COS 214 Project Report

SEG-FAULT FOODS

Click here for our repo!

Cheat GBT

Given Chauke	u21595969
Zion van Wyk	u21655325
Nerina Borchard	u21537144
Siyabonga Mbuyisa	u20491621
Lesedi Kekana	u20486473
Atidaishe Mupanemunda	u22747886
Thabiso Matau	u22609254

Table of Contents

Research

A restaurant is a business that prepares and sells a variety of food and beverages. Restaurants may follow a theme or only serve a specific cuisine.

Types of Restaurants

- Fast Food Restaurants:
    - Quick-service restaurants that offer counter service and/or drive-through service. These tend to be chain restaurants/franchises with high profit margins.
    - Menu options tend to be limited (may exclusively be a variety of burgers, pizzas or other food items).
    - Examples: McDonalds, Burger King

- Casual Restaurants:
    - Guests are seated and waited on in restaurants like these. They offer full-service dining.
    - Menu options tend to be diverse but familiar to the average restaurant-goer. They offer classic items (e.g. sandwiches) and comfort foods (e.g. soups).
    - Examples: Mugg n Bean

- Fine Dining Restaurants:
    - These restaurants offer extreme formality and items tend to be on the pricey side. This style of dining is very nuanced. Many rules apply to fine dining (e.g. dress code, strict eating etiquette).
    - Menu options tend to be pricey, exclusive items like caviar and steak tartare.
    - Examples: Kream, Roma Revolving Restaurant

- Cafe or Coffee Shop:
    - This restaurant focuses on serving a variety of beverages. These range from warm to cold beverages. They tend to have small bakeries serving pastries and snacks like sandwiches and croissants. Service can either be counter-based or customers can be waited on.
    - Menu options tend to range from basic/classic beverages (e.g. cappuccino) to novelty beverages (e.g. themed frappuccinos).
    - Examples: Vida e Caffe, Starbucks

- Specialty Drink Shops:
    - These shops sell specialised drinks that aren't common items you'd find at a cafe such as bubble tea, fresh juices or smoothies. They tend to have a trendy atmosphere and items range from affordable to slightly overpriced.
    - Menu options tend to be limited with the markup on drinks being very high. Service at these shops tend to be counter based.

    - Examples: Kauai, Ben's Bubble Tea

- Buffet Restaurants:
    - Buffet restaurants offer an all-you-can-eat style of dining. Customers serve themselves from a wide range of available options laid out in the restaurant. A set standard price is charged to dine at a buffet.
    - These restaurants rely on high volumes of people to make a profit and hence, have large dining rooms with lots of available seating.
    - Menu options tend to be dominated by comfort foods.
    - Examples: Manhattan Hotel, Tatso

- Food Trucks:
    - These are mobile restaurants that offer popular or niche foods to events, parks, businesses and markets. They are compact kitchens on wheels with few employees that offer window service.
    - Menu options tend to be limited. Food trucks stick to comfort or convenience foods (such as burgers or tacos).
    - Examples:

Restaurant Layouts

Different types of restaurants follow different physical layouts. For example, a cafe may only include the kitchen, counter and a small seating area, whereas a fine dining restaurant may include many more elements.

In general, a restaurant in which guests are waited on should have the following elements as a part of its floor plan:

- Kitchen
- Tables & chairs
- Entrance
- Waiting area
- Bar area (if applicable)
- Restrooms
- Staff area
- Payment systems
- Windows
- Doors
- Emergency exits

Restaurants must ask themselves many questions before finalising their floor plan. The following list provides a few examples of things that may affect the layout of a restaurant:

1. How much privacy do our guests require? (Affects spacing of tables, barriers between tables and types of table layouts that could be used - e.g. booths)
2. How long are the shifts of our employees? Will they need a private area to change clothes or take a break?
3. How visible should the kitchen be to the guests? Should they be allowed to watch the chef prepare their steak?
4. Will the restaurant offer outdoor seating? Will our seating be accessible to the disabled?
5. How big should the waiting area be? Will we be busy enough to have to offer seating to guests who wait for a table?
6. How natural is the flow of our layout? Will our waiters find it easy to get to their tables from the kitchen and payment system?

Types of Restaurant Employees

A restaurant may require a variety of staff to run it, depending on the type of restaurant it is. Here are positions that exist in large restaurants.

Administrative Staff:

- *Manager*
    - This person is responsible for running the restaurant's operations, appearance, administrative decisions and marketing strategies. This employee has the most responsibility at a legal level.
    - Should be educated in administrative studies with knowledge of hospitality.
    - There can be levels to the manager position.

Kitchen Staff:

- *Executive/Head Chef*
    - This employee tailors the menu, directs kitchen staff to their stations and makes administrative decisions about what the restaurant serves.
    - This role requires contact with every other member of kitchen staff.
    - Must have formal culinary education.
- *Kitchen Manager*
    - This position can be absorbed by the head chef or may be standalone.
    - The kitchen manager is in charge of checking inventory and ensuring that all ingredients are available at all times.
- *Sous-chef*
    - These chefs are assistants to the executive chef and must listen to all the head chef’s orders. They must be experienced and have a similar set of skills to the executive chef.
    - Traditionally, the executive chef is responsible for filling the sous-chef position with an employee of choice.
- *Prep Cook*
    - This position is important in restaurants with a high volume of customers or that boast quick service.
    - Prep cooks prepare all menu items for the chefs as quickly and efficiently as possible to make the cooking process faster. This ensures that meals are cooked easily and within a reasonable time.
- *Line Cook*
    - Line cooks handle multiple areas of the kitchen and help streamline the work of other cooks and chefs.
- *Short Order Cook*
    - These employees take care of small or “short” orders such as meals for breakfasts and brunches (e.g. sandwiches, burgers and other light foods)
- *Cleaning Team*
    - These members of kitchen staff are those that wash dishes and clean the tools needed by chefs. They also ensure that the kitchen is clean in general.
    - This position’s importance increases with the size of the restaurant.

Front-of-House Staff:

- *Head Waiter*
    - Also known as Maitre d'
    - These staff members are in charge of training new waiters, give direct orders to waiters and are attentive to the needs of all diners.
    - Employees should be trained in service/customer care.
- *Waiters*
    - The role of waiters is one of the most important in a restaurant.
    - Waiters are in charge of bringing orders to the kitchen, delivering ready orders to guests, cleaning and rearranging tables once guests have left and finalising payment of any customer bills.
    - Waiters should have excellent customer service and communication skills. They should be prepared to receive complaints from guests and deal with them appropriately.
- *Sommelier*
    - These are wine experts that recommend wines appropriate for specific dishes.
    - This employee must be in direct contact with the chefs to pair wines with dishes on the menu.
- *Receptionist*
    - These employees verify and plan for reservations, regulating the entry and exit of guests.
    - Employees should have excellent communication and exceptional presentation.
- *Bar Staff*
    - These include bartenders, bartender assistants and baristas.
    - These positions are generally filled by waiters who have been trained in beverage preparation.

How to Start a Restaurant

1. Pick a Restaurant Type

   First, you need to decide on what kind of service you'd like to offer as a restaurant. Do you want to open a cafe or a fine dining establishment?

2. Create a Business Plan

   Map out your plan for the restaurant. Describe the company, give your restaurant a name, do analysis on your competition, predict your financial projects and organise your staffing plan.

3. Choose a Good Location

   After analysing the market and the competition, choose a location that would suit your businesses' needs the best. For example, a food truck may do well in a park with a lot of foot-traffic. Choose a place where you'll make money.

4. Design the Menu

   Keep your restaurant's image, seasonal ingredients and ingredient costs in mind when designing your menu. Explore various food and beverage options.

5. Prepare Restaurant Costs

   Costs to keep in mind are:

   - Rent
   - Remodelling the property
   - Equipment (Fridges, fryers, dishwashers etc.)
   - Furniture
   - Technology (Payment system)
   - Staff wages

6. Fund the Restaurant

   You may use your own personal funds to pour into the restaurant, however, there is a lot of help available. Grants from the government or culinary schools are available. You may also apply for a business loan to start up your restaurant. If you're really lucky, you may be able to partner up with a private investor that will invest in your business for a portion of the profits.

7. Obtain Licenses & Permits

   You may need a license to operate in certain areas and a permit to sell alcohol on your premises. Do research into what you'll need in terms of paperwork so that you can get ahead in the process. Depending on the restaurant you open up, you may also need health certifications.

8. Hire Qualified Staff

   Ensure that you have a list of employees needed for your restaurant and a thorough interview process in place before trying to hire anyone. When hiring, ensure that employees filling important positions are appropriately trained/formally educated.

References

- https://webstaurantstore.com/article/353/types-of-restaurants.html [Accessed: 05/11/2023]
- https://en.wikipedia.org/wiki/Restaurant [Accessed: 05/11/2023]
- https://pos.toasttab.com/blog/on-the-line/restaurant-floor-plans [Accessed: 05/11/2023]
- https://foyr.com/learn/restaurant-floor-plan/ [Accessed: 05/11/2023]
- https://www.socialtables.com/blog/hospitality/restaurant-layout/ [Accessed: 05/11/2023]
- https://www.waiterio.com/blog/en/restaurant-staff-list/ [Accessed: 05/11/2023]
- https://aptito.com/blog/restaurant-positions-and-descriptions [Accessed: 05/11/2023]
- https://www.escoffier.edu/blog/food-entrepreneurship/how-to-start-a-restaurant-with-little-to-no-money/ [Accessed: 05/11/2023]
- https://www.lightspeedhq.com/blog/how-to-run-restaurant/ [Accessed: 05/11/2023]
- Images:
    - https://beanonline.co.za/wp-content/uploads/2022/06/MB.jpg [Accessed 06/11/2023]
    - https://cdn-ehepc.nitrocdn.com/irBzRYkBekzJBgZevpNPMGNeqdaqURgx/assets/images/optimized/rev-3ca277b/hwmiia.fra1.digitaloceanspaces.com/wp-content/uploads/2021/07/09123842/Vida-Mauritius-new-800x450-1.jpg [Accessed: 06/11/2023]
    - https://www.bensbubbletea.co.za/cdn/shop/files/Screen_Shot_2023-05-23_at_12.51.41_PM_1500x.png?v=1684839127 [Accessed: 06/11/2023]
    - https://s3da-design.com/wp-content/uploads/2022/03/Restaurant-floor-plan.jpg [Accessed:06/11/2023]

Design Decisions

In order to explain our design decisions, below is a step-by-step list of how our group went about planning and designing our project.

1. Research:
    - All of the above research was conducted, summarised and presented to the group by our group leader.
    - Once each group member understood how restaurants worked, what employees are necessary to run a restaurant as well as the systems that should be implemented, we moved onto the next phase.

2. Choosing Restaurant Type:
    - We unanimously decided to stick to a casual restaurant as it is the most common type of restaurant that can be found and is easily understood by all members.

3. Dividing Restaurant into Subsystems:
    - After choosing to implement a casual restaurant, we zoomed in on the elements of what makes a casual restaurant.
    - The following subsystems were identified:
        i. Assigning guest to table
        ii. Processing order of guests
            1. Taking order
            2. Delegating orders to chefs
            3. Delivering order
            4. Rating order
        iii. Processing & payment of bill
            1. Generating bill for group of customers

4. Once we had our subsystems, we discussed what their functionality would be and applied design patterns we thought were best suited to them. These patterns, the reasoning for their implementation and comparatives can be found in the next section.

Functional Requirements

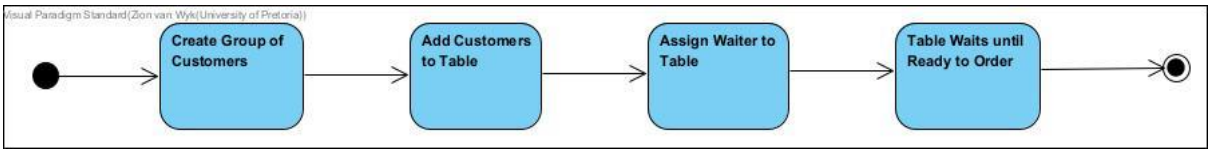
Admitting Customers:

- A group of Customers may acquire a Table.
- Tables may be grouped together.
- Restrictions:
    - No more than 5 customers per Table.
    - Total of 20 Tables make up the Floor.

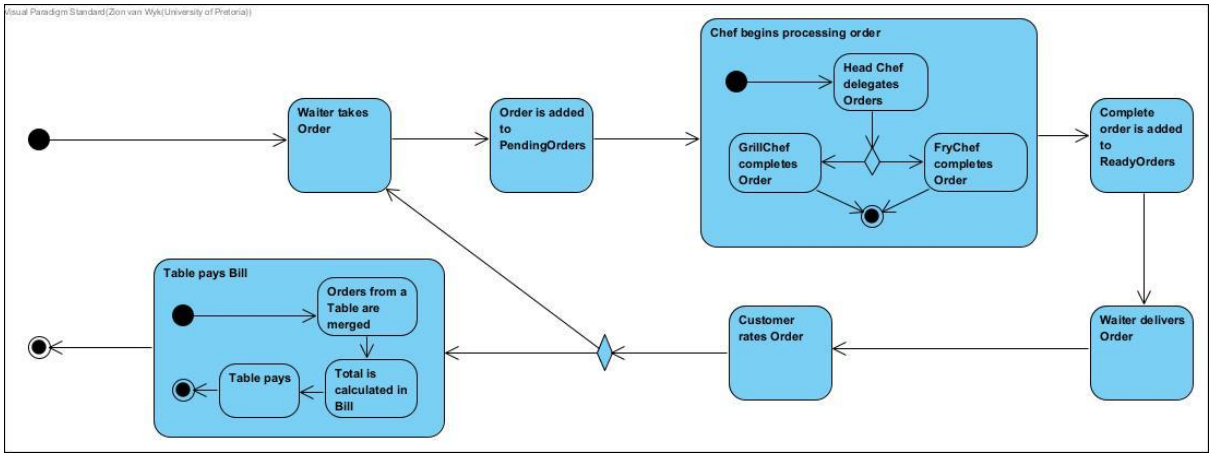
Processing Orders

- Waiter may take an Order if a Table's state is ReadyToOrder
    - If Tables are grouped together, the grouped Table's Order is taken.
- Kitchen may begin processing Orders once the Waiter has added order to the PendingOrders queue.
    - Head chef delegates dishes to Grill Chef or Fry Chef
- Kitchen may add order to ReadyOrders queue once processed.
- Waiter may only deliver Orders in the ReadyOrders queue.
- Customers may rate dishes once Order has been delivered.
    - Customer state may change once Order has been delivered.
- Customers may order again.
- Table may pay the bill once Table state is in PayBill.
    - All orders from a single table are merged into one bill.
    - Total is calculated in a Bill object.
- Customers may leave once the bill is paid.

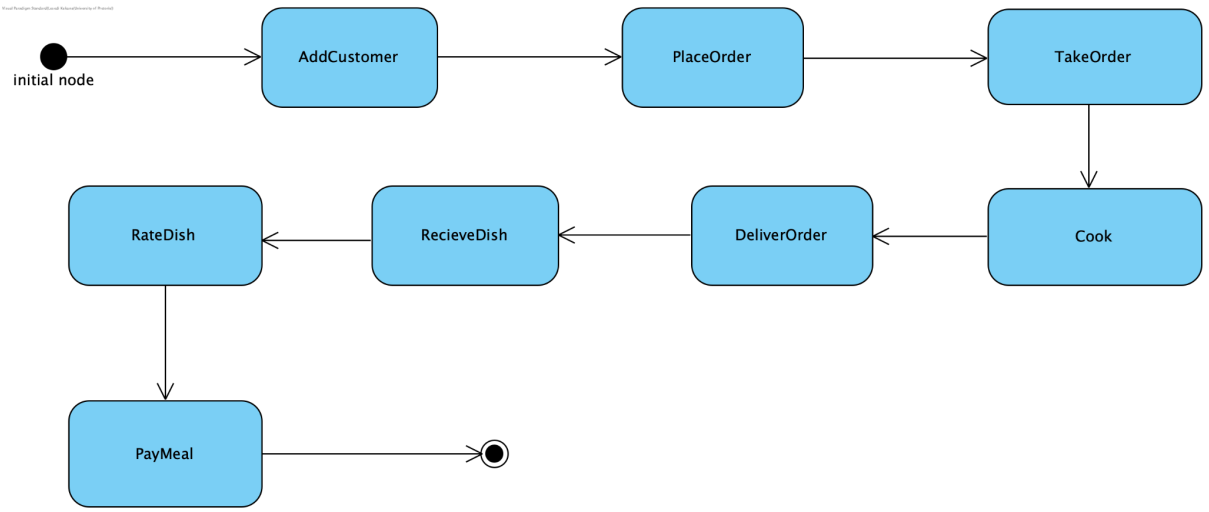
Activity Diagrams showing Processes



Admitting Customers subsystem



Processing Orders subsystem



Program flow

Planned Patterns

In our project, we've implemented various design patterns to improve the organisation and functionality of our codebase. Discussed below is our list of patterns and how we used them for various processes within our system.

Composite

Our tables follow the Composite pattern with AbstractTable as the Component, Table as the Leaf and CombinedTable as the Composite. This functionality allows us to combine tables if groups of customers exceed the maximum number of people allowed at one table.

State

A few classes follow the State pattern to indicate when certain actions should be performed:

- TableState (State), with AbstractTable as its Context, has the following ConcreteStates that indicate which actions the customers at the Table are ready to take: NotOccupied, Waiting, NotReadyToOrder, ReadyToOrder and PayBill.
- CustomerState (State), with Customer as its Context, has the following ConcreteStates that are influenced by the service/food: Happy, Neutral, Angry.
- OrderStatus (State), with Order as its Context, has the following ConcreteStates that indicate the state of a customer's order: Received, Processing, Ready.

Iterator

To ensure a consistent program flow, without using concurrency, we use the Iterator design pattern to control the flow of activities by the Employees. Whether that's chefs preparing

food or waiters checking on orders in the kitchen or checking on customers. The employees will perform tasks based each iteration on it by the Floor.

- With AbstractTable as the Aggregate that we will iterate over and Table as a class that implements its interface (Concrete Aggregate)
- A generic Iterator interface and a specific iterator that implements the interface for the Table iteration process. (ConcreteIterator)

Bridge

The Department class follows the Bridge pattern by bridging the Employees to their respective departments. Department is the Implementor, with KitchenDepartment and FloorDepartment as the ConcreteImplementors, that create a bridge to Employee (Abstraction) with Chef, Waiter and Manager as RefinedAbstractions.

Memento

Memento will be used to keep record of all Bills for the restaurant's bookkeeping. Bill (Originator) stores the order and calculates the total cost for the bill. BillMemento (Memento) then stores the state of the bill. All of these bills are then stored in BillCaretaker for future reference.

Observer

Observer is used to model how the Waiters will keep track of the Orders they take and notify the Kitchen about. KitchenNotifier is the Subject with OrderSubject as its ConcreteSubject. WaiterObserver is the Observer with Waiter as its ConcreteObserver.

Chain of Responsibility

The Chef class makes use of Chain of Responsibility to indicate the different Chefs available to work on Orders. Chef is the Handler while GrillChef, FryerChef and HeadChef are the ConcreteHandlers. This will pass a dish onto various chefs for each step in the cooking process necessary for each dish.

Strategy

Chef also makes use of Strategy as GrillChef, FryerChef and HeadChef (ConcreteStrategies) use similar algorithms inherited from the Chef (Strategy) class. The various implementations of the Chef class will produce specific implementations of the cook() function unique to their role within the cooking process.

Template Method

The Chef class can also be considered a Template as the skeleton of Chef is used to define its children classes. With Chef as the AbstractClass and GrillChef, FryerChef and HeadChef as the ConcreteClasses.

Decorator

We use the Decorator pattern to add variety to our dishes by allowing customers to choose different types of pizzas, burgers and add toppings to their pizzas. The Food class is our Component. This signifies a generic dish for which we will use decorators to make into speciality dishes like Pizza, Pasta, etc.

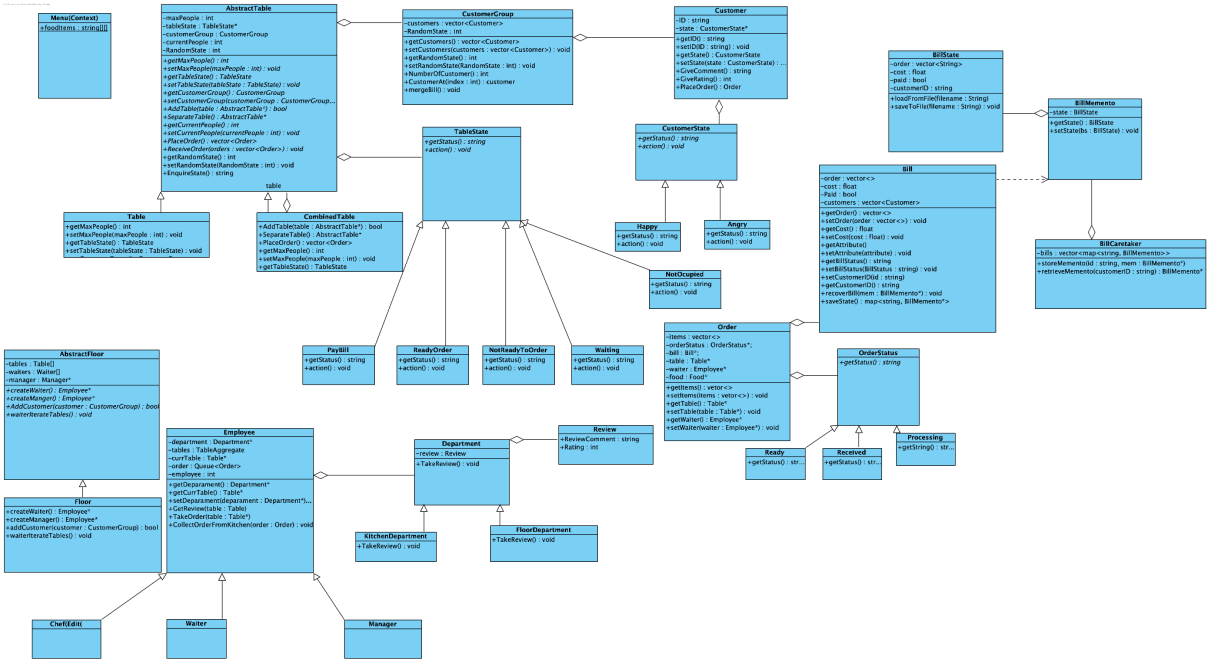
Builder

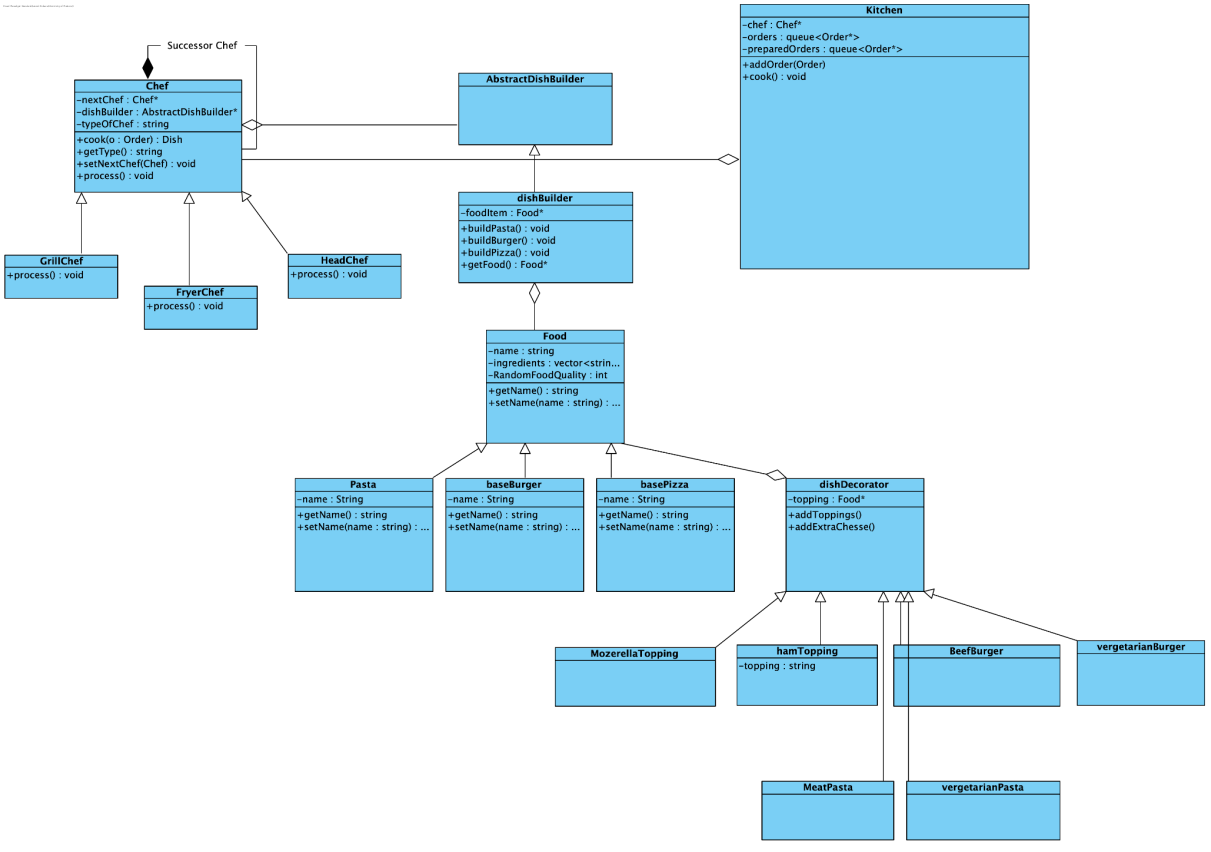
The Builder pattern is used to create dishes. AbstractDishBuilder is the Builder, dishBuilder is the ConcreteBuilder, Chef is the Director and Food will be our Product.

Façade

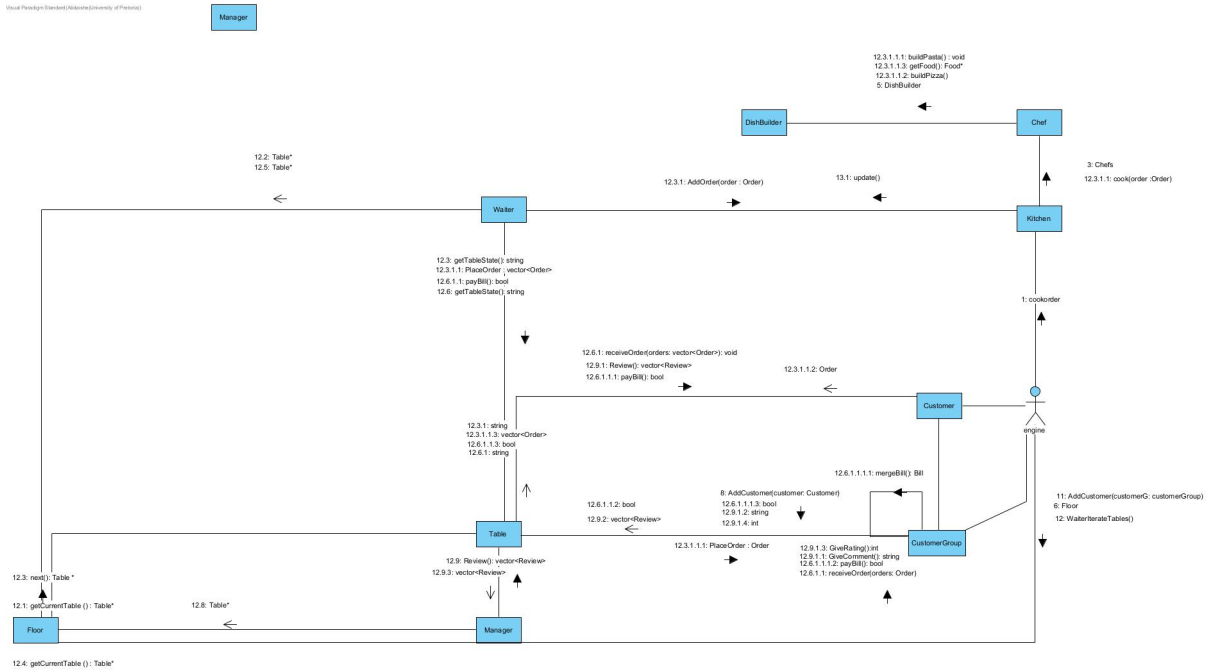
We will use the Façade pattern as our central engine for tying all sections of our code, the main sections being the Employees, Customers, Kitchen and Floor into a single class to reduce the complexities of using all of our many projected classes at once.

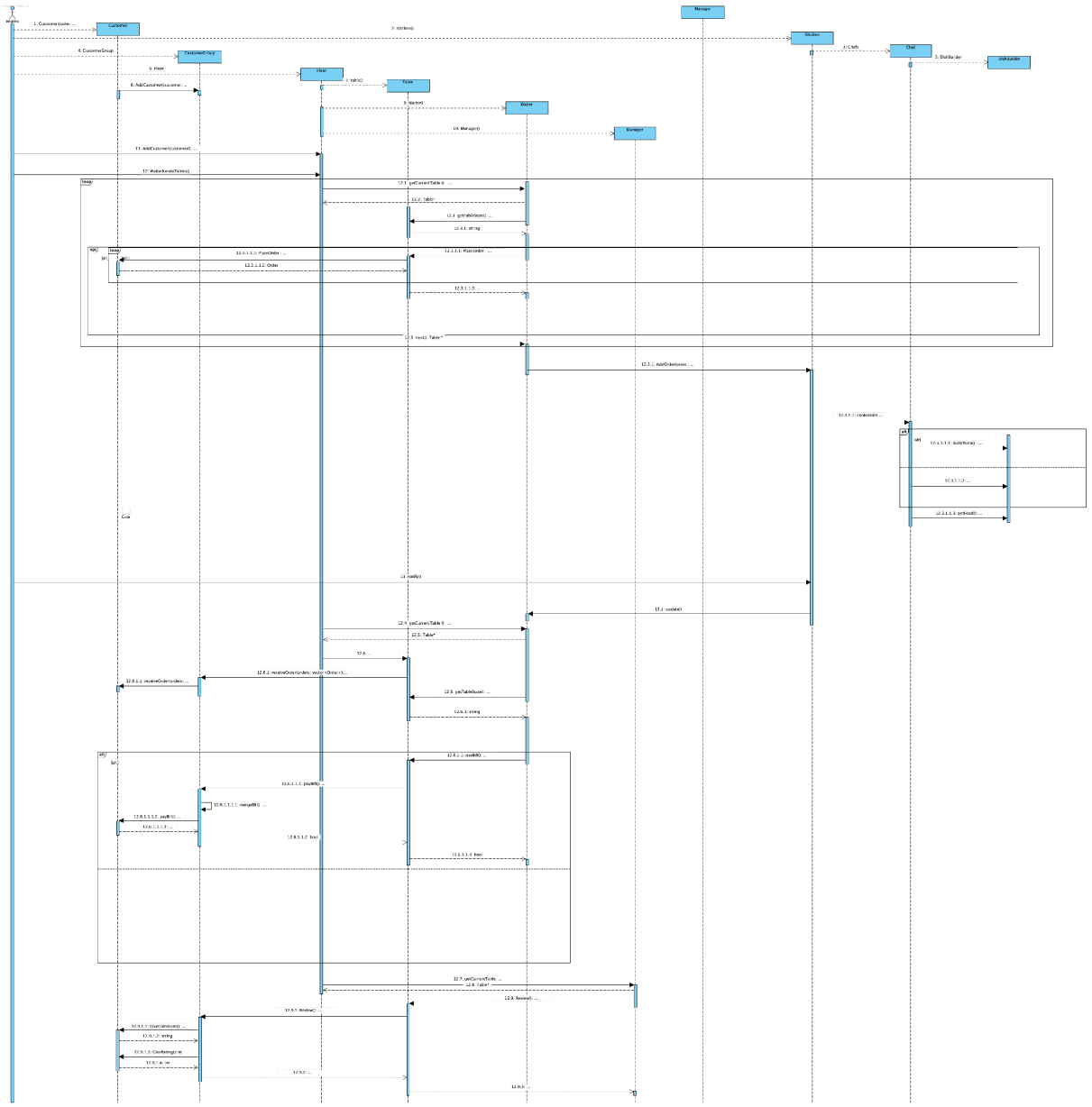
Initial Draft UML Class Diagram



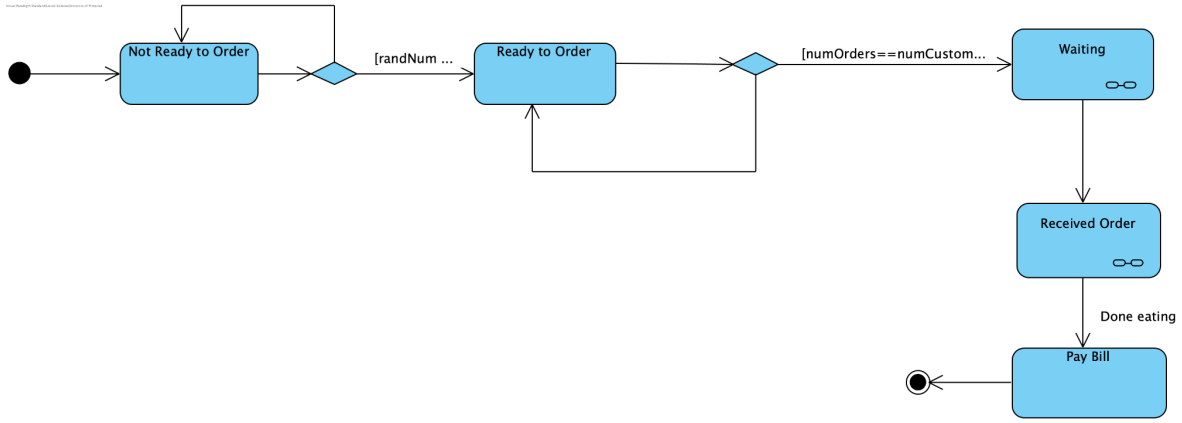


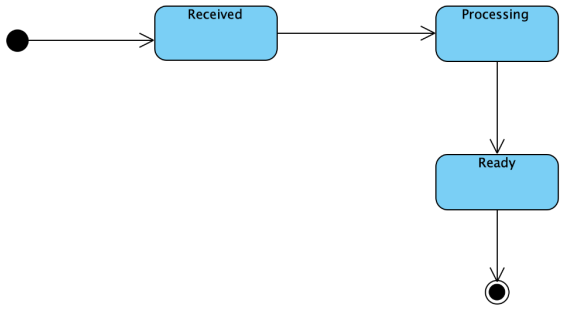
Example of Message passing through System

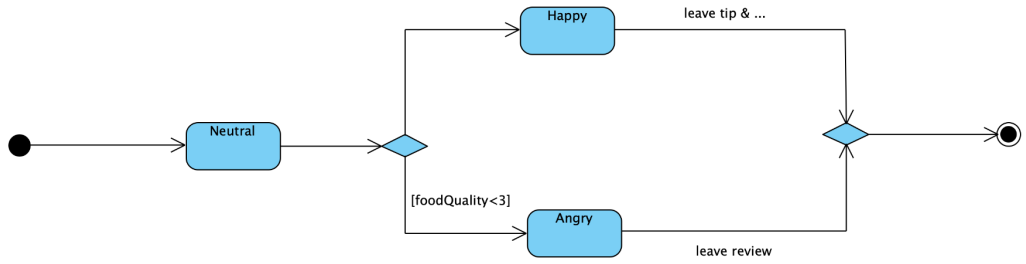


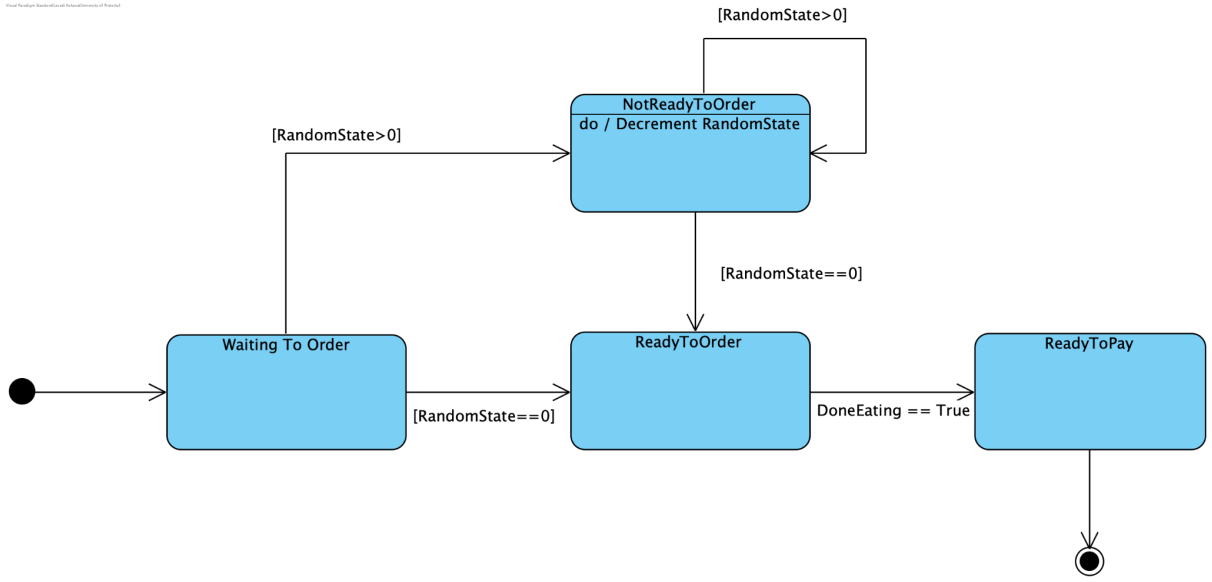

Customer
Manager
Kitchen
Chef
DishBuilder
CustomerGroup
Floor
Table
Waiter
Manager
1: CustomerInvite: ...
2: dishes()
3: Cook()
4: CustomerGroup
5: Floor
6: AddCustomerGuscomer: ...
7: tables()
8: Waiter()
9: Manager()
11: AddCustomerGroupCustomerG...
12: MakeWaiterGoTables()
loop
12.1: getCurrentTable () : ...
12.2: Table*
12.3: getTableStatus() : ...
12.4.1: string
opt
Happy
[alt]
12.4.1.1.1: PlaceOrder (...
12.3.1.1.2: Order
12.3.1.1.3: ...
12.3: next(): Table *
12.3.1: AddOrderInKitchen : ...
12.3.1.1: createOrder : ...
alt
12.3.1.1.1: buildCurry() : ...
12.3.1.1.2: getMeal() : ...
Cook
13: notify()
13.1: notified()
12.4: getCurrentTable () : ...
12.5: Table*
12.6: ...
12.6.2.1: receiveOrderInOrders: vector<Order>*...
12.6.2.2: receiveOrderInOrders: ...
12.6: getTableStatus() : ...
12.6.1: string
alt
[alt]
12.6.1.1.1: payBill() : ...
12.6.1.1.1.1: mergeBill() : ...
12.6.1.1.2: payBill() : ...
12.6.1.1.2 : ...
12.6.1.1.2: bool
12.6.1.1.4: bool
12.7: getCurrentTable : ...
12.8: Table* : ...
12.9: ReviewS : ...
12.9.1: ReviewS()
12.9.1.1: ChefCommand() : ...
12.9.1.2: string
12.9.1.3: CookedOrder
12.9.1.4: int
12.9.2: ...
12.9.3: ...

Examples of the following Objects changing State over time in the System

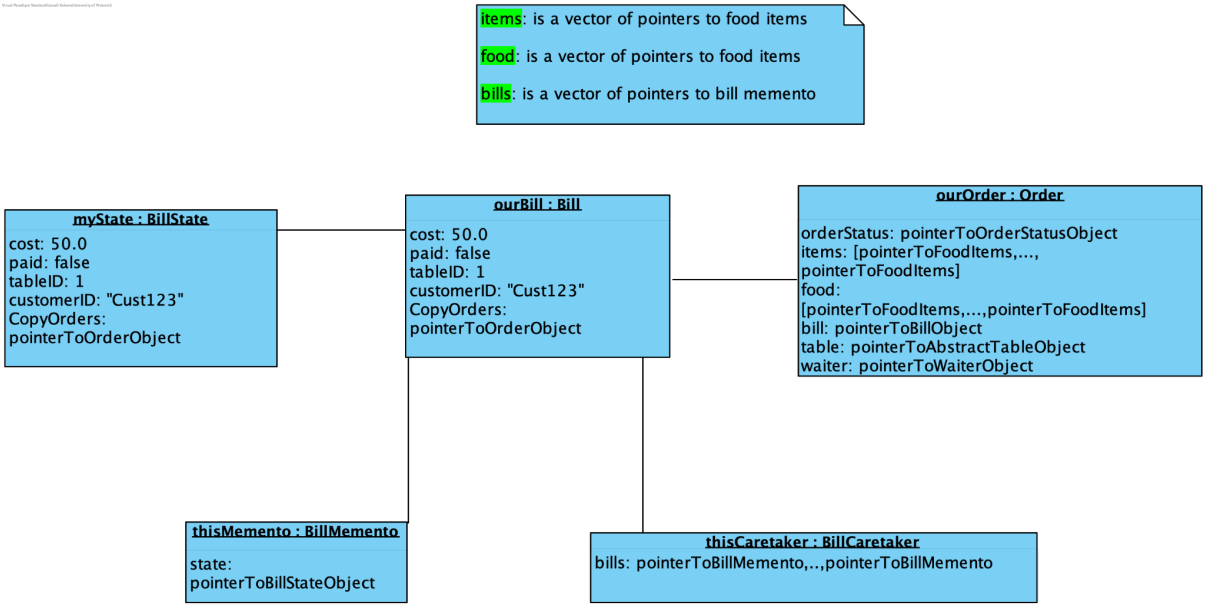








Example of Object in System as it runs



Application of Design Patterns

Composite

- AbstractTable (*Component*) will represent the common interface for all table types.
- Table (*Leaf*) will be a concrete table with a specific capacity.
- CombinedTable (*Composite*) will allow tables to be combined when groups exceed the maximum capacity.

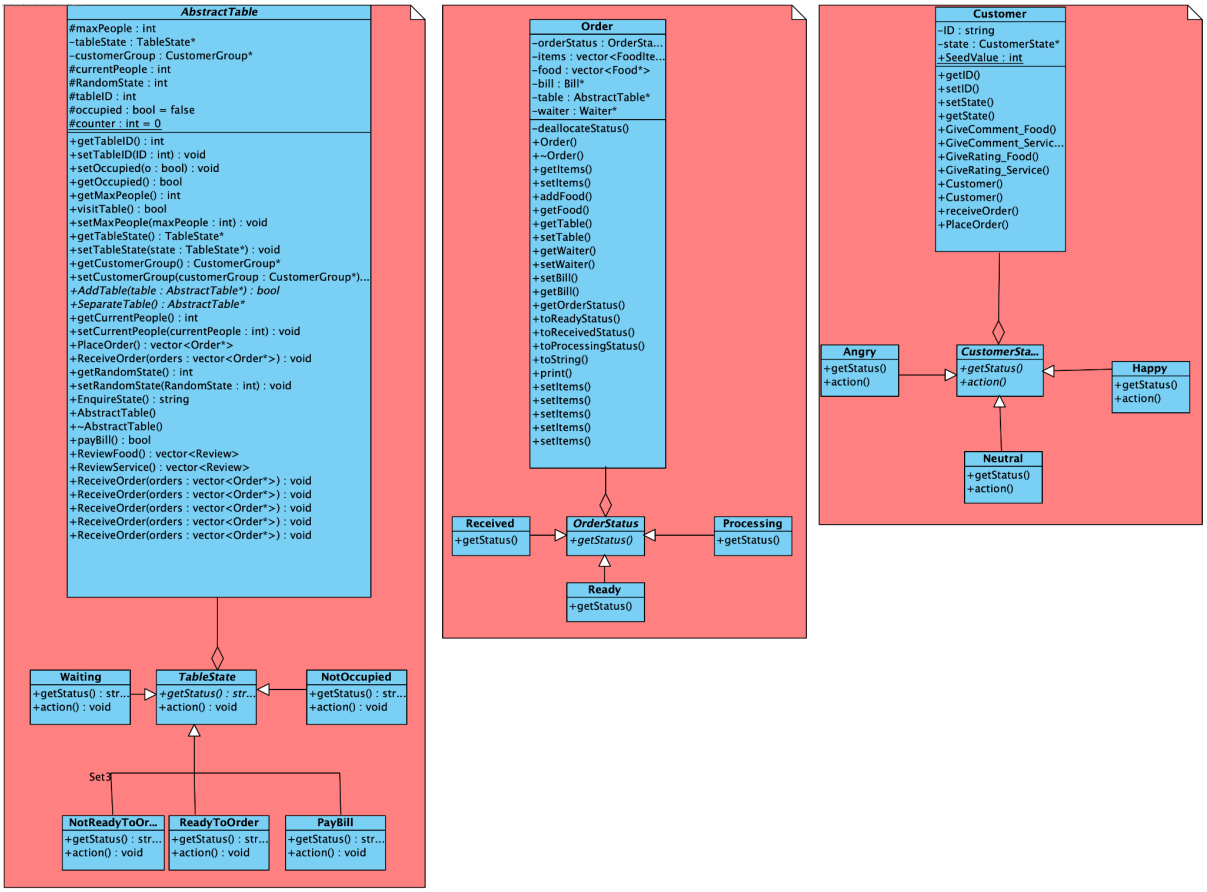
Design decision:
This structure allows us to group tables when necessary, accommodating groups of customers exceeding the maximum table capacity. This pattern assists in managing table allocation efficiently.

Assumptions: None. The classes perform full validation for all possible inputs and other classes base their assumptions on this class working correctly.

Alterations:

- Added an attribute to indicate if a table is occupied.
- Added a function to check if TableState is properly created.
- Added a function `visitTable` to see if the Table is ready to order (returns true); if not applicable, applies pressure.

State



1. **TableState** (*State*) with NotOccupied, Waiting, NotReadyToOrder, ReadyToOrder and PayBill (*Concrete Participants*) to manage the state of tables
2. **CustomerState** (*State*) with Happy, Neutral and Angry (*Concrete Participants*) to represent customer satisfaction.
3. **OrderStatus** (*State*) with Received, Processing and Ready (*Concrete Participants*) to track order progress.

Design decisions: This pattern helps us manage and control various states and their corresponding actions.

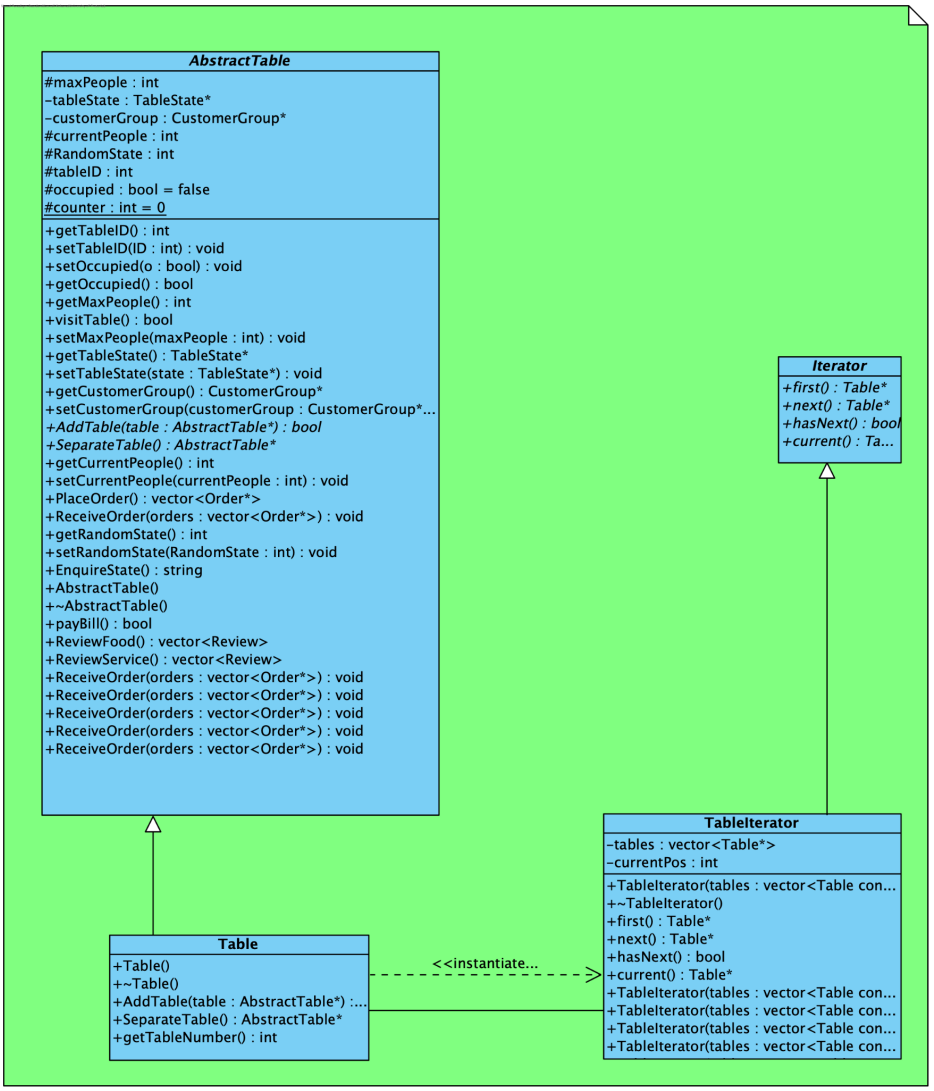
Assumptions:
The states identified are the only states necessary for each of the classes implemented.

Alterations:
None.

Iterator

Iterator will be used for Floor and Employee objects, allowing traversal through tables and staff members.

- Iterator
- Concrete Iterator
- Aggregate
- Concrete Aggregate



Design decisions:

- The Iterator pattern is chosen to provide a consistent way to iterate through collections of Floor and Employee objects, such as tables and staff members.
- Iterators will be implemented for Floor and Employee collections, providing methods like "next," "hasNext," and "current" to traverse the objects.
- The Iterator pattern simplifies the code by abstracting the traversal logic from the concrete collections.

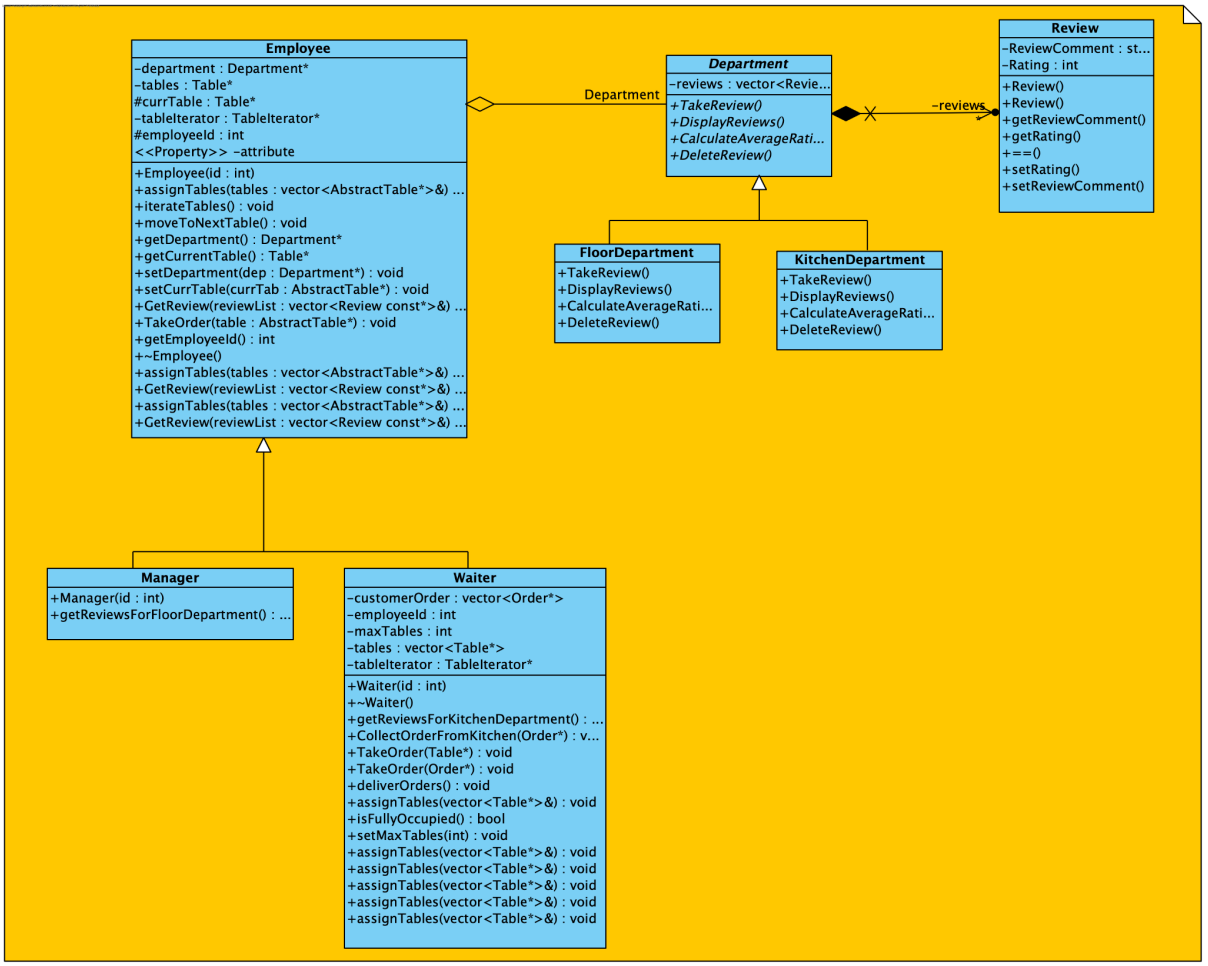
Assumptions:

- The Floor and Employee objects need to be iterated for various operations like customer seating, order management, and staff assignments.
- The Iterator pattern enhances the system's flexibility, making it easier to navigate and manipulate these collections.
- The Iterator pattern will be consistent across different parts of the system, making it more maintainable and easier to extend.

Alterations:

- In the Abstract Class:
    - `static int counter;`
    - `int tableID;`
    - `bool occupied=false;`
    - These three attributes were added to the class.
    - `int counter` was added to give each table a unique ID.
    - `tableID` was added to give each table a unique ID.
    - `occupied` is used to check if a table is occupied.
    - As a result, getters and setters were added for these attributes.
- Redesigned the entire design for Iterator and renamed classes to:
    - AbstractTable
    - Table
    - TableIterator
    - Iterator

Bridge



- Department (*Implementor*) bridges Employees to their respective departments.
- KitchenDepartment and FloorDepartment (*Concrete Implementors*) handle specific department logic.
- Employee (*Abstraction*) is bridged by Department, with Chef, Waiter, and Manager (*Refined Abstractions*)

Design decisions:

- The Bridge pattern is chosen to decouple the abstraction (Employee) from its implementation (Department) to allow flexibility in adding new departments or employees without modifying existing code.
- KitchenDepartment and FloorDepartment are used as ConcreteImplementors to handle department-specific operations.
- The Employee class is the Abstraction, allowing employees to be assigned to different departments.
- The Bridge pattern simplifies the relationship between employees and departments, making it easier to maintain and expand the system.

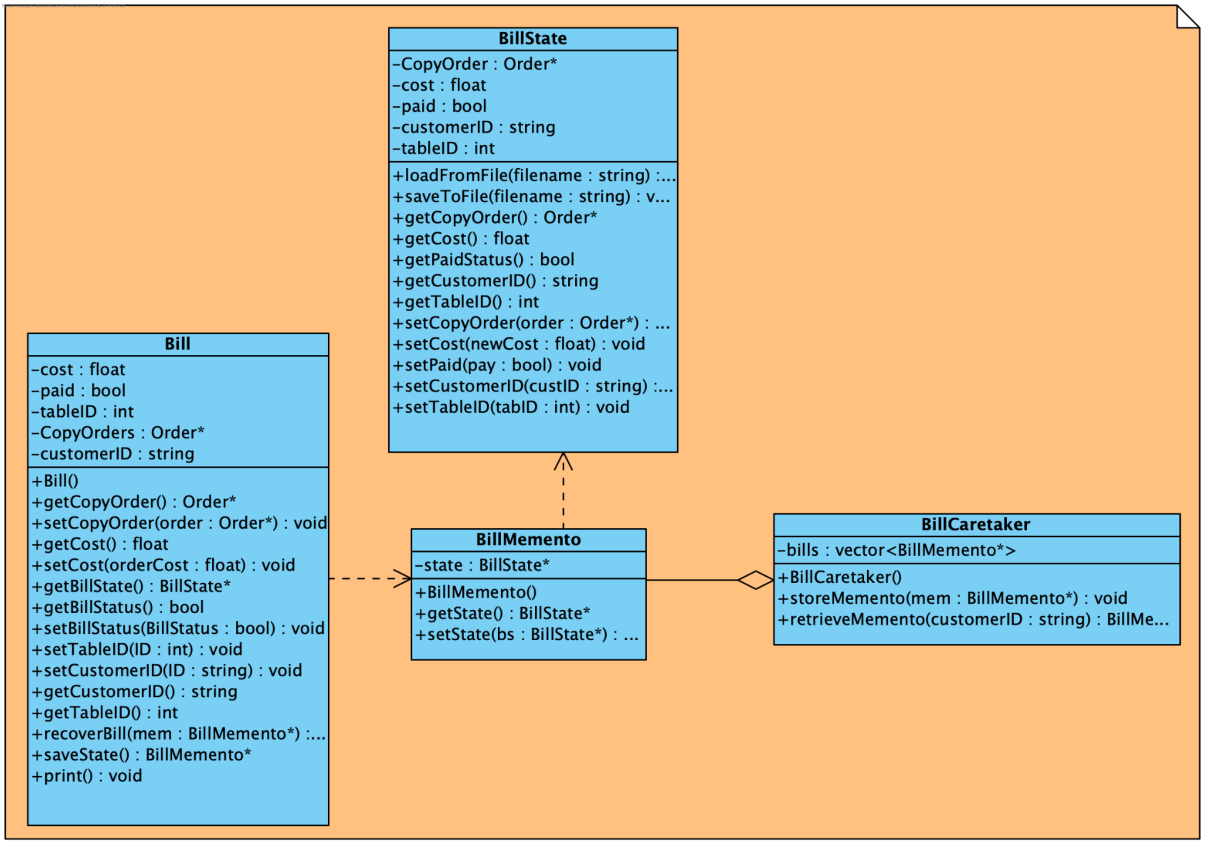
Assumptions:

- The Employee hierarchy includes Chef, Waiter, and Manager, which will be connected to specific departments (KitchenDepartment and FloorDepartment) through the Bridge pattern.
- The Bridge pattern provides an extensible way to manage employees, and new employee types can be added without altering the existing department or employee classes.
- The Employee hierarchy may grow in the future, and this pattern will continue to facilitate the relationship between employees and departments.

Alterations:

- Added the following functions to separate the implementation of review from Employee:
    - `TakeReview(review : const Review) : void`
    - `DisplayReviews() : void`
    - `CalculateAverageRating() : double`
    - `DeleteReview(review : const Review) : void
- Waiter:
    - Added attributes (max table, vector of tables).
    - Used a vector instead of a queue for Orders.
    - Added a function `assignTables` to add tables to a waiter's existing tables and set an iterator for the table.
    - Added `isFullyOccupied` to check if a waiter has the maximum possible tables.

Memento



Memento will be used to store bill information for bookkeeping.

- Bill (*Originator*) stores order details and calculates the total cost.
- BillMemento (*Memento*) stores the state of the bill.
- BillCaretaker (*Caretaker*) manages the storage of bill mementos for reference.

Design decisions:

This pattern was chosen in order to keep track of the restaurant's finances by storing all proof of income from paid orders.

Assumptions:

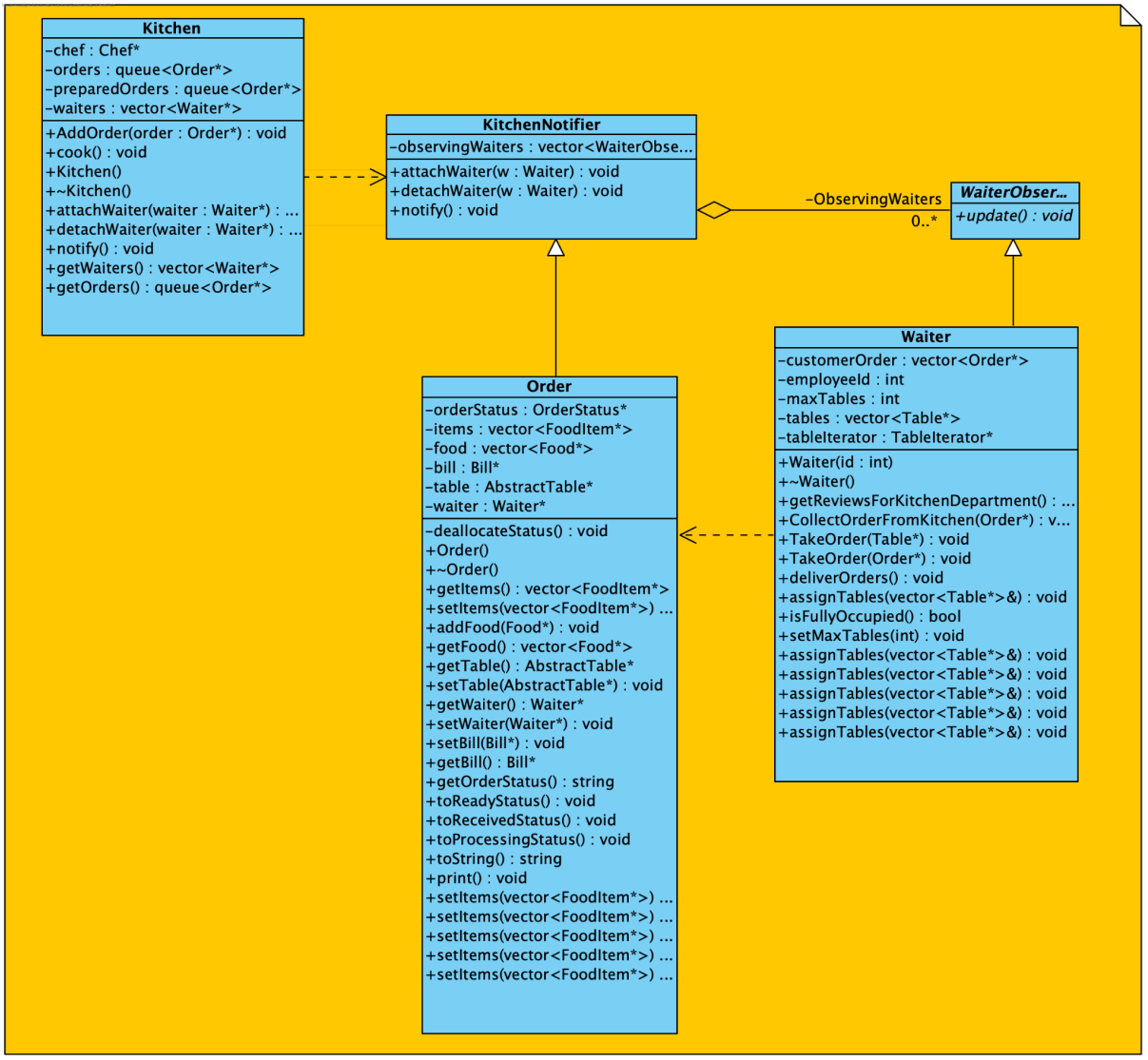
None.

Alterations:

- In the Bill Class and BillState:
  - `order : vector<string>` was changed to `CopyOrder: Order*` (a deep copy).
  - `customers : vector<Customer>` was changed to `customerID: string` (a bill can belong to one customer).
  - `tableID : int` was also added. When waiters are delivering, they need to know the table.
  - As a result, getters and setters were added.

- In the BillCareTaker:
    - `bills : vector<BillMemento>` was changed to `bills: vector<BillMemento*>` (shallow copies of objects with pointers).

Observer



Observer will be employed to notify the kitchen about new orders.

- KitchenNotifier (*Subject*) with OrderSubject (*ConcreteSubject*) to keep track of orders.
- WaiterObserver (*Observer*) with Waiter (*ConcreteObserver*) to observe and report new orders

Design decisions:
The Observer pattern is implemented to enable waiters to keep track of orders and notify the kitchen. This pattern aids in real-time order management and communication.

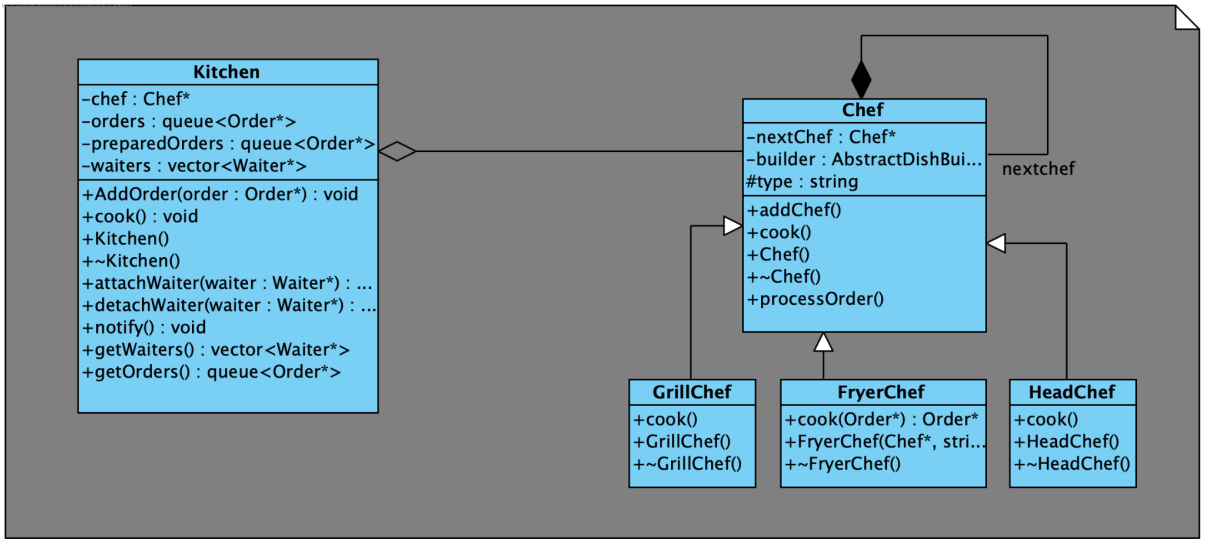
Assumptions:

- The Kitchen expects orders from waiters, which contain a vector of food items, including the name, price, preparation method, and type of food.
- The Kitchen passes this to the `cook` method and then calls the chain of chefs, giving it to the chef in charge of a certain preparation.

Alterations:

- Removed `orderNotifiers` vector.
- Added constructors and destructors.
- Added `attachWaiter()`, `detachWaiter()`, `notify()`, `getWaiters()`, and `getOrders()`.

Chain of Responsibility



Chain of Responsibility is used in the Chef class to assign orders to different chefs.

- Chef (*Handler*) processes orders, with GrillChef, FryerChef, and HeadChef as *ConcreteHandlers*.

Note: Strategy and Template patterns use the same UML as the Chain of Responsibility.

Strategy

Strategy will be applied within the Chef class to provide different cooking strategies.

- Chef (*Strategy*) defines the interface, and GrillChef, FryerChef, and HeadChef (*Concrete Strategies*) implement specific cooking methods.

Template Method

Chef class can be seen as a Template with a skeleton algorithm structure.

- Chef (*Abstract Class*) defines the cooking process, with GrillChef, FryerChef, and HeadChef (*Concrete Classes*) providing concrete implementations.

Design decisions: For Chain of Responsibility,we use it to manage the cooking process, we employ the Chain of Responsibility pattern within the Chef class. This pattern helps in passing dishes through various chefs at each stage of the cooking process.

For Strategy, the pattern allows different chef roles to inherit similar cooking algorithms from the Chef class while implementing specific cook() functions unique to their roles.

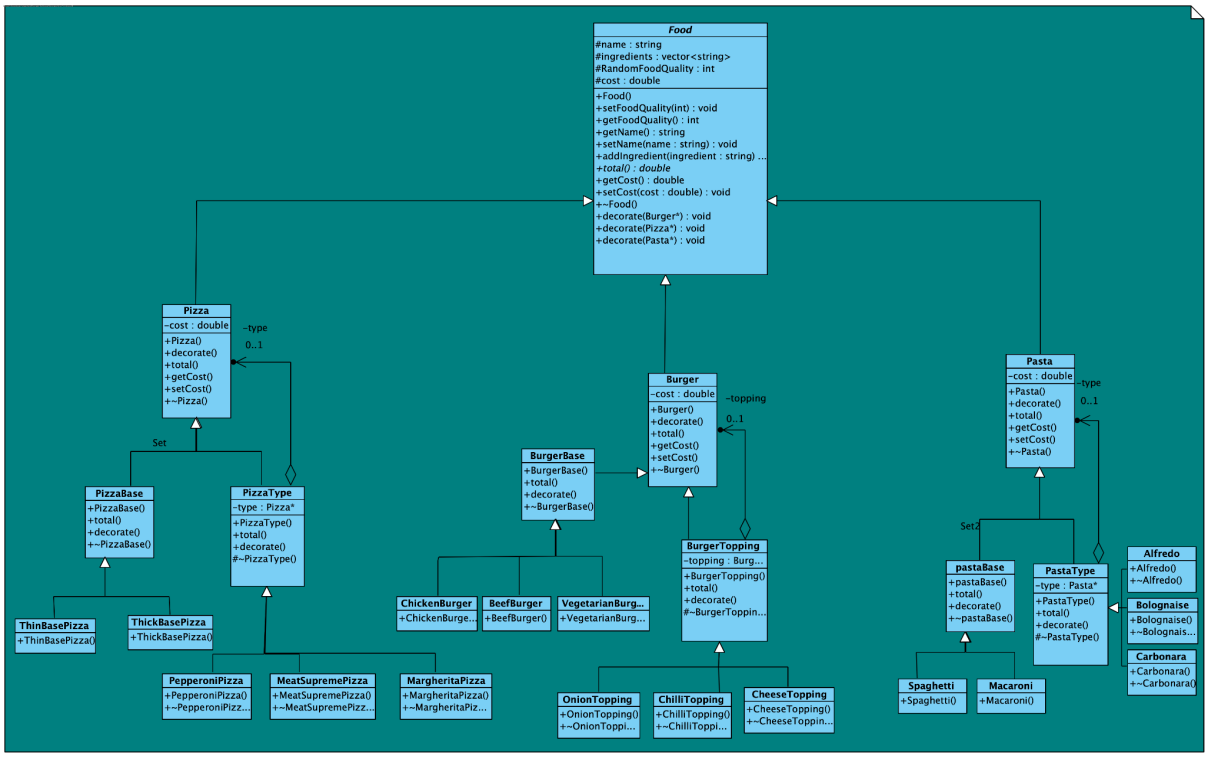
For Template Method, we will use this pattern to outlines the structure for chef-related processes, which are similar for stages of cooking and replace inner unique steps for each chef with different algorithms.

Assumptions:

Alterations:

- Chef:
    - Made the `type` member private.
    - Changed `process()` to `processOrder()`.
    - Removed `getType()`, `setNextChef()`, and `chefChainHandler()`.
    - Added `addChef()`, constructor, and destructor for Chef.
- HeadChef, GrillChef, and FryerChef:
    - Added `cook()`, constructors, and destructors for all.
    - Removed `chefChainHandler()`.

Decorator



Decorator will be used to allow customization of dishes, such as adding toppings to pizzas.

- Food is an abstract class that represents the base food item from which the Components implement
- Pizza, Burger and Pasta (*Component participants*) are the 3 types of food that our restaurant offers.

Design decisions:

Initially, we considered using the decorator pattern to add extras and toppings to food items. However, to streamline the integration of decorators and to maintain a clear separation of concerns, we switched to the builder pattern. The builder pattern allowed us to build a food item and then decorate it as needed, achieving a more elegant and extensible solution.

Assumptions:

- Each order will be sent to the builder and then the decorator to create the correct order.

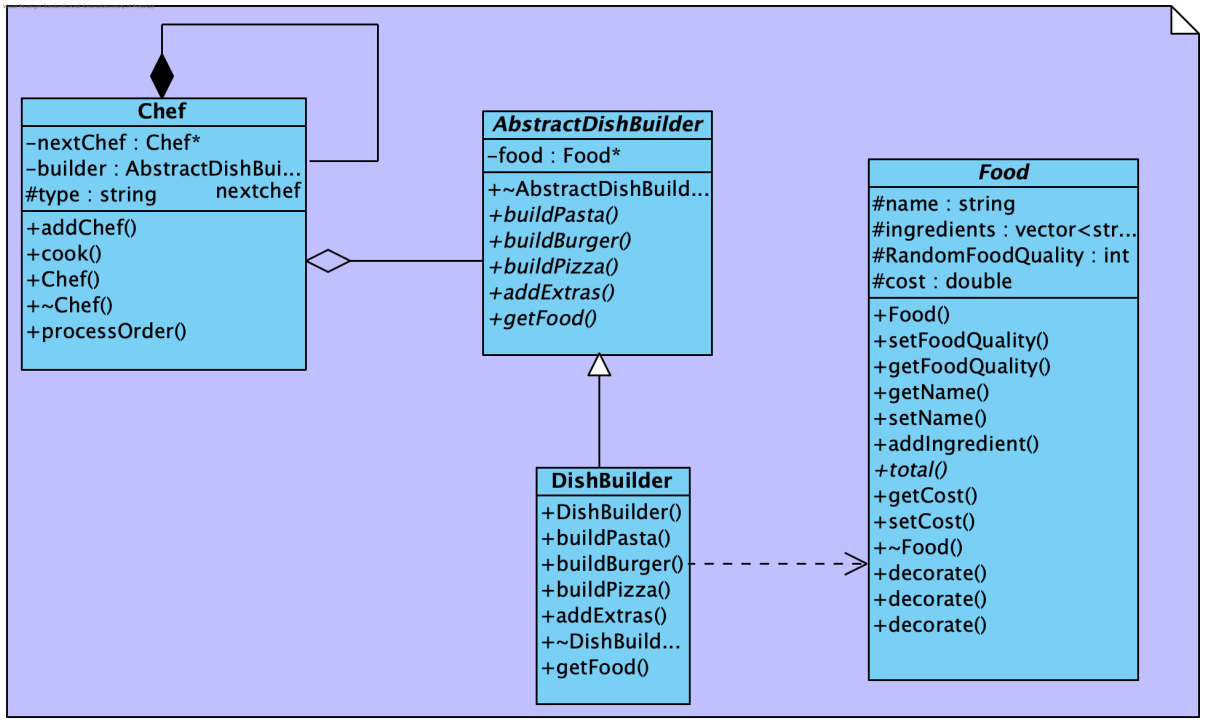
Alterations:

- Changed the class layout so that there are basically 3 decorators for Pizza, Pasta, and Burger (expanded as it was missing a lot of classes in the original).
- Added/changed to `Pizza()`, `Burger()`, `Pasta()`, `PizzaBase()`, `PastaBase()`, `BurgerBase()`, `PizzaType()`, `BurgerTopping()`, `PastaType()`, `Spaghetti()`, `Macaroni()`, `Alfredo()`, `Carbonara()`, `Bolognaise()`, `ThickBasePizza()`,

`ThinBasePizza()`, `PepperoniPizza()`, `MeatSupremePizza()`, `MargaritaPizza()`, `ChickenBurger()`, `BeefBurger()`, `VegetarianBurger()`, `OnionTopping()`, `CheeseTopping()`, `ChilliTopping()`.

- All these functions decorate the dish the way the customer ordered.
- Added necessary functions for each class.

Builder



Builder pattern is used to create dishes with an abstract builder.

- AbstractDishBuilder (*Builder*) defines the building process, with dishBuilder (*Concrete Builder*) constructing specific dishes.
- Chef (*Director*) directs the construction of dishes, and Food (*Product*) represents the final dish.

Design decisions:

We chose the builder pattern to create food items because it provides a structured and step-by-step approach to construct complex objects. Each concrete builder (e.g., PizzaBuilder, PastaBuilder, BurgerBuilder) is responsible for building a specific type of food. This separation of responsibilities allows us to create food items with different combinations of ingredients and toppings easily.

Assumptions:

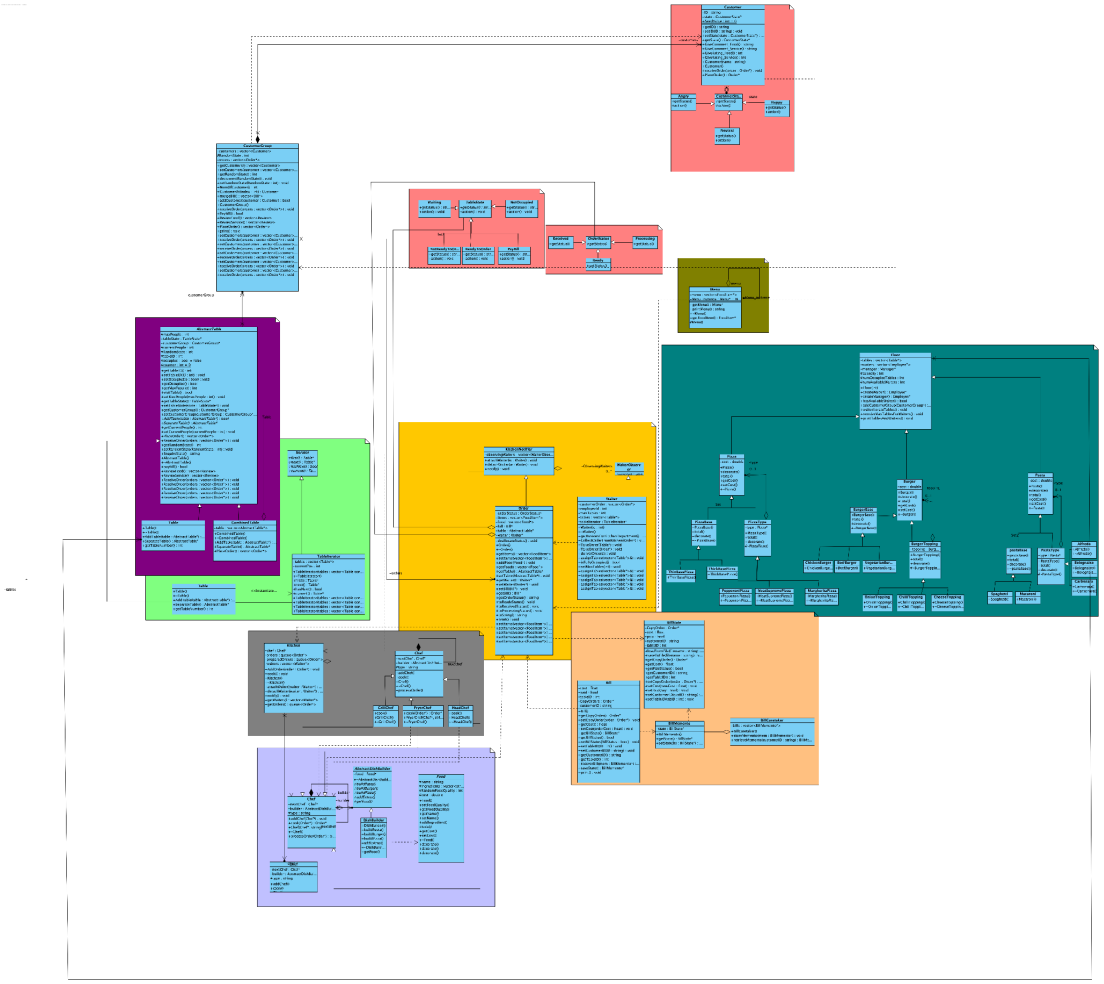
- Orders will be received through the `cook` method, containing a vector of food items used to create the specific dishes required for each customer.
- Each order will be sent to the builder and then the decorator to create the correct order.

Alterations:

- Added constructor and destructor for `AbstractDishBuilder`.
- Removed all functions in the first UML as they weren't the same ones in the builder class.

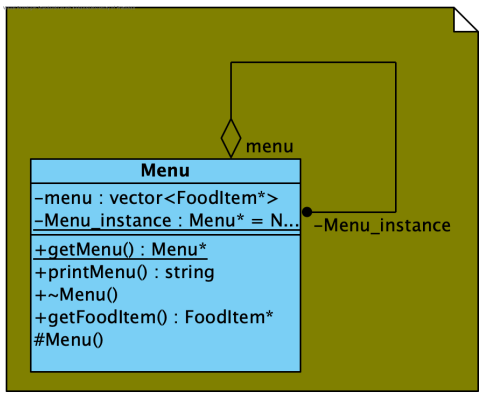
- Added `buildPasta()`, `buildPizza()`, `buildBurger()`, `addExtras()`, and `getFood()` as the code calls for the decorators.
- Added a food pointer.
- dishBuilder:
    - Added constructor and destructor for `DishBuilder`.
    - Added `addExtras()` and `getFood()`.
    - Removed the food pointer.
- Food:
    - Made all members private.
    - Added a `cost` member.
    - Added `decorate(Burger*)`, `decorate(Pizza*)`, `decorate(Pasta*)`, `setFoodQuality()`, `getFoodQuality()`, `setCost()`, `getCost()`, and `total()`.
    - Added constructor and destructor for `Food`.

Facade



The Engine class acts as a facade that provides an interactive interface for the restaurant simulation. It enables various operations like adding customers, processing orders, and handling bill payments, making the restaurant's features easily accessible.

Singleton



The menu uses the singleton pattern. This class was used to encapsulate a vector of foodItem (which are presented to the user as a struct). FoodItems structs, have a name, price, type of food and a method of preparation, all necessary for the generation of food, served to the user.

These design patterns complement the requirements and processes of the restaurant simulation system, enhancing its modularity, flexibility, and maintainability while adhering to the identified design patterns.

Final Class diagram