****

**CE/CZ2002: Object-Oriented Design & Programming**

**Assignment**

**SS7 - Group 5**

Building an OO Application

**Declaration of Original Work for CE/CZ2002 Assignment**

We hereby declare that the attached group assignment has been researched, undertaken, completed and submitted as a collective effort by the group members listed below.

We have honored the principles of academic integrity and have upheld Student Code of Academic Conduct in the completion of this work.

We understand that if plagiarism is found in the assignment, then lower marks or no marks will be awarded for the assessed work. In addition, disciplinary actions may be taken.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Course (CE2002 or CZ2002) | Lab Group | Signature /Date |
| Nan Shi Yuan | CZ2002 | 7 | Shi yuan 11/12/2021 |
| Tan Ying Xia Cheryl | CZ2002 | 7 | Cheryl 11/12/2021 |
| Villaplana Hannah Danielle Ladera | CZ2002 | 7 | Hannah 11/12/2021 |
| Wong Li Han | CZ2002 | 7 | Lihan 11/12/2021 |

**Design Considerations**

**Approach Taken:**

By using object-oriented concepts, we aimed for our application to be resource-efficient. The main way we achieved this is by reusing resources through passing objects to various functions and the whole application revolves around those objects. While also ensuring that we avoid creating any new object unnecessarily. Furthermore, we wanted to hide the details of the methods from our users and made sure that users are unable to directly interact with the data.

**Use of Object-Oriented Concepts:**

**Abstraction:**

Our use of Abstraction can be found in our Main class also known as RRPSS, where we call various classes and their corresponding methods. The details as to what the classes do cannot be seen by the user. This makes it easier for us as we would only need to focus on the specific class RRPSS calls when we need it to change it. Abstraction can be seen throughout our application, RRPSS being just an example of it.

**Encapsulation/ Information Hiding:**

We ensured that our code is encapsulated with every attribute being private, so to access attributes, we would need to use get and set methods. Some attributes can only be accessed via get methods making them read-only attributes. It improves maintainability and flexibility and re-usability. This would also hide the complexity behind the methods.

**Inheritance:**

We think that the use of Inheritance is crucial for a project that has a large number of functions as it ensures that functions are segmented properly. However, since our application is relatively simple and small, we feel that the use of inheritance is not as important for keeping track of the attribute and uses of each function.

**Polymorphism:**

There is little overlap between our classes since they all have their individual responsibilities and hence we could not find an opportunity to group classes or function using polymorphism.

**Use of Design Principles:**

**S**ingle responsibility principle:

We made sure that every class had only 1 responsibility. For example, Order class would be in charge of the details behind each order, meanwhile, OrderDatabase class manages the Array list of all the Orders. Other examples of this can be seen with our use of a class with its corresponding manager, in Invoice, Customer, Reservation and Table.

**O**pen-closed principle:

Open-closed principle states that a module should be open for extension but closed for modification. We did this by making sure that each class has basic and essential functions such as returning their attributes so that other classes can have access to their attribute without any modification to the class that contains the attribute.

**L**iskov substitution principle:

The Liskov substitution principle states that any subclass object should be replaceable for the superclass that the object is derived from. For our application, we tried to abide by this principle with our AlaCarte and Promotional Set. These 2 classes are quite similar which would make it seem suitable to be under an interface or Promotional Set being an extension, however, Promotional Set would require more information than a regular AlaCarte item. Thus, violating Liskov substitution principle if an interface or extension is used. Thus, for our application, we did not create any class interface and thus this principle is not applied to our design

**I**nterface segregation principle:

Interface segregation principle states that many client-specific interfaces are better than one general-purpose interface. This means that we should avoid creating interfaces that are too generalized such that any class could be categorized under it. For our application, we did not create any class interface and thus this principle is not applicable to our design.

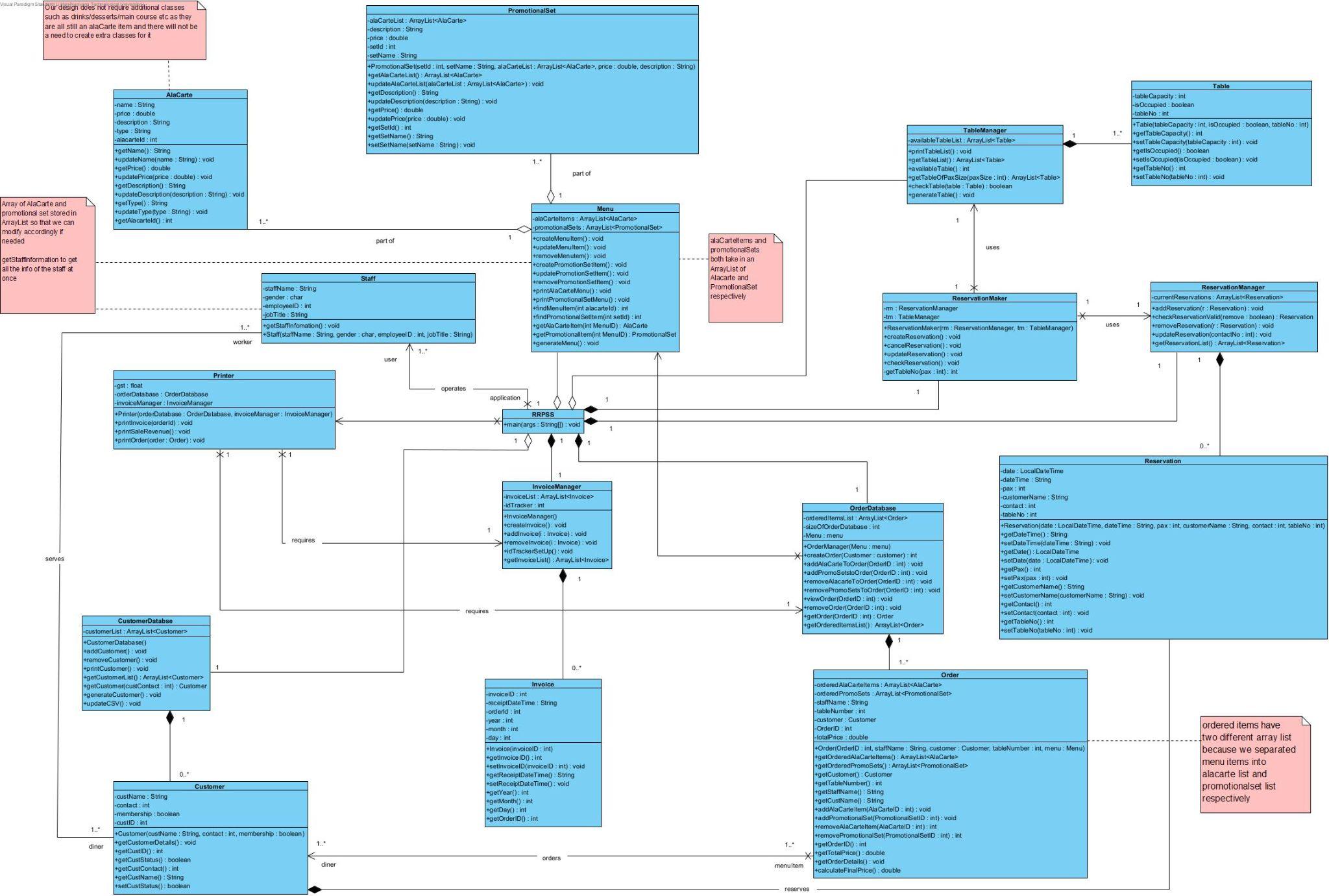
**D**ependency inversion principle:

Dependency inversion principle states that high level modules should not depend on low level modules. They should both depend on abstractions instead. This means that similar modules should be independent of each other. Since we do not use any abstraction class, we did not violate the principle.

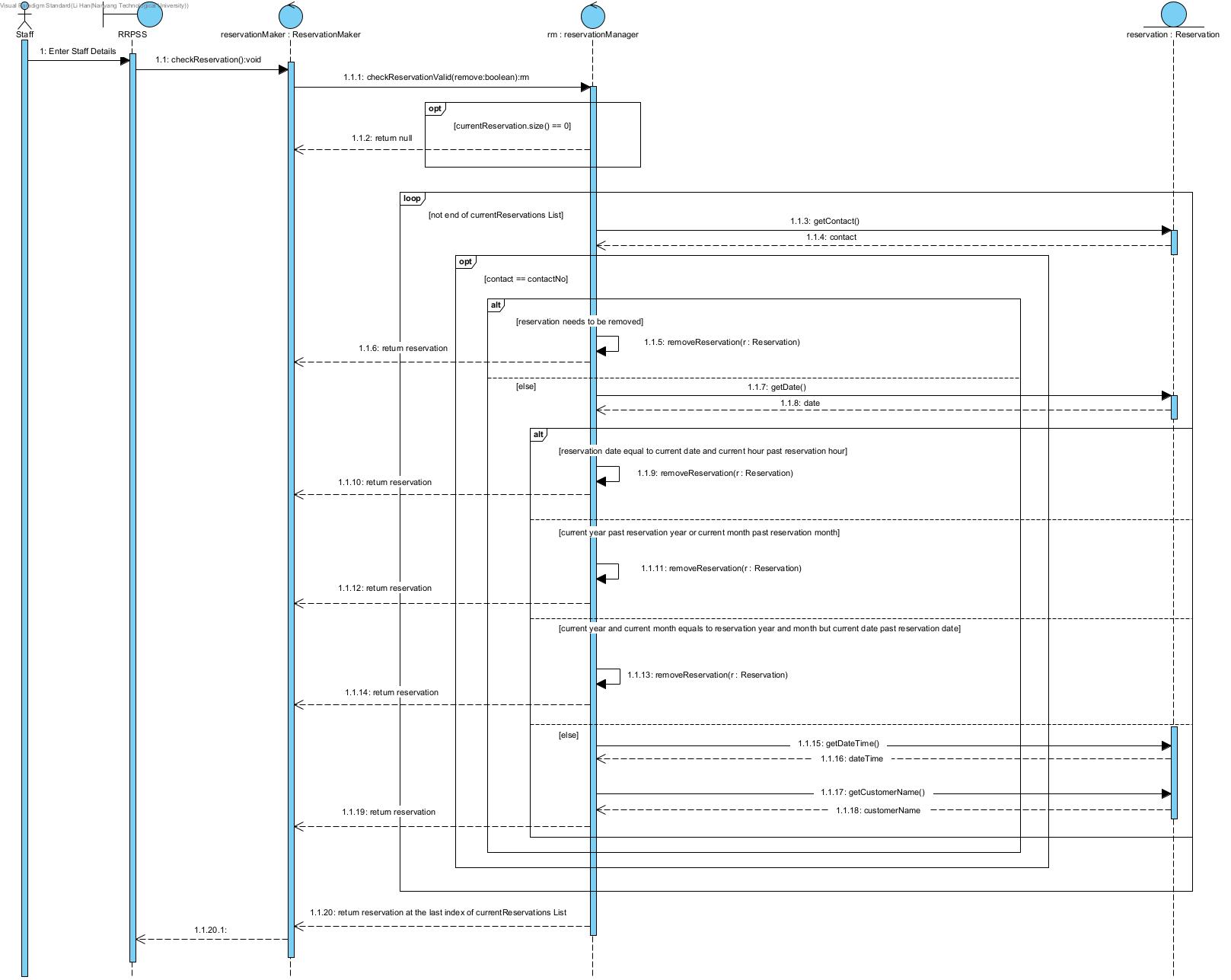
**Assumptions made:**

Our restaurant is reservation based only. We assume that the employee knows how to use the application and will not purposely enter invalid input. We also assume that the shop closes at 10pm and the last reservation entry is at 9pm. Customers are assumed to wait for their reservations if they come earlier than their reservation slot, and each customer can only book one reservation at a time. Assume there are no duplicates of contact number and a customer’s contact number is a primary key and a unique attribute. Next, we also assume that new customers’ membership only takes effect on their next visit and only existing members are entitled to a discount. Assume the staff would also know the table number of the customer dining at the restaurant.

**Detailed Class Diagram:**

****

**Detailed Sequence Diagram:**



**Test Cases:**

* **Reservation:**

1. Multiple reservations made for the same table
2. Update Reservation:
   * + Change reservation time & table
     + Change reservation pax

A. Multiple reservations made for the same table

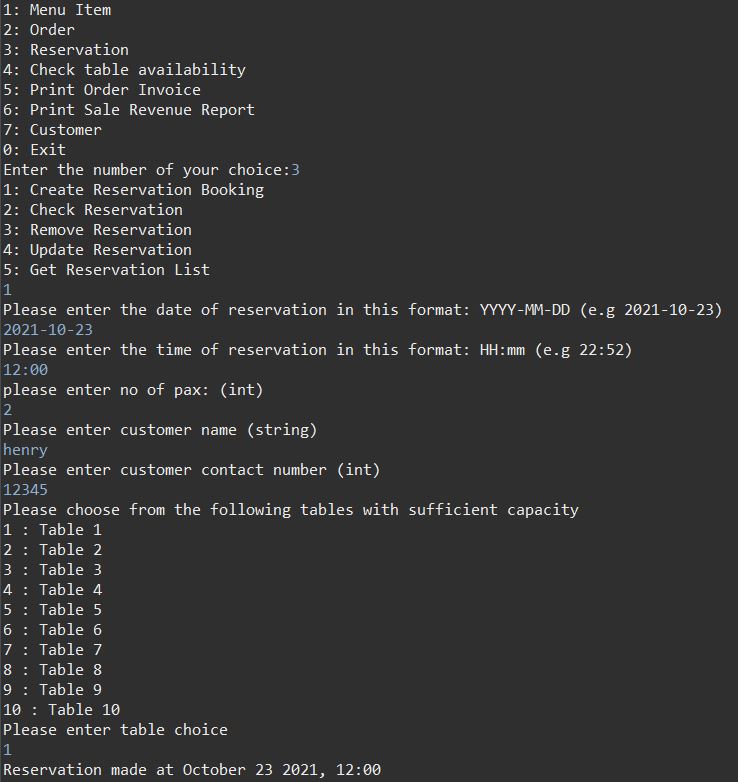


Figure 1a: Adding a reservation for customer Henry at Table 1 at 2021-10-23 , 12:00.

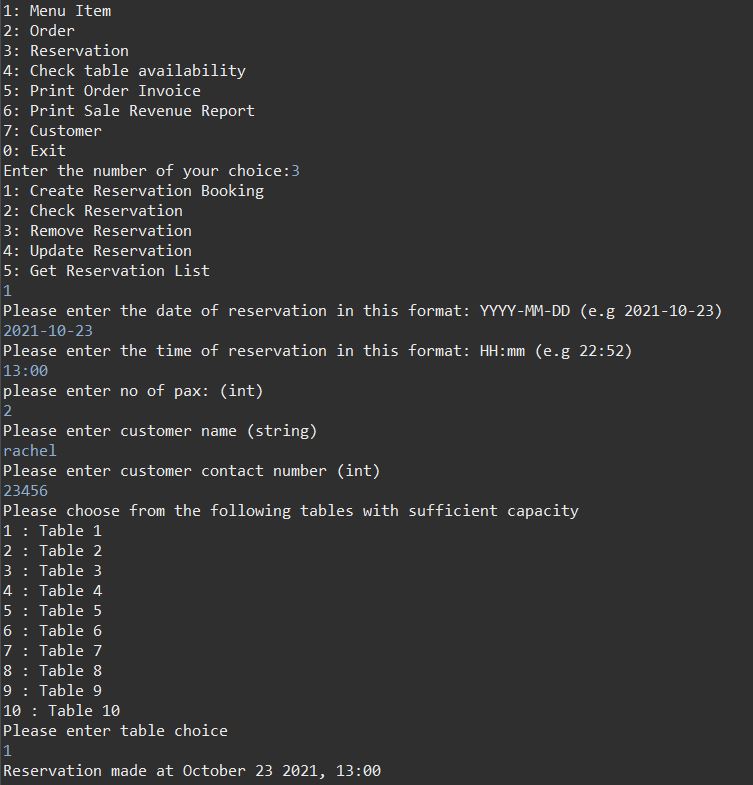


Figure 2a: Adding a reservation for customer Rachel at Table 1 at 2021-10-23, 13:00.

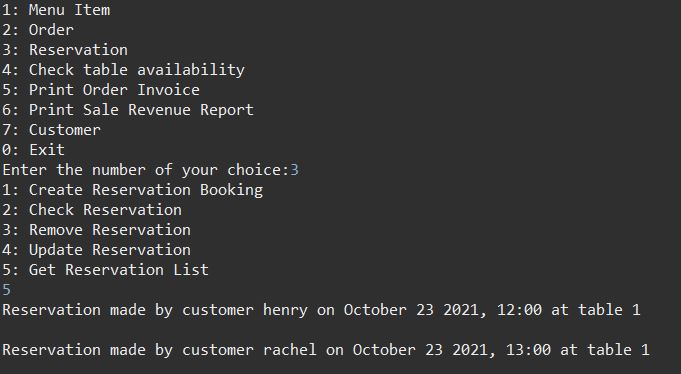


Figure 3a: 2 reservations being done at the same Table 1

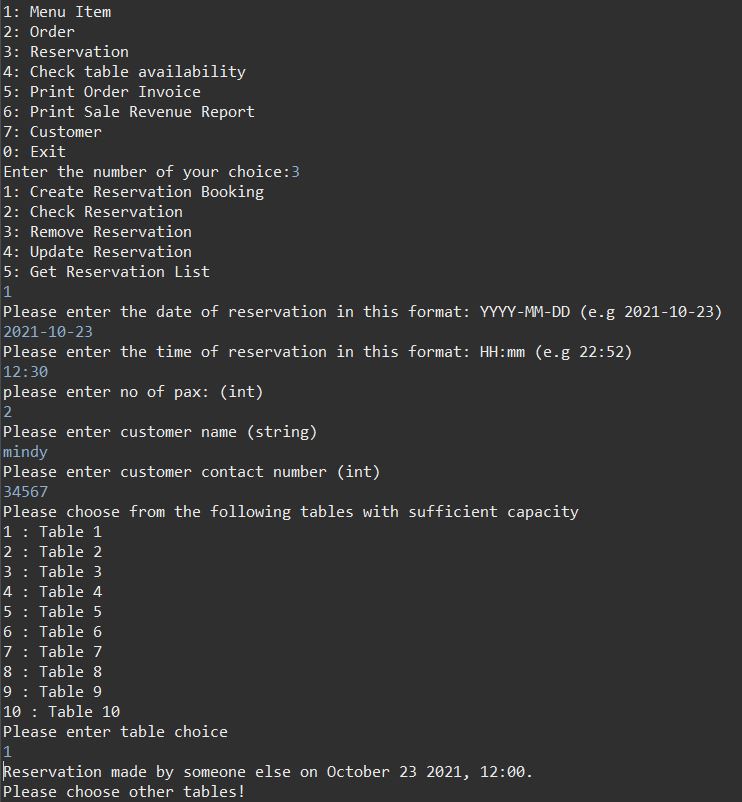


Figure 4a: Reservations between 2 customers at the same table must be done at least 1 hour apart

B. Update Reservation - changing time & table

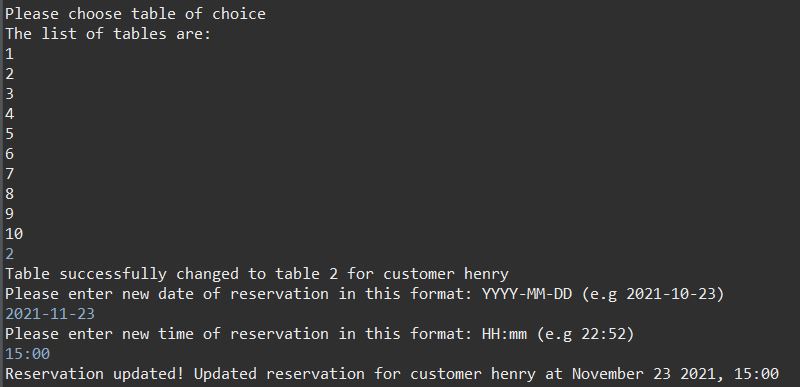


Figure 1b: The customer Henry updating his reservation time and table from 2021-10-23, 12:00 at table 1 to 2021-11-23, 15:00 at table 2.

B. Update Reservation - changing pax

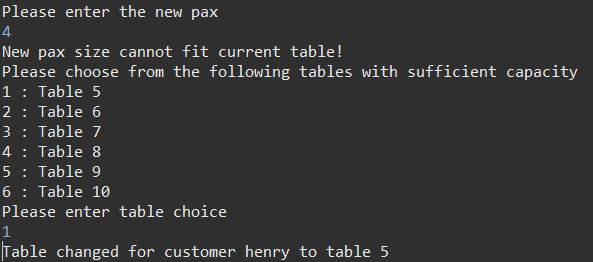


Figure 2b: The customer Henry updating the pax of his reservation up 4 and selecting a new table.

**Link to Video Presentation:**

<https://www.youtube.com/watch?v=lyV84ykYHKk>