****

**CZ2002 OBJECT-ORIENTED DESIGN AND PROGRAMMING**

***AY21/22 Semester 2 Group Assignment***

***Restaurant Reservation and Point of Sales System (RRPSS)***

**Lab Group: CS2**

|  |  |  |
| --- | --- | --- |
| **Group Members** | | |
| S/N | Matriculation No. | Student Name |
| 1. | U |  |
| 2. | U |  |
| 3. | U |  |
| 4. | U |  |
| 5. | U |  |

**Date of Submission: 17th April 2022**

**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**

**NANYANG TECHNOLOGICAL UNIVERSITY**

Contents

[1. Application Overview 4](#_Toc6530564)

[2. Assumptions Made in Accordance to Application 4](#_Toc6530565)

[3. Design Considerations 4](#_Toc6530566)

[3.1. SOLID Approach 4](#_Toc6530567)

[3.1.1. Single-Responsibility Principle 4](#_Toc6530568)

[3.1.2. Open-Closed Principle 5](#_Toc6530569)

[3.1.3. Liskov Substitution Principle 5](#_Toc6530570)

[3.1.4. Interface Segregation Principle 5](#_Toc6530571)

[3.1.5. Dependency Injection Principle 5](#_Toc6530572)

[3.2. Object-Oriented Principles 6](#_Toc6530573)

[3.2.1. Inheritance 6](#_Toc6530574)

[3.2.2. Encapsulation 6](#_Toc6530575)

[3.2.3. Polymorphism 6](#_Toc6530576)

[3.3. Proposed Future Features 7](#_Toc6530577)

[3.3.1. Restaurant Membership Feature 7](#_Toc6530578)

[3.3.2. Staff Login Feature 7](#_Toc6530579)

[4. UML Class Diagram for RRPSS (separate file outside of report) 8](#_Toc6530580)

[5. Sequence Diagram for Print Bill Invoice (separate file outside of report) 8](#_Toc6530581)

[6. Test Cases 9](#_Toc6530582)

[6.1. Reservation Management 9](#_Toc6530583)

[6.2. Check for Table Availability 9](#_Toc6530584)

[6.3. Create and Add Items to Order 10](#_Toc6530585)

[6.4. Checkout Order and Print Bill Invoice 10](#_Toc6530586)

[6.5. Generating Sales Revenue 10](#_Toc6530587)

[6.6. Menu Item Management 11](#_Toc6530588)

[6.7. Promotion Management 11](#_Toc6530589)

# Application Overview

The Hotel Reservation and Payment System (HRPS) is an application to facilitate processes of reserving rooms, making orders, and searching for records by introducing computer programming. It’s client is hotel staffs only.

# Assumptions Made in Accordance to Application

Excluding the assumptions already specified in the manual, below assumptions are made during building process of HRPS:

* Guest

1. Guests information are stored in the system, even if they have already checked out.
2. No two guests have the same contact number.
3. We assume the guest can manage the space well so we do not limit the number of adults and children in a reservation, even if the number of adults and children are more than the room can logically accommodate.

* Room

1. Room conditions:
   1. The hotel consists of 48 rooms in total that locates at level 2 to 7. Each level consists of 8 rooms.
   2. At level 2-5, there are 4 double-bed rooms and 4 single-bed room each floor.
   3. At level 6, there are 8 vip rooms.
   4. At level 7-8, there are 4 deluxe rooms and 4 suite room each floor.
   5. As a total, there are 16 single rooms, 16 double rooms, 8 vip rooms, 8 deluxe rooms and 8 suite.
   6. 4 four facing are evenly distributed among the rooms at each floor for each type.
   7. There are 2 WiFi-disabled single rooms and 2 WiFi-disabled double rooms.
   8. There are 4 non-smoking-free single rooms and 4 non-smoking-free double rooms.
2. Room status change follows a logic chain.
   1. For normal reservation process, it follows VACANT→RESERVED→OCCUPIED→VACANT.
   2. For walk-in reservation, it is allowed to have VACANT→OCCUPIED→VACANT.
   3. For maintenance and finish maintenance, it is allowed to have VACANT→MAINTENANCE and MAINTENANCE→VACANT only.
3. For room under maintenance, the guest can't make any reservation as it is not known that when the maintenance will be finished. To make maintaining room available for reservation, the user have to manually finish the maintenance of that room.

* Reservation

1. Reservation status change follows the chains that is specified in each subclass. For instance, checkOut() method that is able to turn a Reservation to a CheckedOutReservation would only be available to CheckedInReservation.
2. Assume that all checked-out reservation and expired reservation will not be further use. However, we have propose to keep them in the system for history tracking in future. (See "Our own proposed feature")
3. If a guest wish to change the room number under that particular reservation, he has to cancel the reservation and make a new one with the new room number.
4. Time is static for the reservations. We would need manual check of the current time in the following cases
   1. When a guest is checking in and checking out:

we need user to key in the actual check in time during check in. If this is a guest from a confirmed reservation, we will use the actual check in time to see if the confirmed reservation has expired (≥24 hours after expected check in time). After checking, if the reservation has expired, the user will not be able to check in to the reserved room under that reservation.

we need user to key in the check out time to see if there is early check out.

* 1. When drop reservation from wait list:

we need user to check one by one whether the waiting reservation's desired check in time has been reached, and drop it from the wait list if it does.

* Room Service

1. Room Service Orders can’t be made unless the Reservation is checked in.
2. Orders are not refundable. Once an order is made, it can't be cancelled and the guest has to pay for this order.
3. Items in the same batch of order has the same status(confirmed, processing or delivered) as they are processed together.

* Payment

1. A credit card has to be given regardless of which payment method would be used during checkout.
2. Payments made are not refundable.
3. Promotion rate are set to be a 10% discount. Goods & Services Tax is 7% and is applied to the subtotal after promotion deductions.
4. Credit card with 16 digits is assumed to be valid.

# Design Considerations

Below are some of the design considerations we have taken in order to make our application, the RRPSS.

## SOLID Approach

The team has taken the SOLID Design Principle approach in order to enhance the reusability of the application and aims to make software designs more understandable, flexible and maintainable.

### Single-Responsibility Principle

The Single-Responsibility Principle (SRP) assumes that each class will only have the responsibility of its own, and no other responsibilities, i.e. each class is responsible for its functions. For example, in our RRPSS, we have a separate menu user interface (UI) class for each recurring menu. Each menu will have its own menu flow, as compared to clustering every menu under one menu class, or the main application file.

### Open-Closed Principle

The Open-Closed Principle defines that a module should be open for extension but closed for modification. The team would want to modify a class by extend it to a subclass in the future if we have modifications to it.

For example, the Table class has the enumerator class of TableSeats, to determine how many seats are there at that table. The team currently does not have information about what special attributes each table have such that it would be substantial to become a subclass of Table by itself. However, if the team were provided with such information in the future, the team would be able to extend the Table to its subclasses without having major modifications to the original source code of Table.

### Liskov Substitution Principle

The Liskov Substitution Principle defines that the subclasses must be substitutable for their base class, offering no more or no less as compared to their base class.

For example, the team has implemented the BaseMenuUI class, with many menu UI classes inheriting from the BaseMenuUI class. The inherited classes behave as expected of the BaseMenuUI – to ensure the menu flow of each function of the RRPSS is run smoothly, containing its own ability, no more and no less.

### Interface Segregation Principle

The Interface Segregation Principle defines the classes should not implement interfaces that they do not use.

In the application, the only classes that implements an interface class is the comma-separated values (CSV) helper classes, which implement ICsvSerialisable, as each CSV helper class has their own defined file name and headers to import the data from. ICsvSerialisable forces the CSV helper classes that implements itself to define their respective read from and/or write to functions for their differing CSV filenames.

### Dependency Injection Principle

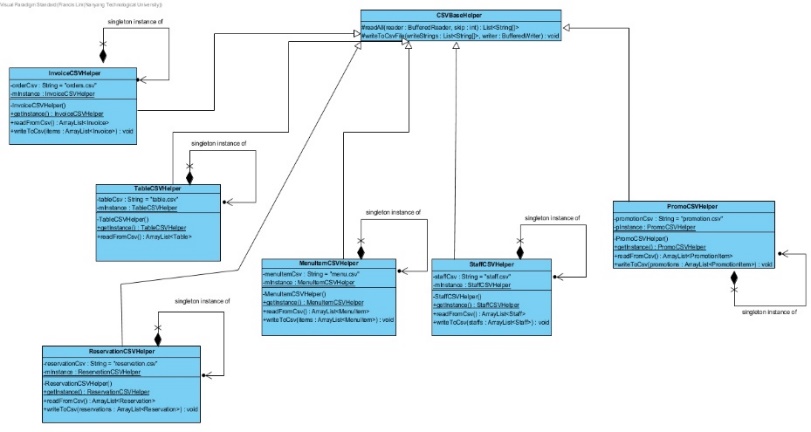
The Dependency Injection Principle states that higher level modules in the application does not depend on lower level modules. Rather they should depend on abstractions.

One of the ways we have implemented this is with Interface Injection. This can be found in all our classes that we require to be serializable such as the Invoices, RestaurantItems, and Reservation classes. For these classes, we implemented an interface ICsvSerializable which contains the method that they will have to implement to serialize the object data into a CSV String readable format. This will be called by all the classes that interacts with our CSV data files.

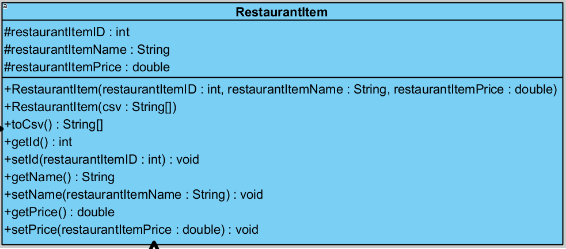
## Object-Oriented Principles

The RRPSS was implemented with various object-oriented (OO) concepts to enhance reusability and maintainability of the application.

### Inheritance

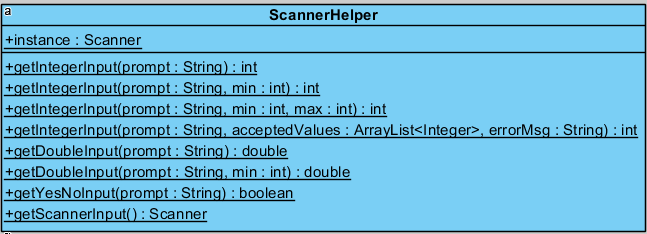
Inheritance is a concept that allows the subclasses to inherit attributes and functions from a superclass. This concept is prominent in our application, as many of our classes, specifically our CSVHelper and MenuUI classes inherit from a base class, as their respectively use similar attributes and/or functions from the super class.

### Encapsulation

Encapsulation is a concept where certain classes attributes are only accessible via their accessors and mutator (or get and set) methods, keeping its visibility to private or protected and hiding it from other classes, or only maintaining its visibility to its subclasses or classes within the same package.

The RestaurantItem class demonstrates the concept of encapsulation by keeping its attributes to the protected state while keeping its accessor and mutator methods publicly accessible.

### Polymorphism

Polymorphism is the concept of allowing an object to take on different forms based on the type of data being used in the method or class. In our RRPSS system, we have multiple instances of this happening. An example of it can be found in our ScannerHelper class, which is the class that provides all the helper static methods for dealing with and validating our user input. In this case we have employed an example of polymorphism known as “Overloading” where we implement multiple methods of the same name but with different parameters that they accept. This will allow us to use the same method to do multiple task based on what we require the method to do

## Proposed Future Features

The current RRPSS application has been designed in such a way that it could give way to two proposed future features that would be implemented with less hassle.

### Restaurant Membership Feature

The Restaurant Membership Feature will realise customers being able to accumulate “loyalty points” as a form of customer loyalty when they dine at the restaurant often.

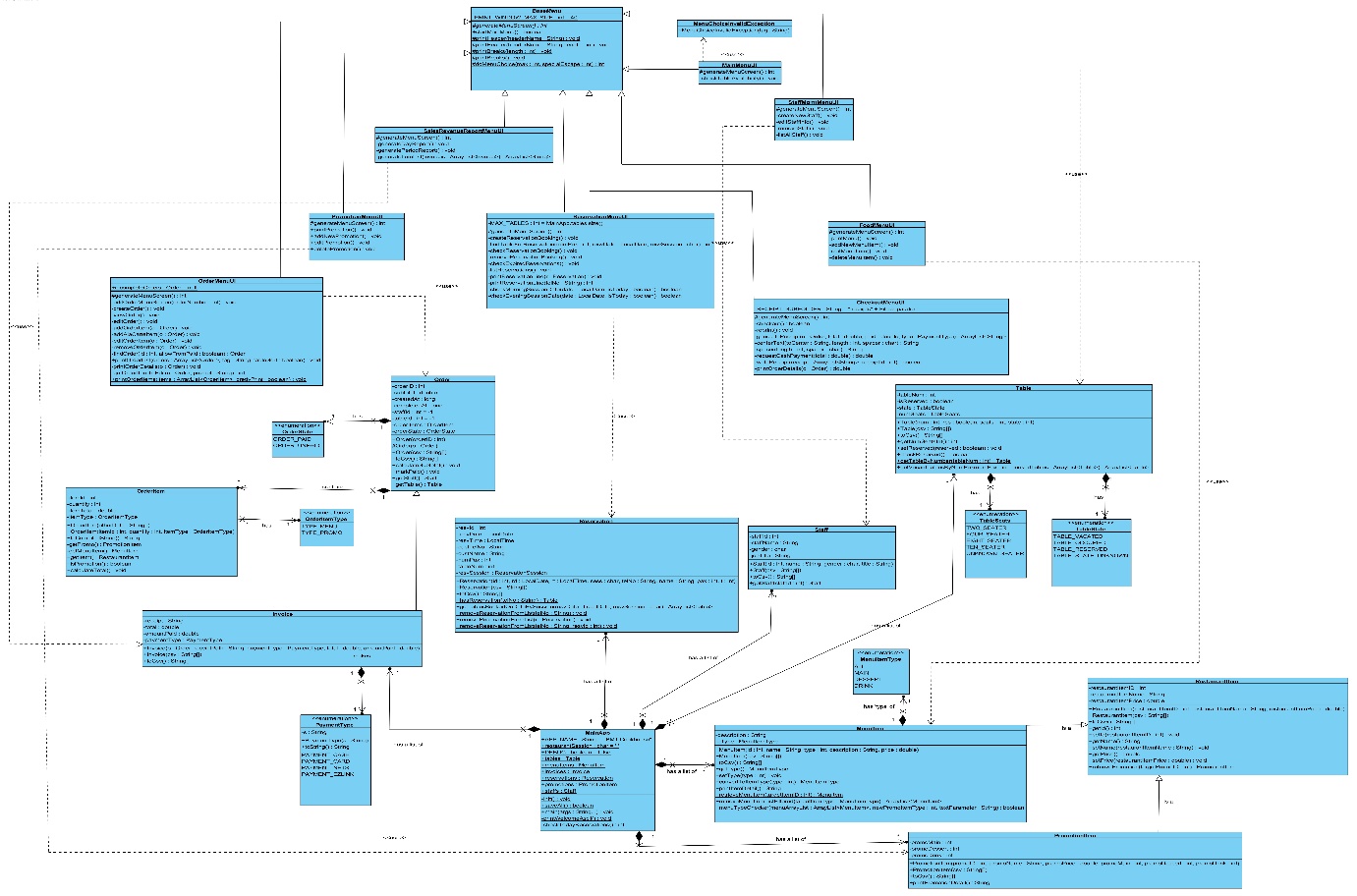
In this case, our application can cater to a new Customer class, storing all the essential Customer information like name, telephone number, date of birth, membership ID, and so on. This removes the need for separate attributes such as customer telephone number and customer name for the Reservation class and the customer telephone number for identifying a reservation on creating an Order. This thus adheres to the Open-Closed Principle.

### Staff Login Feature

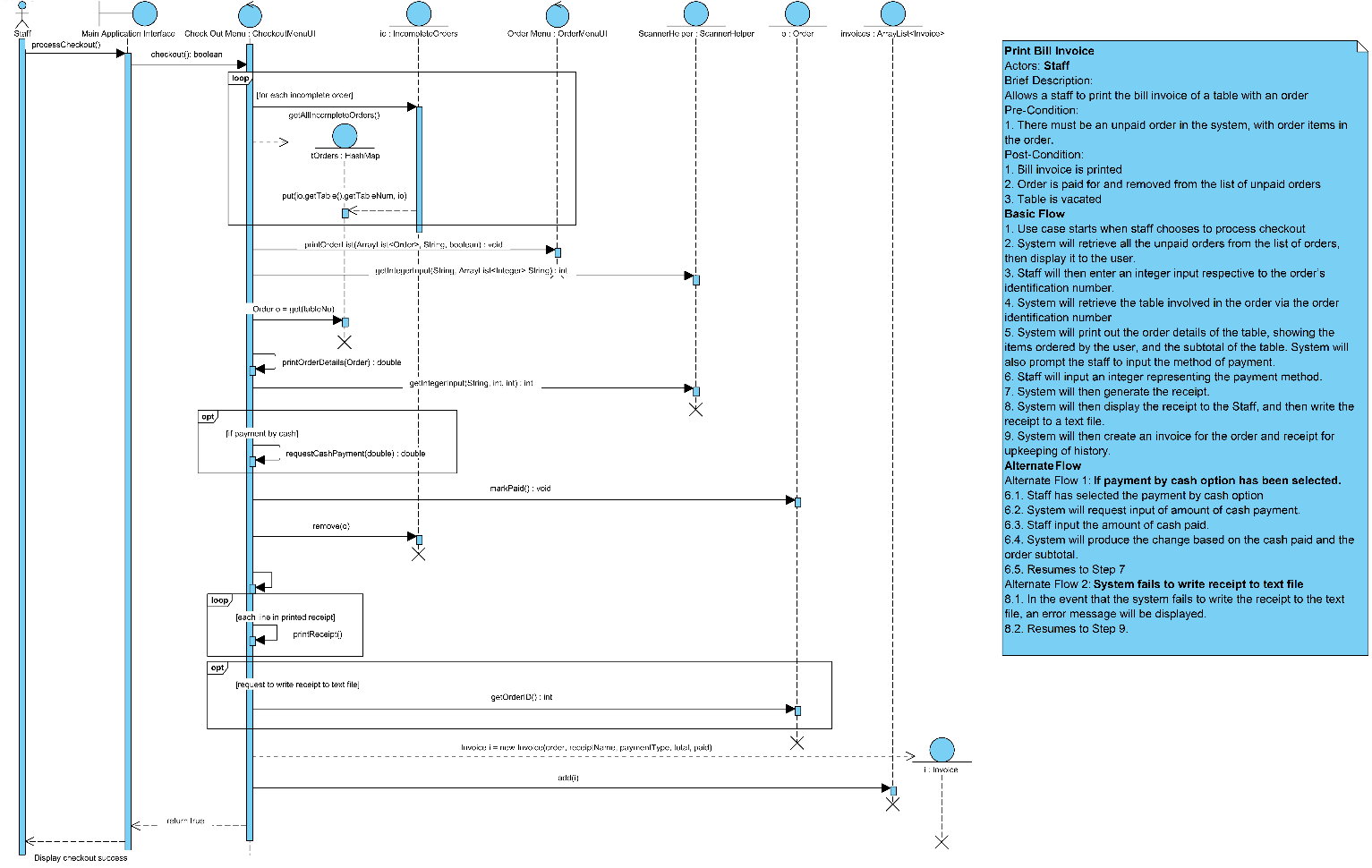
The staff login feature will realise restaurant staff being able to log in as their actual identity instead of the currently feature which sees the entire application being the entirety of all the staffs in the restaurant.

With this feature, the Staff class would have a more substantial reason to be a class on its own, as each Staff would have their own capability. For example, a waiter would be only able to take orders. A supervisor can take orders, manage reservations and checkout orders to print bill invoices. A restaurant manager can do all the above, and additionally, print sales revenue. This would also realise the Single Responsibility Principle where it is the Staff with respective capabilities being able to perform certain functions, instead of the main application interface.

# UML Class Diagram for RRPSS (separate file outside of report)



# Sequence Diagram for Print Bill Invoice (separate file outside of report)



# Test Cases

## Reservation Management

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test case | Expected outcome | Actual outcome |
| 1. | User makes reservation with all fields appropriately filled in, with available tables | User’s reservation made under his name |  |
| 2. | User makes reservation, all fields appropriately filled in, but all reservation slots for that session has been booked | User will be asked to input another date |  |
| 3. | Checking and removing of reservation booking – telephone number linked to more than 1 reservation | User will be prompted to input reservation ID, thereafter the reservation will be removed. |  |
| 4. | Checking for expired reservations – if there are some reservations expired on **runtime** | Reservations will be removed. |  |
| 5. | Checking for invalid inputs | Respective error messages will be shown |  |

## Check for Table Availability

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test case | Expected outcome | Actual outcome |
| 1. | User checks for table availability | Lists out all available tables and how many seats |  |

## Create and Add Items to Order

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test case | Expected outcome | Actual outcome |
| 1. | Create Order | Order created successfully |  |
| 2. | Add item to order | Item Added to Order |  |
| 3. | Invalid Data Entries | Appropriate Error Messages |  |
| 4. | Invalid Inputs | Appropriate Error Messages displayed |  |

## Checkout Order and Print Bill Invoice

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test case | Expected outcome | Actual outcome |
| 1. | Checkout Order | Successful Checkout and Receipt Printing |  |
| 2. | Invalid Checkout Entries | Appropriate Error Message Displayed |  |
| 3. | Insufficient Cash Payment | Appropriate Error message and prompt to enter until enough cash |  |
| 4. | Check Table Cleared after Checkout | Table becomes available again | Before: After: |

## Generating Sales Revenue

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test case | Expected outcome | Actual outcome |
| 1. | Input a single date | Display total quantity of items sold, number of orders, and total revenue. |  |
| 2. | Input 2 range of dates (30 days) | Display total quantity of items sold, number of orders, and total revenue. |  |

## Menu Item Management

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test case | Expected outcome | Actual outcome |
| 1. | Create or update new menu item | Item added or updated successfully. |  |
| 2. | Remove menu item | Item removed successfully |  |
| 3. | Invalid inputs | Respective error messages shown |  |

## Promotion Management

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test case | Expected outcome | Actual outcome |
| 1. | Create or update new Promotion | Item added or updated successfully. |  |
| 2. | Remove promotion | Promotion removed successfully |  |
| 3. | Invalid inputs | Respective error messages shown |  |

