requirement **F11.1** (upgrade ships) to be met as there is a 'purchase' method. The class also allows us to meet the requirement **F5** (game must have at least three departments) by its inclusion.
College	As shown in Island class, College allows us to meet requirement **F8**, of capturable colleges through association. The class also allows us to meet the requirement **F5** (game must have at least five colleges) by its inclusion.
Minigame	Minigame class allows us to meet the requirement **F12** (must be a minigame). The attribute bet and method "start", allow **F12.1** (betting within minigame), to be achieved.
Weather	Weather class allows us to meet requirement **F6.3**, (points will be awarded through survival from weather).

Sequence Diagram Justification

We chose to use a sequence diagram to illustrate how a player's turn would work. We chose a sequence diagram over other designs available as it shows clearly and concisely each step possible when a player moves 1 square. This is not over complicated (as many other designs can be) which made it a lot easier to plan out classes for our UML diagrams. The simplicity of this diagram means that when we eventually make many changes to the game, the sequence diagram will still be relevant and useful. If too many specifics had been used, there is the risk that when something changes in the game, the diagram would no longer serve a purpose.