ISEP / LEI
ESINF
2021/2022
G76
Autores – Jorge Ferreira (1201564) Rafael Leite (1201566) Rui Pina (1201568)

RELATÓRIO ESINF

 # Abstract #

Uma empresa de logística de transporte requer um Sistema de software capaz de gerir as suas logísticas. Esta empresa opera tanto no mar, como em terra, por vários continentes e possui diversos armazéns espalhados pelo globo.

 # Introdução #

A finalidade deste relatório é de documentar o desenvolvimento das User Stories, no âmbito de ESINF. Todos os passos percorridos para a elaboração de uma solução para os problemas apresentados, assim como a complexidade temporal de cada algoritmo adotado.

De acordo com as boas práticas aprendidas em ESOFT, no semestre passado foi adotado uma metodologia ágil de Scrum e as práticas de Programação Orientada a Objetos.

Com isto as UserStories foram divididas de maneira justa por todos os membros da equipa.

 # Responsáveis das User Stories #

US 101 – Rui Pina 1201568

US 102 – Rui Pina 1201568

US 103 – Rafael Leite 1201566

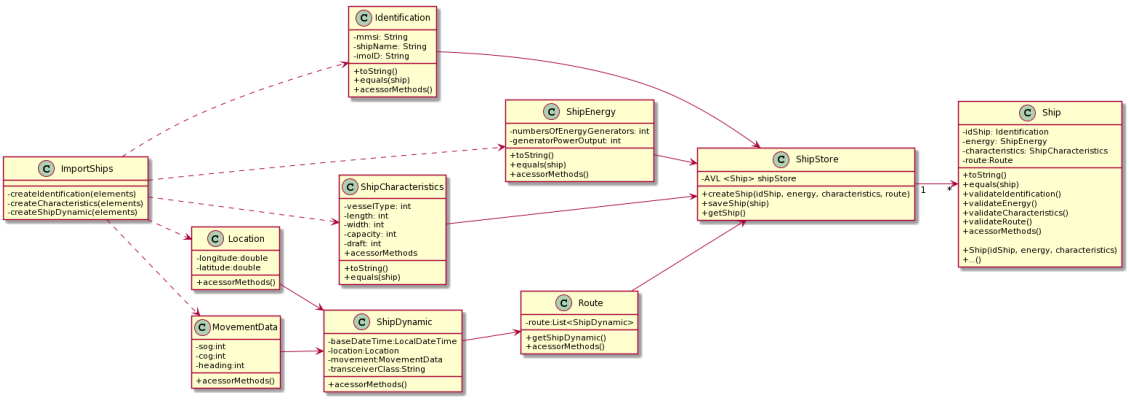
US 104 – Jorge Ferreira 1201564

US 106 – Jorge Ferreira 1201564

US 107 – Rafael Leite 1201566

- **US101**

Class Diagram



- Em conformidade com as boas práticas adquiridas em ESOFT, nós decidimos usar o padrão Store e, por esse motivo criamos a classe “ShipStore” que contem uma instância de uma árvore AVL, que nos permite procurar os detalhes dos navios com grande eficiência.

- Na Ship Store nós iremos precisar guardar instâncias de navios que serão importados por um csv, consequentemente nós criamos a class “Ship”. Dado que, é uma class bastante pesada em termos do número de atributos, nós dividimos-a em diferentes classes com apenas uma função, de acordo com o padrão GRASP.

-Na classe Identificação haverá os atributos mmsi, shipName, imoID e searchCode, que como o nome da classe implica, se referem à identificação de cada navio.

- Na classe ShipCharacteristics haverá os atributos vesselType, length, width, capacity e draft.

- Na classe Location haverá os atributos latitude e longitude.

- Na classe Movement haverá os atributos sog, cog, heading.

- Por fim, na classe ShipDynamic haverá baseDateTime, location, movement, cargo, transceiverClass.

- Route é uma lista de ShipDynamic.

Algoritmo 1 ImportShips

Este **Algoritmo** foi dividido em 3 partes lógicas.

```
public static List<Ship> importShips(String fileName) {
    List<Ship> ships = new ArrayList<>();
    Identification idShip;
    ShipCharacteristics characteristics;
    ShipDynamic shipDynamic;
    String line;
    String splitBy = ",";
    BufferedReader br = null;
    Ship ship = null;
    Route route = null;
```

Este pedaço de código apenas inicializa variavéis que serão úteis para o algoritmo em si. Com isto conseguimos ver que cada uma destas linhas têm complexidade temporal de O(1), porque apenas serão executados uma vez.

```
try {
    br = new BufferedReader(new FileReader(fileName));
    br.readLine();
    while ((line = br.readLine()) != null) {
        size++;
        String[] elements = line.split(splitBy);
        if (ship == null || !ship.getShipId().getMmsi().equals(elements[0])) {

            try {
                route = new Route();
                idShip = createIdentification(elements);
                characteristics = createCharacteristics(elements);
                ship = new Ship(idShip, characteristics, route: null);
                ships.add(ship);
            } catch (Exception e) {
                ship = null;
                LOGGER.log(Level.INFO, String.format("Failed to import line %d", size));
            }
        }
```

Neste pedaço de Código, conseguimos ver que as primeiras 2 linhas são excutadas apenas uma vez, já que não estão dentro de um loop. Após isso todas as outras linhas irão ser executadas n vezes já que estão dentro do while loop.

```
    if (ship != null) {
        try {
            shipDynamic = createShipDynamic(elements);
            route.add(shipDynamic);
            ship.setRoute(route);
        } catch (Exception e) {
            LOGGER.log(Level.INFO, String.format("Failed to import line %d", size));
        }
    }
}
```

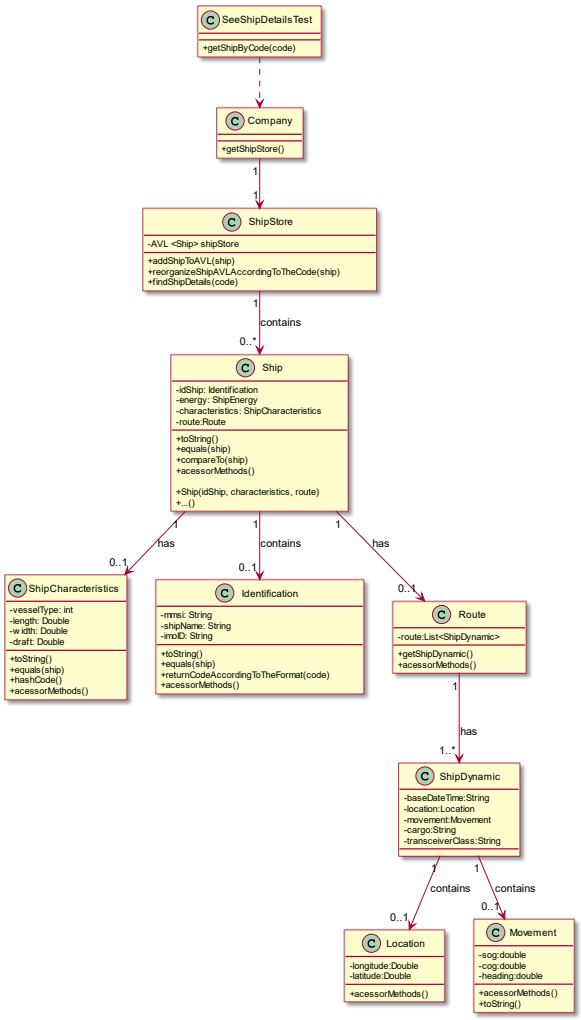
Apesar de não ser possível ver aqui. Continuamos ainda dentro do mesmo ciclo e por esse motivo, estas linhas de código irão correr no pior caso n vezes.

```
for (Ship ship1 : ships) {
    store.addShipToAVL(ship1);
}
```

Por fim adicionamos todos os ships que foram criados à AVL. O que terá uma complexidade termporal de O(nlogn) já que percorremos os ships todos O(n) e O(logn) para inseri-los na AVL.

Como isso temos **O(1)** do **primeiro** de pedaço de código **O(n)** para o **segundo** , **O(n)** para **terceiro** **O(nlogn)** para o ultimo. Logo O(1+ n + n + nlogn) ⇔O(nlogn).

Complexidade temporal: O(nlogn)



US102

- **Class Diagram**

- Nesta User Story nós usamos estas classes em conformidade com o que foi referido na US101.

- Além disso foi adicionado alguns métodos referentes à US102, nomeadamente o método returnCodeAccordingToTheFormat na classe Identification e a findShipDetails e reorganizeAVLAccordingToTheCode da AVL.

Análise da complexidade temporal US102

```
public Ship findShipDetails(String code) {
    BST.Node<Ship> s;
    AVL<Ship> shipAVL;
    Ship ship = null;
    ArrayList<Ship> ship2 = new ArrayList<>();
    for (Ship ships : store.posOrder()) {
        ships.getShipId().setSearchCode(code);
        ship2.add(ships);
    }
    shipAVL = reorganizeShipAVLAccordingToTheCode(ship2);
```

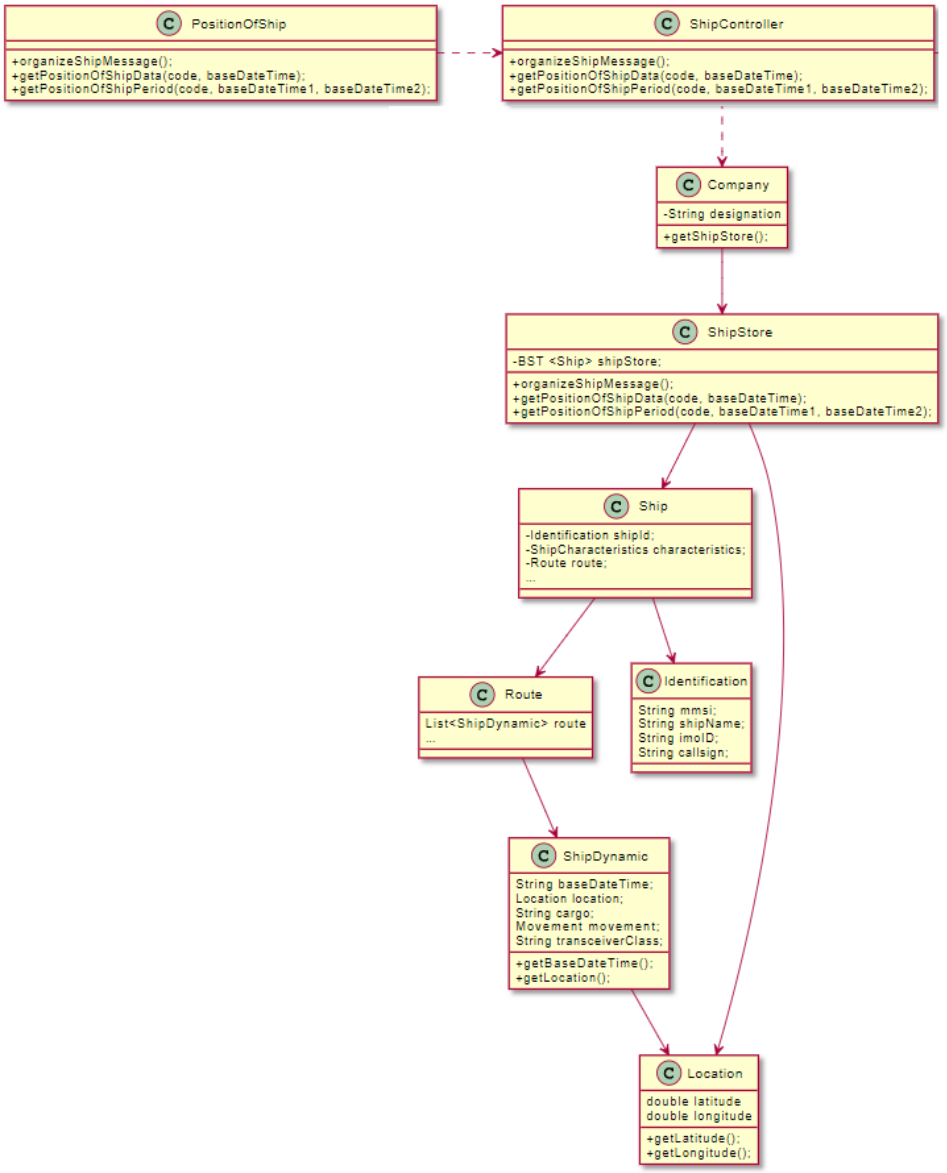
No início nós inicializamos todas as variáveis que serão usados pelo método e após isso iremos percorrer os ships para reorganizá-los consoante o tipo de código de pesquisa que foi utilizado. Quer este seja o IMO, MMSI ou o Callsign. Daí tiramos que temos O(nlogn) já que temos que inserir n navios numa estrutura em que o algoritmo de inserção tem a complexidade de O(logn) daí temos que a complexidade temporal da inserção de n elementos numa AVL é O(nlogn).

```
    for (Ship value : shipAVL.posOrder()) {
        s = shipAVL.find(value);
        if (s.getElement().getShipId().getImoID().equals(code) || s.getElement().getShipId().getCallsign().equals(code)
                || s.getElement().getShipId().getMmsi().equals(code)) {
            ship = s.getElement();
        }
    }
    if (ship == null) {
        throw new IllegalArgumentException();
    }
    return ship;
```

Com o AVL reorganizado nós passamos percorremos por ele e procuramos o navio que possui um dos atributos de identificação iguais ao do código que tamos à procura. Se o encontrarmos retornamos esse navio, se não retornamos uma exceção para tratarmos no momento de 'UI'.

Com tudo isto temos que O(nlogn + n) = O(nlogn).

Complexidade temporal: O(nlogn)



- **US103**

Class Diagram

A classe *PositionOfShip* não foi necessária implementar, pois não era obrigatório implementarmos uma UI, já que podemos demonstrar a aplicação com recurso a testes.

Depois temos interligados *ShipController*, *Company* e *ShipStore*, em conformidade com o que aprendemos em ESOFT. A *ShipStore* contem uma AVL<Ship> com todos os ships e métodos necessários ao funcionamento da US.

O método organizeShipMessage, que tem ligação com as classes *Ship*, *Route* e *ShipDynamics*, organiza para cada navio as suas mensagens temporais pelo seu baseDateTime, atributo da classe *ShipDynamics*.

A classe *Route* consiste numa lista de *ShipDynamics* porque um navio pode ter várias rotas. *ShipDynamics* contêm a informação dinâmica dos navios, mais concretamente o conteúdo das mensagens temporais.

O método getPositionOfShipData retorna uma posição (*Location*) de um navio identificado pelo seu MMSI (*Identification*) numa determinada data. Já o método getPositionOfShipPeriod retorna as posições de um navio num determinado período (data inicial e data final).

Este diagrama de classes permite que a classe Ship não fique sobrecarregada com atributos, diminuindo assim a complexidade da mesma devido a estes estarem distribuídos em outras classes. As classes têm nomes explícitos e intuitivos.

```
public void organizeShipMessage() {
    Map<Integer, List<Ship>> shipsByLevel = store.nodesByLevel();
    for (Map.Entry<Integer, List<Ship>> entry : shipsByLevel.entrySet()) {
        for (Ship ship : entry.getValue()) {
            ship.getRoute().getRoute().sort(Comparator.comparing(ShipDynamic::getBaseDateTime));
        }
    }
        final String fileToBeWrittenTo = "shipsOrganized.txt";
        try {
            PrintToFile.print(store.inOrder().toString(), fileToBeWrittenTo);
        } catch (IllegalArgumentException | IOException e) {
            System.out.println("Error");
        }
    }
```

Methods

- **OrganizeShipMessage**

Este método, que está inserido na class TestStore, organiza a **store** e envia para um ficheiro .txt, as mensagens temporais organizadas e associadas a cada navio.

store = AVL com todos os navios.

Complexidade Temporal: O(nlog n . n) = O (logn . n^2)

- **getPositionOfShipData**

```java
public Location getPositionOfShipData(String mMSI, String baseDateTime) {
    Location location = null;

    for (Ship ship : store.inOrder()) {
        for (int i = 0; i < ship.getRoute().getRoute().size(); i++)
            if (ship.getRoute().getRoute().get(i).getBaseDateTime().equals(baseDateTime)
                    && ship.getShipId().getMmsi().equals(mMSI)) {
                location = ship.getRoute().getRoute().get(i).getLocation();
            }
    }

    assert location != null;
    return location;
}
```

Este método retorna uma Location (posição) do navio, identificado pelo seu código mMsi, numa determinada data (baseDateTime).

```java
public List<Location> getPositionOfShipPeriod(String mMSI, String baseDateTime1, String baseDateTime2)
        throws ParseException {
    organizeShipMessage();
    SimpleDateFormat sdf = new SimpleDateFormat( pattern: "dd/MM/yyyy hh:mm");
    Date d1 = sdf.parse(baseDateTime1);
    Date d2 = sdf.parse(baseDateTime2);

    List<Location> position = new ArrayList<>();

    for (Ship ship : store.inOrder()) {
        for (int j = 0; j < ship.getRoute().getRoute().size(); j++)
            if (sdf.parse(ship.getRoute().getRoute().get(j).getBaseDateTime()).after(d1) &&
                    sdf.parse(ship.getRoute().getRoute().get(j).getBaseDateTime()).before(d2)
                    && ship.getShipId().getMmsi().equals(mMSI)) {
                position.add(ship.getRoute().getRoute().get(j).getLocation());
            }
    }
    return position;
}
```

Complexidade Temporal: O(n log n . n) = O(log n . n^2)

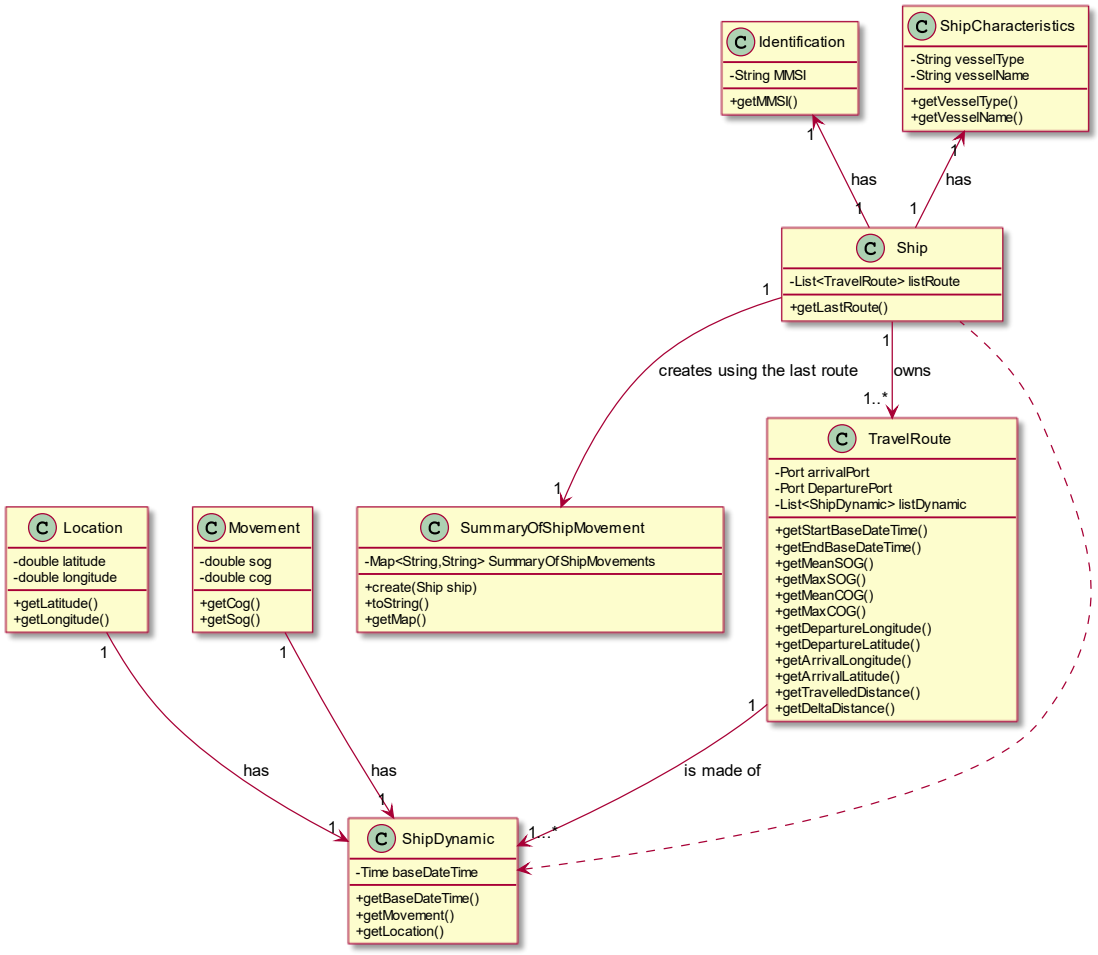
- **getPositionOfShipPeriod**

Este método retorna uma lista com todas as posições de um navio, identificado pelo seu código Mmsi, num determinado período (obtido pela data inicial e data final).

Complexidade Temporal: O(n.log n . n) = 0(log n . n^2)

US104

- **Class Diagram**



A US104, resumidamente requer a angariação de dados de um navio e juntá-los numa classe, que neste diagrama contem um mapa para criar a possibilidade de ter acesso a um dado específico.

Complexidade temporal

Como maior parte dos métodos são de arranjar dados a complexidade temporal simples, para além dos métodos get() simples existem os seguintes métodos com uma maior complexidade temporal

```java
public double getMaxSog(){
    double temp=0;
    for (ShipDynamic shipd: route) {
        if (shipd.getSog()>temp){
            temp=shipd.getSog();
        }
    }
    return temp;
}
public double getMaxCog(){
    double temp=0;
    for (ShipDynamic shipd: route) {
        if (shipd.getCog()>temp){
            temp=shipd.getCog();
        }
    }
    return temp;
}
public double getMeanSog(){
    double sum =0;
    for (ShipDynamic shipd: route) {
        sum+=shipd.getSog();

    }
    return sum/route.size();
}
public double getMeanCog(){
    double sum =0;
    for (ShipDynamic shipd: route) {
        sum+=shipd.getCog();

    }
    return sum/route.size();
```

Métodos

getMaxSog()

getMaxCog()

getMeanSog()

getMeanCog()

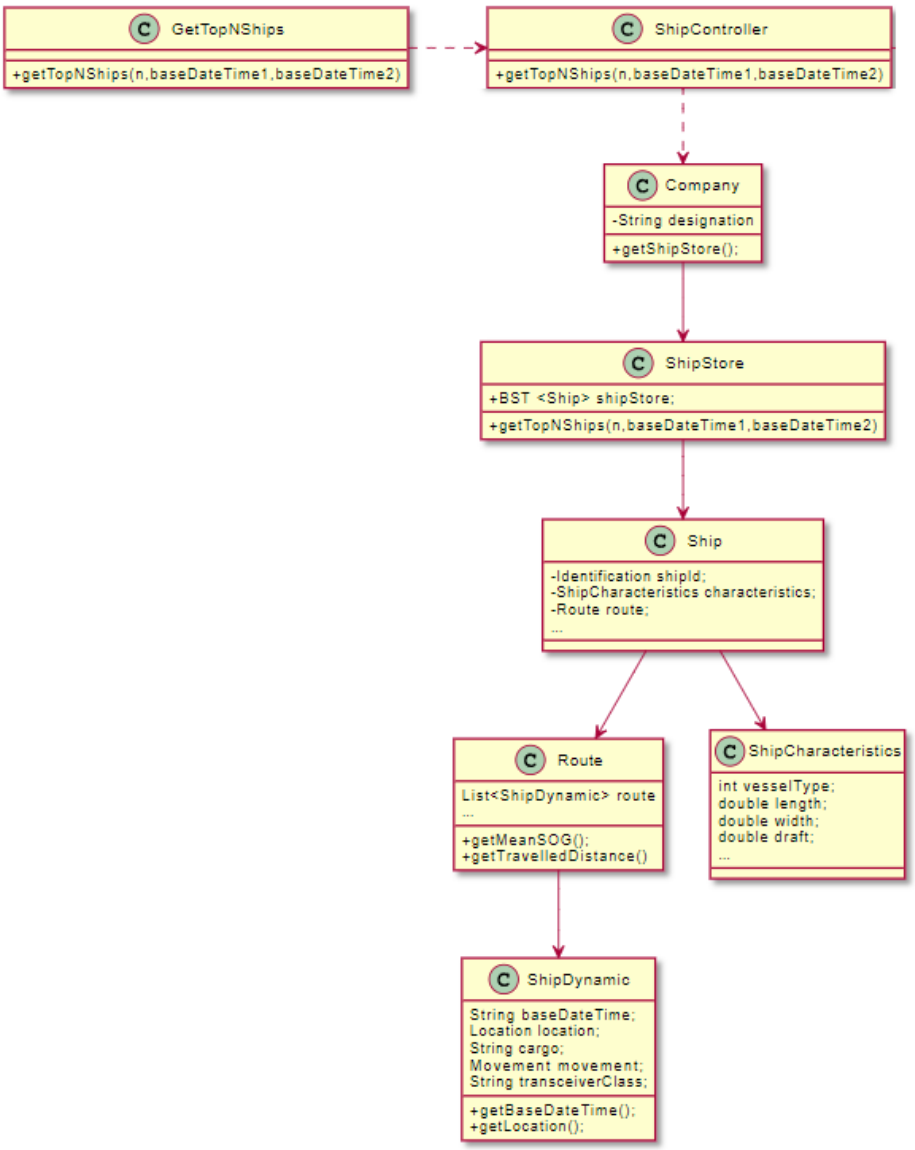
getTravelledDistance()

Todos estes métodos tem complexidade natural de O(n)

```java
public double getTravelledDistance(){
    double distanceSum=0;
    for (int i=0;i<route.size()-1;i++){
        String lat1=route.get(i).getLatitude();
        String long1=route.get(i).getLongitude();
        String lat2=route.get(i+1).getLatitude();
        String long2=route.get(i+1).getLongitude();
        distanceSum+=distance(Double.parseDouble(lat1),Double.parseDouble(long1),Double.parseDouble(lat2),Double.parseDouble(long2));
    }
    return distanceSum;
}
```

Os restantes métodos desta US têm complexidade temporal elementar.

US106



- **Class Diagram**

ShipController, *Company* e *ShipStore* estão interligados, em conformidade com o que aprendemos em ESOFT. A *ShipStore* contem uma AVL<Ship> com todos os ships.

O método getTopNShips da classe *TestStore*, retorna um Map com os top N navios (+km) para cada Vessel Type(*ShipCharacteristics*) e envia para um txt as distancia percorrida e o MeanSOG, obtidos apartir da classe *Route*, num dado período.

Este diagrama de classes permite que a classe *Ship* não fique sobrecarregada com atributos, diminuindo assim a complexidade da mesma devido a estes estarem distribuídos em

outras classes. A classe *Route* torna-se bastante útil, pois contêm todas as mensagens e assim consegue facilmente calcular o MeanSOG e distância percorrida.

Method

- **GetTopNShips**

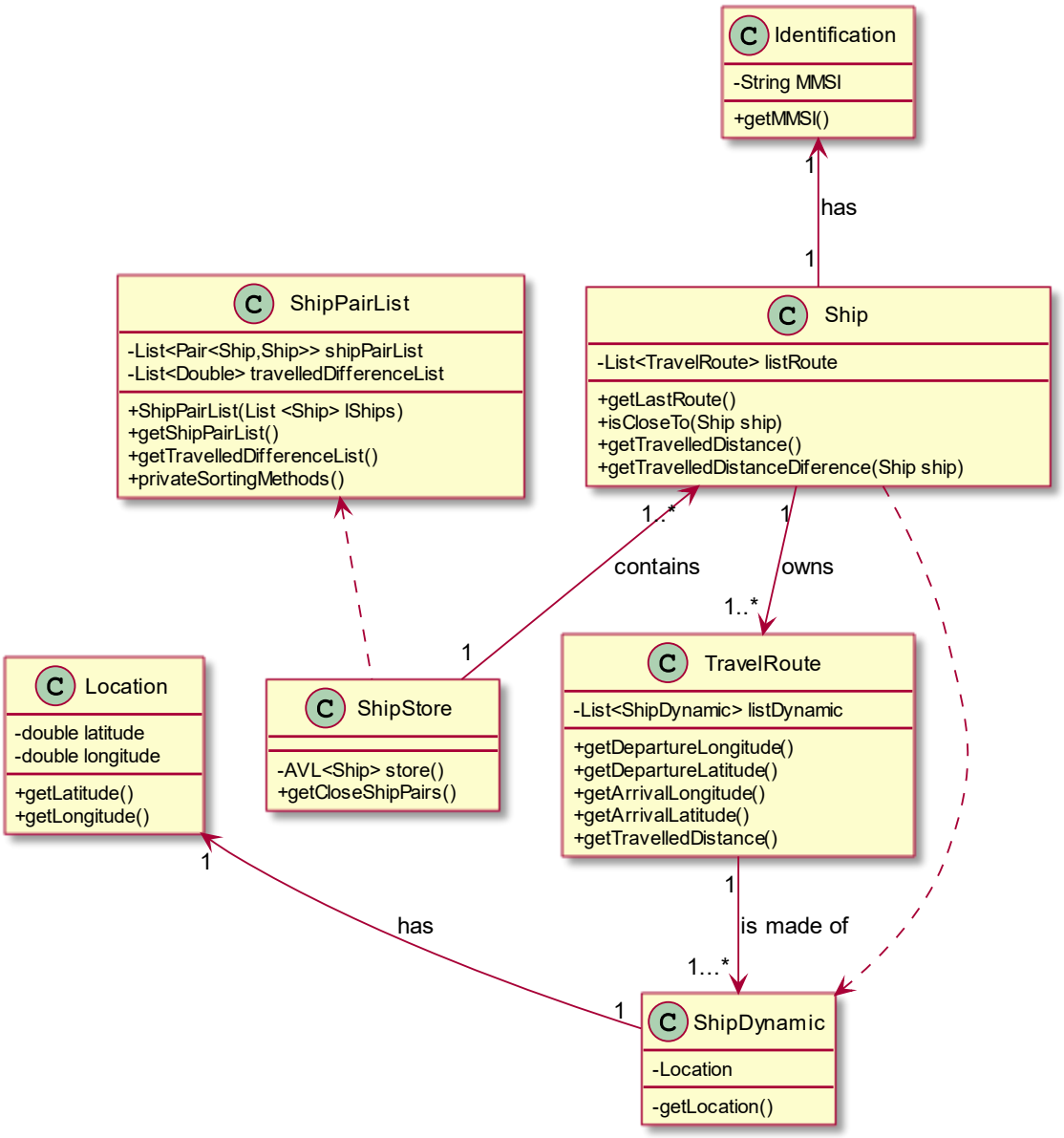
Este método retorna um Map<Integer, List <Ships>> (Key = Vessel Type) com os top N navios com maior distância percorrida para cada Vessel Type, num determinado período. Envia para um .txt toda a informação, juntamente com o MeanSOG, de forma a cumprir o pedido.

```java
public Map<Integer, List<Ship>> getTopNShips(int n, String start, String end) throws ParseException, IOException {
    SimpleDateFormat sdf = new SimpleDateFormat( pattern: "dd/MM/yyyy hh:mm");
    Date d1 = sdf.parse(start);
    Date d2 = sdf.parse(end);

    AVL<Ship> avl = store;
    organizeShipMessage();
    //remover rotas fora do período
    for (Ship ship : avl.inOrder()) {
        for (int j = 0; j < ship.getRoute().getRoute().size(); j++) {
            if (!(sdf.parse(ship.getRoute().getRoute().get(j).getBaseDateTime()).after(d1) &&
                    sdf.parse(ship.getRoute().getRoute().get(j).getBaseDateTime()).before(d2))) {
                ship.getRoute().getRoute();}}}
    //bst -> list
    List<Ship> ships = new ArrayList<>();
    for (Ship ship : avl.inOrder()) {
        ships.add(ship);
    }
    //ships ordenados por maior distancia percorrida
    ships.sort(Comparator.comparing(Ship::getTravelledDistance));
    Collections.reverse(ships);
    //ships ordenados por vessel type
    ships.sort(Comparator.comparing(Ship::getVesselType));
    Collections.reverse(ships);
    //ships ordenados por vessel type
    ships.sort(Comparator.comparing(Ship::getVesselType));
    //colocar os vessel types e respetivos navios num map
    Map<Integer, List<Ship>> map = new HashMap<>();
    Integer temp = 0;
    ArrayList<Ship> topNShips = new ArrayList<>();
    for (Ship ship : ships) {
        if (temp != ship.getVesselType()) {
            topNShips = new ArrayList<>();}
        topNShips.add(ship);
        map.put(ship.getVesselType(), topNShips);
        temp = ship.getVesselType();
    }
    //get os top n navios
    Map<Integer, List<Ship>> map2 = new HashMap<>();
    topNShips = new ArrayList<>();
    List<Ship> sh = new ArrayList<>();
    for (Integer vesselType : map.keySet()) {
        for (int i = 0; i < n && n <= map.get(vesselType).size(); i++) {
            topNShips.add(map.get(vesselType).get(i));
            sh.add(map.get(vesselType).get(i));
            map2.put(vesselType, topNShips);

        }
        topNShips = new ArrayList<>();
    }
    //escrever a informação necessária para um txt
    final String fileToBeWrittenTo = "getTopNShips.txt";
    StringBuilder sout = new StringBuilder("");
    for (int i = 0; i < sh.size(); i++) {
        sout.append("Vessel Type - ").append(sh.get(i).getVesselType()).append(" MMSI - ")
                .append(sh.get(i).getShipId().getMmsi()).append(" Travelled Distance = ").
                append(sh.get(i).getTravelledDistance()).append(" Mean SOG = ").append(sh.get(i).getMeanSOG());
        sout.append('\n');
    }
    try {
        PrintToFile.printB(sout, fileToBeWrittenTo);
    } catch (IllegalArgumentException e) {
        System.out.println("Error");
    }
    return map2;
    }
```

- **Complexidade Temporal: O(n log n . n)** = O(log n . n^2)

US 107

- **Class Diagram**



Para realizar esta US tive que criar um método para a classe Ship a comparar com outro objeto dessa classe para ver se eles estavam na condição pedida de terem um local de partida e de chegada parecidos(isCloseTo())e também precisa de calcular a diferença de distancia entre os dois.

Após isso criei o método getCloseShipPairs() na ShipStore que percorre a AVL com Ships e devolve uma lista com esses Pares de Ships.

A classe ShipPairList contém essa lista e também outra lista com as diferenças de travelled distance no mesmo index.

Por fim faltava apenas dar sort a essas duas listas da mesma forma, da forma como era pedido nas Acceceptance Criteria, o que foi realizado através dos private sorting methods.

Complexidade temporal

Métodos

```
public List<Pair<Ship, Ship>> getCloseShips() {
    List<Pair<Ship, Ship>> pairList = new ArrayList<>();
    for (Ship ship : store.inOrder()) {
        if (ship.getRoute().getTravelledDistance() > 10) {
            for (Ship ship2 : store.inOrder()) {
                if (ship.isClose(ship2)) {
                    Pair<Ship, Ship> pair1 = new Pair<>(ship, ship2);
                        pairList.add(pair1);
                }
            }
        }
    }
    return pairList;
}
```

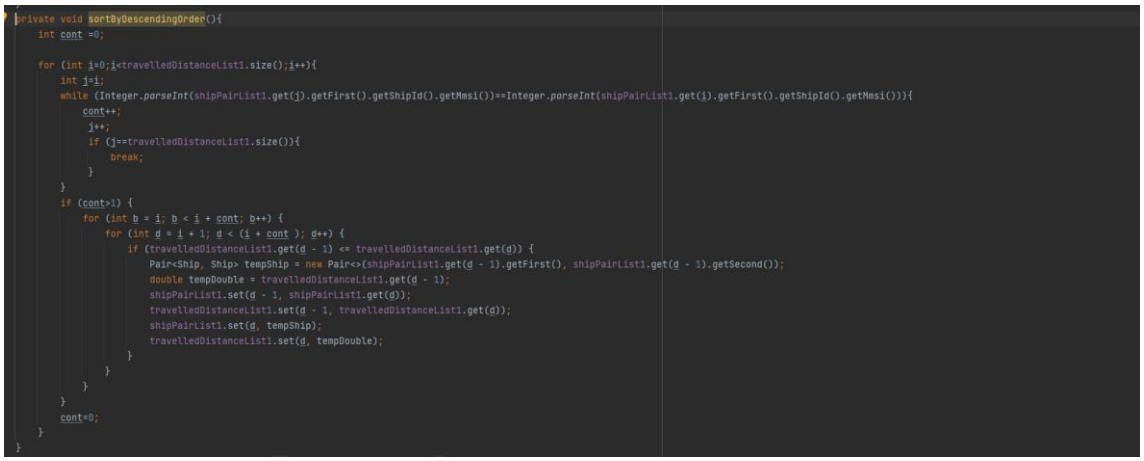
getCloseShips() tem complexidade temporal O(n^2)- um for dentro do outro.

```
private List<Double> sortShipList(){
    int n = shipPairList1.size();
    for (int i=0;i<n;i++){
        for (int j=1;j<(n-i);j++){
            if (Double.parseDouble(shipPairList1.get(j-1).getFirst().getShipId().getMmsi()) > Double.parseDouble(shipPairList1.get(j).getFirst().getShipId().getMmsi())){
                Pair<Ship, Ship> tempShip = new Pair<>(shipPairList1.get(j-1).getFirst(), shipPairList1.get(j-1).getSecond());
                shipPairList1.set(j-1,shipPairList1.get(j));
                shipPairList1.set(j,tempShip);
            }
        }
    }
    List<Double> travelledDistanceList = new ArrayList<>();
    for (Pair<Ship, Ship> shipShipPair : shipPairList1) {
        travelledDistanceList.add(Math.abs(shipShipPair.getFirst().getRoute().getTravelledDistance() - shipShipPair.getSecond().getRoute().getTravelledDistance()));
    }
    return travelledDistanceList;
}
```

sortShipList() é um bubble sort (O(n^2)) e de seguida tem a criação de uma lista O(n).

```
private void removeRepeats(){
    int cont =0;
    for (int i=0;i<shipPairList1.size();i++){
        for (int j=0;j<shipPairList1.size();j++){
            if (equalShipPairs(shipPairList1.get(i),shipPairList1.get(j))){
                cont++;
            }
            if (cont==2){
                shipPairList1.remove(j);
                travelledDistanceList1.remove(j);
                cont=0;
            }
        }
        cont=0;
    }
}
```

removeRepeats() tem um for dentro do outro(O(n^2) e serve para remover pares repetidos .



```
private void sortByDescendingOrder(){
    int cont =0;

    for (int i=0;i<travelledDistanceList1.size();i++){
        int j=i;
        while (Integer.parseInt(shipPairList1.get(j).getFirst().getShipId().getMmsi())==Integer.parseInt(shipPairList1.get(i).getFirst().getShipId().getMmsi())){
            cont++;
            j++;
            if (j==travelledDistanceList1.size()){
                break;
            }
        }
        if (cont>1) {
            for (int b = i; b < i + cont; b++) {
                for (int d = i + 1; d < (i + cont ); d++) {
                    if (travelledDistanceList1.get(d - 1) <= travelledDistanceList1.get(d)) {
                        Pair<Ship, Ship> tempShip = new Pair<>(shipPairList1.get(d - 1).getFirst(), shipPairList1.get(d - 1).getSecond());
                        double tempDouble = travelledDistanceList1.get(d - 1);
                        shipPairList1.set(d - 1, shipPairList1.get(d));
                        travelledDistanceList1.set(d - 1, travelledDistanceList1.get(d));
                        shipPairList1.set(d, tempShip);
                        travelledDistanceList1.set(d, tempDouble);
                    }
                }
            }
        }
        cont=0;
    }
}
```

sortByDescendingOrder() apesar de ter dois ciclos interligados no inicio, esses so percorrem a lista uma vez(O(n)) e depois tem um bubble sort(O(n^2),logo tem complexidade temporal de O(n^2)