ERROR_418

Restaurant Simulation

Graeme Blain (u22625462)

Aidan Chapman (u22738917)

Kabelo Mahleka Chuene (u14046492)

Douglas Porter (u21797545)

Sange Tshakumane (u21479748)

Boikanyo Tshwale (u20785233)

GitHub repository:

https://github.com/GremBleen/COS214-PROJECT

Contents

2.1) System Requirements:

(Bold text represents participants in the system)

The restaurant simulator system will comprise two main areas: the **Floor** and the **Kitchen**, both of which are essential for a pleasant customer experience. The Floor management includes **Customer** seating, order taking, and bill processing. A customer requests a **Table**, after which a **Waiter** will seat the customer/s; and then takes their **Order** (once the customer is ready and willing to place their order). The Waiter then delivers the order to the Kitchen who will prepare said order. Once the order has been prepared by the Kitchen, the Waiter delivers it to the table at which the customer is seated. Table management will take the form of assigning specific tables to a specific waiter. Customer **Satisfaction** plays a vital role, influencing how much customer's will tip based on their experience. The system will allow customers to create orders based on a set-menu as well as choose the method of cooking that they prefer (Grilled or Fried).

The kitchen is the centre of the restaurant where orders are received (by Waiters who place the order), and different chefs manage distinct preparation stages. Each chef handles specific tasks, such as the head chef plating dishes, and the fry cook working at the fryer. Orders move between stations for preparation before returning to the head chef for final plating and delivery to the table (and thus customer) by the waiter. Upon completion of the order (i.e., once the Head Chef has plated the order), the kitchen notifies the waiter to collect the order for delivery to the customer.

Key interactions involve rounds by waiters, and managers to ensure customers are receiving adequate service; customers may also make a complaint during a waiter or manager's round if they are unhappy with the service they have received. The system may allow for potential extensions like a bar with various cocktails, and a tab system such that a customer may order any number of items and pay the bill once they are ready to leave the restaurant.

The system design focuses on simulating the intricate processes of a restaurant, bridging communication between the floor and kitchen, managing customer expectations and complaints; as well as delivering a versatile dining experience by means of custom orders. The optional extensions provide avenues for further complexity and functionality down the line to accurately simulate the running of a real Restaurant.

4.1) Research:

Restaurants (and the effort that goes into making your food):

Chefs:

https://www.freepik.com/free-photo/chef-

The team of chefs in a restaurant, consisting of the head chef, side chef, and main chef, play a crucial role in delivering a seamless and exquisite dining experience to customers. Each chef has a specific set of responsibilities that work in tandem to create a variety of dishes according to a customer's request. They are each responsible for a portion of a meal; and together form a culinary brigade.

A culinary brigade is a hierarchical system of kitchen organization that originated in France in the 19th century. It is based on the principle of division of labor, with each chef having a specific set of responsibilities.

The head chef leads the brigade and is responsible for overseeing all aspects of the kitchen operation, including menu development, recipe creation, and staff training. The other chefs in the brigade are each responsible for a specific station or task, such as preparing appetizers, main courses, or desserts.

The culinary brigade system is designed to ensure that food is prepared efficiently and to a high standard. By working together as a team, the chefs in a brigade can create delicious and visually appealing meals that are sure to please customers.

Side Chef:

The side chef is responsible for preparing and cooking the side dishes that accompany the main course. This may include items such as roasted vegetables, mashed potatoes, or salads. The side chef must be able to work quickly and efficiently to ensure that the side dishes are ready to be served alongside the main course; as there are likely less Side Chefs as opposed to other Chefs (such as the Main Chef) since in theory, sides take less time to prepare than the main component of a dish.

In addition to preparing side dishes, the side chef may also be responsible for other tasks, such as:

- Garnishing dishes
- Making sauces and dressings
- Assisting the head chef with plating dishes
- Cleaning and maintaining the kitchen.

Some examples of a side chef's day-to-day responsibilities include:

- Arrive at work early to prepare for the lunch shift.
- Wash, cut, and season vegetables for roasting.

- Make a batch of mashed potatoes.
- Prepare a salad dressing.
- Deliver the side dishes to the pass station for pickup by other kitchen staff.
- Clean and maintain the kitchen throughout the day.

Side chefs play a vital role in the kitchen, and although they do not receive much recognition for their part in creating a meal; no meal would be complete without them.

Main Chef:

The main chef is responsible for cooking and preparing the main component of the dish, such as the meat, poultry, or fish. The main chef must have a deep understanding of different cooking techniques and be able to cook food to perfection; independent of the type of meat/food item being cooked.

Main chefs must also be able to work under pressure and handle multiple orders simultaneously. This is especially important during peak hours (such as lunch time), when the kitchen can become extremely busy. Main chefs must be able to stay organized and efficient, and they must be able to communicate effectively with the other members of the kitchen team even in high pressure situations.

Some examples of a Main chef's day-to-day responsibilities include:

- Arrive at work early to prepare for the lunch shift.
- Season and marinate meat for grilling and roasting.
- Set up the grill and oven.
- Grill and roast meat to order.
- Clean and maintain the grill and oven.

The Main chef is likely the most well-known member of the kitchen; as they are responsible for the core components of a meal and therefore naturally receive the most credit for the role they play.

Head Chef:

Some more examples of a Head chef’s responsibilities include:

- Costing and menu planning
- Food safety and sanitation
- Wine pairings and beverage selection

The head chef sets the tone for the entire kitchen staff, maintaining their creative vision and standards within the kitchen. They are responsible for creating a positive and productive work environment where chefs can thrive and express their creativity, while still ensuring the highest standards are upheld. The head chef also plays a key role in building relationships with suppliers and customers, and in promoting the restaurant's brand.

Here is an example of a head chef's day-to-day responsibilities:

- Arrive at work early to meet with the other chefs and discuss the day's menu and orders.
- Check the quality of the ingredients that have arrived from the suppliers.
- Work on developing and refining new recipes.

- Oversee the preparation of all dishes, ensuring that they meet the highest standards of quality and presentation.
- Plate dishes and deliver them to the pass station for pickup by the waiters.
- Receive feedback from customers (by means of feedback from waiters) and use it to improve the menu and overall dining experience.

The head chef is a demanding role, but it is also incredibly rewarding. Head chefs can create their own unique culinary vision and to share it with the world in the form of the dishes their kitchen creates. They also have the responsibility of managing a team of chefs to create delicious and memorable meals for their customers.

Waiters:

Image credit

Image credit

Waiters and waitresses are responsible for providing excellent customer service. This includes greeting customers promptly and seating them at a clean table, taking orders accurately, delivering food and drinks to customers in a timely manner, checking in with customers to make sure they are satisfied with their meal, clearing empty plates and glasses, bringing the bill to customers, and processing payments. Waiters and waitresses should also be knowledgeable about the menu (including possible specials and popular menu items) and the

restaurant's policies as well as procedures and be able to answer customer's questions or resolve any issues a customer may encounter.

Waiters and waitresses should dress professionally and have a clean and well-groomed appearance. They should also always be polite and courteous to customers. Waiters and waitresses should be able to carry a tray with at least four food items and two drinks without dropping anything, a skill that requires practice and focus. They should also be able to remember customer orders without having to write them down; although this is not strictly necessary it is beneficial to the overall efficiency of the restaurant.

some additional responsibilities of a waiter/waitress:

- Upsell and cross-sell menu items: Waiters and waitresses can increase sales by recommending complementary dishes and drinks to customers. For example, they might suggest a side of fries or a glass of wine to go with a main course.
- Handle complaints: Waiters and waitresses are often the first line of defense when customers have a complaint. They should be able to handle complaints in a calm and professional manner, and they should escalate the issue to a manager or the head chef if necessary.
- Promote the restaurant: Waiters and waitresses can help to promote the restaurant by telling customers about upcoming events and promotions. They can also encourage customers to leave reviews online.
- Maintain the dining room: Waiters and waitresses should help to keep the dining room clean and tidy. This includes wiping down tables, sweeping floors, and restocking supplies.

In addition to these responsibilities, waiters and waitresses should also be knowledgeable about the following:

- Wine and beer pairings: Waiters and waitresses should be able to recommend wines and beers that will pair well with customers' meals.
- Food allergies and dietary restrictions: Waiters and waitresses should be aware of common food allergies and dietary restrictions. They should be able to answer customer questions about the menu and make recommendations accordingly.
- Restaurant policies and procedures: Waiters and waitresses should be familiar with the restaurant's policies and procedures. This includes things like dress code, payment options, and reservation policies.

By understanding and fulfilling all their responsibilities, waiters and waitresses are essential to the success of a restaurant.

Managers:

Managers oversee the operation of the restaurant and ensure that customers have a positive dining experience. They are responsible for a wide range of tasks, including but not limited to:

- Scheduling and managing staff: Managers create and manage employee schedules, ensuring that there are enough staff members on hand to meet customer demand (A tricky yet crucial skill). They also train and develop staff members, as well as provide feedback on their performance.
- Tracking inventory: Managers track the restaurant's inventory of food and beverage supplies, as well as ensure that orders are placed in time to avoid stockouts. They also work with suppliers to negotiate contracts and get the best prices on ingredients while also maintaining the highest level of quality.
- Ensuring food safety and sanitation: Managers are responsible for ensuring that the restaurant complies with all food safety and sanitation regulations. They develop and implement food safety procedures, and train staff members on how to follow them.

- Resolving customer issues and complaints: Managers handle customer complaints and issues promptly and professionally. They strive to resolve all issues to the customer's satisfaction.
- Maintaining a clean and comfortable dining environment: Managers oversee the cleanliness and maintenance of the restaurant's dining room and kitchen areas.

Here are some examples of how managers oversee the operation of a restaurant:

- A manager may meet with the head chef each morning to discuss the day's menu and any special orders that have been placed.
- A manager may review employee schedules and adjust as needed to ensure that there is adequate coverage during peak hours.
- A manager may place orders with suppliers to ensure that the restaurant has enough food and beverage supplies on hand.
- A manager may inspect the kitchen to ensure that food safety and sanitation procedures are being followed.
- A manager may speak with customers to get their feedback on their dining experience and to resolve any issues that they may have had.

  Managers are yet another pillar in the success of any restaurant. By overseeing the operation of the restaurant and ensuring that everything is running smoothly, they help to create a positive dining experience for customers and keep service up to the standard expected by customers.

4.2&3) Design Patterns used and reasoning:

Observer:

The Observer design pattern is well-suited for a system where a waiter needs to monitor and respond to the satisfaction of a customer. This pattern facilitates a one-to-many dependency between objects, ensuring that when one object (the subject) changes its state, all its dependents (observers) are notified and updated automatically.

By utilizing the Observer pattern, the system allows for a more responsive and customer-centric service. It enables the waiter to be more attentive to customers' needs, thereby potentially improving overall customer satisfaction and experience.

The Observer design pattern is beneficial in a restaurant system for waiters to actively monitor customer satisfaction. It promotes a more responsive and customer-focused approach, allowing for timely intervention and improved customer experience based on real-time feedback and observations.

- Context: Restaurant
- Observer/ConcreteObserver: Customer
- Subject/ConcreteSubject: Waiter
- Client: main

State:

The State pattern allows the Customer object to encapsulate the behavior associated with different satisfaction states. Each Concrete State (Unhappy,

Neutral, Satisfied) encapsulates the behavior related to tipping within its specific state.

It provides a way to represent varying customer states without changing the Customer class. This enables easy addition of new states or modification of existing states without impacting the core Customer class.

- Dynamic Behavior: The State pattern allows the system to handle customer satisfaction changes dynamically, ensuring that the tipping behavior adapts accordingly. For example, an Unhappy state might reduce the tip amount, while a Satisfied state might increase it.

- Scalability: As the system evolves, the addition of new states or modifications to existing states (Unhappy, Neutral, Satisfied) becomes more manageable without extensive changes to the Customer class, contributing to a more scalable and adaptable system.

- Context: Customer
- State: Rating
- ConcreteStates: Unhappy, Neutral, & Satisfied
- Client: Interface

Mediator:

The Mediator pattern allows the Restaurant, acting as the mediator, to control and coordinate interactions between different components (Floor, Kitchen, Table, and Waiter) without direct coupling among them.

Components within the system don't need direct knowledge of each other. Instead, they communicate solely through the Mediator (Restaurant), reducing interdependencies and making the system more flexible and maintainable.

- Order Management: The Mediator (Restaurant) can handle the communication between the Waiter and Kitchen. When an order is placed, the Waiter informs the Restaurant, which then communicates with the Kitchen to ensure proper meal preparation and delivery.

- Customer Seating: The Restaurant, acting as the Mediator, coordinates seating arrangements managed by the Floor and Table components, ensuring efficient table allocation without direct interactions between them.

- Communication Streamlining: Various notifications or updates between the Waiter, Kitchen, Floor, and Table can be centralized through the Mediator, reducing complexities and enabling a more organized and manageable system.

- Mediator/ConcreteMediator: Restaurant
- Colleagues/ConcreteColleagues: Floor, Kitchen, Table, & Waiter
- Context: Restaurant
- Client: Interface

Façade:

This pattern serves to simplify and unify the complex subsystems by providing a unified interface, the Interface participant, making the system more manageable and easier to use.

The Façade design pattern, represented by the Interface participant, offers a simplified and high-level interface for creating and managing various subsystems

(customers, kitchen, restaurant, tables, waiters, and orders) in the Restaurant system.

The pattern encapsulates the complexities and intricacies of subsystem interactions within the restaurant system. Users interacting with the system only need to utilize the Façade's simple interface, shielding them from the underlying complexities of subsystem interactions. The Interface acts as a centralized access point through which clients interact with different subsystems. This simplifies the initialization and management process for customers, kitchen, restaurant, tables, waiters, and orders.

- Initialization and Management: The Interface participant in the Facade pattern manages the creation of customers, kitchen, restaurant, tables, waiters, and orders, providing a clear and simplified interface for these actions.

- Ordering and Service Flow: Users interacting with the system, such as staff members or even customers in certain scenarios, can utilize the Facade to place orders, manage table arrangements, and handle customer requests, all without having to understand the intricate details of how these actions are processed internally.

- Centralized Interface: By offering a centralized interface, the Interface participant simplifies the interaction with the various subsystems, making it easier for users to perform operations within the restaurant system without delving into the complexities of each subsystem.

- Context: Restaurant
- Façade: Interface
- Subsystems: Customer, Waiter, Kitchen, Restaurant, Floor, Chef, Order, Table, & Rating.
- Client: main

Adapter:

The Adapter design pattern is beneficial in a Restaurant system, especially when the Chef participant is represented by multiple subclasses (MainChef, SideChef, DrinkChef, HeadChef) that might have differing functionalities, but need to be utilized interchangeably or independently. This pattern allows for a consistent and unified way to interact with these different chef subclasses, providing a common interface for the Kitchen to engage with any type of chef without requiring specific knowledge of each chef type.

- Interchangeable Usage: The Adapter pattern enables the independent use of different Chef subclasses without needing detailed knowledge about their specific roles or functions.

- Consistent Operations: Kitchen can interact uniformly with any type of Chef subclass, promoting consistent usage and simplifying code in the restaurant system.

- Enhanced Maintainability: Modifications or expansions in Chef functionality can be implemented through adapters without affecting the client code, contributing to a more maintainable system.

- Context: Kitchen
- Target: Kitchen

- Adapter: Chef
- Adaptees: MainChef, SideChef, DrinkChef, & HeadChef
- Client: Restaurant

Strategy:

The Strategy design pattern is beneficial in a Restaurant system where an ItemBuilder encapsulates three subclasses (DrinkBuilder, MainBuilder, SideBuilder) used to create Items based on the type required (Drink, side, or main). This pattern facilitates a dynamic and interchangeable approach to item creation by allowing the ItemBuilder to change its behavior at runtime based on the required item type.

- Flexibility in Item Creation: The Strategy pattern enables the ItemBuilder to dynamically switch between different creation strategies, allowing a Chef to generate different types of items based on demand.

- Improved Maintainability: Encapsulating item creation logic within separate subclasses promotes a modular and maintainable code structure, facilitating easier modifications or additions to item creation strategies.

- Reduction in Code Duplication: The pattern ensures that item creation strategies are distinct and reusable, reducing the need for redundant code and enhancing overall system organization.

- Context: Chef
- Strategy: ItemBuilder
- ConcreteStrategies: DrinkBuilder, MainBuilder, & SideBuilder
- Client: Kitchen

Builder:

The Builder pattern is apt for building complex objects by breaking down the creation process into multiple steps. In a restaurant context, the creation of varied and composed items like meals or drinks with different components aligns well with this pattern.

Each subclass within the ItemBuilder hierarchy represents specific components or variations of an item (e.g., different types of drinks, mains, or side dishes). This allows the gradual construction of a complete Item by adding various elements step by step.

- Customized Item Composition:

  Subclasses within each subclass of the ItemBuilder hierarchy offer tailored ways to construct items of different types, providing customization and specificity for each component or combination.

- Separation of Concerns:

  The Builder pattern separates the construction process from the representation of the final object. It allows individual classes to handle the creation of different parts, maintaining a clear division of responsibilities.

- Scalability and Extendibility:

  Adding new subclasses or elements to existing builders becomes straightforward, providing scalability and extensibility to accommodate new types or variations of items in the restaurant system.

- Context: Restaurant
- Director: Chef
- Builder: ItemBuilder, MainBuilder, SideBuilder, & DrinkBuilder

- ConcreteBuilders: WaterBuilder, BeerBuilder, SodaBuilder, SteakBuilder, BurgerBuilder, FishBuilder, ChipsBuilder, & SaladBuilder
- Product: Item
- Client: Kitchen

Template Method:

The Template Method pattern provides a common blueprint for creating items, allowing the structure of the creation process to be standardized across various item types. Each builder class provides a common structure for creating different categories of items (drinks, mains, side dishes), while deferring specific steps to their respective subclasses.

- Consistent Structure, Varied Implementation: The Template Method pattern ensures a standardized structure for item creation while allowing subclasses to modify or override specific steps, tailoring the creation process as per the item's type.

- Unified Workflow, Customization in Subclasses: It maintains a coherent workflow across different item types, ensuring a standardized sequence of steps, while enabling flexibility and customization in specific subclasses.

- Promotion of Reusability and Maintainability: The pattern enhances code reusability by providing a common structure for item creation, making it easier to maintain and extend the system.

- Context: ItemBuilder
- AbstractClasses: DrinkBuilder, MainBuilder, & SideBuilder

- ConcreteClasses: WaterBuilder, BeerBuilder, SodaBuilder, BurgerBuilder, FishBuilder, SteakBuilder, SaladBuilder, & ChipsBuilder

Chain of Responsibility:

The Chain of Responsibility design pattern was used because for a Restaurant system where a Chef entity encompasses various subclasses: MainChef, SideChef, DrinkChef, and HeadChef; this pattern enables a Chef to receive an order and pass the request to one or more other chefs to fulfill the order, creating a chain-like structure where each Chef handles specific tasks based on their specialization before being handed off to the Head Chef for plating (the final chef in the chain), ensuring a smooth and flexible workflow for order fulfillment.

- Specialization in Order Handling: Each Chef subclass specializes in handling specific types of orders, ensuring that the appropriate Chef is responsible for fulfilling their designated part of the order.

- Flexibility and Scalability: The pattern allows for easy addition or modification of handling logic by adding new Chef subclasses or adjusting the sequence in the processing chain.

- Improved Decoupling and Extensibility: Decoupling the order sender (Chef) from its receiver (subclasses) provides flexibility, making it simpler to add, remove, or adjust the processing logic without affecting the core system.

- Context: Restaurant
- Client: Kitchen
- Handler: Chef
- ConcreteHandlers: MainChef, SideChef, DrinkChef, & HeadChef

Composite:

The Composite design pattern was chosen for this Restaurant system because orders need to be structured and processed uniformly, whether they are standalone individual items or compositions of multiple suborders. In our case, the Order class serves as the base component, with ComplexOrder inheriting from it. The Item class, with its subclasses (Side, Drink, Main), is treated as individual objects that can be composed into more complex orders, ensuring a unified handling approach for both individual items and composed orders. The Composite pattern establishes a hierarchical structure where individual items (Sides, Drinks, Mains) are treated as leaf components, and ComplexOrders represent compositions of these leaf components, allowing for flexible and varied order structures.

- Unified Handling and Processing: The pattern enables a unified approach to handling both simple items and composed orders, simplifying operations for the Restaurant system.

- Flexible Structure: The Composite pattern allows for the creation of various order structures, ranging from simple individual items to more complex composed orders, providing flexibility in managing diverse orders.

- Ease of Manipulation and Operations: It allows for uniform operations and manipulations on both individual items and complex orders, enhancing ease of management within the system.

- Client: ItemBuilder
- Component: Order
- Composite: ComplexOrder

- Leaf: Side, Drink, Main, Salad, Chips, Soda, Water, Beer, Steak, Fish, Burger

4.4) Assumptions, alterations, and design decisions:

Assumptions:

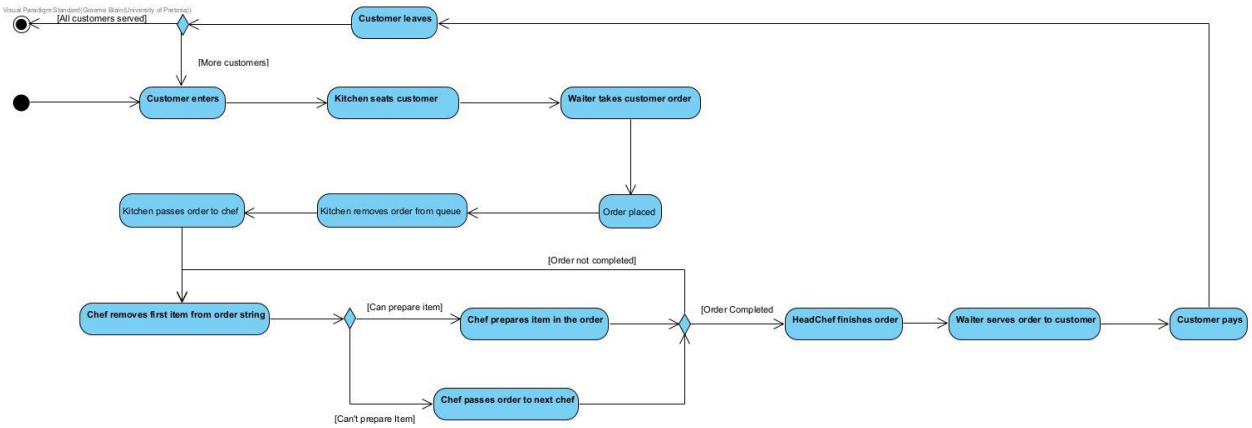
1. One customer will be seated at one table and served by one waiter.
2. Each order will comprise of mains, sides, and drinks.
3. The kitchen will handle orders one at a time, from a queue as there is no multithreading.
4. One customer is assumed to be one or more people with a single person paying.
5. There is a set menu for customers to order from.
6. Once a customer has been served an order they immediately pay and leave the restaurant, freeing up a table.
7. Waiters will be assigned specific customers and once served the waiter is free again. Lmao slavery
8. Customers will have a satisfaction level based on how fast the food was made and served.
9. Tables will always be able to seat a customer if no customer is already seated, regardless of the number of people the customer represents.
10. If a customer orders an item, it is assumed they can afford it.

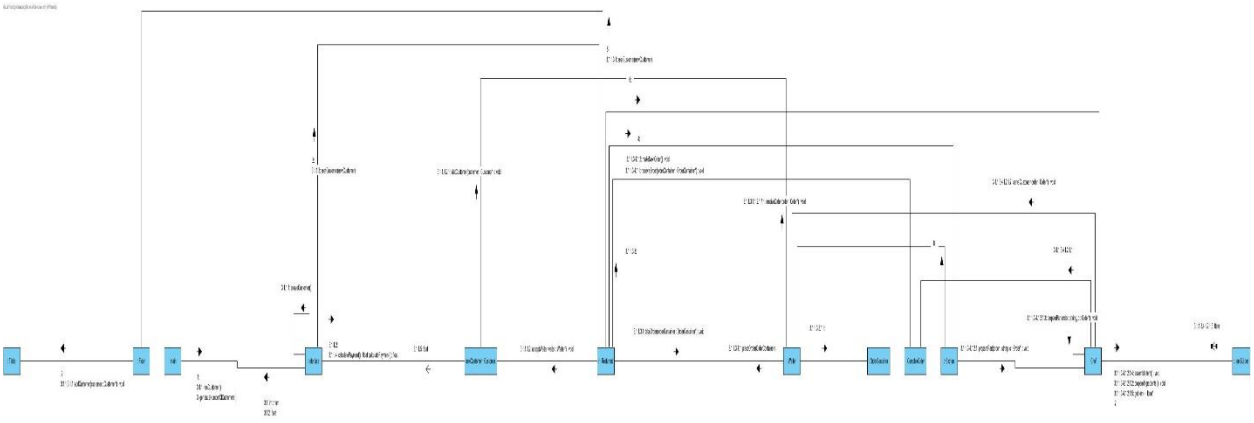
Alterations & Design Decisions:

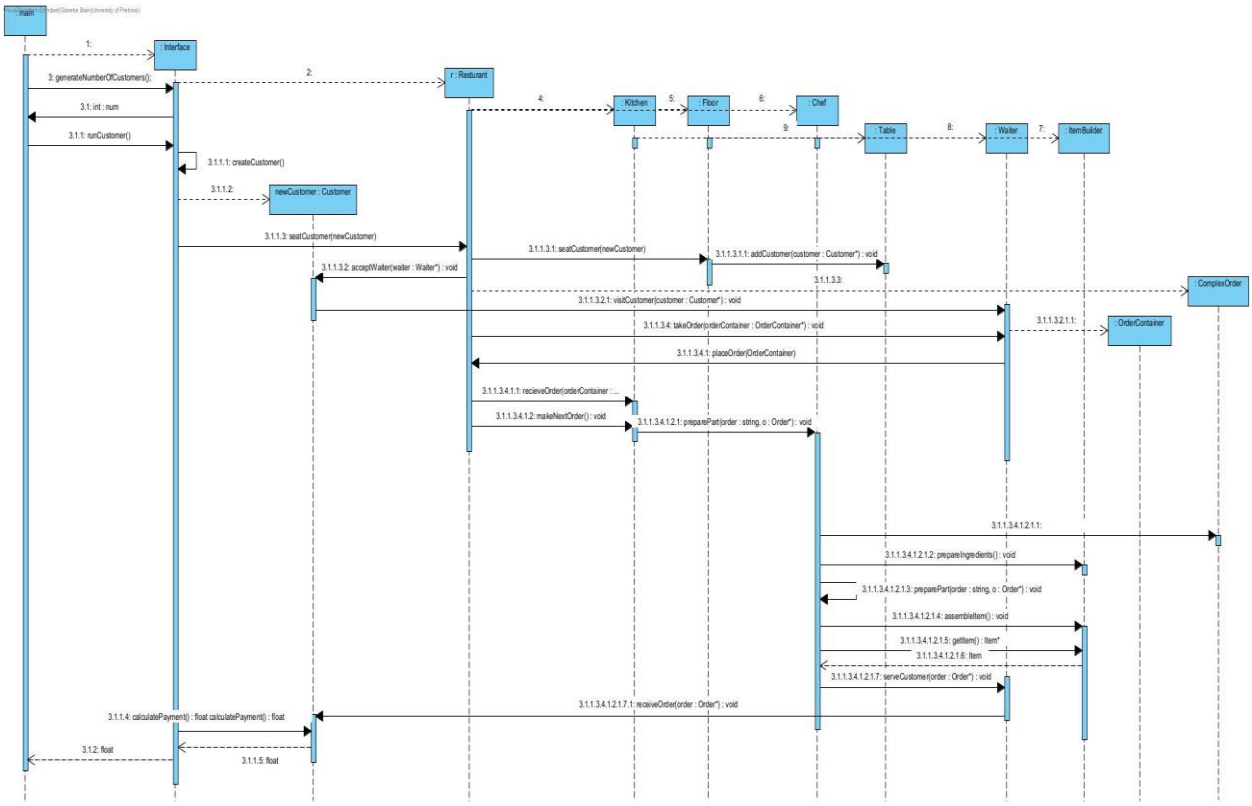
- Instead of a Manager doing rounds to ensure customer satisfaction, a Waiter in our system is responsible for visiting tables and checking on the satisfaction of customers.
- We have used a set-menu in our system in place of different cooking methods. Each menu item has a specific preparation method (For example, a Steak is grilled)

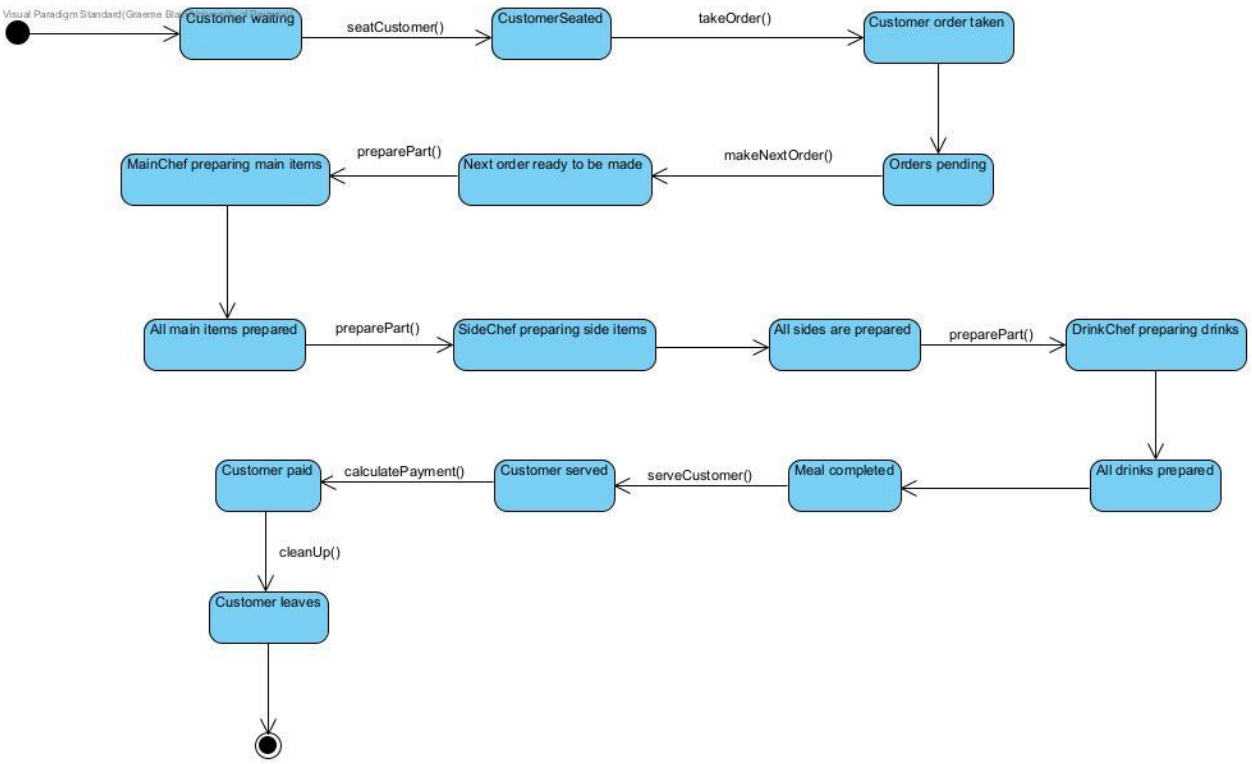
- In our system a waiter can only be assigned to a single table as opposed to any number of tables. This simplifies the seating, order placement, and serving within our system.

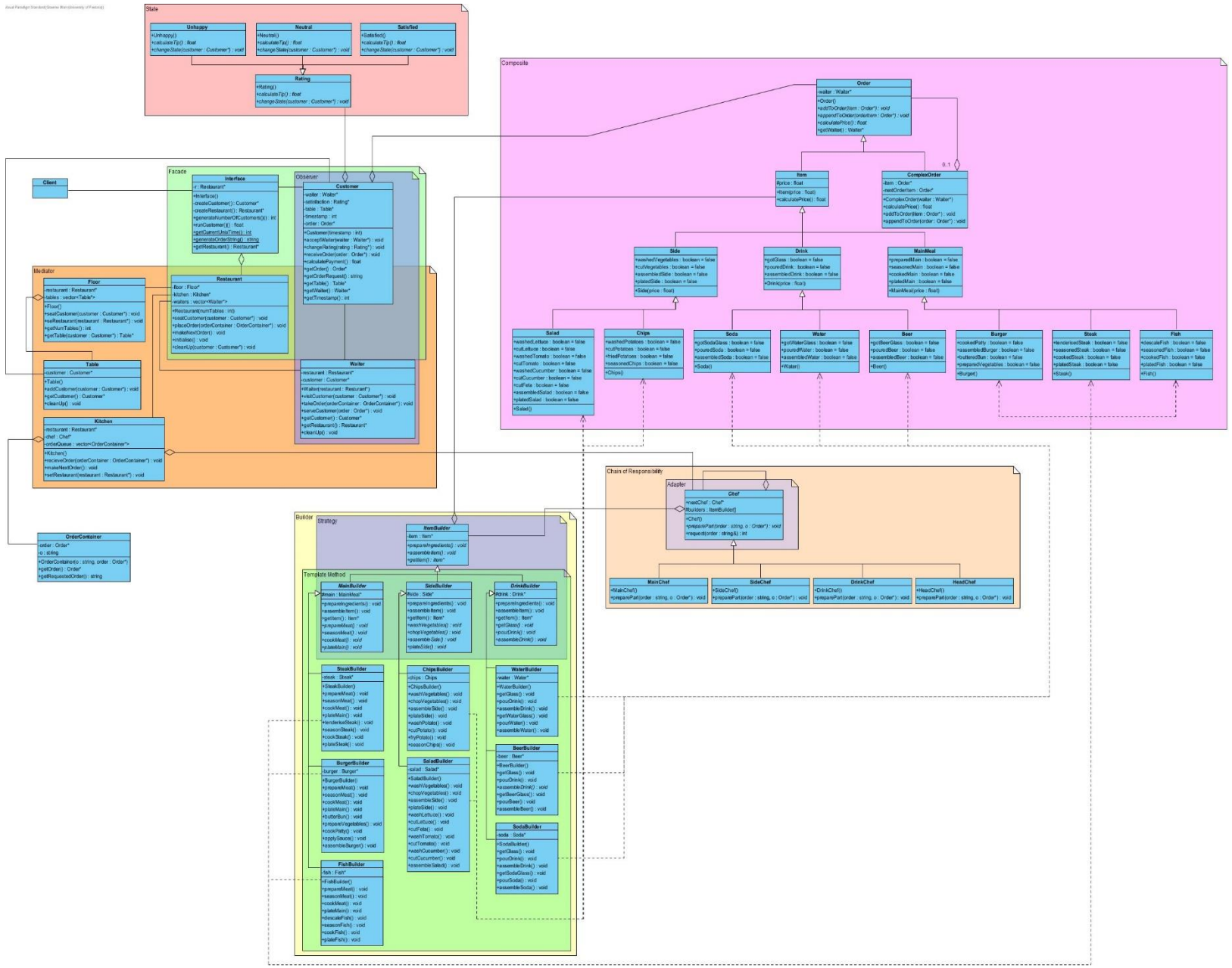
4.5) UML Diagrams:






main
: Interface
r : Resturant
: Kitchen
: Floor
: Chef
Table
: Waiter
: ItemBuilder
newCustomer : Customer
: ComplexOrder
OrderContainer
1:
2:
3: generateNumberOfCustomers():
3.1: int : num
3.1.1: runCustomer()
3.1.1.1: createCustomer()
3.1.1.2:
3.1.1.3: seatCustomer(newCustomer)
3.1.1.3.1: seatCustomer(newCustomer)
3.1.1.3.1.1: addCustomer(customer : Customer*) : void
3.1.1.3.2: acceptWaiter(waiter : Waiter*) : void
3.1.1.3.3:
3.1.1.3.2.1: visitCustomer(customer : Customer*) : void
3.1.1.3.4: takeOrder(orderContainer : OrderContainer*) : void
3.1.1.3.2.1.1:
3.1.1.3.4.1: placeOrder(OrderContainer)
3.1.1.3.4.1.1: recieveOrder(orderContainer : ...
3.1.1.3.4.1.2: makeNextOrder() : void
3.1.1.3.4.1.2.1: preparePart(order : string, o : Order*) : void
3.1.1.3.4.1.2.1.1:
3.1.1.3.4.1.2.1.2: prepareIngredients() : void
3.1.1.3.4.1.2.1.3: preparePart(order : string, o : Order*) : void
3.1.1.3.4.1.2.1.4: assembleItem() : void
3.1.1.3.4.1.2.1.5: getItem() : Item*
3.1.1.3.4.1.2.1.6: Item
3.1.1.3.4.1.2.1.7: serveCustomer(order : Order*) : void
3.1.1.3.4.1.2.1.7.1: recieveOrder(order : Order*) : void
3.1.1.4: calculatePayment() : float calculatePayment() : float
3.1.2: float
3.1.1.5: float


Customer waiting
seatCustomer()
CustomerSeated
takeOrder()
Customer order taken
Orders pending
makeNextOrder()
Next order ready to be made
preparePart()
MainChef preparing main items
All main items prepared
preparePart()
SideChef preparing side items
All sides are prepared
preparePart()
DrinkChef preparing drinks
All drinks prepared
Meal completed
serveCustomer()
Customer served
calculatePayment()
Customer paid
cleanUp()
Customer leaves


State
Unhappy
+Unhappy()
+calculateTip() : float
+changeState(customer : Customer*) : void
Neutral
+Neutral()
+calculateTip() : float
+changeState(customer : Customer*) : void
Satisfied
+Satisfied()
+calculateTip() : float
+changeState(customer : Customer*) : void
Rating
+Rating()
+calculateTip() : float
+changeState(customer : Customer*) : void
Composite
Order
-waiter : Waiter*
+Order()
+addToOrder(item : Order*) : void
+appendToOrder(orderItem : Order*) : void
+calculatePrice() : float
+getWaiter() : Waiter*
0..1
Item
#price : float
+Item(price : float)
+calculatePrice() : float
ComplexOrder
-item : Order*
-nextOrderItem : Order*
+ComplexOrder(waiter : Waiter*)
+calculatePrice() : float
+addToOrder(item : Order*) : void
+appendToOrder(order : Order*) : void
Side
+washedLettuce : boolean = false
+cutVegetables : boolean = false
+assembledSalad : boolean = false
+platedSide : boolean = false
+Side(price : float)
Drink
+gotGlass : boolean = false
+pouredDrink : boolean = false
+assembledDrink : boolean = false
+Drink(price : float)
MainMeal
+preparedMain : boolean = false
+seasonedMain : boolean = false
+cookedMain : boolean = false
+platedMain : boolean = false
+MainMeal(price : float)
Salad
+washedLettuce : boolean = false
+cutLettuce : boolean = false
+washedTomato : boolean = false
+cutTomato : boolean = false
+washedCucumber : boolean = false
+cutCucumber : boolean = false
+cutFeta : boolean = false
+assembledSalad : boolean = false
+platedSalad : boolean = false
+Salad()
Chips
+washedPotatoes : boolean = false
+cutPotatoes : boolean = false
+friedPotatoes : boolean = false
+seasonedChips : boolean = false
+Chips()
Soda
+gotSodaGlass : boolean = false
+pouredSoda : boolean = false
+assembledSoda : boolean = false
+Soda()
Water
+gotWaterGlass : boolean = false
+pouredWater : boolean = false
+assembledWater : boolean = false
+Water()
Beer
+gotBeerGlass : boolean = false
+pouredBeer : boolean = false
+assembledBeer : boolean = false
+Beer()
Burger
+cookedPatty : boolean = false
+assembledBurger : boolean = false
+butteredBun : boolean = false
+preparedVegetables : boolean = false
+Burger()
Steak
+tenderisedSteak : boolean = false
+seasonedSteak : boolean = false
+cookedSteak : boolean = false
+platedSteak : boolean = false
+Steak()
Fish
+descaleFish : boolean = false
+seasonedFish : boolean = false
+cookedFish : boolean = false
+platedFish : boolean = false
+Fish()
Client
Facade
Interface
r : Restaurant*
+Interface()
-createCustomer() : Customer*
-createRestaurant() : Restaurant*
+generateNumberOfCustomers() : int
+runCustomer() : float
+getCurrentUnixTime() : int
+generateOrderString() : string
+getRestaurant() : Restaurant*
Observer
Customer
-waiter : Waiter*
-satisfaction : Rating*
-table : Table*
-timestamp : int
-order : Order*
+Customer(timestamp : int)
+acceptWaiter(waiter : Waiter*) : void
+changeRating(rating : Rating*) : void
+receiveOrder(order : Order*) : void
+calculatePayment() : float
+getOrder() : Order*
+getOrderRequest() : string
+getTable() : Table*
+getWaiter() : Waiter*
+getTimestamp() : int
Mediator
Floor
-restaurant : Restaurant*
-tables : vector<Table*>
+Floor()
+seatCustomer(customer : Customer*) : void
+setRestaurant(restaurant : Restaurant*) : void
+getNumTables() : int
+getTable(customer : Customer*) : Table*
Restaurant
-floor : Floor*
-kitchen : Kitchen*
-waiters : vector<Waiter*>
+RestaurantInit(numTables : int)
+seatCustomer(customer : Customer*) : void
+placeOrder(orderContainer : OrderContainer*) : void
+makeNextOrder() : void
+initialise() : void
+cleanUp(customer : Customer*) : void
Table
-customer : Customer*
+Table()
+addCustomer(customer : Customer*) : void
+getCustomer() : Customer*
+cleanUp() : void
Waiter
-restaurant : Restaurant*
-customer : Customer*
+Waiter(restaurant : Restaurant*)
+visitCustomer(customer : Customer*) : void
+takeOrder(orderContainer : OrderContainer*) : void
+serveCustomer(order : Order*) : void
+getCustomer() : Customer*
+getRestaurant() : Restaurant*
+cleanUp() : void
Kitchen
-restaurant : Restaurant*
-chef : Chef*
-orderQueue : vector<OrderContainer*>
+Kitchen()
+recieveOrder(orderContainer : OrderContainer*) : void
+makeNextOrder() : void
+setRestaurant(restaurant : Restaurant*) : void
OrderContainer
-order : Order*
-o : string
+OrderContainer(o : string, order : Order*)
+getOrder() : Order*
+getRequestedOrder() : string
Chain of Responsibility
Adapter
Chef
-mainChef : Chef*
-Builders : ItemBuilder[]
+Chef()
+preparePart(order : string, o : Order*) : void
+request(order : string&) : int
MainChef
+MainChef()
+preparePart(order : string, o : Order*) : void
SideChef
+SideChef()
+preparePart(order : string, o : Order*) : void
DrinkChef
+DrinkChef()
+preparePart(order : string, o : Order*) : void
HeadChef
+HeadChef()
+preparePart(order : string, o : Order*) : void
Builder
Strategy
ItemBuilder
-item : Item*
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
Template Method
MainBuilder
#main : MainMeal*
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
+prepareMeal() : void
+seasonMeal() : void
+cookMeat() : void
+plateMain() : void
SideBuilder
#side : Side*
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
+washVegetables() : void
+chopVegetables() : void
+assembleSide() : void
+plateSide() : void
DrinkBuilder
#drink : Drink*
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
+getGlass() : void
+pourDrink() : void
+assembleDrink() : void
SteakBuilder
-steak : Steak*
+SteakBuilder()
+prepareMeat() : void
+seasonMeat() : void
+cookMeat() : void
+plateMain() : void
+tenderiseSteak() : void
+seasonSteak() : void
+cookSteak() : void
+plateSteak() : void
ChipsBuilder
-chips : Chips*
+ChipsBuilder()
+washVegetables() : void
+chopVegetables() : void
+assembleSide() : void
+plateSide() : void
+washPotato() : void
+cutPotato() : void
+fryPotato() : void
+seasonChips() : void
WaterBuilder
-water : Water*
+WaterBuilder()
+getGlass() : void
+pourDrink() : void
+assembleDrink() : void
+getWaterGlass() : void
+pourWater() : void
+assembleWater() : void
BeerBuilder
-beer : Beer*
+BeerBuilder()
+getGlass() : void
+pourDrink() : void
+assembleDrink() : void
+getBeerGlass() : void
+pourBeer() : void
+assembleBeer() : void
BurgerBuilder
-burger : Burger*
+BurgerBuilder()
+prepareMeat() : void
+seasonMeat() : void
+cookMeat() : void
+plateMain() : void
+butterBun() : void
+prepareVegetables() : void
+cookPatty() : void
+applySauce() : void
+assembleBurger() : void
SaladBuilder
-salad : Salad*
+SaladBuilder()
+washVegetables() : void
+chopVegetables() : void
+assembleSide() : void
+plateSide() : void
+washLettuce() : void
+cutLettuce() : void
+cutFeta() : void
+washTomato() : void
+cutTomato() : void
+washCucumber() : void
+cutCucumber() : void
+assembleSalad() : void
SodaBuilder
-soda : Soda*
+SodaBuilder()
+getGlass() : void
+pourDrink() : void
+assembleDrink() : void
+getSodaGlass() : void
+pourSoda() : void
+assembleSoda() : void
FishBuilder
-fish : Fish*
+FishBuilder()
+prepareMeat() : void
+seasonMeat() : void
+cookMeat() : void
+plateMain() : void
+descaleFish() : void
+seasonFish() : void
+cookFish() : void
+plateFish() : void

References

Escoffier. *https://www.escoffier.edu/blog/culinary-pastry-careers/executive-chef-vs-head-chef-what-is-the-diference/#:~:text=Executive%20chefs%20will%20likely%20be%20responsible%20for%20keeping%20an%20eye,budgeting%20for%20staff%20and%20equipment.* 2023. Website. 5 November 2023. <https://www.escoffier.edu/blog/culinary-pastry-careers/executive-chef-vs-head-chef-what-is-the-diference/#:~:text=Executive%20chefs%20will%20likely%20be%20responsible%20for%20keeping%20an%20eye,budgeting%20for%20staff%20and%20equipment.>.

Hospitality Jobs Africa. *https://hospitalityjobsafrica.com/executive-head-chef-job-description/#:~:text=Responsible%20directly%20for%20overseeing%20and,the%20Food%20and%20Beverage%20Manager.* n.d. Website. 5 November 2023. <https://hospitalityjobsafrica.com/executive-head-chef-job-description/#:~:text=Responsible%20directly%20for%20overseeing%20and,the%20Food%20and%20Beverage%20Manager.>.

Join.com. *https://join.com/job-descriptions/waiter-waitress#:~:text=The%20Waiter%20or%20Waitress%20will,fast%20food%20chains%20or%20bars.* 2023. Website. 5 November 2023. <https://join.com/job-descriptions/waiter-waitress#:~:text=The%20Waiter%20or%20Waitress%20will,fast%20food%20chains%20or%20bars.>.

National Restaurant Association. *https://restaurant.org/education-and-resources/learning-center/workforce-engagement/restaurant-industry-job-descriptions/*. 2023. Website. 5 November 2023. <https://restaurant.org/education-and-resources/learning-center/workforce-engagement/restaurant-industry-job-descriptions/>.

Wikipedia. *Wikipedia: Brigade de cuisine*. n.d. Website. 5 November 2023.

Workable. *https://resources.workable.com/restaurant-manager-job-description#:~:text=Restaurant%20Managers%20take%20on%20many,safety%20regulations%20and%20manage%20inventory.* 2023. Wesbite. 5 November 2023. <https://resources.workable.com/restaurant-manager-job-description#:~:text=Restaurant%20Managers%20take%20on%20many,safety%20regulations%20and%20manage%20inventory.>.