Enrique Munoz

2/13/2023

CS 3331 – Advanced Object-Oriented Programming – Spring 2023

Dr Mejia

PA1 MinerAir

This work was done individually and completely on my own. I did not share, reproduce, or alter any part of this assignment for any purpose. I did not share code, upload this assignment online in any form, or view/received/modified code written from anyone else. All deliverables were produced entirely on my own. This assignment is part of an academic course at The University of Texas at El Paso and a grade will be assigned for the work I produced.

Program Explanation

The objective of PA1 was to design a flight system where a customer could interact and buy tickets from a specific flight using Object-Oriented programming design. From PA0 to PA1, I had to include the new classes in order to create more interaction between the Flight class and an individual using the system. Creating new classes such as a person, customer, and ticket classes which were the most essential in this new lab. The other classes that I created had no use and would be used for later projects. I tackled the problem by first setting up the classes with the attributes and methods I thought were necessary. As I progressed through the project, I realized I had to change and add some extra attributes and methods, but it was much easier to understand these changes as I went on. I relied on going through each part of the project's instructions, and cleaning and optimizing my code as I went through each step. Basically, breaking the problem down and following a specific set of instructions than just going through and trying to finish the classes without having the right applications.

What did I learn?

After finishing PA1, I learned more about class abstraction, inheritance, and shared aggregation. The person class was abstract and the customer inherited from this class. Taking the first and last name of the person class as its own attribute since the customer would also be considered a person but almost like a specific genre of a person. Meaning that it needed all of the attributes that a person has too but becomes separated as it also includes the attributes of a customer. A ticket object also needed to be present in the flight class and customer class. Every time a customer would buy a ticket, a ticket object would have to be added to both the customer and a specific flight object. This was an example of aggregation as a person needs a ticket and a flight object also has a ticket. These classes can exist without being part of one another, meaning that they don't need the whole. My solution can be improved by trying to divide each flight object

into its own category. Making the flight class abstract and using the domestic and international classes would demonstrate more of an Object-Oriented design. The csv file needs to specify a country in order to complete this. Separating the file into international and domestic flights would make this solution possible since we wouldn't know which flight is international or domestic without its country. It took me about 9 hours to complete this program.

Solution Design

Creating the classes was fairly simple, all I had to do was incorporate a person, customer, and ticket class. I added the flight class from the previous programming assignment as I would need to utilize it for the customer to buy tickets from a specific flight object. Initially, I did not know which attributes to create for these classes so I started with my own intuition, creating classes that I thought were needed. As I went forward, I started seeing new ways to implement some attributes and methods. Especially in the customer class, I realized that some of the initial attributes I created were not needed and I had also not incorporated some of the essential attributes. Going step by step allowed me to clean and optimize my code as I mentioned in the programming explanation. With each new step, I incorporated something new into my code. In the beginning, I had used an Array List to store my customer and flight objects, but I realized that this wasn't the most efficient data structure to use in this case. First, I wouldn't need to add new flight objects since I was reading off of a csv file of flights. Second, I would need to access a specific flight object as fast as possible, meaning that a HashMap was the best data structure in this case. I would incorporate the HashMap by using the flight ID as the key and the flight object as the value and customer name and last name with the value being the customer object for the customer HashMap. Finding and accessing a specific flight object would be in O(1) compared to O(N) in an Array List as I would have to go through each flight object and check if the flight ID matches the user's input. On the other hand, since I needed to store tickets in the flight and customer classes, I utilized an Array List. This time I was going to access a specific ticket and was instead going to add ticket objects to this Array List. To create the user menu I created two methods, one was a login menu, and the other was a flight menu where the user interacts with the flight class after logging in. The only roadblock I came across was in the creation of a buffered writer in order to update a log text file with the interactions of the customer. Since the creation of a buffered writer needs an exception to be thrown, and needs to write each interaction, the try and catch block would have to be incorporated in the whole main method. Any object such as a file reader of in this case the buffered writer cannot be referenced outside of the try and catch block. In order to fix this, I utilized the try and catch block in the flight menu and ended up fixing the problem of trying to reference the buffered writer outside of this specific block.

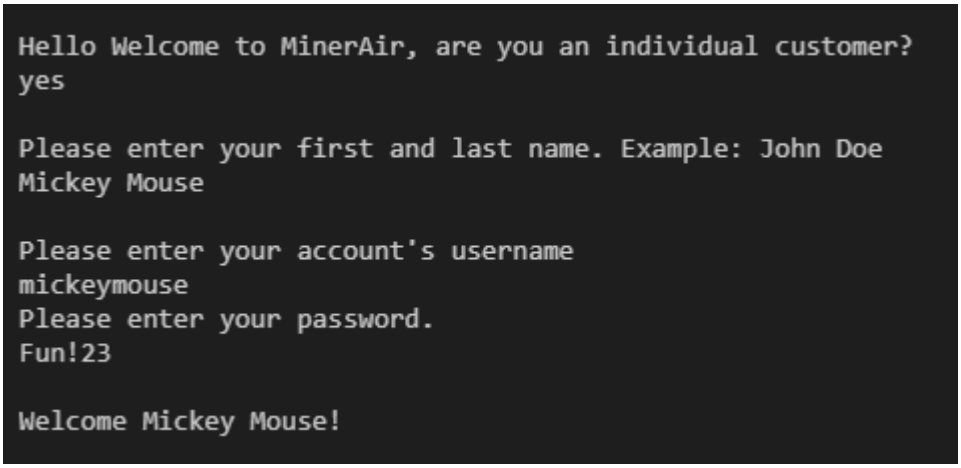
Testing

I tested my program by using both black-box and white-box testing. Specifically, in the creation of customer interaction, I utilized black-box testing. I used different inputs such as customer names or names that didn't even exist in the customer HashMap. I observed the output that my

specific inputs would give me and either kept my code the same or revised my code when it did not give me the output that was required. Logging the customers' interaction white box testing. I tried many different attempts of solving the logging problem I was having by delicately looking at my code, and seeing if I had any exceptions that needed to be thrown or dead code. I tried referencing the buffered writer outside of the try and catch block which would run me into errors. I had to search through documentation and look closely at my code to fix this problem.

Test Results

Test of User Correct Inputs in Login Menu



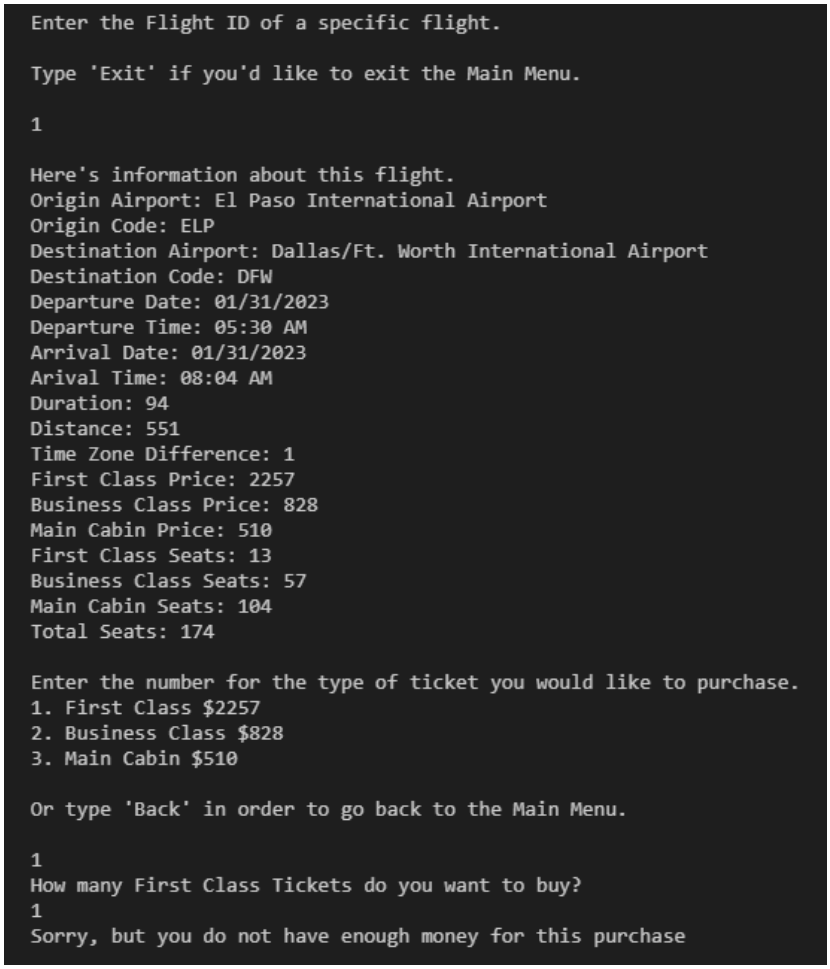
- The user here uses correct inputs to login into their account. Each input is met with the correct response and output.

Test of User Incorrect Inputs in Login Menu

```
Hello Welcome to MinerAir, are you an individual customer?
no
Sorry, this system is for individual customers only.

You have been logged out of your account.
Type 'Exit' if you wish to end the program, or type 'Enter' to login again.

enter

Hello Welcome to MinerAir, are you an individual customer?
yes

Please enter your first and last name. Example: John Doe
Joe
Unfortunately, this name is not in our database. Please try again.

Please enter your first and last name. Example: John Doe
Mickey Mouse

Please enter your account's username
mick
Incorrect username, please try again.
mickeymouse
Please enter your password.
a
Please enter your password.
b
Please enter your password.
c
Max number of password attempts has been reached please try again later.

You have been logged out of your account.
Type 'Exit' if you wish to end the program, or type 'Enter' to login again.
```

- In this example, the user used incorrect inputs in the login menu. They first specify that they aren't individual customers, and the program exits them out but still allows them to try to log in. After they use an incorrect name and the program meets them with the right response, doing the same when they input an incorrect username. When they put their password incorrectly three times in a row, they are logged out of that account and are forced to restart the process again.

Customer Cannot Afford Ticket

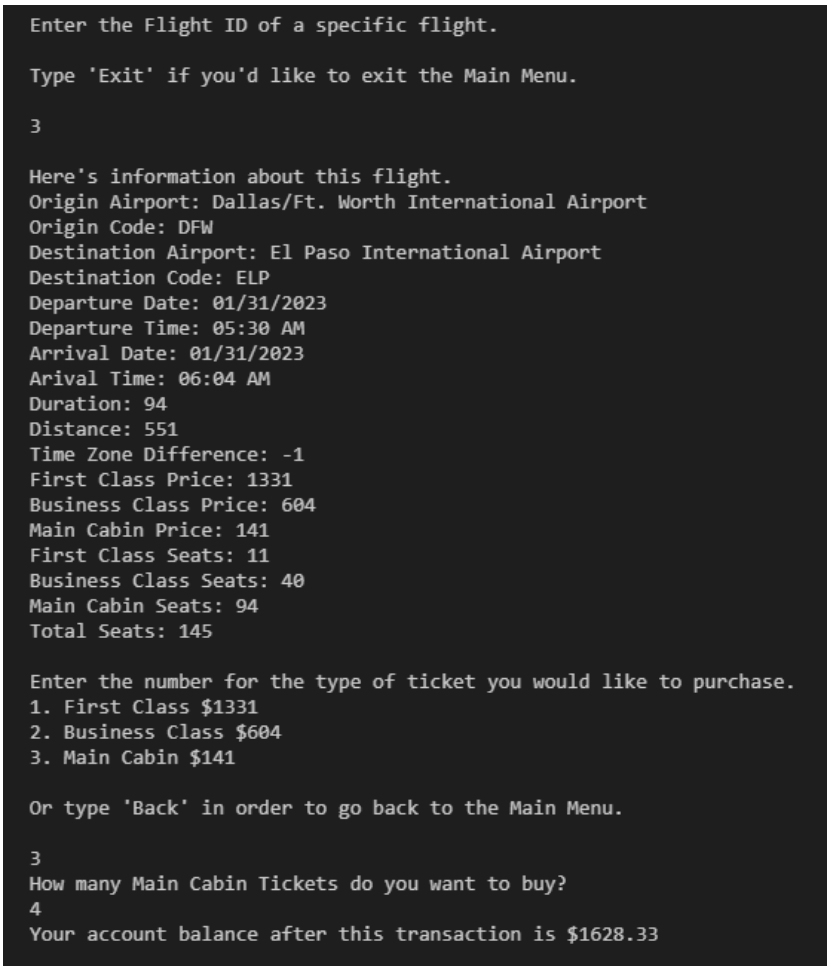


- After logging in correctly, the user has the option to find a flight using the flight ID. They are met with the flight's information and the ticket pricing. In this example, the user doesn't have enough money to buy a specific ticket, so they are met with the correct output specifying that they don't have enough money.

User Exiting Flight Menu

Here the user can be seen exiting the flight menu. Typing “back” in the paying process allows him to search for another flight ID. Typing “exit” terminates the flight menu and puts him in the login phase again where he types “exit” again and terminates the whole program.

Successful User Input Flight Menu



Finally, in this case, the user completes the transaction process. Going forth and buying 4 main cabin tickets. The customer purchased the tickets, and their quantity of money was subtracted from the total price of the tickets.

Code Reviews

The readability of my code looks adequate because of the proper Java Convention of variable naming, and correct code indentation. As seen from the testing section, the input and output are

displayed correctly. The user is met with the correct outputs depending on the input that he gives to the program. For example, if he logs in and accesses the flight menu, he is met with all the possible choices in buying a ticket. If he doesn't have enough money, he is met with a statement referring to his low quantity of money and backing him back to the purchase section. The functionalities of PA1 are all met as I followed each step of the PA's instructions. The program handles exceptions, especially when reading files, and writing into the log. If the program cannot find the correct files and an exception is thrown. A log text file is a used to update all of the interactions that the user has with the flight menu and the report also meets every requirement. All the properties of Object-Oriented programming are conducted well in this program, abstraction and inheritance are used correctly especially in the person and customer classes. Shared aggregation is also demonstrated well as the flight and customer classes store ticket objects to show the tickets that they have bought. The program contains well documented code and a header section.