ASSIGNMENT 2

Restaurant Information System

TRAN DUC ANH DANG
ERTESAM NASER ZARIF
LUAN NGUYEN
Saw Ko Ko Oo

Abstract

The Restaurant Information System (RIS) aims to enhance operational efficiency and customer service at The Relaxing Koala, a café/restaurant expanding its capacity from 50 to 150 seats. As the establishment grows, the limitations of its current manual and low tech systems, particularly in order taking, kitchen communication, and payment processing, have necessitated the development of a more sophisticated information system. This system is designed to streamline operations, integrate various operational aspects seamlessly, and leverage advanced data analytics to monitor and optimize service flow, thereby improving customer satisfaction by reducing wait times and personalizing service based on customer data. The object design of the RIS, detailed through candidate classes, UML diagrams, and CRC cards, focuses on automating key functions such as reservations, order management, kitchen operations, and payment transactions. The design also prioritizes scalability and adaptability to future enhancements. This abstract summarizes the intent, scope, and architectural strategy of the RIS, poised to transform the operational capabilities of The Relaxing Koala.

Table of Contents

Introduction

The expansion of The Relaxing Koala, a café/restaurant located on Glenferrie Road, from a 50-seat to a 150-seat capacity, necessitates a substantial enhancement in its operational infrastructure to support increased customer volume and service demands. Recognizing the limitations of its current low tech and mostly manual operational methods, particularly in order taking, kitchen communication, and payment processing, there is a pressing need for a more sophisticated information system. This system aims to streamline operations, enhance the customer service experience, and integrate various operational aspects seamlessly. To further support these endeavors, the proposed system incorporates advanced data analytics to monitor and optimize the flow of service activities. This will not only improve operational efficiency but also enhance customer satisfaction by reducing wait times and personalizing the dining experience based on customer preferences and historical data.

This document presents the object design for the proposed Restaurant Information System (RIS) intended to meet these needs. The design will detail the candidate classes, their responsibilities, interactions, and the overall system architecture using UML diagrams. This effort aligns with the primary goal of automating processes such as reservations, order management, kitchen operations, and payment transactions to handle the increased throughput efficiently. The object design has been formulated to address the specific operational challenges identified in the case study provided in "Assignment 1" and the requirements outlined in the "Case Study Restaurant Information System-4.pdf." By employing a systematic design approach, the proposed RIS will not only cater to the immediate operational needs but also ensure scalability and adaptability to future enhancements.

In summary, this document will outline the object-oriented design considerations, justify the chosen design patterns, and demonstrate the system's potential through detailed class descriptions and interaction diagrams. This approach is intended to provide a comprehensive blueprint for developers and stakeholders involved in the system's development and implementation.

System Requirements and Analysis

Problem Domain Analysis:

Goal

The overarching goal of the system is to expedite service efficiency and cope with the expansion of The Relaxing Koala restaurant. This builds on the requirements provided in the previous Software Requirements Specification (SRS) document to create a proposed system for the restaurant. This will be achieved by meeting the requirements followed:

- Manage reservations and walk-in customers through the same system.
- Process orders from the customer and forward it to the kitchen.
- Allow for multiple forms of payments for customers and create invoices.
- Automatic management of inventory through items being used.
- Maintain table turnover between departing and arriving customers in the system.
- Provide a form for customer feedback for future review.
- Create an interface for staff roster schedules.

To further enhance the system's capabilities, integration with digital marketing tools will be explored to leverage customer data for targeted promotions and improve customer engagement. Additionally, real-time reporting features will be developed to provide management with insights into daily operations, helping to make informed decisions quickly and efficiently.

Assumptions:

The following assumptions have been made in creating the object design:

- **A1** Each customer is associated with either a single reservation or walk-in.
- **A2** Customers with reservations are linked with a specific time slot.
- **A3** All orders are linked with specific tables.
- **A4** Each order is linked to a payment.
- **A5** Menu items will have a unique identifier in the system.
- **A6** Menu items cannot be changed or varied for different customers.
- **A7** Inventory is updated based on stock level from orders and usage.
- **A8** A customer may order specific menu items and then later on order something else and is linked to the allocated table.
- **A9** An order is deemed complete once it has been served to the allocated table.
- **A10** The restaurant will serve a maximum of 150 customers at any point.
- **A11** Each table has a turnover period indicating the time required to prepare it for the next customer.
- **A12** Feedback forms are provided to the customer at the end of their meal.
- **A13** All staff members will have their availability time periods.
- **A14** Roster schedules will include specific roles and assigned shifts.
- **A15** All orders will be backed up in a database for the management to analyse.
- **A16** Invoices will contain information about the items ordered.

Simplifications Made:

To reduce complexity and improve efficiency in the system, the following simplifications have been made:

- Once entered in the system, orders from reservations and walk-in customers are processed in the same manner. This unified processing approach will be supported by a sophisticated algorithm that prioritizes orders based on a variety of factors, including customer arrival time, order complexity, and available resources, ensuring optimal service delivery and customer satisfaction.

System Design Overview

Candidate Classes:

List of System Classes: Names and Purposes

- **MenuUpdateManager**: Manages updates to the restaurant menu, including adding, removing, or modifying menu items.
- **SupplierCoordinator**: Coordinates orders and deliveries from suppliers to ensure adequate inventory levels.
- InvoiceManager: Handles the creation, processing, and tracking of customer invoices.
- **SupplyChainAnalyzer**: Analyzes supply chain operations to identify improvements in supplier performance and inventory management.
- **ReservationManager**: Manages all activities related to booking, modifying, and canceling reservations.
- **OrderManager**: Handles the receipt, processing, and modification of customer orders.
- **PaymentProcessor**: Processes all customer payments and manages financial transactions including invoice creation and dispute resolution.
- **InventoryController**: Monitors stock levels and manages inventory reordering based on usage.
- **TableManager**: Manages table assignments and tracks table availability and occupancy status.
- **FeedbackHandler**: Collects and processes customer feedback to improve service quality.
- **StaffScheduler**: Schedules and manages staff shifts, handling requests and conflicts for shift changes.
- **KitchenDisplaySystem**: Displays current order information to kitchen staff and notifies them when dishes are ready.
- **CustomerInterface**: Provides an interface for customers to place orders and submit feedback.

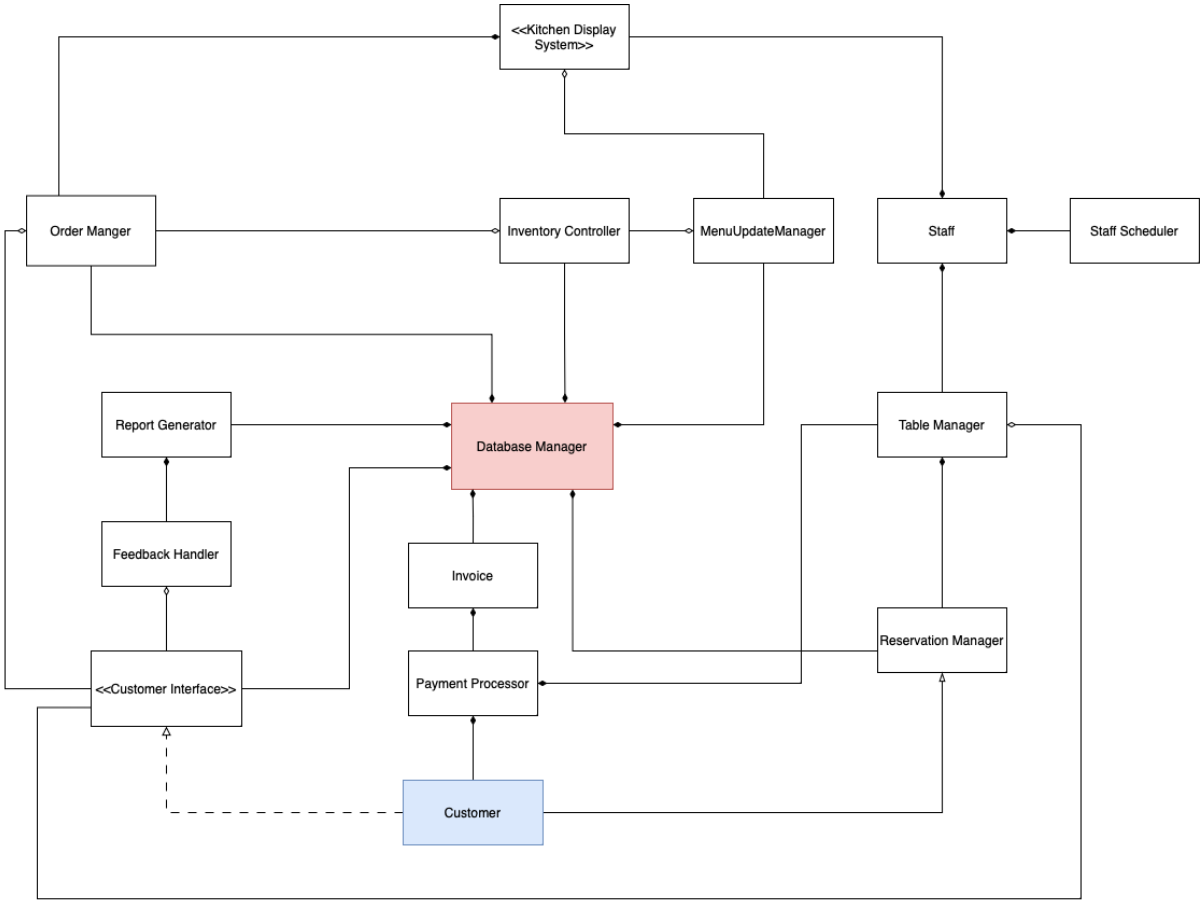
- **ReportGenerator**: Generates reports detailing operations and customer interactions.
- **DatabaseManager**: Manages data storage, retrieval, integrity, and ensures data security and privacy.
- **Customer**: Represents the users who interact with the system to access various services.
- **Staff**: Represents restaurant employees who interact with the system components as part of their job roles.

Justification for Class Choices:

Each class in the Restaurant Information System is designed to specifically address distinct aspects of restaurant operations, thereby pro moting efficiency, enhancing customer service, and ensuring effective management. Key considerations include:

- **Operational Efficiency**: Classes like ***OrderManager*** and KitchenDisplaySystem are critical for minimizing customer wait times and improving kitchen operations.
- **Customer Interaction**: ***CustomerInterface*** and ***FeedbackHandler*** enhance customer engagement and satisfaction by providing modern, user friendly interfaces and actively using customer feedback.
- **Staff Management**: ***StaffScheduler*** and ***Staff*** ensure that the restaurant operates smoothly, particularly during peak hours.

UML Diagram:



CRC Cards:

Detailed Responsibility and Collaboration Cards for Each Class.

MenuUpdateManager

Attribute	**Details**
Super Class	-
Responsibilities	Manage updates to the menu, including adding, removing, or modifying items. Adjust offerings based on seasonality and ingredient availability.
Collaborators	InventoryController KitchenDisplaySystem DatabaseManager

SupplierCoordinator

Attribute	**Details**
Super Class	-

Responsibilities	Coordinate orders with suppliers, ensure timely delivery of goods, manage supplier relationships, and handle contract negotiations.
Collaborators	InventoryController DatabaseManager

InvoiceManager

Attribute	**Details**
Super Class	-
Responsibilities	Create, process, and manage customer invoices. Ensure accuracy and compliance in billing.
Collaborators	PaymentProcessor DatabaseManager CustomerInterface

ReservationManager

Attribute	**Details**
Super Class	-
Responsibilities	Manage all types of reservations. Allocate and reallocate tables as needed.
Collaborators	Customer TableManager DatabaseManager

OrderManager

Attribute	**Details**
Super Class	-
Responsibilities	Receive and process orders. Update order status and handle modifications.
Collaborators	Customer Interface KitchenDisplaySystem DatabaseManager

PaymentProcessor

Attribute	**Details**
Super Class	-
Responsibilities	Process payments. Manage invoice creation and payment disputes.
Collaborators	Customer Interface DatabaseManager

InventoryController

Attribute	**Details**
Super Class	-
Responsibilities	Monitor and reorder inventory. Update stock levels based on usage.
Collaborators	OrderManager DatabaseManager

TableManager

Attribute	**Details**
Super Class	-
Responsibilities	Manage table assignments. Track table availability and status.
Collaborators	ReservationManager Staff

FeedbackHandler

Attribute	**Details**
Super Class	-
Responsibilities	Collect and process customer feedback.
Collaborators	Customer Interface DatabaseManager

StaffScheduler

Attribute	Details
Super Class	-
Responsibilities	Schedule and manage staff shifts. Handle shift change requests and conflicts.
Collaborators	Staff DatabaseManager

KitchenDisplaySystem

Attribute	Details
Super Class	-
Responsibilities	Display current orders to kitchen staff. Notify staff when dishes are ready.
Collaborators	OrderManager Staff

CustomerInterface

Attribute	Details
Super Class	-
Responsibilities	Provide interface for customer order and feedback submission.
Collaborators	Customer OrderManager FeedbackHandler

ReportGenerator

Attribute	Details
Super Class	-
Responsibilities	Generate detailed reports on operations and customer interactions.
Collaborators	DatabaseManager FeedbackHandler

DatabaseManager

Attribute	Details
Super Class	-
Responsibilities	Manage data storage, retrieval, and integrity. Ensure data security and privacy.
Collaborators	All other system classes

Customer

Attribute	Details
Super Class	-
Responsibilities	Interact with the system for services.
Collaborators	CustomerInterface Reservation Manager

Staff

Attribute	Details
Super Class	-
Responsibilities	Interact with system components relevant to their roles.
Collaborators	TableManager KitchenDisplaySystem StaffScheduler

Design Heuristics and Patterns

Application of Design Heuristics:

- H1. A class should capture one and only one key abstraction. Each class, such as **ReservationManager** or **PaymentProcessor**, focuses solely on one key function, ensuring simplicity and clarity in system design.
- H2.Keep related data and behaviour in one place. Classes like **InventoryController** encapsulate all behaviors and data related to inventory management.
- H3.Distribute system intelligence horizontally as uniformly as possible, that is, top-level classes of the system should share their work uniformly.
- Work is uniformly distributed among classes, with no single class handling excessive responsibilities.

- H4.Do not create god classes/objects in the system. There is no single class that controls multiple major functions, preventing overcomplexity and single points of failure.
- H5.Practice proper use of Inheritance. Inheritance is used appropriately to model is-a relationships, such as between a general Employee class and specific roles like **staff**.
- H6. Practice limits Depth of Inheritance. Inheritance hierarchies are kept shallow to make them understandable and maintainable.
- H7.Choose Polymorphism over Case Analysis.Polymorphism is used to handle functions that might require different implementations in derived classes, such as different types of payments processed by **PaymentProcessor**.
- H8.All abstract classes must be base classes. Key foundational classes like **DatabaseManager** are designed to be abstract, ensuring they provide a base for more specific subclasses without being instantiated directly.
- H9.Minimize the number of classes with which another class collaborates. Classes are designed to operate with minimal dependencies, reducing the complexity of interactions and increasing modularity.

Design Patterns Implemented:

Factory Method Design Pattern

The ***Factory Method*** pattern is ideal for scenarios where the system needs to manage the creation of various types of objects that share a common base class or interface but require different instantiation specifics. In this context, managers like ***MenuUpdateManager***, ***OrderManager***, and ***ReservationManager*** could extend a common ***Manager*** class that defines a standard interface. Each subclass would implement a factory method to create specific components or handle particular requests tied to their management domain.

Singleton Design Pattern

The Singleton pattern is particularly useful for managing resources that are central to system operations and need to be accessed in a controlled manner. ***DatabaseManager***, managing all data storage and retrieval operations, is a perfect candidate for this pattern. Implementing ***DatabaseManager*** as a ***Singleton*** ensures that there is only a single instance managing database operations, which helps maintain data integrity and prevents conflicts in data access across the system.

Strategy Design Pattern

The Strategy pattern allows the system to define a family of algorithms, encapsulate each one, and make them interchangeable. This flexibility is crucial for components like ***PaymentProcessor***, which can use different strategies for processing payments such as credit card, online wallet, or cash. Each payment method would implement a common payment interface. Similarly, ***InventoryController*** could employ various

strategies for inventory reordering like just-in-time, economic order quantity, or demand forecasting, each adhering to a common strategy interface.

Model-View-Controller (MVC) Pattern

The MVC pattern divides an application into three main logical components, **Model**, **View**, and **Controller**. Which helps in managing complexity by isolating the user interface, data, and business logic. In this setup:

- **Model**: ***DatabaseManager*** handles all data-related logic.
- **View**: ***CustomerInterface*** and ***KitchenDisplaySystem*** act as the user interfaces, where CustomerInterface allows customers to interact with the system, and ***KitchenDisplaySystem*** displays order information to kitchen staff.
- **Controller**: ***OrderManager*** processes and manages customer orders, directing how data flows into model objects and updates the view as necessary.

Command Pattern

The Command design pattern is instrumental in enhancing system flexibility by encapsulating requests as objects, which allows for parameterization of clients with various requests, and supports the queuing, logging, and undoing of these requests. In practical terms, this pattern is applied across several system components: the **OrderManager** uses it to manage diverse order-related operations such as addition, modification, or cancellation, allowing for undo capabilities and operational logging. The **KitchenDisplaySystem** leverages commands to manage the display of order information, facilitating dynamic updates and removals. The **PaymentProcessor** benefits from this pattern by encapsulating different payment methods into commands, making it easy to extend payment options without extensive code modifications. Similarly, the **FeedbackHandler** uses command objects to streamline the collection and processing of customer feedback, ensuring flexibility in handling future changes. Overall, the Command pattern significantly decouples the execution from the implementation of operations, providing a modular and extensible framework that simplifies future enhancements and maintenance.

Bootstrap and Initialization Process

System Startup Sequence:

- **DatabaseManager**: Initialized first to establish a stable and secure connection to the database, enabling reliable data operations throughout the system.
- **CustomerInterface**: Initialized early to ensure the user interface is ready for interaction as soon as other system components are operational.
- **ReservationManager** and **OrderManager**: Loaded following the DatabaseManager. These managers handle bookings and orders and are dependent on the DatabaseManager for data retrieval and storage.

- **PaymentProcessor** and **InventoryController**: Crucial for managing financial transactions and inventory levels, these components are initialized after the **OrderManager** to process payment and stock data effectively.
- **MenuUpdateManager** and **SupplierCoordinator**: These classes manage realtime menu updates and supplier relationships. They coordinate with the **InventoryController** to adjust the menu based on stock levels and with the **DatabaseManager** to update supplier and inventory information.
- **InvoiceManager**: Responsible for managing invoicing and special event bookings, ensuring that financial transactions and event details are captured and stored accurately.
- **TableManager**: Initializes to manage table seating and availability, essential for operations once reservations and orders are being processed.
- **KitchenDisplaySystem**: Essential for communicating order details to the kitchen staff effectively; activated once the order and inventory systems are operational.
- **FeedbackHandler** and **StaffScheduler**: These components are initialized to manage customer feedback and staff shifts, crucial for ongoing operational adjustments and customer service.
- **SupplyChainAnalyzer**: Loaded to analyze and enhance the efficiency of supply chain operations, working in tandem with the SupplierCoordinator and InventoryController.
- **ReportGenerator**: Prepared to generate reports based on operational data from other system components, providing insights and analytics for management.

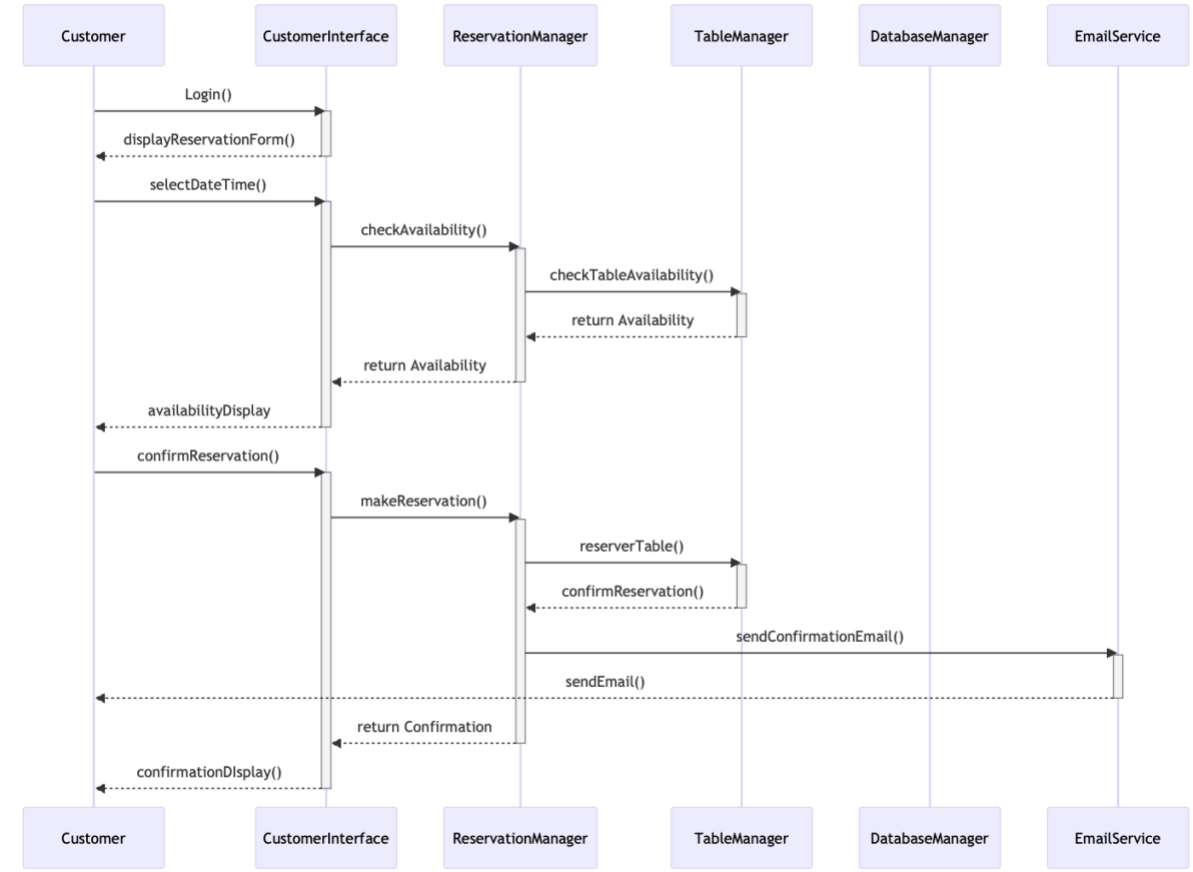
Design Verification and Scenarios

Scenario Analysis:

Use Case 1: Making a Reservation
Description: A customer uses the CustomerInterface to make a dinner reservation.
Steps:

1. **Customers** log into their account.
2. **Customers** select the date, time, and number of people for the reservation.
3. **ReservationManager** checks table availability via **TableManager**.
4. **Customers** confirm the reservation.
5. System sends a confirmation email.

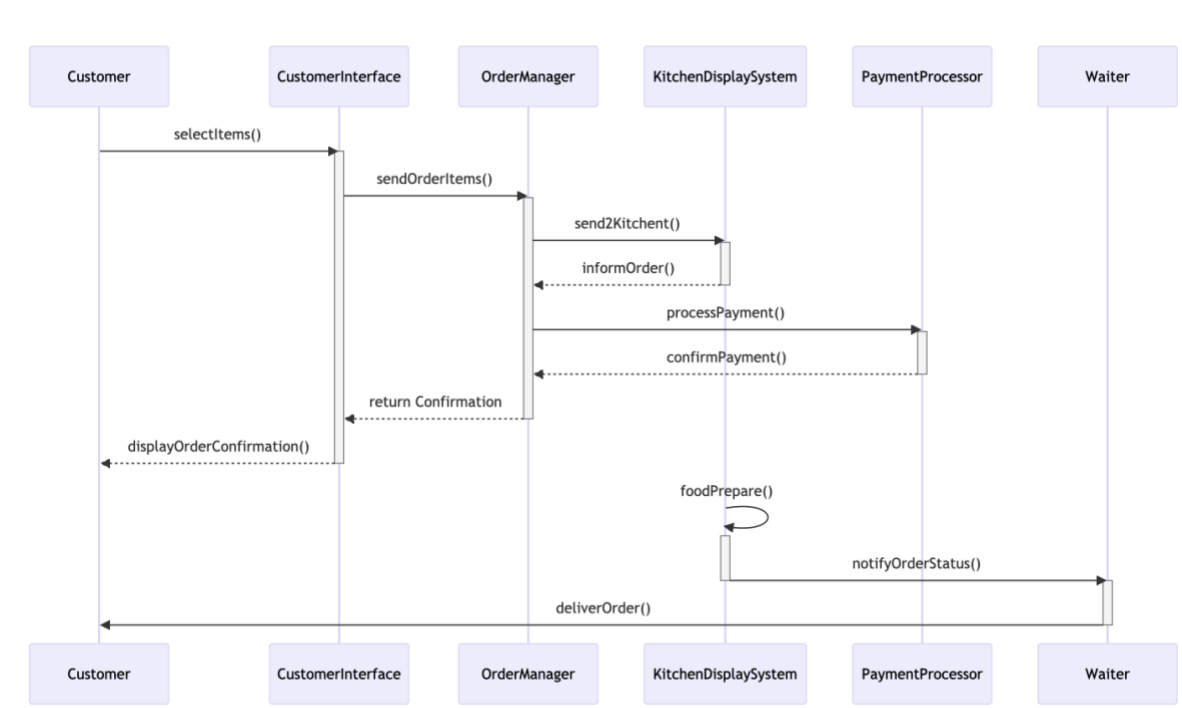
Outcome: Reservation is logged in the system, and the customer receives a confirmation.
UML:



Use Case 2: Ordering Food
Description: A customer places an order through the CustomerInterface.
Steps:

1. **Customer** selects items from the menu.
2. **OrderManager** processes the order and sends details to the **KitchenDisplaySystem**.
3. Kitchen prepares the food.
4. **PaymentProcessor** handles the payment transaction.
5. Order is delivered to the customer's table.

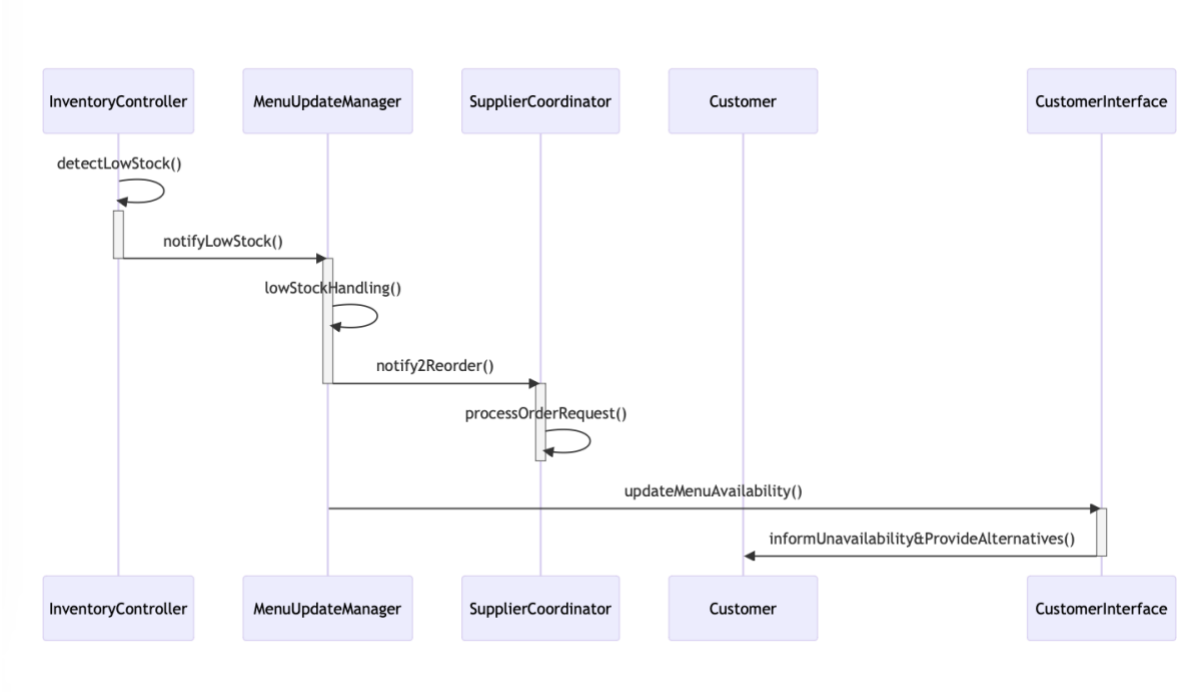
Outcome: Order is processed efficiently, payment is secured, and the customer receives their meal as ordered.
UML:



Use Case 3: Handling a Stock Outage
Description: An item on the menu runs out of stock.
Steps:

1. **InventoryController** detects low stock and alberts **MenuUpdateManager**.
2. **MenuUpdateManager** temporarily removes the item or suggests alternatives.
3. **SupplierCoordinator** is notified to reorder the stock.
4. **Customers** are informed about the unavailability or provided with alternatives.

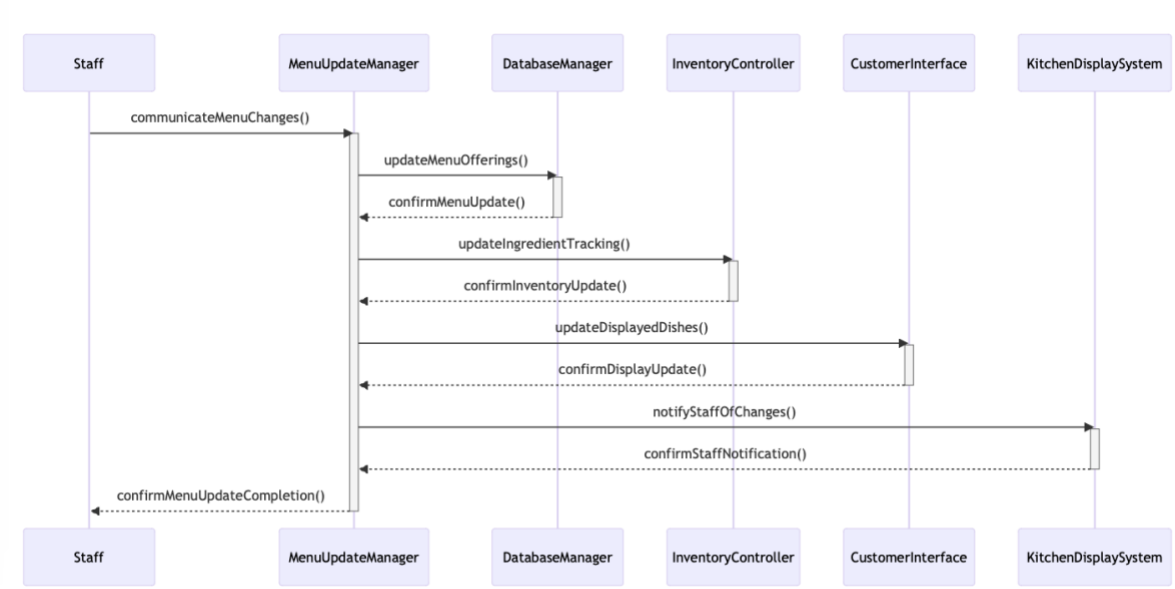
Outcome: The system handles stock outages smoothly, minimizes customer inconvenience, and updates inventory promptly.
UML:



Use Case 4: Updating the Menu
Description: The restaurant introduces seasonal dishes and needs to update the menu.
Steps:

1. **Staff** (Chef) decides on new dishes and communicates the changes to the **MenuUpdateManager**.
2. **MenuUpdateManager** accesses the **DatabaseManager** to update the menu offerings.
3. **InventoryController** is updated to track new ingredients.
4. **CustomerInterface** is updated to display new dishes.
5. **Staff** is notified of the changes through the **KitchenDisplaySystem**.

Outcome: The menu is updated across all platforms, inventory tracking adjusts to new requirements, and staff are informed of the change, ensuring seamless transition and customer experience.
UML:



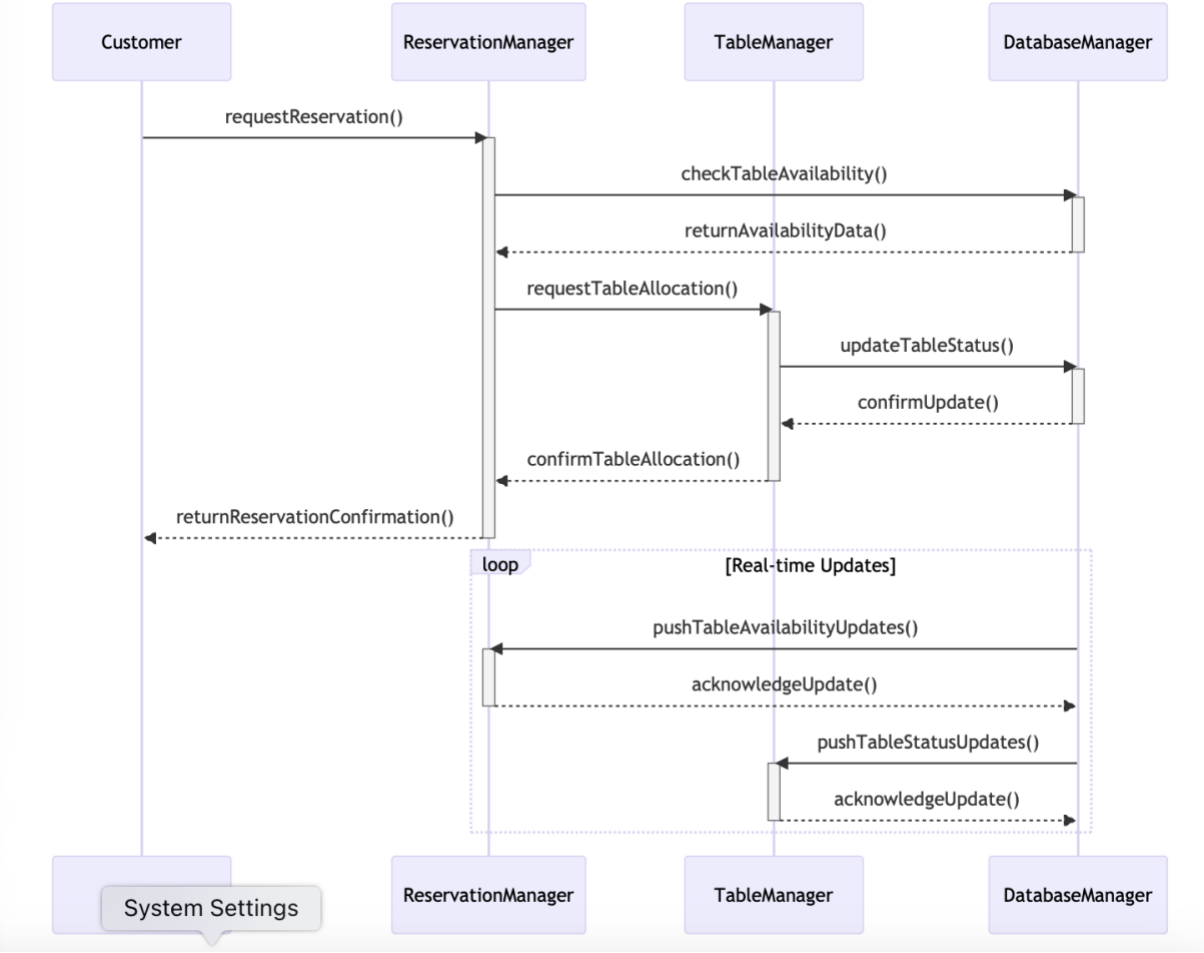
Impact Analysis:

Scenario 1: High Volume Traffic
Description: The system experiences unexpectedly high customer traffic both online and in the restaurant.
Impact:

- **ReservationManager** and **TableManager** are tested for performance under stress. The system's response time and data handling must remain optimal to prevent crashes or slowdowns.

Mitigation Measures:

- Implement load balancing and provide realtime data updates to manage reservations and table allocations efficiently.

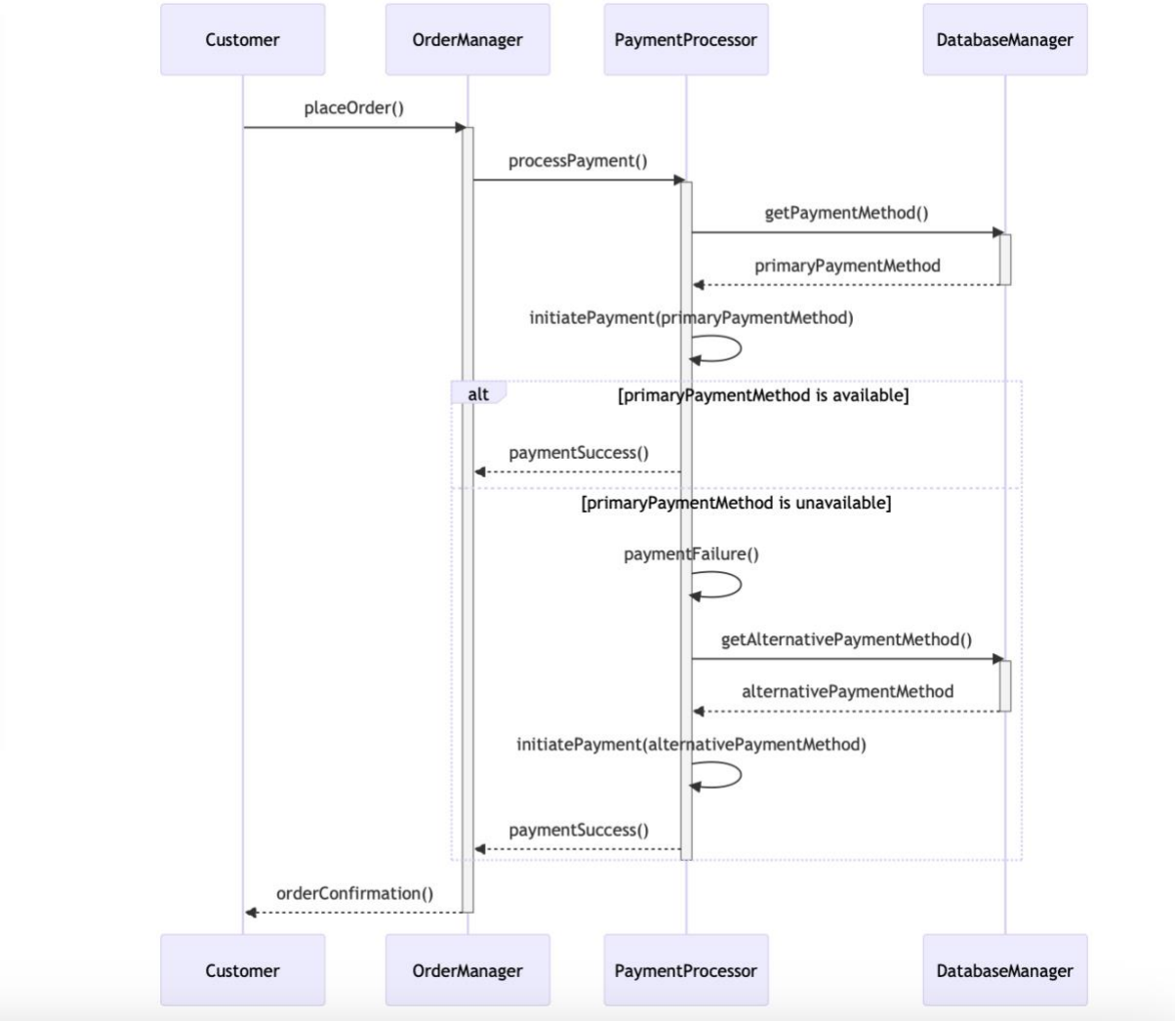


Scenario 2: Payment System Failure
Description: The primary payment gateway experiences an outage.
Impact:

- **PaymentProcessor** needs to switch to a secondary payment method without interrupting customer transactions.

Mitigation Measures:

- Use the Strategy Pattern to quickly switch between different payment providers or methods to ensure continuous operation.

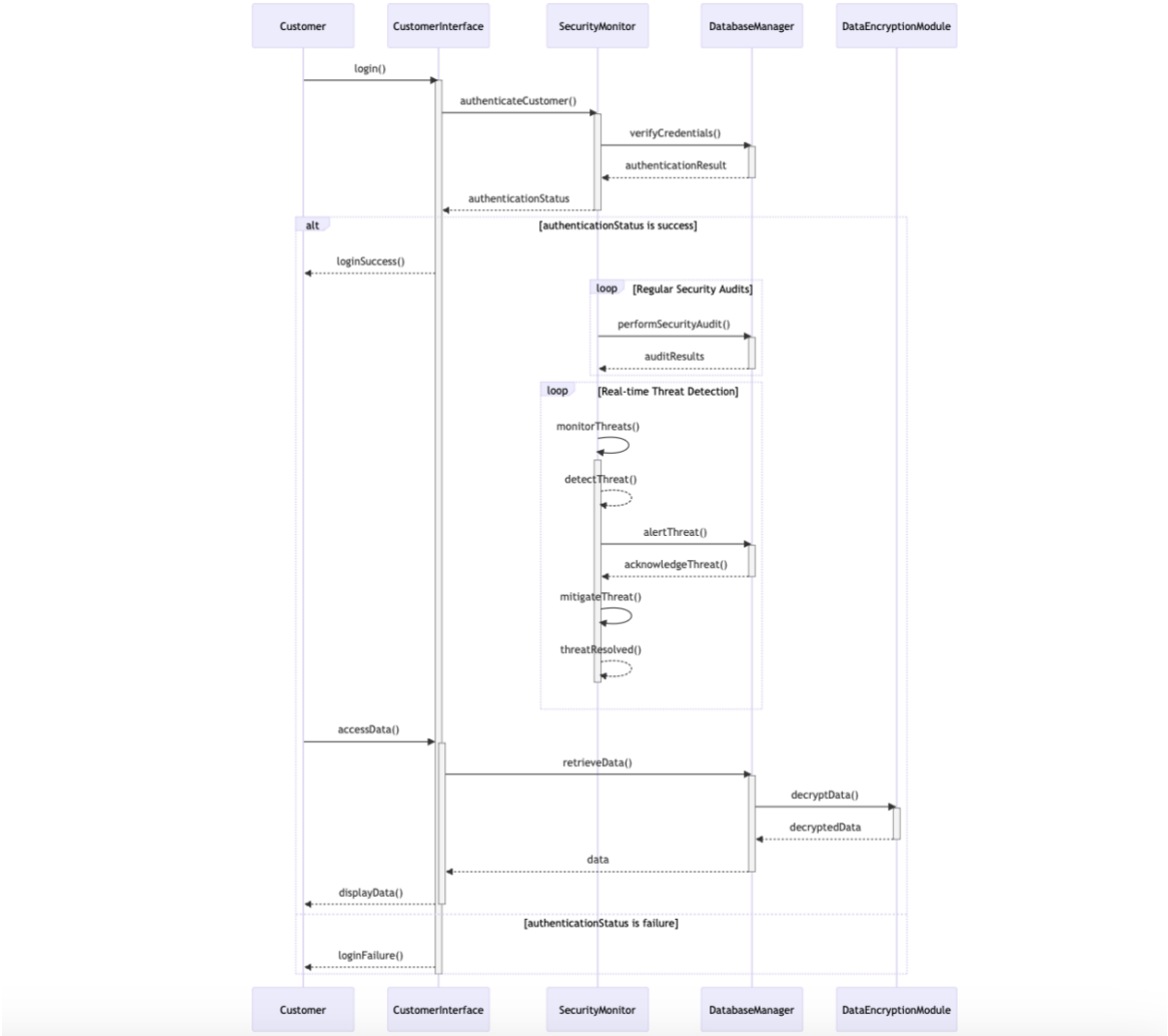


Scenario 3: Data Breach or Security Threat
Description: The system faces a cybersecurity threat, risking data integrity and privacy.
Impact:

- DatabaseManager and the overall security architecture are crucial in defending against such threats.

Mitigation Measures:

- Regular security audits, robust encryption practices, and real-time threat detection systems to safeguard data.

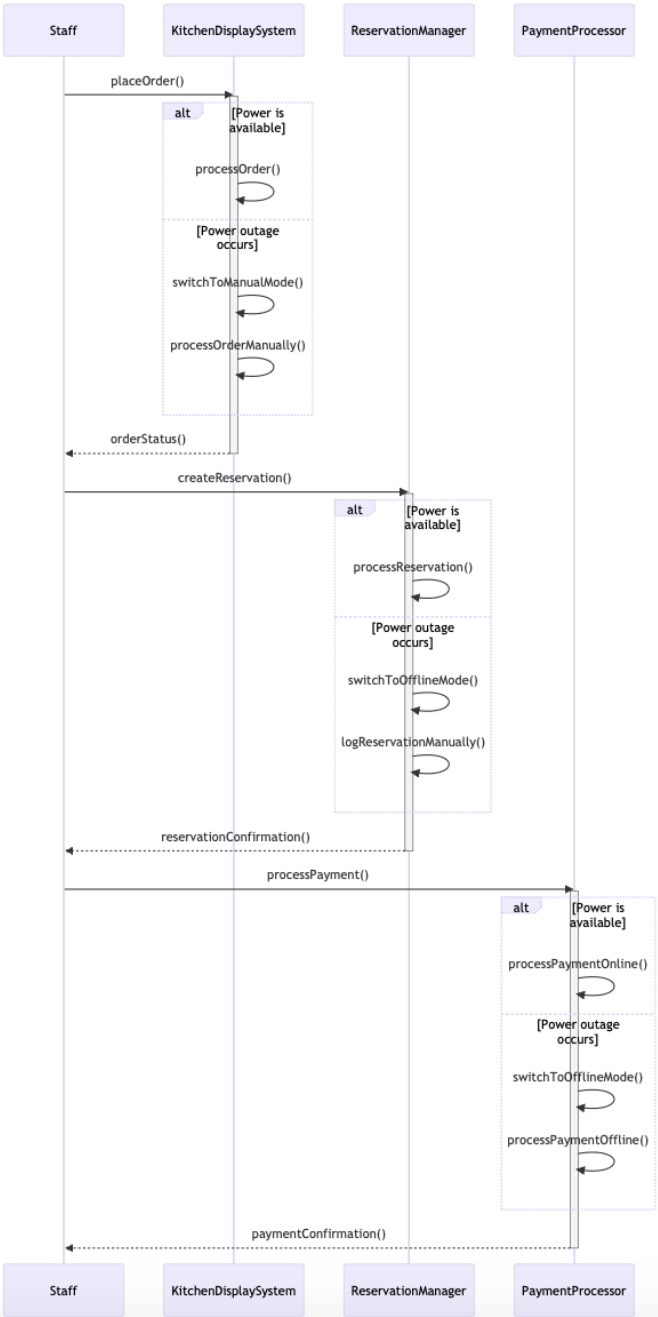


Scenario 4: Power Outage
Description: The restaurant faces an unexpected power outage.
Impact:

- Critical systems need to continue operating to maintain order processing, reservations, and basic lighting.

Mitigation Measures:

- Emergency power generators kick in to ensure continuity of the **KitchenDisplaySystem** and critical computing infrastructure.
- **ReservationManager** switches to an offline mode, where reservations can still be logged and managed manually if necessary.
- **PaymentProcessor** utilizes offline transaction processing to handle payments without connectivity.



References

- Assignment 1 - Requirements Specification | Assignment1.docx