SPRINT 1

RELATORY
ESINF

1201518 | ANA ALBERGARIA

1201284 | DIOGO VIOLANTE

1200049 | JOÃO WOLFF

1201592 | MARTA RIBEIRO

G54

CLASS 1DF

PROFESSOR: FÁTIMA RODRIGUES

NOVEMBER 2021

Índice

I. US101 – Import Ships from a text file into a BST

1.1 Problem and justification

US-101-As a traffic manager I wish to **import ships from a text file into a BST**.

- **Acceptance criteria:**
  - No data lost.
- **Contributes:**
  - **Code, tests and complexity analysis:**
    - Marta Ribeiro, 1201592

1.2 Class Diagram Analysis

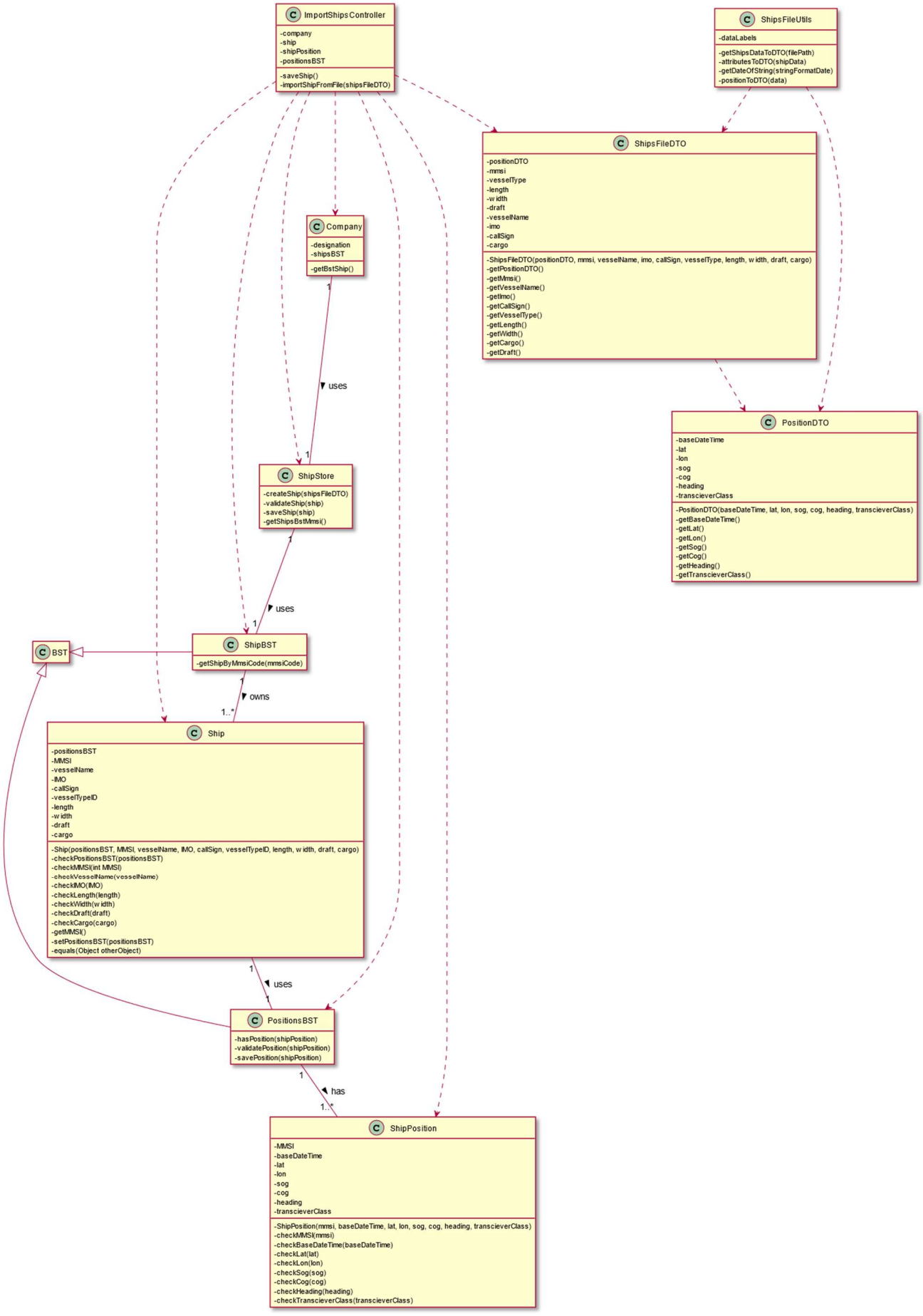
The Class Diagram contains the relevant concepts from the Domain Model for the US101: Company, Ship and ShipPosition.

Using good practices of OO software design, a Controller Class was created – `ImportShipsController`.

To meet the acceptance criteria of the US, a Binary Search Tree for the Ships in the system – `ShipBST` – was created. It was also created a BST for the Ship's position – `PositionsBST` – , so that each ship would have one and store every of its Positions. To successfully implement the BSTs, there were created two important Objects: `Ship` and `ShipPosition`. Both the `ShipBST` and the `PosiitionsBST` inherit the class `BST` develop during ESINF classes.

To import a Ship from a .CSV file, all the data read from the file is passed to a controller in the format of a `ShipsFileDTO`, to reduce coupling between layers and reduce the number of arguments needed. There is also a `PositionDTO`, contained by the `ShipsFileDTO` to know the information related to the Position of the Ship.

Considering all the solicited requisites and to ensure them, the following Class Diagram was designed:


ImportShipsController
-company
-ship
-shipPosition
-positionsBST
-saveShip()
-importShipFromFile(shipsFileDTO)
ShipsFileUtils
-dataLabels
-getShipsDataToDTO(filePath)
-attributesToDTO(shipData)
-getDateOfString(stringFormatDate)
-positionToDTO(data)
ShipsFileDTO
-positionDTO
-mmsi
-vesselType
-length
-width
-draft
-vesselName
-imo
-callSign
-cargo
-ShipsFileDTO(positionDTO, mmsi, vesselName, imo, callSign, vesselType, length, width, draft, cargo)
-getPositionDTO()
-getMmsi()
-getVesselName()
-getImo()
-getCallSign()
-getVesselType()
-getLength()
-getWidth()
-getCargo()
-getDraft()
Company
-designation
-shipsBST
-getBstShip()
uses
ShipStore
-createShip(shipsFileDTO)
-validateShip(ship)
-saveShip(ship)
-getShipsBstMmsi()
PositionDTO
-baseDateTime
-lat
-lon
-sog
-cog
-heading
-transcieverClass
-PositionDTO(baseDateTime, lat, lon, sog, cog, heading, transcieverClass)
-getBaseDateTime()
-getLat()
-getLon()
-getSog()
-getCog()
-getHeading()
-getTranscieverClass()
BST
ShipBST
-getShipByMmsiCode(mmsiCode)
owns
Ship
-positionsBST
-MMSI
-vesselName
-IMO
-callSign
-vesselTypeID
-length
-width
-draft
-cargo
-Ship(positionsBST, MMSI, vesselName, IMO, callSign, vesselTypeID, length, width, draft, cargo)
-checkPositionsBST(positionsBST)
-checkMMSI(int MMSI)
-checkVesselName(vesselName)
-checkIMO(IMO)
-checkLength(length)
-checkWidth(width)
-checkDraft(draft)
-checkCargo(cargo)
-getMMSI()
-setPositionsBST(positionsBST)
-equals(Object otherObject)
PositionsBST
-hasPosition(shipPosition)
-validatePosition(shipPosition)
-savePosition(shipPosition)
has
ShipPosition
-MMSI
-baseDateTime
-lat
-lon
-sog
-cog
-heading
-transcieverClass
-ShipPosition(mmsi, baseDateTime, lat, lon, sog, cog, heading, transcieverClass)
-checkMMSI(mmsi)
-checkBaseDateTime(baseDateTime)
-checkLat(lat)
-checkLon(lon)
-checkSog(sog)
-checkCog(cog)
-checkHeading(heading)
-checkTranscieverClass(transcieverClass)

1.3 Complexity Analysis

1.3.1 Code and Type Of Algorithm

The main functionality of this US is the import of Ships from a .CSV file. Due to the lack of an UI, the import is done on the `ImportShipsControllerTest` Class, as can be seen here:

```
private Company comp;
private List<ShipsFileDTO> shipsOfFile, shipsOfFileExp;
private File file1, /*file2,*/ fileTest, expFileTest/*, fileTestPairsOfShips*/;
private ImportShipsController ctrl;

@BeforeEach
public void SetUp() {
    comp = new Company( designation: "Company");
    this.shipsOfFile = Collections.emptyList();
    this.shipsOfFileExp = Collections.emptyList();
    file1 = new File( pathname: "data-ships&ports/bships.csv");
    //file2 = new File("data-ships&ports/sships.csv");
    fileTest = new File( pathname: "data-ships&ports/testFile.csv");
    expFileTest = new File( pathname: "data-ships&ports/expImpTestFile.csv");
    //fileTestPairsOfShips = new File("data-ships&ports/testImpShip366998510.csv");
    this.ctrl = new ImportShipsController(comp);
}

@Test
public void testImpShip(){
    ShipsFileUtils shipsFileUtils = new ShipsFileUtils();
    int j=0;
    try {
        shipsOfFile = shipsFileUtils.getShipsDataToDto(file1.toString());
        for (int i = 0; i < shipsOfFile.size(); i++) {
            if (!ctrl.importShipFromFile(shipsOfFile.get(i))) {
                System.out.println("DIDN'T IMPORT LINE " + i + "\n");
            } else {
                j++;
            }
        }
    } catch (IllegalArgumentException e){
        System.out.println("NOT ADDED : " + e);
    }
    System.out.println("TOTAL IMPORTED: " + j + "\n");
}
```

The main functionality's algorithm is **non deterministic**, because its complexity can vary depending on the data in the .CSV file.

1.3.1.1 Best Case

This functionality's best case scenario occurs when the file isn't formatted in the preferred way, as can be seen in this excerpt of the `getShipsDataToDto` method, contained In the `ShipsFileUtils` Class and called by the `testImpShip` function:

```
String line = bufferedReader.readLine();
if (line==null || !line.equals("MMSI,BaseDateTime,LAT,LON,SOG,COG,Heading,VesselName,IMO," +
        "CallSign,VesselType,Length,Width,Draft,Cargo,TranscieverClass")){
    throw new IllegalArgumentException("Incompatible file format.");
}
```

As the importation file follows a specific format, we should test first if that format is valid and if not, the method stops.

Time Complexity: O(1)

1.3.1.2 Worst Case

The worst case scenario of this algorithm is proportional to the number of Ships to import. That is, if the number of Ships to import is bigger, the time complexity of the algorithm is also bigger.

Time complexity: O(n)

Despite not changing the time complexity of the algorithm, is also important to note that the runtime of the algorithm will be bigger if the Ships to import don't have any type of error regarding their information, compared to the case in which there would be errors. This happens because, for each Ship, an amount of time will be waisted checking every possible error that could exist, but no exception will be caught. This can be seen in the following `importShipFromFile` method, contained in the `ImportShipsController` Class:

```
public boolean importShipFromFile(ShipsFileDTO shipsFileDTO) {
    if (this.company.getShipStore().getShipsBstMmsi().getShipByMmsiCode(shipsFileDTO.getMmsi())==null){
        try {
            this.ship = this.company.getShipStore().createShip(shipsFileDTO);
            this.shipPosition = new ShipPosition(shipsFileDTO.getMmsi(),shipsFileDTO.getPositionDTO().getBaseDateTime(),
                    shipsFileDTO.getPositionDTO().getLat(),shipsFileDTO.getPositionDTO().getLon(),
                    shipsFileDTO.getPositionDTO().getSog(), shipsFileDTO.getPositionDTO().getCog(),
                    shipsFileDTO.getPositionDTO().getHeading(), shipsFileDTO.getPositionDTO().getTranscieverClass());
        }catch (IllegalArgumentException e){
            System.out.println("NOT ADDED : " + e); return false;
        }
        return saveShip();
    } else {
        try{
            this.ship=this.company.getShipStore().getShipsBstMmsi().getShipByMmsiCode(shipsFileDTO.getMmsi());
            this.shipPosition = new ShipPosition(shipsFileDTO.getMmsi(),shipsFileDTO.getPositionDTO().getBaseDateTime(),
                    shipsFileDTO.getPositionDTO().getLat(),shipsFileDTO.getPositionDTO().getLon(),
                    shipsFileDTO.getPositionDTO().getSog(), shipsFileDTO.getPositionDTO().getCog(),
                    shipsFileDTO.getPositionDTO().getHeading(), shipsFileDTO.getPositionDTO().getTranscieverClass());
        }catch (IllegalArgumentException e){
            System.out.println("NOT ADDED : " + e); return false;
        }
        try {
            this.positionsBST=this.ship.getPositionsBST();
            if (!(positionsBST.hasPosition(shipPosition))) {
                return saveShip();
            }
        }catch (IllegalArgumentException e){
            System.out.println("NOT ADDED : " + e);
        }
    }
    return false;
}
```

II. US102 – Search the details of Ship using MMSI, IMO or Call Sign

2.1 Problem and justification

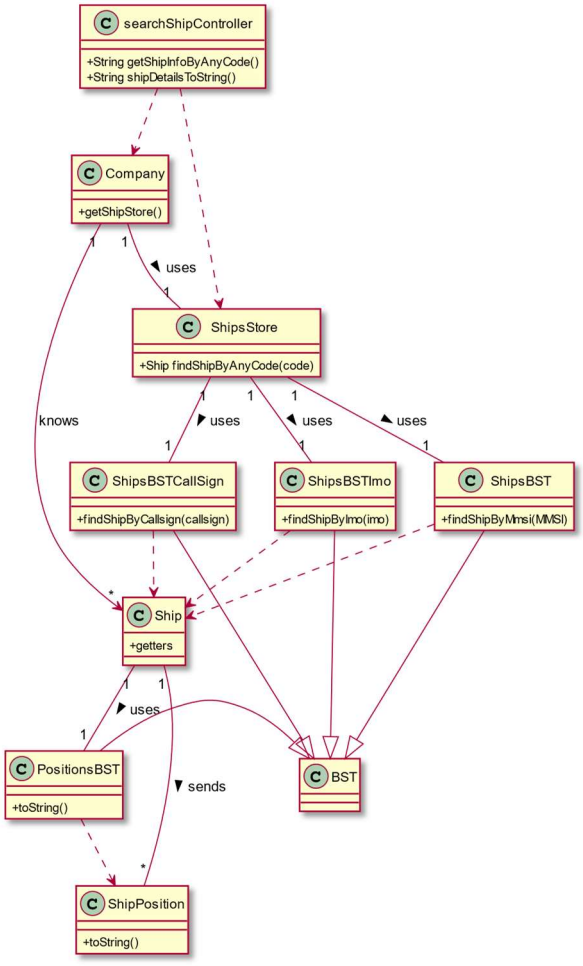
US-102-As a traffic manager I wish to search the details of a ship using any of its codes: MMSI, IMO or Call Sign.

- **Contributes:**
  - **Code, tests and complexity analysis:**
    - João Wolff, 1200049

For the task of searching for a ship and its details by any of the codes of it we have chosen to make use of three Binary Search Trees, where in each of the structures one of the codes were being compared.

2.2 Class Diagram Analysis

This leading to the three classes: ShipBSTCallSign, ShipBST and ShipBSTImo. All those which inherit the class BST develop during ESINF classes in the way that can be seen in the class diagram:



The way we made possible to exist three BST's relating the same ship object was overriding the insert method inside the callsign and imo tree classes in a way that it would use the given codes as reference for the ordering.

With the given structure looking forward to implementing the search with any code the user desires each tree has the method for retrieving the ship by its code, and the class which decides which method and tree to use is the ShipStore, with the method findShipByAnyCode(String code) that decides based on the syntax of the code the method to call.

2.3 Complexity Analysis

The main part of the search is done by three functions that behave the same and have the same time complexity:

```
public Ship getShipByCallSign(String callSign)  {
    return getShipByCallSign(root, callSign);
}

private Ship getShipByCallSign(Node<Ship> node, String callSign) {
    if(node == null)
        return null;
    if(node.getElement().getCallSign().equals(callSign)) {
        return node.getElement();
    }
    if(node.getElement().getCallSign().compareTo(callSign) > 0) {
        return getShipByCallSign(node.getLeft(), callSign);
    }
    if(node.getElement().getCallSign().compareTo(callSign) < 0) {
        return getShipByCallSign(node.getRight(), callSign);
    }
    return node.getElement();
}
```

The annexed image of the search method is the one for the callsign code, and the only change between it and the other methods is the attribute that is compared by the function.

This method has the same complexity as the BST search:

- It has the best case of O(1) if the searched element is the root of the tree
- A worst case of O(h) where h is the height of the tree
- If the tree is totally unbalanced it becomes the number of elements making it as bad as a search in a linked list with complexity O(N)
- And an average case when the tree is balanced, meaning that the height of tree is now log(n), with complexity of the search being O(logN)

2.4 Observations

There is one special remark to the solution of this task.

The ship class and its BSTs could make better use of the OOP concepts with an abstract ship class and three different classes implementing it one for each ship code, and this way having only one BST class and less repetitive code.

Unfortunately, we didn't think of this alternative in time and decided to make these major changes in the whole system architecture in the next sprint with more time to avoid unexpected breaking bugs, in order to deliver a functional product by the end of the sprint.

III. US103 – Show Positional Messages

3.1 Problem and justification

US103 - As a traffic manager, I wish to have **the positional messages temporally organized and associated with each of the ships.**

- **Acceptance criteria:**
  - Efficient access of any position value(s) of a ship on a period or date.
- **Contributes:**
  - **Code, tests and complexity analysis:**
    - Ana Albergaria, 1201518

3.2 Class Diagram Analysis

The Class Diagram contains the relevant concepts from the Domain Model for the US103: Company, Ship and ShipPosition.

Using good practices of OO software design, a Controller **Class was** created – ShowPositionalMessagesController**.**

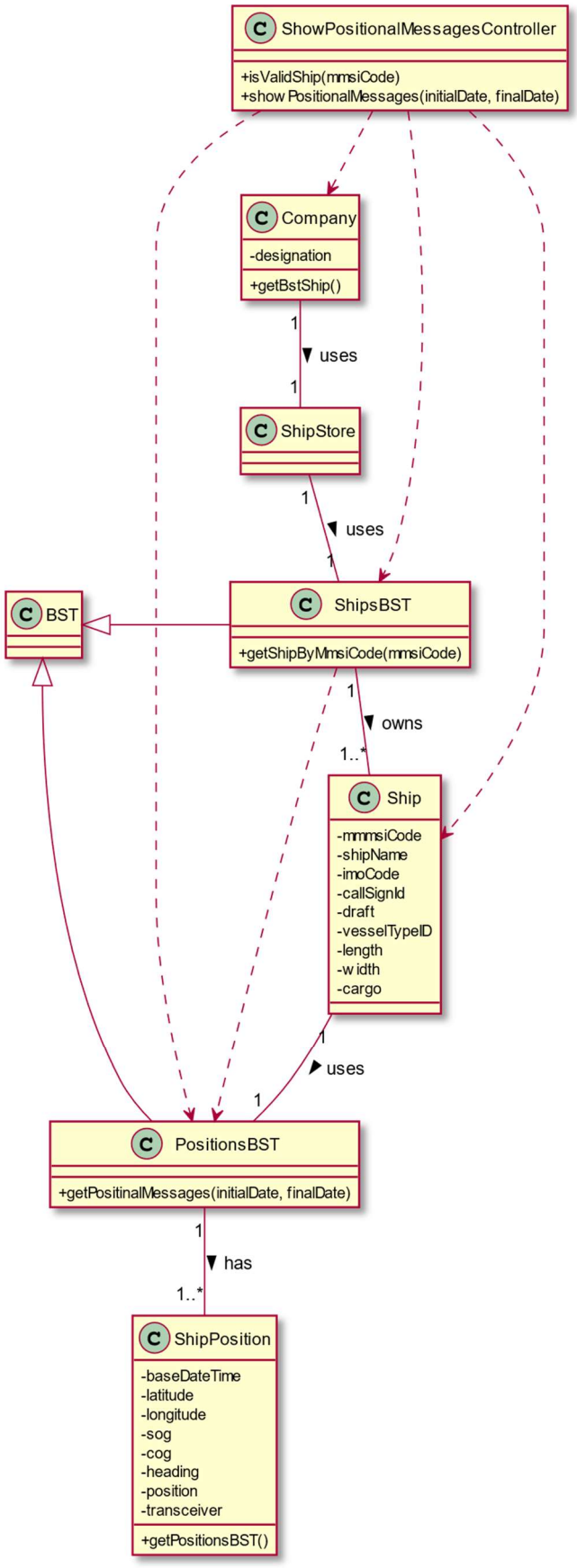
In order to meet the **acceptance criteria** of the US, the use of a binary search tree temporally organized (that is, according to the base date time of the positional messages) was used and therefore the PositionsBST class was created. That class is a Binary Search Tree which contains all the ship positions represented by the class ShipPosition.

As represented in the Class Diagram, it inherits all the methods from a generic BST class.

The responsibility to obtain the wished positional messages through the code `getPositionalMessages(initialDate, finalDate)` was initially assigned to Ship, because, according to the Information Expert Pattern, this is the class which knows the tree containing the ship positions. However, that responsibility was then delegated to PositionsBST.

In order to reduce responsibilities assigned to Company, the method to obtain the ship selected by the user – getShipByMmsiCode(mmsiCode)- was assigned to the class ShipsBST – the Binary Search Tree containing all the ships in the system (also according to the IE pattern).

In short, the Class Diagram was designed to ensure the solicited requisites.


ShowPositionalMessagesController
+isValidShip(mmsiCode)
+show PositionalMessages(initialDate, finalDate)
Company
-designation
+getBstShip()
1
uses
1
ShipStore
1
uses
1
BST
ShipsBST
+getShipByMmsiCode(mmsiCode)
1
owns
1..*
Ship
-mmmsiCode
-shipName
-imoCode
-callSignId
-draft
-vesselTypeID
-length
-w idth
-cargo
1
uses
1
PositionsBST
+getPositinalMessages(initialDate, finalDate)
1
has
1..*
ShipPosition
-baseDateTime
-latitude
-longitude
-sog
-cog
-heading
-position
-transceiver
+getPositionsBST()

3.3 Complexity Analysis

3.3.1 Code and Type Of Algorithm

```java
public List<String> getPositionalMessages(Date initialDate, Date finalDate) {
    List<String> listPositionalMessages = new ArrayList<>();

    getPositionalMessages(root, listPositionalMessages, initialDate, finalDate);

    return listPositionalMessages;
}
```

```java
private void getPositionalMessages(Node<ShipPosition> node,
                                   List<String> listPositionalMessages,
                                   Date initialDate,
                                   Date finalDate) {

    if(node == null)
        return;

    getPositionalMessages(node.getLeft(), listPositionalMessages, initialDate, finalDate);

    Date currentBaseDateTime = node.getElement().getBaseDateTime();

    if( !(currentBaseDateTime.before(initialDate) || currentBaseDateTime.after(finalDate)) ) {
        listPositionalMessages.add(node.getElement().toString());
    }

    getPositionalMessages(node.getRight(), listPositionalMessages, initialDate, finalDate);

}
```

The main functionality's algorithm is **non deterministic**, because its complexity can vary:

- It depends on the received data on parameters
- The information of the positional messages are organized temporally **in a Binary Search Tree**
- This Binary Search Tree is called **PositionsBST** which inherits all the methods from a generic BST class.

3.3.1.1 Best Case

The best case scenario is presented in this code fragment:

```
if(node == null)
    return;
```

- As the first element to be tested is the root of the tree, we should test first **if the root is null** and if so, it stops the method.
- **Time Complexity:** O(1)

3.3.1.2 Worst Case

The worst case scenario would be **all the base date times of the ship's positional messages falling between the period of dates or date** chosen by the user.

The private method `getPositionalMessages` goes through each node of the tree according to the **in Order** traversal. Therefore, as it's the worst case scenario, the tree is totally unbalanced and the height of the binary search tree becomes n. The time complexity is O(n).

Then, each position need to be added to the list containing the wished positional messages after the if clause (which is an ArrayList) – `listPositionalMessages`. The time complexity for adding a position is O(1).

Therefore, the time complexity of the method is O(n).

IV. Summary of Ship's Movements

4.1 Problem and justification

US-104 - As a traffic manager I which to make a Summary of a ship's movements.

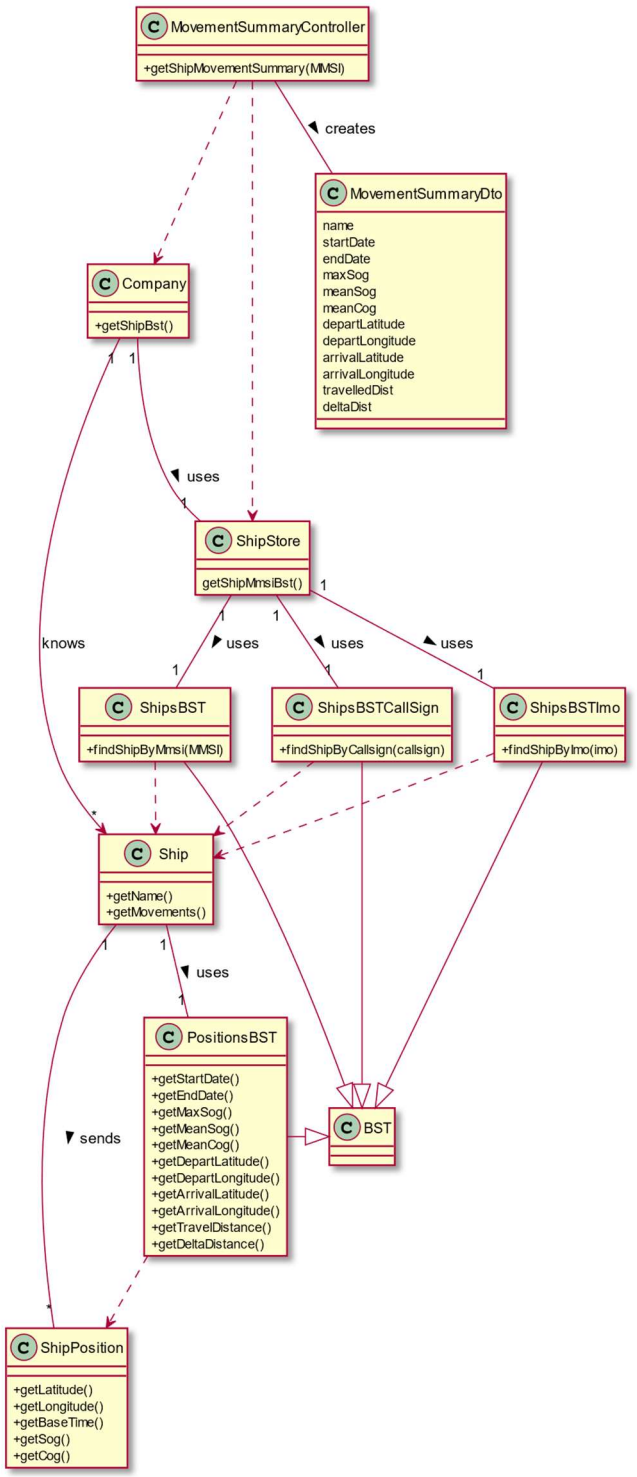
- For a given ship return in an appropriate structure one of its codes (MMSI, IMO or Call Sign), Vessel Name, Start Base Date Time, End Base Date Time, Total Movement Time, Total Number of Movements, Max SOG, Mean SOG, Max COG, Mean COG, Departure Latitude, Departure Longitude, Arrival Latitude, Arrival Longitude, Travelled Distance (incremental sum of the distance between each positioning message) and Delta Distance (linear distance between the coordinates of the first and last move.

- **Contributes:**
    - **Code, tests and complexity analysis:**
        - João Wolff, 1200049

4.2 Class Diagram Analysis

In order to find a ship and make a summary of its movements, the method developed in US 102 was used to get the ship by any of its codes, followed by the insertion of the wanted data inside a data transfer object of the ship summary, retrieving the positional data of the ship from the positions BST that composes the ship.

The separation of responsibilities, classes and its methods can be seen in the following class diagram:



The main functionality of this task happens to be beyond the search of the ship, already analyzed in US102, is the retrieved data from the PositionsBST which is the binary search tree that contains all the existent positions of a given ship. This tree is organized based on the base date time of each position object, making it easy to retrieve data from the arrival and depart positions, and also getting the positions in order in an efficient way.

4.3 Complexity Analysis

One of the main methods of the user story is the following:

```
public MovementsSummaryDto getShipMovementsSummary(String code){
    ShipStore shipStore = this.company.getShipStore();
    Ship currentShip = shipStore.getShipByAnyCode(code);
    PositionsBST shipMovements = currentShip.getPositionsBST();
    String name = currentShip.getVesselName();
    Date startDate = shipMovements.getStartDate();
    Date endDate = shipMovements.getEndDate();
    double maxSog = shipMovements.getMaxSog();
    double meanSog = shipMovements.getMeanSog();
    double meanCog = shipMovements.getMeanCog();
    double departLat = shipMovements.getDepartLatitude();
    double departLon = shipMovements.getDepartLongitude();
    double arrivalLat = shipMovements.getArrivalLatitude();
    double arrivalLon = shipMovements.getArrivalLongitude();
    double travDist = shipMovements.getTotalDistance();
    double deltaDist = shipMovements.getDeltaDistance();

    return new MovementsSummaryDto(name, startDate, endDate, maxSog, meanSog,
            meanCog, departLat, departLon, arrivalLat, arrivalLon, travDist, deltaDist);
}
```

As can be seen the method retrieves the ship by the search method developed in US102, which already has been analyzed in its section.

Following the next lines we have the methods:

- getPositionsBST(), getVesselName(), which are O(1), just retrieving the attributes
- then we have the getStartDate() and getEndDate() methods which have a time complexity of O(h), h being the height of the left and the height of the right subtree respectively
- regarding the methods getMaxSog(), getMeanSog(), getMeanCog() we have an complexity of O(N) mainly because to get the means or know the max value we must test all the data
- the arrival and departure coordinates methods follow the same O(h) of the dates, that is because the smallest and biggest items are still the way they are all found.
- about the distance methods:
    - first we have the total distance getter which also has an O(N) complexity being the sum of all the distance between positions;
    - now the delta distance have a time complexity of O(h) because it only needs to calculate the distance between the first and last position, going back to the smallest and biggest item methods.

The mentioned biggestElement and smallestElement methods can be seen in the following annexes respectively:

```
public ShipPosition biggestElement() { return biggestElement(root); }

protected ShipPosition biggestElement(Node<ShipPosition> node){
    if(node == null){
        return null;
    }
    if(node.getRight() == null){
        return node.getElement();
    }
    return biggestElement(node.getRight());
}
public E smallestElement() { return smallestElement(root); }

protected E smallestElement(Node<E> node){
    if(node == null){
        return null;
    }
    if(node.getLeft() == null){
        return node.getElement();
    }
    return smallestElement(node.getLeft());
}
```

In conclusion the final complexity of the method getShipMovementSummary() is O(max(functions)), resulting in O(N) in the most efficient way we could implement it.

V. US105 – List all Ships

5.1 Problem and justification

US105 - As a traffic manager I which to list for all ships the MMSI, the total number of movements, Travelled Distance and Delta Distance.

Acceptance criteria:

- Ordered by Travelled Distance and total number of movements (descending/ascending).

- **Contributes:**
  - **Code, tests and complexity analysis:**
    - Diogo Violante, 1201284

5.2 Class Diagram Analysis

The Class Diagram contains the relevant concepts from the Domain Model for the US103: Company, Ship, ShipPosition and VesselType.

Using good practices of OO software design, a Controller and UI Class were created – AllShipMMSIController and AllShipUI.

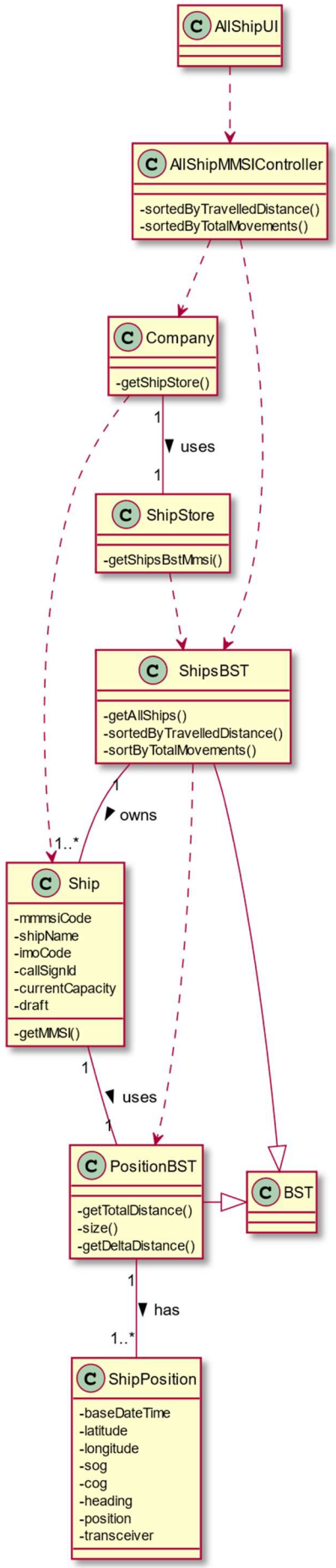
To correctly implement the requirements and meet the acceptance criteria of this US, we agreed in the use of binary search trees: ShipBST and PositionBST. The first one contains all Ships and their informations (every Node contains a Ship instance) and the second one all the Positions of a specific Ship where each Node has a ShipPosition instance.

As represented in the Class Diagram, it inherits all the methods from a generic BST class.

The two sorting algorithms required for this US: `sortedByTravelledDistance()` and `sortedByTotalMovements()`, were assigned to the ShipBST, due to this class knowing all the ships and their information, accordingly to the IE pattern.

`getAllShips()` was also delegated to ShipBST since this class contains all the ships to be analysed, reducing responsibilities of the class Company.

In short, the Class Diagram was designed to ensure the solicited requisites.


AllShipUI
AllShipMMSIController
-sortedByTravelledDistance()
-sortedByTotalMovements()
Company
-getShipStore()
1
uses
1
ShipStore
-getShipsBstMmsi()
ShipsBST
-getAllShips()
-sortedByTravelledDistance()
-sortByTotalMovements()
1
owns
1..*
Ship
-mmmsiCode
-shipName
-imoCode
-callSignId
-currentCapacity
-draft
-getMMSI()
1
uses
1
PositionBST
-getTotalDistance()
-size()
-getDeltaDistance()
BST
1
has
1..*
ShipPosition
-baseDateTime
-latitude
-longitude
-sog
-cog
-heading
-position
-transceiver

5.3 Complexity Analysis

5.3.1 Code and Type Of Algorithm

```
public Map<Integer, Set<Double>> sortedByTravelledDistance() {
    Map<Integer, Set<Double>> mapByTravelled = new LinkedHashMap<>();
    List<Ship> list = getAllShips();
    sortedByTravelledDistance(mapByTravelled, list);
    return mapByTravelled;
}
```

```
public void sortedByTravelledDistance(Map<Integer, Set<Double>> map, List<Ship> list) {
    ShipTravelledDistanceComparator comparator = new ShipTravelledDistanceComparator();
    Collections.sort(list, comparator);

    for (Ship x : list) {
        if (!map.containsKey(x.getMMSI())) {
            Set<Double> setter = new LinkedHashSet<>();
            setter.add(x.getPositionsBST().getDeltaDistance());
            setter.add(x.getPositionsBST().getTotalDistance());
            setter.add((double) x.getPositionsBST().size());
            map.put(x.getMMSI(), setter);
        }
    }
}
```

```
public Map<Integer, Set<Double>> sortedByTotalMovements() {
    Map<Integer, Set<Double>> mapByMovements = new LinkedHashMap<>();
    List<Ship> list = getAllShips();
    sortedByTotalMovements(mapByMovements, list);
    return mapByMovements;
}
```

```
public void  sortedByTotalMovements(Map<Integer, Set<Double>> map, List<Ship> list) {
    ShipDeltaDistanceComparato comparator = new ShipDeltaDistanceComparato();
    Collections.sort(list, comparator);

    for (Ship x : list) {
        if (!map.containsKey(x.getMMSI())) {
            Set<Double> setter = new HashSet<>();
            setter.add(x.getPositionsBST().getDeltaDistance());
            setter.add(x.getPositionsBST().getTotalDistance());
            setter.add((double) x.getPositionsBST().size());
            map.put(x.getMMSI(), setter);
        }
    }
}
```

The main functionality's algorithm is **non deterministic**, because its complexity can vary:

It depends on the number of ships contained in the ShipBST.

ShipBST is a **Binary Search** Tree which inherits all the methos from a generic BST class.

5.3.1.1 Best Case

The best-case scenario is presented in this code fragment:

```
if(node == null)
    return;
```

Figura 4 - Best Case Code Fragment

As the first element to be tested is the root of the tree, we should test first **if the root is null** and if so, it stops the method.

Time Complexity: O(1)

5.3.1.2 Worst Case

The worst-case scenario would be the root was not null, then all the ships in ShipBST would be involved in the sorting algorithm.

The private method get`AllShips` goes through each node of the tree according to the **Pre Order** traversal. So, in a worst-case scenario, the height of the binary search tree is n, which has a time complexity of O(n). Every ship is then added to a ArrayList, allShip – time complexity is O(1). With that, **the time complexity of this method is O(n).**

The private methods sortedByTravelledDistance and sortedByTotalMovements contain a Collections.sort() with time complexity of O(nlogn), a simple for loop with time complexity of O(n), an if condition of LinkedHashMap with time complexity of O(n), all the needed information is added to a LinkedHashSet with time complexity of O(1) and that information is put on a LinkedHashMap with time complexity of O(n). Therefore, **the time complexity of both methods is O(nlogn).**

VI. US106 – Get Top-N Ships

6.1 Problem and justification

US106 - Get the top-N ships with the most kilometres travelled and their average speed (MeanSOG).

Acceptance criteria:

- in a period (initial/final Base Date Time) grouped by Vessel Type.

- **Contributes:**
  - **Tests and complexity analysis:**
    - Diogo Violante, 101284
  - **Code:**
    - Diogo Violante, 1201284 -85%
    - João Wolff, 1200049 – 15%

    (some code was reused from US104)

6.2 Class Diagram Analysis

The Class Diagram contains the relevant concepts from the Domain Model for the US106: Company, Ship, ShipPosition and VesselType.

Using good practices of OO software design, a Controller and UI Class were created – TopNController and TopNUI.

To correctly implement the requirements and meet the acceptance criteria of this US, we agreed in the use of binary search trees: ShipBST and PositionBST. The first one contains all Ships and their informations (every Node contains a Ship instance) and the second one all the Positions of a specific Ship where each Node has a ShipPosition instance.

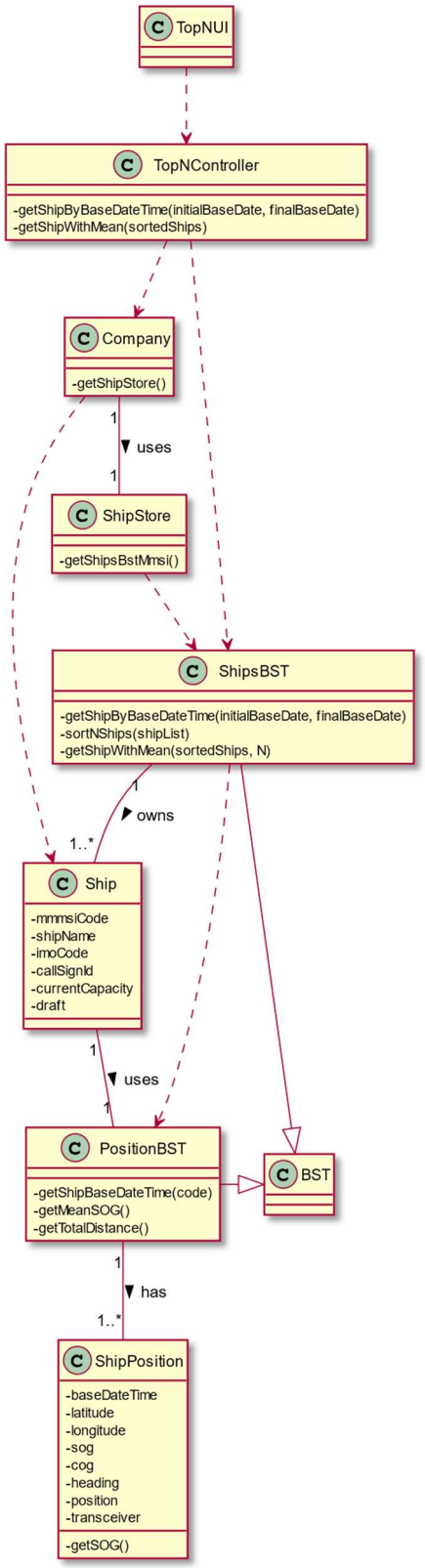
As represented in the Class Diagram, it inherits all the methods from a generic BST class.

The main method of this functionality is `getShipWithMean()`, which returns a map with a map associated with each vesselTypeId, where the second map has each ship associated with a List with their Travelled Distance, Delta Distance and Total Movements.

This method was assigned to the ShipBST, due to this class knowing all the ships and their information, accordingly to the IE pattern.

`getShipsByDate(initialBaseDate, finalBaseDate)` was also delegated to ShipBST since this class contains all the ships to be analysed.

In short, the Class Diagram was designed to ensure the solicited requisites.



6.3 Complexity Analysis

6.3.1 Code and Type Of Algorithm

```
public List<Ship> getShipsByDate(Date initialDate, Date finalDate) {

    List<Ship> shipList = new ArrayList<>();
    getShipsByDate(root, initialDate, finalDate, shipList);
    return shipList;
}
```

```
private void getShipsByDate(Node<Ship> node, Date initialDate, Date finalDate, List<Ship> shipList) {
    if(node==null) return;

    if (!shipList.contains(node.getElement())){
    if(node.getElement().getPositionsBST().getShipDate(node.getElement().getMMSI()).after(initialDate)
            && node.getElement().getPositionsBST().getShipDate(node.getElement().getMMSI()).before(finalDate)){

            shipList.add(node.getElement());

        }
    }
    getShipsByDate(node.getLeft(), initialDate, finalDate, shipList);
    getShipsByDate(node.getRight(), initialDate, finalDate, shipList);

}
```

```
public Map<Integer, Map<Ship, Set<Double>>> getShipWithMean(List<Ship> listShip, int number) {
    Map<Integer,  Map<Ship, Set<Double>>> map = new HashMap<>();
    Map<Ship, Set<Double>> shipMap ;
    Set<Double> setter ;
    Integer vessel = null;

    listShip = sortNShips(listShip);

    for (Ship x: listShip) {
        if (!map.containsKey(x.getVesselTypeID())){
            shipMap = new HashMap<>();
            vessel = x.getVesselTypeID();
            setter = new HashSet<>();
            setter.add(x.getPositionsBST().getTotalDistance());
            setter.add(x.getPositionsBST().getMeanSog());
            if (shipMap.size() <= number) {
                shipMap.put(x, setter);
                map.put(vessel, shipMap);
            }
        }
    }

    return map;


}
```

The main functionalities' algorithm are **non deterministic**, because their complexity can vary:

It depends on the number of ships contained in the ShipBST and the Base Date Time gap.

ShipBST is a **Binary Search** Tree which inherits all the methos from a generic BST class.

6.3.1.1 Best Case

The best-case scenario is presented in this code fragment:

```
if(node == null)
    return;
```

- As the first element to be tested is the root of the tree, we should test first **if the root is null** and if so, it stops the method.
- **Time Complexity:** O(1)

6.3.1.2 Worst Case

The worst-case scenario would be the root was not null and all th ships would be in the Base Date Time gap.

The private method `getShipsByDate(initialBaseDate, finalBaseDate)` goes through each node of the tree according to the **Pre Order** traversal. So, in a worst-case scenario, the height of the binary search tree is n, which has a time complexity of O(n). Every ship is then added to a ArrayList, shipList – time complexity is O(1). With that, **the time complexity of this method is O(n).**

`getShipWithMean()` contains a simple for loop with time complexity of O(n), an if condition of HashMap with time complexity of O(n), each ship's Travelled Distance and MeanSOG is added to a HashSet with time complexity of O(1). The HashSet is associated with each ship and put in a HashMap with time complexity of O(n) and that information is put on a HashMap associated with the Ship's MMSI with time complexity of O(n). Therefore, **the time complexity of both methods is O(n).**

VII. US107 – Show Pairs Of Ships

7.1 Problem and justification

US107 - As a traffic manager, I wish to have the pairs of ships with routes with close departure/arrival coordinates (no more than 5 Km away) and with different Travelled Distance.

- **Acceptance criteria:**
  - Sorted by the MMSI code of the 1st ship and in descending order of Travelled Distance difference.
  - Do not consider ships with Travelled Distance less than 10 kms.
- **Contributes:**
  - **Tests and complexity analysis:**
    - Ana Albergaria, 1201518
  - **Code:**
    - Ana Albergaria, 1201518 – 85%
    - João Wolff, 1200049 – 15%

      (some methods from the US104 were reused)

7.2 Class Diagram Analysis

The Class Diagram contains the relevant concepts from the Domain Model for the US107: Company, Ship and ShipPosition.

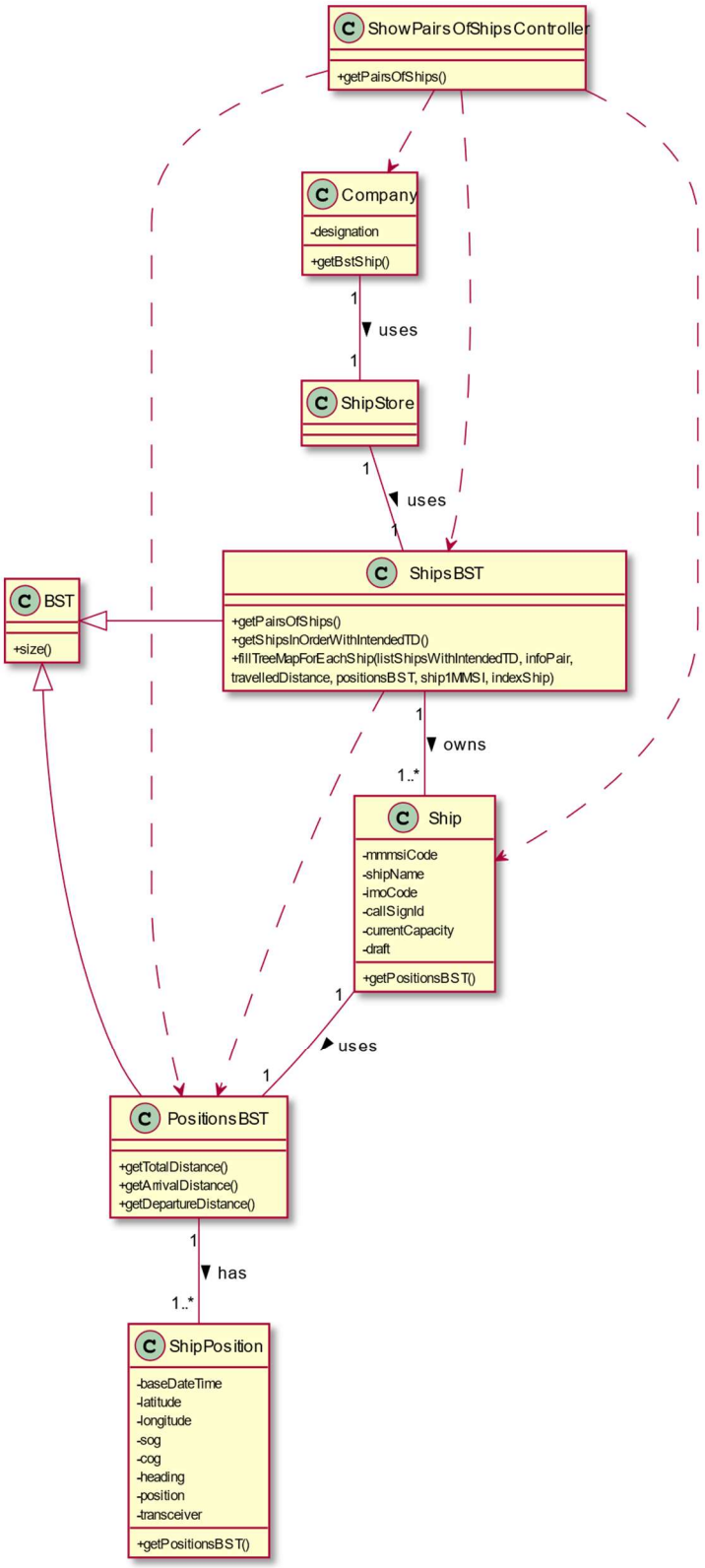
Using good practices of OO software design, a Controller Class was created – ShowPairsOfShipsController.

In order to meet the acceptance criteria of the US, **the ships are organized by the MMSI code in a Binary Search Tree called ShipsBST. That way,** we can obtain the pairs of ships sorted by the MMSI of the 1st ship in an efficient way**.** That class is a Binary Search Tree which contains all the ship positions represented by the class Ship **and inherits all the methods from a generic BST class**.

The responsibility of obtaining the pairs of ships through the method `getPairsOfShips` is assigned to ShipsBST class, because, according to the IE pattern, it's the class who knows all the ship objects. For the same reason, this class also contains the `getShipsInOrderWithIntendedTD` and `fillTreeMapForEachShip`, methods which are called by the `getPairsOfShips` method.

Using the same logic of thought, the class PositionsBST, which is a Binary Search Tree containing the positions of a Ship object, contains the methods to obtain information about the positions: `getTotalDistance()`, `getArrivalDistance` and `getDepartureDistance`.

When we want to access the ShipsBST class, the Controller tells Company to get that Class, hence the `getBstShip()`. The same happens with Ship with its positions – to obtain the PositionsBST of a certain Ship, we call its `getPositionsBST()` method, respecting, thus, the domain model: Ship has ShipPosition.



7.3 Complexity Analysis

7.3.1 Code and Type Of Algorithm

The main functionality's algorithm is **non deterministic**, because its complexity can vary:

- It depends on the received data on parameters
- The ships are organized by the MMSI code **in a Binary Search Tree**
- The Ship Positions are organized by the Base Date Time **in a Binary Search Tree**
- This Binary Search Trees are called **ShipsBST** and **PositionsBST** which inherits all the methods from a generic BST class.

```
public List<TreeMap<Double, String>> getPairsOfShips() {
    List<TreeMap<Double, String>> listPairsOfShips = new ArrayList<>();

    List<Ship> listShipsWithIntendedTD = (List<Ship>) getShipsInOrderWithIntendedTD(); //O(n)

    for (Ship ship : listShipsWithIntendedTD) { //O(n)
        TreeMap<Double, String> infoPair = new TreeMap<>(Collections.reverseOrder());

        PositionsBST positionsBST = ship.getPositionsBST();
        Double travelledDistance = positionsBST.getTotalDistance(); //O(n)
        int indexShip = listShipsWithIntendedTD.indexOf(ship);

        fillTreeMapForEachShip(listShipsWithIntendedTD, infoPair, travelledDistance, positionsBST, ship.getMMSI(), indexShip);

        if(!infoPair.isEmpty())
            listPairsOfShips.add(infoPair);
    }
    return listPairsOfShips;
}
```

```java
public void fillTreeMapForEachShip(List<Ship> listShipsWithIntendedTD,
                                   TreeMap<Double, String> infoPair,
                                   Double travelledDistance,
                                   PositionsBST positionsBST,
                                   int ship1MMSI, int indexShip) {

    for (int j = indexShip+1; j < listShipsWithIntendedTD.size(); j++) {

        Ship ship2 = listShipsWithIntendedTD.get(j);

        PositionsBST positionsBST2 = ship2.getPositionsBST();
        Double travelledDistance2 = positionsBST2.getTotalDistance(); //O(n)

        if(!Objects.equals(travelledDistance, travelledDistance2)) {
            Double arrivalDistance = positionsBST.getArrivalDistance(positionsBST2); //O(h)

            if(arrivalDistance <= Constants.LIMIT_COORDINATES) {
                Double depDistance = positionsBST.getDepartureDistance(positionsBST2); //O(h)

                if(depDistance <= Constants.LIMIT_COORDINATES) {
                    int numMovs = positionsBST.size()-1, numMovs2 = positionsBST2.size()-1; //O(n)
                    double diffTravDist = Math.abs(travelledDistance - travelledDistance2);
                    String allInfo = String.format("%-15d%-15d%-15.3f%-15.3f%-15d%-15.3f%-15d%-15.3f%n",
                            ship1MMSI, ship2.getMMSI(), arrivalDistance, depDistance, numMovs, travelledDistance, numMovs2, travelledDistance2);
                    infoPair.put(diffTravDist, allInfo);
                }
            }
        }
    }
}
```

In order to obtain the intended pairs of ships, first the method `getShipsInOrderWithIntended` is called. This method:

- Goes through the nodes of the Ship Positions' Binary Search Tree through the in Order traversal so that we obtain the 1st Ship ordered by the MMSI code.
- Verifies **which ships on the tree have a Travelled Distance equal or greater than 10** and if so, adds them to the list `listShipsWithIntendedTD`.

7.3.1.1 Best Case

The best case scenario is presented in this code fragment present in `getShipsInOrderWithIntendedTD`:

```java
if(node == null)
    return;
```

- As the first element to be tested is the root of the tree, we should test first **if the root is null** and if so, it stops the method.
- **Time Complexity:** O(1)

7.3.1.2 Worst Case

- **Time Complexity of** `getShipsWithIntendedTD`: O(n)

After obtaining the lists of ships with Travelled Distance greater or equal to 10, a loop going through all the elements of the list is implemented.

- **Time Complexity of the loop:** O(n)

Then, the Positions and the Travelled Distance and index of the current ship of the list are obtained (see Time Complexities in figure).

The method `fillTreeMapForEachShip` is required to store the information about the current ship of the `listShipsWithIntendedTD` with all the others ships left to be compared to. Hence why this loop starts at the index of the Ship + 1.

Time Complexity of the loop:

- The variable `j` executes (n-1) + (n-2) + (n-3) … (n-i), until (n-i) reaches 0. This is the equivalent of:

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = \frac{n^2 + 1}{2}$$

- Therefore, the Time Complexity is **$O(n^2)$.**

In order to feel the TreeMap with the information, three conditions should be met:

- The Travelled Distances between the ship and ship2 are different;
  - **Time Complexity to get Travelled Distance:** O(n)
- The Arrival Distance between them is less or equal to 5 km;
  - **Time Complexity to get Arrival Distance:** O(h)
- The Departure Distance between them is less or equal to 5 km.
  - **Time Complexity to get Departure Distance:** O(h)

Observation: The Time Complexity of this methods was already explained previously in the US104 Complexity Analysis.

If these conditions are met, then the information about the current ship and ship2 are stored, using put method.

- **Time Complexity:** O(1)

All of the other lines are of constant complexity as well.

Why a TreeMap?

- In order to obtain **the pairs of ships in descending order of the Travelled Distance difference** – the key.

Finally, if the TreeMap isn't empty, it will store in a list containing a list of TreeMaps – `getPairsOfShips()`

As the Time Complexity is obtained through O(max(functions)), it results in O(n^2).