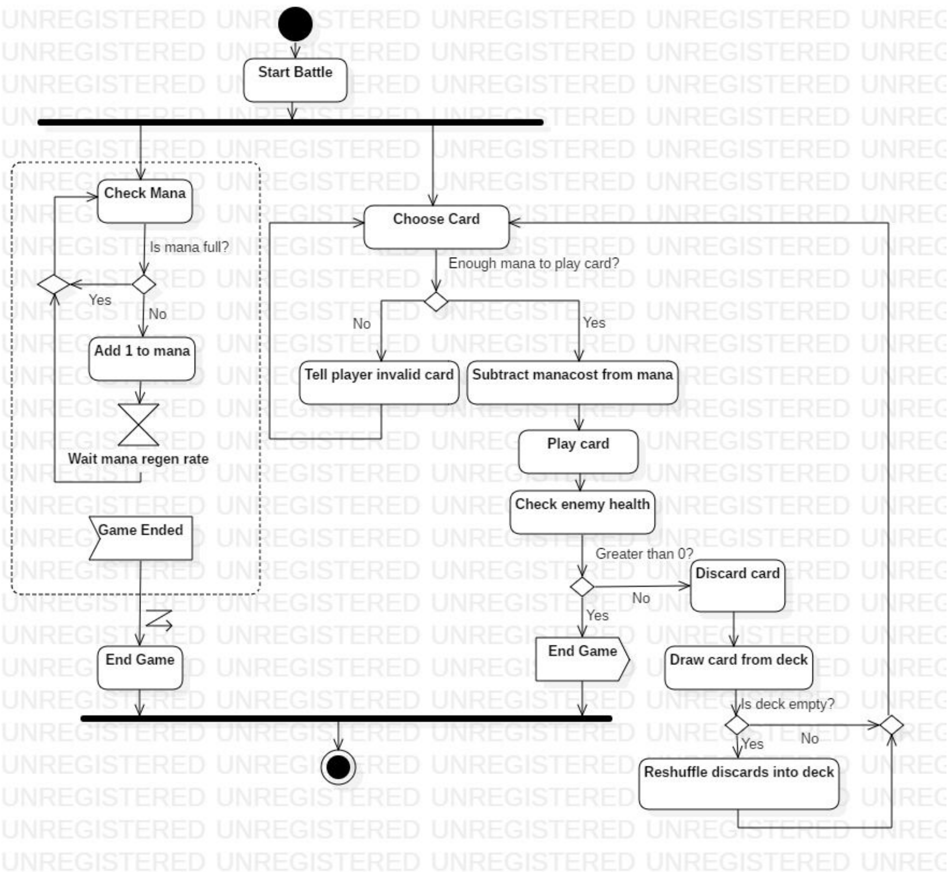
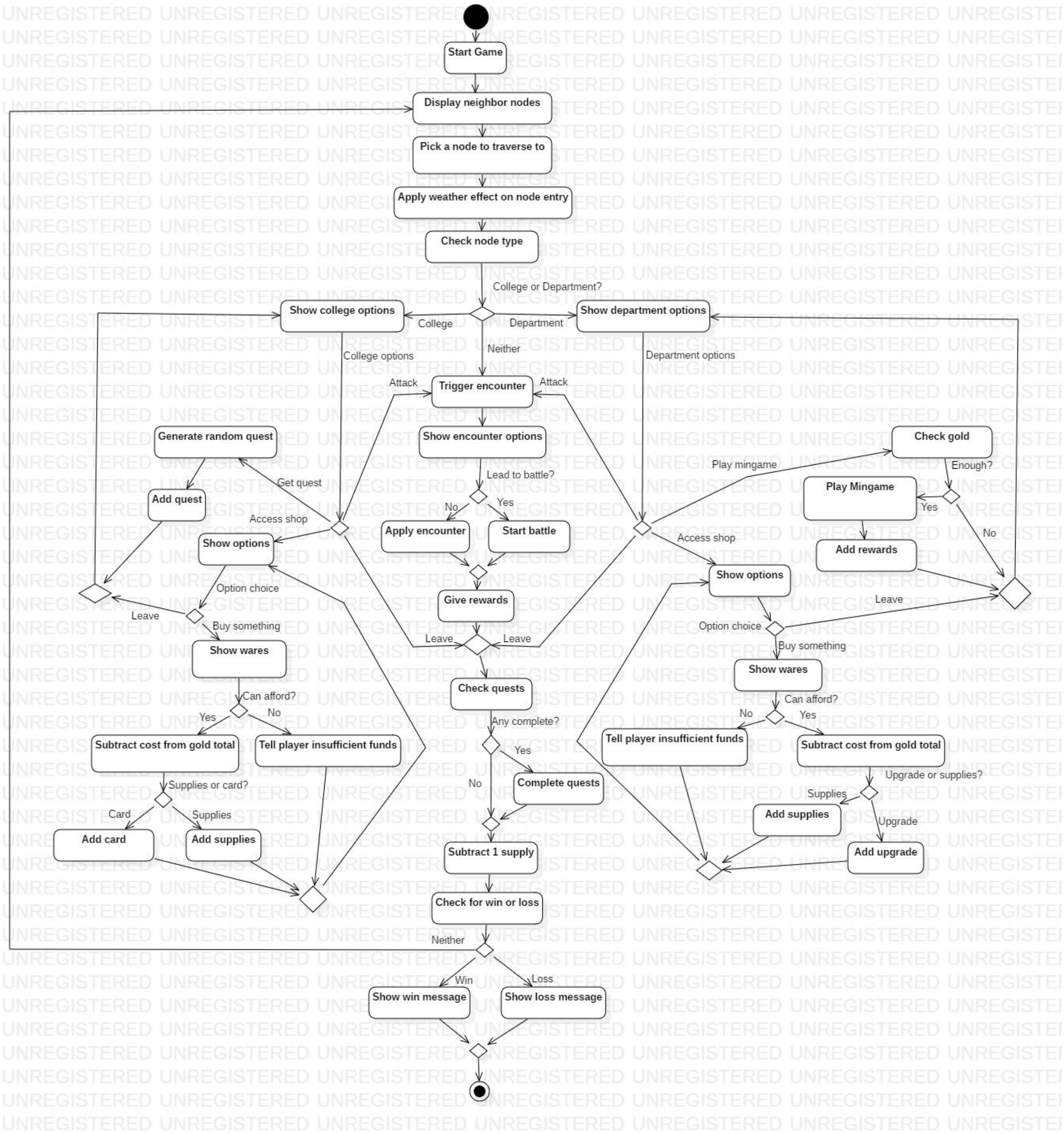
In our proposed architecture, we used a program called StarUML [1] to implement UML 2 diagrams. We chose UML 2 due to its use as an industry standard with regards to tools to create diagrams as well as the diagrams themselves, and the availability online of general requirements and restrictions. This should make it easier to more rigidly follow UML 2 guidelines by utilising tools which help restrict what you can do when drawing a model as well as make it easier for others outside the group to understand the architecture, which is essential in this project as games are swapped later on. StarUML was chosen due to its ease of use, which we considered important because no members had prior experience utilising UML or any other modelling language to create models for a system architecture.

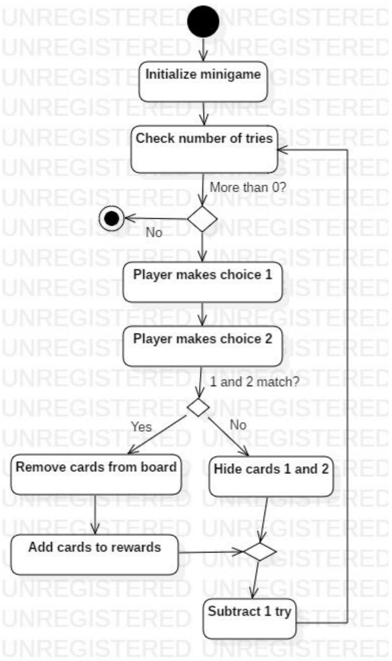
After going through the design requirements given by the brief and clearing our ideas with the client, we created UML activity diagrams to try and map through how our game is going to be played. This first diagram we made was for the battle section of the game, and was done following UML2 conventions [1] for activity diagrams and followed the process from the perspective of the player. This describes the game play loop during the battle, going through the process of playing cards and updating the mana throughout the battle. Whilst this is describing the loop for the player, the loop from the perspective of the enemy ship AI is almost identical. In the AI's case there would have to be some additional process to make the choice of card. Due to the real-time nature of battles it includes the recurring mana management within an interruptible action on game end, which is triggered by either ship reaching 0 health. Game end can be triggered by either the player or the enemy ship, which isn't really shown by the diagram due to it being purely from player perspective. Any gold costs for cards (which only applies in specific cases) is not included in this diagram for brevity's sake, but it can be assumed that they are checked and applied simultaneously along with mana.



[1] "StarUML", *StarUML,* 2018. [Online]. Available: http://staruml.io/ [Accessed: 25-Oct-2018]
[2]"UML Activity Diagrams - Graphical Notation Reference", *uml-diagrams.org*, 2018. [Online]. Available: https://www.uml-diagrams.org/activity-diagrams-reference.html [Accessed: 03- Nov- 2018].

The second activity diagram is that of the sailing portion of the game, again from the player's perspective. The turn based nature of the sailing lends itself nicely to the activity diagram, where actions can be taken sequentially and then loop back to the beginning if the game hasn't finished as a result of the turn. We made sure to include checks on the game state at the end of the turn in order to make sure that the game ends properly if the overall objective (which cannot be completed immediately) is completed on that turn. Supplies is subtracted just after turn completion because it puts it in order to avoid immediately losing supply on initialization and to allow quests to potentially add to supplies before subtraction.

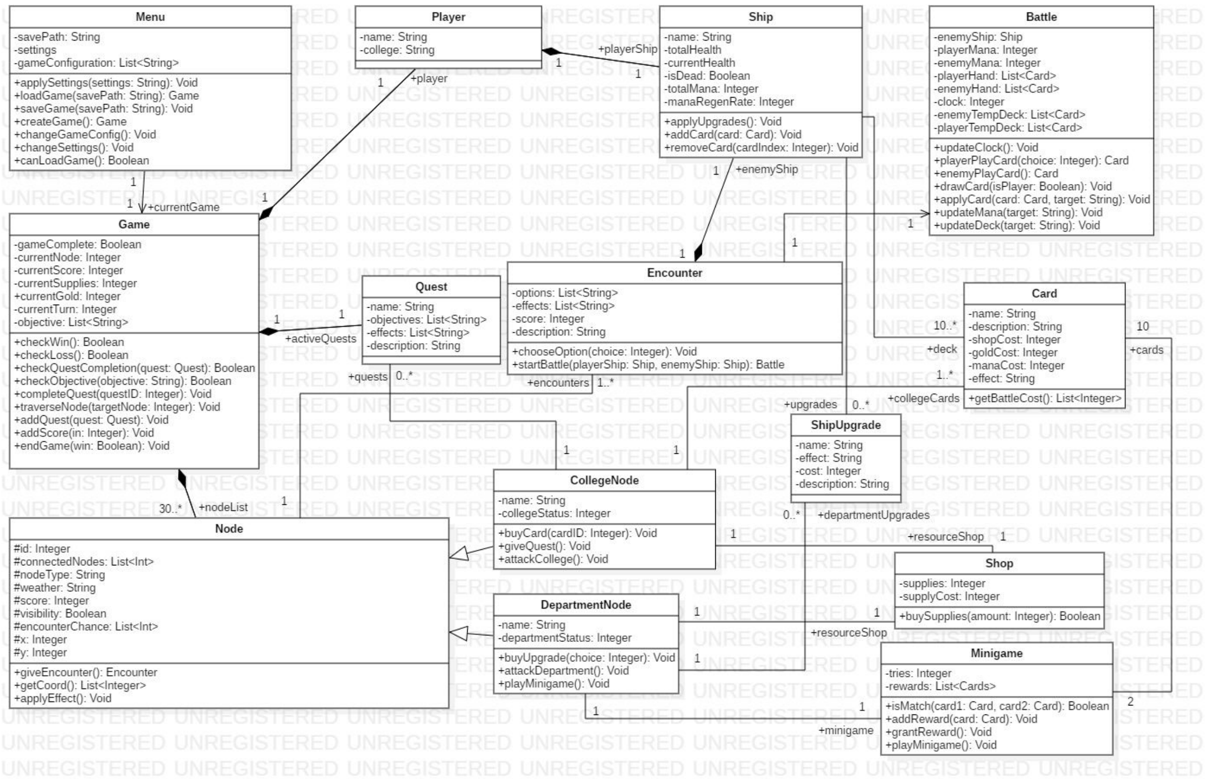




Our third activity diagram outlines the process followed by the minigame which can be called from a department. This follows the same guidelines as above

Through the use of our activity diagrams as well we devised an initial class diagram for the structural model, with the inclusion of a menu class. This was made to manage any settings implemented during development as well as allowing for there to be a way to start/create games. Whilst not elicited as a requirement, a save functionality was included as a possible future requirement due to how common it is. Beyond that, language used follows UML guidelines as outlined in our lectures relating to UML, with the inclusion of uni-directional relations for Game and Battle. This diagram was then iterated upon and changed as it was being created to account for anything not outlined explicitly in the activity diagrams.

Actions/operations such as “Play card” are left deliberately vague in order to allow for different interpretations on the sort of cards that can be made, as we feel that there is such a variety that we don’t want to constrict ourselves by our model and to let it be expansive. All diagrams in our model can also be found on our project website.



**Architecture Justification**

Most attributes are private by default, or for Nodes, protected. This is to prevent classes corrupting the other classes and to preserve data encapsulation. If an attribute doesn't follow this, the reasoning will be explained separately. The architecture attempts to follow an abstracted OOP design, as although using Java is required, an abstracted design gives more implementation flexibility and avoids cluttering with implementation specific design details which may not apply.

Operations such as constructors, or get/set operations for individual attributes, are also generally ignored even though they may appear in any implementation of the software. They were only included if they followed a non-conventional design or were specifically relevant, avoiding overcrowding the diagram.

**Game:** The class *Game* contains the majority of the functionality and loop for sailing mode, including the overworld map, the resources of player, active quests and the majority of functions required for game play outside of battles. This includes the map management, consisting of *currentNode*, *nodeList* and *traverseNode*. *currentNode* and *nodeList* manage player position on the map and their available movement, whereas *traverseNode* functions as both navigation of the map as well as the turn change (updating *currentTurn*), with many different operations occurring during or with relation to turn change. Resource/score management is also included within *Game*. As *Game* encapsulates the majority of the game, attributes such as *currentScore*, *currentSupplies* and *currentGold* function almost as psuedo-global variables for the software, which is especially relevant for *currentGold* which remains a public attribute for ease of use in *Battle*. *Player* could also be considered as a part of this group as it needs to stay persistent through encounters and as such is kept in *Game*. *addScore* is included as *Score* is only added to, so a traditional set operator is unnecessary. Finally, another key aspect of the sailing mode is also managed in this class, that being the overall objective in *objective* as well as quests (*activeQuests*). These are frequently evaluated on turn change and after encounters as well as mainly affecting the global resources, so keeping them within the *Game* class makes the most sense to give them easy access to these attributes. Check operators allow for different variables, such as game won/lost or quest completed, to be checked on every turn quickly. This ensures when the game is won/lost it actually triggers.

**Node:** The general *Node* class is used for the map, with relevant information pertaining to the nodes as well as the main body of the sailing mode, *encounters*. Possible encounters are based off the current node, the function to give encounters is contained in the Node class as well as a list of *encounterChances* for each encounter. This allows for a variety of encounters to be available for each generic node, also forcing specific encounters such as an end game encounter, a quest encounter or attacking a college/department. Other relevant node-specific information (meaning it changes node to node) include the *ID* and *connectedNodes*, allowing for the formation of a node map for the player to traverse, the node's type and weather, which could impact the player, by taking extra resources to traverse (implemented through the operation apply effect), the score granted by traversing a node, the visibility of the node to the player and its cartesian coordinates x and y, utilised for finding neighbours and displaying the node map as a 3D image. Implemented by *getCoord,* a get operation included as it returns a list of x and y together, limiting the need for two get operations.

**CollegeNode:** Subclass of *Node* (Colleges can only exist in nodes) containing college attributes. There must be 5+ Colleges do not automatically trigger encounters, so it contains a specific *attackCollege* function to allow the player to manually trigger an encounter. The infrastructure for buying cards/issuing quests exist within this class despite active quests and the players deck being contained elsewhere, as cards and quests are accessed at colleges.
**DepartmentNode:** Subclass of *Node* as all departments exist within nodes. Like Colleges, departments do not automatically have encounters, upgrades can be bought at department instead of cards. Departments also contain the minigame, a matching card game which can be played at a gold cost. There must be 3+ departments.
**Quest:** A class signifying quests, this contains the simple data types relating to quests as actual quest functionality is handled by the *Game* class.
**Shop:** The primary way for players to gain supplies, appearing in both *DepartmentNodes* and *CollegeNodes*. It is not part of a superclass of both because, whilst it isn't currently implemented, there has been consideration of shops appearing in random encounters. Providing a class of its own favors future adaptability. Buy supplies is included as buying supplies is a game mechanic, and acts as a test to ascertain if there was a legal buy.
**Card:** Not unlike *quests*, *Card* consists mainly of simple data types relating to usage as card use is managed in other classes. *getBattleCost()* contains both the *goldCost* and the *manaCost* together so they can be handled simultaneously when the card is played.
**Minigame:** A class managing the minigame, which is playable at a *DepartmentNode*. Contains the operations to play the game, such as match checking and player rewards, and the game manager operation in *playMinigame*.
**Player:** *Player* contains the base information relating to the player, such as *name* and affiliated *college* and most importantly their *playerShip* which is essential for battles.
**Ship:** *Ship* is the main class for implementing both the player and enemy within a *Battle*. Containing stats such as health/total health of ship, mana cap/regen rate and card deck available to draw from. Upgrades for ships can be bought at *DepartmentNodes*, and these have a range of effects to both the stats and the cards. Operations in the class do not relate only to ship management, not battle management, such as upgrades and deck alterations.
**ShipUpgrade:** A class consisting of basic data types, as actual implementation of upgrades is managed by *Ship* and buying upgrades is done in *DepartmentNodes*.
**Encounter:** An *encounter* consists of a set of options and effects, displayed appropriately, as well any potential enemy ship involved in the encounter. These are managed in class by *chooseOption*, which takes in the choice from the options and applies the effect relating to the index of that option. Options can lead to different battles per encounter, battle starts are triggered and managed in *Encounter*. During the battle, it is managed internally in *Battle*.
**Battle:** The class contains all information and operations relevant to conducting a battle in *game*. Temporary data relating to the battle is stored here such as the combatant hand or mana. The clock is used for time management, specifically for mana, whose generation is tied to time passed. The operations execute the game play loop of the battle, with relevant operations such as player's choice/generating enemy's choice, drawing/playing cards, and general management of the game state during battle such as updating mana, refreshing emptied deck and updating the clock (functioning as a check on battle state and performing real time actions). This remains self contained as relevant outside information can be accessed from other classes and managing everything underneath one class, not unlike *Game*, made the most sense logically. Rewards are considered an *Encounter* effect.