PORTFOLIO

Software Development F23

Sebastian Antonio Lira
Studnr. 68932

Navn: Sebastian Antonio Lira	Software Development F23	Dato: 04-05-23
Stud nr. 68932	Roskilde Universitetscenter	Eksamen: Endelig Portefølje

Indhold

Portfolio 1:

Introduktion:

Denne portfolio beskriver design og implementering af et objektorienteret skibshåndteringssystem, der fokuserer på forskellige skibstyper og deres kapaciteter. Systemet er designet ved hjælp af grundlæggende objektorienteret programmering og gør brug af koncepter som inheritance og polymorphism for at skabe en fleksibel og genanvendelig kodebase. Målet med dette projekt er at udvikle et simpelt, men effektivt system til at håndtere lastning og overvågning af last på forskellige skibstyper, herunder containerskibe, tankskibe og RoRo-skibe.

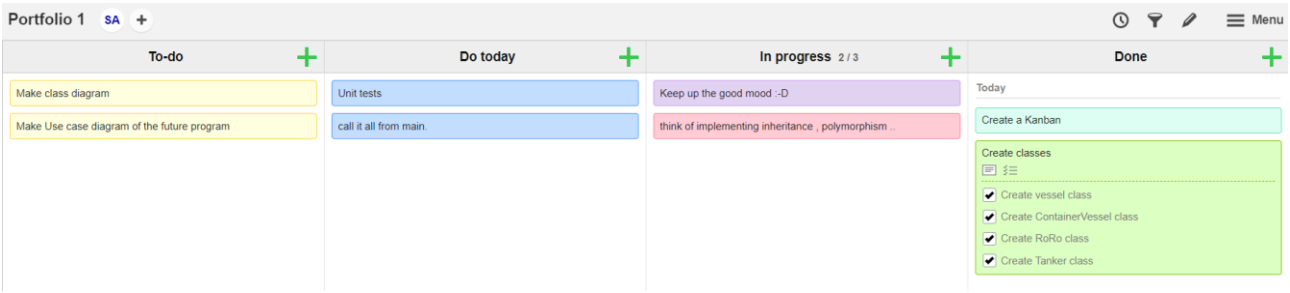
Porteføljen præsenterer først en detaljeret gennemgang af den eksisterende kode, der beskriver funktionerne og interaktionerne mellem de forskellige klasser og objekter. Dernæst følger en analyse af, hvordan koden kan udvides og forbedres for at imødekomme yderligere krav og funktioner, såsom lastning og aflastning af last samt visning af lastinformation.

Endelig vil opgaven indeholde et use case diagram, der beskriver de potentielle use cases og aktører i forbindelse med den fremtidige kode. Dette diagram vil tjene som en guide for den videre udvikling af systemet og give en bedre forståelse af, hvordan de forskellige komponenter i systemet arbejder sammen for at håndtere last på skibene.

Gennem denne opgave vil læseren få indsigt i objektorienteret programmering, inheritance, polymorphism og design af et simpelt skibshåndteringssystem. Denne viden kan bruges som grundlag for yderligere udvikling og forbedring af systemet og tilføjelse af nye funktioner, der øger systemets anvendelighed og effektivitet.

Kanban:

Kanban er et projektledelsesværktøj, der hjælper med at gøre arbejdsprocessen mere overskuelig og struktureret. Det er et nyttigt redskab til at identificere eventuelle oversete aspekter og prioritere opgaver. Kanban-tavlen, som kan ses på billedet herunder, er opdelt i fire kolonner: To-do, Do today, In progress og Done. Billedet viser et udsnit fra før projektets afslutning for at illustrere de forskellige kolonner i brug.



Kanban-metoden har vist sig at være effektiv i arbejdsprocessen, da den opretholder fokus og sikrer, at projektet ikke går i unødige retninger. Ved at begrænse antallet af opgaver i gang og visualisere projektets fremdrift kan teamet hurtigt identificere flaskehalse og justere processen for at forbedre effektiviteten. Kanban kan ogsåbidrager også til bedre kommunikation og samarbejde inden for et team, hvilket fører til hurtigere levering og kortere arbejdstid.

Sammenfattende hjælper Kanban med at skabe en mere strømlinet og effektiv arbejdsproces ved at visualisere opgaver og prioritere arbejdet. Dette resulterer i øget produktivitet, hurtigere levering og en bedre forståelse af projektets status og fremdrift.

Koder:

Kode 1 - Main.java:

Dette er hovedklassen, hvor programmet udføres. Den opretter objekter af forskellige skibstyper (ContainerVessel, Tanker, RoRo) og kalder metoderne loadingCargo() og loadFraction() for hver af dem. Dette er eksempler på, hvordan man opretter og interagerer med objekter af disse skibstyper.

Kode 2 - Vessel.java:

Dette er en grundlæggende abstrakt klasse, som repræsenterer et skib. Den har en kapacitet og to metoder - loadingCargo() og loadFraction(). Denne klasse fungerer som en skabelon for de forskellige skibstyper.

Kode 3 - ContainerVessel.java:

Denne klasse udvider Vessel-klassen og repræsenterer et container-skib. Det har en ekstra instansvariabel numberOfContainers. Metoden loadingCargo() er blevet tildelt specifik funktionalitet for container-skibe, og loadFraction() metoden returnerer en brøkdel af den aktuelle last i forhold til skibets kapacitet.

Kode 4 - RoRo.java:

RoRo-klassen udvider også Vessel-klassen og repræsenterer et rulle-på-rulle-af-skib. Den har to ekstra instansvariabler, numberOfCars og numberOfTrucks. Metoden loadingCargo() er blevet tildelt specifik funktionalitet for RoRo-skibe, og loadFraction() metoden returnerer en brøkdel af den aktuelle last i forhold til skibets kapacitet.

Kode 5 - Tanker.java:

Denne klasse udvider også Vessel-klassen og repræsenterer et tankskib. Det har en ekstra instansvariabel compartments, som er et array af Strings. Metoden loadingCargo() er blevet tildelt specifik funktionalitet for tankskibe, og loadFraction() metoden returnerer en brøkdel af de aktuelt benyttede rum i forhold til det samlede antal rum.

Når koden køres:

For at forklare, hvordan koden fortsætter med at arbejde, vil jeg beskrive, hvad der sker, når Main.java køres, og hvordan de forskellige objekter og metoder interagerer med hinanden:

- Main.java opretter et ContainerVessel objekt med en kapacitet på 100 containere.
- loadingCargo() metoden kaldes på ContainerVessel objektet med 50 containere og "TEU" (Twenty-foot Equivalent Units) som inputparametre. Eftersom enheden er "TEU", vil numberOfContainers blive sat til 50.
- loadFraction() metoden kaldes på ContainerVessel objektet, og resultatet, som er en brøkdel af den aktuelle last i forhold til skibets kapacitet, udskrives.
- Et Tanker objekt oprettes med en kapacitet på 500 m^3.
- loadingCargo() metoden kaldes tre gange på Tanker objektet med forskellige inputparametre. Rum 1, 5 og 10 fyldes op og markeres som "occupied" i compartments arrayet.
- loadFraction() metoden kaldes på Tanker objektet, og resultatet, som er en brøkdel af de aktuelt benyttede rum i forhold til det samlede antal rum, udskrives.
- Et RoRo objekt oprettes med en kapacitet på 200 biler.

- loadingCargo() metoden kaldes to gange på RoRo objektet med forskellige inputparametre. Først lastes 80 biler, og derefter lastes 10 lastbiler.
- loadFraction() metoden kaldes på RoRo objektet, og resultatet, som er en brøkdel af den aktuelle last i forhold til skibets kapacitet, udskrives.
- Når Main.java udføres, vil det oprette objekter af de forskellige skibstyper og demonstrere, hvordan man interagerer med dem ved at kalde loadingCargo() og loadFraction() metoderne.

Inheritance/polymorphism:

Koderne gør brug af Inheritance og polymorphism gennem objektorienteret programmering ved at oprette en grundlæggende klasse Vessel og derefter udvide denne klasse med underklasser for hver skibstype. Her er en detaljeret forklaring af, hvordan koderne bruger Inheritance og polymorphism:

Inheritance:

Inheritance er en mekanisme i objektorienteret programmering, der gør det muligt for en klasse at Inheritance egenskaber og metoder fra en anden klasse. I dette tilfælde arver underklasserne ContainerVessel, Tanker og RoRo fra grundklassen Vessel.

Grundklassen Vessel definerer en kapacitetsinstansvariabel og to metoder: loadingCargo() og loadFraction(). Disse metoder fungerer som skabeloner for, hvad hver underklasse skal implementere.

Underklasserne ContainerVessel, Tanker og RoRo udvider Vessel-klassen ved at bruge nøgleordet extends og Inheritance derved instansvariabler og metoder fra grundklassen. Hver underklasse tilpasser de arvede metoder til sin egen skibstype ved at overskrive dem med @Override-annotationen.

Polymorphism:

Polymorphism er en mekanisme, der gør det muligt for en metode i en klasse at have forskellige implementeringer i dens underklasser. Polymorphism opnås i dette tilfælde ved at overskrive metoderne loadingCargo() og loadFraction() i underklasserne ContainerVessel, Tanker og RoRo.

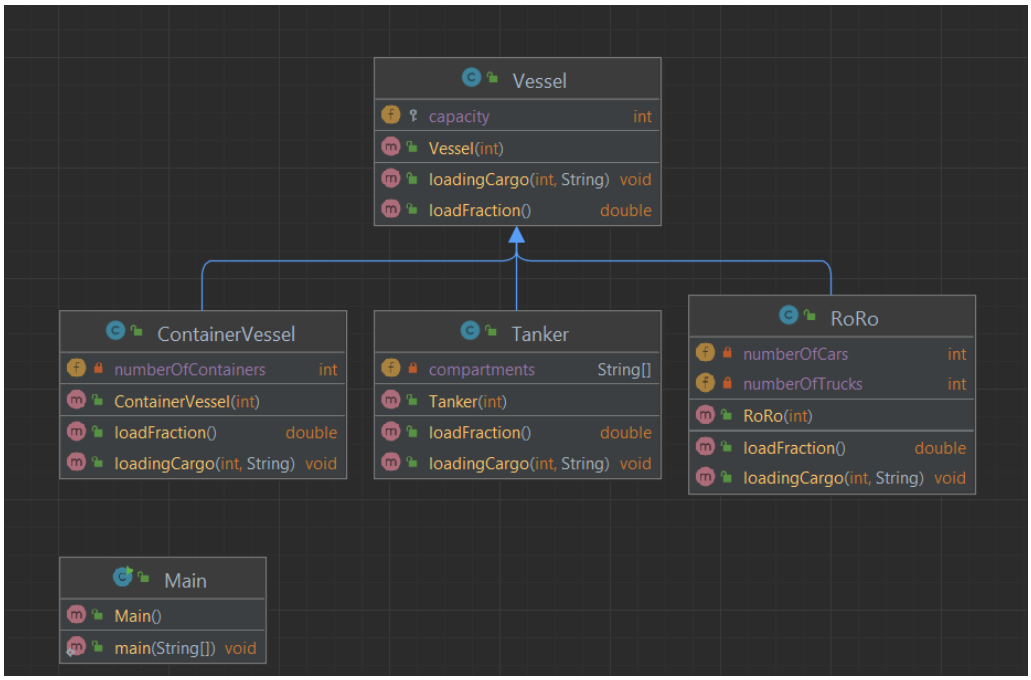
Når metoderne loadingCargo() og loadFraction() kaldes på et objekt af en bestemt skibstype, vil den tilpassede version af metoden i den relevante underklasse blive udført i stedet for metoden i grundklassen Vessel. Dette er et eksempel på polymorphism, da metoderne i grundklassen kan have forskellige implementeringer afhængigt af den aktuelle skibstype.

Polymorphism gør det muligt at behandle objekter af forskellige skibstyper på en generel måde ved hjælp af deres fælles grundklasse Vessel. Dette forenkler koden og gør den mere fleksibel, da nye skibstyper kan tilføjes ved blot at udvide Vessel-klassen og tilpasse de nødvendige metoder.

Således bruger koderne Inheritance og polymorphism til at organisere og strukturere programmet på en objektorienteret måde og for at opnå en fleksibel og genanvendelig kodebase.

Class diagram:

Herunder er der blevet udarbejdet et class diagram, der viser hvordan de forskellige klasser hænger sammen.



I dette class diagram:

Vessel er en abstrakt klasse, der repræsenterer et generisk skib med en kapacitet. Den indeholder metoderne loadingCargo() og loadFraction() og en konstruktør, der tager en kapacitet som parameter.

ContainerVessel, Tanker og RoRo er underklasser, der udvider Vessel-klassen. De indeholder deres egne instansvariabler og overskriver metoderne fra Vessel-klassen for at tilpasse dem til deres respektive skibstyper. Hver underklasse har en konstruktør, der tager en kapacitet som parameter og kalder superkonstruktøren fra Vessel-klassen.

Dette klassediagram repræsenterer den aktuelle kode og kan bruges som en visualisering for at forstå systemets struktur og de forskellige klasser og metoder, der anvendes i koden.

Fremtidig vision for koden:

Der er et stort potentiale, for at arbejde videre med koden. Der er en masse flere funktioner, der kan blive implementeret i fremtiden. Nogle af de tanker jeg har gjort mig, at der kunne gøres ved koden, er følgende:

Tilføj metoder til at fjerne last fra skibene:

For hver af skibstyperne (ContainerVessel, Tanker, RoRo), tilføj en metode unloadingCargo(), der fjerner last baseret på de specifikke krav for hver skibstype.

Håndter kapacitetsgrænser:

Opdater loadingCargo() metoderne for hver skibstype, så de kontrollerer, om der er tilstrækkelig kapacitet til at laste den ønskede mængde last, før de faktisk laster den. Hvis der ikke er nok kapacitet, skal der vises en passende fejlmeddelelse.

Forbedre datarepræsentationen af lasten i Tanker klassen:

I stedet for at bruge et String-array til at repræsentere compartments, kan du bruge et boolean[] eller int[]-array, hvor true eller 1 repræsenterer et besat rum, og false eller 0 repræsenterer et tomt rum. Dette vil gøre det lettere at arbejde med lastdataene og gøre koden mere effektiv.

Tilføj metoder til at vise lastinformation:

Tilføj en metode displayCargo() til hver af skibstyperne, der udskriver detaljeret lastinformation, såsom antallet af containere, biler, lastbiler eller besatte rum.

Opret en brugergrænseflade:

For at gøre programmet mere interaktivt og brugervenligt, kan du oprette en simpel tekstbaseret eller grafisk brugergrænseflade, der giver brugeren mulighed for at vælge skibstyper, indtaste lastinformation og udføre handlinger som at laste og aflaste last og se lastinformation.

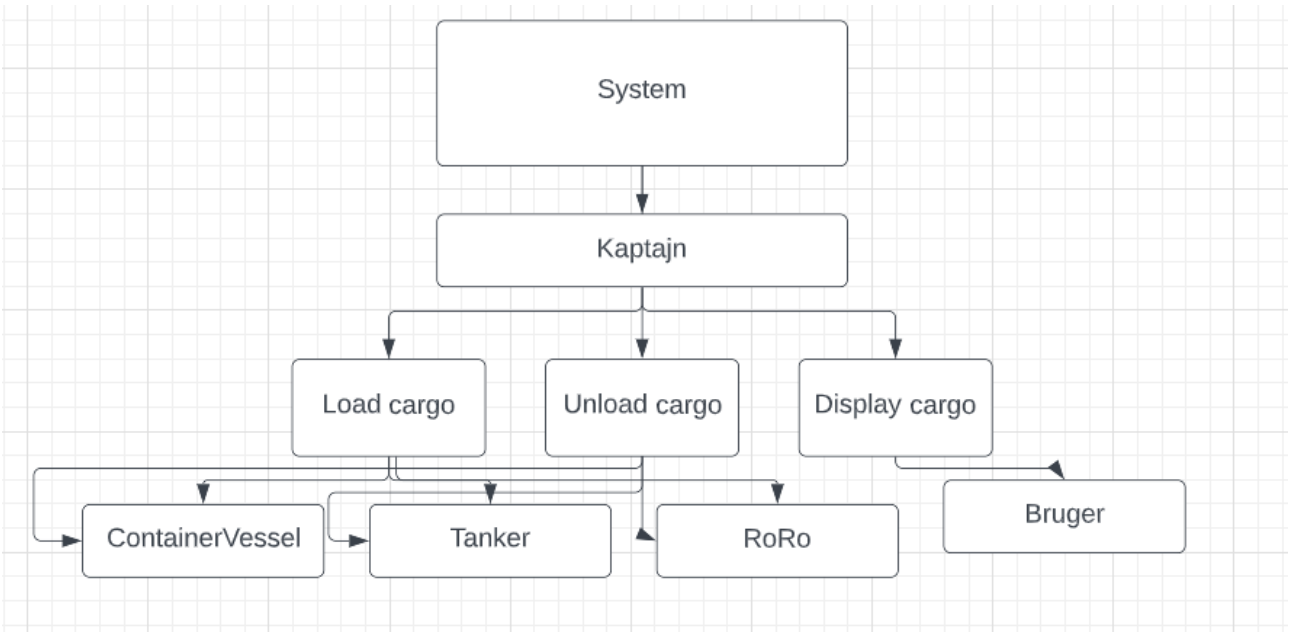
Implementer fejlhåndtering og validering af input:

Forbedre inputvalideringen og fejlhåndteringen i loadingCargo() metoderne for hver skibstype ved at kontrollere for ugyldige inputværdier, som f.eks. negative tal eller ugyldige tekststrenger.

Ved at tilføje disse funktioner og forbedringer vil du kunne udvide programmets funktionalitet og gøre det mere robust og brugervenligt.

Use case diagram:

Her er der blevet udarbejdet et use case diagram, der beskriver de potentielle use cases og aktører i forbindelse med den fremtidige kode. Det skal lige stilføjes, at det er blevet udarbejdet med tanke omkring de nævnte forbedringer, er blevet implementeret i koden.



Aktører:

Kaptajnen: Denne aktør er ansvarlig for at styre skibenes lastning, aflastning og visning af lastinformation.

Brugeren: Denne aktør repræsenterer brugeren af systemet, der interagerer med systemet gennem en brugergrænseflade.

Use Cases:

Load Cargo: Denne use case håndterer lastning af last på et skib (ContainerVessel, Tanker, RoRo) baseret på skibstypen og lastens specifikationer.

Unload Cargo: Denne use case håndterer fjernelse af last fra et skib (ContainerVessel, Tanker, RoRo) baseret på skibstypen og lastens specifikationer.

Display Cargo: Denne use case håndterer visning af lastinformation for et skib (ContainerVessel, Tanker, RoRo), så brugeren kan se detaljerne om lasten ombord på skibet.

System:

Systemet repræsenterer den samlede softwareapplikation, der styrer skibene og deres last.

Dette use case diagram skal mere ses som en guide for at tilføje yderligere funktionalitet og forbedre den eksisterende kode, som beskrevet i tidligere svar.

Portfolio 2:

Introduktion:

Porteføljen består i at udvikle en brugervenlig og interaktiv applikation, der hjælper brugerne med at finde rejser mellem forskellige havne. Applikationen skal oprette forbindelse til en SQLite-database for at hente oplysninger om havne og rejser og præsentere dem i en grafisk brugergrænseflade (GUI) ved hjælp af JavaFX-teknologi. Brugerne skal kunne vælge afgangshavn, ankomsthavn, dato og volumen og udføre en søgning for at finde en passende rejse. Resultaterne skal vises i GUI'en, så brugeren kan træffe en informeret beslutning. Opgaven kræver effektiv brug af databasetilslutninger, fejlhåndtering og brugergrænsefladedesign for at skabe en funktionel og brugervenlig applikation.

ER-diagram:

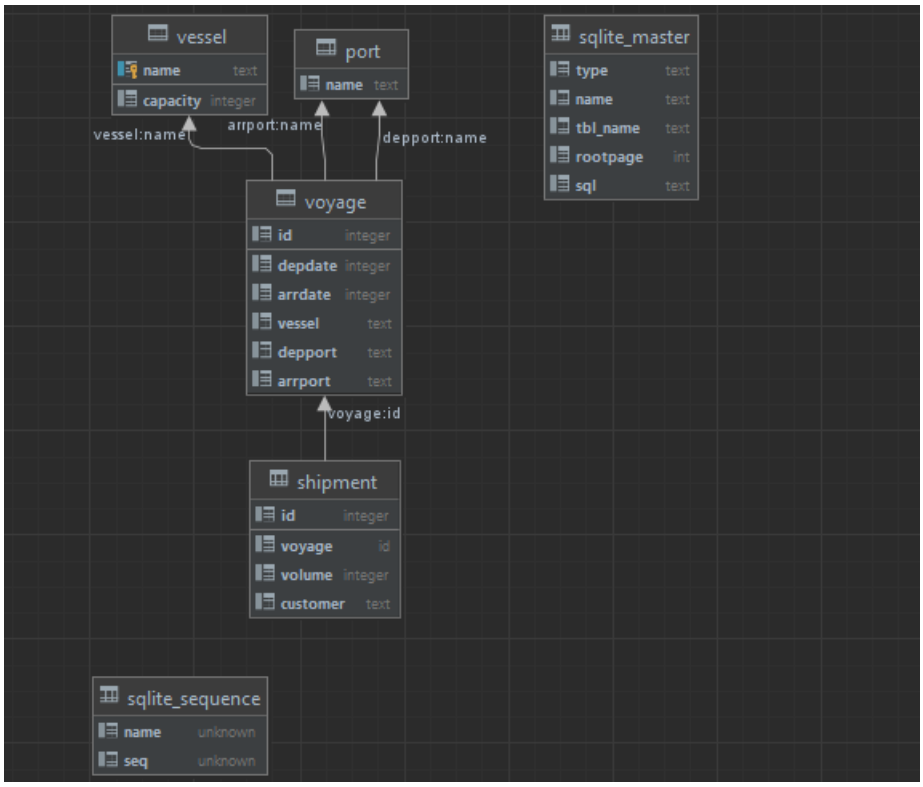
Der er blevet udarbejdet et E/R-diagram, der har til formål med at hjælpe med at forstå, hvordan dataene er organiseret og forbundet i databasen. I den givne kode er der flere SQL-forespørgsler, der arbejder med disse entiteter og deres relationer. Her er nogle eksempler på, hvordan diagrammet hjælper med at forstå koden:

I funktionen findVoyage, bruges SQL-forespørgslen til at finde et tilgængeligt skib (vessel) for en bestemt rute og dato. E/R-diagrammet viser, at der er en relation mellem "voyage" og "vessel" gennem attributten "vessel", hvilket gør det lettere at forstå, hvordan søgningen udføres ved at sammenkoble de to entiteter.

I funktionen getFailedVoyages, udføres SQL-forespørgsler for at finde "voyage"-entiteter, der ikke opfylder visse betingelser. E/R-diagrammet hjælper med at forstå strukturen af "voyage"-entiteten og dens attributter, hvilket gør det lettere at formulere SQL-forespørgsler for at finde de fejlslagne rejser.

I den udkommenterede funktion addShipment, ville SQL-forespørgslen blive brugt til at indsætte en ny forsendelse (shipment) i databasen. E/R-diagrammet viser, at der er en relation mellem "shipment" og "voyage", hvilket hjælper med at forstå, hvordan man korrekt indsætter en ny forsendelsespost med de korrekte fremmednøgle-værdier.

Generelt hjælper E/R-diagrammet med at forstå de underliggende datastrukturer og relationer i koden og kan bruges som en reference, når man arbejder med SQL-forespørgsler og datamanipulation. Diagrammet kan ses herunder:



Der er tre entiteter i diagrammet: voyage, vessel og shipment. Hver entitet har visse attributter som beskrevet nedenfor:

voyage (rejse):

id (Primary key)

depdate (afgangsdato)

arrdate (ankomstdato)

depport (afgangshavn)

arrport (ankomsthavn)

vessel (foreign key, der refererer til vessel.name)

vessel (skib):

name (primary key)

capacity (kapacitet)

shipment (forsendelse):

id (primær nøgle)

vessel (foreign key, der refererer til vessel.name)

depport (afgangshavn)

arrport (ankomsthavn)

depdate (afgangsdato)

volume (volumen)

Der er to relationer i diagrammet, der forbinder entiteterne:

"made by": En "voyage" er altid lavet af et enkelt "vessel", og et "vessel" kan lave flere "voyage". Dette skaber en 1-til-mange relation mellem "vessel" og "voyage". På E/R-diagrammet er dette repræsenteret med en linje mellem entiteterne og kardinalitetetiketter "1" og "1..*" for henholdsvis "vessel" og "voyage".

"assigned to": En "shipment" (forsendelse) er altid tildelt til en bestemt "voyage". En "voyage" kan have flere "shipment" tildelt. Dette skaber en 1-til-mange relation mellem "voyage" og "shipment". På E/R-diagrammet er dette repræsenteret med en linje mellem entiteterne og kardinalitet etiketter "1" og "1..*" for henholdsvis "voyage" og "shipment".

E/R-diagrammet hjælper med at visualisere strukturen i databasen og forholdet mellem entiteterne.

Kode til at tilføje til routes.csv:

Denne Java-kode demonstrerer, hvordan man tilføjer en ny række til en eksisterende CSV-fil. Koden læser først den nuværende indhold af CSV-filen og tilføjer derefter en ny række med foruddefinerede kolonneværdier. Til sidst skrives det opdaterede indhold tilbage til filen. Koden benytter BufferedReader og BufferedWriter klasserne for at læse fra og skrive til filen og StringBuilder klassen til at opbygge den opdaterede CSV-tekst.

Udsnit af koden:

```java
public class Main {
    public static void main(String[] args) {
        String csvFile = "C:\\Users\\sebas\\root mappe til intellij\\routes.csv";
        String line;
        String csvSeparator = ",";

        try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {
            StringBuilder sb = new StringBuilder();

            // Add table headers
            sb.append(230529);
            sb.append(csvSeparator);
            sb.append(230530);
            sb.append(csvSeparator);
            sb.append("SAMPLE VESSEL");
            sb.append(csvSeparator);
            sb.append("Stettin");
            sb.append(csvSeparator);
            sb.append("Copenhagen");
            sb.append("\n");

            // Read the existing file content
            while ((line = br.readLine()) != null) {
                sb.append(line);
                sb.append("\n");
            }
```

Her er forklaring trin for trin på hvad koden gør helt specifikt.

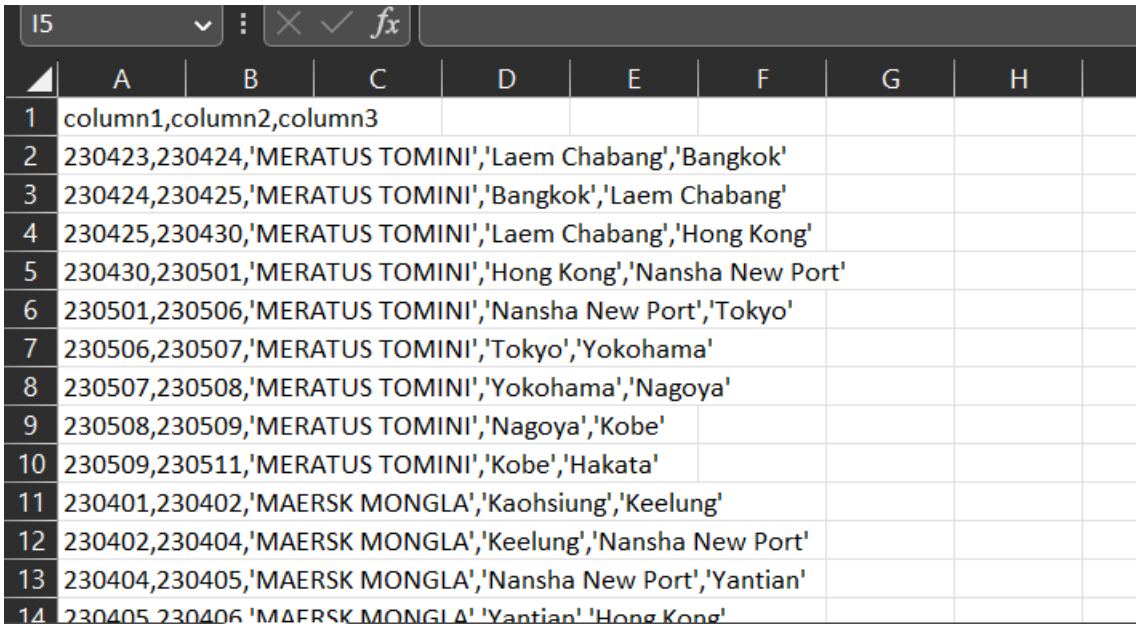
1. Importerer nødvendige biblioteker til at læse og skrive filer.
2. Klassen Main og dens main metode er defineret.

3. Filens sti (path) til "routes.csv" er angivet i variablen csvFile.
4. Variablerne line og csvSeparator er defineret. csvSeparator er sat til komma (,), som er den typiske separator for CSV-filer.
5. En try-with-resources blok er anvendt til at oprette et BufferedReader objekt, som læser indholdet af CSV-filen. Dette lukker automatisk ressourcerne, når blokken er færdig.
6. Et StringBuilder objekt kaldet sb oprettes for at bygge den nye CSV-tekst, der skal skrives til filen.
7. En ny række med kolonneværdier tilføjes til StringBuilder (i dette tilfælde, "230529,230530,SAMPLE VESSEL,Stettin,Copenhagen"). Dette er den række, der vil blive tilføjet til din CSV-fil.
8. Den eksisterende CSV-fil læses linje for linje ved hjælp af en while-løkke. For hver linje, der læses, tilføjes den til StringBuilder og efterfulgt af en ny linje ("\n").
9. En anden try-with-resources blok er anvendt til at oprette et BufferedWriter objekt, som skriver den nye CSV-tekst til filen. Dette lukker også automatisk ressourcerne, når blokken er færdig.
10. Hvis der opstår en fejl ved skrivning til filen, fanges denne ved hjælp af en catch-blok, og fejlmeddelelsen udskrives.
11. Hvis der opstår en fejl ved læsning af filen, fanges denne også ved hjælp af en catch-blok, og fejlmeddelelsen udskrives.

Koden tilføjer en ny række til CSV-filen ved først at tilføje den nye række til StringBuilder og derefter læse den eksisterende fil og tilføje dens indhold til StringBuilder. Til sidst skrives det samlede indhold af StringBuilder tilbage til filen, hvilket tilføjer den nye række til filen.

Processen kan ses herunder:

Før:



	A	B	C	D	E	F	G	H
1	column1,column2,column3							
2	230423,230424,'MERATUS TOMINI','Laem Chabang','Bangkok'							
3	230424,230425,'MERATUS TOMINI','Bangkok','Laem Chabang'							
4	230425,230430,'MERATUS TOMINI','Laem Chabang','Hong Kong'							
5	230430,230501,'MERATUS TOMINI','Hong Kong','Nansha New Port'							
6	230501,230506,'MERATUS TOMINI','Nansha New Port','Tokyo'							
7	230506,230507,'MERATUS TOMINI','Tokyo','Yokohama'							
8	230507,230508,'MERATUS TOMINI','Yokohama','Nagoya'							
9	230508,230509,'MERATUS TOMINI','Nagoya','Kobe'							
10	230509,230511,'MERATUS TOMINI','Kobe','Hakata'							
11	230401,230402,'MAERSK MONGLA','Kaohsiung','Keelung'							
12	230402,230404,'MAERSK MONGLA','Keelung','Nansha New Port'							
13	230404,230405,'MAERSK MONGLA','Nansha New Port','Yantian'							
14	230405,230406,'MAERSK MONGLA','Yantian','Hong Kong'							

I gang:

```
17
18          // Add table headers
19          sb.append(230529);
20          sb.append(csvSeparator);
21          sb.append(230530);
22          sb.append(csvSeparator);
23          sb.append("SAMPLE VESSEL");
24          sb.append(csvSeparator);
25          sb.append("Stettin");
26          sb.append(csvSeparator);
27          sb.append("Copenhagen");
28          sb.append("\n");
29
30          // Read the existing file content
31          while ((line = br.readLine()) != null) {
32              sb.append(line);
33              sb.append("\n");
34          }
```

```
Run:  Main
C:\Users\sebas\.jdks\openjdk-17.0.2\bin\java.exe "-javaagent:C:\Program Files\J

Process finished with exit code 0
```

Efter:

H4

	A	B	C	D	E	F	G	H	I
1	230529,230530,SAMPLE VESSEL,Stettin,Copenhagen								
2	column1,column2,column3								
3	230423,230424,'MERATUS TOMINI','Laem Chabang','Bangkok'								
4	230424,230425,'MERATUS TOMINI','Bangkok','Laem Chabang'								
5	230425,230430,'MERATUS TOMINI','Laem Chabang','Hong Kong'								
6	230430,230501,'MERATUS TOMINI','Hong Kong','Nansha New Port'								
7	230501,230506,'MERATUS TOMINI','Nansha New Port','Tokyo'								
8	230506,230507,'MERATUS TOMINI','Tokyo','Yokohama'								
9	230507,230508,'MERATUS TOMINI','Yokohama','Nagoya'								
10	230508,230509,'MERATUS TOMINI','Nagoya','Kobe'								
11	230509,230511,'MERATUS TOMINI','Kobe','Hakata'								
12	230401,230402,'MAERSK MONGLA','Kaohsiung','Keelung'								
13	230402,230404,'MAERSK MONGLA','Keelung','Nansha New Port'								
14	230404,230405,'MAERSK MONGLA','Nansha New Port','Yantian'								

Kode til Query for at se om rejserne mislykkes:

Der er blevet udarbejdet et Java-kodeeksempel fokuserer på at identificere og rapportere fejlslagne rejser fra en SQLite-database baseret på tre forskellige betingelser ved hjælp af SQL-søgninger. Koden benytter JDBC (Java Database Connectivity) til at oprette forbindelse til databasen, udføre forespørgsler og hente resultater. Fejlslagne rejser for hver betingelse hentes og udskrives ved at kalde en hjælpemetode, getFailedVoyages. Dette eksempel demonstrerer effektiv brug af JDBC og databaseforbindelser, ressourcehåndtering og fejlhåndtering ved at udnytte try-catch blokke og finally blokke for at sikre korrekt lukning af ressourcer.

Udsnit af koden:

```
public static void main(String[] args) {
    // fail conditions
    String query1 = "SELECT id FROM voyage WHERE (arrdate - depdate) <= 0";
    String query2 = "SELECT id FROM voyage WHERE depport = arrport";
    String query3 = "SELECT voyage.id FROM voyage " +
            "JOIN vessel ON voyage.vessel = vessel.name " +
            "WHERE vessel.capacity < 3000";

    // tells what number of the arrays will fail
    List<Integer> failedVoyages1 = getFailedVoyages(query1);
    List<Integer> failedVoyages2 = getFailedVoyages(query2);
    List<Integer> failedVoyages3 = getFailedVoyages(query3);

    // print the potential fails.
    System.out.println("Failed voyages for condition 1: " + failedVoyages1);
    System.out.println("Failed voyages for condition 2: " + failedVoyages2);
    System.out.println("Failed voyages for condition 3: " + failedVoyages3);
```

Her er en trinvis forklaring af, hvad der sker i koden:

1. I main-metoden defineres tre SQL-søgninger (query1, query2, query3), som hver repræsenterer en fejlbetingelse for rejser:

- query1: Rejser, hvor ankomstdatoen minus afrejsedatoen er mindre end eller lig med 0.
- query2: Rejser, hvor afgangshavnen og ankomsthavnen er den samme.
- query3: Rejser, hvor skibets kapacitet er mindre end 3000.

2. De fejlslagne rejser for hver betingelse hentes ved at kalde getFailedVoyages-metoden med den relevante SQL-søgning som parameter.
3. Resultaterne (fejlslagne rejser) udskrives for hver betingelse.
4. getFailedVoyages-metoden tager en SQL-søgning som input og returnerer en liste over fejlslagne rejser (id'er). Metoden gør følgende:

- Opretter en tom liste failedVoyages til at gemme fejlslagne rejser.
- Opretter en forbindelse til SQLite-databasen ved hjælp af JDBC.
- Opretter et Statement-objekt og udfører SQL-søgningen ved at kalde executeQuery-metoden.

- Læser resultatsættet (ResultSet) linje for linje ved hjælp af en while-løkke. For hver linje hentes rejsens id og tilføjes til listen over fejlslagne rejser.
- Lukker ressourcer som Statement og Connection i finally-blokke for at sikre, at ressourcerne lukkes korrekt, uanset om der opstår en fejl eller ej.

Et udsnit af getFailedVoyages metoden:

```
123   @    static List<Integer> getFailedVoyages(String query) {
124            List<Integer> failedVoyages = new ArrayList<>();
125            Connection conn = null;
126            try {
127                String url = "jdbc:sqlite:identifier.sqlite";
128                conn = DriverManager.getConnection(url);
129
130                Statement stmt = null;
131
132                try {
133                    stmt = conn.createStatement();
134                    ResultSet rs = stmt.executeQuery(query);
135                    while (rs.next()) {
136                        int voyageId = rs.getInt( columnLabel: "id");
137                        failedVoyages.add(voyageId);
138                    }
139                } catch (SQLException e) {
140                    throw new Error( message: "Problem", e);
141                } finally {
142                    if (stmt != null) {
143                        stmt.close();
144                    }
145                }
```

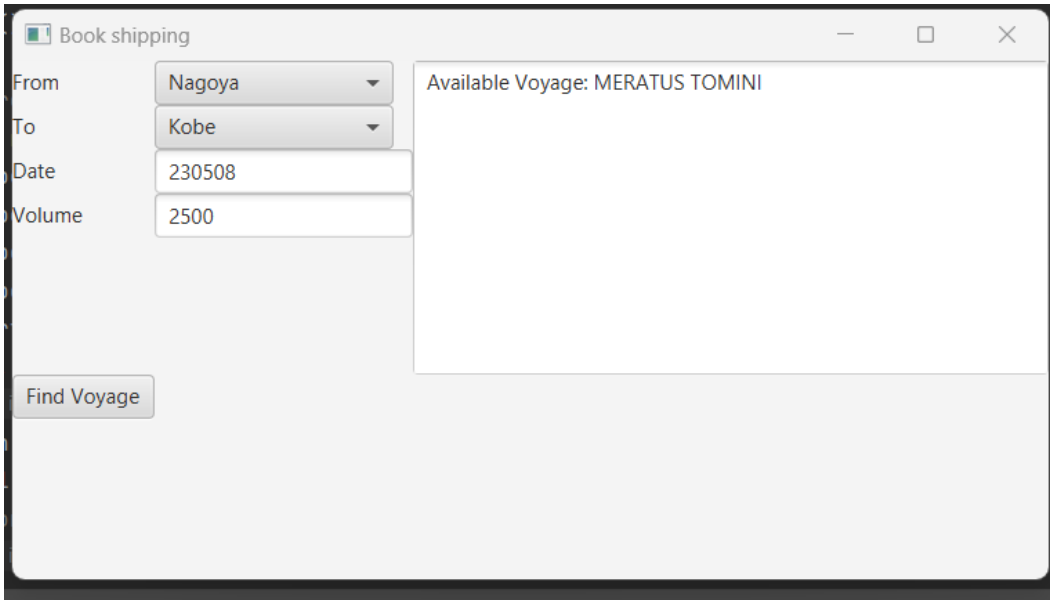
JavaFx interface kode:

Denne Java-kode er et JavaFX-baseret program, der hjælper brugerne med at finde rejser mellem forskellige havne ved hjælp af en interaktiv grafisk brugergrænseflade (GUI). Programmet opretter forbindelse til en SQLite-database for at hente oplysninger om havne og rejser og præsenterer dem i et brugervenligt format. Brugerne kan vælge afgangshavn, ankomsthavn, dato og volumen og udføre en søgning for at finde en passende rejse. Resultaterne af søgningen vises i GUI'en, så brugeren kan se de tilgængelige muligheder og tage en informeret beslutning. Programmet demonstrerer effektiv brug af JavaFX-teknologi, databasetilslutninger og fejlhåndtering, hvilket resulterer i en funktionel og brugervenlig applikation til søgning af rejser.

Udsnit af koden:

```
33        // grit structure JavaFX
34        GridPane root = new GridPane();
35        root.add(new Label( s: "From"),  i: 0,  i1: 1);
36        root.add(new Label( s: "To"),  i: 0,  i1: 2);
37        root.add(new Label( s: "Date"),  i: 0,  i1: 3);
38        root.add(new Label( s: "Volume"),  i: 0,  i1: 4);
39        root.add(fromport,  i: 1,  i1: 1);
40        root.add(toport,  i: 1,  i1: 2);
41        root.add(dato,  i: 1,  i1: 3);
42        root.add(volumen,  i: 1,  i1: 4);
43        root.add(button1,  i: 0,  i1: 9);
44       // root.add(button2, 0, 10);
45        root.add(area,  i: 3,  i1: 1,  i2: 9,  i3: 6);
46
47        button1.setOnAction(e -> doSearch(fromport.getValue(), toport.getValue(), dato.getText(), volumen.getText()));
48        //button2.setOnAction(e -> bookVoyage(fromport.getValue(), toport.getValue(), dato.getText(), volumen.getText()));
49
50        Scene scene = new Scene(root,  v: 600,  v1: 300);
51        stage.setTitle("Book shipping");
52        stage.setScene(scene);
53        stage.show();
54    }
55
```

Billede af GUI:



Her er en trin for trin beskrivelse af koden:

1. HelloApplication-klassen udvider Application-klassen fra JavaFX-biblioteket og overskriver start-metoden.
2. start-metoden opretter en GUI med knapper, tekstfelter og en rulleliste for at tillade brugeren at vælge havne, angive dato og volumen og udføre søgninger.
3. getPorts-metoden henter en liste over unikke havne fra databasen og returnerer dem som en liste af strenge. Denne liste bruges til at udfylde rullelisterne fromport og toport i GUI'en.
4. doSearch-metoden kaldes, når brugeren klikker på "Find Voyage"-knappen. Den tager brugerens input og kalder findVoyage-metoden for at søge efter en tilgængelig rejse i databasen.

5. findVoyage-metoden udfører en SQL-søgning i databasen baseret på brugerens input og returnerer navnet på det første tilgængelige skib, der opfylder søgekriterierne. Hvis ingen rejse findes, returnerer den null.
6. Efter søgningen vises resultaterne i en TextArea-widget i GUI'en. Hvis en tilgængelig rejse findes, vises skibets navn. Hvis ikke, vises en besked, der siger "No available voyages found."

Portfolio 3:

Introduktion:

I denne Portefølje gennemgås der et udviklet et program til at modellere et skibsfartsnetværk mellem forskellige havne. Netværket vil blive repræsenteret som en vægtet graf, hvor hvert knudepunkt er en havn, og kanterne mellem knudepunkterne repræsenterer rejser mellem havnene med deres respektive rejsedage som vægte. Programmet skal kunne læse forbindelserne mellem havnene og deres rejsedage fra en tekstfil, bygge grafen, afgøre, om grafen er sammenhængende, og beregne det minimale spændetræ for grafen. Resultaterne bliver præsenteret.

Koder:

Main:

Main er hovedprogrammet, der fungerer som en driver for hele applikationen. Først opretter den et ShippingNetwork-objekt ved at indlæse data fra en tekstfil, der indeholder oplysninger om forskellige havne og deres forbindelser samt varigheden i dage for at rejse mellem dem.

Derefter oprettes der en Map<String, List<Edge>>-struktur, som repræsenterer den graf, der indeholder havne og deres forbindelser. For hver havn i grafen udskrives havnens navn og dens tilsluttede destinationer sammen med rejsetiden i dage mellem dem.

Programmet tester også, om grafen er sammenkoblet, hvilket betyder, at man kan nå fra en hvilken som helst havn til en anden havn direkte eller indirekte. Hvis grafen er sammenkoblet, vil outputtet bekræfte det.

Dernæst beregner programmet minimum spanning Tree (MST) for grafen ved hjælp af Kruskal's algoritme. Et minimum spanning tree er en under-graf, der forbinder alle knuderne med den mindst mulige samlede vægt (i dette tilfælde varighed i dage). Dette er vigtigt for at finde den mest effektive ruteplan for skibsfarten mellem havnene.

Endelig udskriver programmet kanterne i minimum spanning tree sammen med deres vægt (dage). Det beregner og udskriver også den samlede varighed (i dage) af MST. Dette giver et overblik over de mest effektive forbindelser og den samlede rejsetid for at dække hele netværket.

Udsnit af main:

```java
public class Main {
    public static void main(String[] args) {
        ShippingNetwork shippingNetwork = new ShippingNetwork( filename: "C:\\Users\\sebas\\IdeaProjects\\portfolio3\\src\\network.txt");
        Map<String, List<Edge>> graph = shippingNetwork.getGraph();

        for (String port : graph.keySet()) {
            System.out.println("Port: " + port);
            for (Edge edge : graph.get(port)) {
                System.out.println("  Destination: " + edge.getDestination() + ", Days: " + edge.getDays());
            }
        }
        System.out.println();
        System.out.println("Graph is connected: " + shippingNetwork.isConnected());
        System.out.println();

        List<Edge> mstEdges = shippingNetwork.minimumSpanningTree();
        int totalDuration = mstEdges.stream().mapToInt(Edge::getDays).sum();

        System.out.println("Minimum Spanning Tree edges:");
        for (Edge edge : mstEdges) {
            System.out.println(edge.getSource() + " -> " + edge.getDestination() + ", Days: " + edge.getDays());
        }

        System.out.println("Total duration of the Minimum Spanning Tree: " + totalDuration + " Days");

    }
}
```

DisjointSet:

Klassen DisjointSet repræsenterer en klasse kaldet DisjointSet, som er en datastruktur, der bruges til at holde styr på en samling af delmængder. Denne datastruktur er nyttig for at implementere Kruskal's algoritme i minimums spændetræs beregningen. DisjointSet har to hovedopgaver:

1. At finde repræsentanten (rod) for en given delmængde.
2. At forene to delmængder, hvis de ikke allerede er i den samme delmængde.

Klassen indeholder to HashMap-objekter:

```java
public class DisjointSet {
    private final Map<String, String> parent = new HashMap<>();
    private final Map<String, Integer> rank = new HashMap<>();

```

- parent: Mapper hvert element (havn) til dets overordnede element (havn) i delmængden.
- rank: Holder styr på træets dybde for hver rod, hvilket hjælper med at optimere datastrukturen.

makeSet-metoden initialiserer en ny delmængde med et enkelt element. I denne metode sættes elementets forælder til sig selv, og rangen sættes til 0.

find-metoden finder roden (repræsentanten) for delmængden, som det givne element tilhører. Den bruger sti-komprimering for at optimere ydeevnen, hvilket betyder, at under søgningen opdateres hver nodes forælder til at pege direkte på roden.

```
15      public String find(String x) {
16          if (!parent.get(x).equals(x)) {
17              parent.put(x, find(parent.get(x)));
18          }
19          return parent.get(x);
20      }
```

union-metoden forsøger at forene de to delmængder, som de givne elementer tilhører. Hvis elementerne allerede er i den samme delmængde, foretages der ingen handling, og metoden returnerer false. Hvis de er i forskellige delmængder, forenes delmængderne ved at sammenkoble deres rødder. For at opretholde en afbalanceret struktur bruger metoden ranghævning: roden af træet med lavere rang bliver barn af roden af træet med højere rang. Hvis begge træer har samme rang, vælges en af dem som ny rod, og dens rang øges med 1. Metoden returnerer true for at indikere, at en sammenkobling blev foretaget.

DisjointSet-klassen bruges i Kruskal's algoritme, når man bestemmer, om en kant, der overvejes, vil skabe en cyklus i grafen. Hvis de to havne, der er forbundet af kanten, allerede er i samme delmængde, vil tilføjelsen af kanten skabe en cyklus, og kanten ignoreres. Ellers forenes delmængderne, og kanten tilføjes til MST'et.

ShippingNetwork:

Kode 3 repræsenterer en klasse kaldet ShippingNetwork, som bruges til at modellere og analysere et netværk af havne og ruter mellem dem. Denne klasse indeholder en intern datastruktur kaldet graph, som er en HashMap, der bruges til at gemme et tilstødende liste repræsentation af grafen.

ShippingNetwork-klassen har en konstruktør, der tager et filnavn som parameter. Denne fil skal indeholde information om netværket af havne og ruter, hvor hver linje repræsenterer en rute mellem to havne og varigheden af turen i dage. Konstruktøren kalder en privat metode readNetworkFromFile for at læse netværket fra filen og opbygge grafen.

readNetworkFromFile-metoden åbner og læser filen linje for linje ved hjælp af en BufferedReader. For hver linje i filen splitter den linjen i tre dele ved hjælp af kommaet som separator og tildeler dem til variablerne port1, port2 og days. Denne metode kalder derefter en anden privat metode addEdge med de ekstraherede værdier som argumenter for at tilføje ruten til grafen.

```
15      private void readNetworkFromFile(String filename) {
16          try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
17              String line;
18              while ((line = reader.readLine()) != null) {
19                  String[] tokens = line.split( regex: ",");
20                  if (tokens.length != 3) {
21                      System.err.println("Invalid line format: " + line);
22                      continue;
23                  }
24                  String port1 = tokens[0].trim();
25                  String port2 = tokens[1].trim();
26                  int days = Integer.parseInt(tokens[2].trim());
27
28                  addEdge(port1, port2, days);
29              }
30          } catch (IOException e) {
31              e.printStackTrace();
32          }
33      }
```

addEdge-metoden tager tre parametre: to havne og antallet af dage, det tager at rejse mellem dem. Denne metode opdaterer grafen ved først at sikre, at begge havne er til stede i grafen ved hjælp af putIfAbsent-metoden. Derefter tilføjer den en ny Edge-instans for hver retning af ruten mellem de to havne med den givne varighed i dage.

```
36      private void addEdge(String port1, String port2, int days) {
37          graph.putIfAbsent(port1, new ArrayList<>());
38          graph.putIfAbsent(port2, new ArrayList<>());
39          graph.get(port1).add(new Edge(port1,port2, days));
40          graph.get(port2).add(new Edge(port1,port2, days));
41      }
```

getGraph-metoden returnerer grafen som en Map af havne til en liste af Edge-objekter, der repræsenterer forbindelserne mellem havnene.

isConnected-metoden tjekker, om grafen er sammenkoblet eller ej ved hjælp af en dybde-først-søgning (DFS) algoritme. Den starter med en tilfældig havn og besøger alle tilgængelige havne i grafen ved hjælp af en rekursiv DFS-metode.

dfs-metoden er en privat metode, der implementerer DFS-algoritmen. Den besøger rekursivt alle naboer til den aktuelle havn og tilføjer dem til et visitedPorts-set. Når DFS er færdig, sammenligner isConnected-metoden størrelsen på det besøgte sæt med størrelsen på grafen for at afgøre, om alle havne er sammenkoblet.

minimumSpanningTree-metoden beregner minimums spændetræet (MST) for grafen ved hjælp af Kruskal's algoritme. Den samler først alle kanterne i grafen og sorterer dem i stigende rækkefølge efter vægt (dage). Den initialiserer en DisjointSet-instans for at holde styr på delmængder af havne og oprette disjunkte sæt for hver havn. Derefter itererer den gennem de sortererede kanter og inkluderer en kant i MST, hvis de to havne i kanten ikke allerede er i samme delmængde. Dette gøres ved at kalde union-metoden på disjointSet-instansen. Hvis union-metoden returnerer true, betyder det, at de to havne ikke var i samme delmængde før, og kanten kan tilføjes til MST. Efter at have gennemgået alle kanterne, returnerer metoden listen over kanter, der udgør MST.

Denne klasse giver dig mulighed for at læse et netværk af havne og ruter fra en tekstfil, oprette en intern repræsentation af dette netværk som en graf, kontrollere, om netværket er sammenkoblet, og beregne MST for netværket. Du kan bruge denne klasse i forbindelse med de andre klasser og hovedprogrammet for at udføre forskellige operationer på netværket og få indsigt i dets struktur og egenskaber.

Edge:

Edge-klasse repræsenterer en kant i grafen. En kant er en forbindelse mellem to havne og har en bestemt vægt, som repræsenterer rejsens varighed i dage. Klassen indeholder tre private variabler: source, destination og days. source og destination er strenge, der repræsenterer havnene i kantens endepunkter, og days er et heltal, der repræsenterer rejsens varighed.

Klassens konstruktør tager tre parametre: source, destination og days. Den initialiserer de tilsvarende private variabler med de angivne værdier. Klassen indeholder også get-metoder (getSource, getDestination, getDays) for at få adgang til værdierne af de private variabler.

```
4   public class Edge implements Comparable<Edge> {
5       private final String source;
6       private String destination;
7       private int days;
8
```

Derudover implementerer Edge-klassen Comparable<Edge>-grænsefladen, hvilket gør det muligt at sammenligne Edge-objekter og sortere dem. compareTo-metoden defineres i klassen og sammenligner to kanter baseret på deres days-attribut. Dette bruges i minimumSpanningTree-metoden i ShippingNetwork-klassen for at sortere kanterne inden beregning af MST.

Edge-klassen indeholder også equals- og hashCode-metoder, som er nødvendige for korrekt at sammenligne og lagre Edge-objekter i datastrukturer som sæt og kort. equals-metoden sammenligner to Edge-objekter ved at kontrollere, om deres destination og days-attributter er ens. hashCode-metoden genererer en hashværdi baseret på disse to attributter.

Sammenfattende er Edge-klassen en vigtig komponent i programmet, da den repræsenterer forbindelserne mellem havne og deres respektive rejsedage. Den gør det også muligt at sammenligne og sortere kanter baseret på deres vægt og sikrer korrekt funktionalitet i datastrukturen.

Beskrivelse af hvordan systemet interagerer:

1. Main-klassen opretter et ShippingNetwork-objekt ved at indlæse en tekstfil, der indeholder forbindelserne mellem havne og rejsedage.
2. ShippingNetwork-klassen bygger grafen ved hjælp af en HashMap med lister af Edge-objekter for hver havn. Den læser filen linje for linje og tilføjer kanter til grafen med addEdge-metoden.
3. Efter opbygning af grafen udskrives forbindelserne mellem havne ved hjælp af getGraph-metoden og for-løkker i Main-klassen.
4. isConnected-metoden i ShippingNetwork-klassen bruges til at afgøre, om grafen er sammenhængende eller ej. Hvis den er sammenhængende, vil alle havne være tilgængelige fra ethvert andet havn.
5. minimumSpanningTree-metoden i ShippingNetwork-klassen beregner det minimale spændetræ for grafen ved hjælp af Kruskal's algoritme. Den returnerer en liste af Edge-objekter, der udgør det minimale spændetræ.
6. Main-klassen udskriver kanterne i det minimale spændetræ og beregner den samlede varighed af spændetræet ved at opsummere dage for hver kant.

DisjointSet-klassen bruges i minimumSpanningTree-metoden for at holde styr på, hvilke havne der allerede er forbundet i det minimale spændetræ. Klassen implementerer en datastruktur kaldet "disjoint-set" eller "union-find" og indeholder metoder til at oprette, finde og forene mængder af havne.

For at opsummere, programmet læser forbindelserne mellem havne og deres rejsedage fra en tekstfil, bygger en graf for at repræsentere forbindelserne, afgør, om grafen er sammenhængende, og beregner det minimale spændetræ. Resultaterne udskrives, så brugeren kan se de forskellige forbindelser mellem havne, om grafen er sammenhængende, og hvordan det minimale spændetræ ser ud med den samlede varighed.

Udsnit kan ses herunder:

getGraph:

```
Port: Mombasa
  Destination: Mombasa, Days: 13
  Destination: Dar es Salaam, Days: 5
Port: Bangkok
  Destination: Bangkok, Days: 1
  Destination: Laem Chabang, Days: 1
  Destination: Bangkok, Days: 1
  Destination: Laem Chabang, Days: 1
```

isConnected:

```
Graph is connected: false
```

MST:

```
Tanjung Pelepas -> Jakarta, Days: 3
Tanjung Pelepas -> Laem Chabang, Days: 3
Ningbo -> Nansha New Port, Days: 4
PortB -> PortC, Days: 4
Hakata -> Yantian, Days: 4
Mombasa -> Dar es Salaam, Days: 5
Chittagong -> Singapore, Days: 8
Tanjung Pelepas -> Mombasa, Days: 13
Total duration of the Minimum Spanning Tree: 75 Days
```