

Food Order System Requirements

User

A food order and delivery system should offer a convenient and efficient way for users to explore, order, and receive meals from a wide range of local and international cuisines. It should provide an accessible and easy-to-navigate interface, available on both web and mobile platforms, where users can browse menus, read reviews, and customize orders to their preferences. The system should feature real-time order tracking, estimated delivery times, and notifications to keep users updated from the moment an order is placed until it is delivered. Integration with various payment gateways should ensure secure and flexible payment options. Additionally, the system should support promotional features, loyalty programs, and personalized recommendations based on user preferences and order history, enhancing the overall user experience and encouraging repeat business.

Identify functional and non-functional requirements from the system description.

ChatGPT

Functional Requirements:

1. **User Interface**:

- Accessible and easy-to-navigate interface for both web and mobile platforms.
- Ability to browse menus and read reviews.
- Features for customizing orders according to user preferences.

2. **Order Management**:

- Real-time order tracking.
- Display of estimated delivery times.
- Notifications to update users from the moment an order is placed until it is delivered.

3. **Payment Processing**:

- Integration with various payment gateways.

- Secure and flexible payment options.

4. **Promotional and Loyalty Features**:

- Support for promotional features.
- Implementation of loyalty programs.
- Personalized recommendations based on user preferences and order history.

Non-Functional Requirements:

1. **Usability**:

- The interface must be intuitive and easy to use for all user demographics.
- Consistency in design across web and mobile platforms.

2. **Performance**:

- Real-time updates for order tracking and notifications.
- Quick loading times and responsiveness.

3. **Security**:

- Secure payment processing.
- Protection of user data and order history.

4. **Reliability**:

- High availability and uptime of the system.
- Robust handling of high traffic during peak times.

5. **Scalability**:

- Ability to handle an increasing number of users and orders.
- Efficient performance regardless of the number of concurrent users.

6. **Maintainability**:

- Ease of updates and maintenance of the system.
- Clear documentation for future development and troubleshooting.

7. **Compatibility**:

- Support for various web browsers and mobile operating systems.
- Seamless integration with different payment gateways and other third-party services.

User

Create a use case model for the system.

ChatGPT

Use Case Model for Food Order and Delivery System

Actors:

1. **User**
2. **Restaurant**
3. **Delivery Person**
4. **System Administrator**
5. **Payment Gateway**

Use Cases:

1. **Browse Menus**
2. **Read Reviews**
3. **Customize Order**
4. **Place Order**
5. **Track Order**
6. **Receive Notifications**
7. **Make Payment**
8. **Manage Promotions**
9. **Participate in Loyalty Program**
10. **Receive Personalized Recommendations**
11. **Manage Account**
12. **Update Order Status**
13. **Manage Restaurant Menu**
14. **Administer System**

Use Case Descriptions:

1. **Browse Menus**

- **Actor**: User

- **Description**: Users can browse through various restaurant menus available in the system.

2. **Read Reviews**

- **Actor**: User

- **Description**: Users can read reviews of restaurants and dishes before making a selection.

3. **Customize Order**

- **Actor**: User
- **Description**: Users can customize their orders by selecting specific ingredients, portion sizes, and other preferences.

4. **Place Order**

- **Actor**: User
- **Description**: Users can place an order for their selected items.

5. **Track Order**

- **Actor**: User, Delivery Person
- **Description**: Users can track the status of their orders in real-time. Delivery persons can update the status of the order as they progress.

6. **Receive Notifications**

- **Actor**: User
- **Description**: Users receive notifications about the status of their order from placement to delivery.

7. **Make Payment**

- **Actor**: User, Payment Gateway
- **Description**: Users can make payments using various secure payment gateways integrated with the system.

8. **Manage Promotions**

- **Actor**: System Administrator, Restaurant

- **Description**: Administrators and restaurants can manage promotional offers available to users.

9. **Participate in Loyalty Program**

- **Actor**: User
- **Description**: Users can participate in loyalty programs to earn rewards and benefits.

10. **Receive Personalized Recommendations**

- **Actor**: User
- **Description**: Users receive personalized recommendations based on their order history and preferences.

11. **Manage Account**

- **Actor**: User
- **Description**: Users can manage their account details, including personal information and preferences.

12. **Update Order Status**

- **Actor**: Delivery Person, Restaurant
- **Description**: Delivery persons and restaurants can update the status of the order at various stages of preparation and delivery.

13. **Manage Restaurant Menu**

- **Actor**: Restaurant
- **Description**: Restaurants can manage their menus, including adding, updating, or removing items.

14. ****Administer System****

- ****Actor****: System Administrator

- ****Description****: System administrators manage the overall system, including user accounts, security settings, and system performance.

Use Case Diagram:

```plaintext



|         |                             |  |
|---------|-----------------------------|--|
|         | +-----+                     |  |
|         | 9. Participate in Loyalty   |  |
|         | +-----+                     |  |
|         | 10. Receive Recommendations |  |
|         | +-----+                     |  |
|         | 11. Manage Account          |  |
|         | +-----+                     |  |
| +-----+ |                             |  |
|         |                             |  |
| +-----+ |                             |  |
|         | Restaurant                  |  |
|         |                             |  |
|         | +-----+                     |  |
|         | 13. Manage Restaurant Menu  |  |
|         | +-----+                     |  |
|         | 12. Update Order Status     |  |
|         | +-----+                     |  |
|         | 8. Manage Promotions        |  |
|         | +-----+                     |  |
| +-----+ |                             |  |
|         |                             |  |
| +-----+ |                             |  |
|         | Delivery Person             |  |
|         |                             |  |
|         | +-----+                     |  |



```
| 5. Track Order | |
| +-----+ |
| 12. Update Order Status | |
| +-----+ |
+-----+

+-----+
System Administrator	
+-----+	
8. Manage Promotions	
+-----+	
14. Administer System	
+-----+	
+-----+

+-----+
Payment Gateway	
+-----+	
7. Make Payment	
+-----+	
+-----+

...
```

This use case model outlines the primary actors and use cases involved in the food order and delivery system, ensuring a comprehensive overview of system functionalities and interactions.

## User

Create use case specifications for identified use cases.

## ChatGPT

### ### Use Case Specifications

#### #### 1. Browse Menus

- **Actors**: User
- **Description**: Allows users to browse through various restaurant menus available in the system.
- **Preconditions**: User must be logged into the system.
- **Postconditions**: User views the menu of selected restaurants.
- **Main Flow**:
  1. User selects the 'Browse Menus' option.
  2. System displays a list of available restaurants.
  3. User selects a restaurant.
  4. System displays the menu of the selected restaurant.
- **Alternative Flows**:
  - If no restaurants are available, the system displays a message indicating no restaurants are currently available.

#### #### 2. Read Reviews

- **Actors**: User

- **Description**: Allows users to read reviews of restaurants and dishes.
- **Preconditions**: User must be logged into the system.
- **Postconditions**: User reads reviews for selected restaurant or dish.
- **Main Flow**:
  1. User selects a restaurant or dish.
  2. User selects the 'Read Reviews' option.
  3. System displays reviews for the selected restaurant or dish.
- **Alternative Flows**:
  - If no reviews are available, the system displays a message indicating no reviews are currently available.

### #### 3. Customize Order

- **Actors**: User
- **Description**: Allows users to customize their orders by selecting specific ingredients, portion sizes, and other preferences.
- **Preconditions**: User must be browsing a restaurant menu.
- **Postconditions**: User customizes and saves the order.
- **Main Flow**:
  1. User selects an item from the menu.
  2. User selects the 'Customize Order' option.
  3. System displays customization options for the selected item.
  4. User selects preferred options and confirms customization.
  5. System saves the customized order.
- **Alternative Flows**:

- If customization options are not available for the item, the system displays a message indicating the item cannot be customized.

#### #### 4. Place Order

- **Actors**: User
- **Description**: Allows users to place an order for their selected items.
- **Preconditions**: User must have items in the cart.
- **Postconditions**: Order is placed, and confirmation is sent to the user.
- **Main Flow**:
  1. User reviews items in the cart.
  2. User selects the 'Place Order' option.
  3. System displays order summary and payment options.
  4. User selects payment method and confirms the order.
  5. System processes the order and sends a confirmation to the user.
- **Alternative Flows**:
  - If payment fails, the system displays a message indicating the payment error and prompts the user to try again.

#### #### 5. Track Order

- **Actors**: User, Delivery Person
- **Description**: Allows users to track the status of their orders in real-time.
- **Preconditions**: Order must be placed and in progress.
- **Postconditions**: User views the current status of their order.

- **\*\*Main Flow\*\***:
  1. User selects the 'Track Order' option.
  2. System displays the current status of the order (e.g., order received, preparing, out for delivery).
  3. Delivery person updates the order status at various stages.
  4. System updates the order status for the user in real-time.

- **\*\*Alternative Flows\*\***:
  - If the order status is not available, the system displays a message indicating the status is currently unavailable.

#### #### 6. Receive Notifications

- **\*\*Actors\*\***: User
- **\*\*Description\*\***: Allows users to receive notifications about the status of their order from placement to delivery.
- **\*\*Preconditions\*\***: Order must be placed and in progress.
- **\*\*Postconditions\*\***: User receives timely updates on their order status.
- **\*\*Main Flow\*\***:
  1. User places an order.
  2. System sends notifications to the user about the order status (e.g., order confirmed, preparing, out for delivery, delivered).
- **\*\*Alternative Flows\*\***:
  - If notifications fail, the system retries sending the notifications or provides an alternative means of communication.

#### #### 7. Make Payment

- **\*\*Actors\*\***: User, Payment Gateway

- **Description**: Allows users to make payments using various secure payment gateways integrated with the system.
- **Preconditions**: User must have items in the cart and ready to place an order.
- **Postconditions**: Payment is processed, and the order is confirmed.
- **Main Flow**:
  1. User selects items and proceeds to checkout.
  2. User selects a payment method.
  3. System redirects to the payment gateway.
  4. User enters payment details and confirms payment.
  5. Payment gateway processes the payment and confirms to the system.
  6. System confirms the order and sends a confirmation to the user.
- **Alternative Flows**:
  - If payment fails, the system notifies the user and prompts to try another payment method.

#### #### 8. Manage Promotions

- **Actors**: System Administrator, Restaurant
- **Description**: Allows administrators and restaurants to manage promotional offers available to users.
- **Preconditions**: User must have the necessary permissions (administrator or restaurant owner).
- **Postconditions**: Promotional offers are created, updated, or removed.
- **Main Flow**:
  1. User logs in with administrator or restaurant owner account.
  2. User selects the 'Manage Promotions' option.
  3. User creates, updates, or removes promotional offers.

4. System saves the changes and updates the promotional offers for users.

- **Alternative Flows**:

- If there are errors in promotional details, the system notifies the user and prompts for corrections.

#### #### 9. Participate in Loyalty Program

- **Actors**: User

- **Description**: Allows users to participate in loyalty programs to earn rewards and benefits.

- **Preconditions**: User must be logged into the system.

- **Postconditions**: User enrolls in or earns points in the loyalty program.

- **Main Flow**:

1. User selects the 'Loyalty Program' option.

2. User enrolls in the loyalty program or views their current points.

3. User makes purchases and earns loyalty points.

4. System updates the user's loyalty points and rewards.

- **Alternative Flows**:

- If there are issues with the loyalty program, the system notifies the user and provides support options.

#### #### 10. Receive Personalized Recommendations

- **Actors**: User

- **Description**: Allows users to receive personalized recommendations based on their order history and preferences.

- **Preconditions**: User must have a history of orders and preferences.

- **\*\*Postconditions\*\***: User receives personalized recommendations.
- **\*\*Main Flow\*\***:
  1. User logs in and navigates to the recommendations section.
  2. System analyzes user's order history and preferences.
  3. System displays personalized recommendations for the user.
- **\*\*Alternative Flows\*\***:
  - If there is insufficient data for recommendations, the system notifies the user and provides general recommendations.

#### #### 11. Manage Account

- **\*\*Actors\*\***: User
- **\*\*Description\*\***: Allows users to manage their account details, including personal information and preferences.
- **\*\*Preconditions\*\***: User must be logged into the system.
- **\*\*Postconditions\*\***: User updates their account details.
- **\*\*Main Flow\*\***:
  1. User selects the 'Manage Account' option.
  2. User updates personal information and preferences.
  3. System saves the updated account details.
- **\*\*Alternative Flows\*\***:
  - If there are errors in the account details, the system notifies the user and prompts for corrections.

#### #### 12. Update Order Status



- **Actors**: Delivery Person, Restaurant
- **Description**: Allows delivery persons and restaurants to update the status of the order at various stages of preparation and delivery.
- **Preconditions**: Order must be in progress.
- **Postconditions**: Order status is updated in the system.
- **Main Flow**:
  1. Delivery person or restaurant logs in and selects the 'Update Order Status' option.
  2. User updates the order status (e.g., preparing, out for delivery, delivered).
  3. System saves the updated status and notifies the user.
- **Alternative Flows**:
  - If there are errors in updating the status, the system notifies the user and provides support options.

#### ### 13. Manage Restaurant Menu

- **Actors**: Restaurant
- **Description**: Allows restaurants to manage their menus, including adding, updating, or removing items.
- **Preconditions**: Restaurant must be logged into the system.
- **Postconditions**: Menu is updated in the system.
- **Main Flow**:
  1. Restaurant logs in and selects the 'Manage Menu' option.
  2. Restaurant adds, updates, or removes menu items.
  3. System saves the changes and updates the menu for users.
- **Alternative Flows**:

- If there are errors in the menu details, the system notifies the restaurant and prompts for corrections.

#### #### 14. Administer System

- **Actors**: System Administrator
- **Description**: Allows system administrators to manage the overall system, including user accounts, security settings, and system performance.
- **Preconditions**: User must have administrator privileges.
- **Postconditions**: System settings and performance are managed.
- **Main Flow**:
  1. Administrator logs in

#### ChatGPT

and selects the 'Administer System' option.

2. Administrator manages user accounts, security settings, and system performance.

3. System saves the changes and updates the settings.

- **Alternative Flows**:

- If there are errors in system administration, the system notifies the administrator and provides support options.

These specifications provide detailed descriptions of each use case, ensuring clear understanding and implementation of system functionalities.

#### User

Create a domain model based on use case specifications.

#### ChatGPT

### Domain Model for Food Order and Delivery System

#### #### Entities:

1. \*\*User\*\*
2. \*\*Restaurant\*\*
3. \*\*Menu\*\*
4. \*\*MenuItem\*\*
5. \*\*Order\*\*
6. \*\*OrderStatus\*\*
7. \*\*Review\*\*
8. \*\*Payment\*\*
9. \*\*Promotion\*\*
10. \*\*LoyaltyProgram\*\*
11. \*\*Recommendation\*\*
12. \*\*Account\*\*
13. \*\*Notification\*\*
14. \*\*DeliveryPerson\*\*
15. \*\*SystemAdministrator\*\*

#### #### Relationships:

1. \*\*User\*\* can browse \*\*Menu\*\* (1:N)
2. \*\*User\*\* can read \*\*Review\*\* (1:N)
3. \*\*User\*\* can customize \*\*Order\*\* (1:N)
4. \*\*User\*\* places \*\*Order\*\* (1:N)
5. \*\*User\*\* tracks \*\*Order\*\* (1:N)
6. \*\*User\*\* receives \*\*Notification\*\* (1:N)
7. \*\*User\*\* makes \*\*Payment\*\* (1:N)

```
8. **User** participates in **LoyaltyProgram** (1:1)
9. **User** receives **Recommendation** (1:N)
10. **User** manages **Account** (1:1)
11. **Restaurant** has **Menu** (1:1)
12. **Restaurant** has **MenuItem** (1:N)
13. **Restaurant** manages **OrderStatus** (1:N)
14. **Restaurant** receives **Review** (1:N)
15. **Restaurant** manages **Promotion** (1:N)
16. **DeliveryPerson** updates **OrderStatus** (1:N)
17. **SystemAdministrator** administers **System** (1:1)
```

### Class Diagram

```plaintext

```
+-----+
|           User           |
+-----+
| - userID: int            |
| - name: String           |
| - email: String          |
| - address: String        |
| - phone: String          |
+-----+
| + browseMenu()           |
| + readReview()           |
```

```
| + customizeOrder() |
| + placeOrder() |
| + trackOrder() |
| + receiveNotification() |
| + makePayment() |
| + participateLoyaltyProgram() |
| + receiveRecommendation() |
| + manageAccount() |
+-----+

+-----+

| Restaurant |
+-----+

| - restaurantID: int |
| - name: String |
| - address: String |
| - phone: String |
| - email: String |
+-----+

| + manageOrderStatus() |
| + receiveReview() |
| + managePromotion() |
+-----+

+-----+
```

```
|           Menu           |
```

```
+-----+
```

```
| - menuID: int           |
```

```
| - restaurantID: int     |
```

```
+-----+
```

```
+-----+
```

```
|       MenuItem         |
```

```
+-----+
```

```
| - itemID: int           |
```

```
| - menuID: int           |
```

```
| - name: String          |
```

```
| - description: String   |
```

```
| - price: double         |
```

```
+-----+
```

```
+-----+
```

```
|           Order        |
```

```
+-----+
```

```
| - orderID: int          |
```

```
| - userID: int           |
```

```
| - restaurantID: int     |
```

```
| - status: String        |
```

```
| - totalAmount: double   |
```

```
+-----+
```

```
+-----+
```

```
|      OrderStatus      |
```

```
+-----+
```

```
| - statusID: int      |
```

```
| - orderID: int       |
```

```
| - status: String     |
```

```
| - timestamp: DateTime|
```

```
+-----+
```

```
+-----+
```

```
|      Review          |
```

```
+-----+
```

```
| - reviewID: int      |
```

```
| - userID: int        |
```

```
| - restaurantID: int  |
```

```
| - rating: int        |
```

```
| - comment: String    |
```

```
| - timestamp: DateTime|
```

```
+-----+
```

```
+-----+
```

```
|      Payment          |
```

```
+-----+
```

```
| - paymentID: int     |
```

```
| - orderID: int      |
| - amount: double    |
| - method: String     |
| - timestamp: DateTime|
```

```
+-----+
```

```
+-----+
```

```
|      Promotion      |
```

```
+-----+
```

```
| - promotionID: int   |
| - restaurantID: int  |
| - description: String|
| - discount: double   |
| - startDate: DateTime|
| - endDate: DateTime  |
```

```
+-----+
```

```
+-----+
```

```
|  LoyaltyProgram  |
```

```
+-----+
```

```
| - loyaltyID: int     |
| - userID: int        |
| - points: int        |
| - rewards: String    |
```

```
+-----+
```



```
+-----+
| Recommendation |
+-----+
| - recommendationID: int |
| - userID: int          |
| - restaurantID: int   |
| - description: String |
+-----+
```

```
+-----+
| Account |
+-----+
| - accountID: int |
| - userID: int    |
| - username: String |
| - password: String |
+-----+
```

```
+-----+
| Notification |
+-----+
| - notificationID: int |
| - userID: int        |
| - orderID: int       |
```

```

| - message: String      |
| - timestamp: DateTime |
+-----+

+-----+

|   DeliveryPerson      |
+-----+

| - deliveryPersonID: int |
| - name: String         |
| - phone: String        |
+-----+

| + updateOrderStatus() |
+-----+

+-----+

| SystemAdministrator   |
+-----+

| - adminID: int        |
| - name: String        |
| - email: String       |
+-----+

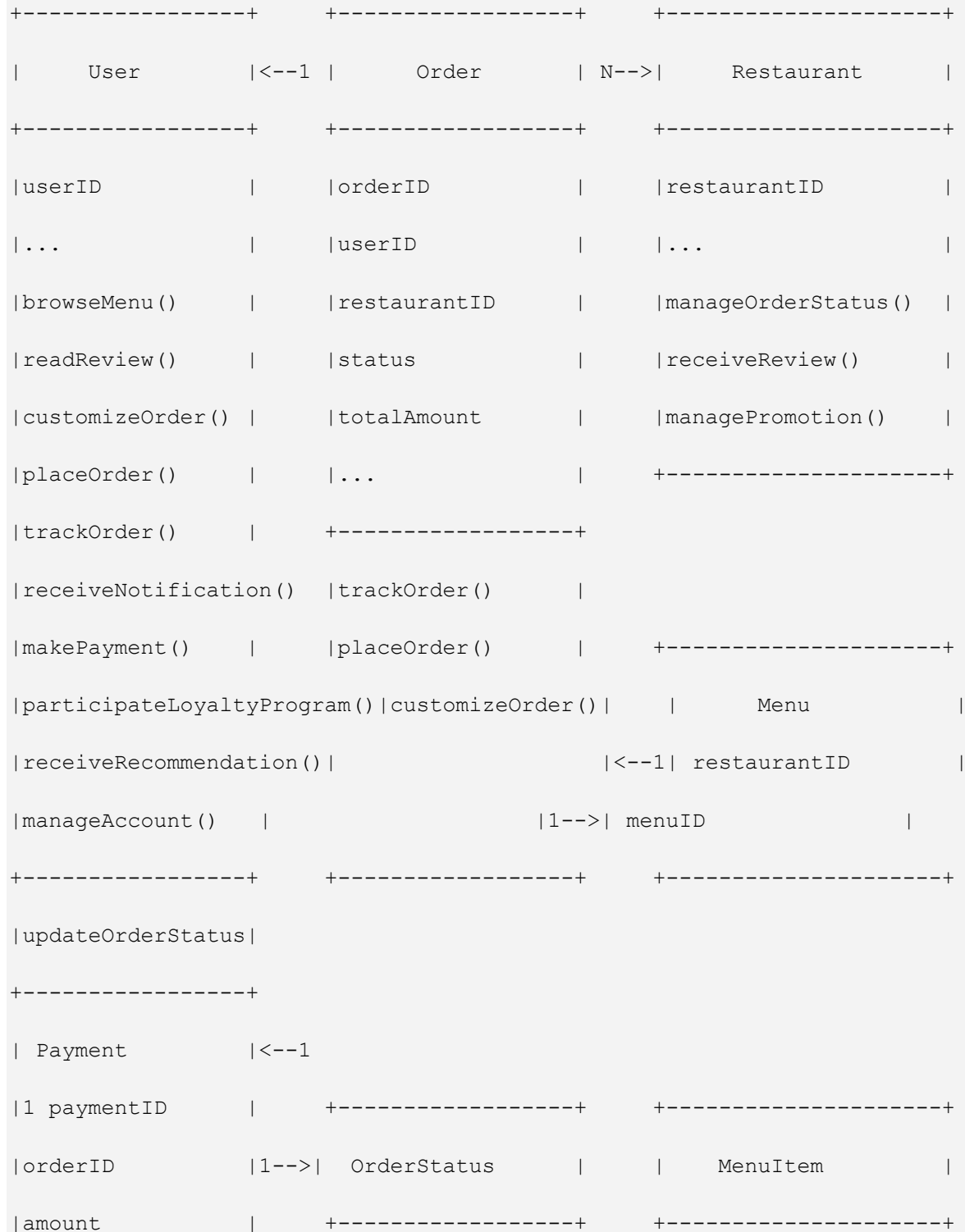
| + administerSystem()  |
+-----+

...

```

Relationships Diagram

```plaintext



```

|method | |statusID | |menuID |
|timestamp | |orderID | |itemID |
+-----+ |status | |name |
 |timestamp | |description |
 +-----+ |price |
 +-----+
...

```

### ### Explanation:

- **User** interacts with the system by browsing menus, reading reviews, placing and tracking orders, customizing orders, making payments, participating in loyalty programs, receiving personalized recommendations, managing their account, and receiving notifications.
- **Restaurant** manages its menu and menu items, receives reviews, and manages order statuses and promotions.
- **Order** and **OrderStatus** capture the details and status of user orders.
- **Payment** handles the payment transactions for orders.
- **Promotion** manages promotional offers from restaurants.
- **LoyaltyProgram** keeps track of users' loyalty points and rewards.
- **Recommendation** provides personalized suggestions to users.
- **Account** stores user account details.
- **Notification** keeps users informed about their order status.
- **DeliveryPerson** updates the order status during delivery.
- **SystemAdministrator** manages the overall system.

### User

Identify system operations from use case specifications.

### ChatGPT

### ### System Operations Identified from Use Case Specifications

#### #### 1. Browse Menus

- **Operation**: `displayAvailableMenus()`
  - **Description**: Fetch and display a list of available restaurant menus.
  - **Preconditions**: User must be logged into the system.
  - **Postconditions**: List of menus is displayed to the user.

#### #### 2. Read Reviews

- **Operation**: `displayReviews(restaurantID, dishID)`
  - **Description**: Fetch and display reviews for a specific restaurant or dish.
  - **Preconditions**: User must be logged into the system.
  - **Postconditions**: Reviews are displayed to the user.

#### #### 3. Customize Order

- **Operation**: `displayCustomizationOptions(itemID)`
  - **Description**: Fetch and display customization options for a selected menu item.
  - **Preconditions**: User must be browsing a restaurant menu.
  - **Postconditions**: Customization options are displayed to the user.
- **Operation**: `saveCustomizedOrder(userID, orderDetails)`
  - **Description**: Save the customized order details.
  - **Preconditions**: User has selected customization options.
  - **Postconditions**: Customized order is saved.

#### #### 4. Place Order

- **Operation**: `createOrder(userID, orderDetails)`
  - **Description**: Create and place a new order for the user.
  - **Preconditions**: User must have items in the cart.
  - **Postconditions**: Order is placed, and confirmation is sent to the user.
- **Operation**: `confirmOrderPayment(orderID, paymentDetails)`
  - **Description**: Process the payment and confirm the order.
  - **Preconditions**: User has selected a payment method and confirmed the order.
  - **Postconditions**: Payment is processed, and order is confirmed.

#### #### 5. Track Order

- **Operation**: `fetchOrderStatus(orderID)`
  - **Description**: Fetch the current status of a specific order.
  - **Preconditions**: Order must be placed and in progress.
  - **Postconditions**: Current order status is displayed to the user.
- **Operation**: `updateOrderStatus(orderID, status)`
  - **Description**: Update the status of a specific order.
  - **Preconditions**: Order must be in progress.
  - **Postconditions**: Order status is updated.

#### #### 6. Receive Notifications

```
- **Operation**: `sendOrderNotification(userID, orderID, message)`

- **Description**: Send a notification about the order status to the user.

- **Preconditions**: Order must be placed and in progress.

- **Postconditions**: User receives notification.
```

#### #### 7. Make Payment

```
- **Operation**: `processPayment(orderID, paymentDetails)`

- **Description**: Process the payment for a specific order.

- **Preconditions**: User has selected a payment method and confirmed the order.

- **Postconditions**: Payment is processed, and order is confirmed.
```

#### #### 8. Manage Promotions

```
- **Operation**: `createPromotion(promotionDetails)`

- **Description**: Create a new promotional offer.

- **Preconditions**: User must have the necessary permissions (administrator or restaurant owner).

- **Postconditions**: New promotion is created.

- **Operation**: `updatePromotion(promotionID, promotionDetails)`

- **Description**: Update an existing promotional offer.

- **Preconditions**: User must have the necessary permissions (administrator or restaurant owner).

- **Postconditions**: Promotion is updated.

- **Operation**: `deletePromotion(promotionID)`
```

- **Description**: Delete an existing promotional offer.
- **Preconditions**: User must have the necessary permissions (administrator or restaurant owner).
- **Postconditions**: Promotion is deleted.

#### ### 9. Participate in Loyalty Program

- **Operation**: `enrollInLoyaltyProgram(userID)`
  - **Description**: Enroll a user in the loyalty program.
  - **Preconditions**: User must be logged into the system.
  - **Postconditions**: User is enrolled in the loyalty program.
- **Operation**: `updateLoyaltyPoints(userID, points)`
  - **Description**: Update the loyalty points of a user.
  - **Preconditions**: User must be enrolled in the loyalty program.
  - **Postconditions**: User's loyalty points are updated.

#### ### 10. Receive Personalized Recommendations

- **Operation**: `generateRecommendations(userID)`
  - **Description**: Generate personalized recommendations for a user.
  - **Preconditions**: User must have a history of orders and preferences.
  - **Postconditions**: Personalized recommendations are displayed to the user.

#### ### 11. Manage Account

- **Operation**: `updateAccountDetails(userID, accountDetails)`
  - **Description**: Update the account details of a user.



- **\*\*Preconditions\*\***: User must be logged into the system.
- **\*\*Postconditions\*\***: User's account details are updated.

#### #### 12. Update Order Status

- **\*\*Operation\*\***: `changeOrderStatus(orderID, status)`
  - **\*\*Description\*\***: Change the status of a specific order.
  - **\*\*Preconditions\*\***: Order must be in progress.
  - **\*\*Postconditions\*\***: Order status is updated.

#### #### 13. Manage Restaurant Menu

- **\*\*Operation\*\***: `addItem(restaurantID, menuItemDetails)`
  - **\*\*Description\*\***: Add a new item to the restaurant's menu.
  - **\*\*Preconditions\*\***: Restaurant must be logged into the system.
  - **\*\*Postconditions\*\***: New menu item is added.
- **\*\*Operation\*\***: `updateMenuItem(itemID, menuItemDetails)`
  - **\*\*Description\*\***: Update an existing item in the restaurant's menu.
  - **\*\*Preconditions\*\***: Restaurant must be logged into the system.
  - **\*\*Postconditions\*\***: Menu item is updated.
- **\*\*Operation\*\***: `deleteMenuItem(itemID)`
  - **\*\*Description\*\***: Delete an item from the restaurant's menu.
  - **\*\*Preconditions\*\***: Restaurant must be logged into the system.
  - **\*\*Postconditions\*\***: Menu item is deleted.

#### #### 14. Administer System

- **Operation**: `manageUserAccounts(adminID, userAccountDetails)`
  - **Description**: Manage user accounts in the system.
  - **Preconditions**: Administrator must be logged into the system.
  - **Postconditions**: User accounts are managed.
  
- **Operation**: `manageSecuritySettings(adminID, securitySettings)`
  - **Description**: Manage security settings of the system.
  - **Preconditions**: Administrator must be logged into the system.
  - **Postconditions**: Security settings are updated.
  
- **Operation**: `monitorSystemPerformance(adminID)`
  - **Description**: Monitor the performance of the system.
  - **Preconditions**: Administrator must be logged into the system.
  - **Postconditions**: System performance is monitored and maintained.

These system operations ensure the functional requirements of the food order and delivery system are met efficiently and effectively.

#### **User**

Create design sequence diagrams for system operations.

#### **ChatGPT**

#### ### Design Sequence Diagrams for System Operations

#### #### 1. Browse Menus

```
Operation: `displayAvailableMenus()``
```

```
```plaintext
```

```
User -> System: select 'Browse Menus'
```

```
System -> Menu: getAvailableMenus()
```

```
Menu -> System: return available menus
```

```
System -> User: display available menus
```

```
```
```

#### #### 2. Read Reviews

```
Operation: `displayReviews(restaurantID, dishID)``
```

```
```plaintext
```

```
User -> System: select 'Read Reviews'
```

```
System -> Review: getReviews(restaurantID, dishID)
```

```
Review -> System: return reviews
```

```
System -> User: display reviews
```

```
```
```

#### #### 3. Customize Order

```
Operation: `displayCustomizationOptions(itemID)``
```

```
```plaintext
```

```
User -> System: select 'Customize Order'

System -> MenuItem: getCustomizationOptions(itemID)

MenuItem -> System: return customization options

System -> User: display customization options

...

**Operation**: `saveCustomizedOrder(userID, orderDetails)`
```

```
```plaintext
```

```
User -> System: save customized order

System -> Order: saveOrderDetails(userID, orderDetails)

Order -> System: return confirmation

System -> User: display order confirmation

...
```

```
4. Place Order
```

```
Operation: `createOrder(userID, orderDetails)`

```plaintext

User -> System: place order

System -> Order: createOrder(userID, orderDetails)

Order -> System: return orderID and details

System -> User: display order summary and payment options

...
```

```
**Operation**: `confirmOrderPayment(orderID, paymentDetails)`
```

```
```plaintext
```

```
User -> System: confirm payment
```

```
System -> PaymentGateway: processPayment(orderID, paymentDetails)
```

```
PaymentGateway -> System: return payment confirmation
```

```
System -> Order: updateOrderStatus(orderID, 'Confirmed')
```

```
Order -> System: return updated status
```

```
System -> User: display order confirmation
```

```
...
```

```
5. Track Order
```

```
Operation: `fetchOrderStatus(orderID)`
```

```
```plaintext
```

```
User -> System: track order
```

```
System -> OrderStatus: getOrderStatus(orderID)
```

```
OrderStatus -> System: return current status
```

```
System -> User: display current order status
```

```
...
```

```
**Operation**: `updateOrderStatus(orderID, status)`
```

```
```plaintext
```

```
DeliveryPerson -> System: update order status
```

```
System -> OrderStatus: updateStatus(orderID, status)
```

```
OrderStatus -> System: return updated status
```

```
System -> User: send notification with updated status
```

```
```
```

6. Receive Notifications

```
**Operation**: `sendOrderNotification(userID, orderID, message)`
```

```
```plaintext
```

```
System -> Notification: createNotification(userID, orderID, message)
```

```
Notification -> System: return notification confirmation
```

```
System -> User: send notification
```

```
```
```

7. Make Payment

```
**Operation**: `processPayment(orderID, paymentDetails)`
```

```
```plaintext
```

```
User -> System: make payment
```

```
System -> PaymentGateway: processPayment(orderID, paymentDetails)
```

```
PaymentGateway -> System: return payment confirmation
```

System -> Order: updateOrderStatus(orderID, 'Paid')

Order -> System: return updated status

System -> User: display payment confirmation

...

#### #### 8. Manage Promotions

**\*\*Operation\*\*:** `createPromotion(promotionDetails)`

```plaintext

Admin/Restaurant -> System: create promotion

System -> Promotion: addPromotion(promotionDetails)

Promotion -> System: return promotion confirmation

System -> Admin/Restaurant: display promotion confirmation

...

****Operation**:** `updatePromotion(promotionID, promotionDetails)`

```plaintext

Admin/Restaurant -> System: update promotion

System -> Promotion: updatePromotion(promotionID, promotionDetails)

Promotion -> System: return update confirmation

System -> Admin/Restaurant: display update confirmation

...

**\*\*Operation\*\*:** `deletePromotion(promotionID)`

```plaintext

Admin/Restaurant -> System: delete promotion

System -> Promotion: removePromotion(promotionID)

Promotion -> System: return deletion confirmation

System -> Admin/Restaurant: display deletion confirmation

...

9. Participate in Loyalty Program

****Operation**:** `enrollInLoyaltyProgram(userID)`

```plaintext

User -> System: enroll in loyalty program

System -> LoyaltyProgram: addUserToProgram(userID)

LoyaltyProgram -> System: return enrollment confirmation

System -> User: display enrollment confirmation

...

**\*\*Operation\*\*:** `updateLoyaltyPoints(userID, points)`

```plaintext

System -> LoyaltyProgram: updatePoints(userID, points)

LoyaltyProgram -> System: return points update confirmation

System -> User: display updated points

...

10. Receive Personalized Recommendations

****Operation**:** `generateRecommendations(userID)`

```plaintext

User -> System: request recommendations

System -> Recommendation: getRecommendations(userID)

Recommendation -> System: return recommendations

System -> User: display recommendations

...

#### #### 11. Manage Account

**\*\*Operation\*\*:** `updateAccountDetails(userID, accountDetails)`

```plaintext

User -> System: update account details

System -> Account: updateAccount(userID, accountDetails)

Account -> System: return update confirmation

System -> User: display update confirmation

...

12. Update Order Status

****Operation**:** `changeOrderStatus(orderID, status)`

```plaintext

DeliveryPerson/Restaurant -> System: update order status

System -> OrderStatus: updateStatus(orderID, status)

OrderStatus -> System: return updated status

System -> User: send notification with updated status

```

13. Manage Restaurant Menu

****Operation**:** `addMenuItem(restaurantID, menuItemDetails)`

```plaintext

Restaurant -> System: add menu item

System -> Menu: addItem(restaurantID, menuItemDetails)

Menu -> System: return add confirmation

System -> Restaurant: display add confirmation

```

****Operation**:** `updateMenuItem(itemID, menuItemDetails)`

```plaintext

```
Restaurant -> System: update menu item

System -> MenuItem: updateItem(itemID, menuItemDetails)

MenuItem -> System: return update confirmation

System -> Restaurant: display update confirmation

...
```

```
Operation: `deleteMenuItem(itemID)`
```

```
```plaintext
```

```
Restaurant -> System: delete menu item

System -> MenuItem: removeItem(itemID)

MenuItem -> System: return deletion confirmation

System -> Restaurant: display deletion confirmation

...
```

```
#### 14. Administer System
```

```
**Operation**: `manageUserAccounts(adminID, userAccountDetails)`
```

```
```plaintext
```

```
Admin -> System: manage user accounts

System -> Account: updateUserAccount(adminID, userAccountDetails)

Account -> System: return update confirmation

System -> Admin: display update confirmation

...
```

```
Operation: `manageSecuritySettings(adminID, securitySettings)`
```

```
```plaintext
```

```
Admin -> System: manage security settings
```

```
System -> Security: updateSettings(adminID, securitySettings)
```

```
Security -> System: return update confirmation
```

```
System -> Admin: display update confirmation
```

```
...
```

```
**Operation**: `monitorSystemPerformance(adminID)`
```

```
```plaintext
```

```
Admin -> System: monitor system performance
```

```
System -> PerformanceMonitor: getSystemPerformance()
```

```
PerformanceMonitor -> System: return performance data
```

```
System -> Admin: display performance data
```

```
...
```

These sequence diagrams illustrate the interactions between users, the system, and other entities to achieve the various system operations.

### **User**

Create design class diagrams based on the domain model and sequence diagrams.

### **ChatGPT**

### Design Class Diagrams for Food Order and Delivery System

Based on the domain model and sequence diagrams, the design class diagrams will include key classes, their attributes, methods, and relationships.

#### #### Class Diagram for User and Related Operations

```plaintext

+-----+

| User |

+-----+

| - userID: int |

| - name: String |

| - email: String |

| - address: String |

| - phone: String |

+-----+

| + browseMenus() |

| + readReviews() |

| + customizeOrder() |

| + placeOrder() |

| + trackOrder() |

| + receiveNotifications() |

| + makePayment() |

| + participateLoyaltyProgram() |

| + receiveRecommendations() |

```
| + manageAccount() |
+-----+

+-----+

|      Account      |
+-----+

| - accountID: int   |
| - userID: int      |
| - username: String |
| - password: String |
+-----+

| + updateAccountDetails() |
+-----+

+-----+

|    LoyaltyProgram    |
+-----+

| - loyaltyID: int      |
| - userID: int         |
| - points: int         |
| - rewards: String     |
+-----+

| + enrollUser()       |
| + updatePoints()     |
+-----+
```

```

+-----+
| Recommendation |
+-----+

| - recommendationID: int |
| - userID: int          |
| - restaurantID: int   |
| - description: String |
+-----+

| + generateRecommendations() |
+-----+

+-----+

+-----+
| Notification |
+-----+

| - notificationID: int |
| - userID: int        |
| - orderID: int       |
| - message: String    |
| - timestamp: DateTime|
+-----+

| + sendNotification() |
+-----+

...

```

Class Diagram for Restaurant and Menu Management

```plaintext

+-----+

|        Restaurant        |

+-----+

| - restaurantID: int    |

| - name: String        |

| - address: String     |

| - phone: String       |

| - email: String       |

+-----+

| + manageOrderStatus() |

| + receiveReview()     |

| + managePromotion()   |

| + manageMenu()        |

+-----+

+-----+

|        Menu        |

+-----+

| - menuID: int        |

| - restaurantID: int   |

+-----+

| + getAvailableMenus() |



```

| + addItem() |
| + updateItem() |
| + removeItem() |
+-----+

+-----+

| MenuItem |
+-----+

| - itemID: int |
| - menuID: int |
| - name: String |
| - description: String |
| - price: double |
+-----+

| + getCustomizationOptions() |
| + saveCustomizedOrder() |
+-----+

+-----+

| Promotion |
+-----+

| - promotionID: int |
| - restaurantID: int |
| - description: String |
| - discount: double |

```

```

| - startDate: DateTime|
| - endDate: DateTime |
+-----+
| + createPromotion() |
| + updatePromotion() |
| + deletePromotion() |
+-----+

+-----+

| Review |
+-----+
| - reviewID: int |
| - userID: int |
| - restaurantID: int|
| - rating: int |
| - comment: String |
| - timestamp: DateTime|
+-----+
| + getReviews() |
+-----+

...

```

#### Class Diagram for Order and Payment Management

```plaintext

```
+-----+
|      Order      |
+-----+
| - orderID: int   |
| - userID: int    |
| - restaurantID: int |
| - status: String |
| - totalAmount: double|
+-----+
| + createOrder()   |
| + updateOrderStatus() |
| + getOrderStatus() |
+-----+

+-----+
|      OrderStatus  |
+-----+
| - statusID: int    |
| - orderID: int     |
| - status: String   |
| - timestamp: DateTime|
+-----+
| + updateStatus()   |
| + getOrderStatus() |
+-----+
```

```

+-----+
|      Payment      |
+-----+

| - paymentID: int   |
| - orderID: int     |
| - amount: double   |
| - method: String    |
| - timestamp: DateTime |
+-----+

| + processPayment() |
+-----+

+-----+

|  DeliveryPerson  |
+-----+

| - deliveryPersonID: int |
| - name: String          |
| - phone: String          |
+-----+

| + updateOrderStatus() |
+-----+

...

```

Class Diagram for System Administration

```plaintext

+-----+

| SystemAdministrator |

+-----+

| - adminID: int |

| - name: String |

| - email: String |

+-----+

| + manageUserAccounts() |

| + manageSecuritySettings() |

| + monitorSystemPerformance() |

+-----+

+-----+

| PerformanceMonitor |

+-----+

| - monitorID: int |

| - performanceData: String |

+-----+

| + getSystemPerformance() |

+-----+

+-----+

| Security |

```

+-----+
| - securityID: int |
| - settings: String |
+-----+

| + updateSettings() |
+-----+

...

```

### Explanation:

- **User Class**: Represents users who interact with the system. Methods include browsing menus, reading reviews, customizing orders, placing orders, tracking orders, receiving notifications, making payments, participating in the loyalty program, receiving recommendations, and managing account details.

- **Restaurant Class**: Represents restaurants that manage menus, receive reviews, and manage order statuses and promotions.

- **Menu and MenuItem Classes**: Represent the menu and individual items within a restaurant's menu, including operations for adding, updating, and removing items, as well as customizing orders.

- **Order and OrderStatus Classes**: Represent orders placed by users and their statuses, including methods for creating orders, updating statuses, and tracking orders.

- **Payment Class**: Represents payment transactions, including processing payments for orders.

- **Notification Class**: Represents notifications sent to users regarding their order status.
- **Promotion Class**: Represents promotional offers managed by restaurants.
- **Review Class**: Represents reviews left by users for restaurants and dishes.
- **LoyaltyProgram and Recommendation Classes**: Represent the loyalty program users participate in and the personalized recommendations they receive.
- **SystemAdministrator Class**: Represents system administrators who manage user accounts, security settings, and monitor system performance.
- **PerformanceMonitor and Security Classes**: Represent system performance monitoring and security settings management.

This design class diagram captures the essential entities and their interactions, providing a detailed view of the system's structure and operations.

## User

Develop a Java implementation for the system as specified in the class diagram and sequence diagrams.

## ChatGPT

### Java Implementation for the Food Order and Delivery System

Below is a simplified Java implementation of the core classes and methods based on the class and sequence diagrams provided. This implementation captures the essential functionalities of the system.

```
User.java
```

```
```java
```

```
public class User {
```

```
    private int userID;
```

```
    private String name;
```

```
    private String email;
```

```
    private String address;
```

```
    private String phone;
```

```
    // Constructors, getters, and setters
```

```
    public void browseMenus() {
```

```
        // Implementation to browse menus
```

```
    }
```

```
    public void readReviews(int restaurantID, int dishID) {
```

```
        // Implementation to read reviews
```

```
    }
```

```
    public void customizeOrder(int itemID, OrderDetails orderDetails) {
```

```
        // Implementation to customize order
```

```
    }
```

```
    public void placeOrder(OrderDetails orderDetails) {
```

```
        // Implementation to place order
```



```
}

public void trackOrder(int orderID) {

    // Implementation to track order

}

public void receiveNotifications(int userID, int orderID, String message)
{

    // Implementation to receive notifications

}

public void makePayment(int orderID, PaymentDetails paymentDetails) {

    // Implementation to make payment

}

public void participateLoyaltyProgram() {

    // Implementation to participate in loyalty program

}

public void receiveRecommendations(int userID) {

    // Implementation to receive recommendations

}

public void manageAccount(AccountDetails accountDetails) {

    // Implementation to manage account

}
```

```
    }  
  
}  
  
````  

Restaurant.java

```java  
public class Restaurant {  
  
    private int restaurantID;  
  
    private String name;  
  
    private String address;  
  
    private String phone;  
  
    private String email;  
  
  
    // Constructors, getters, and setters  
  
  
    public void manageOrderStatus(int orderID, String status) {  
  
        // Implementation to manage order status  
  
    }  
  
  
    public void receiveReview(Review review) {  
  
        // Implementation to receive review  
  
    }  
  
  
    public void managePromotion(Promotion promotion) {  
  
        // Implementation to manage promotion
```

```

    }

    public void manageMenu(MenuItem menuItem, String action) {

        // Implementation to manage menu

        switch (action) {

            case "add":

                // Add menu item

                break;

            case "update":

                // Update menu item

                break;

            case "delete":

                // Delete menu item

                break;

        }

    }

}

...

```

Menu.java

```
```java
```

```
import java.util.List;
```

```
public class Menu {
```

```
 private int menuID;
```

```
private int restaurantID;

private List<MenuItem> items;

// Constructors, getters, and setters

public List<MenuItem> getAvailableMenus() {

 // Implementation to get available menus

 return items;

}

public void addItem(MenuItem item) {

 // Implementation to add menu item

 items.add(item);

}

public void updateItem(MenuItem item) {

 // Implementation to update menu item

 // Find and update item in the list

}

public void removeItem(int itemID) {

 // Implementation to remove menu item

 // Find and remove item from the list

}

}
```

```
...
```

```
MenuItem.java
```

```
```java
```

```
public class MenuItem {  
  
    private int itemID;  
  
    private int menuID;  
  
    private String name;  
  
    private String description;  
  
    private double price;  
  
  
    // Constructors, getters, and setters  
  
    public void getCustomizationOptions(int itemID) {  
  
        // Implementation to get customization options  
  
    }  
  
  
    public void saveCustomizedOrder(int userID, OrderDetails orderDetails) {  
  
        // Implementation to save customized order  
  
    }  
}
```

```
...
```

```
#### Order.java
```

```
```java
```

```
public class Order {

 private int orderID;

 private int userID;

 private int restaurantID;

 private String status;

 private double totalAmount;

 // Constructors, getters, and setters

 public void createOrder(int userID, OrderDetails orderDetails) {

 // Implementation to create order

 }

 public void updateOrderStatus(int orderID, String status) {

 // Implementation to update order status

 }

 public String getOrderStatus(int orderID) {

 // Implementation to get order status

 return status;

 }

}

...

```

#### OrderStatus.java

```
```java

import java.util.Date;

public class OrderStatus {

    private int statusID;

    private int orderID;

    private String status;

    private Date timestamp;

    // Constructors, getters, and setters

    public void updateStatus(int orderID, String status) {

        // Implementation to update status

    }

    public String getOrderStatus(int orderID) {

        // Implementation to get order status

        return status;

    }

}

```
```

#### Payment.java

```
```java

import java.util.Date;
```

```

public class Payment {

    private int paymentID;

    private int orderID;

    private double amount;

    private String method;

    private Date timestamp;


    // Constructors, getters, and setters


    public void processPayment(int orderID, PaymentDetails paymentDetails) {

        // Implementation to process payment

    }

}

...

```

Notification.java

```

```java

```

```

import java.util.Date;

```

```

public class Notification {

 private int notificationID;

 private int userID;

 private int orderID;

 private String message;

```



```

private Date timestamp;

// Constructors, getters, and setters

public void sendNotification(int userID, int orderID, String message) {
 // Implementation to send notification
}
}
...

Promotion.java
```java
import java.util.Date;

public class Promotion {
    private int promotionID;
    private int restaurantID;
    private String description;
    private double discount;
    private Date startDate;
    private Date endDate;

    // Constructors, getters, and setters

    public void createPromotion(Promotion promotion) {

```

```
        // Implementation to create promotion

    }

    public void updatePromotion(int promotionID, Promotion promotion) {

        // Implementation to update promotion

    }

    public void deletePromotion(int promotionID) {

        // Implementation to delete promotion

    }

}

...

```

Review.java

```
```java
```

```
import java.util.Date;
```

```
public class Review {

 private int reviewID;

 private int userID;

 private int restaurantID;

 private int rating;

 private String comment;

 private Date timestamp;
}
```

```

// Constructors, getters, and setters

public void getReviews(int restaurantID, int dishID) {

 // Implementation to get reviews

}

}

...

LoyaltyProgram.java

```java
public class LoyaltyProgram {

    private int loyaltyID;

    private int userID;

    private int points;

    private String rewards;

    // Constructors, getters, and setters

    public void enrollUser(int userID) {

        // Implementation to enroll user in loyalty program

    }

    public void updatePoints(int userID, int points) {

        // Implementation to update loyalty points

    }

```

```

}

...

#### Recommendation.java

```java
public class Recommendation {

 private int recommendationID;

 private int userID;

 private int restaurantID;

 private String description;

 // Constructors, getters, and setters

 public void generateRecommendations(int userID) {

 // Implementation to generate recommendations

 }

}

...

DeliveryPerson.java

```java
public class DeliveryPerson {

    private int deliveryPersonID;

    private String name;

    private String phone;

```

```

        // Constructors, getters, and setters

        public void updateOrderStatus(int orderID, String status) {

            // Implementation to update order status

        }
    }
}
```

SystemAdministrator.java

```java
public class SystemAdministrator {

    private int adminID;

    private String name;

    private String email;

    // Constructors, getters, and setters

    public void manageUserAccounts(int userID, AccountDetails accountDetails)
    {

        // Implementation to manage user accounts

    }

    public void manageSecuritySettings(SecuritySettings securitySettings) {

        // Implementation to manage security settings
    }
}

```

```

    }

    public void monitorSystemPerformance() {

        // Implementation to monitor system performance

    }

}

```

```

#### PerformanceMonitor.java

```

```java

public class PerformanceMonitor {

    private int monitorID;

    private String performanceData;

    // Constructors, getters, and setters

    public void getSystemPerformance() {

        // Implementation to get system performance

    }

}

```

```

#### Security.java

```

```java

public class Security {

```

```

private int securityID;

private String settings;

// Constructors, getters, and setters

public void updateSettings(SecuritySettings securitySettings) {

    // Implementation to update security settings

}

}

...

```

Explanation

- Each class represents a key entity in the system, with attributes and methods based on the use case specifications.
- Relationships and interactions between classes are captured through method calls and data exchange.
- Some classes (e.g., `User`, `Restaurant`, `Order`, etc.) have methods corresponding to their operations identified in the sequence diagrams.
- Simplified constructors, getters, and setters are included to manage the attributes of each class.

This implementation provides a foundation for the food order and delivery system, enabling further development and integration as needed.

User

Develop tests including unit tests, integration tests, and system tests for the implementation.

ChatGPT

Testing the Food Order and Delivery System

Below are examples of unit tests, integration tests, and system tests for the Java implementation of the food order and delivery system. These tests will be implemented using the JUnit framework.

Unit Tests

UserTest.java

```
```java

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

public class UserTest {

 private User user;

 @BeforeEach

 public void setUp() {

 user = new User(1, "John Doe", "john@example.com", "123 Main St",
"555-1234");

 }

 @Test

 public void testBrowseMenus() {

 // Assuming browseMenus() returns a list of menus

 List<Menu> menus = user.browseMenus();
```



```
 assertNotNull(menus);

 assertTrue(menus.size() > 0);
 }

 @Test

 public void testPlaceOrder() {

 OrderDetails orderDetails = new OrderDetails();

 user.placeOrder(orderDetails);

 // Verify order is placed

 // Assuming getOrderStatus() returns the status of the last order
placed

 String status = user.trackOrder(orderDetails.getOrderID());

 assertEquals("Placed", status);
 }

 @Test

 public void testMakePayment() {

 PaymentDetails paymentDetails = new PaymentDetails();

 user.makePayment(1, paymentDetails);

 // Verify payment is processed

 // Assuming getOrderStatus() returns the status of the last order
placed

 String status = user.trackOrder(1);

 assertEquals("Paid", status);
 }
}
```

```
```\n\n##### RestaurantTest.java\n\n```java\nimport static org.junit.jupiter.api.Assertions.*;\n\nimport org.junit.jupiter.api.BeforeEach;\n\nimport org.junit.jupiter.api.Test;\n\npublic class RestaurantTest {\n\n    private Restaurant restaurant;\n\n    @BeforeEach\n\n    public void setUp() {\n\n        restaurant = new Restaurant(1, "Pizza Place", "456 Elm St", "555-5678", "info@pizzaplace.com");\n\n    }\n\n    @Test\n\n    public void testManageMenu() {\n\n        MenuItem menuItem = new MenuItem(1, "Margherita", "Classic pizza", 8.99);\n\n        restaurant.manageMenu(menuItem, "add");\n\n        // Verify menu item is added\n\n        Menu menu = restaurant.getMenu();\n\n        assertTrue(menu.getItems().contains(menuItem));\n\n    }\n}
```

```

@Test

public void testManagePromotion() {

    Promotion promotion = new Promotion(1, "Discount", 10.0, new Date(),
new Date());

    restaurant.managePromotion(promotion);

    // Verify promotion is added

    assertEquals(promotion, restaurant.getPromotion(1));

}

}

...

```

OrderTest.java

```

```java

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

public class OrderTest {

 private Order order;

 @BeforeEach

 public void setUp() {

 order = new Order(1, 1, 1, "Placed", 20.0);

 }

```

```

@Test

public void testCreateOrder() {

 OrderDetails orderDetails = new OrderDetails();

 order.createOrder(1, orderDetails);

 // Verify order is created

 assertNotNull(order);

 assertEquals(1, order.getUserID());

}

@Test

public void testUpdateOrderStatus() {

 order.updateOrderStatus("Preparing");

 // Verify order status is updated

 assertEquals("Preparing", order.getStatus());

}

}

...

Integration Tests

UserRestaurantIntegrationTest.java

```java

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.BeforeEach;

```

```
import org.junit.jupiter.api.Test;

public class UserRestaurantIntegrationTest {

    private User user;

    private Restaurant restaurant;

    @BeforeEach

    public void setUp() {

        user = new User(1, "John Doe", "john@example.com", "123 Main St",
"555-1234");

        restaurant = new Restaurant(1, "Pizza Place", "456 Elm St", "555-
5678", "info@pizzaplace.com");

    }

    @Test

    public void testPlaceOrderAndUpdateStatus() {

        MenuItem menuItem = new MenuItem(1, "Margherita", "Classic pizza",
8.99);

        restaurant.manageMenu(menuItem, "add");

        OrderDetails orderDetails = new OrderDetails();

        user.placeOrder(orderDetails);

        Order order = new Order(orderDetails.getOrderID(), 1, 1, "Placed",
20.0);

        restaurant.manageOrderStatus(order.getOrderID(), "Preparing");

        // Verify order status is updated
    }
}
```

```
        String status = user.trackOrder(order.getOrderID());

        assertEquals("Preparing", status);
    }
}
...

```

PaymentIntegrationTest.java

```
```java

```

```
import static org.junit.jupiter.api.Assertions.*;

```

```
import org.junit.jupiter.api.BeforeEach;

```

```
import org.junit.jupiter.api.Test;

```

```
public class PaymentIntegrationTest {

```

```
 private User user;

```

```
 private Order order;

```

```
 @BeforeEach

```

```
 public void setUp() {

```

```
 user = new User(1, "John Doe", "john@example.com", "123 Main St",
"555-1234");

```

```
 order = new Order(1, 1, 1, "Placed", 20.0);

```

```
 }

```

```
 @Test

```

```
 public void testPlaceOrderAndMakePayment() {

```

```

 OrderDetails orderDetails = new OrderDetails();

 user.placeOrder(orderDetails);

 PaymentDetails paymentDetails = new PaymentDetails();

 user.makePayment(orderDetails.getOrderID(), paymentDetails);

 // Verify payment is processed

 String status = user.trackOrder(orderDetails.getOrderID());

 assertEquals("Paid", status);
 }
}
...

```

#### System Tests

##### SystemTest.java

```java

```
import static org.junit.jupiter.api.Assertions.*;
```

```
import org.junit.jupiter.api.BeforeEach;
```

```
import org.junit.jupiter.api.Test;
```

```
public class SystemTest {
```

```
    private User user;
```

```
    private Restaurant restaurant;
```

```
    private SystemAdministrator admin;
```

```
@BeforeEach

public void setUp() {

    user = new User(1, "John Doe", "john@example.com", "123 Main St",
"555-1234");

    restaurant = new Restaurant(1, "Pizza Place", "456 Elm St", "555-
5678", "info@pizzaplace.com");

    admin = new SystemAdministrator(1, "Admin", "admin@system.com");

}

@Test

public void testFullOrderProcess() {

    // Admin sets up restaurant and menu

    MenuItem menuItem = new MenuItem(1, "Margherita", "Classic pizza",
8.99);

    restaurant.manageMenu(menuItem, "add");

    // User places an order

    OrderDetails orderDetails = new OrderDetails();

    user.placeOrder(orderDetails);

    // Restaurant updates order status

    Order order = new Order(orderDetails.getOrderID(), 1, 1, "Placed",
20.0);

    restaurant.manageOrderStatus(order.getOrderID(), "Preparing");

    // User tracks order

    String status = user.trackOrder(order.getOrderID());
```



```

    assertEquals("Preparing", status);

    // User makes payment

    PaymentDetails paymentDetails = new PaymentDetails();

    user.makePayment(order.getOrderID(), paymentDetails);

    // Verify payment is processed and order status is updated

    status = user.trackOrder(order.getOrderID());

    assertEquals("Paid", status);

    // Restaurant updates order status to delivered

    restaurant.manageOrderStatus(order.getOrderID(), "Delivered");

    status = user.trackOrder(order.getOrderID());

    assertEquals("Delivered", status);

}

}

...

### Explanation

- Unit Tests: These tests focus on individual classes and their methods to ensure they work as expected in isolation.

- Integration Tests: These tests verify the interactions between different classes and components, ensuring they work together correctly.

- System Tests: These tests cover end-to-end scenarios, verifying that the entire system functions as expected from a user perspective.

```

Each test class includes setup methods to initialize objects and test methods to verify various functionalities. The tests use assertions to check that the expected outcomes match the actual results.