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.

AI Model

I'd be happy to identify the functional and non-functional requirements from the food order and delivery system description. Let's break them down:

Functional Requirements:

1. Menu browsing: Users can explore menus from various restaurants.

2. Order placement: Users can select items and place orders.

3. Order customization: Users can modify their orders according to preferences.

4. Review system: Users can read reviews of restaurants and dishes.

5. Real-time order tracking: Users can track the status of their orders in real-time.

6. Notifications: The system sends updates about order status to users.

7. Multiple payment options: Integration with various payment gateways.

8. Promotional features: The system supports discounts, offers, etc.

9. Loyalty program: A system to reward repeat customers.

10. Personalized recommendations: The system suggests items based on user preferences and order history.

Non-Functional Requirements:

1. Accessibility: The system should be easy to use for all users.

2. Multi-platform support: The system should work on both web and mobile platforms.

3. User-friendly interface: The system should have an easy-to-navigate interface.

4. Performance: The system should provide real-time updates and estimated delivery times.

5. Security: Secure payment processing and user data protection.

6. Scalability: The system should handle a wide range of cuisines and restaurants.

7. Reliability: The system should be available and functional at all times.

8. Efficiency: The system should provide a convenient and efficient ordering process.

Would you like me to elaborate on any of these requirements or discuss how they might be implemented?

User

Create a use case model for the system.

AI Model

Certainly. I'll create a use case model for the food order and delivery system. This model will outline the main actors and their interactions with the system. Here's a textual representation of the use case model:

Actors:

1. Customer

2. Restaurant Owner

3. Delivery Driver

4. System Administrator

Use Cases:

1. Register/Login

   Actor: Customer, Restaurant Owner, Delivery Driver

2. Browse Restaurants and Menus

   Actor: Customer

3. Place Order

   Actor: Customer

   Include: Customize Order

4. Process Payment

   Actor: Customer

5. Track Order

   Actor: Customer

6. Receive Notifications

   Actor: Customer, Restaurant Owner, Delivery Driver

7. Manage Restaurant Profile

   Actor: Restaurant Owner

8. Update Menu

   Actor: Restaurant Owner

9. Manage Orders

   Actor: Restaurant Owner

10. Accept/Reject Order

    Actor: Restaurant Owner

11. Update Order Status

    Actor: Restaurant Owner, Delivery Driver

12. Pick Up and Deliver Order

    Actor: Delivery Driver

13. Rate and Review

    Actor: Customer

14. Manage Promotions and Loyalty Program

    Actor: System Administrator

15. Generate Reports

    Actor: System Administrator

16. Provide Customer Support

    Actor: System Administrator

Relationships:

- "Place Order" includes "Customize Order"

- "Track Order" extends "Receive Notifications"

- "Manage Orders" includes "Accept/Reject Order" and "Update Order Status"

This use case model provides a high-level overview of the system's functionality and the interactions between actors and the system. It covers the main features described in the system description, including ordering, delivery, restaurant management, and administrative functions.

Would you like me to explain any specific use case in more detail or discuss how these use cases relate to the functional requirements we identified earlier?

User

Create use case specifications for identified use cases.

AI Model

I'll create use case specifications for some of the key use cases we identified. These specifications will provide more detailed information about each use case. I'll focus on three important use cases to give you a sense of how these specifications are structured.

1. Use Case: Place Order

Actor: Customer

Description: Customer selects items from a restaurant's menu and places an order.

Preconditions:

- Customer is logged in

- Customer has selected a restaurant

Main Flow:

1. Customer browses the restaurant's menu

2. Customer adds desired items to the cart

3. System displays the cart contents and total price

4. Customer reviews the order

5. Customer chooses delivery address

6. Customer selects payment method

7. System validates the order

8. Customer confirms the order

9. System processes the payment

10. System confirms the order and provides an order number

Alternative Flows:

4a. Customer modifies the order

   1. Customer adjusts quantities or removes items

   2. System updates the cart and total price

   3. Return to step 4

7a. Order validation fails

   1. System displays error message

   2. Customer corrects the issue

   3. Return to step 7

Post-conditions:

- Order is placed in the system

- Restaurant is notified of the new order

- Customer receives order confirmation

2. Use Case: Track Order

Actor: Customer

Description: Customer views the real-time status of their order.

Preconditions:

- Customer has placed an order

- Order is in progress

Main Flow:

1. Customer selects the option to track their order

2. System retrieves the current order status

3. System displays the order status, estimated delivery time, and delivery person's location (if applicable)

4. Customer views the tracking information

5. System provides options for the customer to contact support or the delivery person

Alternative Flows:

2a. Order status cannot be retrieved

1. System displays an error message

2. System provides customer support contact information

Post-conditions:

- Customer is informed of the current order status

3. Use Case: Manage Orders

Actor: Restaurant Owner

Description: Restaurant owner views and manages incoming orders.

Preconditions:

- Restaurant owner is logged in

- Restaurant is open for orders

Main Flow:

1. System displays a list of new and ongoing orders

2. Restaurant owner selects an order to view details

3. Restaurant owner accepts or rejects the order

4. If accepted, restaurant owner updates the order status as it's being prepared

5. When the order is ready, restaurant owner marks it for pickup/delivery

6. System notifies the customer and (if applicable) assigns a delivery driver

Alternative Flows:

3a. Restaurant owner rejects the order

1. Restaurant owner provides a reason for rejection
2. System cancels the order and notifies the customer
3. System initiates a refund process

4a. Restaurant owner needs to modify the order

1. Restaurant owner contacts the customer through the system
2. Customer approves or rejects the modification
3. Restaurant owner updates the order accordingly

Post-conditions:

- Order status is updated in the system
- Customer is notified of any changes to their order

These use case specifications provide a more detailed look at how specific functions of the system would work. They outline the steps involved, possible alternative scenarios, and the conditions that must be met before and after the use case is executed.

Would you like me to create specifications for any other use cases, or do you need any clarification on these?

User

Create a domain model based on use case specifications.

AI Model

Certainly. I'll create a domain model based on the use case specifications we've discussed. This domain model will represent the key concepts and their

relationships within the food order and delivery system. Here's a textual representation of the domain model:

Entities:

1. User

   Attributes:

   - userID

   - name

   - email

   - phoneNumber

   - address

2. Customer (subclass of User)

   Attributes:

   - orderHistory

   - loyaltyPoints

3. RestaurantOwner (subclass of User)

   Attributes:

   - restaurantID

4. DeliveryDriver (subclass of User)

   Attributes:

   - vehicleInfo

- currentLocation

5. Restaurant

   Attributes:

   - restaurantID

   - name

   - address

   - cuisine

   - rating

6. Menu

   Attributes:

   - menuID

   - restaurantID

7. MenuItem

   Attributes:

   - itemID

   - name

   - description

   - price

   - category

8. Order

   Attributes:

- orderID
- customerID
- restaurantID
- orderStatus
- totalPrice
- orderDate

9. OrderItem

   Attributes:

   - orderItemID
   - menuItemID
   - quantity
   - specialInstructions

10. Payment

    Attributes:

    - paymentID
    - orderID
    - amount
    - paymentMethod
    - paymentStatus

11. Review

    Attributes:

    - reviewID

    - customerID

    - restaurantID

    - rating

    - comment

    - date

12. Promotion

    Attributes:

    - promotionID

    - description

    - discountAmount

    - validFrom

    - validTo

Relationships:

- Customer places Order (1 to many)

- Order contains OrderItem (1 to many)

- OrderItem references MenuItem (many to 1)

- Restaurant has Menu (1 to 1)

- Menu contains MenuItem (1 to many)

- Restaurant is managed by RestaurantOwner (many to 1)

- Order is delivered by DeliveryDriver (many to 1)

- Order has Payment (1 to 1)

- Customer writes Review for Restaurant (many to many)

- Promotion applies to Order (many to many)

- Restaurant offers Promotion (many to many)

This domain model captures the main entities and their relationships based on the use cases we've discussed. It provides a foundation for understanding the data structure of the system and can be used as a starting point for database design.

Some key points about this model:

1. The User entity is the parent class for Customer, RestaurantOwner, and DeliveryDriver, allowing for shared attributes and polymorphism.

2. The Order entity is central, connecting customers, restaurants, and delivery drivers.

3. The Menu and MenuItem entities allow for flexible menu management by restaurants.

4. The Review entity enables the customer feedback system.

5. The Promotion entity allows for implementation of special offers and discounts.

Would you like me to elaborate on any part of this domain model or discuss how it relates to the use cases we specified earlier?

User

Identify system operations from use case specifications.

AI Model

Certainly. I'll identify the system operations based on the use case specifications we discussed earlier. These operations represent the key functionalities that the system needs to implement. Here's a list of system operations derived from our use cases:

1. From "Place Order" use case:

   - browseMenu()
   - addItemToCart(menuItem, quantity)
   - removeItemFromCart(menuItem)
   - updateCartQuantity(menuItem, newQuantity)
   - calculateTotalPrice()
   - setDeliveryAddress(address)
   - selectPaymentMethod(paymentMethod)
   - validateOrder()
   - processPayment(amount, paymentMethod)
   - confirmOrder(orderDetails)
   - generateOrderNumber()
   - notifyRestaurant(orderDetails)

2. From "Track Order" use case:

   - retrieveOrderStatus(orderID)
   - getEstimatedDeliveryTime(orderID)
   - getDeliveryPersonLocation(orderID)
   - displayTrackingInformation(orderID)
   - contactSupport(orderID)
   - contactDeliveryPerson(orderID)

3. From "Manage Orders" use case:
   - displayOrderList(restaurantID, orderStatus)
   - getOrderDetails(orderID)
   - acceptOrder(orderID)
   - rejectOrder(orderID, reason)
   - updateOrderStatus(orderID, newStatus)
   - markOrderReady(orderID)
   - assignDeliveryDriver(orderID, driverID)
   - notifyCustomer(orderID, message)
   - initiateRefund(orderID)
   - modifyOrder(orderID, modifications)

Additional system operations based on other use cases:

4. User Management:
   - registerUser(userDetails)
   - loginUser(credentials)
   - updateUserProfile(userID, newDetails)

5. Restaurant Management:
   - createRestaurantProfile(restaurantDetails)
   - updateRestaurantProfile(restaurantID, newDetails)
   - addMenuItem(restaurantID, menuItemDetails)
   - removeMenuItem(restaurantID, menuItemID)

   - updateMenuItem(restaurantID, menuItemID, newDetails)

6. Review and Rating:

   - submitReview(customerID, restaurantID, rating, comment)

   - getRestaurantRating(restaurantID)

   - displayReviews(restaurantID)

7. Promotion and Loyalty:

   - createPromotion(promotionDetails)

   - applyPromotion(orderID, promotionID)

   - calculateLoyaltyPoints(orderID)

   - redeemLoyaltyPoints(customerID, points)

8. Reporting:

   - generateSalesReport(restaurantID, dateRange)

   - generateCustomerActivityReport(dateRange)

   - generatePopularItemsReport(restaurantID, dateRange)

These system operations cover the main functionalities required by the food order and delivery system based on our use case specifications. They provide a foundation for designing the system's API and implementing the backend logic.

Each of these operations would typically be implemented as methods or functions in the system's codebase, with appropriate parameters and return values. The exact implementation details would depend on the chosen technology stack and architecture.

Would you like me to elaborate on any specific operation or discuss how these operations relate to the domain model we created earlier?

User

Create design sequence diagrams for system operations.

AI Model

I'll create design sequence diagrams for a few key system operations to illustrate how different objects in the system interact to fulfill these operations. I'll focus on three important operations: placeOrder, trackOrder, and manageOrder. These diagrams will be presented in a textual format.

1. placeOrder() Sequence Diagram:

```
Customer -> OrderController: placeOrder(items, address, paymentMethod)

OrderController -> ShoppingCart: calculateTotalPrice()

ShoppingCart --> OrderController: totalPrice

OrderController -> PaymentGateway: processPayment(totalPrice, paymentMethod)

PaymentGateway --> OrderController: paymentConfirmation

OrderController -> Order: createOrder(items, address, totalPrice)

Order -> Restaurant: notifyNewOrder(orderDetails)

Restaurant --> Order: acknowledgement

Order --> OrderController: orderConfirmation

OrderController --> Customer: orderConfirmation
```

2. trackOrder() Sequence Diagram:

```
Customer -> OrderController: trackOrder(orderID)

OrderController -> Order: getOrderStatus(orderID)
```

```
Order --> OrderController: orderStatus

OrderController -> DeliveryService: getEstimatedDeliveryTime(orderID)

DeliveryService --> OrderController: estimatedTime

OrderController -> DeliveryService: getDeliveryPersonLocation(orderID)

DeliveryService --> OrderController: location

OrderController --> Customer: trackingInfo(orderStatus, estimatedTime,
location)
```

3. manageOrder() Sequence Diagram (for restaurant owner):

```
RestaurantOwner -> OrderManagementController: getOrderList(restaurantID)

OrderManagementController -> Restaurant: fetchOrders(restaurantID)

Restaurant --> OrderManagementController: orderList

OrderManagementController --> RestaurantOwner: displayOrderList(orderList)

RestaurantOwner -> OrderManagementController: acceptOrder(orderID)

OrderManagementController -> Order: updateOrderStatus(orderID, "Accepted")

Order -> Customer: notifyOrderStatus("Accepted")

Order --> OrderManagementController: statusUpdateConfirmation

OrderManagementController --> RestaurantOwner: confirmationMessage
```

These sequence diagrams illustrate the flow of messages between objects for each operation. Here's a brief explanation of each:

1. placeOrder():

   This diagram shows the process of placing an order, from calculating the total price to processing payment and notifying the restaurant. It involves interactions between the Customer, OrderController, ShoppingCart, PaymentGateway, Order, and Restaurant objects.

2. trackOrder():

This diagram illustrates how a customer can track their order. It involves the Customer, OrderController, Order, and DeliveryService objects to gather and display tracking information.

3. manageOrder():

This diagram shows how a restaurant owner manages orders, including fetching the order list and accepting an order. It involves the RestaurantOwner, OrderManagementController, Restaurant, Order, and Customer objects.

These diagrams help visualize the interactions between different parts of the system and can guide the implementation of these operations. They also highlight the relationships between the objects in our domain model.

Would you like me to create sequence diagrams for any other operations, or do you need any clarification on these diagrams?

User

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

AI Model

Certainly. I'll create a design class diagram based on our domain model and the sequence diagrams we've discussed. This diagram will include the main classes, their attributes, methods, and relationships. I'll present this in a textual format, describing the classes and their relationships.

Design Class Diagram:

1. User (Abstract Class)

Attributes:

- userID: int

```
   - name: String

   - email: String

   - phoneNumber: String

   - address: String

   Methods:

   + updateProfile(userDetails: UserDetails): void



2. Customer (extends User)

   Attributes:

   - orderHistory: List<Order>

   - loyaltyPoints: int

   Methods:

   + placeOrder(items: List<MenuItem>, address: String, paymentMethod:
String): Order

   + trackOrder(orderID: int): TrackingInfo

   + submitReview(restaurantID: int, rating: int, comment: String): void

   + redeemLoyaltyPoints(points: int): void



3. RestaurantOwner (extends User)

   Attributes:

   - restaurantID: int

   Methods:

   + updateRestaurantProfile(restaurantDetails: RestaurantDetails): void

   + manageOrders(): List<Order>

   + updateMenuItem(menuItemID: int, details: MenuItemDetails): void
```

4. DeliveryDriver (extends User)

   Attributes:

   - vehicleInfo: String

   - currentLocation: Location

   Methods:

   + updateLocation(location: Location): void

   + acceptDelivery(orderID: int): void

   + completeDelivery(orderID: int): void

5. Restaurant

   Attributes:

   - restaurantID: int

   - name: String

   - address: String

   - cuisine: String

   - rating: float

   Methods:

   + updateProfile(details: RestaurantDetails): void

   + addMenuItem(item: MenuItem): void

   + removeMenuItem(itemID: int): void

   + getMenu(): Menu

6. Menu

   Attributes:

- menuID: int

- restaurantID: int

- items: List<MenuItem>

Methods:

+ addItem(item: MenuItem): void

+ removeItem(itemID: int): void

+ updateItem(itemID: int, details: MenuItemDetails): void

7. MenuItem

Attributes:

- itemID: int

- name: String

- description: String

- price: float

- category: String

Methods:

+ updateDetails(details: MenuItemDetails): void

8. Order

Attributes:

- orderID: int

- customerID: int

- restaurantID: int

- items: List<OrderItem>

- status: OrderStatus

```
   - totalPrice: float

   - orderDate: DateTime

   Methods:

   + updateStatus(newStatus: OrderStatus): void

   + addItem(item: OrderItem): void

   + removeItem(itemID: int): void

   + calculateTotal(): float
```

9. OrderItem

```
   Attributes:

   - orderItemID: int

   - menuItemID: int

   - quantity: int

   - specialInstructions: String

   Methods:

   + updateQuantity(newQuantity: int): void
```

10. Payment

```
    Attributes:

    - paymentID: int

    - orderID: int

    - amount: float

    - paymentMethod: String

    - paymentStatus: PaymentStatus

    Methods:
```

```
    + processPayment(): boolean


11. Review
    Attributes:
    - reviewID: int
    - customerID: int
    - restaurantID: int
    - rating: int
    - comment: String
    - date: DateTime


12. Promotion
    Attributes:
    - promotionID: int
    - description: String
    - discountAmount: float
    - validFrom: DateTime
    - validTo: DateTime
    Methods:
    + isValid(): boolean
    + applyDiscount(amount: float): float


13. OrderController
    Methods:
```

```
    + placeOrder(customerID: int, items: List<MenuItem>, address: String,
paymentMethod: String): Order

    + trackOrder(orderID: int): TrackingInfo

    + cancelOrder(orderID: int): boolean



14. RestaurantController

    Methods:

    + getRestaurantDetails(restaurantID: int): RestaurantDetails

    + updateRestaurantProfile(restaurantID: int, details: RestaurantDetails):
boolean

    + getMenu(restaurantID: int): Menu



15. PaymentGateway

    Methods:

    + processPayment(amount: float, paymentMethod: String): boolean



Relationships:

- Customer "places" Order (1 to many)

- Order "contains" OrderItem (1 to many)

- OrderItem "references" MenuItem (many to 1)

- Restaurant "has" Menu (1 to 1)

- Menu "contains" MenuItem (1 to many)

- Restaurant "is managed by" RestaurantOwner (many to 1)

- Order "is delivered by" DeliveryDriver (many to 1)

- Order "has" Payment (1 to 1)

- Customer "writes" Review for Restaurant (many to many)
```

- Promotion "applies to" Order (many to many)

- Restaurant "offers" Promotion (many to many)

This design class diagram incorporates the main entities from our domain model and adds methods based on the system operations and sequence diagrams we discussed. It also includes controller classes (OrderController, RestaurantController) and a PaymentGateway class to handle specific system operations.

The diagram shows the attributes and key methods for each class, as well as the relationships between classes. This structure provides a blueprint for implementing the system, showing how different parts of the system interact and what functionalities each class is responsible for.

Would you like me to elaborate on any specific part of this design class diagram?

User

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

AI Model

Certainly! I'll provide a Java implementation for key parts of the system based on our class diagram and sequence diagrams. Due to the size of the system, I'll focus on implementing core classes and a few important methods. This implementation will give you a solid starting point that you can expand upon.

Let's start with some of the main classes:

```java
import java.time.LocalDateTime;

import java.util.ArrayList;

import java.util.List;
```

```
// User (Abstract Class)
public abstract class User {
    protected int userID;
    protected String name;
    protected String email;
    protected String phoneNumber;
    protected String address;

    public void updateProfile(UserDetails userDetails) {
        // Implementation
    }
}

// Customer Class
public class Customer extends User {
    private List<Order> orderHistory;
    private int loyaltyPoints;

    public Order placeOrder(List<MenuItem> items, String address, String
paymentMethod) {
        // Implementation will be shown in OrderController
        return null;
    }
```

```
    public TrackingInfo trackOrder(int orderID) {
        // Implementation will be shown in OrderController
        return null;
    }

    public void submitReview(int restaurantID, int rating, String comment) {
        // Implementation
    }

    public void redeemLoyaltyPoints(int points) {
        // Implementation
    }
}

// Restaurant Class
public class Restaurant {
    private int restaurantID;
    private String name;
    private String address;
    private String cuisine;
    private float rating;
    private Menu menu;

    public void updateProfile(RestaurantDetails details) {
        // Implementation
```

```
    }

    public void addMenuItem(MenuItem item) {
        menu.addItem(item);
    }

    public void removeMenuItem(int itemID) {
        menu.removeItem(itemID);
    }

    public Menu getMenu() {
        return menu;
    }
}

// Menu Class
public class Menu {
    private int menuID;
    private int restaurantID;
    private List<MenuItem> items;

    public void addItem(MenuItem item) {
        items.add(item);
    }
```

```
    public void removeItem(int itemID) {
        items.removeIf(item -> item.getItemID() == itemID);
    }

    public void updateItem(int itemID, MenuItemDetails details) {
        // Implementation
    }
}

// MenuItem Class
public class MenuItem {
    private int itemID;
    private String name;
    private String description;
    private float price;
    private String category;

    public void updateDetails(MenuItemDetails details) {
        // Implementation
    }

    public int getItemID() {
        return itemID;
    }
```

```
    // Other getters and setters
}

// Order Class
public class Order {
    private int orderID;
    private int customerID;
    private int restaurantID;
    private List<OrderItem> items;
    private OrderStatus status;
    private float totalPrice;
    private LocalDateTime orderDate;

    public void updateStatus(OrderStatus newStatus) {
        this.status = newStatus;
    }

    public void addItem(OrderItem item) {
        items.add(item);
    }

    public void removeItem(int itemID) {
        items.removeIf(item -> item.getMenuItemID() == itemID);
    }
```

```
    public float calculateTotal() {

        return items.stream().map(OrderItem::getSubtotal).reduce(0f,
Float::sum);

    }

}



// OrderItem Class

public class OrderItem {

    private int orderItemID;

    private int menuItemID;

    private int quantity;

    private String specialInstructions;

    private float price;



    public void updateQuantity(int newQuantity) {

        this.quantity = newQuantity;

    }



    public float getSubtotal() {

        return price * quantity;

    }



    public int getMenuItemID() {

        return menuItemID;

    }
```

```
    // Other getters and setters

}



// OrderController Class

public class OrderController {

    private PaymentGateway paymentGateway;



    public Order placeOrder(int customerID, List<MenuItem> items, String
address, String paymentMethod) {

        Order order = new Order();

        order.setCustomerID(customerID);



        for (MenuItem item : items) {

            OrderItem orderItem = new OrderItem();

            orderItem.setMenuItemID(item.getItemID());

            orderItem.setQuantity(1); // Default quantity, can be modified

            orderItem.setPrice(item.getPrice());

            order.addItem(orderItem);

        }



        float totalPrice = order.calculateTotal();



        if (paymentGateway.processPayment(totalPrice, paymentMethod)) {

            order.setStatus(OrderStatus.PLACED);
```

```
            order.setOrderDate(LocalDateTime.now());
            // Save order to database
            return order;
        } else {
            throw new RuntimeException("Payment failed");
        }
    }

    public TrackingInfo trackOrder(int orderID) {
        Order order = getOrderFromDatabase(orderID);
        DeliveryService deliveryService = new DeliveryService();

        TrackingInfo trackingInfo = new TrackingInfo();
        trackingInfo.setOrderStatus(order.getStatus());

trackingInfo.setEstimatedDeliveryTime(deliveryService.getEstimatedDeliveryTim
e(orderID));

trackingInfo.setDeliveryPersonLocation(deliveryService.getDeliveryPersonLocat
ion(orderID));

        return trackingInfo;
    }

    private Order getOrderFromDatabase(int orderID) {
        // Implementation to fetch order from database
        return new Order();
```

```
    }
}


// PaymentGateway Class
public class PaymentGateway {
    public boolean processPayment(float amount, String paymentMethod) {
        // Implement payment processing logic
        return true; // Return true if payment is successful
    }
}


// Enum for OrderStatus
enum OrderStatus {
    PLACED, PREPARING, READY_FOR_DELIVERY, OUT_FOR_DELIVERY, DELIVERED,
CANCELLED
}


// Helper classes
class UserDetails {}
class RestaurantDetails {}
class MenuItemDetails {}
class TrackingInfo {
    private OrderStatus orderStatus;
    private LocalDateTime estimatedDeliveryTime;
    private String deliveryPersonLocation;
```

```
    // Getters and setters
}

class DeliveryService {
    public LocalDateTime getEstimatedDeliveryTime(int orderID) {
        // Implementation
        return LocalDateTime.now().plusMinutes(30); // Example
    }

    public String getDeliveryPersonLocation(int orderID) {
        // Implementation
        return "123 Main St"; // Example
    }
}
```

This implementation covers the core classes and some of the main functionalities of the food order and delivery system. Here are some key points:

1. The `User` class is abstract, with `Customer` extending it.

2. The `Order` class manages a list of `OrderItem` objects and can calculate the total price.

3. The `OrderController` handles the main logic for placing and tracking orders, interacting with other classes like `PaymentGateway` and `DeliveryService`.

4. The `Restaurant` class manages its menu, which consists of `MenuItem` objects.

5. Enums like `OrderStatus` are used for predefined sets of values.

6. Helper classes like `UserDetails`, `RestaurantDetails`, etc., are included as placeholders for more detailed implementations.

This implementation provides a foundation for the system. You would need to expand on this by:

1. Implementing database interactions (possibly using JPA or JDBC).

2. Adding more detailed error handling and validation.

3. Implementing the remaining classes and methods from the class diagram.

4. Adding a user interface layer (web or mobile app).

5. Implementing security features for user authentication and data protection.

6. Adding logging and monitoring capabilities.

Would you like me to elaborate on any specific part of this implementation or expand on any particular functionality?

User

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

AI Model

Certainly! I'll develop a set of tests for our food order and delivery system implementation. We'll create unit tests, integration tests, and system tests. For this example, I'll use JUnit 5 for our testing framework and Mockito for mocking dependencies.

Let's start with some example tests for each category:

1. Unit Tests

We'll begin with unit tests for the OrderController class:

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

class OrderControllerTest {

    @Mock
    private PaymentGateway paymentGateway;

    private OrderController orderController;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.openMocks(this);
        orderController = new OrderController();
        orderController.setPaymentGateway(paymentGateway);
    }
```

```
    @Test
    void testPlaceOrder_Success() {
        // Arrange
        int customerID = 1;
        List<MenuItem> items = Arrays.asList(
            new MenuItem(1, "Burger", "Tasty burger", 10.0f, "Main"),
            new MenuItem(2, "Fries", "Crispy fries", 5.0f, "Side")
        );
        String address = "123 Test St";
        String paymentMethod = "Credit Card";

        when(paymentGateway.processPayment(anyFloat(),
anyString())).thenReturn(true);

        // Act
        Order result = orderController.placeOrder(customerID, items, address,
paymentMethod);

        // Assert
        assertNotNull(result);
        assertEquals(OrderStatus.PLACED, result.getStatus());
        assertEquals(15.0f, result.calculateTotal());
        assertEquals(2, result.getItems().size());
        verify(paymentGateway).processPayment(15.0f, paymentMethod);
    }
```

```
    @Test

    void testPlaceOrder_PaymentFailure() {

        // Arrange

        int customerID = 1;

        List<MenuItem> items = Arrays.asList(new MenuItem(1, "Burger", "Tasty
burger", 10.0f, "Main"));

        String address = "123 Test St";

        String paymentMethod = "Credit Card";



        when(paymentGateway.processPayment(anyFloat(),
anyString())).thenReturn(false);



        // Act & Assert

        assertThrows(RuntimeException.class, () -> {

            orderController.placeOrder(customerID, items, address,
paymentMethod);

        });

    }



    @Test

    void testTrackOrder() {

        // Arrange

        int orderID = 1;

        Order mockOrder = mock(Order.class);

        when(mockOrder.getStatus()).thenReturn(OrderStatus.PREPARING);
```

```
        // Mocking the private method using PowerMockito
        OrderController spyOrderController = spy(orderController);

doReturn(mockOrder).when(spyOrderController).getOrderFromDatabase(orderID);

        // Act
        TrackingInfo result = spyOrderController.trackOrder(orderID);

        // Assert
        assertNotNull(result);
        assertEquals(OrderStatus.PREPARING, result.getOrderStatus());
        assertNotNull(result.getEstimatedDeliveryTime());
        assertNotNull(result.getDeliveryPersonLocation());
    }
}
```

2. Integration Tests

Now, let's create an integration test that checks the interaction between OrderController, PaymentGateway, and a mock database:

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

```
import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

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

import static org.mockito.Mockito.*;



class OrderIntegrationTest {



    @Mock

    private PaymentGateway paymentGateway;



    @Mock

    private OrderRepository orderRepository;



    private OrderController orderController;



    @BeforeEach

    void setUp() {

        MockitoAnnotations.openMocks(this);

        orderController = new OrderController(paymentGateway,
orderRepository);

    }



    @Test

    void testOrderFlow() {

        // Arrange
```

```
        int customerID = 1;

        List<MenuItem> items = Arrays.asList(new MenuItem(1, "Pizza",
"Delicious pizza", 15.0f, "Main"));

        String address = "456 Integration St";

        String paymentMethod = "PayPal";



        when(paymentGateway.processPayment(anyFloat(),
anyString())).thenReturn(true);

        when(orderRepository.save(any(Order.class))).thenAnswer(invocation ->
{

            Order savedOrder = invocation.getArgument(0);

            savedOrder.setOrderID(1); // Simulating database assigning ID

            return savedOrder;

        });



        // Act

        Order placedOrder = orderController.placeOrder(customerID, items,
address, paymentMethod);



        // Assert

        assertNotNull(placedOrder);

        assertEquals(1, placedOrder.getOrderID());

        assertEquals(OrderStatus.PLACED, placedOrder.getStatus());



        // Act

        TrackingInfo trackingInfo =
orderController.trackOrder(placedOrder.getOrderID());
```

```
        // Assert
        assertNotNull(trackingInfo);
        assertEquals(OrderStatus.PLACED, trackingInfo.getOrderStatus());

        verify(paymentGateway).processPayment(15.0f, paymentMethod);
        verify(orderRepository).save(any(Order.class));
    }
}
```

3. System Tests

For system tests, we'll create an end-to-end scenario that simulates a user placing an order and tracking it. This would typically be done with a tool like Selenium for web applications, but for this example, we'll create a simplified version:

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class FoodOrderSystemTest {

    private CustomerApp customerApp;
    private RestaurantApp restaurantApp;
```

```
    private DeliveryApp deliveryApp;


    @BeforeEach
    void setUp() {
        // Initialize your system components
        customerApp = new CustomerApp();
        restaurantApp = new RestaurantApp();
        deliveryApp = new DeliveryApp();
    }


    @Test
    void testCompleteOrderFlow() {
        // 1. Customer logs in
        assertTrue(customerApp.login("customer@example.com", "password"));


        // 2. Customer browses restaurants
        List<Restaurant> restaurants = customerApp.browseRestaurants();
        assertFalse(restaurants.isEmpty());


        // 3. Customer selects a restaurant
        Restaurant selectedRestaurant = restaurants.get(0);
        Menu menu =
customerApp.viewMenu(selectedRestaurant.getRestaurantID());
        assertFalse(menu.getItems().isEmpty());
```

```
        // 4. Customer places an order

        List<MenuItem> selectedItems = Arrays.asList(menu.getItems().get(0),
menu.getItems().get(1));

        Order order =
customerApp.placeOrder(selectedRestaurant.getRestaurantID(), selectedItems,
"789 System Test Ave", "Credit Card");

        assertNotNull(order);

        assertEquals(OrderStatus.PLACED, order.getStatus());



        // 5. Restaurant receives and accepts the order

        assertTrue(restaurantApp.login("restaurant@example.com",
"password"));

        List<Order> newOrders = restaurantApp.getNewOrders();

        assertTrue(newOrders.stream().anyMatch(o -> o.getOrderID() ==
order.getOrderID()));

        assertTrue(restaurantApp.acceptOrder(order.getOrderID()));



        // 6. Restaurant prepares the order

        assertTrue(restaurantApp.updateOrderStatus(order.getOrderID(),
OrderStatus.PREPARING));

        assertTrue(restaurantApp.updateOrderStatus(order.getOrderID(),
OrderStatus.READY_FOR_DELIVERY));



        // 7. Delivery driver picks up the order

        assertTrue(deliveryApp.login("driver@example.com", "password"));

        assertTrue(deliveryApp.acceptDelivery(order.getOrderID()));

        assertTrue(deliveryApp.updateOrderStatus(order.getOrderID(),
OrderStatus.OUT_FOR_DELIVERY));
```

```
        // 8. Customer tracks the order

        TrackingInfo trackingInfo =
customerApp.trackOrder(order.getOrderID());

        assertEquals(OrderStatus.OUT_FOR_DELIVERY,
trackingInfo.getOrderStatus());

        assertNotNull(trackingInfo.getEstimatedDeliveryTime());

        assertNotNull(trackingInfo.getDeliveryPersonLocation());



        // 9. Delivery driver completes the delivery

        assertTrue(deliveryApp.completeDelivery(order.getOrderID()));



        // 10. Customer checks the completed order

        Order completedOrder =
customerApp.getOrderDetails(order.getOrderID());

        assertEquals(OrderStatus.DELIVERED, completedOrder.getStatus());



        // 11. Customer leaves a review


assertTrue(customerApp.submitReview(selectedRestaurant.getRestaurantID(), 5,
"Great food and quick delivery!"));

    }

}
```


These tests cover different aspects of our system:

1. Unit tests focus on individual components, testing their behavior in isolation.

2. Integration tests verify that different components of the system work together correctly.

3. System tests simulate real-world scenarios, testing the entire system end-to-end.

To run these tests, you would need to:

1. Set up a testing environment with JUnit 5 and Mockito.

2. Implement the necessary mock objects and stubs for dependencies.

3. Ensure that your application's components (CustomerApp, RestaurantApp, DeliveryApp) have the required methods to support these tests.

4. Run the tests as part of your continuous integration pipeline.

Remember to expand on these tests as you develop more features and to maintain them as your codebase evolves. Also, consider adding more edge cases and error scenarios to ensure robust testing coverage.