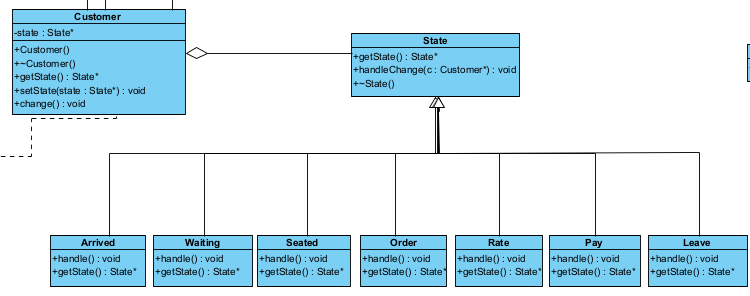
**COS 214 Task 4 Report**

**Task 4.2 – Design Decisions with reasoning**

1. Object-Oriented Design: The system seems to be designed using an object-oriented approach. This is evident from the different entities identified such as Customers, Waiters, Tables, and Kitchen which can be represented as classes in the system.
2. Customizability: The system allows for a high degree of customizability. This is seen in the ability for customers to customize their orders and request special cooking instructions. This design choice enhances the user experience and makes the simulation more realistic.
3. Modularity: The system is divided into distinct modules such as Customer Management, Waiter Management, Table Management, Kitchen Management, and Inventory Management. This design choice allows for better organization of code and easier maintenance and updates.
4. Communication between modules: There’s a clear communication protocol between different parts of the system. For example, waiters pass orders from customers to the kitchen and then serve the order when it’s ready. This design choice ensures smooth operation of the restaurant.
5. Error Handling and Reporting: The system is designed to handle unexpected scenarios and provide error messages. This design choice improves the robustness of the system and helps in troubleshooting issues.
6. Inventory Management: The system keeps track of ingredients used for dishes and updates the inventory accordingly. This design choice is crucial for avoiding situations where a dish cannot be prepared due to lack of ingredients.
7. Customer Satisfactions: The system allows customers to rate their experience at the restaurant. This design choice provides valuable feedback that can be used to improve the service.
8. Testing and Debugging:
   1. Unit Testing: The decision to have each person do unit testing on their own code ensures that individual components of the system work as expected before they are integrated. This can help catch and fix bugs early in the development process.
   2. Testing Framework: Creating a testing framework helps ensure the correctness of the code. It provides a structured way to write and run tests, making it easier to verify that the system behaves as expected under different conditions.
   3. Code Review Process: Having an admin approve merges on GitHub rather than allowing all users to automatically merge enforces a code review process. This can help catch potential issues and improve code quality.
9. Documentation:
   1. System Documentation: Documenting the system, its components, assumptions, and method of utilization helps users understand how to use the system and developers understand how to maintain or extend it.
   2. Diagram Tracking: Keeping track of all diagrams (like UML Class, Activity etc.) that helped simplify complex coding scenarios provides a visual representation of the system. This can make it easier to understand the system’s structure and behaviour.

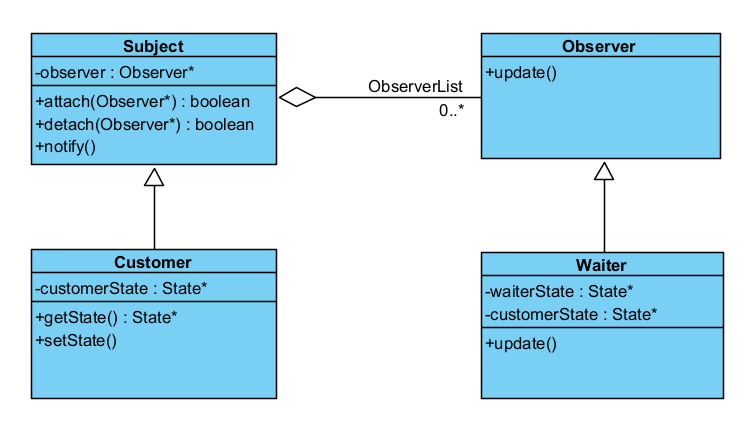
**Task 4.3 - A writeup showing how all the patterns have been used, also note what problem the pattern solved in your implementation**

**The State design Pattern**

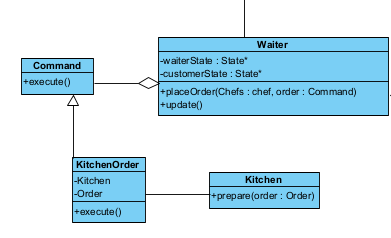


The state pattern allows us to alter the behaviour of certain functions depending on how the customer is feeling at a certain moment in time. This easily solves the issue of modelling a customer’s behaviour and is easily extendible meaning more states can be added if necessary.

**Observer**

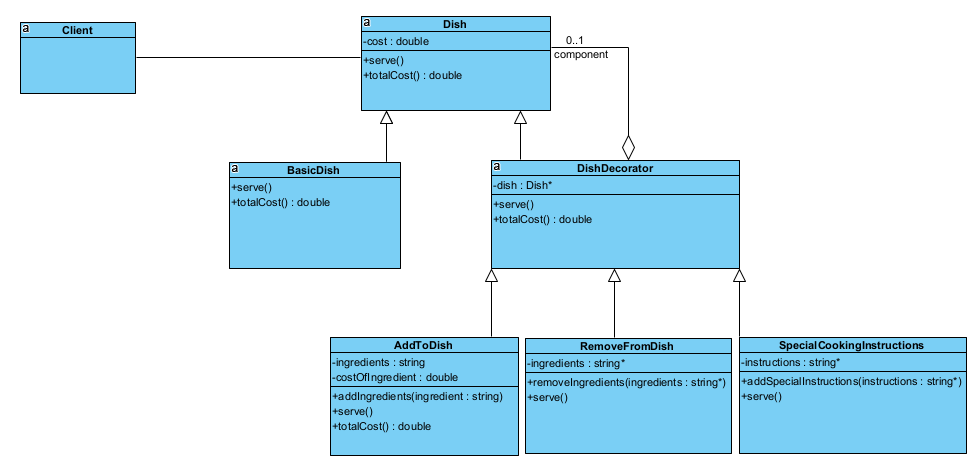


1. **Decoupling**: The Observer pattern allows you to decouple the customer (subject) from the waiter (observer). This means that the customer doesn’t need to know anything about the waiter, it just sends notifications when its state changes. This makes your code easier to maintain and extend.
2. **Dynamic Relationships**: In a restaurant scenario, the relationship between customers and waiters can be dynamic. A customer can have one or more waiters during their visit, and a waiter can serve multiple customers. The Observer pattern allows you to easily add or remove observers (waiters), supporting this dynamic relationship.
3. **State Changes**: In a restaurant, there are many state changes that a customer goes through - being seated, placing an order, receiving food, asking for the bill, etc. Each of these state changes might require interaction with a waiter. By making the waiter an observer of the customer, you ensure that the waiter is notified of these state changes and can react accordingly.
4. **Efficiency**: The Observer pattern can make your code more efficient. Instead of the waiter constantly checking if the customer needs something, the customer notifies the waiter when there’s something to do.

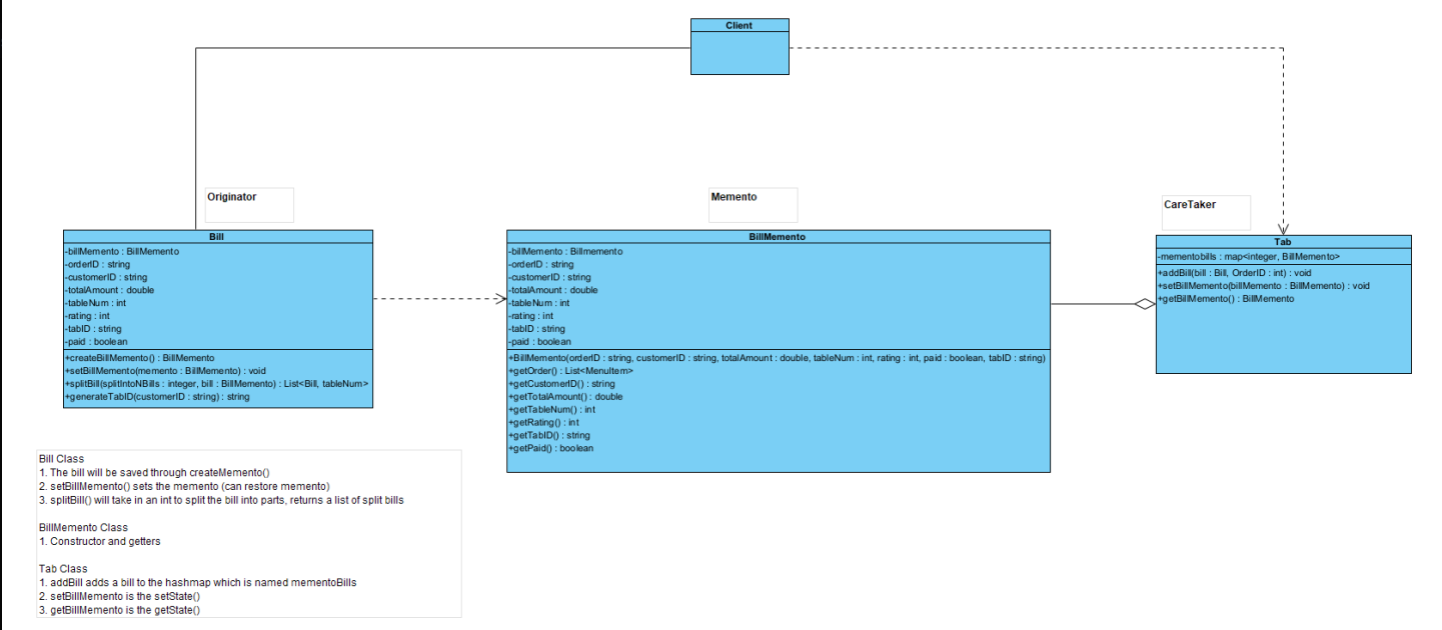
**Command**

1. **Order Handling**: The Command pattern can encapsulate each food order as a command. This allows the system to handle orders as discrete objects that contain all the information needed to prepare the dish. This can simplify the process of passing orders from waiters to the kitchen.
2. **Queue Management**: If the kitchen gets busy and cannot handle all orders immediately, the Command pattern allows you to easily queue orders. The kitchen can then process these commands in the order they were received or according to any other criteria (like order preparation time, priority of tables, etc.).
3. **Order Modification**: If a customer wants to modify their order (like adding or removing an item), you can create a new command that represents the modification and add it to the queue. This way, you don’t need to change existing commands, which might already be in process.
4. **Error Recovery**: If there’s an error in preparing an order (like a dish getting burnt or a wrong ingredient used), you can use the Command pattern to “undo” the command and start it again.
5. **Logging and Debugging**: By representing each order as a command, you can easily log all orders processed by the kitchen. This can be useful for debugging issues, analysing kitchen performance, or even resolving disputes with customers about their orders.

**Decorator**

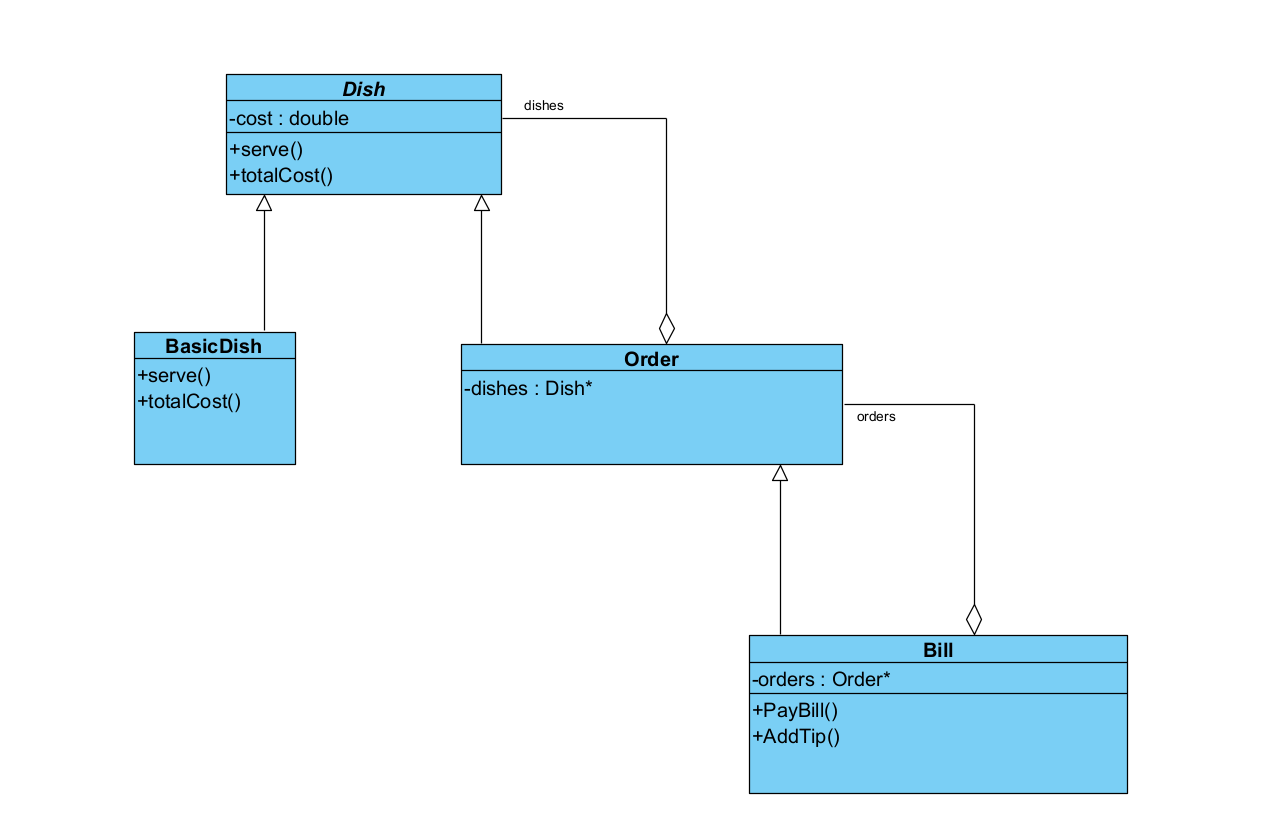


1. **Variability**: In a restaurant, dishes can have many variations. A customer might add extra cheese, remove onions, or double the meat. The Decorator pattern allows you to add or remove these “decorations” dynamically, giving you the flexibility to represent a wide variety of dishes with a single object.
2. **Simplicity**: Without the Decorator pattern, you might need to create a separate class for each possible combination of ingredients. This could quickly become unmanageable. With the Decorator pattern, you can create simple classes for each ingredient and combine them to create complex dishes.
3. **Cost Calculation**: Each addition or removal could affect the cost of the dish. The Decorator pattern makes it easy to calculate the total cost. Each decorator can add its own cost to the base cost of the dish.
4. **Description Generation**: Similarly, each decorator can contribute to the description of the dish. For example, if a customer orders a burger with extra cheese and no onions, your system could dynamically generate a description like “Burger with extra cheese and no onions”.

**Memento**

1. Undo/Redo Operations: If a customer changes their order (like adding or removing an item), you might need to update the bill. If the customer then changes their mind again, you might need to revert the bill to a previous state. The Memento pattern allows you to easily implement this undo/redo functionality.
2. Preserving State: In a restaurant, a bill can go through many states during a customer’s visit. The Memento pattern allows you to save these states and restore them if needed. This can be useful for handling errors or exceptions.
3. Simplicity: The Memento pattern simplifies the originator class (in this case, the bill). The originator doesn’t need to worry about saving its state or managing undo/redo operations. It just creates mementos and lets a caretaker manage them.
4. Encapsulation: The Memento pattern encapsulates the state of the originator. This means that other classes can’t access this state directly, which helps maintain the integrity of the data.

**Composite**



**Prototype**

**A blue box with black text

Description automatically generated**

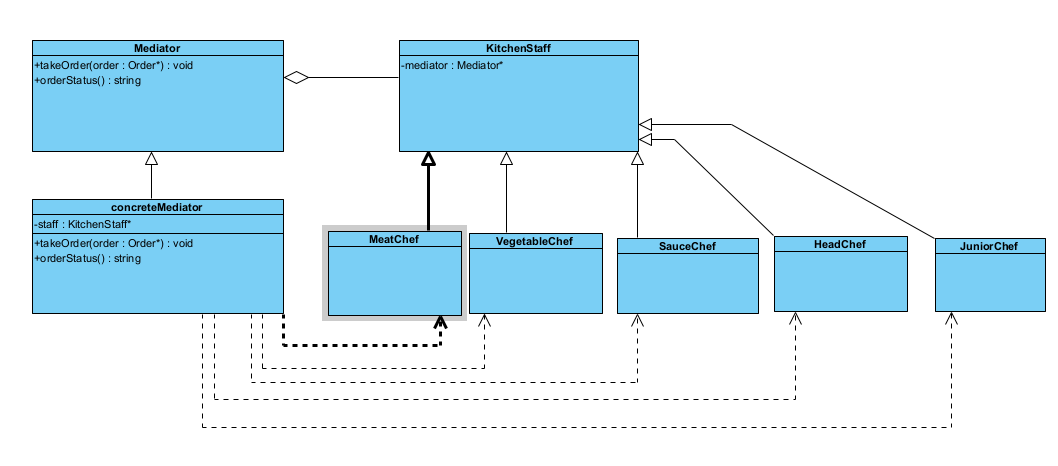
1. Create tables of customers by copying a common Customer object. All customers share functionality, only differ superficially.
2. **Avoid Subclassing**: Avoids the need for subclasses of every type of customer. Instead manage variations by cloning and configuring prototypes.
3. **Conserve System and Programmer Resources**: In a restaurant simulation with many customers, we can avoid the overhead and processing resources of initializing a new Customer object every time one is created.

A diagram of a computer

Description automatically generated**Iterator**

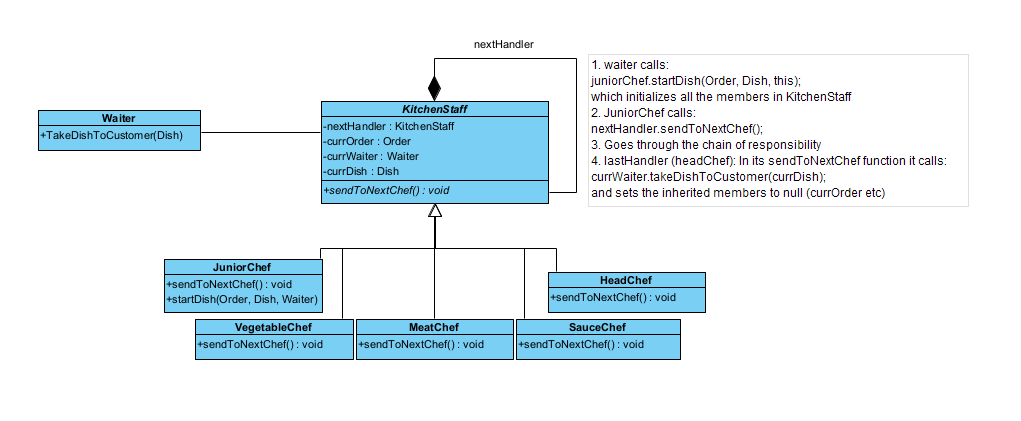
1. **Traversing Aggregates**: The Iterator pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
2. **Decouples Algorithms from Aggregates:** The Iterator pattern allows algorithms to be defined that can work with any aggregate that can provide an iterator. This means that the same algorithm can be used for different aggregates.
3. **Multiple Traversals**: The Iterator pattern allows multiple traversals to be ongoing at once. For example, one waiter might be serving drinks while another is taking orders.

**Mediator**

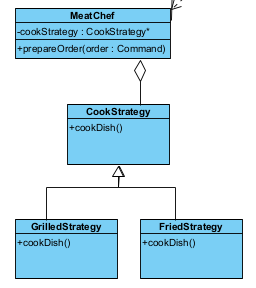


1. Decoupling: The Mediator pattern promotes loose coupling by ensuring that instead of components referring to each other explicitly, their interaction is encapsulated within a mediator object. This means that a chef doesn’t need to know about the other chefs, they just communicate with the mediator.
2. Simplified Communication: In a kitchen, communication between chefs can be complex. The Mediator pattern simplifies this by centralizing communication. Chefs just send their updates to the mediator, which handles forwarding these updates to the appropriate chefs.
3. Dynamic Relationships: The Chain of Responsibility pattern allows you to establish a chain of receivers (chefs) who can handle a request (like preparing a part of a dish). This chain can be dynamically modified at runtime, which is useful in a kitchen where the workflow might change based on the dish or other factors.
4. Workflow Management: The Chain of Responsibility pattern allows you to manage the workflow in a flexible way. You can change the order of chefs in the chain or add/remove chefs without affecting others.
5. Error Handling: If a chef can’t handle a task (maybe they’re busy or lack certain ingredients), they can pass it along the chain. This ensures that tasks don’t get dropped even if one part of the chain fails.

**Chain Of Responsibility**

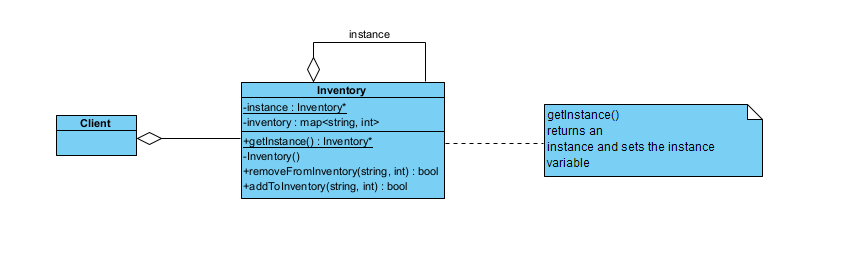


1. Decoupling: The Chain of Responsibility pattern decouples the sender (in this case, the order) and the receiver of a request (the chefs). Each chef in the hierarchy doesn’t need to know about the existence or the responsibilities of other chefs.
2. Dynamic Handling: In a kitchen, different chefs have different responsibilities based on their role and expertise. The Chain of Responsibility pattern allows you to assign these responsibilities dynamically. If a chef can’t handle a task, they can pass it along the chain.
3. Flexibility: The pattern provides flexibility in assigning tasks to chefs. If a chef is not available or busy, the task can be passed on to the next chef in the chain.
4. Workflow Management: The Chain of Responsibility pattern allows you to manage the workflow in a flexible way. You can change the order of chefs in the chain or add/remove chefs without affecting others.
5. Error Handling: If a chef can’t handle a task (maybe they’re busy or lack certain ingredients), they can pass it along the chain. This ensures that tasks don’t get dropped even if one part of the chain fails.

**Strategy**

1. **Flexibility**: The Strategy pattern allows you to switch between different algorithms (in this case, cooking strategies) at runtime. This means you can easily change the way a burger patty is cooked based on the customer’s order.
2. **Decoupling**: The Strategy pattern decouples the context (the dish being cooked) from the strategy (how it’s cooked). This means that the dish doesn’t need to know anything about how it’s cooked, it just delegates this to the strategy object.
3. **Code Organization**: Each cooking strategy can be implemented in its own class, which makes your code more organized and easier to manage. It also makes it easier to add new cooking strategies in the future.
4. **Reuse and Testing**: The Strategy pattern promotes code reuse and makes your code easier to test. Since each strategy is implemented in its own class, you can reuse these strategies in different contexts and test them independently.

**Singleton**



1. Single Instance: In a restaurant, there’s typically only one inventory that all chefs use. The Singleton pattern ensures that there’s only one instance of the inventory in your system, preventing issues like double-counting of ingredients or inconsistent states.
2. Global Access: The Singleton pattern provides a global point of access to the inventory. This means that any part of your system can check the inventory or update it as needed, which is useful in a restaurant where multiple chefs might need to access the inventory.
3. Controlled Access and Update: With the Singleton pattern, you can control how and when the inventory is accessed and updated. For example, you might synchronize access to the inventory to prevent race conditions where two chefs try to use the same ingredient at the same time.
4. Consistency: Since there’s only one instance of the inventory, you can ensure consistency across your system. All chefs see the same inventory and all updates to the inventory are immediately visible to everyone.

**Task 4.4 - Any and all assumptions, alterations, and design decisions**

**Assumptions:**

**Alterations:**

**Design decisions:**