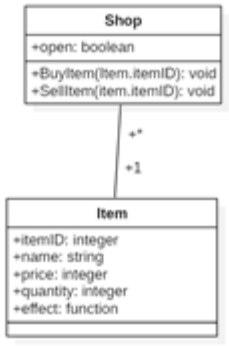
# Architecture

## Architecture Representation

To start with we needed to create an abstract version of our game so that we have something we can see and understand better than the requirements we were given, when it comes to seeing what's going to work and what needs improving/changing. We started this by working on a more detailed list of requirements which gave us a concrete structure to start building classes and elements which we would need in our game **[1]**. We also created a basic paper prototype which incorporated all the main elements of the game such as a ship travelling across the map just to see what elements we created and used, in order to recognise the necessary requirements and what was in fact redundant. Using these very high-level abstractions of the game we iterated through until we thought we had a good enough understanding of the game's functionality to create a conceptual UML diagram of the game. This is still a very vague model of the game as it's still early in the development stages and so things may change and there may be better ways to implement certain aspects.

Being as this version of the UML document is only a conceptual design some details such as some of the attribute's data types have been omitted as we thought it better to leave them out and sort it out later rather than put an erroneous type in and cause problems later. Another thing we decided to leave less concrete was some of the association, as this is the first game most of us are creating we aren't quite sure how it will all eventually fit together so we kept the classes we needed in but where we were uncertain to their associations in the game as a whole they were left to avoid creating a spider's web of associations which would leave us even more confused. Finally, we left out a few of the more complex lower level functions which at this point would just make the diagram harder to read.

To describe the architecture of our game we chose to represent it in UML 2 using a program called StarUML for reasons discussed in the method selection document **[2]**. UML is a standard modelling language that most software engineers use in industry so it's a great tool for us to model our game. UML can model all sorts of different systems, for ours, we used it as a class diagram to demonstrate how all our classes fit together and their attributes and operations.




Each box on the diagram represents a class with the name of that class in the top of the three sections, the middle section indicates the attributes of that class and the bottom is the operations. Most of the classes are associated with each other and this is signified by the lines between them, a single line with nothing on the connecting ends signifies a simple association for example the association between shop and Item is a simple association, next to the line is also the multiplicity of the association, this example shows that the multiplicity of shop to item is many to 1 which means the shop can have many items but there is only one shop for the items to be in. we haven't finalised how many items will be in the game yet so at the moment this reads there can be infinite items however there won't quite be that many, unfortunately.

There are also some associations which are a line with a diamond on one of the connections, these signify that one is composed of the other class. This is a special association which means they are very strongly connected, and the child cannot be there without the container class. We were

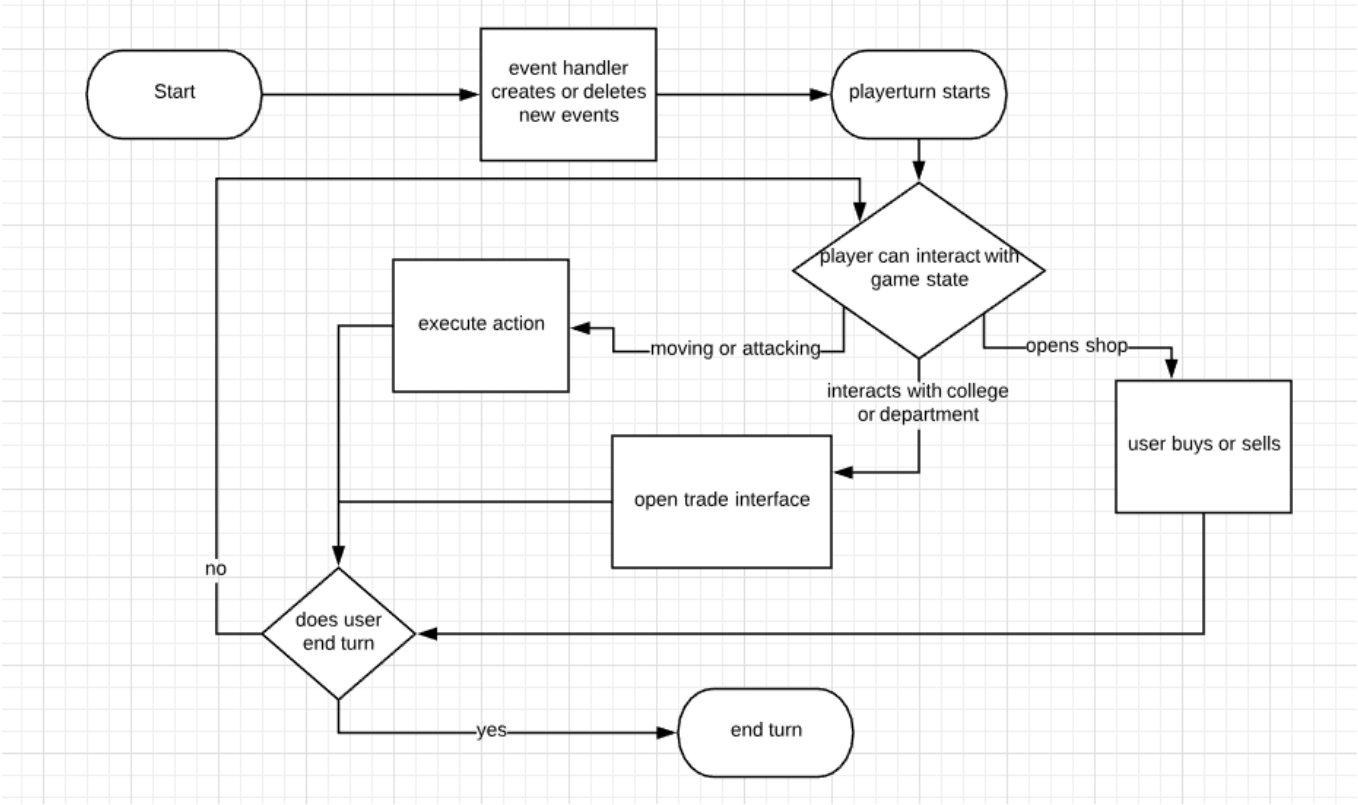
confident enough to use this in the diagram as we felt there was a strong containment within the event handling as well as the overall game containing the instances of Players, Maps and HUDs.

Some of the attributes are missing data types due to aforementioned reasons however most of the attributes are laid out in the format of 'attributeName: DataType'. Similarly the operations are of the form 'operationName(parameter1: DataType, parameter2: DataType….): ReturnType. Lastly the player class is in italics and this signifies it is an abstract class.

## HUD

Aside from the game loop, the player will also be able to interact with the HUD to display any information they wish to see at a given time. This will help them understand what events are occurring, the map state, and their next objective. The HUD will have have a minimap that shows a small overview of the map with dots to signify key objects like events and buildings. It will also be used to quickly get navigate the camera around the map by clicking on the minimap. There will also be a button to open a fullscreen detailed map that shows the player everything they can see.
The player will also be able to focus on categories of information such as their ships, buildings, technologies, or upgrades. This will help give them an idea of what they have and help manage their resources and units.

## Game Loop




The game loop shows the possible structure logic of the program cycle, referencing the architecture relevant to each step. The start of the game loop is considered when the player begins the game (after the main menu).

The game loop starts by handling any random map events such as weather, treasure or any other random events appearing by creating new ones or removing events such as weather that has lasted a number of turns. After this is shown to the player this moves to the bulk of the loop and most of the logic, being the player's actions through their in-game turn.
The player can make any of the following actions in any order until they choose, as well as repeating them (if applicable), to end their turn:

- Move ships
    - A ship can move up to its given maximum per turn, the player can move the ships in any order or the same ship multiple times
    - A ship can still attack after it moves
    - A ship cannot move after it attacks
- Use the ships to attack
    - Each ship can attack once per turn, and similar to the movement the player can choose the order at which their ships attack (or not attack)
    - When the ships attack, this will open up a combat interface. This interface or combat system is yet to be fully decided upon, but will most likely be a simple and short interaction using the existing HUD to display fight information and options
- Accessing the shop
    - The player can access the shop, which will be done by a menu or clicking their main college building
    - The user can purchase the following:
        - Creating new ships
        - Repairing existing ships
        - Upgrading existing ships
        - Buy items to use on the map
        - Upgrade overall technology (acquired from departments and other colleges)
- Interacting with other factions
    - The player can interact with departments or other colleges by trading with them or talking to them
        - Trading will involve trading gold or other resources to gain an exclusive item or technology from the other faction
        - Talking to the department or college would be used to progress the narrative of the story, as well as dialogue choices that impact the way the story progresses
- Play the mini-game (one-off)
    - When certain conditions are met the player will be given the option to play a mini-game which is entirely separate from the main game, and simply gives points which don’t impact gameplay

## Architecture Justification

There are different levels of abstraction and our UML class diagram is only a conceptual model which is the most basic type all it needs to do is meet the requirements and doesn't have much functionality to it. In making this we had to go through all the requirements and make sure they were in the class diagram in some form. The justification of the classes is such:

Game

The Game class is used to start up the game, the render function is what loads up the map, req 1.12, so that the 5 colleges and 3 departments are there as per req 1.1, which is why the map class is composed of this class. It also generates the players both the AI and Human as well as loading the HUD req 1.11. This is also where all of the gameplay stems from, the playerMove() function controls what happens next.

Map/Minimap/Square

The map class stores the data for the "board" that the game is played on - similar to a chess board. Each square on the board will have associated data such as current units occupying it, events (such as weather) as well as if the player has vision over it (for the fog of war, req 1.13) and more. The map is stored as a 2D array of the square data type. The camera class is what allows only a certain area of the whole map to be displayed. req 1.9.
The minimap is displayed on the HUD, req 1.11, and shows the player a small representation of the entire map at all times. The minimap calls the same data as the full-screen map as well as the main game view that the camera focuses on. However, it will only show small dots that will help tell the player at a glance where things generally are on the map.

Player

The player class contains information about the objects the player owns. This includes things such as gold, items, ships, and buildings(req 1.4), but also their points (req 1.5) and other non-gameplay variables. In the future, the player class will also contain data about their current status towards game objectives - which impact the progression of the game and the story (req 1.2). PlayerAI and PlayerHuman are children of the Player class, we decided to have these inherit the player class instead of having them two separate classes to avoid duplication.

HUD

The HUD will be rendered on the screen at all times other than the menus or full-screen map and will contain information on any currently focused object such as a ship, building, event, or simply just a tile. There will also be a section to display data from about the game state such as ships, buildings, or upgrades - all data of which is ultimately stored in the game class (referenced from each specific class for the unique data types). req 1.11.

Events

The events are handled by an EventHandler class which will be called at the start of each turn of the game loop. This will check existing events status to modify or delete them, and then create new ones in suitable locations based on certain rules defined in the class. The main game class will then call this function to update the map with the events, which will then be shown to the player through the maps available as well as the HUD. req 1.10. During the events the switch from sailing mode to combat mode will take place. req 1.3.

Shop/Item

The shop and item classes are where the player can use their gold. It's a very simple design with just a window that pops up on the HUD where the user can select to buy or sell items, as well as upgrade ships. req 1.4.2

Ship/college

The primary requirement of the game is that The map must contain at least 5 colleges and 3 departments and a large body of water (req 1.1). the college class will have instances of each of the colleges and departments and their effect, what they have to offer, this can be items, bonuses or just gold.

Minigame

The mini-game will be handled as a separate part of the program entirely and will just be called in the game loop when the conditions to run it are met. This decision was made since the game will be entirely separate from the rest of the architecture and so won't need to call or reference any of the data or functions used by the main game. req 1.6.1.

**Bibliography**

[1] Requirements - LimeWire

[2] Method Selection and Planning - LimeWire