Group 15
Professor Donyanavard
CS 250
10/11/24

Ramen House Online Ordering System

1. Software Title

Ramen House Online Ordering System

2. Team Members

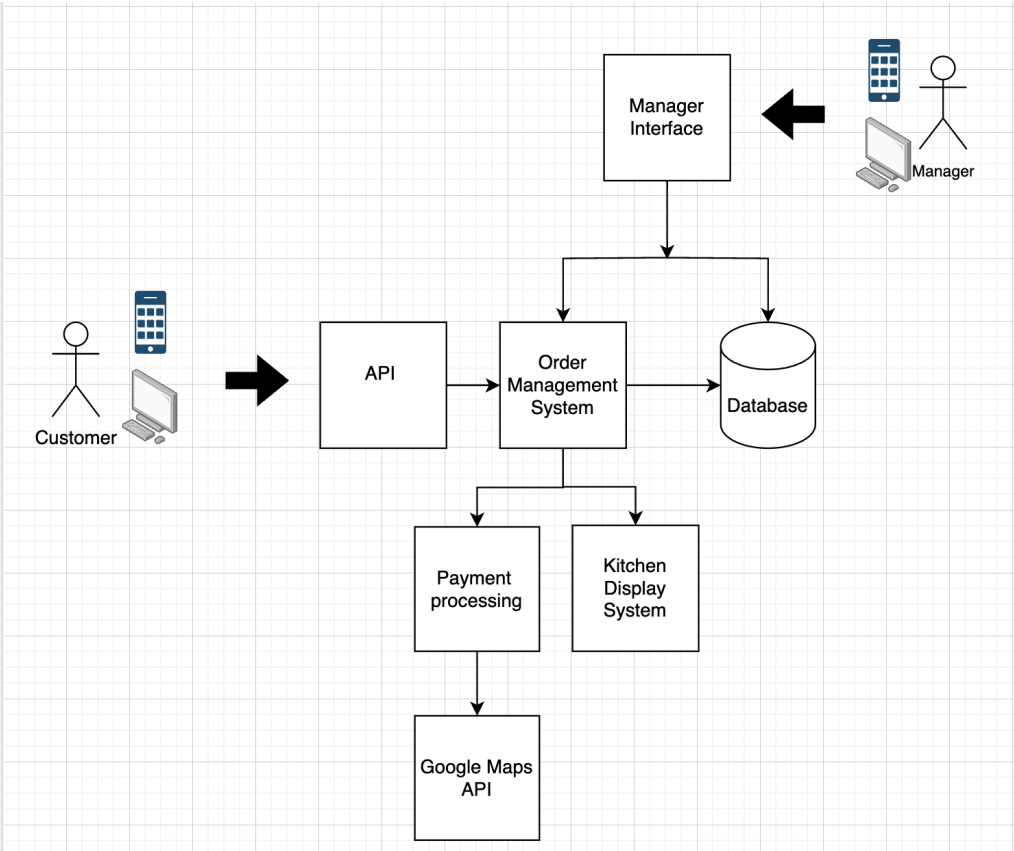
Jennifer Zhu, Amir Javadnia, Julian Stinnett

3. System Description

The Ramen House Online Ordering System intends to promote an increase in profit by enabling customers to conveniently place orders online. Customers can choose to either pick up their orders in-store or have them delivered by Ramen House drivers. They will have access to the entire menu, which includes detailed descriptions and ingredient lists for each dish, as well as the ability to browse their order history. Managers will have the capability to update the menu and pricing as needed, including adding or removing dishes.

4. Software Architecture Overview

4.1 Architecture Diagram

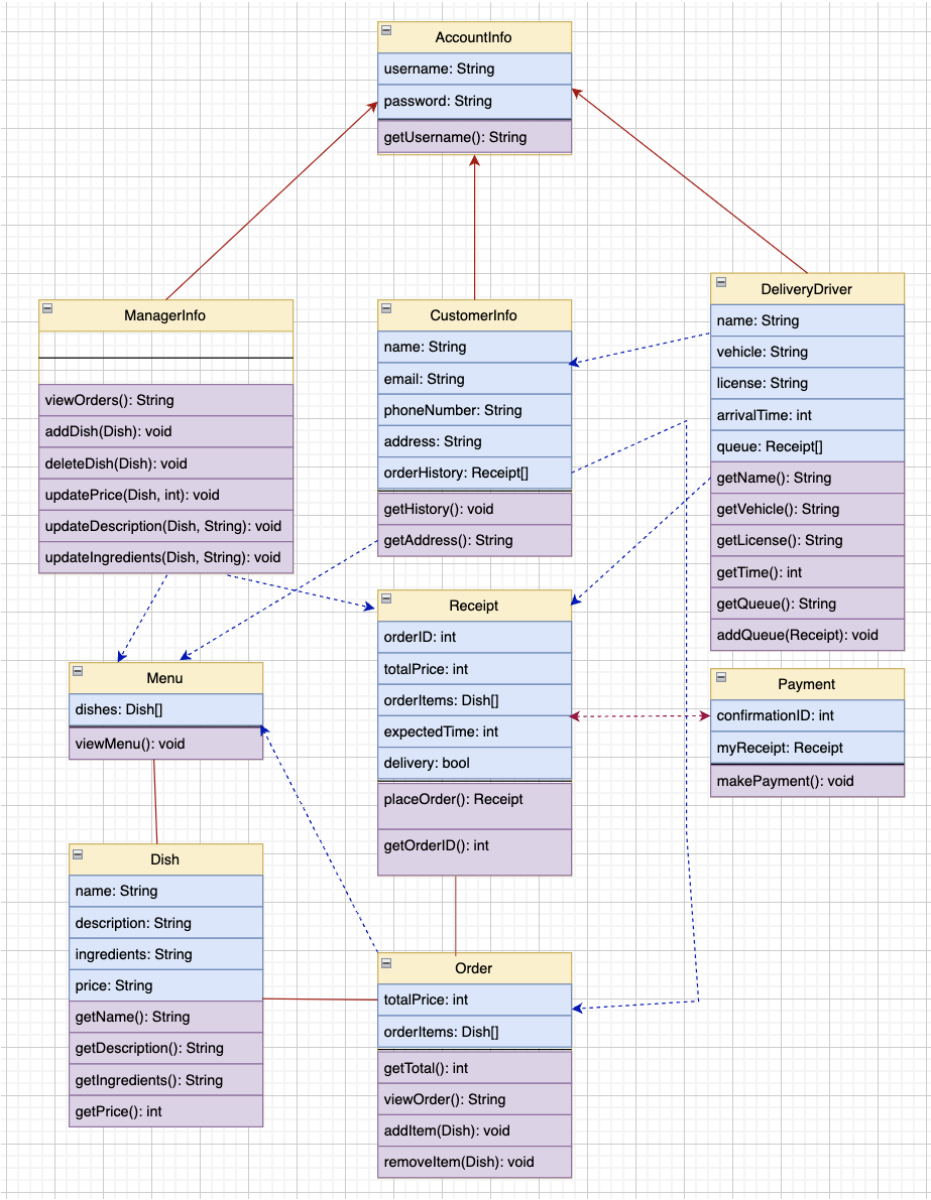


4.2 Description of Architecture Diagram

- **API:** Customers will interact with API to browse the full ramen menu and add items to their cart. When the customers are done adding items to their cart, this order information will go to the Order Management system.
- **Order Management System:** Once the order information is received, this will organize all information and send it to the database to be stored. It will also send the same information to the payment processor, and kitchen display.
- **Database:** This will hold all menu item information like prices and various items. It will also store customer information, revenue and order information.
- **Payment Processing:** this will provide a secure way for customers to checkout when they are finished adding items to their cart.
- **Kitchen Display:** Once information is received from the order management system, this will display all orders in chronological order.

- **Google Maps API:** Once order is complete from the payment processor and the delivery driver takes the order, customers will be sent here to track their orders location while it is enroute.
- **Manager Interface:** this will provide management with the ability to manipulate orders in the order management system. Also it will provide access to the database to retrieve or change information on orders, menu items, and financial information.

4.3 UML Class Diagram



4.4 Description of Classes

- **AccountInfo:** This class manages the login information for all user types, and directs users to the appropriate interface based on whether they are a customer, manager, or delivery driver.
- **CustomerInfo:** This class stores the information customers must provide for order delivery, such as their address, and outlines the functionalities available to them.
- **ManagerInfo:** This class facilitates the actions required by managers to maintain an up-to-date system.
- **DeliveryDriver:** This class provides details about the delivery driver to assist customers in identifying their delivery.
- **Menu:** This class organizes an overview of the main menu to display all available dishes.
- **Dish:** This class contains detailed information about each dish, including its description, ingredients, and price.
- **Order:** This class manages the customer's order cart by tracking added dishes and calculating the total price.
- **Receipt:** This class confirms the customer's order, including a unique order ID, estimated completion time, and whether the order is for pick-up or delivery.
- **Payment:** This class processes payments and provides a confirmation ID and receipt to customers.

4.5 Description of Attributes

- **username: String** - This is a unique username for all users to login with.
- **password: String** - This is a unique password for all users to login with
- **name: String** - This is used in the CustomerInfo and DeliveryDriver class to identify the customer’s or driver’s name.
- **email: String** - This is the customer’s email for receiving receipts and delivery updates.
- **phoneNumber: String** - This is the customer’s phone number for receiving receipts and delivery updates.
- **address: String** - This is the customer’s delivery address for food drop-off by drivers.
- **orderHistory: Receipt[]** - This is an array of receipts that records previous orders.

- **vehicle: String** - This is a description of the delivery driver's vehicle to help customers identify it.
- **license: String** - This is the delivery driver's license plate number for added security in identifying the correct vehicle.
- **arrivalTime: int** - This is the estimated time of arrival for the delivery driver, indicating when customers can expect their order.
- **queue: Receipt[]** - This is the order for the drivers to follow when delivering multiple orders.
- **dishes: Dish[]** - This is an array of dishes currently available on the menu.
- **name: String** - This is the name of the dish.
- **description: String** - This is an appetizing description of the dish.
- **ingredients: String** - This contains a list of ingredients in the dish, allowing customers to check for allergy concerns.
- **price: String** - This is the price of the dish.
- **totalPrice: int** - This is the total price of the dishes added to a customer's cart.
- **orderItems: Dish[]** - This array tracks the dishes in a customer's order.
- **orderID: int** - This is a unique order number on the customer's receipt for easy identification.
- **totalPrice: int** - This is the total price of the order, including tax and tip.
- **orderItems: Dish[]** - This is an array of all dishes ordered by the customer.
- **expectedTime: int** - This is the estimated completion time for the kitchen to prepare the customer's order.
- **delivery: bool** - Delivery will be true if the customer wants a delivery and false if they prefer to pick up the order.
- **confirmationID: int** - This is an ID confirming that the customer's payment was received.
- **myReceipt: Receipt** - This is the customer's copy of the receipt for their records.

4.6 Description of Operations

- **getUsername(): String** - This method returns the user's username.

- **viewOrders(): String** - This method allows the manager to view all incoming orders chronologically, ensuring all are processed.
- **addDish(Dish): void** - This method accepts a Dish as a parameter and adds it to the menu for customer ordering.
- **deleteDish(Dish): void** - This method accepts a Dish as a parameter and removes it from the menu.
- **updatePrice(Dish, int): void** - This method updates the price of the specified dish with the provided amount.
- **updateDescription(Dish, String): void** - This method updates the description of the specified dish with the provided text.
- **updateIngredients(Dish, String): void** - This method updates the ingredients of the specified dish with the provided text.
- **viewMenu(): void** - This method allows customers to view all dishes on the menu by parsing the dishes array.
- **getName(): String** - This method returns the name of the selected dish.
- **getDescription(): String** - This method returns the description of the dish.
- **getIngredients(): String** - This method returns the ingredients of the dish.
- **getPrice(): int** - This method returns the price of the dish.
- **getTotal(): int** - This method calculates and returns the total price of the dishes in the customer's order by summing their prices.
- **viewOrder(): String** - This method returns all of the dishes in the customer's order.
- **addItem(Dish): void** - This method adds the specified Dish to the order.
- **removeItem(Dish): void** - This method removes the specified Dish from the order.
- **placeOrder(): Receipt** - This method completes the transaction, returning a Receipt while adding it to the customer's order history.
- **getOrderID(): int** - This method returns the unique order ID for customers or delivery drivers to identify the correct order.
- **makePayment(): void** - This method handles the payment for the customer's order, offering options for credit card, PayPal, or Apple Pay.
- **getHistory(): void** - This method allows the customer to view their past receipts.

- **getAddress(): String** - This method returns the customer's address for directing the delivery driver.
- **getName(): String** - This method returns the name of the delivery driver, helping customers identify their driver.
- **getVehicle(): String** - This method returns the delivery driver's vehicle description to inform customers which car contains their delivery.
- **getLicense(): String** - This method returns the driver's license plate number for customers to verify the correct vehicle.
- **getTime(): int** - This method returns the estimated arrival time of the driver, indicating when customers can expect their delivery.
- **getQueue(): String** - This method returns the deliveries in the order of first to last for the driver to make.
- **addQueue(Receipt): void** - This method assigns more orders to the back of the queue to indicate the orders drivers are responsible for delivering.

5. Development Plan and Timeline

Because the client requested that this software is completed within six months, the tasks will be delegated to complete this in a timely manner. While responsibilities will be assigned individually, there will be an emphasis on following an iterative process, which includes allocating time for integration and testing to ensure the system is functioning as expected.

5.1 Partitioning of Tasks

There will be three members responsible for the development of this system. To work efficiently, the tasks will be partitioned between the members equally. One member will be responsible for the backend development, including the server-side logic, APIs, and ensuring smooth communication between the frontend and the database. This includes handling user authentication, order processing, and payment integration. The second member will focus on database management, designing and implementing the database to efficiently store and retrieve user information, orders, and menu items. They will also be responsible for optimizing queries to ensure high performance. The third member will handle the frontend development, designing an intuitive user interface that allows customers to easily navigate the menu, place orders, and

receive updates. Every member will work together to test each other's functionality during integration to debug and revise the code as needed. These outlined responsibilities will promote efficient collaboration in building a robust online ordering system for Ramen House.

5.2 Team Member Responsibilities

Effective teamwork is essential for the success of this software. Every team member will be responsible for maintaining consistent communication on progress updates and challenges relevant to the system. If any concerns arise, they should be communicated as soon as possible to be resolved efficiently. Individual responsibilities should be completed within four months to allocate two months to integration, testing, and optimization. Project management tools to track tasks and deadlines will ensure everyone stays on track and is held accountable. Working consistently in an organized manner and fostering a communicative environment will ensure the completion of this online ordering system within the six-month timeline.

Group 15

Professor Donyanavard

CS 250

10/24/24

SDS: Test Plan

Test Set 1: Test Online Ordering

This test verifies the successful placement of a customer's order. It ensures that items are correctly added to a customer's cart and that the order is received by Ramen House.

Unit Test*: Check Add/Remove Item to Order*

Input Vectors:

- Input different ramen dishes like Tonkotsu Ramen, Shoyu Ramen, Curry Ramen, etc. into addItem(Dish)
- Observe if viewOrder() accurately returns all added dish items, including the correct quantity number
- Remove dishes from the order and observe if viewOrder() still correctly reflects the expected order

Targeted Features: Order form functionality

Coverage: This test aims to evaluate the accuracy and reliability of adding and removing dishes from an online order, ensuring that all functionalities work as intended.

Potential Failures: Failures in this test may include the inability to remove a dish from an order, incorrect quantity updates for duplicate items, or inaccurate display of order items.

Example: Add Tonkotsu Ramen, Shoyu Ramen, Tonkotsu Ramen, Curry Ramen. Remove Shoyu Ramen. Viewing the order with viewOrder() should reflect 2 Tonkatsu Ramens, 1 Curry Ramen, 0 Shoyu Ramen.

Integration Test*: Verify Correct Order Receipt*

Input Vectors:

- Add different dishes to an order
- Place the order for Ramen House

- Verify the receipt received by the manager contains correct items

Targeted Features: Ensure correct order is received by restaurant

Coverage: This intends to test the accuracy and reliability of the system transferring the correct order from the customer to the restaurant.

Potential Failures: Failures in this test could include failure to show all items for long orders or failure to reflect correct item quantities.

Example: Add Tonkotsu Ramen, Shoyu Ramen, Tonkotsu Ramen, Curry Ramen, Curry Ramen, Curry Ramen, Lemonade. Remove Curry Ramen. Hit place order. The managers should see an order for 2 Tonkatsu Ramens, 1 Shoyu Ramen, 1 Curry Ramen, and 1 Lemonade.

System Test*: End-to-End Ordering Process*

Input Vectors:

- Test the entire process from browsing the menu to processing payments and receiving an order confirmation.
- Test the past orders are recorded accurately and correctly displayed in the order history.
- Test that the printed receipt paper has all the correct information such as order items, price by each item, total price, order id and the expected delivery time.

Targeted Features: Verifying the reliability of the complete ordering process

Coverage: These tests cover and verify that the entire system of ordering from start to finish functions correctly and ensures a seamless user experience.

Potential Failures: Failures in this test could include issues with payments processing, items unable to get added or removed from the order list, customers might not receive the expected delivery time or the order ID. Additionally, the order history may not display correctly.

Example: A customer will browse the menu and add items to the cart. Then the customer will fill out address and payment information. Once payment is processed the customer will receive an order receipt. As soon the delivery driver takes the order, the customer will have a live google maps location of the driver. Once the order is delivered, all order and payment information will be stored in order history with complete accuracy.

Test Set 2: Test System's Scalability

This test aims to evaluate the reliability of our system in handling large-scale operations that involve numerous sequential actions for all users.

Unit Test: *Reliable Driver Queue*

Input Vectors:

- Add multiple receipts to a driver's delivery queue
- Ensure the queue maintains the correct order for drivers to deliver. Orders should be sorted from earliest to latest order.

Targeted Features: Assigning multiple deliveries to drivers

Coverage: This test aims to evaluate the scalability and reliability of drivers managing multiple order deliveries. The system should effectively prioritize earlier orders to minimize customer wait times.

Potential Failures: Failures in this test may involve an incorrect order of receipts in the queue or the inability to add one or more orders to the queue altogether.

Example: Add multiple receipts to a driver's queue and retrieve the queue using getQueue(). Verify that the queue contains the same number of orders added and that the order of the receipts matches the expected priority sequence.

Integration Test*: Testing Manager's Menu Updates*

Input Vectors:

- Test that the updates in the menu are reflected in the order options.
- Test to ensure that the items from an updated menu can be ordered without issues.

Targeted Features: Verify the manager's menu updates are reflected correctly in the system.

Coverage: These tests check that changes from the manager applied with the addDish(Dish) and deleteDish(Dish) methods are reflected in the orderItems:Dish[], ensuring users can order updated items. The system should be able to handle many updates including additions and removals for a large menu change (i.e. seasonal menu items).

Potential Failures: Failures in this test may involve users not able to order newly added items or continue to see the items that should have been removed.

Example: The manager adds, removes, and updates menu items depending on menu changes. Then, view items in the menu page to check if the item information is correct. If information in the menu is as expected, then add new or updated items to cart and check the menu again. Lastly, complete the order of said items and ensure matching information.

System Test*: High Performance Handling of Customer Orders*

Input Vectors:

- Simulate high volume of order placements to see the system performance and reliability.
- Test the system's ability to handle multiple users placing orders at the same time.

Targeted Features: Handling high volume order

Coverage: These tests assess the system's performance under heavy use to ensure that it is capable of handling high volume orders and maintain its reliability.

Potential Failures: The system struggles to process multiple user orders simultaneously and crashes under heavy use.

Example: Have a test server run our environment and increment the number of simulated customers. Monitor response time for each incrementation and establish where our system response time begins to slow down. Also test to complete failure to find out our maximum possible users at one time. Lastly, increment the number of users submitting an order at the same time to find at what point our system fails.