Digital Ordering System - Software Requirements & Implementation Specifications

Use Case Diagram - User requirements

The Digital Ordering System is looking to set up the electronic website. The main aim of the system is providing a comprehensive and quick way to manage orders in a place where meals may be bought and eaten: cafeterias, bars, restaurants, bistros. Focuses on allowing customers to place orders online, using their devices which have access to the internet, without direct staff interaction. Also the system may work in collaboration with other built in systems in the eating establishments like POS (Point of Sale) but may also work separately. The system will reduce interaction with customers which will increase efficiency by 30% and reduce the amount of workers needed to be hired, which will reduce the expenses. This kind of system is an ideal solution for the eating establishments which experience a big amount of customers which may be hard to handle by small amounts of workers.

Main functionalities that the Digital Ordering System includes: make order Reservation and Add item to order, make In-restaurant order and Add item to order, Register Personal account, review history order, view and change personal data, review loyalty program points, view content of order, Verify dietary preferences, Review the menu. Change data of the items and ingredients, add items and ingredients, remove items and ingredients, change availability of the products and ingredients for the Administrator. Send receipts to the kitchen, mark customers arrival in case of reservation, decline the reservation, review order of the clients, review reservations, close orders of the clients and generate receipt, subtract loyalty points for the Employee.

Customers are able to view the food items on the web portal. Food items are placed in the list simulating physical menu logic, but with the ability to select specific categories. Three main categories are Food, Beverage and Sets of Meal. Food and Beverage include subcategories. Upon clicking on the menu item the system will display the content of the item and provide full information about it, especially name, price, description, ingredients, dietary preferences (for food) or if the item is alcohol (for beverage) and details of the Sets of Meal.

There are two options on how to create an order in the gastronomy hub. Users may access the web page remotely and choose an option to dine at the eating establishment which is called “make a reservation” in the system. Users will be required to specify the information about reservation: place (which restaurant), number of people, time and also identification verification which will demand the user to log in, register or specify name and phone number. After specifying needed information and quick system verification of free tables, the user will be redirected to the page where he is able to view the food items and now with the ability to add desired menu items to the order list. Reservation can’t be done earlier than two

hours ahead from the time of reservation. After customers arrive at the restaurant, they also have an ability to add new menu items to the order just by accessing the menu list using QR-code on the table, the procedure of menu item adding is the same as restaurant order.

In restaurant order, users simply scan the QR-code on the table which will lead to the page where viewing the food items and ability to add items is handled. No need for obligatory identification verification, just the optional possibility to log in or register in order to join a loyalty program.

There might be many orders for one table, this functionality covers the ability to split the bill, every customer can order items separately from the customers he has company with, just for accessing the menu list of the menu items from different devices.

And also after reviewing the menu item system gives users the option to add specific items with the specific desired quantity which will be possible to change any time in the order list. Once the item is added, the website redirects users to continue browsing the menu. Customers can also view the content of their order before confirming it, remove, change countinty. After that user may confirm order by simply clicking the button in the order summary.

Due to the fact that each customer may have their own account, but don’t have to. User can view the personal data in his account and also change this personal data, name, phone number, email. Also there is a possibility to view a history of orders and check the current status of loyalty points. Registration is simple, users are required to set the password and phone number or email instead.

There is no delivery realization in the system. This is simply done by third-party companies like UberEats, Glovo. These applications work separately from the main functionality of the Digital Ordering System.

Payments for a given order can either be done using the payment gateway or the in-currency loyalty points that a customer obtains after the completion of the order finalization process. The process of payment is done in the eating establishments, there is no online payment in the system. After finishing dining, the client follows up to the cashier, also the customer may ask the cashier to subtract loyalty points from the account.

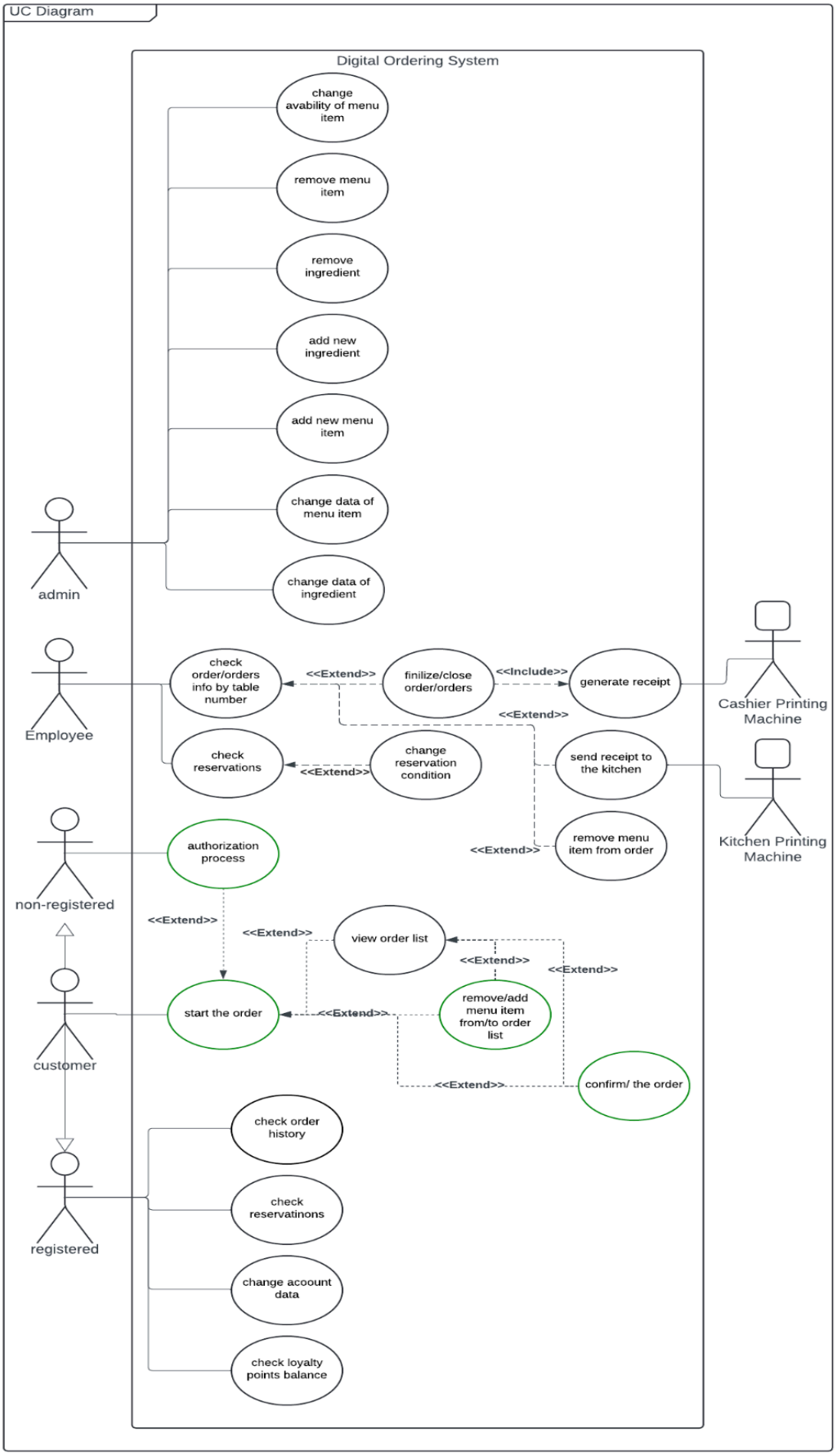
Administrators may change information about the data of ingredients and menu items. By modifying the name, description, composition of ingredients, number of food in set of food, types etc. Also there are possibilities to add and remove ingredients and menu items.Admin which is a manager can change availability of menu items.

In the case of the reservation order, the employee must Confirm customers arrival and send the receipt to the kitchen and may after that , as we already have the menu

items list of the client from the reservation process, where items were added to the order list, but there could be situation where employee can decline the reservation if the customer resign from it or didn’t arrive in order to free the table. Also employees can review the Reservation list.

Employees are obligated to review customers' orders which are on standby, there may be a couple of them for one table, after reviewing customer orders employees must send this order to the kitchen by selecting some of them. When a customer accesses the cashier desk, the employee can finalize the order and after that generate the receipt while also being able to subtract the loyalty points and subtract the amount of total price using loyalty points. One unit is one PLN.

Use case diagram



Use case scenario: “ Start the order ”

start the order - Use Case Scenario

Actor: User

Purpose and context: User wants to make an order in the website.

Assumption:

1. Website working properly and User accessed needed plantform
2. Physical one of the restaurant exists and tables has their QR codes accesiable.

Pre condition:

1. Basic: customer managed to get to the restaurant and managed to find table with QR-code
2. Reservation: customer accesed appropriate page online.

Basic flow of events:

1. Customer came to the physical restaurant and makes order by scuning QR-code.
2. System opens new order for the specific customer.
3. System displays list of menu items which is the Menu Page.

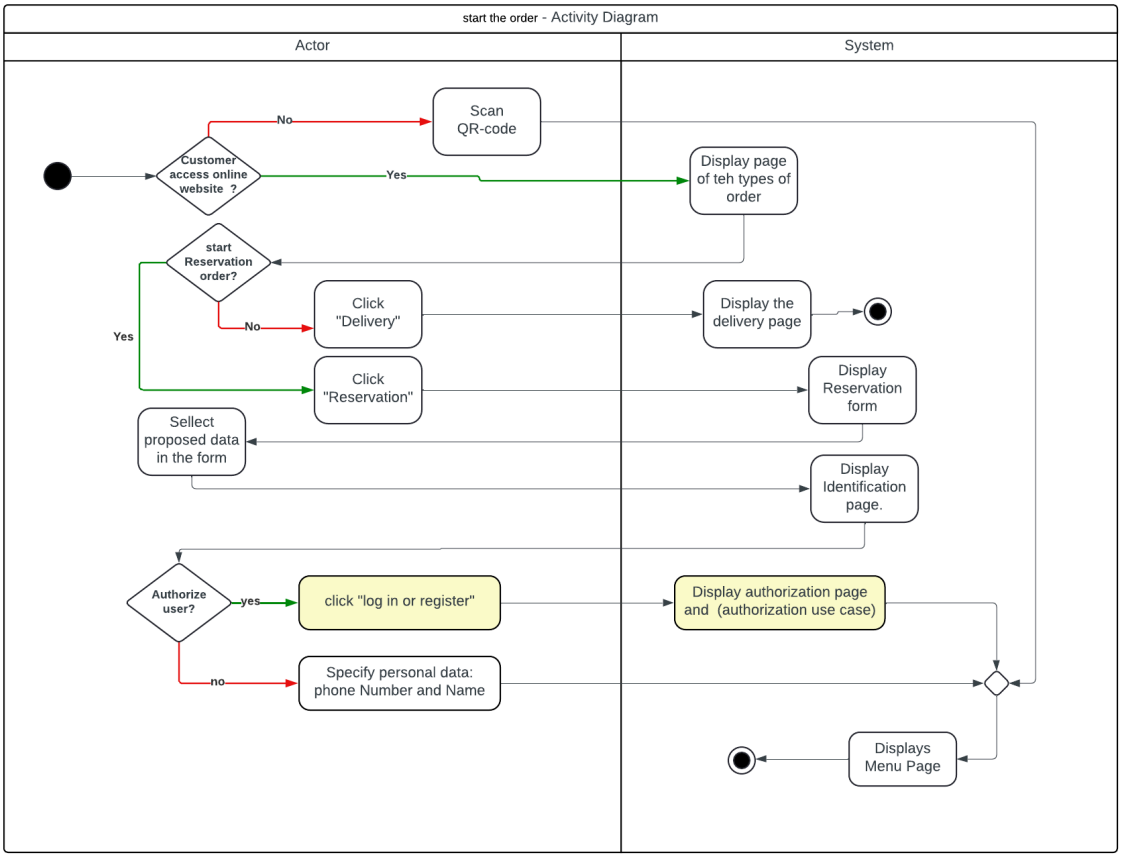
Alternative flow of events:

- **Customer makes reservation order using website**:
  - 1a1. Customer access the web site located in the web accessiable with link
  - 1a2. System displays information related to the type of the order. the website shows two buttons "Reservation" and "Delivery".
  - 1a3. Customer choose desired condition "Reservation"
  - 1a4. Website shows customer needed form to start Reservation process. Which includes: selector of restaurants, couner of people, input of time and date.
  - 1a5. Customer select proposed data in the form and specify information.
  - 1a6. System displays page of identification process which will ask customer to specify needed data: name and phone number. or simply to authorize.
  - 1a7. Customer specify name and phone number.
  - 1a7. Return to 3.
    - **Customer clicks "Delivery":**
      - 1a3a1. Website redirects customer to the page of the food delivery application like "uber eats", "volt", "glovo" etc. which are links to their services.
    - **Customer select to autorize**
      - 1a6a1. Customer click to "log in or register"
      - 1a6a2. System Dispays authorization page and start use case of authorization process ------>
      - 1a6a3. Return to 1a4.

Post condition:

1. The system displays the Menu Page with the list of menu items for selection.

Activity Diagram: “ Start the order ”



Use case scenario: “ Remove/add menu item from/ to order list”

Remove/add menu item from/to order list - Use Case Scenario

Actor: customer

Purpose and context: customer wants to add specific menu item to the order.

Assumption:
1. Website working properly and User accessed needed plantform
2. There are already some menu items in the database, so list of menu items is shown correctly.
3. All menu items are available. When menu items is unavailabe there is no ability to add this specific menu item to the order list.

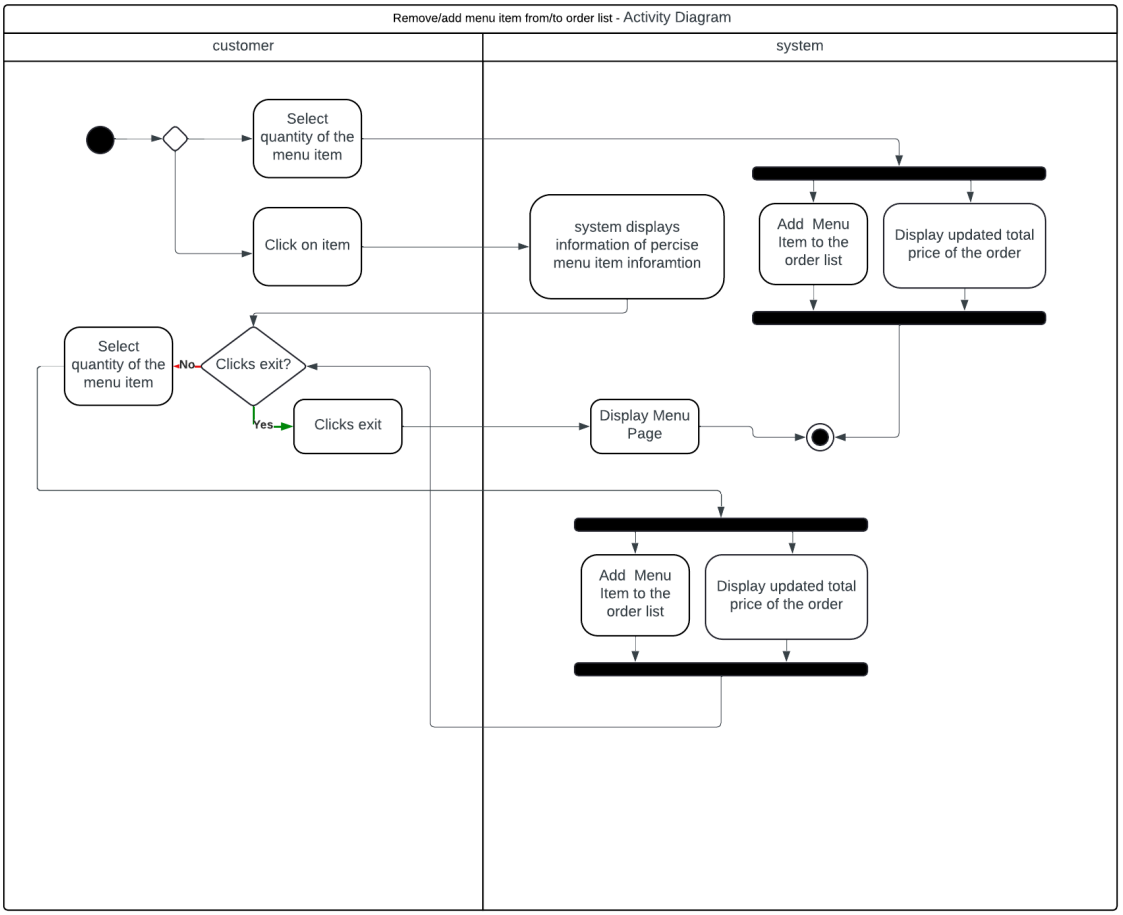
Pre condition:
1. Menu items are displayed on the Menu Page

Basic flow of events:
1. Customer select desired amount of menu item directly from menu item using counter with minus and plus buttons.
2. System adds menu item to the order list
3. System updates the order list price

Alternative flow of events:
- **Customer clicks directly on the menu item.**
  - 1a1. Customer clicks to the menu item they want to view.
  - 1a2. System Display information about menu item such as: full desctiptoin, diatary preference and also counter with minus and plus to choose desired amount of product.
  - 1a3. Customer choose desired amount of menu item
  - 1a4. System adds menu item to the order list
  - 1a5.System updates the order list price
  - 1a5. Customer press "close"
  - 1a6. System return to the Menu Page.

Post condition:
1. menu items are added to the order list or removed.

Activity Diagram: “ Remove/add menu item from/ to order list ”



Use case scenario: “Delete/Add Ingredient From/To MenuItem”

Delete/Add Ingredient From/To MenuItem - Use Case Scenario

Actor: Admin

Purpose and context: Admin wants to delete ingredient from the menuitem

Assumption:

1. Website working properly and admin accessed needed plantform

Pre condition:

1. Admin on the page with menuItems

Basic flow of events:

1. Admin selects MenuItem from the list
2. System returns Detailed info of the MenuItem
3. Admin chooses ingredient to delete
4. Approval message shown
5. Admin clicks “confirm“ button
6. System shows message that indicates succesfull deletion
7. System removes ingredient

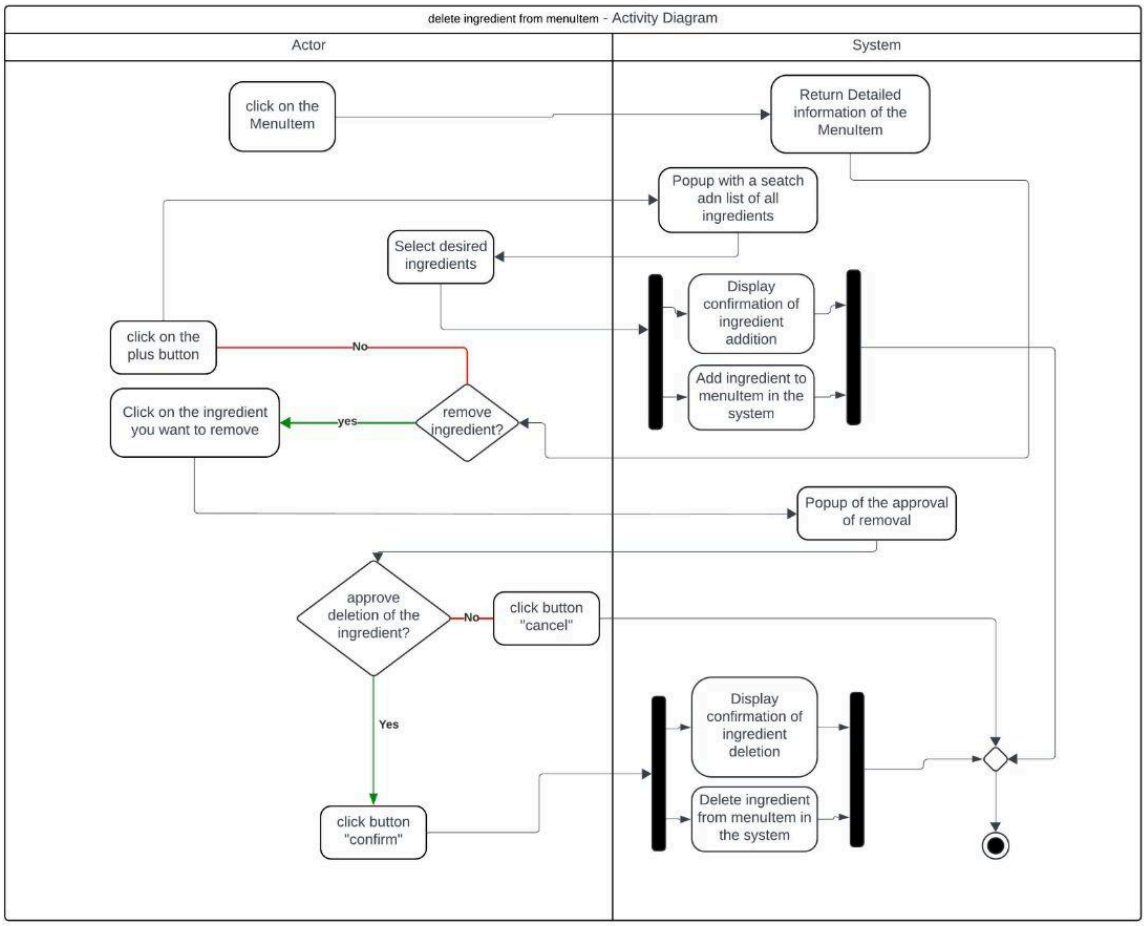
Alternative flow of events:

- **Chose to add ingredient:**
  - 3a1. Click button "plus" to add new ingredietn
  - 3a2. Popup with search and list of all ingredients
  - 3a3. select desired ingredients.
  - 3a4. System displays confirmation of addition
  - 3a5. System adds ingredient to MenuItem

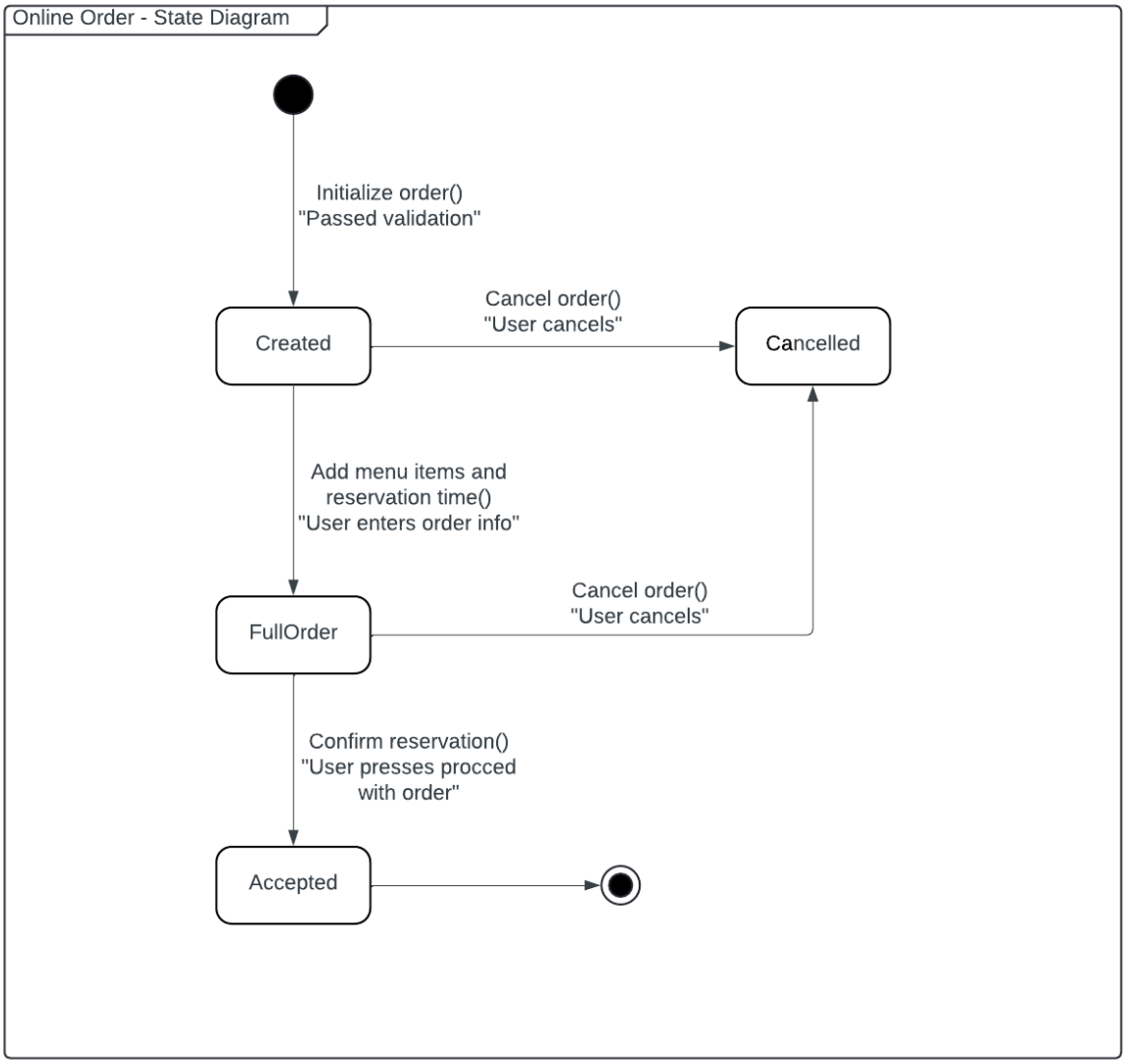
Post condition:

1. System deletes the ingredient and send changes to db.

Activity Diagram: “Delete/Add Ingredient From/To Menultem”



State Diagram



State Diagram - Design Description

The state diagram describes the lifecycle of an Online Order object. It outlines the possible states of an online order as it progresses through various stages, from creation to acceptance or cancellation.

1. **Created State**:
    - The Online Order object is initialized after passing validation checks.
    - The order remains in the "Created" state until the user either cancels it (Cancel order()) or proceeds to provide order details.
2. **FullOrder State**:
    - The order transitions to the "FullOrder" state when the user adds menu items and specifies a reservation time using the Add menu items and reservation time().
    - While in this state, the user has the option to cancel the order (Cancel order()), which moves the order to the "Cancelled" state.

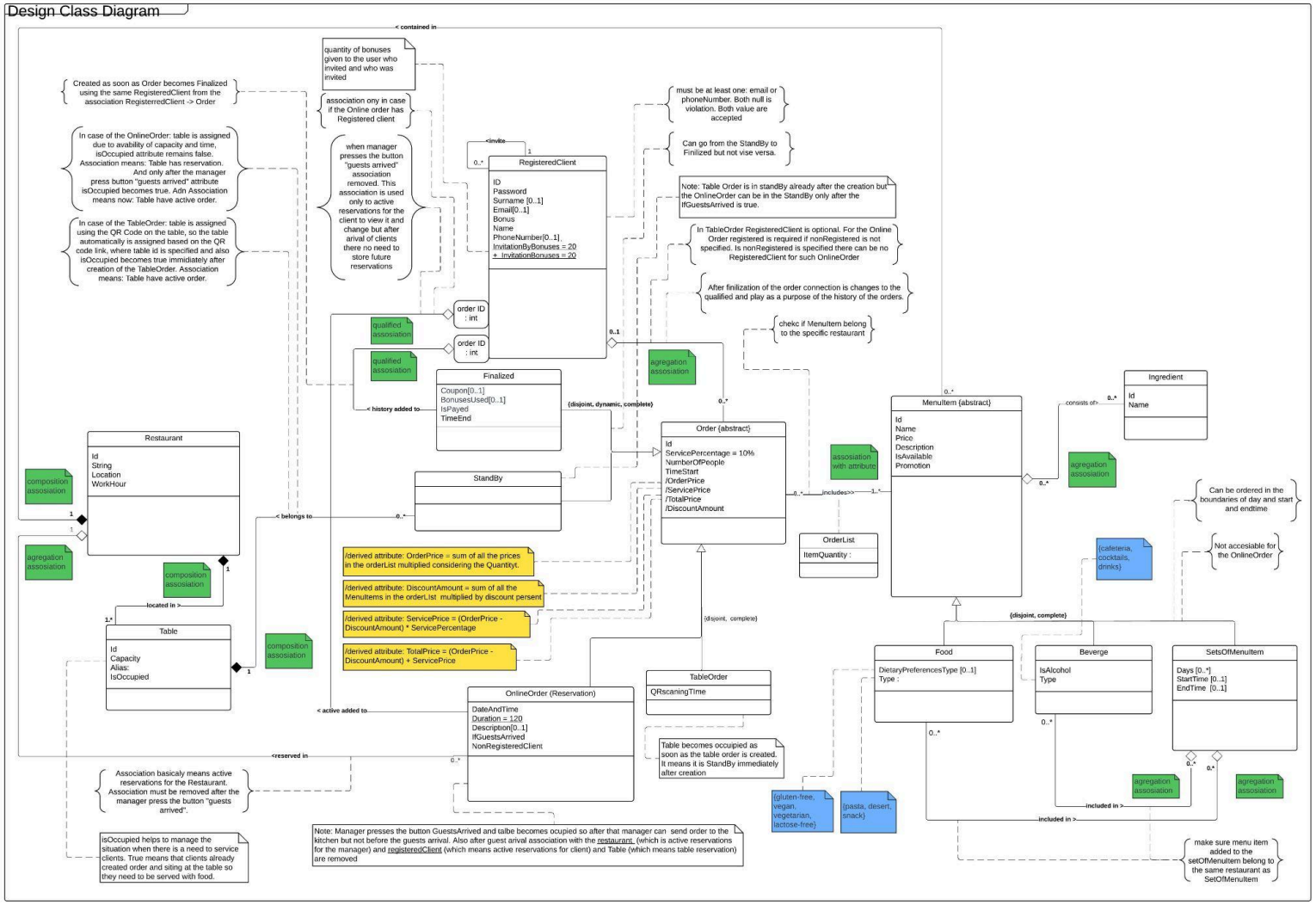
3. **Accepted State**:
    - If the user decides to confirm the reservation, they use the Confirm reservation() method by pressing "Proceed with order," which transitions the order to the "Accepted" state.
    - This is the final state of the order.
4. **Cancelled State**:
    - At any point before reaching the "Accepted" state, the user can cancel the order using the Cancel order() method. This action transitions the order to the "Cancelled" state, marking it as terminated.

The Online Order object has two possible outcomes:

- **Accepted**: The order is successfully completed and confirmed.
- **Cancelled**: The user terminates the order before confirmation.

This state diagram helps in understanding the flow and management of online orders in the system.

Class Diagram - Analytical



Analytical Class Diagram - Design Description

The above analytical class diagram visualizes the digital ordering system. Users can make reservation (online orders) or table orders (using QR code). It defines the relationships, attributes, and associations between entities in the system.

RegisteredClient

Registered clients have a unique ID, password, and optional surname. They must provide at least one of the following: Email: Used for communication.PhoneNumber: Mandatory for Online Orders.Registered clients can receive bonuses and place both online and table orders.

NonRegisteredClient

Only requires a name and optional phone number. Used for placing Online Orders without registering.

Restaurant

Includes details like Id, Name, Address, and WorkHour. Includes associations to manage: Table, MenuItems.

Table

Tables are associated with orders and can be accessed via QR scanning for table order placement. Attributes include: Capacity: Number of people it can accommodate. Alias: Identifier used within the restaurant. IsOccupied: Indicates whether the table is currently occupied. Associated with: Orders: Table becomes occupied upon order placement. Restaurant: Each table belongs to one specific restaurant. QRScanningTime: For tracking when a table is accessed via QR code for ordering.

Order

The base class for all orders with common attributes: Id, ServicePercentage: Default is 10%., OrderPrice: Sum of item prices multiplied by quantities., TotalPrice: OrderPrice + ServicePrice. NumberOfPeople: Number of guests. TimeStart: When the order begins.

Abstract class with specific types: **TableOrder**: Directly placed at a table. **OnlineOrder**: Requires reservation details (e.g., date, time, and guest arrival status).

Each order includes details like service percentage (10%), total price, and number of people. Finalized orders have additional details, such as applied coupons and bonuses used.

MenuItem (Abstract)

MenuItem class is abstract and includes: *Food* with dietary preferences (e.g. vegan, gluten-free) and type (e.g. pasta, desert). *Beverage*, which can include alcoholic or non-alcoholic options and type.

Relationships: *Ingredients*: Represents the components of menu items. Some items, like pre-made drinks, may not have ingredients. *SetsOfMenuItem*: Groups of menu items with defined limits (e.g., 2–4 foods, 1–2 beverages).

Promotions

Discounts associated with menu items. Attributes include: Id, DiscountPercentage, and Name. Min: Minimum discount is 1%. Max: Maximum discount is 99%. Promotions are applied directly to menu items and cannot exist independently.

Associations

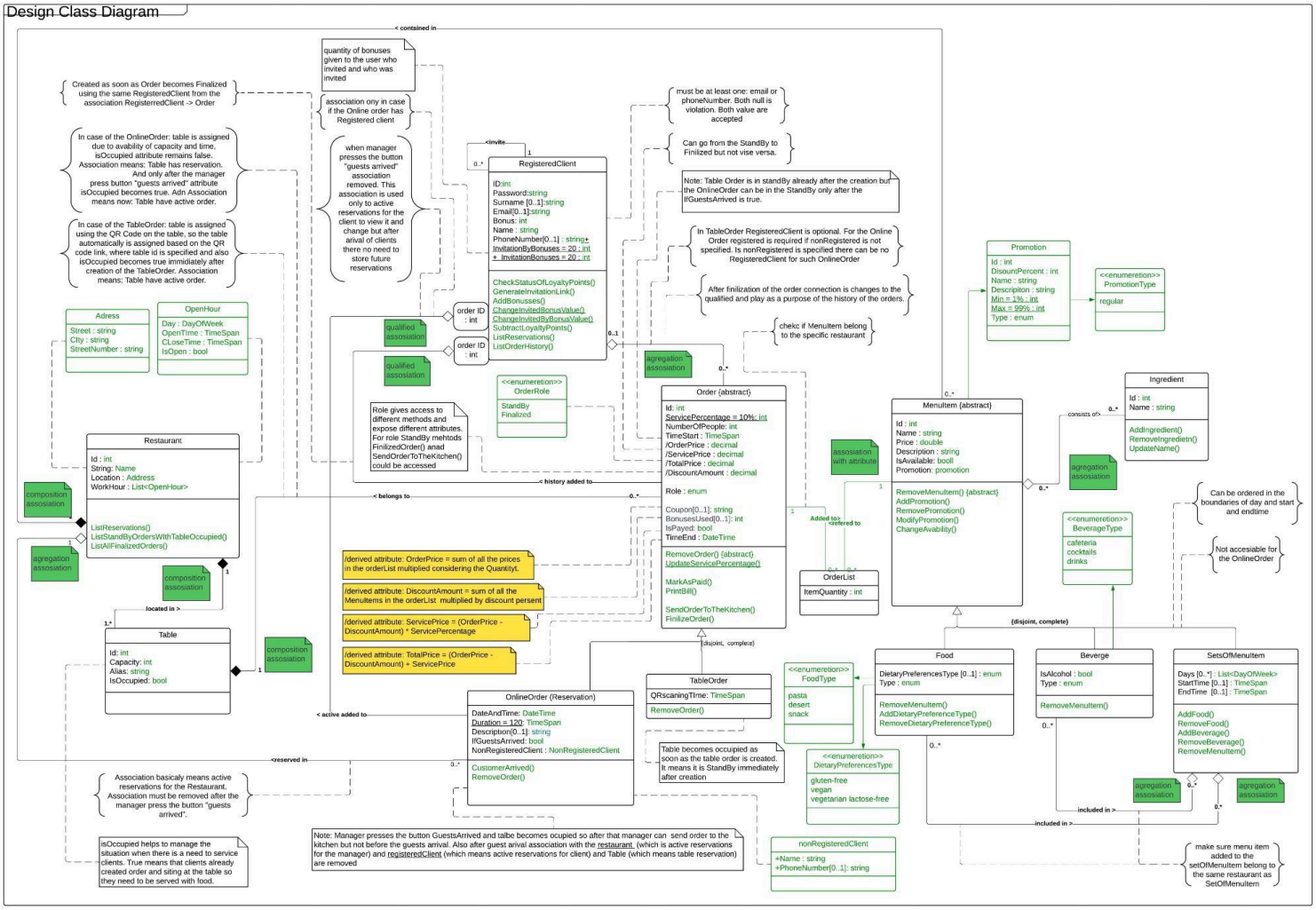
Association class: Link menu items to their ingredients and orders to items.

Qualified associations: Links OnlineOrder and Finalized to RegisteredClient.

Aggregation: Connects Order and NonRegisteredClient, MenuItem and Ingredient, Food+Beverages and SetsOfMenuItem.

Composition: Connects StandBy, TableOrder, OnlineOrder to Table. Table to Restaurant. OnlineOrder and MenuItem to Restaurant. That kind of relationship exists in our diagram because some tables can’t exist without tables they’re related to.

Class Diagram - Design (Initial version)



Design Class Diagram - Design Decisions

Attribute validation

Empty strings, null values, and invalid numbers are not allowed, even for nullable attributes. Validating attributes in the constructor is often better when initial values need to be checked immediately upon object creation. This ensures that even the initial state of the object is consistent and avoids creating invalid instances. For attributes that don’t frequently change, validation should occur in the constructor. For attributes that may be updated later, validation would be enforced in the setter.Setter validation is called "property with backing field" design pattern, where a private field (`_surname`) is used to store the value, and a public property (`Surname`) provides controlled access to it. Setter will simply control how values are assigned to `_surname`. In this example, the setter validates the input using the `ValidateSurname` method before assigning it. This prevents invalid data from

being stored in the attribute. This would help to ensure that all modifications passed to the attribute are going through a single validation point in the setter, maintaining data consistency and integrity.

```
private string? _surname;
public string? Surname
{
   get => _surname;
   private set
   {
       ValidateSurname(value);
       _surname = value;
   }
}
```

Mandatory attributes

Mandatory attributes must always be initialized with valid values to prevent nulls, empty states, or invalid numbers. This can be ensured by explicitly validating these attributes during object construction or through property setters. To uphold these requirements, checks must verify that; *Objects*: Must not be null to ensure that references are valid. *Strings or Lists*: Must not be null or empty to maintain data integrity. *Numbers*: Must be within an acceptable range (e.g., positive values for prices, percentages between 0 and 100). Using a validation method to centralize these checks ensures consistency across the codebase.

Each mandatory attribute should have corresponding validation rules. The validation method should be static because it does not depend on any instance-specific information.

```
private static void ValidateStringMandatory(string name, string
text)
{
   if (string.IsNullOrEmpty(name)) throw new
ArgumentException($"{text} cannot be null or empty");
}
```

Optional attributes

Optional attributes are those that may or may not have values assigned. When designing setters for optional attributes, it is essential to accommodate null values while ensuring that any non-null inputs meet the required constraints. Since simple types like `int` or `double` do not allow null values, the solution is to use nullable types, such as `int?` or `double?`, denoted with a question mark (`?`). This allows the attribute to hold either a value or null. Also Employing nullable types like `string?`, `int?`, or `double?` allows optional attributes to accommodate null values effectively.

When implementing the setter for an optional attribute, if a non-null value is provided, the setter should validate the input against appropriate constraints, such as ensuring a string or list is not empty and meets length requirements or that a number falls within an acceptable range.

```
private static void ValidateStringOptional(string? value, string
propertyName)
{
   if (value == string.Empty)  throw new
ArgumentException($"{propertyName} cannot be empty");
}
```

Object initializers should be used to handle scenarios where optional attributes are or aren't provided during object creation.This ensures that optional attributes are handled gracefully without imposing unnecessary constraints during object creation.

```
public OnlineOrder(int numberOfPeople, string? description = null)
{
   Description = description;
}
```

Complex attributes

Complex attributes represent attributes that consist of multiple related values or involve intricate logic, such as an address or a time schedule. Instead of representing these attributes with primitive types, it is often better to encapsulate them in separate classes. For example, in a `Restaurant` class, attributes like `Location` could be modeled as separate classes. This modular approach ensures that each complex attribute has its own well-defined responsibilities and encapsulation. And of course there could be unique validation rules inside the complex attributes which respond to the consistency and prevent creation of invalid instances. Such attributes are rarely changed, so there must be consideration of making the class immutable to ensure its consistency throughout its lifecycle. If there is a strong need to update data within a complex attribute, the editing should be done through methods defined in the class that encapsulates the attribute.

```
public class Address
{
   private string _street;

   public string Street {
       get => _street;
       private set
       {
           // validation logic
           _street = value;
       }
```

```
    }

    public Address(string street)    {
        Street = street;    }

    public override string ToString()    {
        return $"adress: {Street}";}

    public Address Clone()    {
        return new Address(Street, City, StreetNumber);}
}
```

Multi-value attributes

Represent collections of data associated with a single object, such as a set of working days for a restaurant. Collections should be exposed in a read-only manner to prevent direct modification by external code. This can be achieved by returning copies of the collection or exposing it as a read-only interface. Although setters for collections are not mandatory.

```
private List<OpenHour> _openHours;
public IReadOnlyList<OpenHour> OpenHours =>
_openHours.AsReadOnly();
```

Also When adding, updating, or removing elements in the collection, there should be validation for each item to ensure it meets the defined constraints.

```
public void UpdateOpenHours(List<OpenHour> newOpenHours)
{
    ValidateWorkHours(newOpenHours); // validation logic
    _openHours = newOpenHours;
}
```

Class attributes

Also known as static attributes, represent shared properties or data that belong to the class itself rather than any specific instance. Is declared as `static` fields within the class. This ensures that the attribute is associated with the class rather than individual instances. All instances of the class will reference the same shared attribute, making it ideal for common or globally applicable data

```
private static double _service = 10;
```

If the attribute is mutable, there should be controlled access through a static getter and setter.If the attribute is immutable, it must be declared as `readonly` and assigned its value at declaration or in a static constructor. This prevents further modifications, ensuring consistency.

```
public static double Service
{
   get => _service;
   private set
   {
       ValidateService(value);
       _service = value;
   }
}
```

Derived attributes

Values calculated based on other attributes within the class, providing computed data rather than directly stored values. Method within the class that calculates the derived attribute based on the current state of other attributes should be created such as MakeCalculationOfPrice(). While derived attributes traditionally do not have setters to maintain their computed nature, in some cases approach involves recalculating these values whenever method within class is called, effectively updating attributes, based on the latest data for example `MenuItem`. And in this case setters are used to update the computed values, whenever recalculations are necessary due to changes in underlying attributes or associations.

```
public double OrderPrice { get; private set; } = 0.0;
```

Tagged value

Refer to attributes or properties associated with specific tags or labels, influencing behavior or categorization within the system. These values provide additional context or classification beyond standard attributes, enhancing flexibility and organizational capabilities. Tagged values are realized through attributes like `_dietaryPreferences` and `_foodType`, which are tagged with enums (`DietaryPreferencesType` and `FoodType`) to categorize or classify data based on specific criteria such as dietary preferences or food types. Furthermore, due to the usage of enumerations the design class diagram includes the notations of those classes, which transform the previously used note notation for tagged values to an actual class related to the given attribute.

Associations

In the absence of ORM (Object-relational mapping), reverse connections are implemented across all associations. This approach ensures that each end of the association comprehensively acknowledges and maintains the relationship, promoting data integrity and preventing logical inconsistencies. When creating a relationship between two objects (MenuItem ↔ Ingredient), it’s critical to ensure both sides acknowledge the relationship. Without bidirectional consistency, one object might "think" it is connected while the other doesn't. For example: If AddIngredient

adds a reference in _ingredients but AddMenuItemToIngredient does not update _ingredientInMenuItems, the relationship is incomplete and may lead to bugs.

It is also crucial to handle infinite recursion in case of the reverse connections; methods check whether the specific object is already associated before adding it. This prevents redundant associations and potential stack overflow errors.

General association implementation

Many-to-many association: should be implemented using a reference between objects (no identifiers) Upon the creation of a reference connecting one object to another, a reverse connection must be created. In case of lists - keep them private. Getters should return a copy of the lists to maintain encapsulation - there shouldn't be any setters to set a list, it must rely on using the add and remove methods to add objects to a given list. Method for creating the connection between the objects. Said method should automatically set the reverse reference. The same logic of reverse connection when adding must be applied for removing and editing methods. Also it should take into account any necessary exceptions.

```
// association
private List<Ingredient> _ingredients = new();
public List<Ingredient> Ingredients => [.._ingredients];
public void AddIngredient(Ingredient ingredient)
{
   if(ingredient == null) throw new ArgumentNullException();
   if (!_ingredients.Contains(ingredient))
   {
       _ingredients.Add(ingredient);
       ingredient.AddMenuItemToIngredient(this);
   }
}

// reverse association
private List<MenuItem> _ingredientInMenuItems = new();
public List<MenuItem> IngredientInMenuItems => [.._ingredientInMenuItems];
public void AddMenuItemToIngredient(MenuItem menuItem)
{
   if(menuItem == null) throw new ArgumentNullException();
   if (!_ingredientInMenuItems.Contains(menuItem))
   {
       _ingredientInMenuItems.Add(menuItem);
       menuItem.AddIngredient(this);
   }
```

Similarly to the adding methods Removing method should work. So a method for removing the connection between the objects should automatically remove the reverse reference as well. And also should take into account any necessary exceptions.

A method for updating (editing) certain connections. Said method should ensure that the previous object referred is removed before adding the new object reference.

```
public void UpdateIngredients(List<Ingredient> ingredients)
{
   if(ingredients == null || ingredients.Count == 0) throw new
ArgumentNullException();
   if(_ingredients.Count > 0)
      foreach (var ingredient in _ingredients)
            RemoveIngredient(ingredient);
   foreach (Ingredient ingredient in ingredients)
AddIngredient(ingredient);
}
```

In constructors there should be an option to add all the references of a given associations (it should not be set using a public setter)

```
if (ingredients =≠ null) UpdateIngredients(ingredients);
```

One-to-many association: A method should be provided to establish the association between two objects. This method must ensure that an object is associated with only one other object at a time, maintaining bidirectional consistency by invoking the corresponding method on the related object. Additionally, a method should exist to add an object to the list while performing necessary checks, such as validating for null values to prevent invalid associations and verifying that the object is not already associated with another entity to avoid conflicts.

Order.cs:

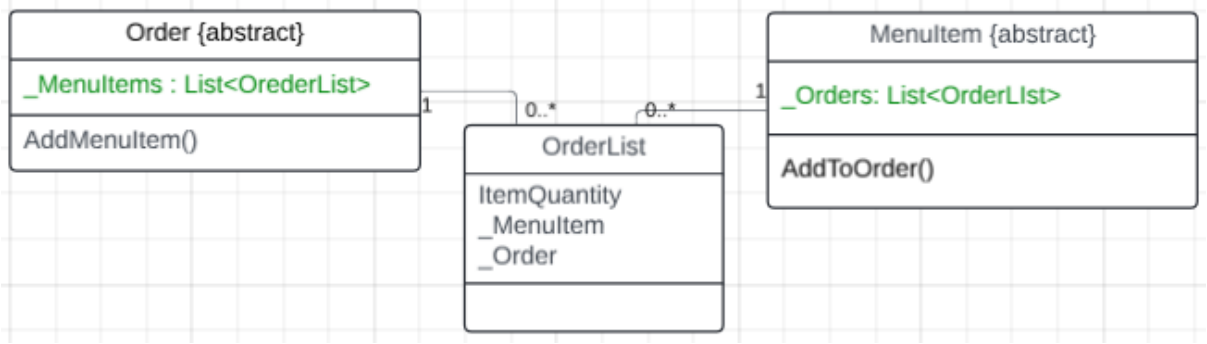
```
private RegisteredClient _registeredClient;
public RegisteredClient RegisteredClient => _registeredClient;
public void AddRegisteredClient(RegisteredClient registeredClient)
{
   if(registeredClient == null) throw new
NullReferenceException("RegisteredClient is null in the AddRegisteredClient
method");
   if(_registeredClient == null){
       _registeredClient = registeredClient;
       registeredClient.AddOrder(this);
   }
}
```

RegisteredClient.cs:

```
private List<Order> _orders = [];
public void AddOrder(Order order)
{
   if (order == null) throw new ArgumentException($"Order can be null in
the AddOrder() Registered client");
   if(order.RegisteredClient != null && order.RegisteredClient != this)
throw new ArgumentException($"Order {order.Id} has already client asigned
to it: {order.RegisteredClient.Id}");
   if (!_orders.Contains(order))
   {
       _orders.Add(order);
       order.AddRegisteredClient(this);
   }
}
```

Association class/attribute

association class serves to capture additional information or behaviors associated with a relationship between two main classes. For example, in the context of `MenuItem` and `Order`, the `OrderList` class captures details such as quantity and specific associations between `MenuItem` and `Order`. Unlike basic associations that directly link two classes, the association class (`OrderList`) manages the relationship by holding references to both `MenuItem` and `Order`, along with additional attributes like `Quantity`.



Methods like `AddMenuItem` and `AddToOrder` within `MenuItem` and `Order` classes respectively instantiate `OrderList` objects, ensuring that all necessary references and attributes are properly initialized. These methods enforce bidirectional consistency by setting up reverse connections between associated objects (`MenuItem` ↔ `OrderList` ↔ `Order`). Also When a new `OrderList` object is instantiated, it initializes references to both `MenuItem` and `Order` using methods like `AddOrderList` and `AddToOrder` ensuring bidirectional connections are

established immediately. So the responsibility for managing the association is centralized within the `OrderList` class. `OrderList` constructor handles this, adding itself to the `_menuItems` list in the `Order` and adding itself to the `_orders` list in the `MenuItem`.

Private lists (`_menuItems` in `Order` and `_orders` in `MenuItem`) manage instances of `OrderList`. These lists are accessed through getter methods (`MenuItems` and `Orders`), which return copies of the lists to maintain encapsulation. This prevents external modifications that could compromise data integrity.

Validation checks within methods (`AddMenuItem`, `AddToOrder`, etc.) ensure that null references are handled appropriately (e.g., throwing `ArgumentNullExceptions`) to maintain robustness and prevent unexpected behaviors.

Order.cs:

```
private List<OrderList> _menuItems = [];
public List<OrderList> MenuItems => [.._menuItems];
public void AddMenuItem(MenuItem menuItem, int quantity = 1)
{
   if(menuItem == null) throw new ArgumentNullException($" {this}: MenuItem
in AddMenuItem can't be null");
   new OrderList(menuItem, this, quantity);
}
public void AddOrderList(OrderList orderList)
{
   if (orderList == null) throw new ArgumentNullException($" {this}:
OrderList can't be null in AddOrderList()");
   if (orderList.Order != this) throw new ArgumentException($"You trying to
add the wrong orderList to the Order in AddOrderList()");
   if (!_menuItems.Contains(orderList)) _menuItems.Add(orderList);
}
```

MenuItem.cs:

```
private List<OrderList> _orders = [];
public List<OrderList> Orders => [.._orders];
public void AddToOrder(Order order, int quantity = 1)
{
   if(order == null) throw new ArgumentNullException($" {this.Name}: Order
in AddToOrder can't be null");
   new OrderList(this, order, quantity);
}
public void AddOrderList(OrderList orderList)
```

```
{
   if (orderList == null) throw new ArgumentNullException($" {this}:
OrderList can't be null in AddOrderList()");
   if (orderList.MenuItem != this) throw new AggregateException($"you are
trying to add the wrong orderList to the MenuItem in AddOrderList()");
   if (!_orders.Contains(orderList)) _orders.Add(orderList);
}
```

OrderList.cs:

```
public int Quantity { get; private set; }
public MenuItem MenuItem { get; private set; }
public Order Order { get; private set; }
private void AddMenuItemToOrderList(MenuItem menuItem)
{
   if (menuItem == null) throw new ArgumentNullException("MenuItem cannot
be null in AddMenuItemToOrderList()");
   MenuItem = menuItem;
   menuItem.AddOrderList(this); // reverse call
}
private void AddOrderToOrderList(Order order)
{
   if (order == null) throw new ArgumentNullException("Order cannot be null
in AddOrderToOrderList()");
   Order = order;
   order.AddOrderList(this); // reverse call
}
public OrderList(MenuItem menuItem, Order order, int quantity = 1)
{
     AddMenuItemToOrderList(menuItem);
     AddOrderToOrderList(order);
     Quantity = quantity;
}
```

Composition associations:

The (part) cannot exist without the (whole). The lifecycle of a part object is tightly coupled with the whole. So the part can’t be independent. When the whole is deleted, all associated part objects are also deleted to prevent orphaned instances. When implementing a composition association, it is essential to enforce that the "part" object cannot exist without an associated "whole." This is achieved by establishing the relationship during the initialization of the "part" object. For example, a constructor can be designed to accept the "whole" object as a parameter, ensuring that the association is created immediately. This guarantees that the "part" is always connected to a valid "whole."

The "part" cannot be shared between multiple "whole" objects, whereas in basic association, relationships may allow multiple connections.. A "part" object can only belong to one "whole" object at a time. To enforce this, validation logic must be implemented during the association process to ensure that a given "part" is not already linked to another "whole." If such a relationship is detected, an exception should be thrown, preventing the creation of conflicting associations.

Additionally, The deletion of the "whole" automatically triggers the deletion of the associated "parts," which is not a requirement in basic associations. This can be achieved by iterating through the "part" objects, removing their references to the "whole," and clearing their relationships entirely. This ensures no orphaned "part" objects remain, maintaining a consistent state in the system.

Table.cs:

```
public Table(Restaurant restaurant)
{
   AddToRestaurant(restaurant);
}

private Restaurant _restaurant;
public Restaurant Restaurant ⇒ _restaurant;
private void AddToRestaurant(Restaurant restaurant)
{
   if(restaurant is null) throw new ArgumentNullException($"Restaurant
can't be null in AddRelationToRestaurant()");
   _restaurant = restaurant;
   restaurant.AddTable(this);
}
private void RemoveFromRestaurant()
{
   _restaurant.RemoveTable(this);
   _restaurant = null;
}

public void DeleteTable()
{
   if (_tables.Contains(this))
   {
       RemoveFromRestaurant();
   }
}
```

Restaurant.cs:

```
private List<Table> _tables = [];
```

```
public List<Table> Tables => [..._tables];

public void AddTable(int capacity, string? alias = null, string?
description = null)
{
   new Table(this, capacity, description, alias);
}
public void AddTable(Table table)
{
   if(table == null) throw new ArgumentNullException($"Table is null");
   if(table.Restaurant != this) throw new ArgumentException();
   if(!_tables.Contains(table)) _tables.Add(table);
}
public void RemoveTable(Table table)
{
   if (table == null) throw new ArgumentNullException($"Table is null");
   if (table.Restaurant != this) throw new ArgumentException($"Table
doesn't belong to this restaurant");
   if (_tables.Contains(table))
   {
       _tables.Remove(table);
       table.DeleteTable();
   }
}
public static void RemoveRestaurant(Restaurant restaurant)
{
   foreach (var table in restaurant.Tables)
   {
       restaurant.RemoveTable(table);
   }
   _restaurants.Remove(restaurant);
}
```

Qualified associations

Qualified associations introduce a more structured way of managing relationships between objects by using a qualifier (key) to identify and access related instances. In a qualified association, a key is introduced to identify each related object. This key is typically a unique attribute of the associated object, such as an ID. The qualifier is used to store and retrieve related objects efficiently within a collection, such as a dictionary. The "whole" object maintains a collection where the keys are qualifiers, and the values are the associated "part" objects. When adding a "part" object to the "whole," the method requires both the "part" and its qualifier as inputs. Before creating the relationship, validation ensures that: the "part" object is not null, The qualifier is valid and unique within the collection, the "part" does not already belong

to the correct "whole.". To remove a "part" object, the key is used to locate and delete the association. Validation ensures that the key exists in the collection and that the "part" is correctly associated with the "whole" before removal. qualified associations, maintaining bidirectional relationships is critical. When a "part" is added to or removed from a "whole," the reverse association in the "part" object is updated simultaneously. This ensures consistency and prevents orphaned or invalid associations.

```
//association with OnlineOrder
private Dictionary<int, OnlineOrder> _onlineOrders = [];
public Dictionary<int, OnlineOrder> OnlineOrders => _onlineOrders;
public void AddOnlineOrder(OnlineOrder onlineOrder)
{
   if(onlineOrder == null) throw new ArgumentNullException();
   if(onlineOrder.RegisteredClient == null) throw new
ArgumentException($"you can't add the online order directly to the user, it
is done automatically after creation of the online order");
   if (onlineOrder.RegisteredClient != this) throw new
AggregateException($"online order you are trying to add belong to different
client");
   if (!_onlineOrders.ContainsKey(onlineOrder.Id))
   {
       _onlineOrders[onlineOrder.Id] = onlineOrder;
       onlineOrder.AddOnlineOrderToRegisteredClient(this);
   }
}
public void RemoveOnlineOrder(OnlineOrder onlineOrder)
{
   if(onlineOrder == null) throw new ArgumentNullException();
   if (onlineOrder.RegisteredClient != this) throw new
ArgumentException($"you are trying to remove online order which doesn't
belong to this client");
   if (_onlineOrders.ContainsKey(onlineOrder.Id))
   {
       _onlineOrders.Remove(onlineOrder.Id);
       onlineOrder.RemoveOrder();
   }
}
//association with Registered client stores online orders (REVERSE)
private RegisteredClient _registeredClientMadeOnlineOrder;
public RegisteredClient RegisteredClientMadeOnlineOrder =>
_registeredClientMadeOnlineOrder;
public void AddOnlineOrderToRegisteredClient(RegisteredClient
registeredClient)
{
```

```
   if(registeredClient == null) throw new ArgumentException($" registered
client: in the AddOnlineOrderToRegisteredClient method in OnlineOrder cant
be null");
   if (_registeredClientMadeOnlineOrder == null)
   {
       _registeredClientMadeOnlineOrder = registeredClient;
       registeredClient.AddOnlineOrder(this);
   }
}
private void RemoveOnlineOrderFromRegisteredClient()
{
   _registeredClientMadeOnlineOrder.RemoveOnlineOrder(this);
   _registeredClientMadeOnlineOrder = null;
}
```

Qualified associations differ from basic associations by introducing a key to identify related objects, whereas basic associations often rely on lists or sets without unique identifiers. Accessing related objects is more efficient in qualified associations because keys allow direct lookup, unlike basic associations, which may require searching through a collection. Qualified associations enforce uniqueness of the key within the collection, ensuring that each "part" can only be associated with one "whole" under a specific key. Basic associations lack this level of structured enforcement.

Reflex Association

Reflexive associations describe relationships within the same class, where instances of a class can be associated with other instances of the same class. In reflexive associations, both the "source" and "target" ends of the relationship are managed within the same class. This requires additional attributes and methods. Reflexive associations require validation to prevent invalid relationships. Reflexive associations involve dual roles for instances of the same class (e.g., inviter and invited). This requires additional logic to differentiate and manage these roles. Reflexive associations introduce more complex validation rules, such as ensuring that an object does not form circular or invalid relationships with itself.

```
// association with Registered client
private RegisteredClient? _invitedBy = null;
public RegisteredClient? InvitedBy => _invitedBy;
private void AddInvitedBy(RegisteredClient registeredClient)
{
   if (registeredClient == null) throw new
ArgumentNullException($"Parameter {nameof(registeredClient)} cannot be
null");
```

```
    if (registeredClient == this) throw new ArgumentException($"You are
trying to  invite yourself");
    if (_invitedBy == null)
    {
        _invitedBy = registeredClient;
        registeredClient.AddInvited(this);
    }
}
// association with Registered client (REVERSED)
private List<RegisteredClient> _invited = [];
public List<RegisteredClient> Invited => _invited;
private void AddInvited(RegisteredClient registeredClient)
{
    if (registeredClient == null) throw new
ArgumentNullException($"Parameter {nameof(registeredClient)} cannot be
null");
    if (registeredClient.InvitedBy != null) throw new
ArgumentException($"you are trying to add an invited client");
    if (registeredClient == this) throw new ArgumentException($"You are
trying to  invite yourself");
    if (!_invited.Contains(registeredClient))
    {
        _invited.Add(registeredClient);
        registeredClient.AddInvitedBy(this);
    }
}
```

Representation of the "source" and "target" ends of the reflexive association are:`_invitedBy` represents the instance that invited this object and `_invited` represents the list of instances invited by this object. Methods are implemented to establish and manage the relationship bidirectionally: `AddInvitedBy()` links the current instance to another as the inviter, `AddInvited()` establishes the reverse link, where the current instance invites another. Validation must state that an instance cannot invite itself and an instance already invited by another should not be reassigned. Both methods must be private as the addition to the both methods must be triggered after the creation of the object.

Inheritances

Multi-Aspect Inheritance

Normal inheritance for one aspect of the design, enabling a natural and straightforward implementation of shared functionality across related roles. For the

other aspect, base class flattening is implemented, allowing the flexibility to mix in or assign specific behaviors without deep inheritance hierarchies. Main object expose functionality for each role through delegation, ensuring a clean separation of concerns and an intuitive interaction surface.

For example multi-aspect inheritance and flattening strategy might be used. Common attributes of `FinalizedOrder` and `StandByOrder` were moved to the `Order` class. Enum to represent the role (`StandBy` or `Finalized`) were added in order to ensure immutability of this role. Methods in the `Order` class must be created to allow orders to change the role from `StandBy` to `Finalized`.

Dynamic analysis notes

Based on the diagrams presented earlier, a dynamic analysis was conducted. The analysis highlighted the need to expand the design class diagram to address newly identified system requirements and behaviors. The final design class diagram, reflecting all the implemented changes, is presented below.

The changes were implemented to address three specific scenarios: "Start Order," "Remove/Add Menu Item," and "Remove Ingredients." Below are the methods identified and introduced for each scenario:

Start Order

The following new methods are introduced to support the dynamic behavior of the "Start the Order" use case::

- VerifyScanedQRCoder(): Captures and processes the QR code data to identify the table and restaurant. This method verifies the QR code, extracts table and restaurant information, and verifies that order will be initiated with the associated table.
- ListMenuForTableOrder(): Displays the menu items available for selection. Retrieves and renders the list of menu items based on the restaurant's menu for the table order without setOfMenuItems.
- ListMenuForOnlineOrder(): Displays the menu items available for selection. Retrieves and renders the list of menu items based on the restaurant's menu for the table order with setOfMenuItems.

Remove/Add Menu Item From/To Order List

The following methods are essential for implementing the selected desired menu items before the confirmation of the order process. Use cases involve customers managing their order list by adding or removing menu items.

- AddMenuItemToOrder(): Adds a specific menu item to the order list. Checks the availability of the menu item and adds the selected quantity to the customer's current order. Updates the order list and price accordingly.
- RemoveMenuItemFromOrder(): Removes a specific menu item from the order list
- DecrementQuantityInOrderList(): Deducts the specified quantity of the menu item from the order. If the quantity reaches zero, remove the menu item from the order entirely.
- MakeCalculationOfPrice(): Recalculates the total price of the order after adding menu items. Sums up the prices of all menu items in the order and applies any discounts or promotions if applicable.

- AdjustPriceOnRemoval(): Recalculate the total price of the order after removing menu items.
- AdjustPriceOnDecrement(): Recalculate the total price of the order after decrementation of menu items.
- DisplayMenuItemDetails(): Displays detailed information about a selected menu item. Retrieves details such as description, dietary preferences, and price from the database.

Confirm Order

The following new methods are crucial for finalizing the order from the client side and create an order:

- AddTable(): Finds free tables in the system and assigns a random available table to an online order. Queries the system for tables marked as free. Ensures the table is suitable for the party size and order type.Randomly selects and assigns a table to the orde
- IsRestaurantOpen(): Verifies if the restaurant is open before assigning a table or creating an order. Check the current time against the restaurant's operating hours. Ensures the restaurant is open before proceeding with table assignment or order creation.
- IsAvailableForOnlineOrder(): Checks if a table is available for online orders. Ensures tables flagged for online orders are free. Filters out tables that are reserved for in-person dining or already occupied.

Delete/Add Ingredient From/To MenuItem

This use case involves the Admin removing an ingredient from a specific MenuItem. The dynamic analysis identifies new methods necessary for implementing this functionality:

- DisplayMenuItemDetails(): Allows the Admin to choose a specific MenuItem from the list for editing. Retrieves the details of the selected MenuItem, including its associated ingredients.
- ListAllIngredients(): Displays the list of ingredients for the selected MenuItem.

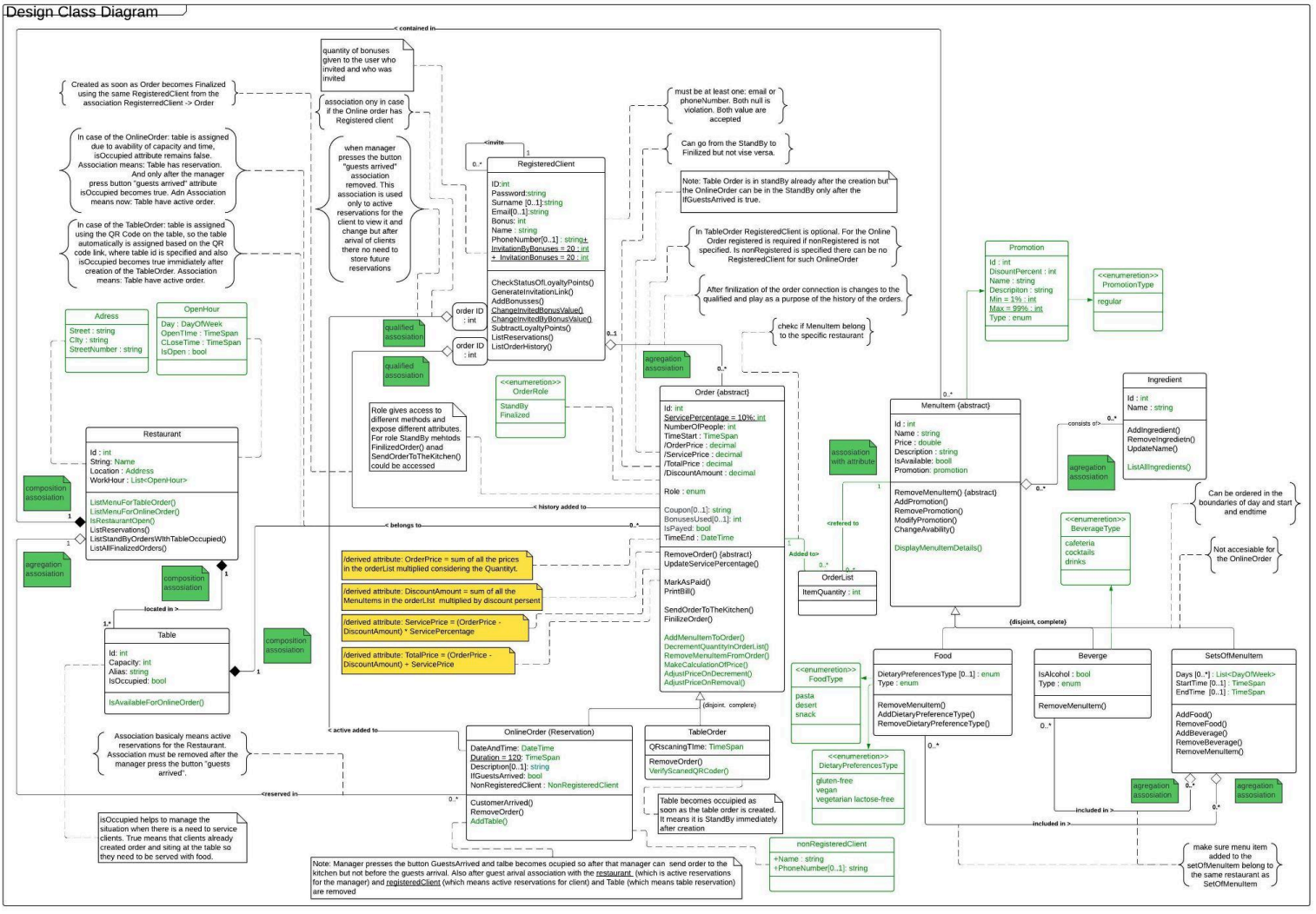
Persistence Considerations

For the scenarios "Delete/Add Ingredient From/To MenuItem," "Remove/Add Menu Item From/To Order List," and "Start Order," the persistence layer of the system leverages Entity Framework Core, allowing operations to interact seamlessly with the database. Methods such as AddTable(). ListMenuForTableOrder() and others might be implemented within a service class, providing access to the DbSet of relevant entities.

This approach ensures efficient data management and separation of concerns, where the service class handles the persistence logic while the business logic remains encapsulated in the methods.

By incorporating these persistence mechanisms, the updated design class diagram effectively captures the expanded requirements and behaviors, ensuring the system design aligns with the dynamic analysis findings.

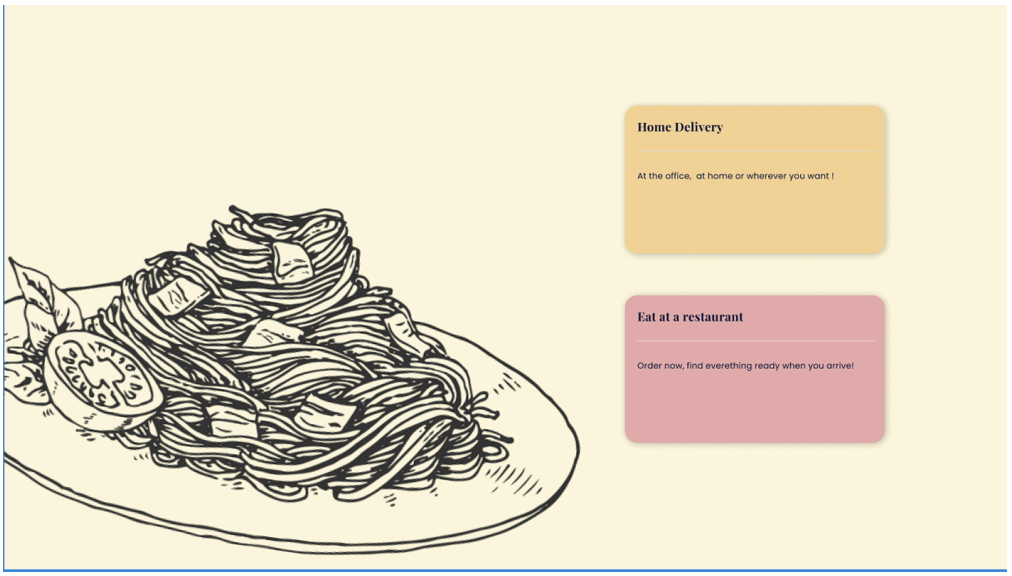
Final Design diagram:



GUI Design discussion

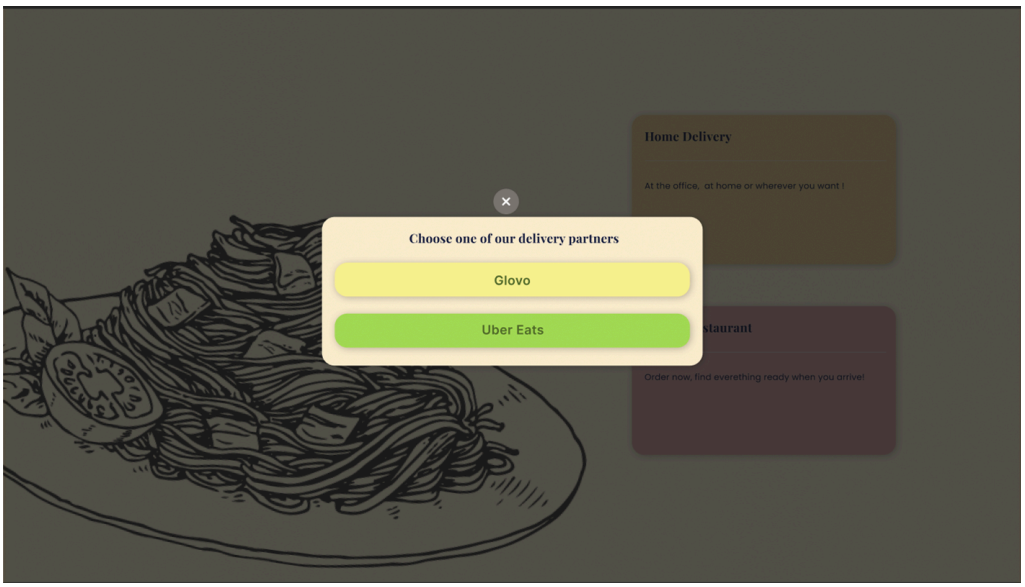
This section introduces the initial GUI design, examines its implications, and explains the thought process behind its creation. Using Figma, a dynamic analysis of the chosen use case was conducted to identify essential UI elements, such as buttons and navigation tools, required for its execution. Additional design components were incorporated to showcase and reinforce the application’s design principles and aesthetic.

The initial GUI prototype was developed as a conceptual tool to explore the design philosophy and overall structure of the application.



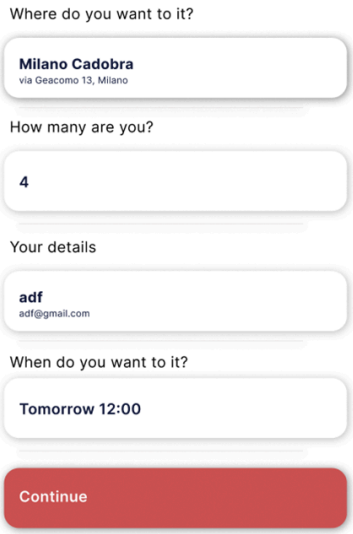
Start the order (step 1a2)

The screen displayed above represents the website's starting page, offering two options: delivery or dining in the restaurant. If the user selects the delivery option, a popup appears, providing access to the delivery service partners' websites, where users can proceed with the delivery details. Pressing the button “dine at the restaurant” will redirect the user to the reservation details page.



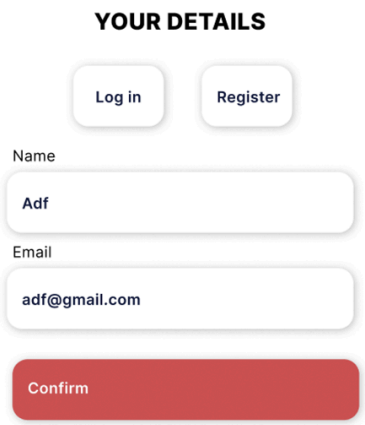
Start the order (step 1a3a1)

his page is designed to redirect customers to external food delivery services (e.g., Uber Eats, Glovo, Volt) where the order will be processed.The page displays a modal or pop-up with the title, "Choose one of our delivery partners." It offers clickable buttons for the available delivery services. Each button is a link that redirects the customer to the corresponding service's website or app. A background featuring food imagery enhances the aesthetic appeal and relevance of the page. The pop-up is central and clearly visible, ensuring easy interaction.Buttons are visually distinct and labeled with the names of the delivery services, using their branding colors for recognition. Upon clicking a button, the system navigates the user to the selected delivery service’s platform.The order details may already be pre-filled if the system is integrated with the delivery partner’s API. If not, the customer will need to manually input order details on the partner’s site.The pop-up includes a close button (visible at the top) that allows users to exit the selection modal and return to the main website if they decide not to proceed with a delivery partner.



Start the order (step 1a4)

The page above showcases the reservation details interface following the selection of the "Dining at the Restaurant" option. Here, users can conveniently choose their desired restaurant from a dropdown menu, specify the number of guests to allow the system to assign an appropriately sized table, and set the date and time using an intuitive and user-friendly UI. Additionally, users must provide their details, as reservations require authentication for completion. Upon pressing the "Your Details" button, the system proceeds to capture the user's information, ensuring a seamless and secure reservation process.



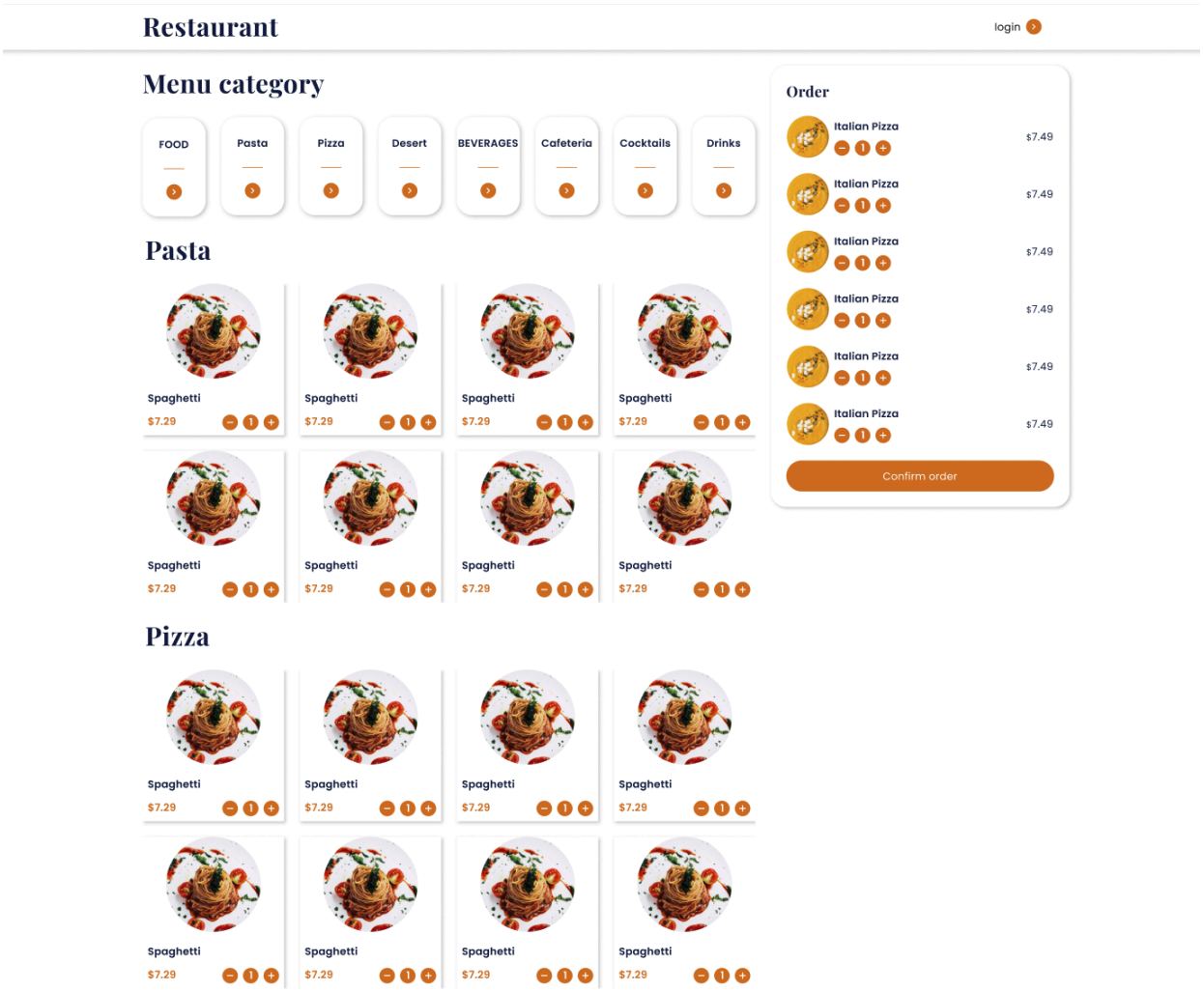
Start the order (step 1a6)

The page above is designed to collect the user's personal information, as reservations cannot be completed without basic authorization. Users are prompted to provide their name and email through input fields or are given the option to log in or register using the buttons displayed at the top of the page.

Start the order (step 1a6a2) or Sign in page

The sign-in page requires users to enter their email and password. It includes a checkbox option to remember their login details and a "Forgot Password" button for

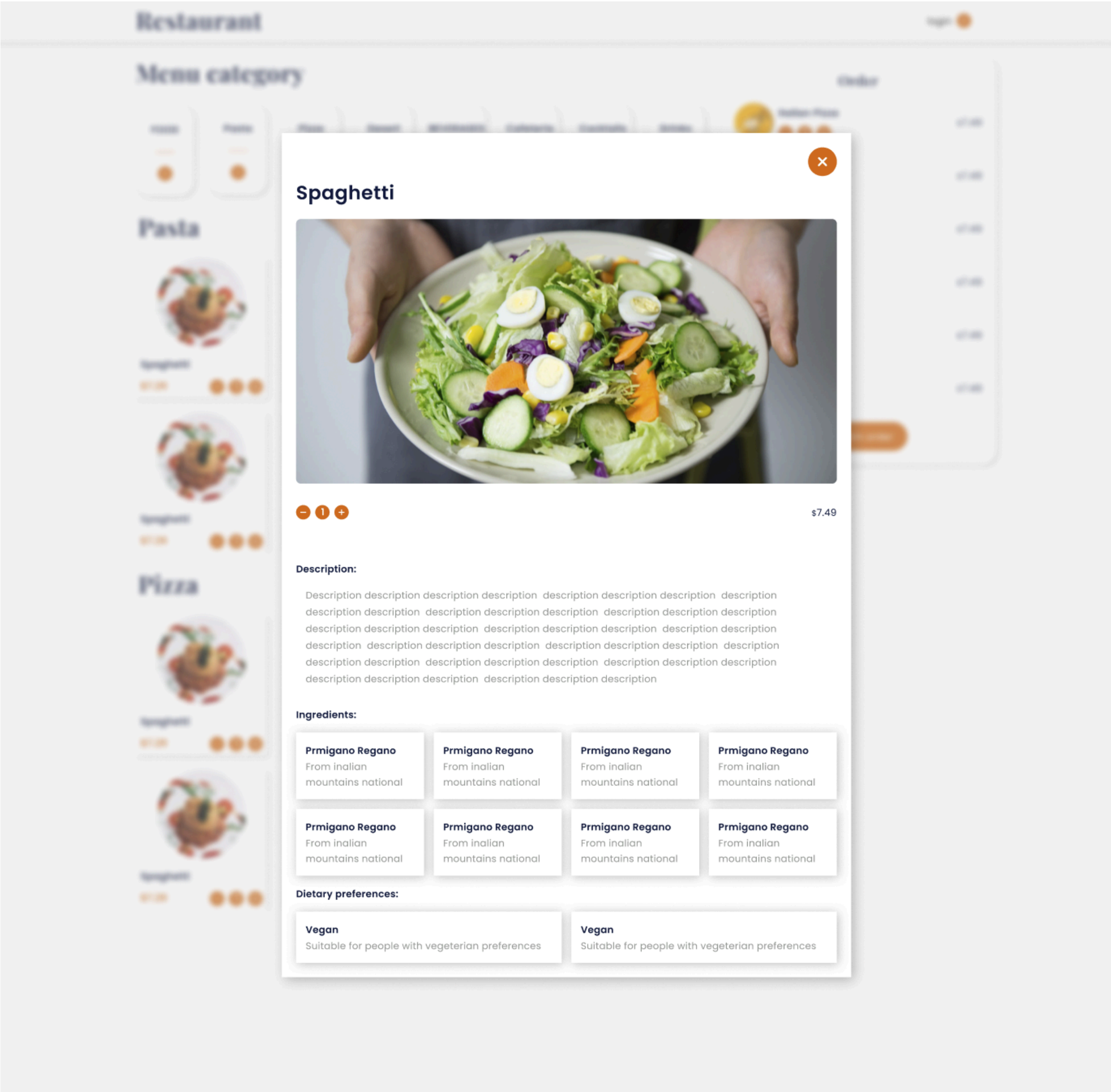
account recovery. After entering their credentials, users can click the "Sign In" button to proceed. Additionally, there is a "Sign Up" button that redirects users to the registration page.



Start the order (step 3) or Menu Page (Main Page)

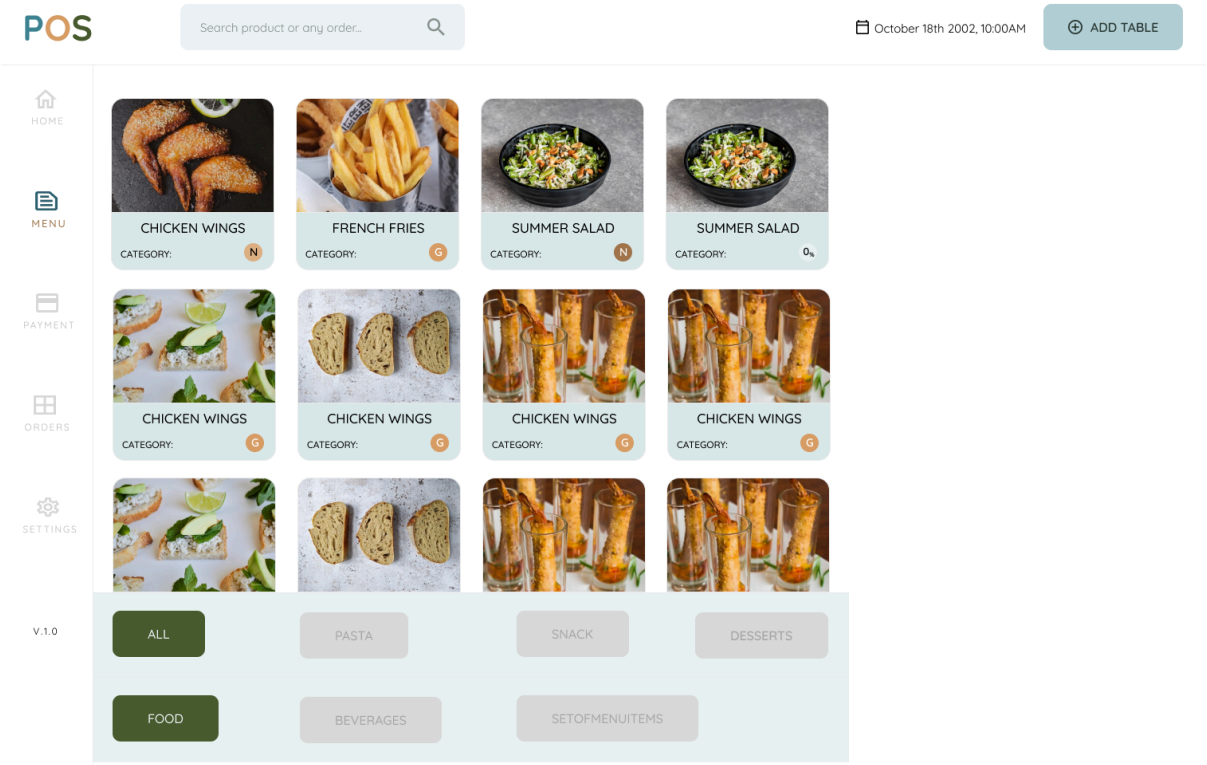
The page displayed above showcases menu items organized into categories. Users can select a category to view all the products within it and browse through them individually. Each product is presented with a photo, name, and price. By clicking on a product, users can access more detailed information about it. Additionally, users have the option to specify the quantity of the product they wish to add to their order.

On the right side of the page, the order list is visible, showing the names of selected products, their quantities, and prices. From this page, users can also navigate to the login screen if needed.



Remove/add menu item from/to order list (step 1a2)

The page shown above displays detailed information about a selected item when the user clicks on it. This page provides an overview of the product, including its name, price, description, a list of ingredients, and dietary preference options. Additionally, users can select the desired quantity of the item and add it to their order directly from this page.



Admin Menu Page

This page serves as the Admin Dashboard for managing the restaurant's menu. Here's a detailed description of the page and its functionality: The page is designed to give administrators an overview of all menu items, allowing them to perform actions such as adding, editing, or deleting ingredients from a menu item. It acts as the starting point for the Delete/Add Ingredient From/To MenuItem use case scenario.

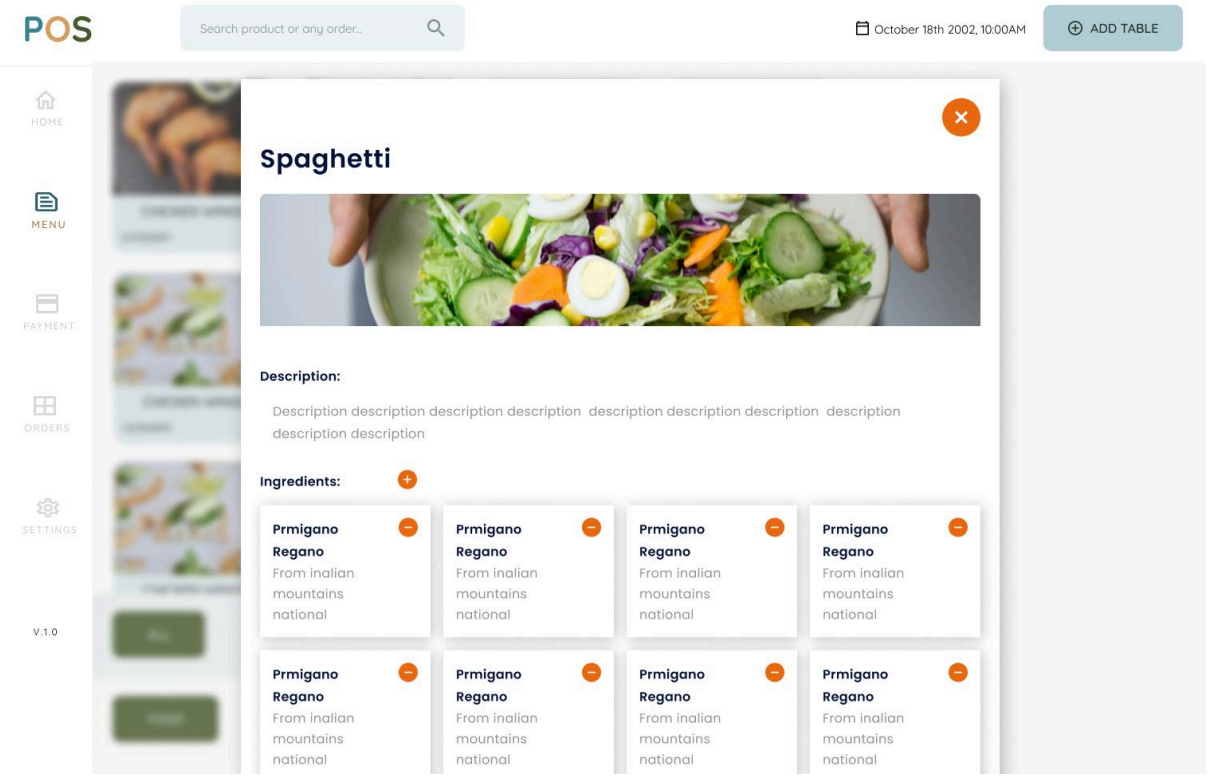
Header Section: A search bar is available at the top, allowing admins to search for specific products or orders. A timestamp displays the current date and time. An "Add Table" button, likely for managing restaurant tables, is positioned in the top-right corner.

Sidebar Menu: The left-hand side contains a navigation bar with icons for different sections like Home, Menu, Payment, Orders, and Settings. This allows admins to switch between functionalities easily.

Main Content Area: The central area displays the menu items in a grid format. Each menu item is represented as a card, showing: The name of the dish (e.g., Chicken Wings, French Fries). A thumbnail image for visual identification. The category the item belongs to (e.g., Food, Beverages). A small icon or button for quick actions like zooming into details or performing edits.

Category Filters: A filtering bar at the bottom allows admins to view menu items based on categories such as Pasta, Snacks, Desserts, Food, Beverages, etc. Active filters can dynamically update the displayed items, making it easier to locate menu items.

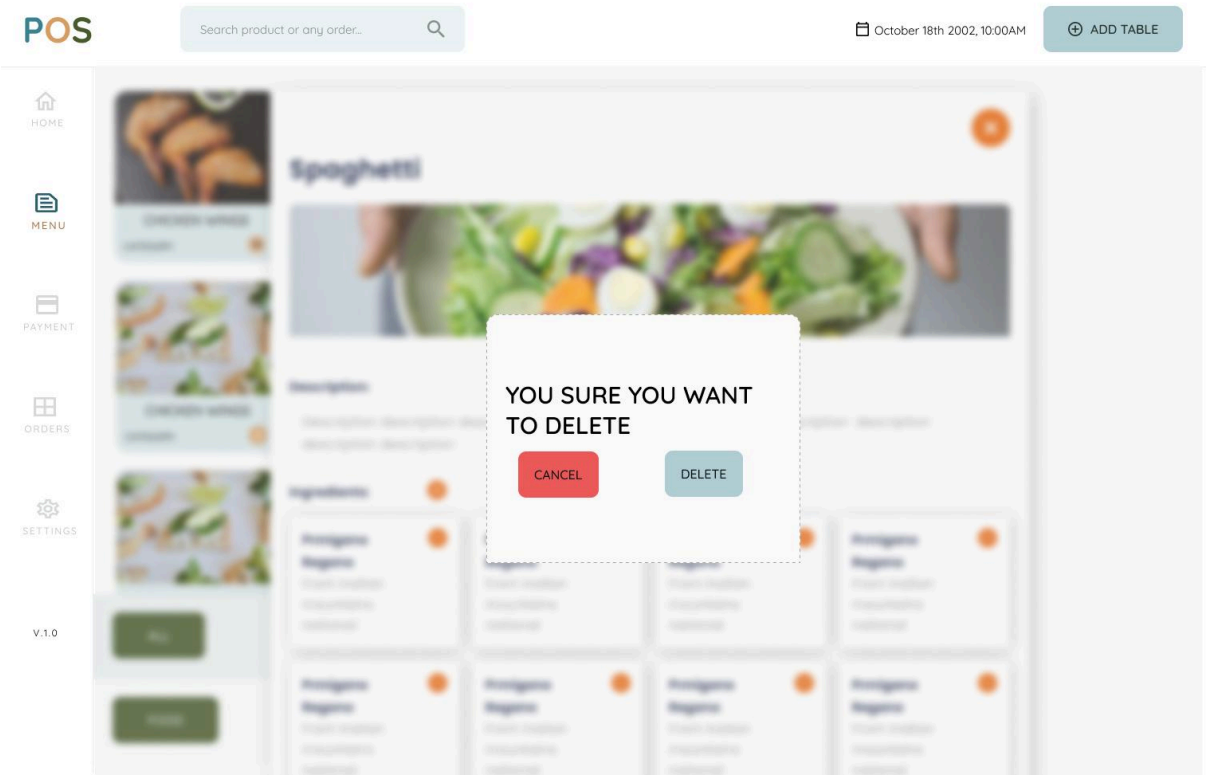
Relevance to the Use Case Scenario: Delete Ingredient: Admins can select a menu item from the grid to view its details. Once a menu item is selected, a dedicated page or modal likely appears, showing the list of ingredients where deletion can be performed. Add Ingredient: Admins can use this interface to navigate to the ingredient management section. They may also search for a menu item, access its details, and proceed with adding ingredients.



Delete/Add Ingredient From/To MenuItem (step 2)

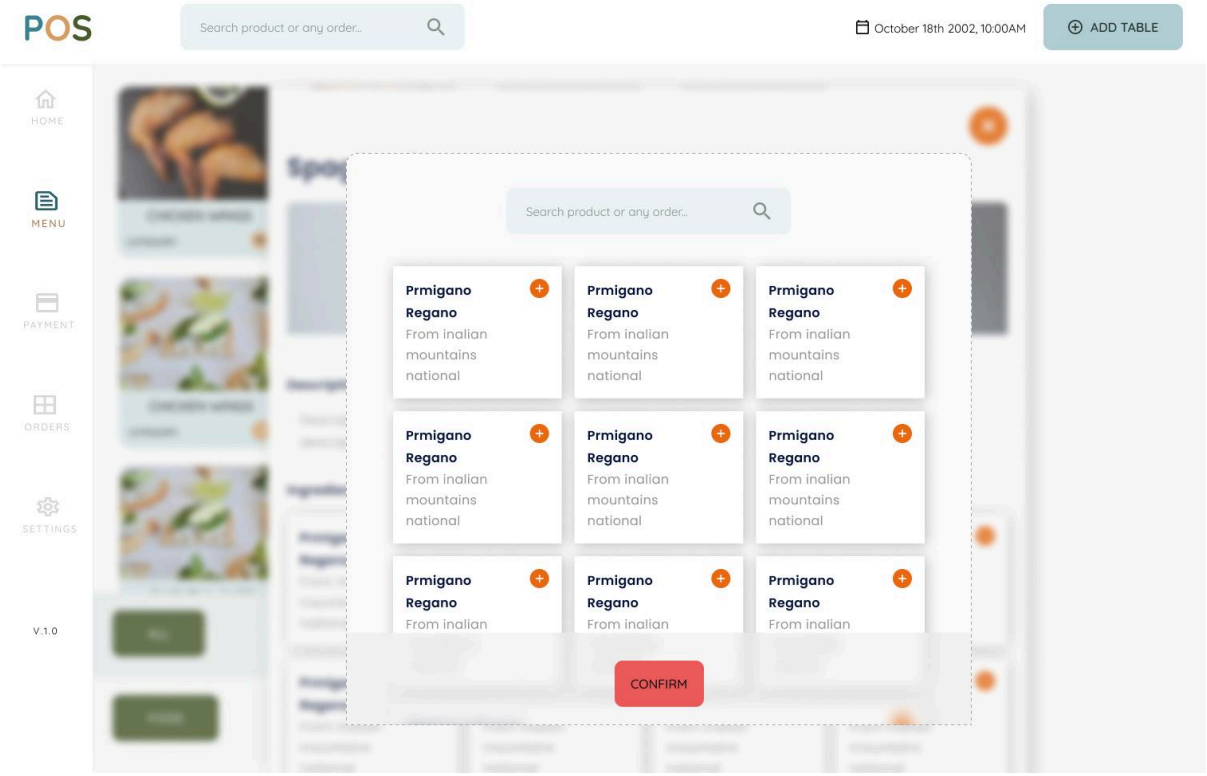
This page is part of a Point of Sale (POS) system interface where the admin interacts with the detailed information of a selected menu item, "Spaghetti." The page displays the following: Header Section: Includes navigation options such as a search bar, a timestamp, and a button labeled "Add Table." Menu Details Section:The item name ("Spaghetti") is displayed prominently. An image of the menu item is shown. A description area provides information about the item. A list of ingredients is presented in a card-like format, with each ingredient having a label and descriptive details, such as origin information.Each ingredient card includes a delete button (orange) for removing an ingredient. The page is part of the workflow where

admins can manage the ingredients of a menu item, either by deleting existing ones or adding new ones through an alternative flow.



Delete/Add Ingredient From/To MenuItem (step 4)

This page is part of the "Delete/Add Ingredient From/To MenuItem" (step 4) use case in the POS system. It displays a confirmation dialog after the admin has selected an ingredient to delete from the menu item ("Spaghetti"). Here's the description: Background Context: The detailed menu item page remains visible in the background, displaying the name "Spaghetti," an image of the dish, and the list of ingredients. Confirmation Dialog: A centered popup window overlays the page. The dialog prompts the admin with the message: "YOU SURE YOU WANT TO DELETE". Two buttons are provided for action: Cancel (red button): Allows the admin to abort the deletion process. Delete (blue button): Confirms the deletion of the selected ingredient. This step ensures the admin explicitly approves the deletion action before the ingredient is removed from the menu item, maintaining clarity and preventing accidental modifications.



Delete/Add Ingredient From/To MenuItem (step 3a2)

This page represents Step 3a2 of the "Delete/Add Ingredient From/To MenuItem" use case in the POS system, where the admin is adding an ingredient to a menu item. The details are as follows: Background Context: The detailed menu item page for "Spaghetti" remains visible in the background. Ingredient Selection Popup: A popup window overlays the page to facilitate the selection of ingredients. The popup contains a search bar at the top, allowing the admin to filter or search for specific ingredients. Below the search bar, a grid of ingredient cards is displayed, showing: Ingredient name (e.g., "Parmigiano Reggiano"). Description details (e.g., origin information). Delete buttons (orange) for removing individual selections. Confirmation Button: At the bottom of the popup, a "CONFIRM" button is displayed in red, allowing the admin to finalize the selected ingredients for addition. This step ensures the admin can browse and select ingredients conveniently before confirming their addition to the menu item.