Analysis and Design Document

Student: Crisan Oana Andra

**Group:30431**

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| <dd/mmm/yy> | <x.x> | <details> | <name> |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

I. Project Specification 4

II. Elaboration – Iteration 1.1 4

1. Domain Model 4

2. Architectural Design 4

2.1 Conceptual Architecture 4

2.2 Package Design 4

2.3 Component and Deployment Diagrams 4

III. Elaboration – Iteration 1.2 4

1. Design Model 4

1.1 Dynamic Behavior 4

1.2 Class Design 4

2. Data Model 4

3. Unit Testing 4

IV. Elaboration – Iteration 2 4

1. Architectural Design Refinement 4

2. Design Model Refinement 4

V. Construction and Transition 5

1. System Testing 5

2. Future improvements 5

VI. Bibliography 5

# Project Specification

The goal of this project is to design and implement a client-server application that can be used by a restaurant to record orders, handle payments, and manage the employees' activity. The system should allow waiters to record orders, handle payments, and apply discounts based on the loyalty of the client. Kitchen employees should be able to send notifications to waiters when the ordered food is ready. A manager should be able to perform CRUD operations on employees' information and check their activity.

The order should have the following attributes: table number, client ID, waiter ID, the list of ordered products (drinks, food, dessert, etc.), and their quantity. The loyalty of a client should be evaluated based on the number of times the client ordered from the restaurant and the cost of each order.

# Elaboration – Iteration 1.1

# Domain Model

The domain model for the restaurant order management system consists of the following classes:

* Order: represents an order made by a customer and contains attributes such as table number, client ID, waiter ID, and the list of ordered products.
* Product: represents a food or drink item that can be ordered by a customer and contains attributes such as name, description, price, and category.
* Client: represents a customer who can order from the restaurant and contains attributes such as name, address, and loyalty points.
* Waiter: represents an employee who can take orders and handle payments and contains attributes such as name, ID, and salary.
* Manager: represents an employee who can perform CRUD operations on employees' information and check their activity and contains attributes such as name, ID, and salary.
* Kitchen: represents the kitchen employees who can send notifications to waiters when the ordered food is ready.

# Architectural Design

## Conceptual Architecture

The restaurant order management system is designed as a client-server application that utilizes a layered architecture style. The system is divided into four layers that separate the concerns of user interface, application logic, data access, and data storage.

The Presentation Layer is responsible for providing a user-friendly interface for the restaurant staff to interact with the system. This layer includes the graphical user interface (GUI) that displays the menu, allows waiters to record orders and handle payments, and enables managers to view and edit employee information. (this part may be done in a different project, depending on which programming language I will choose: Angular or React).

The Business Logic Layer contains the application logic of the system. It includes service objects such as controllers and use cases that manage the workflow of the system.

The Data Access Layer provides an interface between the Business Logic Layer and the Domain Layer. This layer includes the database access layer that retrieves and persists data in the database.

The Domain Layer is responsible for representing the data model and contains the entities and value objects of the system.

The communication between layers is unidirectional, with each layer only interacting with the layer directly below it.

## Package Design

I have divided the system into four packages:

Presentation Layer: This package contains the graphical user interface (GUI) of the system. It is responsible for displaying the menu, recording orders, handling payments, and managing employee activities. The package includes views, presenters, and other GUI-related components.

Business Logic Layer: This package contains the core application logic of the system. It is responsible for implementing the business rules for processing orders, handling payments, and managing employee activity. The package includes controllers, use cases, and other business logic components.

Data Access Layer: This package provides an interface to the database and is responsible for retrieving and persisting data. The package includes the database access layer, repositories, and other data access components.

Domain Layer: This package contains the entities that define the data model and the value objects that encapsulate the data required for loyalty calculations and other business logic.

Diagram

Description automatically generated

## Component and Deployment Diagrams

The client workstation node represents the user interface where the waiters can record orders and handle payments. The application GUI server node hosts the graphical user interface (GUI) of the application, which allows the user to interact with the system. The application server node hosts the application logic, which handles the processing of orders, notifications, and employee management. The database server node hosts the database, which stores the data related to orders, payments, employees, and loyalty program.

**Diagram

Description automatically generated**

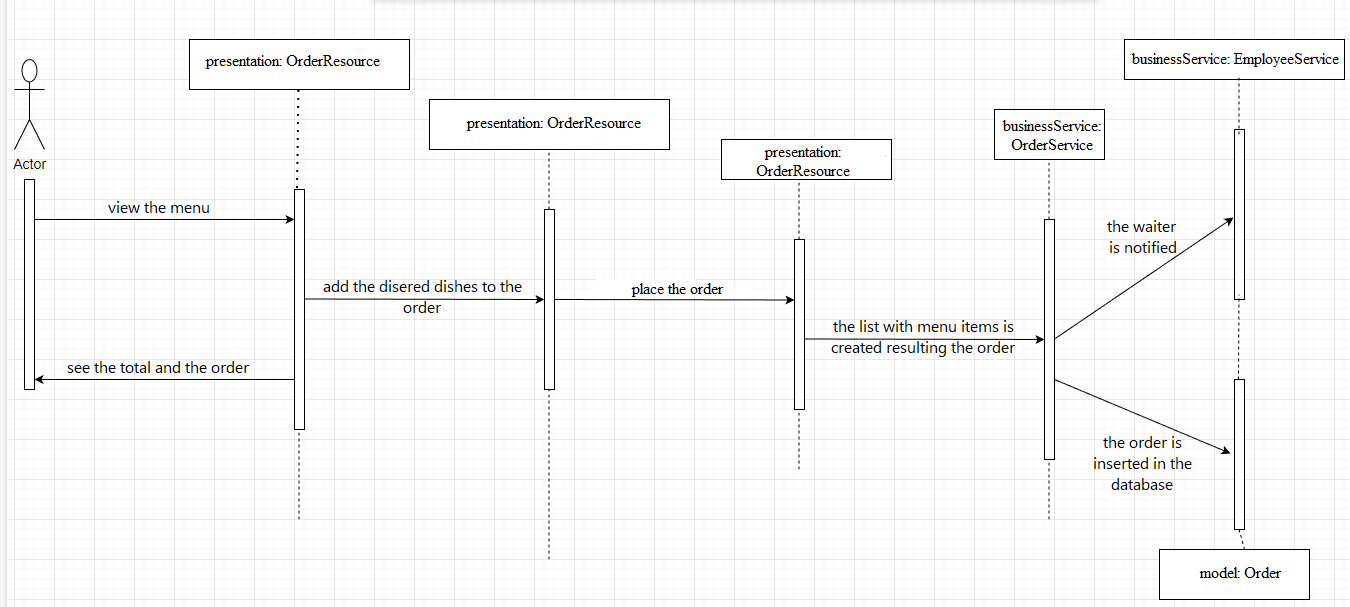
# Elaboration – Iteration 1.2

# Design Model

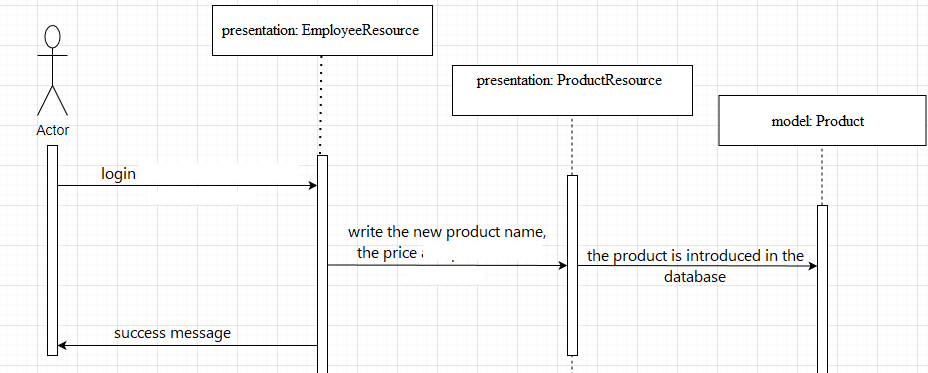
## Dynamic Behavior

Scenario 1: Client placing an order

Sequence diagram:



Scenario 2: Update a product (admin)



## Class Design

Observer Pattern:

The Observer pattern is a design pattern used in software development. It allows objects to be notified automatically when the state of another object changes. In this pattern, there is a one-to-many relationship between the objects: one object, known as the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes.

In Java, the Observer pattern is implemented using the Observer interface and the Observable class. The Observer interface has a single method, update(), which is called by the Observable class when there is a change in its state. The Observable class has two methods, addObserver() and deleteObserver(), to add or remove observers from its list of dependents.

Command Pattern:

The Command pattern is another design pattern used in software development. It is used to encapsulate a request as an object, thereby allowing the request to be handled by different objects without knowing the request's details. In this pattern, there is a separation between the object that invokes the command (the invoker) and the object that knows how to perform the command (the receiver).

In Java, the Command pattern is implemented using the Command interface, which defines the execute() method, and the Invoker and Receiver classes. The Invoker class invokes the command by calling its execute() method, and the Receiver class performs the command.

*Graphical user interface

Description automatically generated*

# Data Model

# Graphical user interface Description automatically generatedUnit Testing

*[Present the used testing methods and the associated test case scenarios.]*

# Elaboration – Iteration 2

# Architectural Design Refinement

*[Refine the architectural design: conceptual architecture, package design (consider package design principles), component and deployment diagrams. Motivate the changes that have been made.]*

# Design Model Refinement

## *[Refine the UML class diagram by applying class design principles and GRASP; motivate your choices. Deliver the updated class diagrams.]*

# Construction and Transition

# System Testing

*[Describe how you applied integration testing and present the associated test case scenarios.]*

# Future improvements

*[Present future improvements for the system]*

# Bibliography