# 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 |
| --- | --- | --- | --- |
| Brendan Ang Wei Jie | CZ2002 | SDDP1 |  |
| Chloe Heng Yi Ning | CZ2002 | SDDP1 |  |
| Malthus Oh Teng Jie | CZ2002 | SDDP1 |  |
| Chua Gim Aik (Cai JinYi) | CZ2002 | SDDP1 |  |
| Leow Ken Hing Bryan | CZ2002 | SDDP1 |  |

Important notes:

1. Name must **EXACTLY MATCH** the one printed on your Matriculation Card.

# 1. Design Considerations

This section of our report will explain and illustrate our use of both Object Oriented Programming (OOP) principles and S.O.L.I.D principles in our Restaurant Reservation and Point of Sale System (RRPSS).

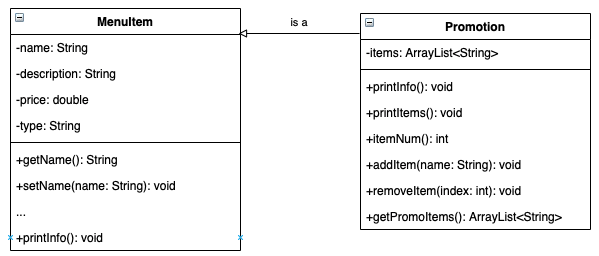
## **1.1. Object-Oriented Principles**

### 1.1.1. Abstraction

| Abstraction refers to consolidating the essential and general attributes and functions of real-world objects. One example of abstraction in our design is the use of our class Staff, seen in **Diagram 1**. The *Staff* class aims to model an employee at the restaurant by distilling the most essential features of an employee into attributes: name, gender, staff ID, and their title. | ***Diagram 1*** |
| --- | --- |

### 1.1.2. Inheritance and Polymorphism

Inheritance allows new classes (subclasses) to be derived from existing classes (superclass) by absorbing their attributes and behaviours, while adding new capabilities in the subclass if needed. One example of inheritance in our design is the inheritance of the *Promotion* class from the *MenuItem* class.



**Diagram 2**

The *MenuItem* class models a normal menu item that can be added onto a menu. It has attributes like name and price, getters and setters for each of these attributes, and a *printInfo()* method that prints these attributes in a nicely formatted string. The *Promotion* class models a set package with the same attributes as a *MenuItem,* along with a list of food items. Since a menu is just an *ArrayList* of *MenuItem* objects, we made *Promotion* a subclass of *MenuItem* class so that it can be upcasted and added to menus as well.

Closely related to inheritance, polymorphism is the ability of subclass objects in an inheritance relationship to be interacted with as if it is a parent class object, while still retaining the special functionality of that specific subclass. Since *Promotion* has an additional item list as an attribute, *printInfo()* needs to print every item in this list on top of the fields in *MenuItem.* To add this functionality, *printInfo*() in *Promotion* overrides *printInfo()* in *MenuItem*. This way, when classes call *printInfo()* on a menu, it will print different things depending on whether the underlying object is a *MenuItem* or *Promotion* instance.

### 1.1.3. Encapsulation & Information Hiding

| Encapsulation and information hiding build a barrier to protect an object’s *private* data, allowing access only through *public* getters and setters. This prevents unwanted access or modification from external classes.  We applied these concepts in designing the *MenuApp* class. All functions within the *MenuApp* class are *private*, besides 3 functions: *getNormalCatalogFunctions*(), *getPromoCatalogFunctions*(), and *getMenuFunctions*(). These 3 functions need to be *public* since they interface with the external *RRPSS* class, which acts as the boundary of control for the entire program. All other functions are *private* since they are only called by one of these 3 *public* functions, and calling them in any other external class would not make sense. Doing so could also trigger errors or illogical code flow. | ***Diagram 3*** |
| --- | --- |

## **1.2. SOLID Design Principles**

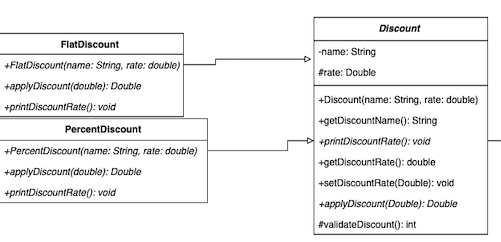
### 1.2.1. Single Responsibility Principle

The Single Responsibility Principle (SRP) details how there should never be more than one reason for a class to change. This also means that classes should only assume 1 responsibility, or “do one thing well”. In our program, we designed our main boundary and control classes with this principle in mind.

| We decided to split our control and boundary logic into multiple *XXXApp* classes which control different parts of our program instead of using one big control class. Examples include *MenuApp*, *StaffApp*, *ReservationApp* which all focus on a single portion of our program. For example, *StaffApp* only handles the creation and management of *Staff* class objects, and nothing else.  This produces high cohesion within each *XXXApp* class as they are in charge of one thing. Our “control center” class *RRPSS* class which delegates control flow to each of these *XXXApp* classes is also highly cohesive. Our design also produces low coupling, where entity classes are not overly dependent on each other and only need to interface with their corresponding *XXXApp*. For example, the *Table* class only needs to interface with the *ReservationApp* class. This minimises inter-dependency between classes and reduces “code change ripples” | **Diagram 4** |
| --- | --- |

### 1.2.2. Open-Closed Principle (OCP)

The Open-Closed Principle (OCP) defines that a module should be open for extension but closed for modification. In other words, we want to be able to change what modules do without changing their source code. As such, we decided to define our *Discount* class as an abstract class. This is because we foresee there being numerous types of discounts that will need their own specific *applyDiscount*() function implementation. For example, there could be flat amount discounts (e.g. take $5 off the order) or percentage discounts (e.g. take 10% off the order). There could also be more complex discount structures that our team cannot account for comprehensively (e.g. take 10% off the first $50 of the bill).



**Diagram 5**

As such, in line with the OCP, we decided to make *Discount* an abstract class and made the *FlatDiscount* and *PercentDiscount* classes extend it. This way, classes that interact with *Discount* objects can access the different functionalities of *FlatDiscount*, *PercentDiscount*, and future discount structures without requiring a change in the source code of *Discount*. Whenever we want to add a new discount structure for the restaurant, we simply need to inherit from the *Discount* abstract class and add any additional functionality in our new concrete class without changing the code for the *Discount* abstraction, thus fulfilling the OCP.

### 1.2.3. Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP) defines that the subclasses must be substitutable for their base class, requiring no more or no less and offering no more or no less as compared to their base class. Referring to **Diagram 4** again, *MenuItem* and *Promotion* objects need to be stored in the same *ArrayList* of *MenuItem* objects to act as menus for the restaurant. Classes that interact with these menus will need to be able to work with both *MenuItem* and *Promotion* objects interchangeably. As such, we made sure that the *Promotion* subclass implements every method in the *MenuItem* class with the same parameters (ensuring equal or weaker pre-conditions) and same return types (ensuring equal or stronger post-conditions). By doing so, *Promotion* is substitutable for its base type *MenuItem* objects and fulfills the LSP.

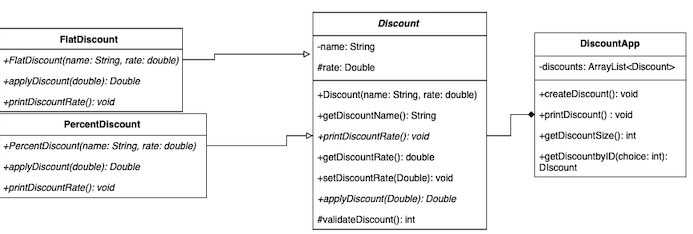
### 1.2.4. Interface Segregation Principle (ISP)

The Interface Segregation Principle defines that classes should not depend on interfaces that they do not use. Interfaces should be separated when necessary, and classes should only implement those interfaces which consist of methods it would use.

| One such specific interface we implemented is *RW* whose purpose is to mandate the *read()* and *write()* methods which read from and write to external files. Every class that requires external file access will implement this specific interface. One of these classes is *SalesReportApp* which stores its attributes into external files which will be used to restore the state of the program upon reboot. This *RW* interface replaces a general interface that defines all external-facing methods (e.g. IO, APIs), which avoids “code change ripples”. | **Diagram 6** |
| --- | --- |

### 1.2.5. Dependency Injection Principle (DIP)

The Dependency Injection Principle states that higher level modules in the application should not depend on lower level modules, but both should depend on abstractions like abstract classes or interfaces.



***Diagram 7***

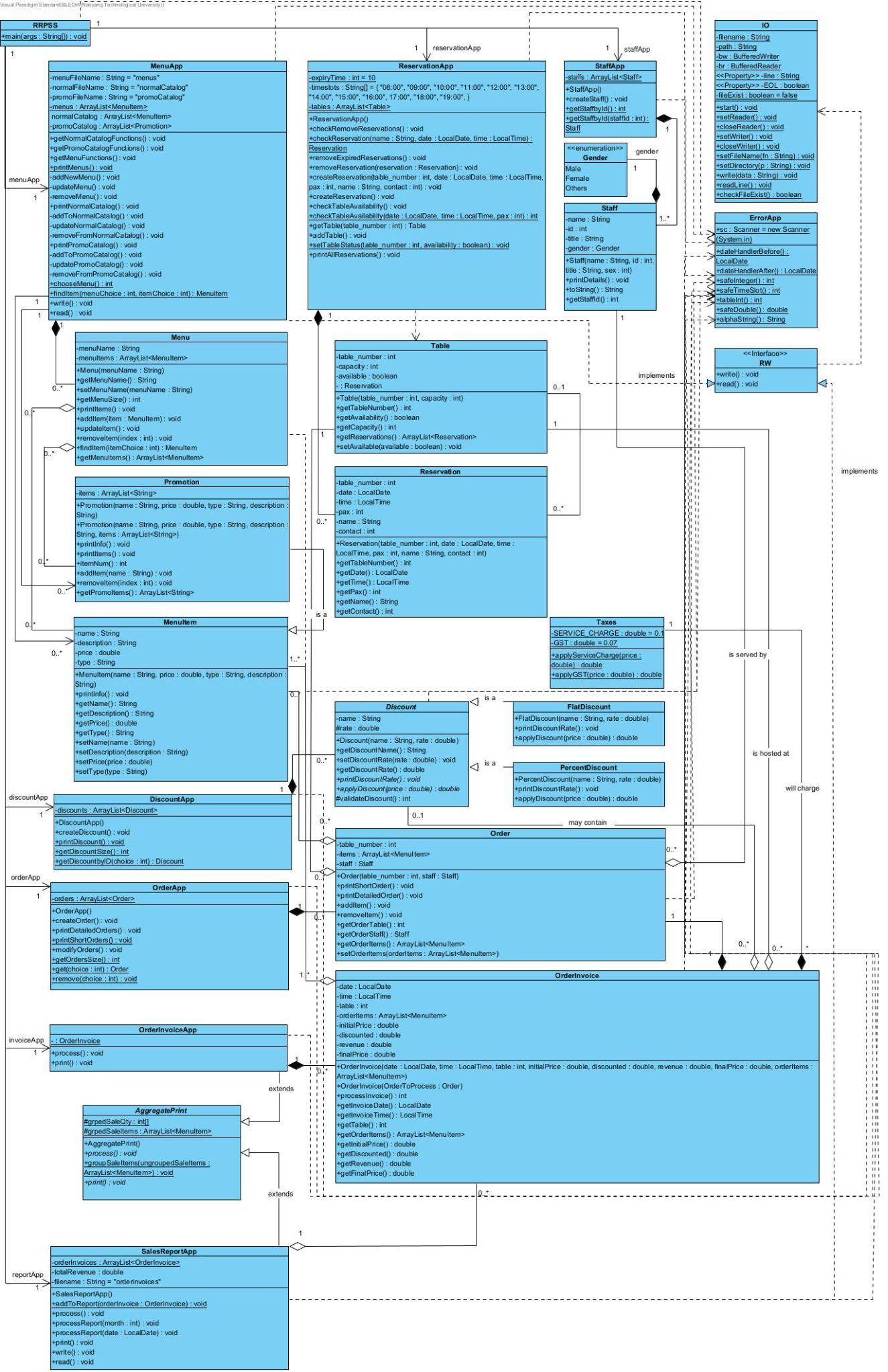
One example is the *DiscountApp* class which needs to interact with the *Discount* class when a customer has a valid discount and would like to use it at the point of payment. However, different discounts could have different methods of computation. For example, one discount type could be a flat rate of $10 off the total bill. Another discount type could take 5% off the total bill. If *Discount* was a concrete class, every new discount type will require code modification in the *applyDiscount()* method. This will trigger code changes for *OrderInvoice* and all other classes that interact with the *Discount* class. Instead, by implementing *Discount* as an abstract class and making new discount types extend *Discount,* every discount type can implement *applyDiscount()* with its own specific functionality. *DiscountApp* will depend on the *Discount* abstraction while *applyDiscount()* will be implemented by concrete classes *PercentDiscount* and *FlatDiscount* that extend *Discount*, fulfilling DIP.

## **1.3. Assumptions**

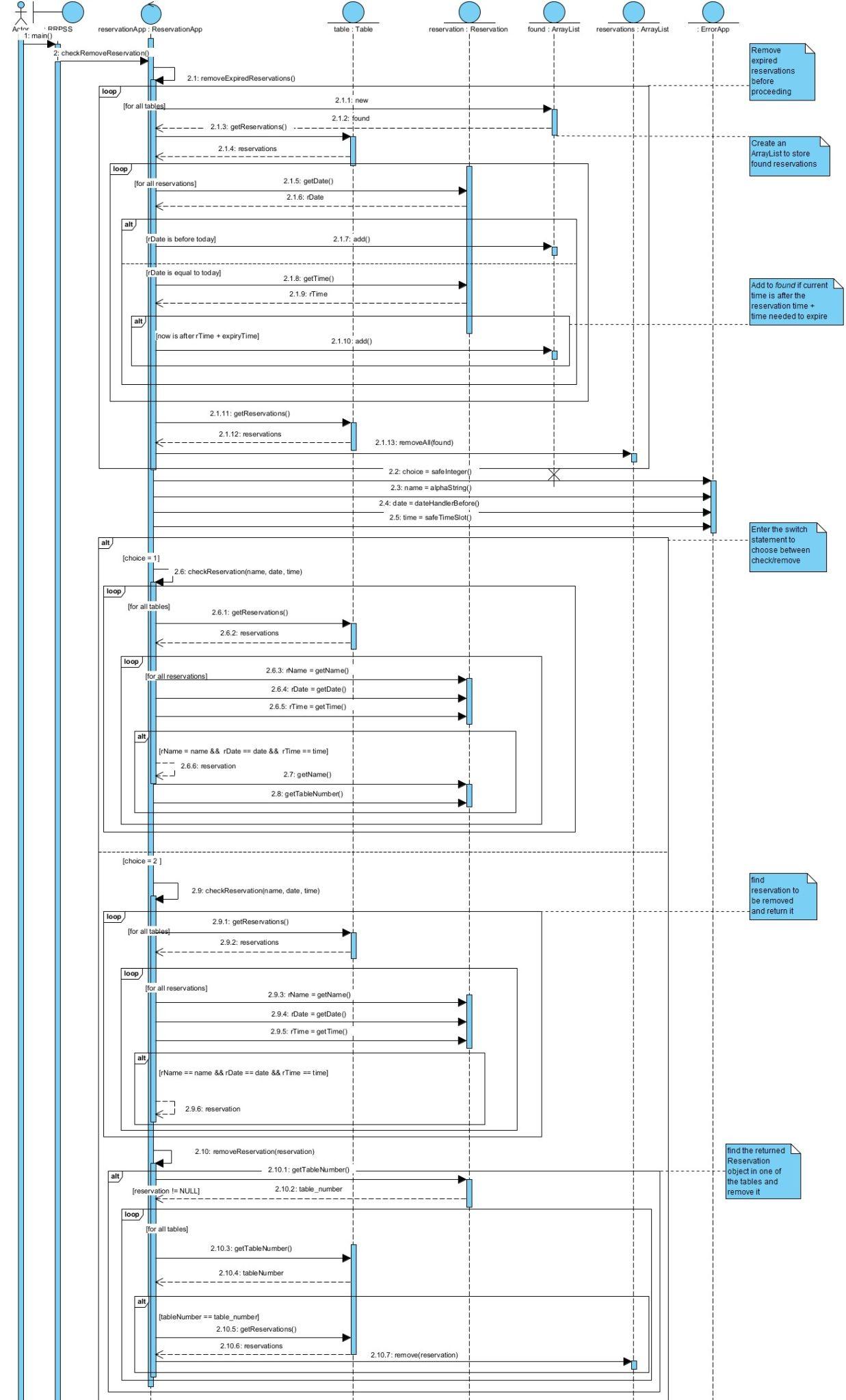
In building our program, our team made a few assumptions:

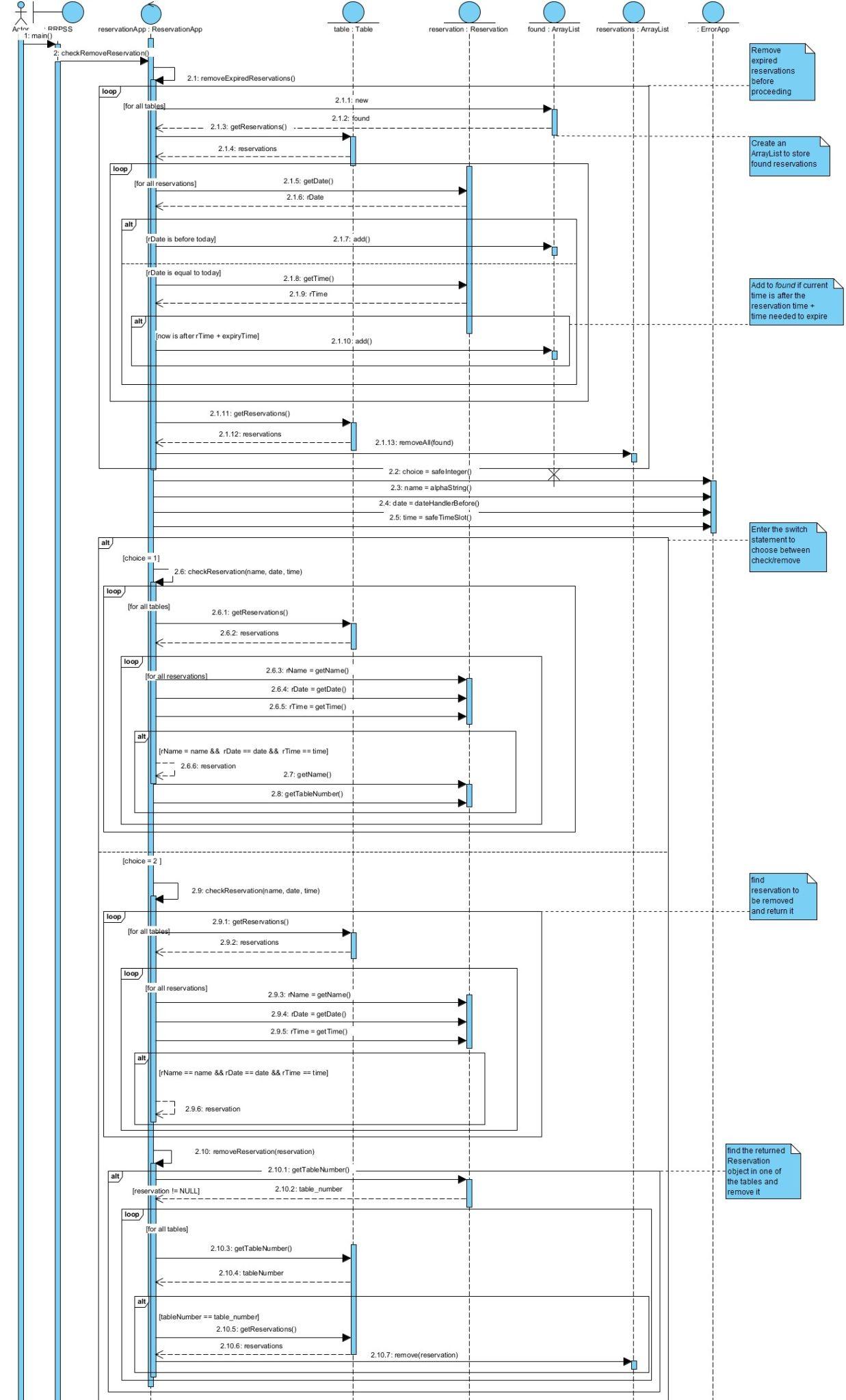
* Restaurant operates from 8am to 8pm and accepts both **reservations** and **walk-ins**
* A **table** is not available if there is a reservation for a specific 1-hour time slot and the slot after.
* Restaurants have **catalogs** that act as archives for menu items and promotions. They will create **menus** by adding items from these catalogs into menus.
* A **staff** uniquely identifies themselves by the employee ID only, hence there is no checking done if the same name for a staff has been inputted.
* One **order** is facilitated by one table and one staff only. One **table** can only host one order, but one staff can serve multiple orders.
* **Discounts** can only be either a flat rate or a percentage discount. Special case discounts, e.g. ‘buy 2nd drink at 50% off’, will not be possible to add in our current system.
* There can be multiple definitions for the same **discount** name.
* An **order** can have a price of $0.00 after **discount**. We ask the user to confirm this before proceeding. This is to simulate an edge case where the discount offers a free meal.
* An **order** can be processed even without any order items. In this case, we will only free the table and not print any order invoice. This is to simulate an edge case where the customer sits down at the restaurant, but changes their mind and leaves without eating.
* An **order invoice** will show the following, applied in order: initial price, price after discount, price after service charge, and price after GST. Price after GST is the important field, since it determines what the customer has to pay.
* A **sales report** will only show items sold, their quantity, and total revenue.
* **Promotional sets** will be displayed as an item on their own, instead of breaking it down into their constituent menu items.
* Revenue in **sales reports** is defined by price after discount and service charge, but before GST. This is because GST is collected by the government, and not kept by the restaurant as revenue.

## **3. Class Diagram**



## **4. Sequence Diagram**



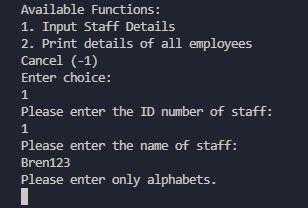


## **5. Test Cases (not covered in demo)**

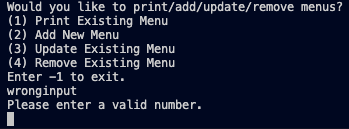
During our video demonstration, we showcased most of the essential functionalities of our program. Due to time constraints, we will showcase our other essential test cases and error handling here.

Check Validity of User Inputs

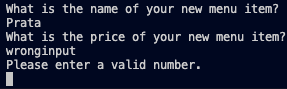
Checks for valid names (i.e. only alphabetical characters) when inputting details for new staff.



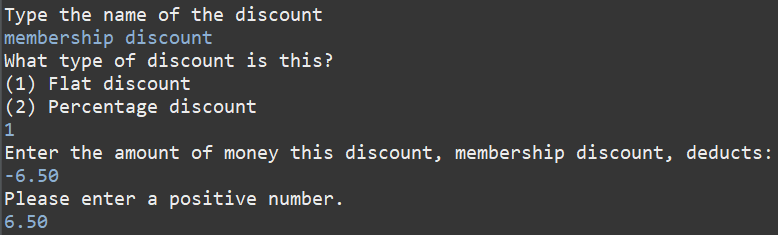
Checks for valid option input from users when navigating the program.



Checks for valid price input by users when creating new menu items.

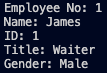


Checks for valid discount input from users

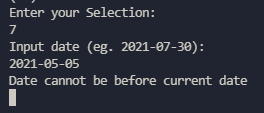


Restaurant-specific Use Case Checks

Checks for invalid ID input i.e ID has been taken by another staff.



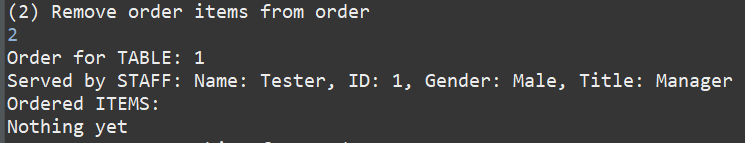
Reservation cannot be made before the current date.



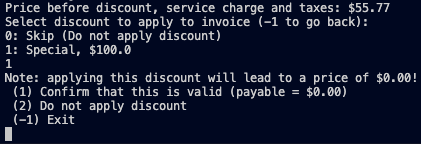
If catalogs are empty and users create new menus, remind users to create menu items first.



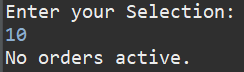
Order items cannot be removed if there is nothing to remove.



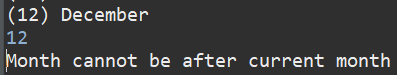
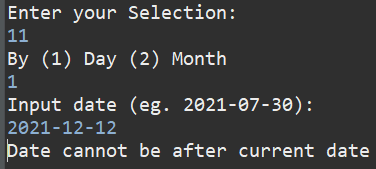
If a discount causes the price of an order to be <=$0.00, the program will warn users before checking out.



Order invoice cannot be printed when there are no active order invoices,



Sales reports cannot be printed for after the current date.



If order invoices don’t exist for a particular date or month OR if data is empty, the sales report portion of our program will flag out that no order invoices were found and thus a sales report cannot be generated.

