CARGO DELİVERY SYSTEM PROJECT CMPE223

Name: Elif
Surname: Konak
ID: 15235113326
Section: 02
Assignment: 01

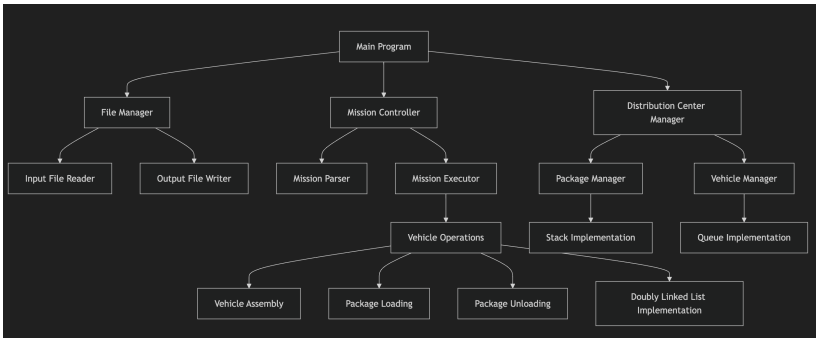
Problem Statement and Code Design

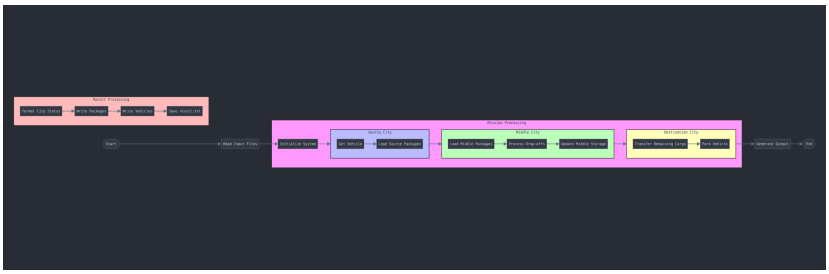
The assignment involves implementing a Java-based logistics system that manages cargo delivery operations between different distribution centers. The system handles packages, vehicles, and missions through various cities using fundamental custom-implemented data structures like doubly linked lists, stacks, and queues.
The system simulates a cargo delivery network where:

1. Multiple distribution centers in different cities handle cargo packages and delivery vehicles
2. Vehicles transport packages between cities following specific mission parameters
3. Package and vehicle management follows specific data structure principles (LIFO for packages, FIFO for vehicles)
4. The system processes missions that define the movement of packages between cities

Main Objectives

1. Implement custom data structures:
   - Doubly linked list
   - Stack (for cargo packages - LIFO)
   - Queue (for delivery vehicles - FIFO)
2. Create a mission execution system that:
   - Assembles vehicles at starting cities
   - Loads and unloads packages at specified locations
   - Follows predetermined routes
   - Manages package distribution
3. Develop file processing capabilities:
   - Read initial configuration from input files
   - Process mission instructions
   - Generate result files showing final distribution status





Module Descriptions

- Main Program
  - Coordinates overall program flow
  - Initializes system components
  - Manages simulation execution
- File Manager
  - Handles input file reading (cities, packages, vehicles, missions)
  - Manages output file generation
  - Validates file formats
- Mission Controller
  - Parses mission instructions
  - Controls mission execution sequence
  - Validates mission parameters
- Distribution Center Manager
  - Manages city distribution centers
  - Controls package and vehicle inventory
  - Handles resource allocation
- Data Structure Implementations
  - Custom implementation of required data structures
  - Manages data organization and access
  - Ensures proper LIFO/FIFO operations

Implementation and Functionality

CargoPackage Class Implementation and Functionality

```
CLASS CargoPackage IMPLEMENTS Comparable<Cargo>
    VARIABLES:
        id (STRING)
    CONSTRUCTOR:
        INPUT: id (STRING)
        OUTPUT: A Cargo package object with the given ID
        IF id is null OR id is empty THEN
          THROW IllegalArgumentException("ID cannot be null or
empty.")
        SET this.id to id
   METHODS:
        FUNCTION getId()
            OUTPUT: String representing the ID of the cargo
            RETURN this.id
        FUNCTION compareTo(other)
            INPUT: other (Cargo)
            OUTPUT: Integer result of comparing this cargo with
another cargo by ID
            IF other is null THEN
                THROW NullPointerException("The other Cargo object
is null.")
            RETURN this.id.compareTo(other.id)
FUNCTION equals(obj)
            INPUT: obj (Object)
            OUTPUT: Boolean indicating if this Cargo is equal to
another object
            IF this is the same object THEN
                RETURN true
            IF obj is null OR obj is not of type Cargo THEN
                RETURN false
            RETURN this.id.equals(((Cargo) obj).id)
        FUNCTION hashCode()
            OUTPUT: Integer hash code of this Cargo object based
on its ID
            RETURN this.id.hashCode()
        FUNCTION toString()
            OUTPUT: String representation of the Cargo object
            RETURN this.id
END CLASS
```

City Class Implementation and Functionality

```
 CLASS City
    VARIABLES:
        name (STRING)
        cargos (STACK OF Cargo)
        vehicles (DOUBLY LINKED LIST OF Vehicle)
    CONSTRUCTOR:
        INPUT: name (STRING)
        OUTPUT: A City object with an empty stack of
        Cargos and an empty queue of vehicles
        SET this.name to name
        SET this.cargos to a new empty Stack of Cargo
        SET this.vehicles to a new empty Doubly Linked List of
 Vehicle
    METHODS:
        FUNCTION getName()
            RETURN this.name
        FUNCTION addCargo(cargo)
            INPUT: cargo (Cargo)
             OUTPUT: None
             CALL cargos.push(cargo)
         FUNCTION addVehicle(vehicle)
             INPUT: vehicle (Vehicle)
             OUTPUT: None
             CALL vehicles.addLast(vehicle)
         FUNCTION removeCargo()
             OUTPUT: Cargo object or null if stack is empty
             IF cargos is empty THEN
                 RETURN null
            RETURN cargos.pop()
         FUNCTION removeVehicleQueue()
             OUTPUT: Vehicle object or null if queue is empty
             IF vehicles is empty THEN
              RETURN null
            RETURN vehicles.removeFirst()
      FUNCTION getCargos()
            OUTPUT: Array of Cargo objects
            SET cargoArray to a new array of size cargos.size
            SET tempStack to a new empty Stack of Cargo
            WHILE cargos is not empty
                CALL tempStack.push(cargos.pop())
            SET i to 0
            WHILE tempStack is not empty
                SET cargo to tempStack.pop()
                SET cargoArray[i] to cargo
                CALL cargos.push(cargo)
                INCREMENT i
            RETURN cargoArray
FUNCTION isCargoEmpty()
            OUTPUT: Boolean indicating if the stack of cargos is
empty
            RETURN cargos.isEmpty()
        FUNCTION isVehicleEmpty()
            OUTPUT: Boolean indicating if the queue of vehicles is
empty
            RETURN vehicles.isEmpty()
      END CLASS
```

CityList Class Implementation and Functionality

```
CLASS CityList
  VARIABLES:
    cities (LIST OF City)
  CONSTRUCTOR:
    CityList()
      Initialize cities as an empty list
  METHODS:
    addCity(city)
      IF city is not null THEN
          ADD city to cities list
    findCity(cityName)
      FOR EACH city in cities
          IF city.getName() equals cityName THEN
              RETURN city
      RETURN null
   getCities()
      RETURN a copy of the cities list
    containsCity(sourceCity)
      FOR EACH city in cities
          IF city.getName() equals sourceCity THEN
              RETURN true
      RETURN false
```

Mission Class Implementation and Functionality

```
CLASS Mission
  VARIABLES:
    sourceCity (STRING)
    middleCity (STRING)
    destCity (STRING)
    totalSourcePackages (INTEGER)
    totalMiddlePackages (INTEGER)
    dropIndices (ARRAY OF INTEGER)
  CONSTRUCTOR:
    Mission(sourceCity, middleCity, destCity, totalSource
Packages, totalMiddlePackages, dropIndices)
     Initialize mission variables
 METHODS:
    getSourceCity()
      RETURN sourceCity
    getMiddleCity()
      RETURN middleCity
    getDestCity()
      RETURN destCity
    getTotalSourcePackages()
      RETURN totalSourcePackages
    getTotalMiddlePackages()
      RETURN totalMiddlePackages
    getDropIndices()
      RETURN dropIndices
```

DoublyLinkedList Class Implementation and Functionality

```
CLASS DoublyLinkedList
  VARIABLES:
    head (NODE)
    tail (NODE)
    size (INTEGER)
  CONSTRUCTOR:
    DoublyLinkedList()
      Initialize head and tail to null
      Initialize size to 0
  METHODS:
    addNode(node)
      IF head is null THEN
          SET head to node
          SET tail to node
      ELSE
          SET tail.next to node
          SET node.prev to tail
          SET tail to node
      INCREMENT size
    removeNode(node)
      IF node is head THEN
          SET head to node.next
      ELSE IF node is tail THEN
          SET tail to node.prev
      ELSE
          SET node.prev.next to node.next
          SET node.next.prev to node.prev
      DECREMENT size
    getNode(index)
      IF index is 0 THEN
          RETURN head
      ELSE IF index is size - 1 THEN
          RETURN tail
      ELSE
          START at head
          FOR i from 0 to index - 1
              MOVE to next node
          RETURN current node

getSize()
      RETURN size
```

Stack Class Implementation and Functionality

```
CLASS Stack
  VARIABLES:
    elements (DOUBLY LINKED LIST)
  CONSTRUCTOR:
    Stack()
      Initialize elements to an empty DoublyLinkedList
  METHODS:
    push(element)
      ADD element to top of elements list
    pop()
      REMOVE top element from elements list
      RETURN removed element
    peek()
      RETURN top element of elements list
    isEmpty()
      RETURN true if elements list is empty, false otherwise
    size()
      RETURN number of elements in elements list
```

Queue Class Implementation and Functionality

```
CLASS Queue
  VARIABLES:
    elements (DOUBLY LINKED LIST)
  CONSTRUCTOR:
    Queue()
      Initialize elements to an empty DoublyLinkedList
  METHODS:
    enqueue(element)
      ADD element to end of elements list
    dequeue()
      REMOVE front element from elements list
      RETURN removed element
    peek()
      RETURN front element of elements list
    isEmpty()
      RETURN true if elements list is empty, false otherwise
    size()
      RETURN number of elements in elements list
```

Main Class Implementation and Functionality

```
CLASS Main
  METHODS:
    main()
      READ test scenarios from file
      FOR EACH test scenario
          READ cities from file
          READ packages from file
          READ vehicles from file
          READ missions from file
          EXECUTE missions

    readCities(citiesFile, cityList)
      READ cities from file
      FOR EACH city
          ADD city to cityList
    readPackages(packagesFile, cityList)
      READ packages from file
      FOR EACH package
          FIND city in cityList
          ADD package to city's cargo stack
    readVehicles(vehiclesFile, cityList)
      READ vehicles from file
      FOR EACH vehicle
          FIND city in cityList
          ADD vehicle to city's vehicle queue
    readMissions(missionsFile, cityList)
      READ missions from file
      FOR EACH mission
          EXECUTE mission using cityList

  performMission(sourceCity, middleCity, destCity,
  totalSourcePackagestotalMiddlePackages, dropIndices, cityList)
  FIND source city, middle city, and destination city in cityList
  REMOVE vehicles and packages from source city
  EXECUTE mission using middle city and destination city
```

Vehicle Class Implementation and Functionality

```
CLASS Vehicle EXTENDS CargoPackage
  VARIABLES:
    city (STRING)
    capacity (DOUBLE)
    cargoList (DoublyLinkedList<CargoPackage>)
    id (STRING) // inherited from CargoPackage
  CONSTRUCTOR:
    Vehicle(id, city, capacity)
      CALL super(id)
      Initialize city and capacity
      Initialize cargoList as new DoublyLinkedList
  METHODS:
    compareTo(other: CargoPackage)
      RETURN comparison of this.capacity with other.capacity
    getId()
      RETURN id
    getCity()
      RETURN city
    getCapacity()
      RETURN capacity
    setCity(city)
      SET this.city to city
    setCapacity(capacity)
      SET this.capacity to capacity
    isEmpty()
      RETURN cargoList.isEmpty()
    addCargo(cargo: CargoPackage)
```

```
    ADD cargo to cargoList at last position
  removeCargo()
    IF cargoList is not empty THEN
      RETURN removed last cargo from cargoList
    ELSE
      RETURN null
  removeCargoAtIndex(dropIndex: INTEGER)
    IF cargoList is not empty THEN
      RETURN removed cargo at dropIndex from cargoList
    ELSE
      RETURN null
  toString()
    RETURN formatted string with id, city, and capacity
  getVehicleCargoCount()
    RETURN size of cargoList
```

Testing

The testing implementation includes comprehensive components:

Data Structure Tests
- DoublyLinkedList operations
- Stack operations
- Queue operations

Component Tests
- DistributionCenter functionality
- CityList operations
- Vehicle and Package management

Integration Tests
- End-to-end mission execution
- File I/O operations
- System state consistency

Edge Cases
- Empty collections
- Invalid operations
- Boundary conditions

Test 1 File Contentes: The folder SampleIO Set1 on LMS

Tests basic package delivery functionality, including multi-city transport, ensuring vehicles use their capacity correctly and packages arrive at the designated cities.,
The result.txt created by the tester and main class in the project is compared with the expectedResult.txt I uploaded, and if there is no difference or error, the test is written as passed.

Output of the test code in the project:

```
==============================
Running Test Scenario 1:
==============================
🚀 Starting test for folder: input_output/test1/
==============================

🎉 No errors found. Logging full city details.

Writing results to: /Users/konak/Downloads/CargoDeliverySystem/input_output/test1/result.txt

✓ TEST PASSED: input_output/test1/result.txt matches expected result.
✅ Test passed for: input_output/test1/ ✅
```

Test 2 File Contentes: The folder SampleIO Set2 on LMS

Checks if packages are distributed accurately to different cities within a mission and verifies each city holds the correct number of packages and vehicles.
The result.txt created by the tester and main class in the project is compared with the expectedResult.txt I uploaded, and if there is no difference or error, the test is written as passed.

Output of the test code in the project:

```
==============================
Running Test Scenario 2:
==============================
🚀 Starting test for folder: input_output/test2/
==============================

🎉 No errors found. Logging full city details.

Writing results to: /Users/konak/Downloads/CargoDeliverySystem/input_output/test2/result.txt

✓ TEST PASSED: input_output/test2/result.txt matches expected result.
✅ Test passed for: input_output/test2/ ✅
```

Test 3 File Contents:

cities.txt
Berlin
Hamburg
vehicles.txt
V1 Berlin 10.0
V2 Hamburg 4.5
packages.txt
P1 Berlin
P2 Hamburg
missions.txt
Berlin-Hamburg-Munich-2-1-1

Evaluates if the system generates **appropriate error messages** when encountering a city mentioned in a mission but not defined in the cities.txt file. This determines if the code verifies correct data input and reacts appropriately to errors.

Output of the test code in the project

```
==============================
Running Test Scenario 3:
==============================
🚀 Starting test for folder: input_output/test3/
==============================

💥 Issues detected during mission execution. Saving error log only.

Writing results to: /Users/konak/Downloads/CargoDeliverySystem/input_output/test3/result.txt

✓ TEST PASSED: input_output/test3/result.txt matches expected result.
✅ Test passed for: input_output/test3/ ✅
```

Test 4 File Contents:

cities.txt
Istanbul
Ankara
Izmir
vehicles.txt
V1 Ankara 5.0
V2 Istanbul 6.5
V3 Istanbul 3.0
V4 Izmir 4.2
packages.txt
P1 Istanbul
P2 Istanbul
P3 Ankara
P4 Istanbul
P5 Izmir
P6 Izmir
P7 Ankara
P8 Izmir
P9 Istanbul
P10 Izmir
missions.txt
Istanbul-Ankara-Izmir-4-1-1,2

Confirms a mission is successfully executed across different cities and that the vehicle and package counts within those cities are updated accordingly at the mission's end. This aims to ensure all data is updated accurately after package transportation.
The result.txt created by the tester and main class in the project is compared with the expectedResult.txt I uploaded, and if there is no difference or error, the test is written as passed.

Output of the test code in the project:

```
==============================
Running Test Scenario 4:
==============================
🚀 Starting test for folder: input_output/test4/
==============================

🎉 No errors found. Logging full city details.

Writing results to: /Users/konak/Downloads/CargoDeliverySystem/input_output/test4/result.txt

✓ TEST PASSED: input_output/test4/result.txt matches expected result.
✅ Test passed for: input_output/test4/ ✅
```

Trouble Points

1. **Concurrent Data Structure Management**
   - Maintaining consistency between different data structures (Stack, Queue, DoublyLinkedList) while executing missions was challenging
   - Ensuring proper synchronization of package and vehicle states across different centers required careful planning
   - Managing indices during package drops needed precise implementation
2. **File Processing Logic**
   - Parsing complex mission formats with multiple parameters
   - Handling various edge cases in input validation
   - Maintaining proper error handling for file operations

Most Challenging Parts

1. **Mission Execution Algorithm**
   - Implementing the complex logic for package loading and unloading
   - Managing the correct order of operations during mission execution
   - Handling edge cases in the delivery sequence
2. **Data Structure Integration**
   - Ensuring proper interaction between different data structures
   - Maintaining consistency across operations
   - Implementing efficient traversal and modification operations

Learning Outcomes and Positive Aspects

1. **Design Patterns**
   - Gained practical experience in implementing generic data structures
   - Learned about inheritance and polymorphism through Vehicle/CargoPackage relationship
   - Understood the importance of proper encapsulation and modularity
2. **Testing Methodology**
   - Developed comprehensive testing strategies
   - Learned about edge case handling and error scenarios
   - Gained experience in writing maintainable and testable code
3. **Project Organization**
   - Improved skills in organizing large-scale Java projects
   - Learned about proper documentation practices
   - Gained experience in file I/O and error handling