Project Report

Group: Suspicious Tree
Member:
- Van Duc Tri: s3978223
- Tran Tuan Bao: s3970254
- Nguyen Ich Kiet: s3978724

1. Introduction

Summary:

Our assigned objective is to design and implement a digital management system that will serve as a substitute for conventional paper-based approaches in the management of container ports. The project is centered on the mitigation of the escalating demands placed on ports to augment their capacity and enhance operational effectiveness, which has arisen as a consequence of the proliferation of international trade.

Scope:

The scope of this project is to create a Container Port Management System using Java, with a focus on Object-Oriented Programming (OOP) principles. The task at hand encompasses the creation of a software system that is adaptable and easy to sustain. This system must tackle the complexities associated with port management by utilizing control statements and algorithms. Additionally, it should incorporate a text-based interface, manage data through file input/output or databases, establish security measures based on user roles, provide essential functionalities such as create, read, update, and delete operations, as well as statistical analysis capabilities. Furthermore, the system should be capable of preserving historical data while maintaining optimal efficiency.

Objective:

The main aim of this project is to improve our hands-on experience in software development, promote the development of problem-solving abilities, and enhance competency in object-oriented programming (OOP) principles. The project functions as a medium through which we can employ theoretical knowledge within a practical framework, thereby equipping us for prospective positions in software development and system design.

2. Project Description

When we were working on developing the Container Port Management System, our main focus was on the technical aspects that were outlined in the project specifications. Our main focus was on creating a strong and effective software solution. It was crucial for us to follow these specifications in order to achieve our goal. Our main focus was the meticulous

implementation of Object-Oriented Programming (OOP). We carefully designed a flexible class hierarchy that consisted of important components like ports, vehicles, containers, users, ships, and trucks. The hierarchical structure, which is in line with OOP principles, ensured that the system is strong and also flexible and easy to maintain.

Moreover, we emphasized the significance of problem-solving by employing algorithms and data structures strategically. In accordance with the project specifications, we utilized control statements, algorithms, and data structures to tackle different port management tasks. The tasks involved in this project included utilizing algorithms to determine fuel consumption, conducting feasibility assessments for vehicle movement, and performing comprehensive statistical analyses. We used data structures to manage entities and historical records in the system. This helped us achieve optimal performance and maintain data integrity.

In addition, we had to create a simple but effective text-based user interface. By utilizing Java's Scanner class, we were able to develop a user-friendly text interface that includes all the necessary menu options. This allows for smooth and effortless interaction with the program. The technical aspects, which are based on the project specifications, highlight our dedication to converting guidelines into a thorough and reliable Container Port Management System. This demonstrates our expertise in software development principles.

3. Implementation Details

All of the details for implementation, illustrations, explanations, etc.

Class diagram for the project:

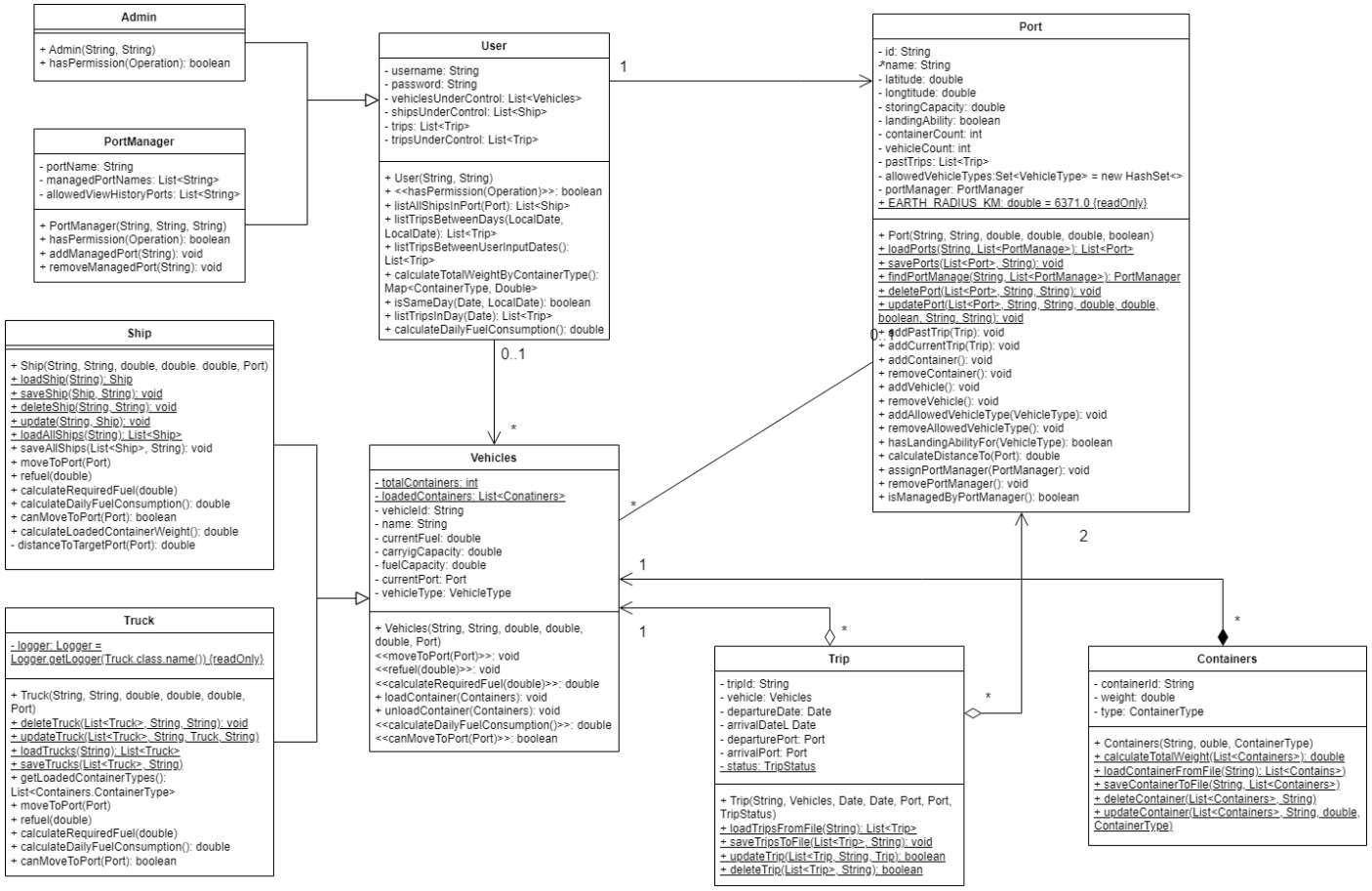

Figure 1. System class diagram

“User” class:

Description

The user class represents a user in the system. The user also has the capability to manage and execute various tasks pertaining to multiple vehicles and ports within the system.

Attributes

The user possesses a unique set of credentials, consisting of a username and password, which grants them access to the system. The user can also check which vehicles and ports are under their control.

Constructor

- `User(String username, String password)`: Initializes a new User object with a username and password. This also initializes lists for vehicles, ships, and trips under user control.

Abstract methods

- `abstract boolean hasPermission(Operation operation)`: checks if the user has permission for a specific operation and returns a boolean value.

Methods

- `List<Ship> listAllShipsInPort(Port port)`: lists all ships in a given port that are under the user's control.
- `List<Trip> listTripsBetweenDays(LocalDate dayA, LocalDate dayB)`: lists trips between two specified dates.

- `List<Trip> listTripsBetweenUserInputDates()`: lists trips between dates entered by the user.
- `Map<ContainerType, Double> calculateTotalWeightByContainerType()`: calculates the total weight of containers by type and returns a Map of container types and their corresponding weights.
- `List<Trip> listTripsInDay()`: lists trips on a specified day.
- `double calculateDailyFuelConsumption()`: calculates the daily fuel consumption of vehicles under the user's control and returns a double value.

Enum:

- `enum Operation`: An enumeration representing different types of field that a specific user can manage, including MANAGE_VEHICLES, MANAGE_CONTAINERS, MANAGE_PORTS, VIEW_HISTORY

"Admin" class:

Description

The Admin class represents a specialized type of user. Administrators typically have elevated privileges and control over all aspects of the system. This class extends the User class, inheriting its attributes and methods, but it provides its own implementation of the hasPermission method to grant permission for all operations, effectively allowing administrators to perform any action within the system.

Attributes

This extended class has no more attributes than the super class.

Constructor

- `Admin(String username, String password)`: Initializes a new Admin object with a username and password. It sets the user's role to "Admin" upon creation.

Methods

This extended method has no more methods than the super class.

"PortManager" class:

Description

The PortManager class represents a specific type of user. Port managers have responsibilities related to managing vehicles and containers within specific ports, and they may have the ability to view historical data for those ports. This class extends the User class and introduces additional properties and methods tailored to the role of port management.

Attributes

- `String portName`: An additional property specific to port managers, representing the name of the port they are responsible for.
- `List<String> managedPortNames`: A list of port names that this PortManager is authorized to manage vehicles for.
- `List<String> allowedViewHistoryPorts`: A list of ports where this PortManager can view historical data.

Constructor

- `PortManager(String username, String password, String portName)`: Initializes a new PortManager object with a username, password, and a specific port name. It sets the user's role to "PortManager" and initializes lists for managed port names and allowed view history ports.

Methods

- `boolean canManageVehicles()`: checks if the PortManager can manage vehicles for the specific port. It verifies whether the portName matches any of the managedPortNames in the list.
- `boolean canManageContainers()`: checks if the PortManager can manage containers for the specific port. It defines a list of allowed ports and checks if the portName matches any of them.
- `boolean canViewHistory()`: checks if the PortManager can view history for the specific port. It examines whether the portName is included in the list of allowedViewHistoryPorts.
- `void addManagedPort(String portName)`: adds a port to the list of managed ports for this PortManager. It allows the PortManager to take responsibility for managing vehicles in the specified port.
- `void removeManagedPort(String portName)`: removes a port from the list of managed ports. This operation indicates that the PortManager no longer manages vehicles in the specified port.

"Vehicles" class:

Description

The Vehicles class is an abstract class that serves as a blueprint for various types of vehicles within a software system focused on vehicle and container management. It defines common attributes and methods that are shared among different vehicle types, such as ships, trucks, and tankers. Vehicles in the system can carry containers, transport goods, and perform trips between ports.

Attributes

- `static int totalContainers`: A static variable that keeps track of the total number of containers across all instances of the Vehicles class.
- `static List<Containers> loadedContainers`: A static list of containers loaded onto vehicles, shared across all instances of the Vehicles class.
- `String vehicleId`: A unique identifier for the vehicle.
- `String name`: The name or identifier of the vehicle.
- `double currentFuel`: The current amount of fuel in the vehicle.
- `double carryingCapacity`: The maximum weight or capacity that the vehicle can carry.
- `double fuelCapacity`: The maximum fuel capacity of the vehicle.
- `Port currentPort`: The current port where the vehicle is located.
- `VehicleType vehicleType`: An enumeration representing the type of the vehicle (e.g., ship, truck, reefer truck, tanker truck).

Constructor

- `Vehicles(String vehicleId, String name, double currentFuel, double carryingCapacity, double fuelCapacity, Port currentPort)`: Initializes a

new Vehicles object with various attributes such as the vehicle's ID and name, currentFuel, carryingCapacity, fuelCapacity, and currentPort. It also initializes the static variables totalContainers and loadedContainers.

Abstract Methods

- `abstract void moveToPort(Port targetPort)`: defines how a vehicle should move to a specified port. It checks if the vehicle is at a port, if the target port has landing ability for the vehicle type, and if the vehicle's load capacity allows it to move.
- `abstract boolean canMoveToPort(Port port)`: checks if a vehicle can move to a specified port based on certain conditions.
- `abstract void refuel(double amount)`: defines how a vehicle should be refueled with a specified amount of fuel.
- `abstract double calculateRequiredFuel(double distance)`: calculates the required fuel for a trip of a given distance.
- `abstract double calculateDailyFuelConsumption()`: calculates the daily fuel consumption of the vehicle.

Methods

- `void loadContainer(Containers container)`: adds a container to the list of loaded containers and updates the total container count.
- `void unloadContainer(Containers container)`: removes a container from the list of loaded containers and updates the total container count.

Enum

- `enum VehicleType`: An enumeration representing different types of vehicles, including SHIP, BASIC_TRUCK, REEFER_TRUCK, and TANKER_TRUCK.

"Ship" class:

Description

The Ship class represents a specific type of vehicle within a software system that is responsible for transporting goods and containers by sea. Ships are a subclass of the Vehicles class and inherit common attributes and methods related to vehicles while adding ship-specific functionality. This class encapsulates ship-related data, actions, and interactions with ports.

Constructor

- `Ship(String vehicleId, String name, double currentFuel, double carryingCapacity, double fuelCapacity, Port currentPort)`: Initializes a new Ship object with attributes such as vehicleId, name, currentFuel, carryingCapacity, fuelCapacity, and currentPort. It sets its vehicle type to SHIP upon creation.

Static methods

- `static Ship loadShip(String filename)`: reads a ship object from a file and returns it.
- `static void saveShip(Ship ship, String filename)`: takes a Ship object and saves it to a file.
- `static void deleteShip(String filename, String vehicleIdToDelete)`: deletes a specific ship from the data file

- `static void updateShip(String filename, Ship updatedShip)`: updates ship data in the data file.
- `static List<Ship> loadAllShips(String filename)`: loads all ship data from the data file
- `static void saveAllShips(List<Ship> ships, String filename)`: saves a list of ships to the data file.

Methods

- `double distanceToTargetPort(Port targetPort)`: calculates the distance (in kilometers) between the ship's current port and a target port using the Haversine formula.

"Truck" class:

Description

The Truck class represents a specific type of vehicle within a software system that is responsible for transporting goods and containers by road. Trucks are a subclass of the Vehicles class and inherit common attributes and methods related to vehicles while adding truck-specific functionality. This class encapsulates truck-related data, actions, and interactions with ports.

Attributes

- `static final Logger logger`: A static logger for logging messages related to the Truck class.

Constructor

- `Truck(String vehicleId, String name, double currentFuel, double carryingCapacity, double fuelCapacity, Port currentPort)`: Initializes a new Truck object with attributes such as vehicleId, name, currentFuel, carryingCapacity, fuelCapacity, and currentPort. This constructor creates a truck object with the specified properties.

Static Methods

- `static void deleteTruck(List<Truck> trucks, String vehicleId, String fileName)`: Deletes a truck from a list of trucks based on its vehicleId and updates the list by saving it to a file.
- `static void updateTruck(List<Truck> trucks, String vehicleId, Truck updatedTruck, String fileName)`: Updates the properties of a truck in the list of trucks based on its vehicleId and saves the updated list to a file.
- `static List<Truck> loadTrucks(String fileName)`: Loads a list of trucks from a file using object deserialization. It returns the list of loaded trucks.
- `static void saveTrucks(List<Truck> trucks, String fileName)`: Saves a list of trucks to a file by converting their data to a comma-separated format.

Methods

- `List<Containers.ContainerType> getLoadedContainerTypes()`: Retrieves the types of containers loaded on the truck as a list.
- `double calculateTotalWeightOfLoadedContainers()`: Calculates the total weight of containers loaded on the truck.

"Port" class:

Description

The Port class represents a port within a software system designed for managing ports, container storage, and port-related activities. Ports are essential components for handling the arrival and departure of containers and vehicles. This class encapsulates various attributes, methods, and actions related to ports and their management.

Attributes

- `static final double EARTH_RADIUS_KM`: A static constant representing the Earth's radius in kilometers, used for distance calculations.
- `String id`: A unique identifier for the port.
- `String name`: The name of the port.
- `double latitude`: The latitude coordinate of the port's location.
- `double longitude`: The longitude coordinate of the port's location.
- `double storingCapacity`: The maximum storing capacity of the port for containers.
- `boolean landingAbility`: A boolean indicating whether the port has the ability to handle landing operations.
- `int containerCount`: The count of containers currently stored at the port.
- `int vehicleCount`: The count of vehicles currently at the port.
- `List<Trip> pastTrips`: A list of past trips related to the port.
- `List<Trip> currentTrips`: A list of ongoing trips related to the port.
- `Set<VehicleType> allowedVehicleTypes`: A set of vehicle types that are allowed to operate at the port.
- `PortManager portManager`: A reference to the PortManager responsible for managing this port.

Constructor

- `Port(String id, String name, double latitude, double longitude, double storingCapacity, boolean landingAbility)`: Initializes a new Port object with the provided attributes, such as id, name, latitude, longitude, storingCapacity, and landingAbility.
- `Port()`: An empty default constructor.

Static Methods

- `static List<Port> loadPorts(String filePath, List<PortManager> portManagers)`: Loads a list of ports from a file, parsing data, and associating them with the provided list of PortManager objects.
- `static void savePorts(List<Port> ports, String filePath)`: Saves a list of ports to a text file by formatting their data and writing it to the specified file.
- `static PortManager findPortManagerByUsername(String username, List<PortManager> portManagers)`: Searches for a PortManager object in a list by username and returns it if found.
- `static void deletePort(List<Port> ports, String portId, String filePath)`: Deletes a port from the list of ports based on its portId and updates the list in the data file.
- `static void updatePort(List<Port> ports, String portId, String newName, double newLatitude, double newLongitude, double`

`newStoringCapacity, boolean newLandingAbility, String filePath)`: Updates the properties of a port in the list of ports based on its portId and saves the updated list to the data file.

Static methods

- `static List<Port> getAllPorts(List<Port> ports)`: Returns a list of all ports.
- `static Port getPortByID(List<Port> ports, String portId)`: Retrieves a port from the list based on its portId.

Methods

- `void addPastTrip(Trip trip)`: Adds a trip to the list of past trips associated with the port.
- `void addCurrentTrip(Trip trip)`: Adds a trip to the list of current trips associated with the port.
- `void addContainer()`: Increments the container count for the port.
- `void removeContainer()`: Decrements the container count for the port, if possible.
- `void addVehicle()`: Increments the vehicle count for the port.
- `void removeVehicle()`: Decrements the vehicle count for the port, if possible.
- `void addAllowedVehicleType(VehicleType vehicleType)`: Adds a vehicle type to the set of allowed vehicle types for the port.
- `void removeAllowedVehicleType(VehicleType vehicleType)`: Removes a vehicle type from the set of allowed vehicle types for the port.
- `boolean hasLandingAbilityFor(VehicleType vehicleType)`: Checks if the port has landing ability for a specific vehicle type based on allowed vehicle types and landing ability status.
- `double calculateDistanceTo(Port otherPort)`: Calculates the distance between this port and another port using the Haversine formula.
- `void assignPortManager(PortManager portManager)`: Assigns a PortManager to control this port, provided there is no existing manager.
- `void removePortManager()`: Removes the PortManager from controlling this port.
- `boolean isManagedByPortManager()`: Checks if the port is currently managed by a PortManager.

“Containers” class:

Description

The Containers class represents individual containers used for storing goods or cargo. It provides attributes and methods to manage containers, including loading, saving, updating, and deleting containers from a text file. Additionally, it tracks information about the container, such as its unique ID, weight, type, and the vehicle on which it is loaded.

Attributes

- `String containerId`: A unique identifier for the container.
- `double weight`: The weight of the container.
- `ContainerType type`: The type of the container
- `String loadedOnVehicleId`: A property to track the ID of the vehicle on which the container is loaded.

Constructor

- `Containers(String containerId, double weight, ContainerType type)`: Initializes a new Containers object with the provided container ID, weight, and type. The loadedOnVehicleId is initially set to null, indicating that the container is not loaded on any vehicle.

Static Methods

- `static double calculateTotalWeight(List<Containers> containersList)`: Calculates and returns the total weight of a list of containers.
- `static List<Containers> loadContainersFromFile(String filePath)`: Loads container data from a text file and returns a list of Containers objects.
- `static void saveContainersToFile(String filePath, List<Containers> containersList)`: Saves a list of Containers objects to a text file.
- `static void deleteContainer(String filePath, String containerIdToDelete)`: Deletes a container from the text file based on its ID.
- `static void updateContainer(String filePath, String containerIdToUpdate, double newWeight, ContainerType newType)`: Updates the weight and type of a container in the text file based on its ID.
- `static void addContainer(String filePath, String containerId, double weight, ContainerType type)`: Adds a new container to the text file.

Methods

- `String getContainerId()`: Returns the container's ID.
- `double getWeight()`: Returns the container's weight.
- `ContainerType getType()`: Returns the container's type.
- `String getLoadedOnVehicleId()`: Returns the ID of the vehicle on which the container is loaded.
- `void setWeight(double weight)`: Sets the weight of the container.
- `void setType(ContainerType type)`: Sets the type of the container.
- `void setLoadedOnVehicleId(String loadedOnVehicleId)`: Sets the ID of the vehicle on which the container is loaded.
- `@Override String toString()`: Provides a string representation of the container's attributes.

Enum

- `enum ContainerType`: An enumeration representing the possible types of containers, including DRY_STORAGE, OPEN_TOP, OPEN_SIDE, REFRIGERATED, and LIQUID.

“Trip” class:

Description

The Trip class represents a journey or transportation trip, containing information about the trip's ID, the vehicle used for the trip, departure and arrival dates, departure and arrival ports, and the trip's status. This class facilitates the management and tracking of trips.

Attributes

- `static TripStatus status`: Represents the status of the trip (e.g., PLANNED, IN_PROGRESS, COMPLETED, CANCELED).

- `String tripId`: A unique identifier for the trip.
- `Vehicles vehicle`: The vehicle used for the trip.
- `Date departureDate`: The date and time of departure for the trip.
- `Date arrivalDate`: The date and time of arrival for the trip.
- `Port departurePort`: The port of departure for the trip.
- `Port arrivalPort`: The port of arrival for the trip.

Constructor

- `Trip(String tripId, Vehicles vehicle, Date departureDate, Date arrivalDate, Port departurePort, Port arrivalPort, TripStatus status)`: Initializes a new Trip object with the provided trip ID, vehicle, departure date, arrival date, departure port, arrival port, and trip status.

Static Methods

- `static List<Trip> loadTripsFromFile(String filename)`: Loads a list of Trip objects from a file.
- `static void saveTripsToFile(List<Trip> trips, String filename)`: Saves a list of Trip objects to a file.
- `static boolean updateTripToFile(String filePath, String tripId, Trip updatedTrip)`: Updates a trip in the list based on its trip ID and saves the updated list to a file.
- `static boolean deleteTripFromFile(String filePath, String tripId)`: Deletes a trip from the list based on its trip ID and saves the updated list to a file.

Methods

- `String getTripId()`: Returns the trip's ID.
- `Vehicles getVehicle()`: Returns the vehicle used for the trip.
- `Date getDepartureDate()`: Returns the departure date and time of the trip.
- `Date getArrivalDate()`: Returns the arrival date and time of the trip.
- `Port getDeparturePort()`: Returns the port of departure for the trip.
- `Port getArrivalPort()`: Returns the port of arrival for the trip.
- `TripStatus getStatus()`: Returns the status of the trip.
- `void setStatus(TripStatus status)`: Sets the status of the trip.
- `@Override String toString()`: Provides a string representation of the Trip object, displaying its attributes.

Enum

- `enum TripStatus`: An enumeration representing the possible status values for a trip, including PLANNED, IN_PROGRESS, COMPLETED, and CANCELED.

4. Project Planning Report

All Team Members	Role and Task Given	Individual Contribution (%)
1. Duc Tri Van	Back-end	40%
2. Tran Tuan Bao	Text-based Interface, method implementation and report	30%

3. Nguyen Ich Kiet	Text-based Interface, method implementation and report	30%
4.		

5. Conclusion

In conclusion, our project to develop the Container Port Management System has been a valuable and enlightening experience. Our team was able to successfully design and implement a software system that is both flexible and maintainable. We achieved this by effectively applying the principles of Object-Oriented Programming. By conducting thorough testing and paying close attention to every detail, we made sure that the system's main features are reliable and accurate. Our main focus was on developing a text-based interface that is easy for users to navigate. Additionally, we put a lot of effort into ensuring that our data management and security measures are strong and reliable. The project has given us important skills and hands-on experience, which will help us in future software development opportunities.