2021/2022

ESINF-2NA-G169

CLASS DIAGRAM JUSTIFICATION AND COMPLEXITY ANALYSIS

Afonso Pimentel 1151399

Rui Rafael 1181357

Ricardo Ribeiro 1181350

Pedro Matias 1181544

Table of Contents

[User Story 401 3](#_Toc93770361)

[Class Diagram Justification 3](#_Toc93770362)

[Complexity Analysis 3](#_Toc93770363)

[User Story 402 4](#_Toc93770364)

[Class Diagram Justification 4](#_Toc93770365)

[shortestPathTerrain 4](#_Toc93770366)

[shortestPathNPlacesBruteForce 5](#_Toc93770367)

[shortestPathNPlaces 5](#_Toc93770368)

[Complexity Analysis 5](#_Toc93770369)

[User Story 403 6](#_Toc93770370)

[Class Diagram Justification 6](#_Toc93770371)

[graphCircuit 6](#_Toc93770372)

[theCircuit 6](#_Toc93770373)

[findingNemo 6](#_Toc93770374)

[Complexity Analysis 6](#_Toc93770375)

# User Story 401

## Class Diagram Justification

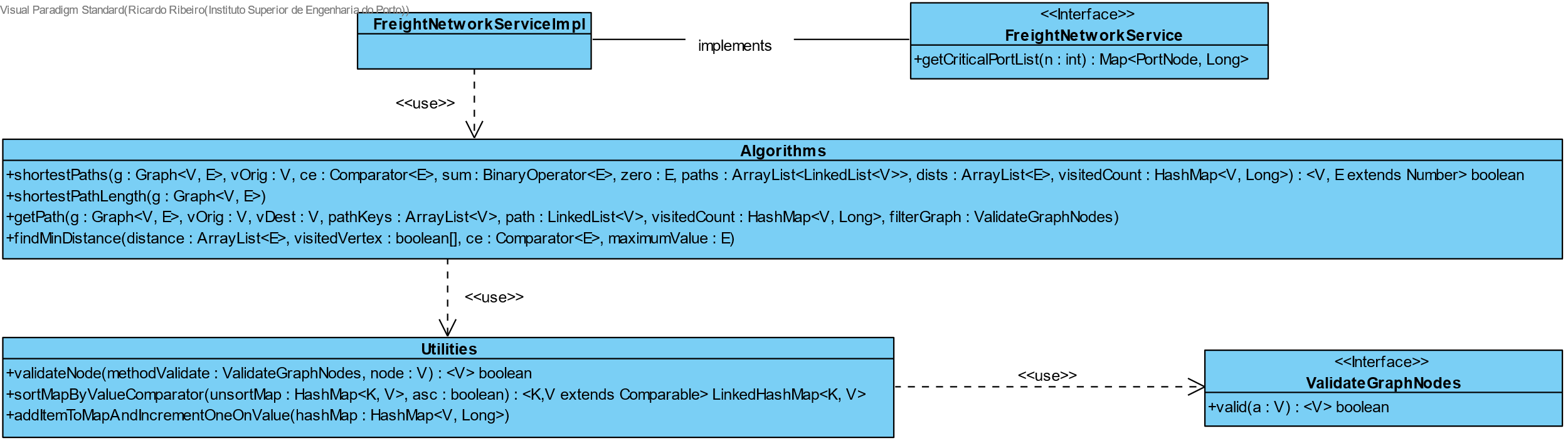


Figure -US401 Class Diagram

US401 Says that a traffic manager needs to be able to know which N ports are more critical in this freight network which means the traffic manager wants to know which ports have the greatest number of shortest paths going through them.

With that being said, our team implement created a public that receives the N number of ports that the Traffic Manager wishes to see and then for each vertice in the graph it will calculate every shortestPath it has using our **shortestPaths** method, counting inside how many times each port appears. After the shortestPaths is complete we order the results in a LinkedHashMap to keep the data properly sorted.

## Complexity Analysis

Since the number of functions used for the implementation of this US were 5 different methods varying their sizes, **we decided to do our Complexity Analysis in an Excel file that we will be submitted alongside this report in the sheet named US401**. This way we seek to improve the teacher’s readability by not filling the report with images.

This US has a total complexity of **O(n^3) in the worst-case scenario.**

# User Story 402

## Class Diagram Justification

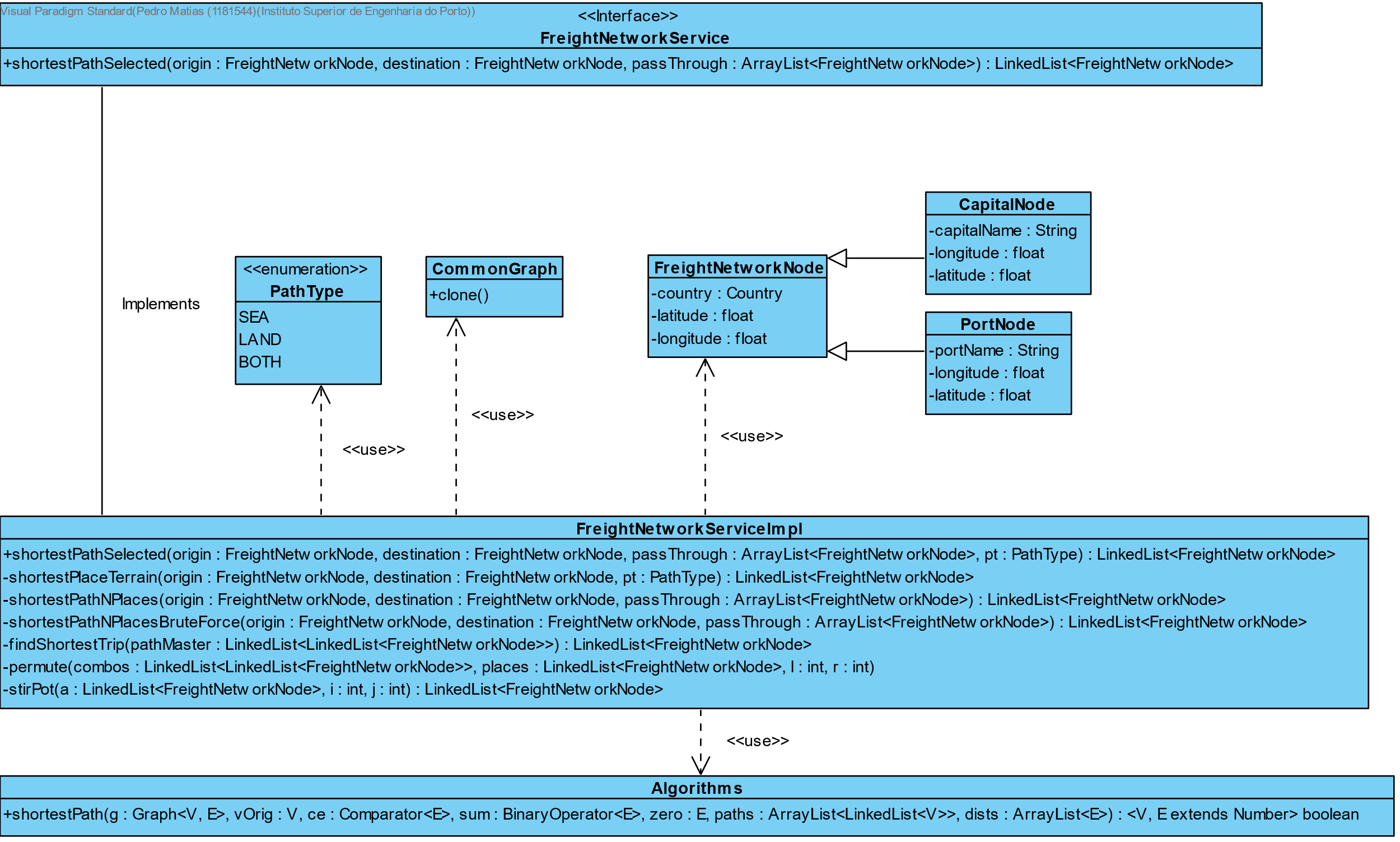


Figure -US402 Class Diagram

US402 Says that a traffic manager must be able know the shortest path between two places, they can be both cities and/or ports. Additionally, the traffic manager also must be able to specify if he wants to go only by land and/or sea. The last acceptance criteria also mentions that the traffic manager can specify N number of places he wants to stop in between the starting and ending point of a trip.

To comply with these acceptance criteria our team decide to create the following structure:

We created a public function named **shortestPathSelected** which will call other function depending on

* If the Traffic Manager didn’t specify any places to stop in between.
* If the Traffic Manager did specify 5 or less places to stop in between.
* If the Traffic Manager did specify 6 or more places to stop in between.

Entering in more detail on each one:

### shortestPathTerrain

If the Traffic Manager didn’t specify any places to stop in between, It will create a clone of the original graph and call **shortestPathTerrain**, passing the graph clone, origin, destination, and path Type as parameters.

This function essentially will remove the clone graph vertex depending on type of terrain the Traffic Manager chooses, if it chooses **Land only** it will remove all port nodes from the graph, with exception to the destination and/or origin if these are ports. And it will do the same logic if it chooses **Sea only** removing all land nodes from the graph. \*

\*We know that this isn’t the most efficient way to do this. We should’ve implemented this ‘filter’ inside the Dijkstra algorithm but due to our lack of time we decided to keep things like this since it was the first thing we implemented.

After filtering the clone graph, we apply the Dijkstra algorithm to find the shortest path between two vertices, returning a LinkedList with all the vertex that make that shortest path.

Making the complexity of this function O(n^2). You can verify this in the Complexity Analysis.

### shortestPathNPlacesBruteForce

If the Traffic Manager specified 5 or less places to stop in between its origin and destination, then we call **shortestPathNPlacesBruteForce**, passing the origin, destination, and the list of places to stop in between as parameters.

This function will create every **premutation** possible to create every possible sequence of stops considering the list that was passed. This way we guarantee that we compute and find the shortest path possible given the Traffic Manager choices. After creating every premutation we compute **each shortest path** using **Dijkstra algorithm** and then we compare the total distance of every permutation created and return a List with the shortest path found.

Making the complexity of this function O(n!) which is by far not the most efficient method but the objective here was to find the best possible solution and not the most efficient one.

### shortestPathNPlaces

Knowing that the function shortestPathNPlacesBruteForce could scale uncontrollably being O(n!) we decided to implement a more efficient solution if the Traffic Manager decides to stop 6 or more times in between origin and destination that function is **shortestPathNPlaces.**

This function will see which in between vertices is the closest to the current we are on, using **Dijkstra algorithm**, advancing to the next stop when it finds the closest one, repeating this process until there’s no in between stops left to go, at the same time constructing the path.

This way we ensure that our method will be faster compared to the brute-force method and at the same time we will return a path that is most certainly not the longest one, making this solution more viable than the brute-force method in a large-scale situation.

Making the complexity of this function O(n^4) which is a lot more efficient than the brute-force method.

## Complexity Analysis

Since the number of functions used for the implementation of this US were 10 different methods varying their sizes, **we decided to do our Complexity Analysis in an Excel file that we will be submitted alongside this report in the sheet named US402**. This way we seek to improve the teacher’s readability by not filling the report with images.

In the end this US has a total complexity of **O(n!) in worst case scenario.**

# User Story 403

## Class Diagram Justification

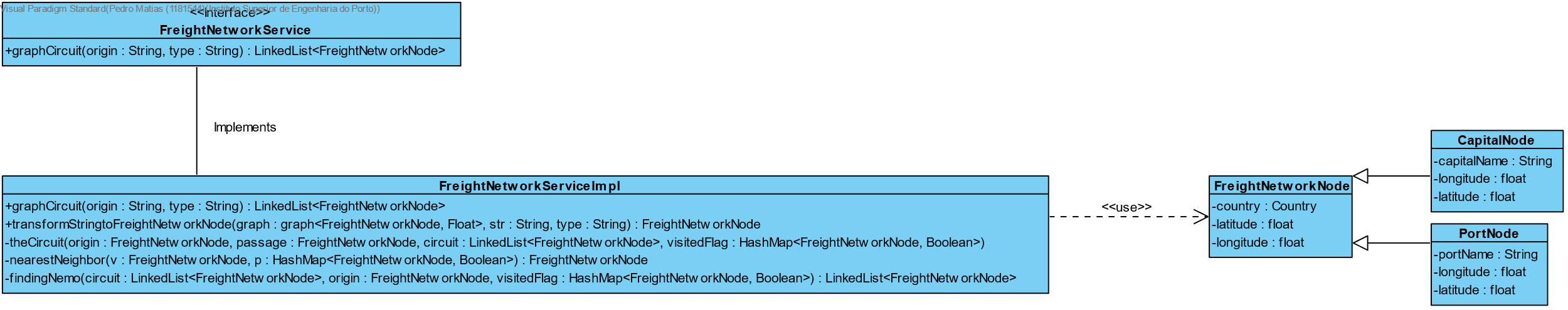


Figure - US403 Class Diagram

US401 States that the Traffic Manager wants to know the most efficient circuit that start at a certain point, and it must pass the maximum number of places while maintaining the shortest distance possible, only passing in each location once.

To achieve that our team decided to implement the nearest neighbor heuristic, since implementing a brute-force method here would not be practical at all, in order to find possibly the largest circuit with the smallest distance, since this is and heuristic it means that the result may not be the most optimal one, but it will be enough since it will a lot more efficient compared to a brute-force method.

Entering more in detail on each method we implemented:

### graphCircuit

We implemented a public function that will receive our starting point and it will retrieve in the end our circuit which is **graphCircuit.** Essentially it will validate the input and it will also validate if a circuit is possible by counting its adjacent vertices.

### theCircuit

After that it will enter in its first recursive method which is **theCircuit** function, which will start constructing a circuit with each nearest neighbor (using our function **nearestNeighbor**) of the vertice we are on, this will repeat up until the function as went through every single vertice in the graph **once**, discarding dead ends and saving the rest. Once it passes every vertice one time it will clean every node that wasn’t saved from the visited HashMap, and it will enter a second recursive method named **findingNemo.**

### findingNemo

The objective of **findingNemo** function as the name suggests is, while going through the previous saved list, from the bottom up, it will go through every adjacent vertice (using **nearestNeighbor** to ensure we always prioritize the smallest distance possible) trying to find the starting point, discarding dead ends. Which eventually will result in the starting point being found, therefore making the circuit complete.

## Complexity Analysis

Since the number of functions used for the implementation of this US were 4 different methods varying their sizes, **we decided to do our Complexity Analysis in an Excel file that we will be submitted alongside this report in the sheet named US403**. This way we seek to improve the teacher’s readability by not filling the report with images.

This US has a total complexity of **O(n^2) worst case scenario**.