# 1. Introduction

The Restaurant Reservation and Point of Sale System(RRPSS) is a console-based application to computerise the process of making reservations, recording orders and displaying sale records. It will be used only by restaurant staff.

This report covers the assumptions made and the design principles (such as the SOLID framework and other OOP concepts) taken into consideration during the implementation of the program. A detailed UML class diagram acts as the foundation of our application’s design. The diagram displays the classes and interfaces used in our program and shows how they interact with each other. The UML sequence diagram that is included in this report demonstrates the flow of messages between relevant actors within our implementation of the “Check/Remove Reservation Booking” function. Essential test cases that are not included in the [demo video](https://youtu.be/Srp23VuX6iU) (refer [appendix](#_cy7z5orpcrcb)) are also explained to confirm that the system can handle edge cases.

# 2. Assumptions

1. The restaurant only operates from 11am to 9pm (1100hrs to 2100hrs).
2. A reservation is automatically (without input from a user) deleted by the program 30 minutes after the actual booking time.
3. Manual deletion of reservations is done via searching for the customer’s contact number in the reservations record.
4. There are exactly 20 tables in the restaurant, with seating capacities of 2, 4, 6, 8, 10 pax. There are 4 tables of each size.
5. Reservations with an odd number of people can be assigned to a table that has an even number of seats.
6. Customers can be assigned to a table that has more seats than necessary (eg. 2 people can be assigned to a 4-seater table if all the 2-seater tables are full).
7. Reservations must be made at least 1 hour in advance of the desired timing.
8. Reservations cannot be made within 1 hour before and 1 hour after if all the tables are occupied at any point of time.
9. Customers will still be able to book a table at a different time if all the tables before their booking time are full.
10. The customers can check their reservation details by providing their contact number to the staff.
11. The duration that a customer eats in the restaurant is assumed to be 1 hour, and hence the booking slots are of length 1 hour.
12. Walk-ins are not allowed.
13. No takeaway and delivery is allowed in the restaurant. Only dine-ins are allowed.
14. Once an order is created, the table is considered occupied.
15. A table is considered available after the customers of the table make a payment even though they might not have left the table yet.
16. An order for a table is removed automatically after the table’s payment is made.
17. Sales reports are generated by specifying a start and end date.
18. Membership discount is 5% and GST is 7%.
19. The customers are expected to show their membership card upon payment; membership details are not stored inside the database.

# 3. Design Considerations

## 3.1 Modelling

### 3.1.1 Entity-Control-Boundary

List of entity classes: *MenuItem, MainCourse, Drink, Dessert, PromotionalItem, Order, Invoice, Staff, Table, Reservation.*

List of control classes: *MenuManager, OrderManager, InvoiceManager, StaffManager, TableManager, ReservationManager, RestaurantApp, ExceptionHandler, InvoiceIO, MenuIO, ReservationIO*

List of boundary classes: *MenuBoundary, OrderBoundary, InvoiceBoundary, ReservationBoundary, TableBoundary, RestaurantBoundary*

Our design follows the entity-control-boundary (ECB) architectural pattern; Each entity class can only talk to a control class, each boundary class can only talk to the users and control classes, and control classes play the role of middlemen between entity and boundary classes (Control classes are able to communicate with other control classes). This helps to achieve a cleaner project architecture and facilitate smoother development of the program now and in the future.

## 3.2. SOLID Design Principles

### 3.2.1. Single-Responsibility Principle

Following the ECB pattern, each class only has a single responsibility to carry out. As an example, the *order class* only stores information that is needed for an order such as *tableID, staffID* and the *ArrayList* of *ordered* items. Other tasks that involve creating, updating, and removing orders (which can be generalised as editing orders) are carried out by *OrderManager class.* The *OrderBoundary* class, however, only deals with the input and output from the user, such as display options for the next step, take user inputs and print out some details. This applies to other classes that were created inside the program. By using this pattern, we avoid implementing a “god class” which serves many roles. Hence, there will only be one reason to make changes to each class.

### 3.2.2. Open-Closed Principle

We have designed each item in the menu as a subclass of *MenuItem. Class menuItem* is a base class of *class MainCourse, Drink, Dessert, PromotionalItem*. This allows the menu items to be further diversified into different categories such as appetiser and allows these classes to be inherited by more specific classes. Say, *MainCourse* can be subdivided into *Pizza* and *Spaghetti. Pizza* and *Spaghetti class* can also be further extended by *Classic, AllTimeFavourite class* and so on. The implementation of these extensions does not need to make any changes for the existing classes we have, since subclasses automatically inherit the general properties and methods of their parent classes.

Another example of extension is that the *Table class* can be used in different layouts. There can be a *TableLayout class* that contains the *Table object* so that different layouts in the restaurant can be possible. A perfect situation for this idea to chip in is during pandemic periods where the restaurant needs to change their table layouts constantly in accordance with safe-distancing measures implemented by the government. The *Table class* does not need to be changed since it only contains the basic information of a table such as *tableID, isAvailable* and *seatCap*. Hence the *TableLayout class* can contain information such as the *numberOfTables* inside the layout.

### 3.2.3. Liskov Substitution Principle

*class MenuItem* is a base class of *class MainCourse, Drink, Dessert, PromotionalItem* and the subclasses inherits all the attributes and methods of the base class.

private ArrayList<MenuItem> menu\_item\_array = new ArrayList<>()

menuItem = new MainCourse(name, description, price);

In this segment the subclasses of *MenuItem* are upcasted to *MenuItem*, and it works perfectly fine. Specifically, the subclasses do not throw any exceptions that the base class does not, neither do the methods implemented return any less than their counterparts in the parent class. This implies that any place where a *MenuItem* is taken in can also take in any subclass of *MenuItem* without causing problems. Hence, the principle is obeyed.

### 3.2.4. Interface Segregation Principle

To save and load *Menu*s, *Invoice*s, and *Reservation*s, we implement classes that translate entity data to and from CSV files, while we use interfaces to link specialised translators with general-purpose CSV reader and writer classes. Due to the vast differences between the structures of each entity, we avoid making catch-all interfaces which cover more than what is needed to maintain modularity and flexibility of our program.

In our program, we could have declared an *Editable* interface to cover both cases where an entity can be either imported or exported; All of *Menu*s, *Invoice*s, and *Reservation*s can be (and are indeed) imported AND exported in our program. We instead declare 2 interfaces with a single purpose each: *Importable*, and *Exportable*. This gives us room for modularity and expandability for potential classes which need exactly one of importing and exporting capability (such as staff lists, which are only imported), hence obeying this principle.

### 3.2.5. Dependency Injection Principle

By using the entity-control-boundary (ECB) pattern, a system with loose coupling is produced since the dependencies between classes is largely reduced. We have also put in some thought for scalability, where the restaurant can have different sets of menus, different floors of table layouts and different modes of consuming our products such as walk-ins, takeaways and online orders. Please refer to ssection “[Proposed Future Features](#_tzcp6v5lpt22)” for improvements that our design allows.

## 3.3. Object-Oriented Concepts

### 3.3.1. Inheritance

Inheritance is implemented in the design for several classes. For example, *MenuItem* acts as a parent class of *MainCourse, Dessert, Drink* and *PromotionalItem.* All the child classes absorb the attributes such as *name, description* and *price* from the parent class. Methods such as *getPrice* and *setPrice*, etc. are also inherited to enable code reuse and reduce the programming effort in implementing new subclasses.

### 3.3.2. Encapsulation

Encapsulation builds a barrier to protect an object’s private data. In our design, variables in the classes are declared as private while public getter and setter methods are provided to view or modify the variable values, adding features such as read-only or write-only.

For example, the *Reservation* class has private attributes such as *customerName, contact, bookingTime*, etc. As they are private, they are only accessible through their respective getter and setter methods. As such, the *Reservation* class has total control over what is stored in its fields, as well as what is shown to outside classes.

Information hiding also hides details of the class from the users. For instance, the staff only needs to know how to enter necessary information to create a reservation but does not need to know the implementation details of the method.

### 3.3.3. Polymorphism

The creation of lower level classes does not affect the programme developed in the higher level class, which is their superclass. This facilitates adding new subclasses to a system with minimal modifications to the system’s code.

For example, the subclasses of MenuItem override the toString() method. The same method and signature cause different actions to occur, depending on the type of object the method is involving. This minimises the change of codes if more menu items are added into the system.

### 3.3.4 Exception handling

The *ExceptionHandler class* has static methods to check for the inputs from the staff. This class has methods to check whether the staff entered the correct:

1. data type (eg. integer, double or string)
2. integer within the range
3. format of date and time
4. format of date

Other than these general inputs, the other classes also implement their own error checking within their classes so that the system will not crash when a staff member enters some wrong details. For example, the *ReservationManager class* checks whether the date and time entered for a reservation is within the operation hours of the restaurant. It also checks that the reservation must be done at least 1 hour before the booking time. When checking a reservation for a customer, the *ReservationManager class* also checks whether the contact number exists in the database so that it does not access a non-existent item.

# 4. UML Class and Sequence Diagram

## 4.1 Class diagram

## 

For higher resolution image of the UML class diagram, please refer to the attached photo in the “Attachments” folder.

## 4.2 Sequence diagram

A picture containing calendar

Description automatically generated

1. Staff member sends the message to ReservationManager to execute the *automatic remove reservations method*

1.1 *ReservationManager class* send a message to *Executors class* to create a new thread

1.2 *ReservationManager class* sends a message to the *executor* to set a scheduled task to run every 60 seconds

2. The thread created calls the Runnable task, which is the *automaticRemoveReservationHelper* method

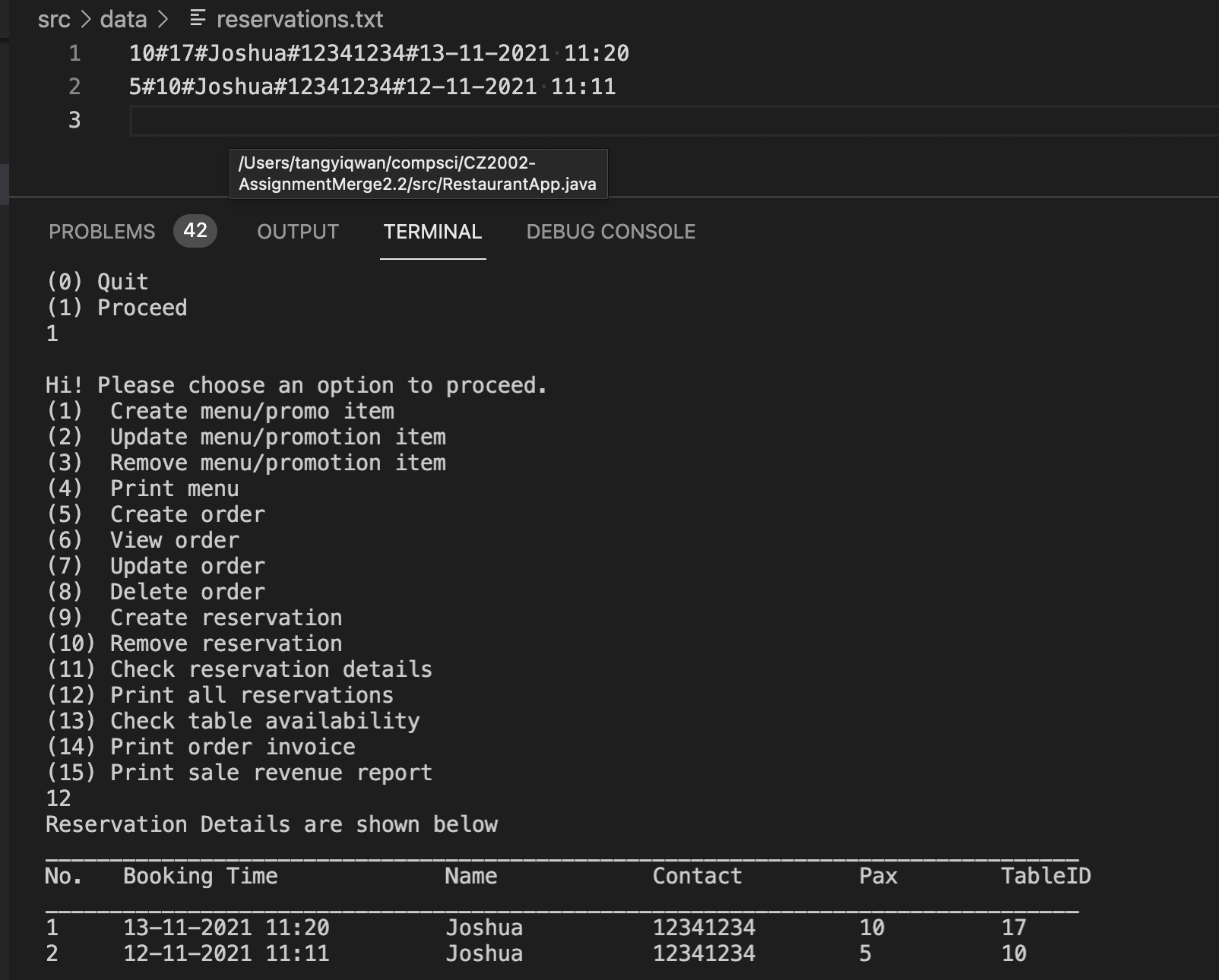
2.1. Loop: Iterates through the reservations *array list.*  ReservationManager calls the *reservation* class to get the expiry time from each reservation instance

2.2 opt:Check if each reservation is expired or not. If expired, the reservation will be removed from the array list of reservations

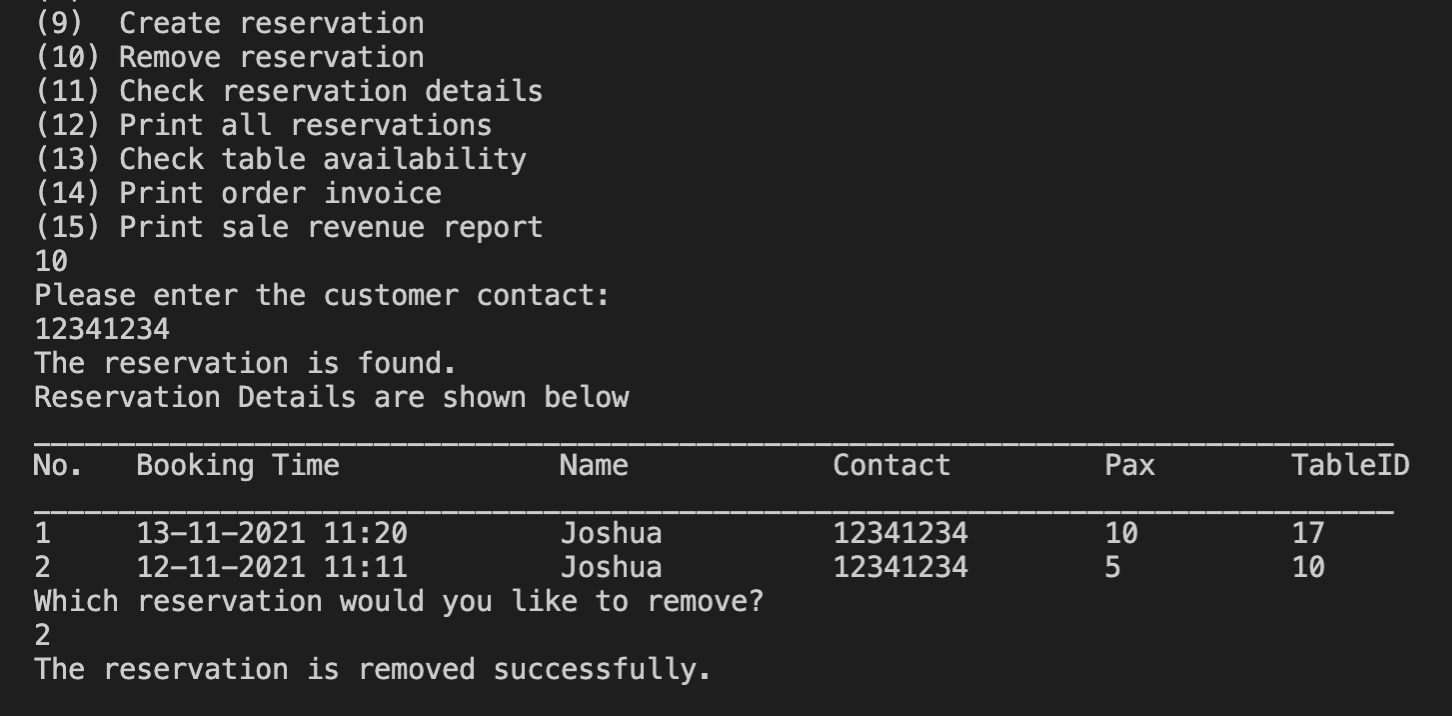
For higher resolution image of the UML sequence diagram, please refer to the attached photo in the “Attachments” folder

# 6. Test Cases

## 6.1. Manually remove a reservation



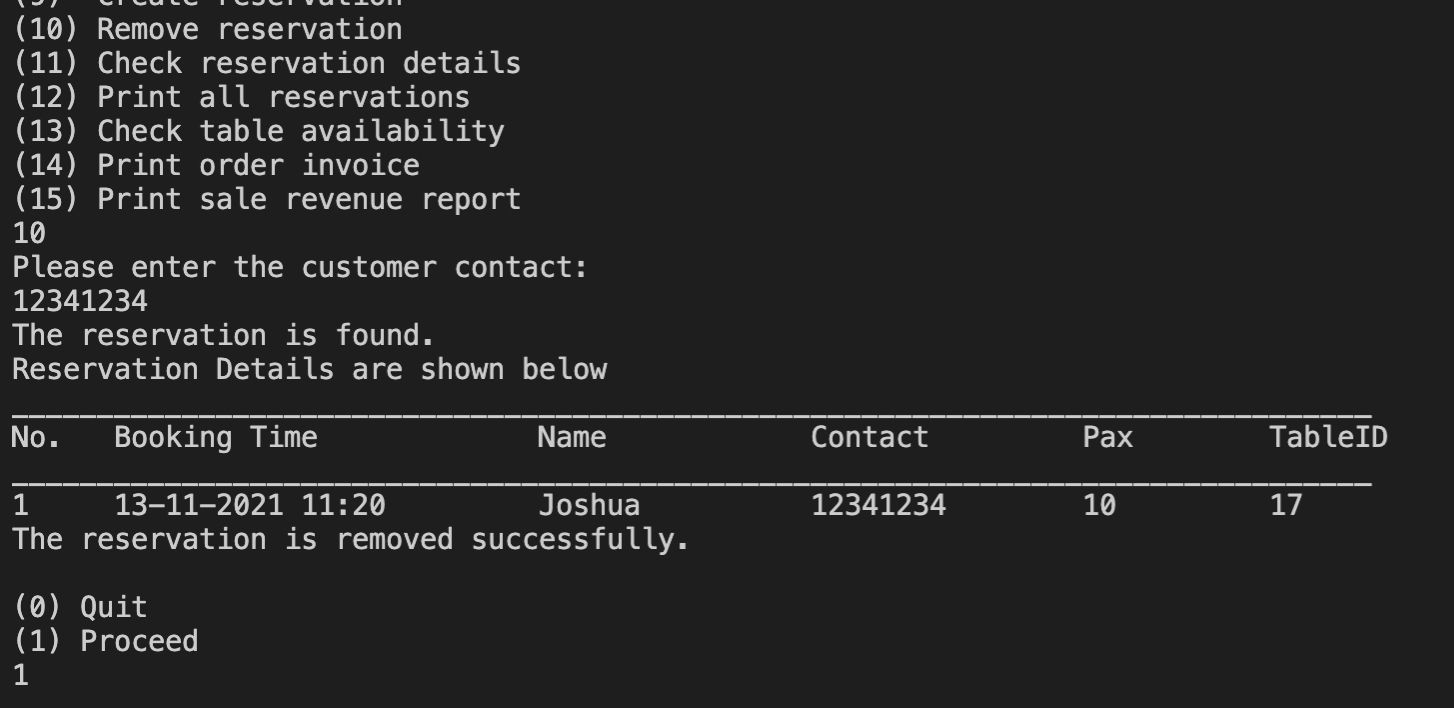
1. There are two reservations made by the same person on different days



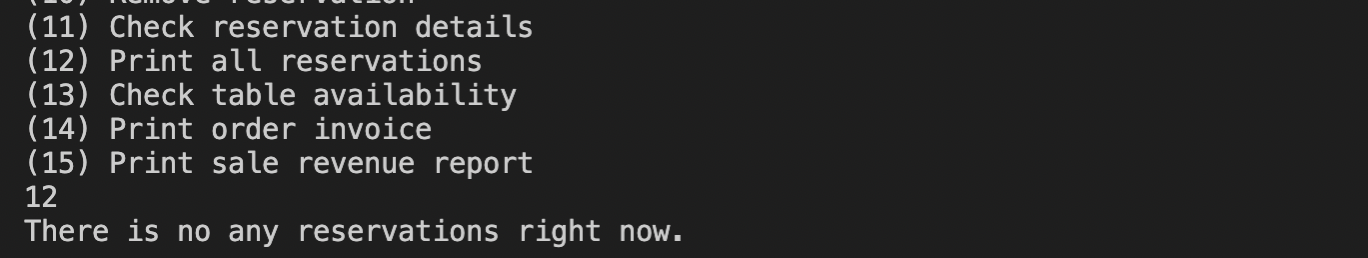
1. After the contact is taken in, the application prints out all the reservations made by that person. The person can choose which reservation to remove by entering the index of the list printed out.



1. There is only one reservation left.

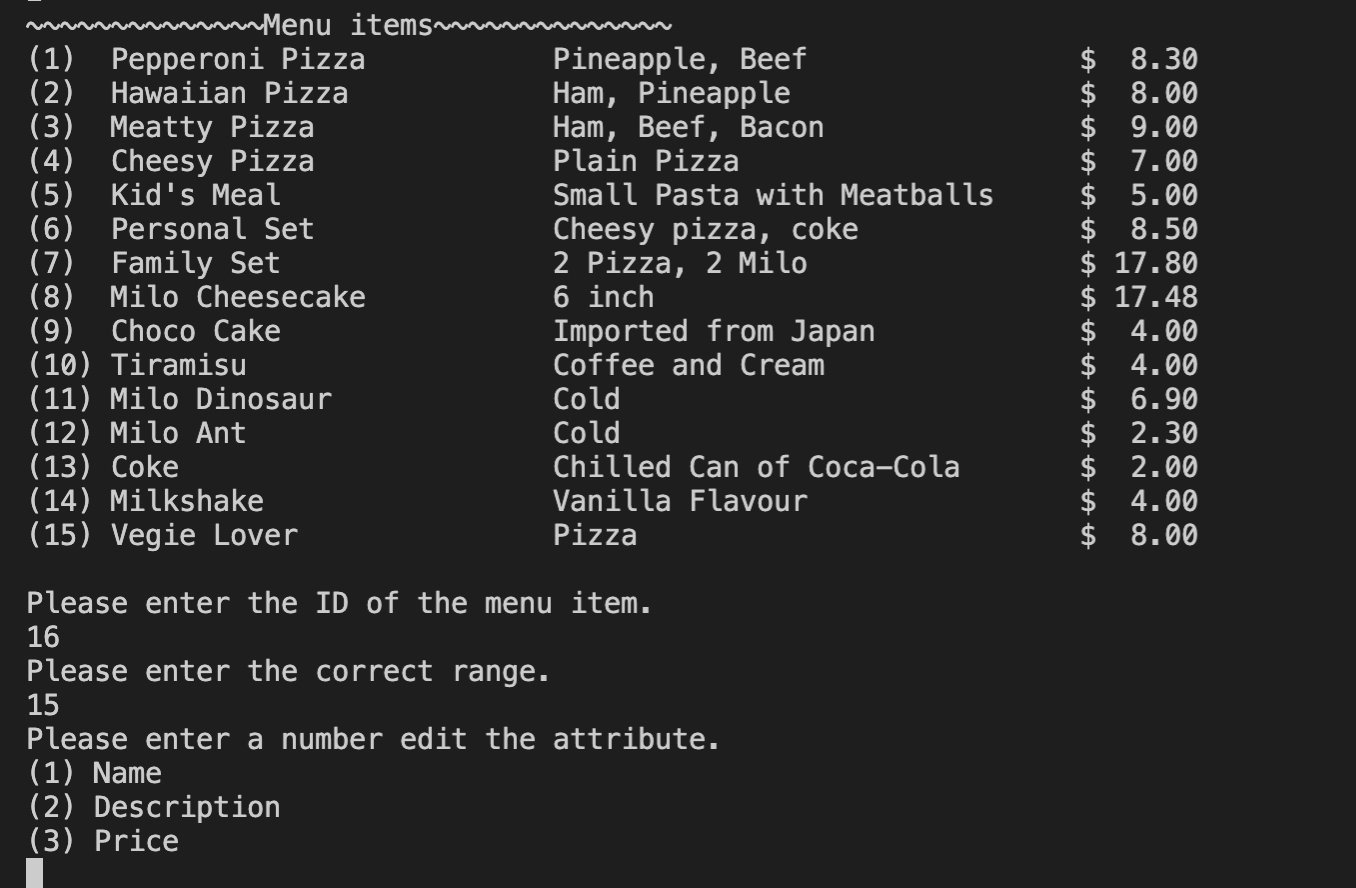


1. The remaining reservation can be removed again.



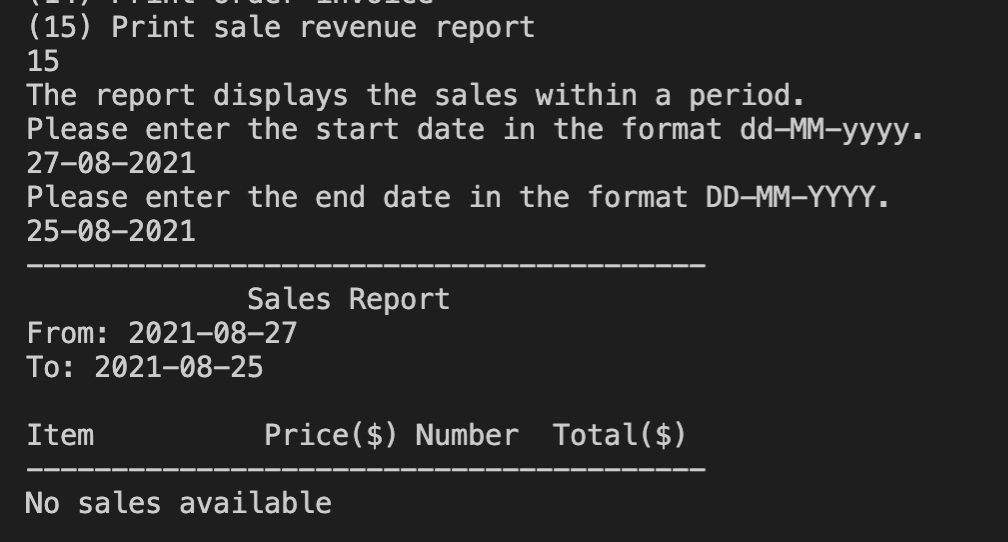
1. No more reservations are available.

## 6.2. Editing menu items



If the user enters a value that is out of the range, the system will check for the range of the input and prompt the user to enter the correct values.

## 6.3. Sales report



If the user enters a start date that is later than the end date, the system will not print any report.

## 7.3. Proposed Future Features

### 7.3.1. Restaurant Membership Feature

In our reservation system we do not store the membership details of our customers. In future designs, we could create new classes and CSV Managers to store the membership information such as membership status, start date of membership and points accumulated in the database. Points can be also used by the customers to redeem free food and drinks in our restaurant.

### 7.3.2. Dine-in, takeaway and online order Features

Currently we only assume that all customers come before they book a table through our reservation system. However, our system can be further scaled up for the restaurant to provide more options such as dine-ins, takeaways and online orders. Different subclasses can be created under *class order* to further diversify different types of orders. This allows different types of invoices to be generated and further analysed through sales reports.

# Appendix

1. Full link of our demo video: <https://youtu.be/Srp23VuX6iU>
   1. Edits done on the video were only to remove extra parts or long pauses.
   2. No human manipulations were done to the functions that were demonstrated in the video.
   3. All background music used in the video does not violate any copyright rules.
   4. The picture taken in the video was taken while abiding safe management measures at the point of time.