**Swinburne University of Technology**

*​School of Science, Computing and Engineering Technologies*

**​ASSIGNMENT AND PROJECT COVER SHEET**

Shape**​**

​Subject Code: SWE30003          Unit Title: Software Architectures and Design

​Assignment number and title: 3, Design Implementation Due date: 11:59pm, 31st May 2024

​

​Tutorial Day and Time: Wednesday 6.30pm Project Group: Wed 6.30pm – Group 3

​

​Tutor: Mandeep Dhindsa

Shape​

​**To be completed as this is a group assignment**

​We declare that this is a group assignment and that no part of this submission has been copied from any other student's work or from any other source except where due acknowledgment is made explicitly in the text, nor has any part been written for us by another person.

​

​ID Number Name Signature

​103795587 Jade Hoang Jade Hoang

​103795561 Henry Le Henry Le

​103541023 Thanh Nam Vu Thanh Nam Vu

​103844421 Dang Khoa Le Dang Khoa Le

Shape​

​Marker's comments:

​

​

​

​Total Mark:

Shape​



Object Design IMPLEMENTATION & Reflection (Assign 3)

Restaurant Information System



Jade Hoang, Henry Le, Thanh Nam Vu, Dang Khoa Le

SWE30003 – Software Architecture and Design

**contents**

[1. Detailed Design and Reflection 3](#_Toc167649988)

[1.1 Design changes and justifications 3](#_Toc167649989)

[1.2 Updated Design 5](#_Toc167649990)

[1.2.1 Chosen classes 5](#_Toc167649991)

[1.2.2 Discarded classes 5](#_Toc167649992)

[1.2.3 UML Diagram 6](#_Toc167649993)

[1.2.4 CRC Cards 7](#_Toc167649994)

[1.2.5 Bootstrap Process 11](#_Toc167649995)

[1.2.6 Verification 12](#_Toc167649996)

[2. Design Quality 17](#_Toc167649997)

[2.1 Good aspects 17](#_Toc167649998)

[2.2 Missing from original design 17](#_Toc167649999)

[2.3 Flawed aspects of original design 17](#_Toc167650000)

[2.4 Level of interpretation required 17](#_Toc167650001)

[3. Lessons Learned 18](#_Toc167650002)

[3.1 From the object design 18](#_Toc167650003)

[3.2 From implementation 18](#_Toc167650004)

[3.3 From working in a team 18](#_Toc167650005)

[4. Implementation 19](#_Toc167650006)

[4.1 Features implemented 19](#_Toc167650007)

[4.2 Platforms 19](#_Toc167650008)

[4.3 Deployment instructions 20](#_Toc167650009)

[4.4 Evidence 21](#_Toc167650010)

[5. Appendix 32](#_Toc167650011)

# Detailed Design and Reflection

## Design changes and justifications

* **Statistics class addition:**
  + **Initial design**: Did not include a dedicated Statistics class for generating order statistics reports.
  + **Change**: Added a Statistics class.
  + **Justification:** The Statistics class is now added to encapsulate responsibilities related to generating order statistics reports, displaying the report to the screen, and saving the report to file storage. It also encapsulates fields such as the order file path and the report file path. This change promotes the separation of concerns by delegating statistics-related functionalities to a dedicated class.
* **Invoice class removal**:
  + **Initial design**: The design included an Invoice class for consolidating and making payments for all orders at once.
  + **Change**: Removed the Invoice class.
  + **Justification**: Payments are now made immediately after placing each order to simplify the process. With immediate payment, there is no need for a separate Invoice class. Additionally, the Receipt class has already included the details needed to be displayed in the Invoice so keeping both the Receipt and Invoice class would be redundant.
* **Delivery address property**:
  + **Initial design**: The delivery address was part of the Delivery class.
  + **Change**: Moved the delivery address property to the Order class.
  + **Justification**: Delivery addresses are specific to each order and are used to manage the delivery process for individual orders. The Order class now encapsulates all information related to a specific order, including delivery details for takeaway orders. The Delivery class now focuses on functionalities such as forwarding orders to third parties and updating delivery status.
* **Payment and Order interaction**:
  + **Initial design**: Included an Invoice class that handled payments for consolidated orders.
  + **Change**: Removed the Invoice class and made the Payment class interact directly with the Order class.
  + **Justification:** Since the Invoice class is removed, the Payment class now collaborates directly with the Order class to obtain order details. Payment becomes the responsibility of the Payment class, which interacts with the Customer class once the customer completes the order placement. The Customer class now includes responsibilities related to placing orders, such as adding to the cart, removing items from the cart, and viewing the cart.
* **Make payment responsibility**:
  + **Initial design**: The Customer class was responsible for making payments.
  + **Change**: Moved the make payment responsibility to the Payment class.
  + **Justification**: The Payment class is now solely responsible for handling the payment process. After the Customer class completes the order placement, it creates an instance of the Payment class and calls its makePayment() method. This change ensures a clearer separation of concerns, with the Payment class dedicated to payment-related functionalities.
* **Reservation’s Table Number property addition**:
  + **Initial design**: The Reservation class did not explicitly track the table number reserved by the customer.
  + **Change**: Added the table number property to the Reservation class.
  + **Justification**: This addition ensures clarity by separating reservation-specific details, such as the reserved table number, from the general table identifier stored in the Table class. It maintains encapsulation and improves the organisation of reservation-related information within the Reservation class.
* **Payment date property addition:**
  + **Initial design**: Did not specify the handling of payment dates in Payment class.
  + **Change**: Added a payment date property to the Payment class.
  + **Justification:** The Payment class is now responsible for knowing payment dates. It ensures that payment dates are recorded accurately and associated with each transaction. This change enhances the Payment class's functionality and ensures that all payment-related details are managed within one class.
* **Payment method property moved**:
  + **Initial design:** The Receipt class was responsible for knowing the payment method.
  + **Change:** Moved the payment method responsibility to the Payment class
  + **Justification:** The Receipt class should focus on generating receipts and confirming payments, while the Payment class manages the details of the payment methods.
* **Methods for Saving and Displaying details addition:**
  + **Initial design**: Did not consider methods for saving order details, reservation details, receipt details, and statistics reports to file storage and displaying those details.
  + **Change**: Added methods to the Order, Receipt, Reservation, and Statistics classes.
  + **Justification**: These additions support responsibilities related to saving detailed information to file storage and displaying them to the terminal. This ensures more comprehensive responsibilities within the system.
* **Bootstrap process**:
  + **Initial design**: Menu creates instances of MenuItem class.
  + **Change**: Program creates instances of MenuItem class.
  + **Justification**: The Menu class should primarily focus on maintaining and displaying the menu items, not on creating them. By creating MenuItem instances in the Program class, the Menu class’s AddMenuItem() method can be used to add items to the menu. This is useful for scenarios where menu items need to be temporarily removed from the Menu without deleting the menu item entirely.

## Updated Design

### Chosen classes

Statistics class is added:

* **Statistics:** This class is responsible for generating order statistics reports that include details such as total order, total revenue and all menu items ordered.

### Discarded classes

Invoice class is removed:

* **Invoice**: Since payments are now made immediately after placing each order to simplify the process. With immediate payment, there is no need for a separate Invoice class. Having an Invoice class will be redundant since it will have the same details , properties, and responsibilities as Receipt class.

### UML Diagram

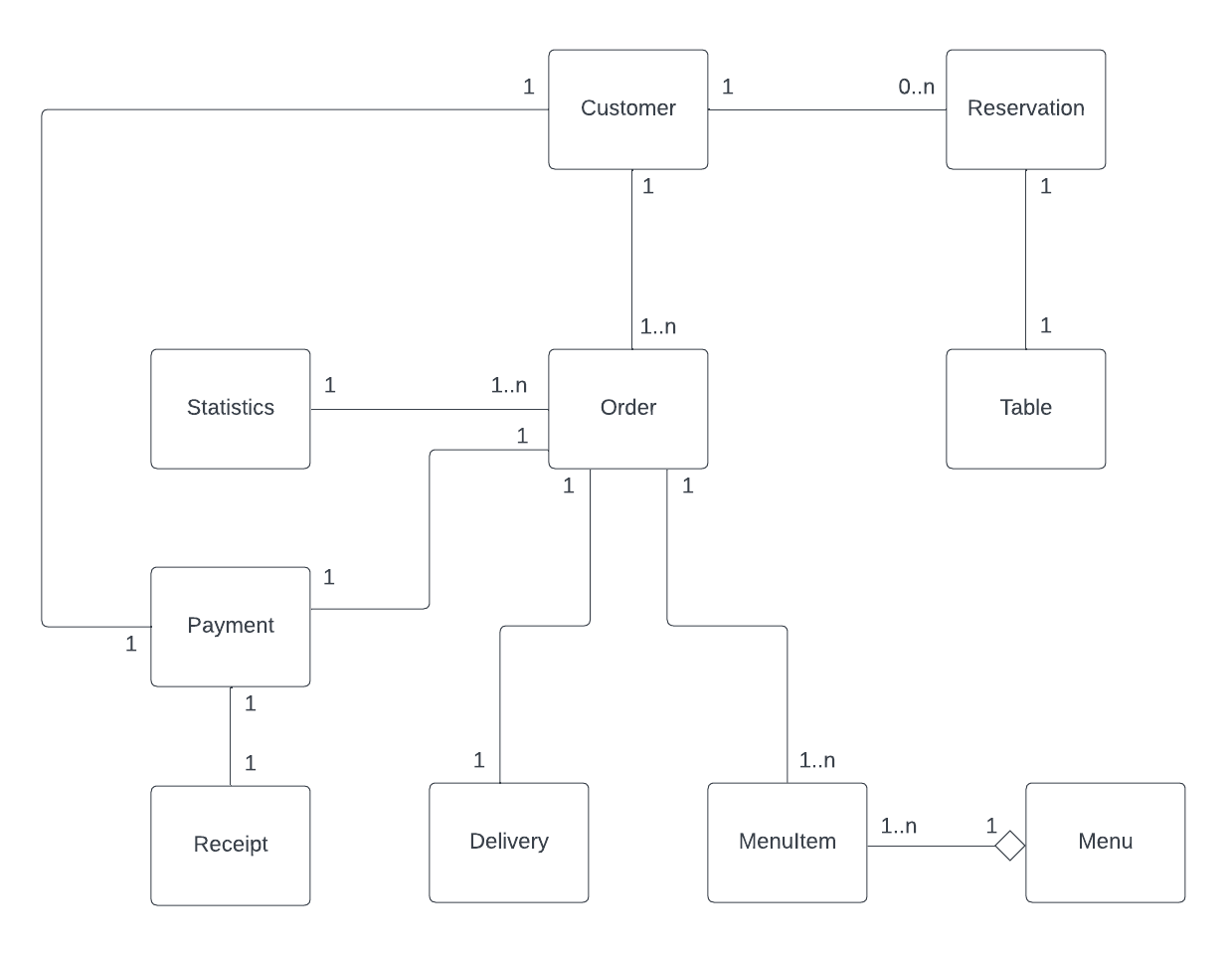


Figure 1 - UML Diagram

### CRC Cards

#### Customer

The Customer class represents a customer interacting with the restaurant system. It holds responsibilities such as knowing and providing the customer's name and contact number, as well as facilitating various interactions within the system.

|  |  |
| --- | --- |
| Class Name: Customer | |
| Responsibilities | Collaborators |
| Know customer’s name | N/A |
| Know customer’s contact number | N/A |
| Get customer’s name and contact number | N/A |
| Place order | Order, Menu, Payment |
| Add menu item to cart | Menu, MenuItem |
| Remove menu item from cart | MenuItem |
| View cart | MenuItem |
| Make reservation | Table, Reservation |

Table 1 - Customer CRC

#### Reservation

This class manages reservations made by customers within the RIS, including date, time, and party size. It ensures the proper allocation of tables based on customer requests.

|  |  |
| --- | --- |
| Class Name: Reservation | |
| Responsibilities | Collaborators |
| Know the reservation ID | N/A |
| Know the table number reserved by customer | N/A |
| Know the reservation date/time, number of guests | N/A |
| Find available tables for the requested date/time and seating capacity | Table |
| Save reservation details (reservation ID, date/time, number of guests) along with reserved table number to file storage | N/A |
| Load existing reservations from file storage | N/A |

Table 2 - Reservation CRC

#### Table

This class represents an individual dining table within the restaurant and manages table information such as seating capacity, table number and table availability status.

|  |  |
| --- | --- |
| Class Name: Table | |
| Responsibilities | Collaborators |
| Know table number | N/A |
| Know table seating capacity (i.e., number of seats) | N/A |
| Know the table availability status | N/A |
| Update table availability status (reserved or available) | N/A |
| Check if the table can accommodate a given number of people | N/A |
| Check if the table is available | Reservation |

Table 3 - Table CRC

#### Menu

This class manages the restaurant's menu, including adding, updating, and removing menu items. The Menu class manages a collection of MenuItem objects.

|  |  |
| --- | --- |
| Class Name: Menu | |
| Responsibilities | Collaborators |
| Maintain a list of menu items | MenuItem |
| Add menu item | MenuItem |
| Remove menu item | MenuItem |
| Get a specific menu item by index | MenuItem |
| Display menu | MenuItem |

Table 4 - Menu CRC

#### MenuItem

This class represents an individual menu item within the menu, including its name, description, price, and category.

|  |  |
| --- | --- |
| Class Name: MenuItem | |
| Responsibilities | Collaborators |
| Know the item’s name, description, and price | N/A |

Table 5 - MenuItem CRC

#### Payment

The Payment class encapsulates the functionality related to processing payments for orders. It allows users to make payments using credit or debit cards. It also interacts with an external payment gateway and generates receipts for successful transactions.

|  |  |
| --- | --- |
| Class Name: Payment | |
| Responsibilities | Collaborators |
| Know payment date | N/A |
| Forward payment details to an external payment gateway | N/A |
| Make payment | MenuItem |
| Generate receipt | Receipt, MenuItem |
| Obtain payment type from the user | N/A |
| Obtain card number form the user | N/A |
| Obtain card expiration date from the user | N/A |
| Obtain card CVV from the user | N/A |

Table 6 - Payment CRC

#### Order

The Order class represents a customer's order within the RIS. It captures the details of the ordered items, their quantities, and facilitates communication with the kitchen.

|  |  |
| --- | --- |
| Class Name: Order | |
| Responsibilities | Collaborators |
| Know orderID | N/A |
| Know order date | N/A |
| Know order items | MenuItem |
| Know order’s delivery address | N/A |
| Know if the order is delivery or dine-in | N/A |
| Add menu item to order | MenuItem |
| Calculate the total price of the order | MenuItem |
| Display the details of the order | N/A |
| Save the order details to a file | N/A |
| Get delivery address from the user | N/A |
| Inform the order to kitchen | N/A |

Table 7 - Order CRC

#### Statistics

The Statistics class is responsible for generating order statistics reports based on data stored in an orders file. It reads information about orders from a specified file, processes the data to calculate various statistics such as total revenue and item counts, generates a report based on these statistics, and saves the report to a specified file path.

|  |  |
| --- | --- |
| Class Name: Statistics | |
| Responsibilities | Collaborators |
| Store the path to the file containing order data | N/A |
| Store the path to the file where the report will be saved | N/A |
| Generate the order statistics report | N/A |
| Save the report to a file | N/A |
| Display the generated report | N/A |

Table 8 - Invoice CRC

#### Receipt

The Receipt class serves as a confirmation document for a successful payment within the RIS. It is responsible for managing the display and saving of receipt information.

|  |  |
| --- | --- |
| Class Name: Receipt | |
| Responsibilities | Collaborators |
| Display receipt details | MenuItem |
| Save the receipt details to a file | MenuItem |

Table 9 - Receipt CRC

#### Delivery

Represents the delivery service for takeaway orders and manages the delivery process. It acts as an intermediary between the RIS and a third-party delivery service. In practical implementation, this class would handle the integration with external services through API communication.

|  |  |
| --- | --- |
| Class Name: Delivery | |
| Responsibilities | Collaborators |
| Forward order details to a third-party delivery service for processing | Order |
| Update delivery status | N/A |

Table 10 - Delivery CRC

### Bootstrap Process

* **Program** creates an instance of the **Menu** class.
* **Program** creates instances of **MenuItem** class.
* **Program** creates instances of the **Table** class to manage table arrangements.
* For each customer interaction, **Program** creates an instance of the **Customer** class.
* When a customer makes a reservation, **Customer** creates an instance of **Reservation** class.
* When the customer places an order, **Customer** creates an instance of the **Order** class.
* **Customer** creates an instance of **Payment** class to process payment.
* **Payment** creates an instance of **Receipt** class to generate and manage the payment receipt.
* If the order requires delivery, **Order** creates an instance of **Delivery** class to handle the delivery process.
* When the user chooses to generate an order statistics report, **Program** creates an instance of the **Statistics** class.

### Verification

#### Make a reservation

In this scenario, the main **Program** calls the **MakeReservation** method of the **Customer** class, passing in a list of available tables. **Customer** prompts the customer for their name, contact number, number of guests, and desired reservation date/time. **Reservation** is called to load existing reservations from a file through the **LoadReservations()** method. **Reservation** finds available tables that can accommodate the requested number of guests and are not already reserved for the given date/time. It does this by calling the **CanAccommodate** method and checking table availability using the **IsTableAvailable()** method of **Table**. The list of available tables is returned to the **Customer**. A new **Reservation** object is created with the chosen table number, reservation date/time, and number of guests. The new reservation is saved to a file by calling the **Save** method of the **Reservation**. The chosen table is marked as reserved by calling the **Reserve** method of the **Table**.

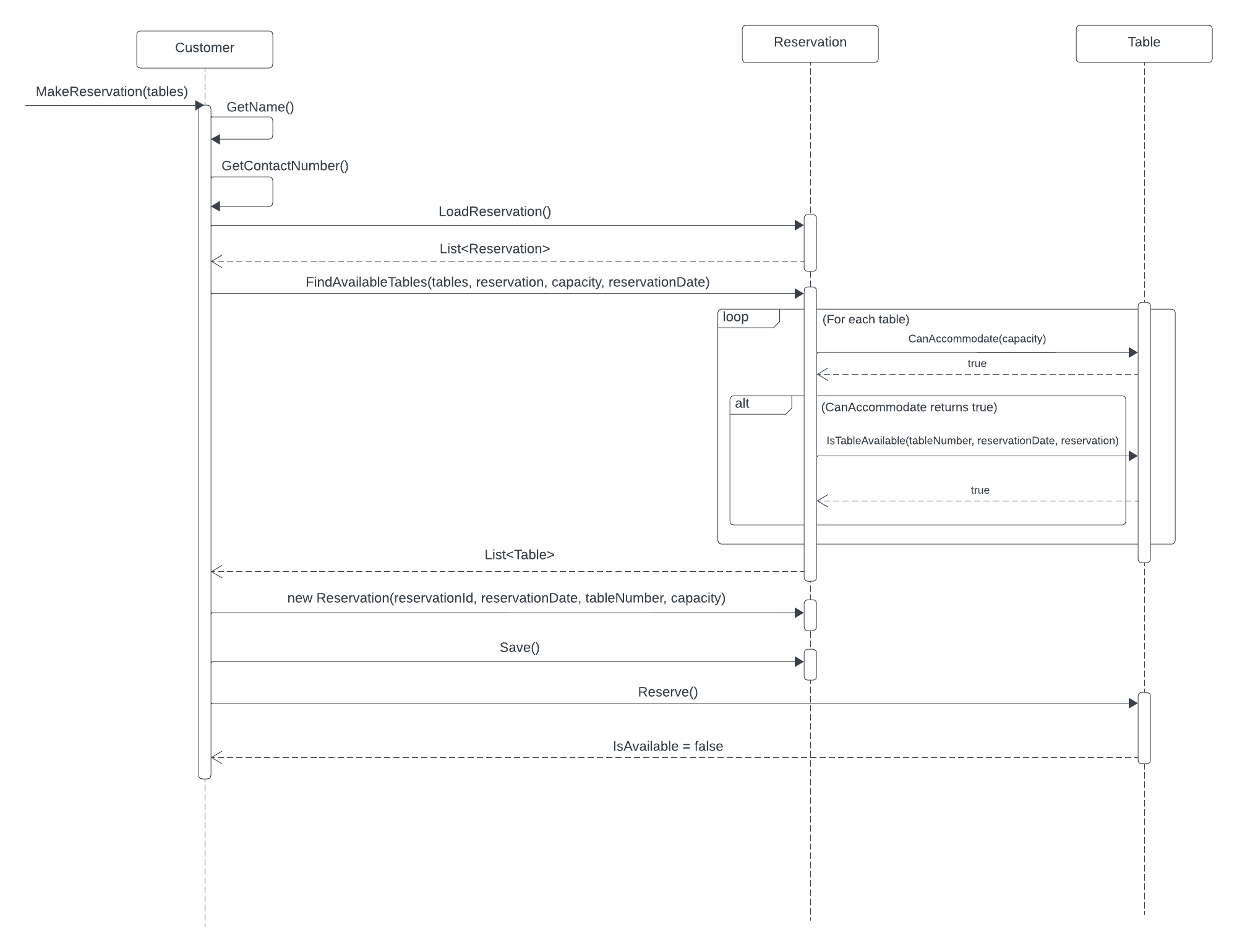


Figure 2 - Make a reservation UML sequence diagram

#### Place an order

The order process begins with calling the **PlaceOrder** method of the **Customer** with the menu as a parameter. The **Customer** then enters their name (**GetName**) and contact number (**GetContactNumber**). The **Customer** requests the **Menu** to be displayed (**DisplayMenu**).

The **Customer** enters a loop to ‘add’, ‘remove’, and ‘view’ items in the cart, or enter ‘menu’ to request the menu to be displayed again (**DisplayMenu)**.

If the choice is “**add”**, **AddToOrder** method will be called and process these steps:

* Prompts the user to enter the number of the item (integer) they want to add to the cart.
* Validates the input to ensure it is a number within the range of available menu items.
* Adds the selected item to the cart if valid, or prints an error message if not.

If the choice is “**remove”**, **RemoveFromCart** method will be called and process these steps:

* Count the item from cart to checks if the cart is empty and informs the user if it is.
* Prompts the user to enter the number of the item (integer) they want to remove from the cart.
* Validates the input to ensure it is a number within the range of items in the cart.
* Removes the selected item from the cart if valid, or prints an error message if not.

If the choice is “**view”**, **ViewCart** method will be called and process these steps:

* Count the item from cart to checks if the cart is empty and informs the user if it is.
* If the cart is not empty, loop through each item and displays a list of items in the cart along with their prices.
* Displays the total cost of all items in the cart.

Then, if the choice is "**done**", which indicates the order is to be finalised and waiting for payment, the following steps occur:

* An **Order** instance is created with the current date.
* The **Customer** adds items to the **Order** within a nested loop (**AddItem**).

The **Customer** specifies whether the order is for delivery or dine-in (**AskDeliveryOrDineIn**). The order details are displayed (**DisplayOrder**). A **Payment** instance is created with the current date. The **Customer** attempts to make a payment (**MakePayment**) with the order details. If successful, the **Payment** class confirms success, and the order is saved (**Save**) and processed (**ProcessOrder**).

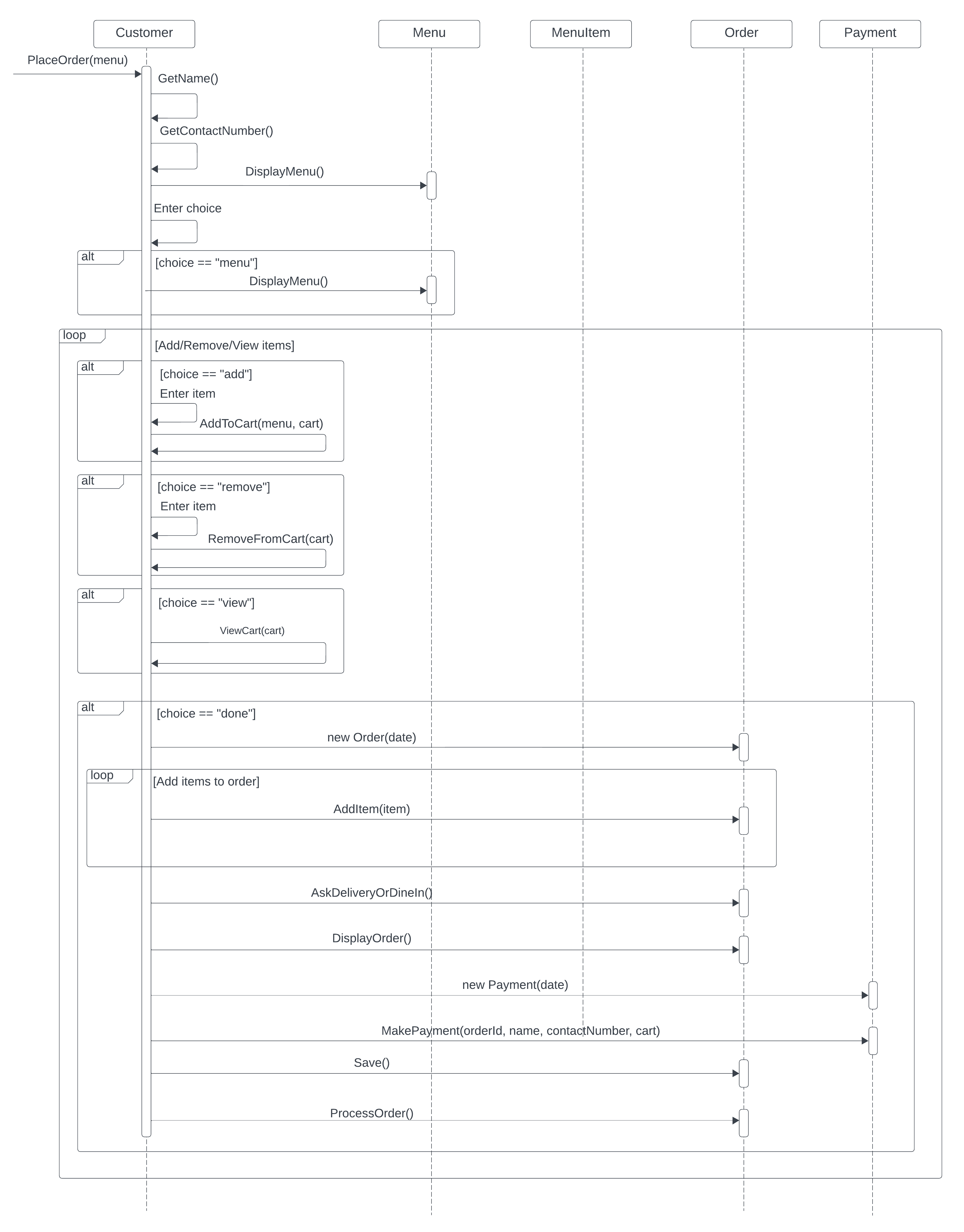


Figure 3 - Place an order UML sequence diagram

#### Make payment

To initiate a payment, **Customer** triggers the process by invoking the **MakePayment** method of the **Payment** class, passing **orderId**, **name**, **contact**, and **cart**. The **Payment** class gathers payment details by calling **GetPaymentType**, **GetCardNumber**, **GetExpirationDate**, and **GetCVV**. The **Payment** class forwards the gathered information to an external gateway for processing via **ForwardToExternalGateway**. If the payment is successful, **Payment** creates a new **Receipt** instance, displays the receipt details using **DisplayReceipt**, and saves the receipt information with **Save** method. Finally, **Payment** returns a success status (true) to the **Customer**. If the payment fails, **Payment** returns a failure status (false) to the **Customer**.

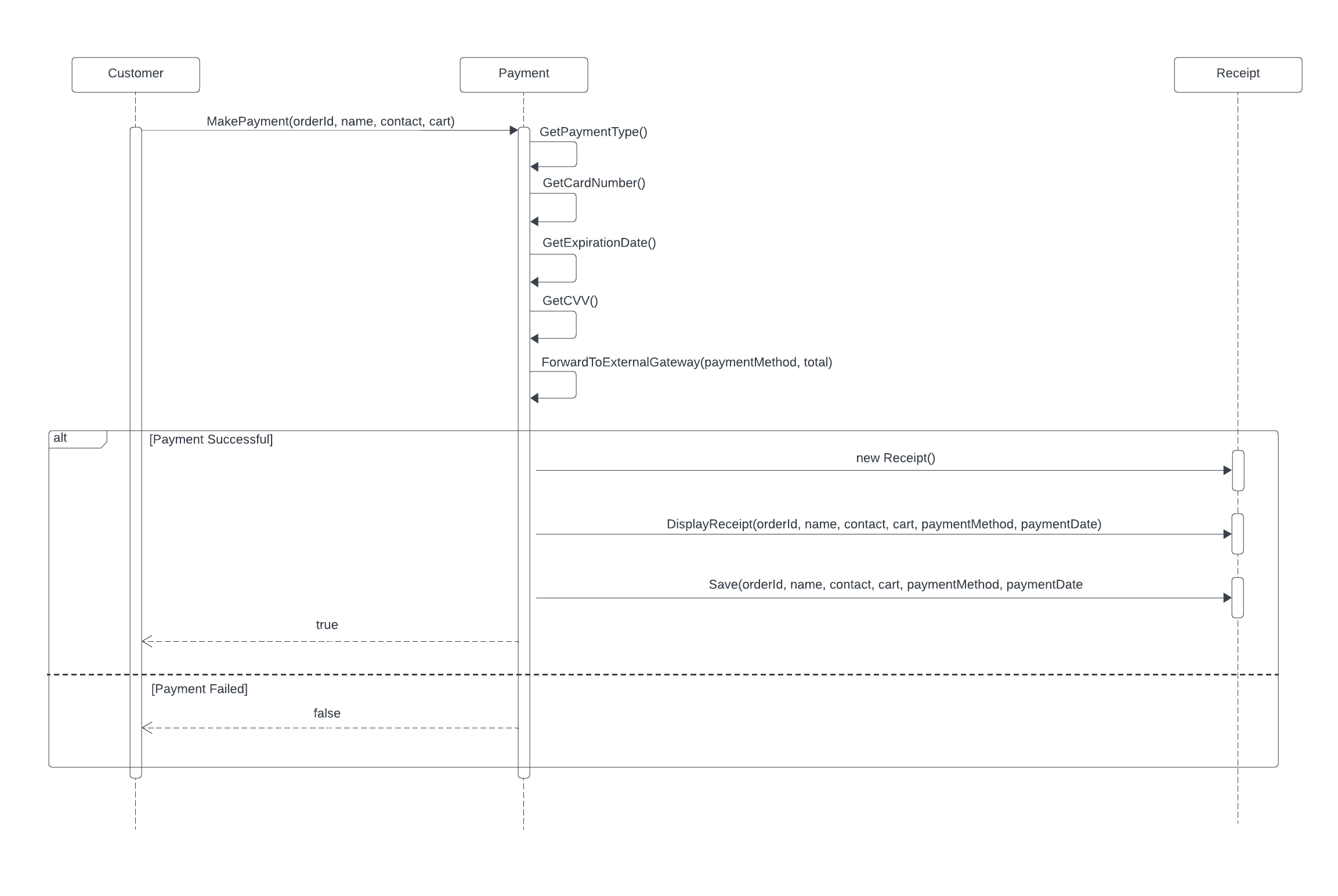


Figure 4 - Make payment UML sequence diagram

#### Handle delivery order

In this scenario, the sequence begins with the **Customer** interacting with **Order** by invoking the method **AskDeliveryOrDineIn()**. If the customer chooses delivery, the **Order** class executes the method **GetDeliveryAddress()** to obtain the delivery address. Regardless of the choice, the **Customer** interacts with the **Order** class to display the order details using the method **DisplayOrder()**. The **Customer** initiates the payment process by invoking the method **MakePayment(orderId, name, contactNumber, cart)** on the **Payment** class. If the payment is successful, the **Payment** class confirms it to the **Customer** and proceeds to save and process the order using the methods **Save()** and **ProcessOrder()** on the **Order** class, respectively. If the order is for delivery, a new **Delivery** instance is created by the **Order** class, and the order is forwarded to a third-party delivery service using the method **ForwardOrderToThirdParty(order)** on the **Delivery** class. The delivery service then updates the delivery status using the method **UpdateDeliveryStatus(order).** If the payment fails, **Payment** informs the **Customer** about the failure by returning Boolean **false**.

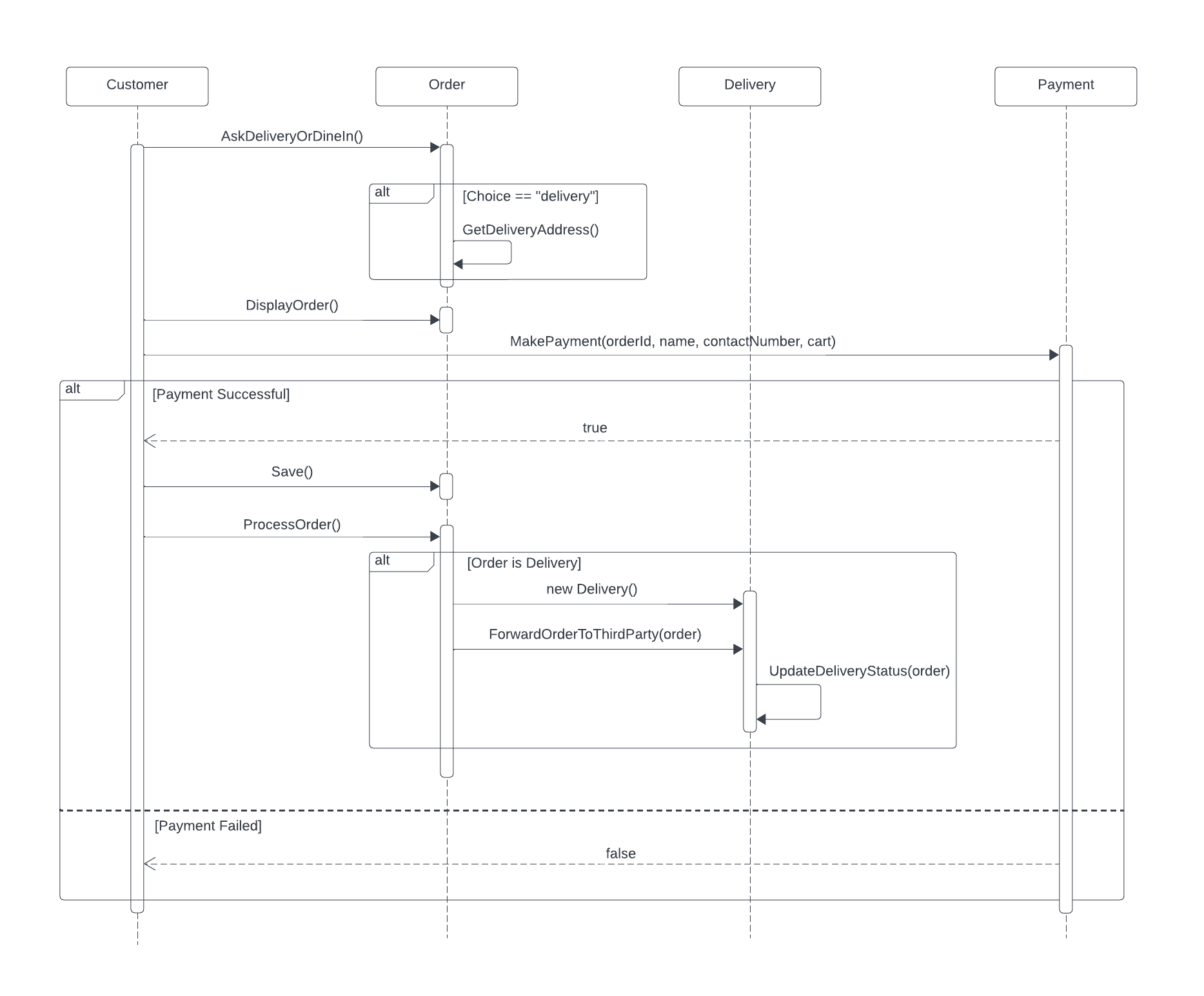


Figure 5 - Handle delivery order UML sequence diagram

# Design Quality

This section discusses the quality of the original design from Assignment 2.

## Good aspects

* The original design correctly identified most of the core classes required for the RIS, such as **Customer**, **Reservation**, **Table**, **Order**, **Menu**, and **MenuItem**. These classes represent the essential entities and concepts within the problem domain.
* The responsibilities of each class were defined reasonably well in the original design. For example, the Customer class was responsible for making reservations and placing orders, the Reservation class managed table assignments and availability, and the Order class handled customer orders and their details.
* The use of UML diagrams helped in visualising the classes and their relationships and CRC cards to brainstorm the responsibility and collaborators of each class.
* The overall structure of the design followed good object-oriented principles, such as encapsulation and separation of concerns. Although there are some flaws in the original design, each class aimed to encapsulate related data and behaviour.

## Missing from original design

* The original design did not include a dedicated class for generating order statistics reports, which is one of the core functional requirements of the RIS. This functionality was added in the refined design with the introduction of the **Statistics** class, which is responsible for generating reports based on order data, calculating statistics like total revenue and item counts, and saving the reports to files.
* There was no explicit handling of files or persistent storage for saving and loading reservations or orders in the original design. This means that any reservations made, or orders placed would only exist within the application's runtime memory and would be lost once the application terminates.

## Flawed aspects of original design

* The **Invoice** class was initially included, but it has been removed in the refined design as payments are now made immediately after placing each order, eliminating the need for a separate Invoice class.
* The placement of certain properties and responsibilities was not optimal. For example, the delivery address was initially part of the **Delivery** class but has been moved to the **Order** class in the refined design.
* The responsibilities related to making payments were initially assigned to the **Customer** class but have been moved to the dedicated **Payment** class in the refined design for better separation of concerns.

## Level of interpretation required

* The original design from Assignment 2 provided a reasonable starting point, but it required some interpretation and refinement to address certain gaps and flaws. The refined design in Assignment 3 demonstrates a more comprehensive understanding of the problem domain and incorporates improvements based on further analysis and design considerations.
* While the core structure of the design remained largely intact, the refinements in Assignment 3 addressed missing components, clarified responsibilities, and optimised the placement of properties and methods across classes. By comparing the two designs, it is clear that the original design from Assignment 2 served as a solid foundation, but the refinement process in Assignment 3 resulted in a more robust and well-rounded design and handled the requirements of the RIS better.

# Lessons Learned

## From the object design

* **UML sequence diagram:** During the initial design phase, we identified some flaws in our UML sequence diagram from Assignment 2. Specifically, we did not accurately reflect the responsibility for the **makeReservation()** method and other scenarios. For example, in Assignment 2, we incorrectly indicated that **makeReservation** was the responsibility of the **Reservation** class, which led to slight inconsistencies in our design between CRC cards and UML sequence diagrams. In CRC cards, **makeReservation()** is the responsibility of the **Customer** class. This should be clearly shown in the diagram, with the **makeReservation()** method being called from **Program** to **Customer**, indicating that **Program** is initiating the reservation process by calling the **makeReservation()** method of the **Customer** class. We have corrected this mistake in Assignment 3, ensuring the diagram accurately reflects the information from the CRC cards.
* **Handling ambiguous design:** The initial design is not as comprehensive as required for implementation. While our design effectively outlines the system's high-level structure, it remains abstract and lacks the details necessary from an implementation viewpoint. This high-level abstraction made it difficult for our team to directly translate the design into functional code, leading to multiple times of modifications and refinements during the development process. One significant lesson learned from this experience is the importance of creating a more detailed design that bridges the gap between conceptual models and practical implementation. If we got to do this differently, we would ensure that the design would include more detailed specifications for each component, comprehensive use case scenarios and both class and sequence diagrams.

## From implementation

During the implementation phase, we realised that our initial design lacked consideration for the database/storage system. As a result, various methods necessary for saving data to file storage were not adequately addressed in the CRC cards. If we were to redo this process, we would integrate the database into the object design phase. This would involve explicitly defining responsibilities related to data storage and retrieval in the CRC cards, thereby simplifying the implementation process and reducing the need for extensive modifications later on.

## From working in a team

* **Communication and Coordination:** As we worked in a team, we realised that everyone had very different perspectives and opinions, even though we were studying the same unit with the same tutor in the same class. To address this, we all need to establish common ground. Communication was key, especially before and during the writing of reports and the design/implementation phases. It is important to come up with the same solutions and plan before implementing them. When further issues come up during implementation or report writing, we will try to communicate either through Teams meetings or Discord group chats to resolve conflicts.
* **Balancing different schedules:** Everyone in the team had very different commitments and available times, making it challenging to find common times to discuss things. Therefore, we all had to put aside personal commitments and meet outside of regular hours. This allows everyone to contribute and collaborate more effectively. Additionally, we worked on the assignment at different times, some worked early in the morning, while others worked late at night, making it difficult to keep up with each other and reply to messages promptly. However, as the deadline approached, our communication improved, and we managed to coordinate more effectively to complete the assignment.

# Implementation

## Features implemented

Our implementation covers all functional requirements stated in the System Requirements Specifications (SRS) document:

1. Make reservation
2. Place order
3. Inform order information to kitchen
4. Handle payment
5. Generate receipt
6. Arrange delivery for takeaway orders
7. Provide an online menu
8. Provide ordered menu item statistics.

Note: To reduce workload and complexity, we did not implement the design patterns specified in Assignment 2.

## Platforms

#### Overview

Our implementation is a console-based application using **C#** programming language. The application presents the user with a menu containing four options:

1. Make reservation
2. Place order
3. Browse online menu
4. Generate order statistics report

Users can interact with the application by entering the number or command corresponding to their desired action as prompted by the menu.

**Note**: The “Place order” task also covers “Handle payment”, “Inform order information to kitchen”, “Arrange delivery for takeaway orders” and “Generate receipts” tasks.

#### User Interface

* A simple textual user interface is provided.
* Each input requirement is clearly labelled, and appropriate input validation and error message is included (e.g., ensuring the name field is not blank and using correct data types).

#### Data Storage

Instead of using a database, we use local file storage to manage reservation, order, and statistics report data. This approach simplifies the implementation and meets the requirement of using files for persistent data storage.

All data files are stored within the "**data**" folder. For instance:

* Reservations are stored in "**data/reservations.txt**".
* Orders are stored in "**data/orders.txt**".
* Order statistics reports are stored in "**data/order\_statistics\_report.txt**".

#### Coding Standards

We adhere to [C# Coding Standards and Naming Conventions](https://github.com/ktaranov/naming-convention/blob/master/C%23%20Coding%20Standards%20and%20Naming%20Conventions.md) to ensure our code is maintainable and professional. Specifically:

* Class Names, Constructor Names and Method Names: **PascalCase**
* Public Field Names: **PascalCase**
* Private Field Names: **\_camelCase**
* Method Arguments and Local Variables: **camelCase**

Additionally, we ensure that variable names are meaningful, avoid abbreviations, and declare all member variables at the top of a class, with static variables at the very top.

## Deployment instructions

**Prerequisite:**

Before you start, ensure that you have the following installed on your system:

* **.NET 6.0 SDK**: Download and install the .NET 6.0 SDK from the official [Microsoft .NET website](https://dotnet.microsoft.com/en-us/download/dotnet/6.0).

**Steps:**

1. Download the source code
2. Unzip the source code folder to a desired location on your system
3. Open the source code folder in any integrated development environment (IDE) such as Visual Studio Code
4. **Open the terminal in the IDE:** Within your IDE, locate the terminal window. In Visual Studio Code:
   1. Navigate to the "Terminal" menu and select "New Terminal."
   2. Alternatively, use the keyboard shortcut:
      * Windows/Linux: **Ctrl + `** (backtick)
      * Mac: **Cmd + `** (backtick)
5. Restore project dependencies:
   1. In the terminal, navigate to the project directory if not already there
   2. Run the following command to ensure all necessary project packages are downloaded:

**dotnet restore**

1. **Run the application:** In the terminal, run the following command to start the application:

**dotnet run**

**Note:** These instructions work on both Windows and macOS, provided the **.NET version 6.0** **SDK** and your chosen IDE (**Visual Studio Code**) are installed and configured properly on your system.

## Evidence

In this section, we will demonstrate our implementation works correctly by displaying different screenshots for each scenario. Additionally, a video file is attached to provide evidence of the implementation in action.

#### Scenario 1: Make reservation

When the user starts the application, they are greeted with a welcome message and presented with a main menu that includes five options: Make Reservation, Place Order, Browse Online Menu, Generate Order Statistics Report, and Exit the program. To initiate a reservation, the user selects option 1.

Upon selecting the option to make a reservation, the user is prompted to enter their name. The system performs validation to ensure that the input is not empty. If the user attempts to proceed without entering a name, an error message is displayed, instructing them that the name cannot be empty. The user must then enter a valid name to continue.

Next, the user is asked to enter their contact number. The contact number must be exactly 10 digits long. If the user provides a number that is not 10 digits, or if the input is empty, the system will display an error message indicating the specific issue. The user will need to re-enter a valid contact number to move forward.

After successfully entering their contact details, the user is asked to specify the seating capacity required for the reservation, which corresponds to the number of guests. The system again checks that the input is not empty. If the user fails to provide a number, an error message is shown, and they must enter the number of guests to proceed.

Following this, the user is prompted to enter the desired reservation date and time in the format dd/MM/yyyy HH:MM. Similar to previous steps, if the input is empty or incorrectly formatted, an error message is displayed, and the user must correct their input.

The system then displays a list of available tables that match the requested seating capacity. Each table is shown with its table number and seating capacity. The user is asked to choose a table by entering the corresponding table number. If the user selects an invalid table number, the system will prompt them to enter a valid table number.

Finally, once the user selects a valid table number, the reservation is confirmed. The system informs the user that the table has been successfully reserved for the specified number of guests and provides a reservation ID for future reference.



Figure 6 – Make reservations process

This completes the reservation process, and a record about the reservation will be added to **data/reservations.txt** file, including reservation ID, reservation date and time, reserved table number, and the number of guests.

A screenshot of a computer

Description automatically generated

Figure 7 – Reservation records

#### Scenario 2: Place order

To place an order, the user selects option 2.

**Dine-In Order**

Upon selecting the option to place an order, the user is prompted to enter their name and contact number. Similar to the previous scenario, there will be input validation for these, and the user will be prompted to re-enter a valid name/contact number if they didn’t meet the requirement.

After successfully entering their contact details, the user is shown the menu of available items along with their descriptions and prices. The user can enter commands such as 'add' to add an item to the cart, 'remove' to remove an item, 'view' to view the cart, 'menu' to view the menu again, or 'done' to finish the ordering process.

The user starts by adding items to their cart. For example, if they enter 'add' and then the number 2, 'Dumpling' is added to their cart, followed by similar actions for other items like 'Carbonara' and 'Pho'. If an invalid command is entered, the system prompts the user to enter a valid command.

A screenshot of a computer program

Description automatically generated

Figure 8 – Place order process

Once all desired items are added, the user can view their cart to see a list of items and the total cost. If the user decides to finish the order by entering 'done', the system will ask if they prefer delivery or dine-in. For a dine-in order, the user enters 'dine in'. The system then displays an order summary including the items, total cost, and order type.

A screenshot of a computer program

Description automatically generated

Figure 9 – User enters Dine-in option

The user is prompted to choose a payment method (credit card or debit card). After selecting the payment method, the user must enter their payment details, including the card number, expiration date, and CVV. If any of these details are missing or incorrect, appropriate error messages are displayed.

A screenshot of a computer

Description automatically generated

Figure 10 – Handle payment process

Once the payment details are validated and the payment is processed, the system confirms the payment and displays the receipt details, including the order ID, customer name, contact number, payment method, date, and items ordered. The receipt is saved as a text file (e.g., data/receipt\_001.txt).

The kitchen is informed of the new order, and the order process is completed with a confirmation message to the user.

A screenshot of a computer

Description automatically generated

Figure 11 – Receipt of the first order saved to file

**Delivery Order**

The process for placing a delivery order is similar to a dine-in order. But after finalising their cart and entering 'done', the user enters 'delivery' as the order type instead of ‘dine in’. The system then prompts the user to enter their delivery address. If the address is empty, an error message is displayed, and the user must provide a valid address to continue.

The system displays an order summary including the items, total cost, order type, and delivery address. The user is again prompted to choose a payment method and enter their payment details. Similar validation and error handling are performed to ensure correct payment information.

A screenshot of a computer program

Description automatically generated

Figure 12 – User enters Delivery option

Once the payment is successfully processed, the system confirms the payment and displays the receipt details, which are saved as a text file (e.g., data/receipt\_002.txt). The kitchen is informed of the new order, and the order is forwarded to a third-party delivery service for processing. The user receives updates about the order status, such as when the order is picked up by the delivery driver and when it is delivered to the specified address.

A screenshot of a computer

Description automatically generated

Figure 13 – Receipt of the second order save to file

For both dine-in and delivery orders, the details of each order are recorded in data/orders.txt, ensuring that all orders are logged for future reference.

A screenshot of a computer

Description automatically generated

Figure 14 – Order records

Each receipt is created as a separate file (e.g., data/receipt\_001.txt and data/receipt\_002.txt) containing detailed information about the order.

A screenshot of a computer screen

Description automatically generated

Figure 15 – Receipt of the second order record

A screenshot of a computer screen

Description automatically generated

Figure 16 – Receipt of the first order record

**Empty cart order:**

In this scenario, the user chooses to complete the order without adding any items to the cart by entering 'done' directly. The system detects that the cart is empty and responds with an error message stating, "No items were selected. Order cancelled."

A screenshot of a computer

Description automatically generated

Figure 17 – Error input when user enters “done” with no item in the cart

#### Scenario 3: Browse online menu

To browse the online menu, the user selects option 3.

Upon selecting the option to browse the online menu, the user is presented with a detailed list of available menu items. Each item includes the name, a brief description, and the price.

A screenshot of a computer program

Description automatically generated

Figure 18 – Browse online menu option

#### Scenario 4: Generate order statistics report

To generate an order statistics report, the user selects option 4. The system then processes the available order data and creates a detailed report. This report provides an overview of the restaurant's order statistics, including total orders, total revenue, and counts for each menu item.

A screenshot of a computer

Description automatically generated

Figure 19 – Generate order the statistics report

The report will be saved to **data/order\_statistics\_report.txt** file.

A screenshot of a computer

Description automatically generated

Figure 20 – Order statistics report record

If no orders have been created yet (i.e., there is no orders.txt file in the file storage), the system will notify the user that there are no orders to report on.

A screenshot of a computer

Description automatically generated

Figure 21 – Error messages of non-existence of order statistics report

# Appendix

Object Design (Assign 2)

Restaurant Information System

Jade Hoang, Henry Le, Thanh Nam Vu, Dang Khoa Le

SWE30003 – Software Architecture and Design

Table of Contents

[1. Executive Summary 35](#_Toc165738752)

[2. Introduction 36](#_Toc165738753)

[3. Assumptions 36](#_Toc165738754)

[4. Problem Analysis 37](#_Toc165738755)

[4.1 Functional requirements 37](#_Toc165738756)

[4.2 Quality attributes 37](#_Toc165738757)

[4.3 Simplification 37](#_Toc165738758)

[5. Candidate Classes 38](#_Toc165738759)

[5.1 Candidate Class List 38](#_Toc165738760)

[5.2 Discarded Classes 39](#_Toc165738761)

[5.3 Chosen Classes 40](#_Toc165738762)

[5.4 UML Diagram 41](#_Toc165738763)

[6. CRC Cards 42](#_Toc165738764)

[6.1 Customer 42](#_Toc165738765)

[6.2 Reservation 42](#_Toc165738766)

[6.3 Table 42](#_Toc165738767)

[6.4 Menu 43](#_Toc165738768)

[6.5 MenuItem 43](#_Toc165738769)

[6.6 Payment 43](#_Toc165738770)

[6.7 Order 44](#_Toc165738771)

[6.8 Invoice 44](#_Toc165738772)

[6.9 Receipt 44](#_Toc165738773)

[6.10 Delivery 45](#_Toc165738774)

[7. Design Solution 46](#_Toc165738775)

[7.1 Design Heuristic 46](#_Toc165738776)

[7.2 Design Patterns 45](#_Toc165738777)

[7.2.1 Behavioural Pattern - Observer pattern 46](#_Toc165738778)

[7.2.2 Structural Pattern - Composite pattern 46](#_Toc165738779)

[8. Bootstrap Process 47](#_Toc165738780)

[9. Verification 48](#_Toc165738781)

[9.1 Make a reservation 47](#_Toc165738786)

[9.2 Place an order 49](#_Toc165738787)

[9.3 Make payment 50](#_Toc165738788)

[9.4 Handle delivery order 51](#_Toc165738789)

[10. Appendix 52](#_Toc165738790)

**1. Executive Summary**

This document outlines the initial object-oriented design for the Restaurant Information System (RIS) of the Relaxing Koala café/restaurant. The RIS aims to modernise and streamline the day-to-day operations of the restaurant, including reservations, order management, invoicing, online menu access, and basic statistics generation.

Through a Responsibility Driven Design approach, this document includes a list of proposed classes, along with a graphical representation of their relationships in the form of a Unified Modelling Language (UML) class diagram. Additionally, it provides CRC (Class-Responsibility-Collaborator) cards for each chosen class, describing their responsibilities and collaborators.

The design incorporates the use of relevant design patterns and heuristics, which are documented and justified based on their applicability to the RIS domain. Moreover, the document illustrates the bootstrap (or initialisation) process of the system, highlighting the sequence in which classes are instantiated.

As part of the verification process, the document outlines four typical interaction scenarios. These scenarios aim to ensure the alignment of the design solution with the stated requirements.

Overall, this document serves as a comprehensive initial design for the RIS, providing a solid foundation for further development and refinement as the project progresses.

1. **Introduction**

Relaxing Koala, a growing café/restaurant, requires a robust system to manage its expanding capacity. This report introduces the initial object-oriented design for a Restaurant Information System (RIS) using the Responsibility Driven Design approach. The RIS aims to streamline operations such as reservations, orders, invoicing, and menu management.

This report details the following components of the design:

* **CRC cards**: High-level descriptions of classes and their respective responsibilities and collaborators within the system.
* **Class relationships**: Representation of class relationships through a UML Class Diagram.
* **UML sequence diagrams**: Illustration of four typical scenarios, showing how the design solution handles various system requirements as part of the verification process.
* **Quality of design solution:** Highlight design heuristics and design patterns used in the design.
* **Bootstrap process**: Highlight the sequence of instances of classes created.

1. **Assumptions**

* Each Order class instance will have a unique orderID assigned to facilitate easy reference and tracking.
* User roles within the restaurant are generalised into one category, "staff," without differentiation. All staff members will have access to the same set of functionalities, eliminating the need to manage various user roles.
* Each delivery will be associated with only one order.
* Staff information is generally not stored in the RIS, although staff will interact with the system.
* When a large group reservation spans multiple tables, the system will simplify the user experience by assigning a single table number or identifier to the reservation.
* A dedicated class will not handle statistical data about menu items. Instead, the system will calculate these statistics dynamically by analysing existing data stored within relevant classes like Order and Invoice. This approach promotes code reusability and avoids creating classes solely for calculations.

1. **Problem Analysis**

The Relaxing Koala restaurant is facing challenges due to its recent expansion. Their current manual processes for handling reservations, orders, communication with the kitchen, invoicing, and basic menu analysis are no longer efficient or scalable to support a larger capacity.

**4.1** **Functional requirements**

The key functional requirements of the RIS identified in the Software Requirements Specifications (SRS) document are:

* Make reservations
* Place order
* Inform order information to kitchen
* Handle payment
* Generate receipt/invoice
* Provide the ordered menu item’s statistic
* Provide an online menu
* Arrange delivery for takeaway orders

**4.2 Quality attributes**

Key quality attributes (non-functional requirements) for the RIS are also identified, including usability, performance, reliability, security, and scalability. These attributes ensure that the system is user-friendly, efficient, robust, secure, and capable of accommodating future growth and expansion.

Simplification

The following simplifications were made during the analysis phase of the RIS:

* Third-party integrations and external services are not explicitly represented as classes within this initial design solution.
* The RIS design simplifies user roles, consolidating all staff into one category without differentiation between roles such as waitstaff, kitchen staff, or managers.
* The design simplifies menu management by consolidating both takeaway and dine-in menus into a single class structure, eliminating the need for complex inheritance or separate divisions within the menu classes.
* The design simplifies large group reservations by assigning a single table number or identifier, rather than managing reservations across multiple tables.
* Split bills or partial payments are not allowed, simplifying payment processing to one payment per order.
* The process of sending order information to the kitchen will be simplified to display order details for kitchen staff visibility.

1. **Candidate Classes**

**5.1 Candidate Class List**

The following list presents all possible classes for the RIS:

* Restaurant
* Capacity
* Customer/People/Guest
* Kitchen
* Accounting
* Owner
* Staff
* InformationSystem
* Reservation
* Order
* Invoice
* Receipt
* Payment
* Menu
* MenuItem
* Statistic
* Table
* Offerings
* Delivery

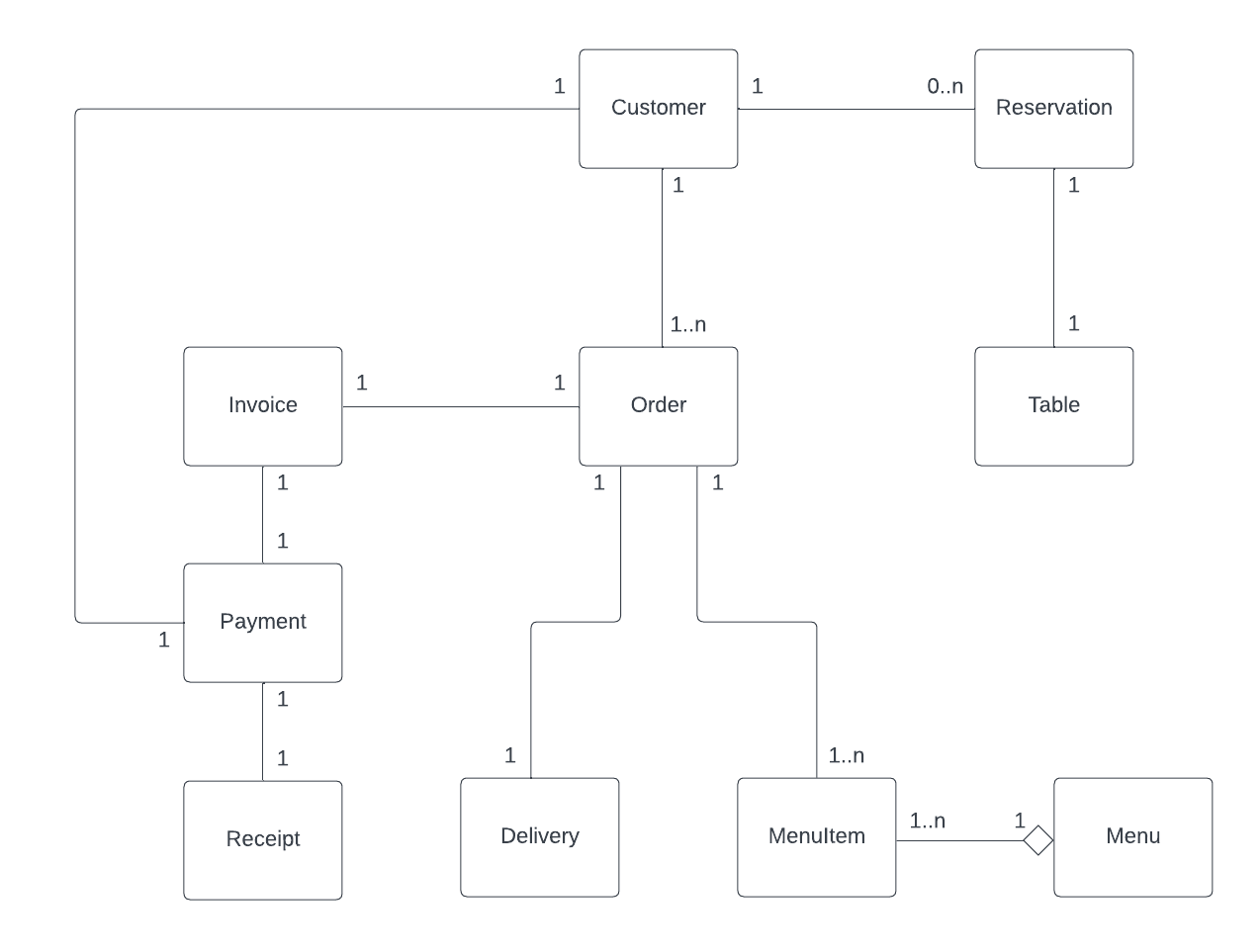
**5.2 Discarded Classes**

* **Statistic**: Menu item statistics can be calculated dynamically by analysing existing data in other classes such as Order and Invoice. This keeps the system focused on its core responsibilities and prevents the proliferation of classes that represent operations rather than fundamental abstractions within the domain model.
* **Offering:** Overlaps with the Menu and MenuItem classes, which already represent the restaurant's available food and beverage options.
* **Capacity:** This isnot a standalone entity with distinct attributes or responsibilities. Capacity information can be managed as an attribute within Table class.
* **Accounting:** This can be considered a responsibility rather than a class. Its functionalities can be incorporated into other classes like Receipt or Payment.
* **Owner:** Owner's information is not stored in the RIS and has no responsibilities/functions that need to be performed.
* **Staff:** RIS does not store staff information or require specific staff-related functions. Existing human resources systems can handle staff information and management.
* **InformationSystem:** This represents the RIS itself and is not necessary as a separate class within the system's design.
* **Restaurant:** This is too broad and encompasses the whole system, making it difficult to store information related to it.
* **Kitchen:** The Kitchen class might be unnecessary if the restaurant's kitchen operations are straightforward and do not require detailed management or tracking. Since the case study does not specify any complex kitchen management requirements, the responsibility of informing kitchen staff about orders can be handled by the Order class itself.

**5.3 Chosen Classes**

* **Customer:** This class is needed to encapsulate essential customer-specific data, including contact details and any dietary preferences/restrictions, and handles responsibilities such as making reservations and placing orders.
* **Reservation:** Represents bookings made by customers, storing information such as reservation details and table assignments while handling responsibilities such as managing table availability.
* **Table:** Thisclass is essential for encapsulating key data related to seating arrangements such as table availability, table number and seating capacity.
* **Order:** Represents requests made by customers for food and drinks, storing information such as items ordered and quantity, and managing order-related functionalities.
* **Invoice:** Invoices need to store information such as the total amount and items ordered while responsible for generating, storing, and calculating invoice details.
* **Receipt:** This class is needed to store transaction details and payment status while managing responsibilities such as formatting and providing proof of payment.
* **Payment:** Even with an external payment gateway, the RIS needs a Payment class to interact with it. This class would handle payment transactions, while storing information about transactions, including payment method, amount, and reference number.
* **Menu:** This class is needed to manage the restaurant's menu, including the ability to add, remove, and update menu items, as well as display the menu for customers.
* **MenuItem:** This class represents individual menu items on the menu, including their names, descriptions, and prices.
* **Delivery:** This class represents the delivery service for takeaway orders and needs to store information such as delivery status, manages the delivery process for takeaway orders, including knowing delivery addresses, coordinating with third-party delivery services, and tracking delivery statuses.

**5.4 UML Diagram**



*Figure 3 - UML Diagram*

1. **CRC Cards**

**6.1** **Customer**

Represents a customer of the restaurant, including their personal information and any dietary preferences/restrictions.

|  |  |
| --- | --- |
| Class Name: Customer | |
| Responsibilities | Collaborators |
| Know customer’s name, contact details, dietary preferences | N/A |
| Make reservation | Reservation |
| Place order | Order, Menu, MenuItem |
| Make payments | Payment, Invoice, Receipt |

*Table 11 - Customer CRC*

**6.2** **Reservation**

This class manages reservations made by customers within the RIS, including date, time, party size, and special requests. It ensures proper allocation of tables based on customer requests.

|  |  |
| --- | --- |
| Class Name: Reservation | |
| Responsibilities | Collaborators |
| Know date, time, number of guests | N/A |
| Check table availability for requested date and time | Table |
| Assign table to reservation | Table |

*Table 12 - Reservation CRC*

**6.3** **Table**

This class manages table arrangements and information such as seating capacity, table number and table availability status.

|  |  |
| --- | --- |
| Class Name: Table | |
| Responsibilities | Collaborators |
| Know table seating capacity (i.e., number of seats) | N/A |
| Know table number (i.e., tableID) | N/A |
| Update table availability status (occupied or available) | N/A |

*Table 13 - Table CRC*

**6.4** **Menu**

Manages the restaurant's menu, including adding, updating, and removing menu items. The Menu class manages a collection of MenuItem objects.

|  |  |
| --- | --- |
| Class Name: Menu | |
| Responsibilities | Collaborators |
| Add menu item | MenuItem |
| Remove menu item | MenuItem |
| Update menu item | MenuItem |
| Display menu | MenuItem |

*Table 14 - Menu CRC*

**6.5** **MenuItem**

Represents an individual menu item within the menu, including its name, description, price, and category.

|  |  |
| --- | --- |
| Class Name: MenuItem | |
| Responsibilities | Collaborators |
| Know item’s ID, name, description, category, and price | N/A |

*Table 15 - MenuItem CRC*

**6.6** **Payment**

Payment class manages the financial transactions initiated by customers. It interacts with an external payment gateway and generates receipts for successful transactions.

|  |  |
| --- | --- |
| Class Name: Payment | |
| Responsibilities | Collaborators |
| Know payment method | N/A |
| Know invoice details | Invoice |
| Forward to external payment gateway | N/A |
| Generate receipt | Receipt |

*Table 16 - Payment CRC*

**6.7** **Order**

The Order class represents a customer's order within the RIS. It captures the details of the ordered items, their quantities, and facilitates communication with the kitchen.

|  |  |
| --- | --- |
| Class Name: Order | |
| Responsibilities | Collaborators |
| Know orderID | N/A |
| Know special requests (e.g. allergy) | N/A |
| Know menu items | Menu, MenuItem |
| Associate a specific menu item with its quantity to the order | MenuItem |
| Send order information to kitchen | N/A |
| Track order status | N/A |
| Submit order details for delivery | Delivery |

*Table 17 - Order CRC*

**6.8 Invoice**

Represents an invoice generated for a customer order, including order details, totals, discounts, and taxes. It serves as a record of the order details and the financial obligation.

|  |  |
| --- | --- |
| Class Name: Invoice | |
| Responsibilities | Collaborators |
| Summarise order details | Customer, Order, MenuItem |
| Calculate the total amount | Order |
| Generate invoice | Customer, Order |

*Table 18 - Invoice CRC*

**6.9 Receipt**

The Receipt class serves as a confirmation document for a successful payment within the RIS. It provides details about the completed transaction.

|  |  |
| --- | --- |
| Class Name: Receipt | |
| Responsibilities | Collaborators |
| Know payment type, payment status | Payment |
| Know transaction details | Payment |

*Table 19 - Receipt CRC*

**6.10 Delivery**

Represents the delivery service for takeaway orders and manages the delivery process. It acts as an intermediary between the RIS and a third-party delivery service

|  |  |
| --- | --- |
| Class Name: Delivery | |
| Responsibilities | Collaborators |
| Know delivery address, delivery instructions | Order |
| Forward orders to third-party delivery services (to process delivery order) | N/A |
| Update delivery status | N/A |
| Track delivery status | N/A |

*Table 20 - Delivery CRC*

**7. Design Solution**

**7.1 Design Heuristic**

* A class should capture one and only one key abstraction.
* Keep related data and behaviour in one place.
* Do not create god classes/objects in your system.
* Minimise the number of classes with which another class collaborates.
* Minimise the amount of collaboration between a class and its collaborators (i.e., the number of messages sent).
* Choose the containment relationship (aggregation) over association when given a choice.
* Do not turn an operation into a class.

**7.2 Design Patterns**

**7.2.1 Behavioural Pattern - Observer pattern**

Observer pattern is a behavioural design pattern that facilitates communication between objects. When a state change occurs in an observable class, it notifies its registered observer classes. This allows the observers to react accordingly, promoting loose coupling between classes. Within our design for the RIS, the Observer pattern is applicable in both order and reservation management.

**Reservation management:**

* The **Reservation** class could be observable. Upon making a reservation, the **Table** class, acting as an observer, would be notified. Consequently, the **Table** class could update its availability status to reflect occupied tables.

**Order management:**

* The **Order** class acts as an observable. Observer classes like **Delivery** and **Customer** could be notified whenever an order is placed, updated, or cancelled.
* **Delivery**: Upon notification, the **Delivery** class schedules delivery for takeaway orders.
* **Customer**: The **Customer** class receives updates on order status, such as "Preparing," or "Ready for Pickup".

**7.2.2 Structural Pattern - Composite pattern**

Composite pattern allows treating a group of objects as a single object. A composite object can contain other composite objects or leaf objects (i.e., objects without children). This pattern is useful for representing hierarchical structures.

In the context of the RIS, the **Menu** and **MenuItem** classes could be structured using the Composite pattern, where a **MenuItem** is a leaf object, and a **Menu** is a composite object containing other **Menu** objects (representing sub-categories) or **MenuItem** objects. This structure allows the building of a hierarchical menu with categories and subcategories.

**8.** **Bootstrap Process**

* The system starts by creating an instance of the **Menu** class.
* **Menu** creates instances of **MenuItem** class.
* Instances of the **Table** class are created to manage table arrangements.
* For each customer interaction, an instance of the **Customer** class is created.
* When a customer makes a reservation, **Customer** creates an instance of **Reservation** class.
* When the customer places an order, **Customer** creates an instance of the **Order** class.
* **Order** creates an instance of **Invoice** class.
* **Customer** creates an instance of **Payment** class.
* **Payment** creates an instance of **Receipt** class for the transaction.
* If the order requires delivery, **Order** creates an instance of **Delivery** class.

**9****. Verification**

**9.1** **Make a reservation**

In this scenario, the **Customer** initiates the reservation process by making a reservation with the **Reservation** class, providing details such as the *date*, *time*, and *party size*. The **Reservation** class then checks table availability by communicating with the **Table** class, which responds with a list of available tables. The **Reservation** class then selects a suitable table and assigns it to itself. Subsequently, the **Table** class updates the availability status of the assigned table. Finally, the **Reservation** class notifies the **Customer** of the successful reservation by providing the *tableID*.

A diagram of a customer service

Description automatically generated

*Figure 4 - Make a reservation UML sequence diagram*

**9.2 Place an order**

Initially, the **Customer** initiates the order process by creating an order through the *placeOrder* method, which collaborates with the **Order** class. Subsequently, the **Order** class communicates with the **Menu** class to retrieve a list of available menu items using the *getMenu* method. The **Menu** class then interacts with the **MenuItem** class to retrieve a list of all menu items using the *getMenuItem* method. Upon receiving the menu items list, both the **Menu** and **Order** classes return it to the **Customer**. The **Customer** then adds selected items to the order by invoking the *addItemToOrder* method, specifying the *menuItemID* and *quantity*. Finally, the **Order** class confirms the order by returning the *orderID* to the **Customer**.

A diagram of a menu

Description automatically generated

*Figure 3 - Place an order UML sequence diagram*

**9.3 Make payment**

To initiate a payment, the **Customer** triggers the process by invoking the *makePayment* method, proving *paymentMethod* and *orderID* to **Payment** class. The **Payment** class then communicates with the **Invoice** class to retrieve the invoice details associated with the order using the *getInvoiceDetails* method, with the *orderID* as a parameter. The **Invoice** class interacts with the **Order** class to gather the necessary order details, returning them to the **Invoice** class.

Using the obtained order details, the **Invoice** class calculates the total amount to be paid and generates an invoice accordingly. This invoice, along with its details, is then returned to the **Payment** class.

Subsequently, the **Payment** class forwards the invoice details to an external payment gateway for processing through the *forwardToExternalGateway* method. Once the payment is successfully processed, the **Payment** class generates a receipt for the transaction. This receipt is provided to the **Customer** by interacting with the **Receipt** class, thus completing the payment process.

A diagram of a payment

Description automatically generated

*Figure 4 - Make payment UML sequence diagram*

**9.4 Handle delivery order**

In this scenario, the **Customer** initiates the process by calling the *placeOrder* method on the **Order** class, specifying that it is a delivery order by setting the *isDelivery* flag to true. Once **Order** class get the order details (like the previous “Place order” scenario), it communicates with the **Delivery** class to submit order details using the *submitOrderDetails* method. The **Delivery** class then forwards the order details to a third-party delivery service for processing using the *forwardOrderToThirdParty* method. The **Delivery** class also updates the delivery status internally using the *updateDeliveryStatus* method. After processing, the **Delivery** class updates the order status and communicates it back to the **Order** class. Finally, the **Order** class relays the updated order status to the **Customer**.

A diagram of a process

Description automatically generated

*Figure 5 - Handle delivery order UML sequence diagram*

**10.** **Appendix**

SOFTWARE REQUIREMENTS SPECIFICATION - Group 3

Restaurant Information System

Jade Hoang, Henry Le, Thanh Nam Vu, Dang Khoa Le

SWE3003 – Software Architecture and Design

**Table of Contents**

[1. Introduction](#_Toc163163042) 23

[2. Project Overview 55](#_Toc163163043)

[2.1 Domain Vocabulary: 55](#_Toc163163044)

[2.2 Goals: 56](#_Toc163163045)

[2.3 Assumptions: 56](#_Toc163163046)

[2.4 Scope: 56](#_Toc163163047)

[2.5 Constraints: 56](#_Toc163163048)

[3. Problem Domain 57](#_Toc163163049)

[3.1 Pain Points 57](#_Toc163163051)

[3.2 Actors 57](#_Toc163163052)

[3.3 Tasks 57](#_Toc163163053)

[4. Domain Model 58](#_Toc163163054)

[4.1. Domain Entities 58](#_Toc163163055)

[4.2. Domain Model 59](#_Toc163163056)

[5. User Tasks 60](#_Toc163163057)

[5.1. Make Reservation 60](#_Toc163163058)

[5.2. Place Order 61](#_Toc163163059)

[5.3. Inform Order Information to Kitchen 63](#_Toc163163060)

[5.4. Handle Payment 64](#_Toc163163061)

[5.5. Generate Receipt/Invoice 65](#_Toc163163062)

[5.6. Provide Ordered Menu Item Statistics 66](#_Toc163163063)

[5.7. Provide Online Menu 67](#_Toc163163064)

[5.8. Arrange Delivery for Takeaway Orders 68](#_Toc163163065)

[6. Workflows 69](#_Toc163163066)

[7. Quality Attributes 70](#_Toc163163067)

[7.1 Usability 70](#_Toc163163070)

[7.2 Performance 70](#_Toc163163071)

[7.3 Reliability 70](#_Toc163163072)

[7.4 Security 70](#_Toc163163073)

[7.5 Scalability 71](#_Toc163163074)

[8. Other Requirements 72](#_Toc163163075)

[8.1 Product level requirements 72](#_Toc163163077)

[8.2 Design level requirements 72](#_Toc163163078)

[9. Validation 73](#_Toc163163079)

[10. Verifiability 74](#_Toc163163080)

**1. Introduction**

This report is a Software Requirements Specifications (SRS) for a Restaurant Information System (RIS) for the Relaxing Koala restaurant located on Glenferrie Road. The system aims to automate various aspects of the restaurant's daily operations, including reservation management, order processing, order transmission to the kitchen, invoicing, and basic statistical analysis of menu item popularity.

This document includes functional requirements and quality attributes, domain-level requirements, a domain model diagram, and a workflow chart. The functional requirements will be broken down into user tasks and follow the Task and Support approach. Additionally, it covers other requirements such as product and design-level requirements, along with evidence of validation and verifiability.

**2. Project** **Overview**

The owners of Relaxing Koala have identified a need to automate some of their operations due to recent expansion. Their current manual processes for order management, kitchen communication, and accounting are no longer viable. To address this, they aim to implement a RIS, developed by Swinsoft Consulting, capable of handling reservations, orders, invoices, payments, and providing menu statistics. The overarching goal is to enhance efficiency, customer service, and scalability.

**2.1 Domain Vocabulary:**

* **Customer**: Individuals who visit Relaxing Koala to dine and make purchases.
* **Delivery**: Service of transporting orders to customers' locations.
* **Invoicing**: Generating bills for the orders placed by customers.
* **Menu statistics**: Gathering data on customer orders and preferences to analyse the popularity of menu items.
* **Online menu**: Digitally accessible version of the restaurant's menu that lists food and beverage options available at the restaurant.
* **Take-away menu**: Menu displayed for customers ordering food to be taken off-site.
* **Order transmission to the kitchen**: System for transmitting orders from the front-end to the kitchen staff.
* **Payments**: Processing transactions for the orders and services used by customers.
* **RIS**: Restaurant Information System, the proposed system aimed at assisting in the daily operations of Relaxing Koala.
* **Website**: Front-end service promoting the restaurant, accessible to the customers.

**2.2 Goals:**

* Streamline restaurant operations to improve efficiency and reduce manual efforts.
* Enhance customer experience, service quality, and accessibility through efficient reservation, ordering, and payment systems.
* Lay the groundwork for future scalability and increased capacity of the business.

**2.3 Assumptions:**

* The system integrates with an external payment gateway for managing customer payments online.
* Delivery services are outsourced to third-party providers. Therefore, RIS is responsible for arranging delivery but not managing the delivery drivers or logistics.
* If dine-in, customers pay for their meals after they have finished dining, allowing them to place multiple orders online during their dine-in experience and settle the bill once at the end of their visit.
* The system generates digital invoices or receipts and sends them to customers via their provided email address or phone number for convenience and accessibility.

**2.4 Scope:**

The project focuses on developing an information system for Relaxing Koala café/restaurant currently undergoing expansion. The information system should enable staff to handle reservations, take customer orders, transmit orders to the kitchen, generate invoices and receipts, and manage payments. Additionally, basic statistics about ordered menu items will be provided. The system will also make menus available online, allowing customers to view offerings, place orders for takeaways, and potentially arrange delivery.

**2.5 Constraints:**

The new system needs to integrate with or replace the existing low-tech, manual processes currently in place at the Relaxing Koala, which may pose challenges in terms of data migration, staff training, and process adaptation.

**3.** **Problem Domain**

**3.1 Pain Points**

* Manual order taking and processing.
* Low-tech accounting procedures.
* Can only support up to 50 customers.
* Lack of an online menu for informing potential customers about offerings.

**3.2 Actors**

* **Customer**: End-user uses the RIS to browse menus, place orders, make reservations, and make payments
* **Waiter**: Takes orders from customers (if not ordered directly through the system) and inputs them into the RIS. They may also access table information and reservations.
* **Cashier**: Handles payments if the RIS does not manage them directly. They might interact with the system to record cash or other offline payments.
* **Manager**: Use the RIS to access ordered menu item statistics.
* **Kitchen staff**: Receive order information and prepare the food accordingly.
* **External payment gateway**: Interact with the RIS to process customer transactions securely.
* **Third-party delivery service**: Receive order details from the RIS and handle deliveries to customer addresses.

**3.3 Tasks**

* Make reservations
* Place order
* Inform order information to kitchen
* Handle payment
* Generate receipt/invoice
* Provide the ordered menu item’s statistic
* Provide an online menu
* Arrange delivery for takeaway orders

**4.** **Domain Model**

**4.1 Domain Entities**

* **Customer**: Individuals who visit the restaurant to dine in, order takeaway, or request delivery services.
* **Order**: A record of selected menu items for consumption or delivery.
* **Menu**: Digital/physical listing of food and beverage offerings.
* **Reservation**: Booking for a specific table at a particular date/time.
* **Payment**: Financial transaction associated with an order.
* **Receipt**: Document summarising order details and payment upon purchase.
* **Invoice**: Detailed bill presented to the customer, often for takeaway orders.
* **Staff**: Restaurant employees who interact with the system.
* **Statistic**: Aggregated data from orders and menus for insights.
* **Table**: Physical dining area with seating capacity and location.
* **Kitchen**: Prepares food based on order information.
* **Delivery**: External service that delivers orders to customers.

**4.2 Domain Model**

A diagram of a company

Description automatically generated

*Figure 5 - RIS domain model*

**5.** **User Tasks**

**5.1 Make Reservation**

|  |  |
| --- | --- |
| Task: | Make Reservation |
| Purpose: | Allow customers to reserve a table online for a specific date and time. The system records the reservation details, marking the table as unavailable for that timeframe. |
| Trigger/Precondition: | Customers want to dine in without waiting for their seats. |
| Frequency: | Average 15 times/day on weekdays and 45 times on weekends. |
| Critical: | Influx of reservation requests during peak hours. |
| **Sub-tasks:** | **Example solution:** |
| 1. Select preference time | System shows a list of timeslots to choose from. |
| 1. Enter the number of customers | System displays a numeric input field labelled "Number of Customers" on the reservation interface. |
| 1. Check availability | Determine the availability of tables based on the requested date, time, and party size. |
| 1. Select table | System shows the table location on a floor plan, allows the customer to choose a specific table based on their selection. |
| 1. Record customer information | System shows a list of input fields for customers to enter their details. |
| 1. Confirm reservation | System shows the reservation has been successfully made message. |
| **Variants** |  |
| 3a. No available tables for the requested date and time. | The system informs the customer that no tables are available at the desired date and time. It suggests alternative options (e.g., waiting list, recommending closest time slots). |
| 6a. Customer wants to modify their reservation | System offers alternative dates and times based on availability. |
| 6b. Customer wants to cancel existing reservation | System prompts for reason (e.g., change of plans, emergency) and updates reservation status accordingly. |

*Table 21 - Make Reservation User Task*

**5.2 Place Order**

|  |  |  |
| --- | --- | --- |
| Task: | Place Order | |
| Purpose: | Allow customers to browse the restaurant menu, select items, and submit their order electronically. | |
| Trigger/Precondition: | A customer visits the restaurant's physical location or accesses the restaurant's website and decides to order food for dine-in, delivery, or pickup. | |
| Frequency: | Approximately 200 orders/day | |
| Critical: | Large orders during high-volume periods. | |
| **Sub-tasks:** | | **Example solution:** |
| 1. Access website | | If dine-in, place QR codes at prominent locations on each table and direct customer to restaurant’s order website upon scanning the QR code with a smartphone camera. |
| 1. Browse menu   **Problem:** Difficulty visualising dishes. | | * System provides a digital menu accessible on the website. * Menu includes high-quality images and descriptions of dishes. |
| 1. Select item   **Problem**: Customer accidentally adds an incorrect item to their cart | | * System offers a user-friendly interface for adding and removing items from the cart. * Provides clear visual confirmation of selected items before checkout. * System includes a “Remove” button next to each item in the cart and displays a confirmation prompt before finalising the order. |
| 1. Customise order   **Problem:** Customer has dietary restrictions (allergy, religious) and needs to customise multiple menu items. | | System allows customisation options (e.g., remove sauce, extra vegetables, allergy) and a “Special Requests:” input field for each item. |
| 1. Choose order type (Delivery/Pickup/Dine-in) | | * System defines clear options to choose between "Dine-in" or "Delivery/Pickup" at the end of the ordering process. * If “Dine-in” is selected: System proceeds to the “Review and Confirm order” page. * If “Delivery” is selected: System provides an address selection process for delivery orders. * If “Pickup” is selected: Provide clear instructions and options for pickup orders (e.g., estimated waiting time). |
| 1. Review and confirm order | | * System displays a comprehensive order summary page for review (items, prices, and type with details). * Allows customers to edit or cancel items before finalising the order. |
| 1. Submit order | | System transmits the order to the kitchen and provides an immediate confirmation with an estimated preparation or delivery time. |
| **Variants** | |  |
|  | |  |

*Table 22 - Place Order User Task*

**5.3 Inform Order Information to Kitchen**

|  |  |  |
| --- | --- | --- |
| Task: | Inform order information to kitchen | |
| Purpose: | Transmit the order to the kitchen staff for preparation | |
| Trigger/Precondition: | Customer successfully places an order online through the restaurant’s website. | |
| Frequency: | Approximately 200 orders/day | |
| Critical: | Influx of orders during peak hours | |
| **Sub-tasks:** | | **Example solution:** |
| 1. Receive order details | | System captures order details, including menu items, quantities, special requests, and table numbers. |
| 1. Manage order queue | | System organises orders in a queue based on order time or preparation time. |
| 1. Display order on the screen | | System displays incoming orders on the kitchen display screens |
| **Variants** | |  |
| 1a. Special requests or instructions in the customer’s order | | System clearly displays any special instructions included in the customer's online order. |

*Table 23 - Inform Order to Kitchen User Task*

**5.4 Handle Payment**

|  |  |  |
| --- | --- | --- |
| Task: | Handle Payment | |
| Purpose: | Process financial transactions for orders placed by customers. | |
| Trigger/Precondition: | Customer has finished their meal or confirmed their delivery/pickup order. | |
| Frequency: | Approximately 200 times/day. | |
| Critical: |  | |
| **Sub-tasks:** | | **Example solution:** |
| 1. Identify the table’s bill (if dine-in) or identify customer’s orders | | System allows staff to search for a customer’s bill by table number and displays a clear summary of all online orders placed for that table during the visit. |
| 1. Review order details | | System provides a detailed breakdown of each online order, including item name and quantity, price per item, etc. |
| 1. Calculate total amount | | System computes the total amount owed by the customer, including all orders and applicable taxes or fees. |
| 1. Choose payment method | | System shows available payment types such as: debit cards, credit card (Visa, Mastercard, etc.), and bank transfer. |
| 1. Process payment | | System interacts with the external payment gateway to securely process customer transactions |
| **Variants** | |  |
| 3a. Customer requests to split the bill | | System allows splitting the bill by item or equally amongst diners at the table. It recalculates the total amount for each diner accordingly. |

*Table 24 - Handle Payment User Task*

**5.5 Generate Receipt/Invoice**

|  |  |  |
| --- | --- | --- |
| Task: | Generate receipt/invoice | |
| Purpose: | Provide customers with transaction proof. | |
| Trigger/Precondition: | Customers made payment | |
| Frequency: | Average 200 times/day. | |
| Critical: |  | |
| **Sub-tasks:** | | **Example solution:** |
| 1. Summarise order details | | System automatically retrieves all relevant order information of the customer during their visit. |
| 1. Generate invoice/receipt | | System generates a clear and well-formatted electronic invoice/receipt |
| 1. Deliver invoice/receipt   **Problem**: Customer might not receive the invoice/receipt due to email delivery issues or incorrect phone number entry. | | * System allows customers to choose their preferred delivery method (email or SMS with a link to the invoice). * System confirms customer-provided email addresses or phone numbers before sending the invoice/receipt. * Implements retry mechanisms for failed email deliveries. |
| **Variants** | |  |
| 3a. Customer requests physical receipt. | | System provides an option for customers to request a printed receipt and print the physical copy of the invoice/receipt. |

*Table 25 - Generate Receipt/Invoice User Task*

**5.6 Provide Ordered Menu Item Statistics**

|  |  |  |
| --- | --- | --- |
| Task: | Provide ordered menu item statistics | |
| Purpose: | Provide a clearer understanding of the types of menu items customers are ordering | |
| Trigger/Precondition: | Customer completes an order, and the transaction is finalised | |
| Frequency: | Daily, weekly, monthly or on-request | |
| Critical: |  | |
| **Sub-tasks:** | | **Example solution:** |
| 1. Collect order data | | System automatically collects data on all menu items included in fulfilled orders |
| 1. Aggregate and analyse data | | System aggregates order data to generate reports on the most popular menu items, least popular menu items, etc. |
| 1. Generate reports | | System generates reports in a user-friendly format (e.g., tables, charts) for easy analysis. |
| 1. Save reports | | System allows saving generated reports in a designated folder within the restaurant's database or cloud storage. |
| **Variants** | |  |
| 4a. Manager wants to export reports in various formats | | Provides options to export generated reports in different formats such as CSV or PDF. |

*Table 26 - Provide Menu Item Statistics User Task*

**5.7 Provide Online Menu**

|  |  |  |
| --- | --- | --- |
| Task: | Provide online menu | |
| Purpose: | Present the menu online and inform potential customers about the restaurant’s offerings | |
| Trigger/Precondition: | A customer visits the restaurant's website. | |
| Frequency: | Always online and accessible | |
| Critical: |  | |
| **Sub-tasks:** | | **Example solution:** |
| 1. Upload menu items   **Problem**: Staff might upload inaccurate/outdated menu information | | * System allows staff to upload the restaurant’s menu items (including images, descriptions, prices, etc.) to the website. * System allows staff to edit or update the existing menu items information. |
| 1. Display menu content | | System renders the complete and up-to-date menu on the website. |
| **Variants** | |  |
| 2a. Customer accesses the online menu on mobile/tablet | | System implements a user-friendly and responsive website design. |

*Table 27 - Provide Online Menu User Task*

**5.8 Arrange Delivery for Takeaway Orders**

|  |  |  |
| --- | --- | --- |
| Task: | Arrange Delivery | |
| Purpose: | Facilitate customer takeaway orders and send them to third-party delivery services. | |
| Trigger/Precondition: | Customer places a takeaway order and chooses the delivery option. | |
| Frequency: | Average 40 times/day | |
| Critical: |  | |
| **Sub-tasks:** | | **Example solution:** |
| 1. Receive delivery order | | System accepts the order from the customer to initiate delivery with the third-party provider. |
| 1. Order forwarding to third-party service provider | | System automatically forwards takeaway orders with delivery selections to the chosen third-party providers. |
| 1. Notify the customer regarding order dispatch | | System implements automated email or SMS notifications to keep customers informed throughout the delivery process |
| **Variants** | |  |
|  | |  |

*Table 28 - Arrange Delivery User Task*

**6.** **Workflows**

A diagram of a restaurant

Description automatically generated

*Figure 6 – RIS’s Workflow*

**7. Quality Attributes**

**7.1 Usability**

Usability is about how easy it is for users to learn, navigate, and interact with the RIS to achieve their goals (e.g., making reservations, placing orders, and viewing menu items).

This quality attribute is important because a user-friendly RIS allows customers of all ages, disabilities, and technical backgrounds, without any technical know-how to easily make reservations or place orders online without confusion or frustration. This is crucial for encouraging repeat visits and retaining a broad customer base for Relaxing Koala. Additionally, restaurant staff should be able to perform their tasks efficiently, minimising training requirements and reducing the risk of errors.

**7.2 Performance**

Performance implies the system’s ability to respond quickly and handle a high volume of transactions and requests without significant delays or bottlenecks.

Fast performance is crucial for customer satisfaction, allowing customers to make reservations, and make payments quickly and efficiently. On the other hand, slow loading times or response times can lead to frustration and customers abandoning reservations or orders midway.

**7.3 Reliability**

Reliability means that the RIS is able to function correctly and consistently without failures or outages, even under varying loads and conditions.

The RIS will be responsible for critical business operations, such as order management, payment processing, and reservation handling. Any system failures or data loss could result in significant disruptions to the restaurant's operations, leading to customer dissatisfaction, lost revenue, and damage to the business's reputation.

**7.4 Security**

Security ensures that the RIS is protected against unauthorised access, data breaches and other malicious threats.

The RIS will handle sensitive customer information, such as contact details and payment data. Therefore, ensuring the security of this information is crucial to maintaining customer trust and complying with relevant data protection regulations. The software must be designed to prevent unauthorised access and store sensitive data with proper protection, ensuring data confidentiality and integrity.

**7.5 Scalability**

Scalability is the ability of the system to handle increasing workloads and accommodate future growth or expansion of the restaurant’s operations.

As The Relaxing Koala has already expanded its capacity, the RIS should be designed with flexible scalability in mind. The system should be able to adapt to potential future growth, such as additional locations, increased customer volumes, delivery and the integration of new features or services without significant disruptions, significant cost adjustment or the need for complete system replacements.

**8. Other Requirements**

**8.1 Product-level requirements**

* The system shall provide a feature for customers to make reservations online.
* The system shall enable order placement and management for customers and staff.
* The system shall facilitate payment processing for customer orders.
* The system shall generate invoices and receipts for customer orders.
* The system shall track and provide menu item statistics.

**8.2 Design-level requirements**

* The system shall have separate user interfaces and access levels for customers, waiters, kitchen staff, and administrators.
* The reservation management interface shall provide a calendar view for staff to visualise and manage reservations effectively.
* The online ordering interface shall have an intuitive shopping cart, with options to add, remove, or modify items, and display a total amount before checkout.
* The website interface shall incorporate the business logo on every page to reinforce brand identity.

**9. Validation**

During the creation of the requirements specification for RIS, various validation steps were conducted to ensure the system aligns with Relaxing Koala’s needs. Firstly, reviews and walkthrough sessions with key stakeholders (owners, managers, staff) to validate requirements against their needs and expectations.

This process includes:

* Identify and go through the user tasks outlined in the case study.
* For each task, brainstorm, and document at least 2-3 alternative solutions that could achieve the desired outcome.
* Reviewing existing task descriptions and assessing if they can be applied to each of the identified solutions. Refine the task descriptions if needed.

Furthermore, as part of the validation, a CRUD (Create, Read, Update, Delete) check was performed to validate that the system adequately supported these fundamental operations as outlined in the requirements.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Entity**  **Task** | **Customer** | **Menu** | **Reservation** | **Payment** | **Staff** | **Order** | **Statistic** | **Invoice/**  **Receipt** | **Delivery** |
| **Make reservations** | C R U D |  | C U |  | R |  |  |  |  |
| **Place order** | C R D | R |  |  |  | C R |  |  |  |
| **Inform order information to kitchen** |  |  |  |  | R | R |  |  |  |
| **Handle payment** | R U |  |  | C R | R | U |  | R |  |
| **Generate receipt/invoice** | R |  |  | C R U | R | R U |  | C R |  |
| **Provide ordered menu item’s statistic** |  | R U |  |  | R | R | C R |  |  |
| **Provide online menu** |  | R |  |  | R U |  |  |  |  |
| **Arrange delivery for takeaway orders** |  |  |  |  | R | R |  |  | C R |

*Table 29 - CRUD Check*

**Note**: "C" denotes Create, "R" denotes Read, and "U" denotes Update, “D” denotes Delete.

**10.** **Verifiability**

**Verifiable requirements:**

All identified functional requirements are verifiable through testing:

* **Make reservations**: perform a series of test cases to make reservations for different dates, times, and party sizes, including edge cases like fully booked scenarios or invalid inputs.
* **Place order**: test cases to verify a complete order placement process, from menu browsing, item selection, customisation, and order submission and measure the order success rate, error rates, response times, and accuracy of order data stored in the database.
* **Inform order information to kitchen**: place orders through RIS and verify that all orders are correctly displayed in the kitchen staff interface within a specified timeframe (e.g., 2 minutes).
* **Handle payment**: test the payment process using different payment methods for a range of order totals and scenarios (e.g., partial payments), record and measure the payment success rate and error rates.
* **Generate invoice/receipt**: generate receipts/invoices for a diverse set of order scenarios, including different item combinations, discounts, taxes, and payment methods. Verify the accuracy of receipt/invoice content, and formatting.
* **Provide ordered menu item statistics**: simulate a defined period of order data (e.g., 1 month) with known order patterns and menu item popularity. Generate reports and statistics for ordered menu items and verify the accuracy, completeness, and consistency of the reported data against expected values calculated from the simulated data.
* **Provide online menu**: conduct cross-browser and cross-device testing to verify the accessibility, responsiveness, and usability of the online menu interface and measure load times.
* **Arrange delivery for takeaway orders**: place a set of takeaway orders with delivery, simulating various delivery locations (e.g., nearby, far away, invalid addresses). Verify the accuracy of captured delivery addresses, order details, and successful delivery arrangement as well as their timeliness.

The majority of quality requirements can be verifiable:

* **Usability**: test with a sample group of users with varying technical skills (beginners, intermediate, advanced) attempting to make online reservations, browse the menu, and measure the time it takes them to complete tasks and collect user feedback through surveys and interviews. Using UI/UX tester tools (e.g., SolarWinds, testsigma) to detect problems and evolve users’ need at early stages.
* **Performance**: use performance testing tools to measure page loading times and system response times for various user tasks.
* **Security (partially)**: simulate a hacking attempt on the RIS database to verify the system’s ability to prevent unauthorised access attempts. Penetrates cybersecurity tester tools (e.g., Wireshark) to detect vulnerabilities in website’s operation.

**Non-verifiable requirements:**

While many requirements can be verified through testing or analysis, the following requirements may not be fully verifiable due to external factors:

* **Security (partially)**: while security can be partially verifiable through penetration testing and security audits, this quality attribute depends on the evolving nature of external threats and potential hackers.
* **Reliability**: the system needs to encounter real-world usage over an extended period, and experience various usage scenarios and edge cases to fully verify reliability requirements.
* **Scalability**: this requirement cannot be tested until the system is deployed and experiences actual usage at scale. Therefore, its verifiability may be limited until real-world usage data is available.