Flight Booking System Specification

Main.java

package bcu.cmp5332.bookingsystem.main;

The Main class for the application. The `main()` method of the class gets executed when the application is started. The method first loads the stored data from the file storage (resources/ data directory) to a `FlightBookingSystem` object using the `FlightBookingSystemData` class. Then it displays a prompt and waits for the user's input at the command line. When a command is given by the user, it gets passed to the `CommandParser` class to be parsed and checked for its validity. If the command is invalid the `CommandParser` throws a `FlightBookingSystemException` which is caught, and its message gets displayed to the user. If the command is valid, the `CommandParser` returns an implementation of the Command interface which is executed. If the command "`exit`" is given by the user, the method stores all data from the Library object to the file storage and then exits.

CommandParser.java

package bcu.cmp5332.bookingsystem.main;

`CommandParser` class is used to parse the input given by the user through the application's command line interface. The class has a static method `parse(String)` that gets the user input as String and returns the corresponding implementation of the Command interface based on that input. The input (variable line) contains the command as the first part (`parts[0]`) and the possible arguments that were given by the user (`parts[1]`, `parts[2]`, `etc.`).
Complete the parse method to return all the appropriate Command objects based on the user input (e.g. return the `ShowCustomer` Command if the user types "`showcustomer`").

FlightBookingSystemException.java

package bcu.cmp5332.bookingsystem.main;

A custom checked exception that is used to notify the user about errors or invalid commands.

FlightBookingSystemData.java

package bcu.cmp5332.bookingsystem.data;

This class is responsible for loading data to and storing data from the flight booking system, using the text file storage.
The class has a static list of objects that implement the `DataManager` interface. The list gets populated by a static block of code (runs only once when the object is loaded to memory) that adds different implementations of the `DataManager` interface - one for each different entity of the library system (e.g. `FlightDataManager` is an implementation that loads and stores data for Flight entities using the text file storage).
The class has two static methods:

- **`load()`** method loads data from all `DataManagers` to the `FlightBookingSystem`. It iterates the list with `DataManager` objects and executes their implementation of `loadData(FlightBookingSystem fbs)` method to load data for all the entities from the file storage (txt files) to a `FlightBookingSystem` object and returns that object.
- **`store(FlightBookingSystem fbs)`** method stores all data from the `FlightBookingSystem` to the text file storage. It iterates the list with `DataManager` objects and executes their implementation of `storeData(FlightBookingSystem fbs)` method to store data from a `FlightBookingSystem` object to the file storage (txt files).

The implementation of the `FlightDataManager` is provided and it can be used as a guide to complete the implementation of `CustomerDataManager` and `BookingDataManager` classes. **You have to uncomment** lines 21 and 22 of the `FlightBookingSystemData` class (inside the static block) when you complete the implementation of `CustomerDataManager` and `BookingDataManager` classes, so they are added to the list of `DataManagers` and their methods get executed.

DataManager.java

package bcu.cmp5332.bookingsystem.data;

This interface declares the `loadData(FlightBookingSystem fbs)` and `storeData(FlightBookingSystem fbs)` methods that are implemented by different `DataManager` classes (`FlightDataManager`, `CustomerDataManager`, `BookingDataManager`) to load data to the flight booking system from the file storage and store data from the flight booking system to the file storage.
The class also has a property `SEPARATOR` which holds the value of the separator that is used to separate the different attributes of the entities when stored in the file storage. The separator value is assigned as "::".

FlightDataManager.java

package bcu.cmp5332.bookingsystem.data;

This class is used to load and store data for Flight objects (entities) using the text file storage (txt files).
This class is already implemented and can be used a s a guide for the implementation of the `CustomerDataManager` and `BookingDataManager` classes.

CustomerDataManager.java

package bcu.cmp5332.bookingsystem.data;

This class is used to load and store data for Customer objects (entities) using the text file storage (txt files).
You have to implement the `loadData(FlightBookingSystem fbs)` and `storeData(FlightBookingSystem fbs)` methods using the already implemented `FlightDataManager` class as guide. Don't forget to uncomment line 21 of the `FlightBookingSystemData` class (inside the static block) for the `CustomerDataManager` to be added in the List of `DataManagers` and have its methods executed (see class `FlightBookingSystemData` above).

BookingDataManager.java
package bcu.cmp5332.bookingsystem.data;

This class is used to load and store data for Booking objects (entities) using the text file storage (txt files).
You have to implement the `loadData(FlightBookingSystem fbs)` and `storeData(FlightBookingSystem fbs)` methods using the already implemented `FlightDataManager` class as guide. Don't forget to uncomment line 22 of the `FlightBookingSystemData` class (inside the static block) for the `BookingDataManager` to be added in the List of `DataManagers` and have its methods executed (see class `FlightBookingSystemData` above).

Flight.java
package bcu.cmp5332.bookingsystem.model;

This class models a flight in the flight booking system. The flight has fields for different book properties such as flight number, origin, destination, and departure date and also has an id field that represents the flight's unique id in the system. Each flight object should have a unique combination of flight number and departure date. This means that there shouldn't be two flights with the same flight number and departure date in the system.

In addition, this class has a "`passengers`" field of type Set, which holds the unique passengers (Customers) for a particular flight. These are the customers that have made a booking to travel with this flight.

The getter and setter methods of the class are already implemented to get and set the values of the different properties. You have to implement:

- the `getDetailsLong()` method to print the details of a flight object (all properties and the list of passengers; see example below)

Flight `getDetailsLong()` example output:

```
Flight #1
Flight No: LH2560
Origin: Birmingham
Destination: Munich
Departure Date: 2020-11-25
---------------------------
Passengers:
 * Id: 2 - Kostas Vlachos - 07596454545
1  passenger(s)
```

- the `addPassenger(Customer passenger)` and `removePassenger(Customer passenger)` methods that add or remove passengers from the "passengers" collection (Set). Both methods are void.

The `addPassenger(Customer passenger)` method should throw a `FlightBookingSystemException` if the passenger to be added is already present in the flight's list of passengers.

The `removePassenger(Customer passenger)` method should throw a `FlightBookingSystemException` if the passenger to be removed is not contained in the flight's list of passengers.

Customer.java

package bcu.cmp5332.bookingsystem.model;

This class models a customer (also passenger) of the flight booking system.
The class has fields for different customer properties such as name and phone number and has an id field that represents the customer's unique id in the system. In addition, the class has a list of the bookings that a customer has made.
You must implement the constructor and getter and setter methods to get and set the values of the different customer properties. In addition, you have to write:

- `getDetailsShort()` and `getDetailsLong()` methods which return the details of a customer. Both methods return a String; `getDetailsShort()` method should return the basic properties of the customer **in one line** and `getDetailsLong()` should return the list of bookings that a customer has made in addition to the customer's attributes (see example below).

  Customer `getDetailsLong()` example output:

```
Customer #2
Name: Kostas Vlachos
Phone: 07596454545
--------------------------
Bookings:
 * Booking date: 2020-11-16 for Flight #1 - LH2560 - Birmingham to
Munich on 25/11/2020
1 booking(s)
```

- `addBooking(Booking booking)` method adds a booking to the customer's list of bookings. The method is void and throws a `FlightBookingSystemException` if the customer's list of bookings already contains a booking for the same flight.
- `cancelBookingForFlight(Flight flight)` method finds the booking for the given flight in the customer's list of bookings and then removes it from the list. The method is void and should throw a `FlightBookingSystemException` if the customer's list of bookings doesn't contain a booking for the flight that is given as parameter.

Booking.java

package bcu.cmp5332.bookingsystem.model;

This class models a booking made by a Customer for a particular Flight in the flight booking system.
The class has a `Customer` and a `Flight` property as well as a `LocalDate` property to hold the date that the booking was made from the customer.

You must implement the constructor and getter and setter methods to get and set the values of the different booking properties.

FlightBookingSystem.java

package bcu.cmp5332.bookingsystem.model;

This class models the entire flight booking system.
The class has two `TreeMap` collections to store the Customers and Flights of the system using their ids as key. The class also has a `systemDate` property which can be used to change the date of the system (e.g. set it in the future) to be able to check some of the system's functionalities (e.g. hiding flights with past departure dates or calculating different fares based on the date that the booking was made). To use this `systemDate` you should call the class `getSystemDate` method.
The following methods are already implemented:

- `getFlights()` that returns an unmodifiable List of all the Flights in the system
- `getFlightById(int id)` that finds and returns a Flight using the flight id
- `addFlight(Flight flight)` that adds a new Flight to the flights `TreeMap`

You have to implement:

- `getCustomers()` method that returns an unmodifiable List of all the Customers in the system
- `getCustomerById(int id)` method that finds a Customer in the system's collection of customers using the customer's id and returns this Customer. The method throws a `FlightBookingSystemException` if there is no customer with the given id in the system.
- `addCustomer(Customer customer)` method which adds a new Customer to the collection (TreeMap) of customers. The method is void and should throw an `IllegalArgumentException` if the id of the Customer object to be added already exists in the TreeMap. (Note: when the email property gets added to the Customer class for the 50-59% mark band, the method should also throw a `FlightBookingSystemException` if a Customer with the same email already exists in the TreeMap).

Command.java

package bcu.cmp5332.bookingsystem.commands;

The Command interface models an action that occurs as a result of a command given by the user.
The interface provides the `execute(FlightBookingSystem fbs)` function which must be implemented by all the objects that implement the Command interface. The function takes as argument a `FlightBookingSystem` object containing all the Flights and Customers that exist in the flight booking system and uses it to make the changes instructed from the command given by the user (e.g. get Flights by using their id, adding new Flights to the system, make Bookings, etc.).
All classes that implement this interface, can have a constructor to initialise additional properties (if needed) for the execution of the command. See the implemented `AddFlight` class as an example of a class that needs a constructor, and `ListFlights` class as an example of one that doesn't.

The interface has a `HELP_MESSAGE` field that holds the message that is shown to the user when the Help Command gets executed.

AddFlight.java

package bcu.cmp5332.bookingsystem.commands;

This class is an implementation of the Command interface and its `execute(FlightBookingSystem fbs)` method adds a new Flight to the system. It is created by the `CommandParser` class when the "`addflight`" command is given by the user. This class is already implemented.
The class has a constructor to initialise the fields (properties) needed to execute the command (e.g. flight number, departure date, etc.).
The `execute(FlightBookingSystem fbs)` method generates a new unique id for the flight and then creates a new Flight object, using this id and the information given by the user (flight number, origin, destination, and departure date) via the command line. Finally, the method adds this Flight to the system by using the appropriate method of the `FlightBookingSystem` class and prints a notification to the user that the Flight was successfully added.

ListFlights.java

package bcu.cmp5332.bookingsystem.commands;

This class is an implementation of the Command interface and its `execute(FlightBookingSystem fbs)` method displays a list of the flights stored in the system to the user. At the end of the list, the total amount of flights in the system is also displayed.
It is created by the `CommandParser` class when the "`listflights`" command is given by the user. This class is already implemented.
The `execute(FlightBookingSystem fbs)` method iterates the list of Flights stored in the system and prints their details using one line per flight (`getDetailsShort()` method of Flight class) as well as the total number of flights.

Help.java

package bcu.cmp5332.bookingsystem.commands;

This class is an implementation of the Command interface and its `execute(FlightBookingSystem fbs)` method prints the `Command.HELP_MESSAGE` to the console.
It is created by the `CommandParser` class when the "help" command is given by the user. This class is already implemented.

AddCustomer.java
package bcu.cmp5332.bookingsystem.commands;

This class is an implementation of the Command interface and its `execute(FlightBookingSystem fbs)` method adds a new Customer to the system.
It is created by the `CommandParser` class when the "addcustomer" command is given by the user.
The class has a constructor to initialise the fields (properties) needed to execute the command (e.g. customer name, phone number, etc.).
You have to implement the `execute(FlightBookingSystem fbs)` method that adds a new `Customer` to the system. The method should generate a new unique id for the customer and then create a new `Customer` object, using this id and the information given by the user (name, phone) via the command line. Finally, the method should add this new `Customer` object to the system by using the appropriate method of the `FlightBookingSystem` class and print a notification to the user that the Customer was successfully added.

ListCustomers.java
package bcu.cmp5332.bookingsystem.commands;

This class is an implementation of the Command interface and its `execute(FlightBookingSystem fbs)` method displays a list of the customers stored in the system to the user. At the end of the list, the total amount of customers in the system is also displayed.
It is created by the `CommandParser` class when the "listcustomers" command is given by the user.
You have to implement the `execute(FlightBookingSystem fbs)` method of the class. The method should iterate the list of Customers stored in the system and print their details using one line per customer (using the `getDetailsShort()` method of `Customer` class). The total number of customers should be also printed at the bottom of the list.

ShowFlight.java
package bcu.cmp5332.bookingsystem.commands;

This class is an implementation of the Command interface and its `execute(FlightBookingSystem fbs)` method displays details about a specific flight to the user.
It is created by the `CommandParser` class when the "showflight" command is given by the user.
You should implement a constructor to initialise the fields needed to execute the command. You also have to implement the `execute(FlightBookingSystem fbs)` method of the class. The method should get the Flight from the flight booking system using the id which is given from the user as part of the command “showflight [flight id]”, and then print its details using the appropriate method of the Flight class.

ShowCustomer.java
package bcu.cmp5332.bookingsystem.commands;

This class is an implementation of the Command interface and its `execute(FlightBookingSystem fbs)` method displays details about a specific customer to the user.
It is created by the `CommandParser` class when the "showflight" command is given by the user.
You should implement a constructor to initialise the fields needed to execute the command. You also have to implement the `execute(FlightBookingSystem fbs)` method of the class. The method should get the Customer from the flight booking system using the id which is given from the user as part of the command "`showcustomer [customer id]`", and then print its details using the appropriate method of the Customer class.

AddBooking.java
package bcu.cmp5332.bookingsystem.commands;

This class is an implementation of the Command interface and its `execute(FlightBookingSystem fbs)` method creates a new Booking in the system.
It is created by the `CommandParser` class when the "addbooking" command is given by the user.
You should implement a constructor to initialise the fields needed to execute the command. You also have to implement the `execute(FlightBookingSystem fbs)` method of the class. The method should first get the Customer and Flight from the flight booking system using their ids which are given by the user as part of the command "addbooking [customer id] [flight id]". Then the method should create a new Booking object using the Customer, the Flight and the date of booking. This Booking should be added to the Customer's list of Bookings using the appropriate method of the Customer class and the Customer should be added to the Flight's collection (HashSet) of passengers. Finally, the method should display a message to the user that the Booking was issued successfully to the customer.

CancelBooking.java
package bcu.cmp5332.bookingsystem.commands;

This class is an implementation of the Command interface and its `execute(FlightBookingSystem fbs)` method cancels a Booking made by a Customer in the system.
It is created by the `CommandParser` class when the "cancelbooking" command is given by the user.
You should implement a constructor to initialise the fields needed to execute the command. You also have to implement the `execute(FlightBookingSystem fbs)` method of the class. The method should first get the Customer and Flight from the flight booking system using their ids which are given by the user as part of the command "`cancelbooking [customer id] [flight id]`". Then the method should remove the Booking for this Flight object from the Customer's list of Bookings using the

appropriate method of the Customer class. Next, the method should remove the Customer from the Flight's collection (Set) of passengers using the appropriate method of the Flight class. Finally, a message should be displayed to the user to notify that the booking was cancelled successfully.

LoadGUI.java

package bcu.cmp5332.bookingsystem.commands;

This class is an implementation of the Command interface and its `execute(FlightBookingSystem fbs)` method launches the GUI of the application.
It is created by the `CommandParser` class when the "loadgui" command is given by the user. This class is already implemented.