C22716399 Aaron Baggot

[Company name] | [Company address]

[Document title]

[Document subtitle]

# Introduction

In the realm of software engineering, the efficacy and elegance of a system's design hold paramount importance. It is within this context that we embark on extending and refining the existing USE model for a restaurant management system. The restaurant domain, with its intricate interplay of customer interactions, table allocations, and payment processing, provides a fertile ground for exploring various software engineering principles and methodologies.

The primary objective of this endeavour is to augment the functionality of the restaurant management system while meticulously adhering to the principles of coupling and cohesion. By seamlessly integrating new use cases, such as table reservation and dynamic table allocation, we aim to enhance the system's versatility and utility. Moreover, we endeavour to implement a robust payment system, ensuring secure and efficient transaction processing.

At the core of our approach lies the concept of design by contract, wherein every component of the system is imbued with preconditions, postconditions, and invariants. These contractual specifications not only serve as a blueprint for the system's behaviour but also facilitate rigorous testing and validation. By employing OCL (Object Constraint Language) contracts, we articulate the precise conditions under which operations can be invoked, thereby ensuring the system's reliability and robustness.

In our pursuit of excellence, we employ a multifaceted methodology encompassing class diagrams, sequence diagrams, state machines, and object diagrams. Each of these artifacts offers unique insights into the system's structure, behaviour, and interactions, enabling comprehensive analysis and refinement. Furthermore, the integration of testing mechanisms, including !openter and !opexit procedures, enables meticulous validation of system constraints and operation behaviour.

Throughout this report, we shall delve into the intricacies of the extended USE model, elucidating the rationale behind each design decision and presenting a comprehensive analysis of the system's functionality. By adhering to industry best practices and leveraging advanced software engineering techniques, we endeavour to deliver a robust, scalable, and user-centric restaurant management system that exemplifies the pinnacle of software engineering excellence.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Software Project Notes

To improve the design of your system, particularly regarding payment and menu handling, you can make several adjustments for better modularity, efficiency, and ease of maintenance. Here are some suggestions:

Payment System:

Decouple Payment from Customer:

Instead of directly connecting the Cash attribute to the Customer, create a more generic association between Payment and Booking. This allows for more flexibility in handling payments.

Consider having separate associations for Cash and Card payments with bookings, allowing customers to choose their preferred payment method.

Refactor Payment Handling:

Implement a more flexible payment system that can handle various payment methods (Cash, Card, etc.) and can process payments for different types of bookings (WalkIn, Reservation, etc.).

Consider creating a separate class for payments (Payment), which can then be associated with bookings. This decouples the payment process from specific booking types and allows for easier extension in the future.

Error Handling and Logging:

Implement error handling mechanisms within the payment system to handle invalid payments, insufficient funds, etc.

Introduce logging mechanisms to keep track of payment transactions for auditing and debugging purposes.

Menu Handling:

Menu Representation:

Create a class to represent the menu items (MenuItem), with attributes such as name, description, price, etc.

Consider using associations to represent the relationship between menu items and bookings. For example, a booking may have associated menu items that the customer has ordered.

Menu Management:

Implement operations for adding, removing, and updating menu items dynamically. This allows for easy modification of the menu without directly modifying the code.

Consider implementing menu categories or sections to organize menu items more effectively.

Integration with Orders:

Integrate the menu system with the ordering process (Order class). When a customer places an order, the system should retrieve the selected menu items and add them to the order.

Localization and Internationalization:

If your restaurant operates in a multilingual environment, consider adding support for multiple languages in the menu system.

By implementing these improvements, you can create a more robust and maintainable system for handling payments and managing the restaurant menu. Additionally, by decoupling components and following object-oriented principles, you ensure that your system remains flexible and extensible for future enhancements.

**Software Engineering Assignment 2 – Designing a Restaurant Ordering System: A UML Modeling Challenge**

For this assignment expanding the restaurant booking system used in the lab test was selected.

* The requirements include
* class diagrams
* sequence diagrams
* state machines charts
* object diagrams

SelectBooking()

Record booking() are already included

*Reservation Status*

* Pending -> Confirmed: When the reservation is confirmed by the restaurant.
* Pending -> Cancelled: When the reservation is cancelled by either the customer or the restaurant.
* Confirmed -> Cancelled: When the reservation is cancelled after being confirmed.

*Table availability*

* Available -> Reserved: When a table is reserved by a customer.
* Reserved -> Available: When a reservation for a table is cancelled.
* Available -> Occupied: When a customer is seated at a table.
* Occupied -> Available: When the customer finishes dining and leaves the table.

Booking Time window

Open

Closed

* Open -> Closed: When the booking time window for a particular day closes.

*Reservation Payment State machine*

Unpaid

Paid

Unpaid -> Paid: When the reservation payment is successfully processed.

Customer Arrival State Machine

Not Arrived

Arrived

* Not Arrived -> Arrived: When the customer arrives at the restaurant for their reservation.

Reservation Modification State Machine:

States: Unmodified, Modified

Transitions:

Unmodified -> Modified: When the reservation details are modified (e.g., change in party size, time, or special requests).

Waitlist Status State Machine:

States: Not Waitlisted, Waitlisted

Transitions:

Not Waitlisted -> Waitlisted: When the customer opts to be added to the waitlist due to unavailability of tables.

Restaurant Closing Status State Machine:

States: Open, Closing, Closed

Transitions:

Open -> Closing: When the restaurant is preparing to close for the day.

Closing -> Closed: When the restaurant completes closing procedures.

Reservation Feedback State Machine:

States: Feedback Pending, Feedback Submitted

Transitions:

Feedback Pending -> Feedback Submitted: When the customer submits feedback for their dining experience.

* **Add/Modify tables:** To add, remove, or modify a table in the system.
* **Search tables:** To search for available tables for reservation.
* **Place order:** Add a new order in the system for a table.
* **Update order:** Modify an already placed order, which can include adding/modifying meals or meal items.
* **Create a reservation:** To create a table reservation for a certain date/time for an available table.
* **Cancel reservation:** To cancel an existing reservation.
* **Check-in:** To let the guest check in for their reservation.
* **Make payment:** Pay the check for the food.

**Description**

You work for a new start-up that wants to develop and sell software that allows restaurants to digitalize their menus and create an order process for their customers. The goal is to streamline the logistics for a food business. We need to track orders going through three stages: customer, kitchen, and waiter/delivery person.

The software should be designed such that it can be used for any type of restaurant, from pizzeria to bistro, from lunch café to snack bar. Both for in-restaurant dining as well as take-away and delivery. Your goal at the start of the development is to create a comprehensive design of all the classes involved.

**Users and Stakeholders:**

The users of the software will be restaurant staff and customers, while the stakeholders will be the restaurant owners and the software development team.

**Features and Requirements:**

The software should have the following features and requirements:

• A restaurant menu presenting all dishes with their ingredients, drinks, and desserts.

• An order-entry module where customers, servers, or other systems can enter orders:

• Through a web-interface for home delivery.

• Through a web-interface for in-restaurant ordering by scanning a QR code, browsing through the menu and ordering.

• By a server/worker in the restaurant who takes and enters the customer’s order.

• By offering their menu through an API to Uber-eats / Thuisbezorgd / Takeaway / Deliveroo etc.

• A delivery module that manages deliveries to customers.

• The ability to calculate prices using custom-made classes that take into account different pricing models, such as discounts and happy hours.

• Compatibility with different types of restaurants and the ability to integrate with third-party delivery services.

identify Additional Use Cases:

Brainstorm and identify additional use cases besides the basic ones like borrowing/returning in the library or selecting booking/recording arrival in the restaurant. For example, you could consider use cases like changing a table or reserving a book.

Modelling Use Cases:

Describe the extra use case scenarios in detail before modeling them. This could involve defining the steps involved, actors participating, and any preconditions or postconditions.

Model these additional use cases using appropriate diagrams such as use case diagrams or activity diagrams.

Implement USE Model:

Create a USE model for the extended scenarios, including preconditions, postconditions, and invariants. Ensure to include the USE code in your report, as well as submitting it separately.

Test the constraints using objects. Utilize SOIL implementations to facilitate testing, but also include testing using !openter and !opexit for at least one method.

Diagram Creation:

Use a tool like UMLet or Visual Paradigm to create the required diagrams (class, sequence, state machine, object). Ensure these diagrams are of high quality and clearly illustrate the system's behaviour and structure.

Capture screenshots of these diagrams and paste them into your report. Use Windows snipping tool or similar tools for this purpose.

Sequence Diagrams in USE:

Create at least two sequence diagrams in USE to depict the interactions between objects or actors in the system for specific scenarios. Ensure these diagrams are well-documented and reflect the system's behaviour accurately.

State Machine Modelling:

Choose a class, such as the Reservation class, and create a state machine diagram to represent its various states and transitions. Test drive this state machine by simulating different scenarios and capturing the results.

Paste these state machine diagrams along with testing results into your report.

Discussion and Analysis:

Provide a detailed discussion and analysis section in your report. Reflect on the process of extending and testing the USE model, discussing any challenges encountered and how they were addressed.

Analyze the effectiveness of the extended model in capturing the system's requirements and constraints.

Report Compilation:

Compile all the diagrams, testing results, and analysis into a Word document. Ensure the document is well-organized and follows the required format.

Convert the Word document to PDF before submission, as per the submission guidelines.

Remember to follow the provided guidelines closely and ensure the quality of both the diagrams and the report content. Effective communication of the extended USE model and its testing is crucial for the success of your project.

**Pricing:**

Different restaurants use different ways to calculate the prices of their dishes. Therefore, you’ll design the software such that it is possible to use on or more custom-made classes that calculate prices. For instance:

• Restaurant A wants their dishes to be 10% more expensive if eaten in.

• Restaurant B offers a discount of 5% for lunch orders placed between 10.00 and 11.00 a.m.

• Restaurant C has a happy hour between 16:00 and 18:00, two cocktails for the price of 1!

• Restaurant C also wants their dishes to be 5% more expensive if eaten in.

Our company will offer this customization as a bespoke service when the restaurant purchases the software. This means that, our company will implement the pricing strategy in Java for the customer. However, the design must allow for uniform implementation, addition, and modification of the pricing strategy. Ensuring that restaurants can easily change their pricing strategy. Therefore, in the design an interface should be included that allows the implementation of arbitrary pricing strategies.

**To achieve this, the system will consist of several modules:**

1. The menu module which presents all dishes with their ingredients, drinks, and desserts. Each restaurant has a menu with different menu items. These items can consist of drinks, main dishes, side dishes, desserts, etc. In some cases, such as main dishes, the price of the dish will depend on its ingredients.
2. The order-entry module allows customers, servers, or other systems to enter orders. Customers can place orders through a web interface for home delivery, through a web interface for in restaurant ordering by scanning a QR code and browsing through the menu, or by a server/worker in the restaurant who takes and enters the customer’s order. The restaurant can also offer their menu through an API to third-party delivery services such as Uber Eats, Takeaway, and Deliveroo.
3. The pricing module allows for custom pricing for each restaurant. Different restaurants use different ways to calculate the prices of their dishes. Therefore, the software should be designed such that it is possible to use one or more custom-made classes that calculate prices.
4. The delivery module manages deliveries to customers. If customers order through the web interface, a delivery must be made. The restaurant has several deliverers who deliver orders. When an order is placed, it takes some time to prepare it in the kitchen. After this, the order should be delivered as soon as possible by an available deliverer.

The goal of this project is to streamline the logistics of a food business by tracking orders going through three stages: customer, kitchen, and waiter/delivery person. To achieve this, we need a comprehensive design of all the classes involved in the software.

**Summary of requested deliverables:**

• **UML class diagram**: This should include important constructors, public methods, and properties of the classes that will implement the software's features. Methods and properties that are not important to understanding how the software functions may be omitted. Ensure that you include them where they matter.

• **Three Sequence diagrams**: Draw separate sequence diagrams for all three ordering-entry methods described in the assignment.

• **All class and interface definitions**: Write the classes, interfaces, fields, and important methods that define your class model. Implementation of methods is not necessary at this stage.

• **Package structure**: Place your classes and interfaces in appropriate packages (for Java, this means using an appropriate directory structure). Ensure that classes that belong together are in the same package (i.e. in the same directory), and create loosely coupled dependencies between packages to avoid circular dependencies and too many dependencies between classes that should not have dependencies. Ensure that your class diagram matches your package structure.

• **Sample instances**: Create a few sample instances of whatever is aggregated by the relations in your class diagram (e.g. starters, main dishes, drinks, desserts for the menu). Create a few orders and deliveries as well. Every relation between two classes in your class diagram must be shown using examples in your code.

**Some remarks and tips:**

• In terms of class design. You do not have to model the graphical user interface (GUI), only the back-end of the software. Meaning all classes that form the structure (the models, controllers and interfaces) of the software and how they relate.

• Make sure to include the user and GUI as actor and lifeline in the sequence diagram.

• You should however, include a class that acts as the API to the external ordering websites (Uber eats etc.). Think about how you could generalize this, internally you have one menu, but you must “present” this menu to different external services. Similarly, external services must be able to connect to your software to place orders. If you have no knowledge of how API’s work, you can either base the class on your own assumptions or first do some research online (search “REST API”) for examples and approaches.

• A good starting point may be to first define the classes needed to model the software. Next, see which classes aggregate others. Can you make the design as compact as possible?

• Once you think you have a correct diagram, write out the classes, make “sample” objects of your classes and see if your model still makes sense. Adjust if needed.

• Consider that the class diagram and your “empty” classes should be sufficient to start implementing the project without having to change the structure later on. Multiple teams will start working on implementing the software and they must all be able to trust the design.

• Check if your design can be structured into cohesive packages that have clear dependencies on other packages. At most one or two classes in a package should depend on another package.

• The pricing strategy should be able to take into account any type of pricing that the customer wants. The three examples in the assignment are nothing more than examples. It's up to you to design a facility in your class diagram that allows programmers to create pricing strategies as "plug-in" classes. Think about what we talked about in the OOD lecture that would allow for such a mechanism

• In your class diagram you may treat the web interface and API as "black boxes" you do not have to model them in detail. So just include one or two classes as you see fit that "create" orders for your system.

• Consider that, in your class diagram you are modeling everything you need to consider to represent the state space of your program. You are not modelling the flow of your program, but how to represent all the possible states your system can be in.

• Please don't include any extra functionality in your design, only what is described in the assignment.

• Is the kitchen an entity that you need to model in your class design?

• Make sure you identify where instances, abstract classes and inheritance are needed

1st step

**Step 1**

Based on the requirements provided, here is a suggested UML class diagram for the Restaurant Ordering System:

+-------------------+

| <<interface>> |

| PricingStrategy |

+-------------------+

| |

| +calculatePrice()|

+-------------------+

^

|

|

+---------------------|---------------------+

| | |

| | |

| +-----------------+ |

| | MenuItem | |

| +-----------------+ |

| | -name: String | |

| | -price: double | |

| +-----------------+ |

| ^ |

| | |

| | |

| +----------------------+ |

| | MainCourse | |

| +----------------------| |

| | -ingredients: String[]| |

| +----------------------+ |

| |

| |

| +----------------------| |

| | Drink | |

| +----------------------| |

| | | |

| +----------------------+ |

| |

| |

| +----------------------| |

| | Dessert | |

| +----------------------| |

| | | |

| +----------------------+ |

| |

| |

| +----------------------| |

| | Menu | |

| +----------------------| |

| | -items: MenuItem[] | |

| +----------------------+ |

| | -addMenuItem() | |

| | -removeMenuItem() | |

| | -getMenuItem() | |

| +----------------------+ |

| |

| |

| +----------------------| |

| | Order | |

| +----------------------| |

| | -items: MenuItem[] | |

| | -status: OrderStatus | |

| +----------------------+ |

| | -addItem() | |

| | -removeItem() | |

| | -calculateTotalPrice()| |

| +----------------------+ |

| |

| |

| +----------------------| |

| | Delivery | |

| +----------------------| |

| | -order: Order | |

| | -status: DeliveryStatus| |

| | -deliverer: Deliverer| |

| +----------------------+ |

| | -assignDeliverer() | |

| | -updateStatus() | |

| +----------------------+ |

| |

| |

| +----------------------| |

| | Restaurant | |

| +----------------------| |

| | -name: String | |

| | -menu: Menu | |

| | -deliveryService: DeliveryService| |

| +----------------------+ |

| | -receiveOrder() | |

| | -updateOrderStatus() | |

| +----------------------+ |

| |

| |

| +----------------------| |

| | DeliveryService | |

| +----------------------| |

| | | |

| +----------------------+ |

| |

| |

| +----------------------| |

| | Deliverer | |

| +----------------------| |

|

| +----------------------| |

| | -name: String | |

| | -available: boolean | |

| +----------------------+ |

| | -markAvailable() | |

| | -markUnavailable() | |

+----------------------| |

| |

| |

+----------------------|---------------------+

| | |

| | |

| +---------------------------------+

| | <<interface>> OrderEntryMethod |

| +---------------------------------+

| | |

| | +enterOrder() |

| +---------------------------------+

| ^

| |

| |

| +-----------------|-----------------+

| | WebInterface | |

| +-----------------|-----------------+

| | -restaurant: Restaurant |

| +-----------------|-----------------+

| |

| |

| +-----------------|-----------------+

| | Server | |

| +-----------------|-----------------+

| | -restaurant: Restaurant |

| +-----------------|-----------------+

| |

| |

| +-----------------|-----------------+

| | ThirdPartyAPI | |

| +-----------------|-----------------+

| | |

| +---------------------------------+

| |

| |

| +-----------------|-----------------+

| | Kitchen | |

| +-----------------|-----------------+

| | -order: Order |

| +-----------------|-----------------+

| |

| |

| +-----------------|-----------------+

| | Customer | |

| +-----------------|-----------------+

| | -name: String |

| +-----------------|-----------------+

|  |  |
| --- | --- |
|  |  |

Note: The above class diagram provides an overview of the class structure and their relationships based on the given requirements. It is not a complete and final design, but rather a starting point for further refinement and implementation.

I will now provide sequence diagrams for each of the three order-entry methods described in the assignment.

**Explanation:**

PricingStrategy (Interface):

This interface represents the pricing strategy that can be implemented by custom-made classes for calculating prices.

It includes a method called **calculatePrice()** that will be implemented by the specific pricing strategy classes.

MenuItem:

Represents a generic menu item that can include drinks, main courses, side dishes, desserts, etc.

It has properties such as **name** and **price**.

MainCourse, Drink, Dessert:

These are specific types of menu items that inherit from the MenuItem class.

Each type of menu item can have its own additional properties or behaviors specific to its category.

Menu:

Represents the restaurant's menu, which is a collection of menu items.

It has an array or list of MenuItem objects.

It provides methods to add, remove, and retrieve menu items.

Order:

Represents an order placed by a customer.

It contains an array or list of MenuItem objects.

The order also has a status (OrderStatus) indicating its progress.

Delivery:

Represents a delivery made to a customer.

It is associated with an Order and a Deliverer.

The delivery has a status (DeliveryStatus) indicating its progress.

Restaurant:

Represents a restaurant.

It has properties like **name**, **menu** (an instance of Menu), and **deliveryService** (an instance of DeliveryService).

The restaurant can receive orders and update their status.

DeliveryService:

Represents a delivery service that handles the delivery process.

It acts as an external entity interacting with the system.

The specific implementation details of the delivery service can be encapsulated within this class.

Deliverer:

Represents a person responsible for delivering orders.

It has properties like **name** and **available**.

The deliverer can be marked as available or unavailable for deliveries.

WebInterface, Server, ThirdPartyAPI:

These classes represent different order entry methods.

WebInterface allows customers to place orders through a web interface.

Server represents the staff in the restaurant who takes and enters customer orders.

ThirdPartyAPI represents an external API through which orders can be placed.

Kitchen:

Represents the kitchen where orders are prepared.

It is associated with an Order and manages the order preparation process.

Customer:

Represents a customer who places an order.

It has a property **name** to identify the customer.

The class diagram shows the relationships between the classes and their properties and methods. It provides an overview of the system's structure and entities involved in the restaurant ordering process.

Next, let's move to the sequence diagrams for the three order-entry methods.

complete UML class diagram for the Restaurant Ordering System:

+-------------------+

| PricingStrategy |

+-------------------+

| |

| +calculatePrice() |

+-------------------+

^

|

|

+---------------------|---------------------+

| | |

| | |

| +-----------------+ |

| | MenuItem | |

| +-----------------+ |

| | -name: String | |

| | -price: double | |

| +-----------------+ |

| ^ |

| | |

| | |

| +----------------------+ |

| | MainCourse | |

| +----------------------| |

| | -ingredients: String[]| |

| +----------------------+ |

| |

| |

| +----------------------| |

| | Drink | |

| +----------------------| |

| | | |

| +----------------------+ |

| |

| |

| +----------------------| |

| | Dessert | |

| +----------------------| |

| | | |

| +----------------------+ |

| |

| |

| +----------------------| |

| | Menu | |

| +----------------------| |

| | -items: MenuItem[] | |

| +----------------------+ |

| | -addMenuItem() | |

| | -removeMenuItem() | |

| | -getMenuItem() | |

| +----------------------+ |

| |

| |

| +----------------------| |

| | Order | |

| +----------------------| |

| | -items: MenuItem[] | |

| | -status: OrderStatus | |

| +----------------------+ |

| | -addItem() | |

| | -removeItem() | |

| | -calculateTotalPrice()| |

| +----------------------+ |

| |

| |

| +----------------------| |

| | Delivery | |

| +----------------------| |

| | -order: Order | |

| | -status: DeliveryStatus| |

| | -deliverer: Deliverer| |

| +----------------------+ |

| | -assignDeliverer() | |

| | -updateStatus() | |

| +----------------------+ |

| |

| |

| +----------------------| |

| | Restaurant | |

| +----------------------| |

| | -name: String | |

| | -menu: Menu | |

| | -deliveryService: DeliveryService| |

| +----------------------+ |

| | -receiveOrder() | |

| | -updateOrderStatus() | |

| +----------------------+ |

| |

| |

| +----------------------| |

| | DeliveryService | |

| +----------------------| |

| | | |

| +----------------------+ |

| |

| |

| +----------------------| |

| | Deliverer | |

| +----------------------| |

| | | |

+----------------------|-----------------+

|

| +----------------------| |

| | -name: String | |

| | -available: boolean | |

| +----------------------+ |

| | -markAvailable() | |

| | -markUnavailable() | |

+----------------------|-----------------+

|

|

+----------------------|---------------------+

| | |

| | |

| +---------------------------------+

| | <<interface>> OrderEntryMethod |

| +---------------------------------+

| | |

| | +enterOrder() |

| +---------------------------------+

|

|

+-----------------|-----------------+

| WebInterface | |

+-----------------|-----------------+

| -restaurant: Restaurant |

+-----------------|-----------------+

|

|

+-----------------|-----------------+

| Server | |

+-----------------|-----------------+

| -restaurant: Restaurant |

+-----------------|-----------------+

|

|

+-----------------|-----------------+

| ThirdPartyAPI | |

+-----------------|-----------------+

| |

+---------------------------------+

|

|

+-----------------|-----------------+

| Kitchen | |

+-----------------|-----------------+

| -order: Order |

+-----------------|-----------------+

|

|

+-----------------|-----------------+

| Customer | |

+-----------------|-----------------+

| -name: String |

+-----------------|-----------------+

Please note that the class diagram provided represents the structure and relationships of the classes involved in the Restaurant Ordering System based on the given requirements. It serves as a foundation for further implementation and refinement of the system.

**Software Engineering Assignment 2 Report: Designing a Restaurant Ordering System**

Introduction:

The aim of this report is to present the design of a Restaurant Ordering System, which is intended to streamline the logistics for food businesses by digitalizing menus and creating an efficient order process for customers. This report will include a UML class diagram, sequence diagrams for different order entry methods, class and interface definitions, package structure, and sample instances.

UML Class Diagram:

The provided UML class diagram outlines the structure of the Restaurant Ordering System. It includes classes such as MenuItem, Menu, Order, Delivery, Restaurant, DeliveryService, Deliverer, PricingStrategy, and various interfaces like OrderEntryMethod. The relationships between these classes depict how they interact within the system to facilitate the ordering process efficiently.

Sequence Diagrams:

Three separate sequence diagrams illustrate the order-entry process through different methods: WebInterface, Server, and ThirdPartyAPI. These diagrams depict the interactions between actors and the system components, showcasing the flow of events during order placement.

Class and Interface Definitions:

Each class and interface in the system is defined with its properties, methods, and relationships. For instance, the MenuItem class represents generic menu items, while subclasses like MainCourse, Drink, and Dessert inherit from it. Similarly, the Order class encapsulates order details, including menu items and status.

Package Structure:

The system's classes and interfaces are organized into packages to manage dependencies and maintain a modular structure. The package structure ensures loose coupling between components and facilitates scalability and maintenance.

Sample Instances:

Sample instances of classes like Menu, Order, Delivery, Restaurant, and Deliverer are provided to demonstrate how the system entities interact and relate to each other. These instances help visualize the practical implementation of the class diagram.

Conclusion:

The design presented in this report lays the foundation for the development of a comprehensive Restaurant Ordering System. It encompasses essential features like menu management, order processing, delivery management, and pricing customization. Further refinement and implementation based on this design will lead to the creation of a robust and user-friendly software solution for restaurants to streamline their operations.

Overall, the design adheres to the specified requirements and provides a clear understanding of the system's architecture and functionality.

**To extend and test the provided restaurant model in USE, we can follow the requirements outlined. Here's a breakdown of the steps we can take:**

Identify Additional Use Cases:

Since the provided model focuses on booking and seating customers in a restaurant, we can consider adding use cases like modifying a booking, canceling a booking, or viewing available tables.

Modeling Use Cases:

Describe these additional use case scenarios in detail before modeling them. Define the steps involved, actors participating, and any preconditions or postconditions.

Extend the Model:

Modify the BookingSystem class to include operations for the additional use cases identified. These operations should have appropriate preconditions, postconditions, and invariants.

Diagram Creation:

Create or modify diagrams such as class diagrams, sequence diagrams, state machine diagrams, and object diagrams to accommodate the extended model. Ensure these diagrams are comprehensive and clearly illustrate the system's behavior and structure.

Testing:

Test the extended model by simulating different scenarios and ensuring that preconditions, postconditions, and invariants are met.

Utilize SOIL implementations to facilitate testing constraints with objects. Also, conduct testing using !openter and !opexit for at least one method.

Report Compilation:

Compile all the diagrams, testing results, and analysis into a Word document.

Use Windows snipping tool or similar tools to capture screenshots of diagrams and testing results and paste them into your report.

Provide a detailed discussion and analysis section reflecting on the process, challenges encountered, and the effectiveness of the extended model.

To proceed with extending and testing the model, you can follow these steps and provide more details about the specific additional use cases you'd like to include and any modifications you'd like to make to the existing model. Let me know if you need further assistance with any specific aspect of this task!