Bill for a table of customers

The bill system in the restaurant simulator uses two design patterns: the Decorator and the Composite. The Decorator pattern allows the bill to be customized with additional features, such as a custom tip, without changing its structure. The Composite pattern allows the bill to be composed of different components, such as sub-bills and items, that can be treated uniformly. The bill system consists of several classes that implement these patterns.

The AbstractBill class is the common interface for all bills. It defines the getTotalCost() method that returns the total cost of the bill.

The Bill class is the main bill that contains the customer's information, the payment strategy, the table, and the sub-bill. It also defines the paymentMethod() and getSubBill() methods that are implemented by its subclasses.

The BillItem class is an item in the bill that has a name and a price. It implements the paymentMethod(), getTotalCost(), and getSubBill() methods from the Bill class.

The SubBill class is a sub-bill in the bill that has a list of items. It implements the paymentMethod(), getTotalCost(), addItem(), and getSubBill() methods from the Bill class.

The BillDecorator class is the abstract decorator that wraps a bill object and delegates the getTotalCost() method to it. It also defines the getDescription() method that returns the description of the decorator.

The CustomTipDecorator class is the concrete decorator that adds a custom tip to the bill. It overrides the getTotalCost(), printBill(), and getDescription() methods from the BillDecorator class.

The PaymentStrategy class is the abstract strategy that defines the paymentMethod() method for the bill. It has a pointer to a bill object.

The Card and Cash classes are the concrete strategies that implement the paymentMethod() method for the bill. They also have methods to get the payment method.

The bill system works by creating a bill object and then wrapping it with one or more decorators. The bill object can also contain sub-bills and items that can be added or removed dynamically. The bill object can also set the payment strategy for the bill. The bill object can then get the total cost and print the bill details by calling the methods from the AbstractBill interface. The bill system is flexible and scalable, as new features, components, and payment methods can be added or removed easily.

Waiter and Maitre D:

The waiter and maitre d are the classes that represent the staff in the restaurant. They are responsible for serving the customers, taking their orders, delivering their dishes, and managing their bills. They use different design patterns to achieve their tasks, such as the abstract factory, the template method, the composite, the decorator, and the observer.

The abstract factory pattern is used to create the different types of waiters, such as the general waiter and the maitre d. The abstract factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. In this case, the abstract factory pattern allows the system to create different types of waiters depending on the situation and the customer's needs.

The template method pattern is used to define the common operations for all waiters, such as visiting tables, getting orders, sending orders, delivering orders, and preparing dishes. The template method pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. In this case, the template method pattern allows the subclasses of the waiter class to implement the performTask() method differently depending on their role and the situation.

The composite pattern is used to interact with different objects and components in the restaurant, such as the table, the chef, and the floor. The composite pattern allows the waiter to compose objects into tree structures to represent part-whole hierarchies. In this case, the composite pattern allows the waiter to treat individual objects and compositions of objects uniformly. The waiter can have a reference to a table object, which can be either a regular table or a VIP table. The waiter can also have a reference to a chef object, which can be either a head chef or a sous chef. The waiter can also have a reference to a floor object, which represents the floor of the restaurant.

The decorator pattern is used to create and manage the bill for each table. The decorator pattern allows the waiter to add new functionality to an existing object without changing its structure. In this case, the decorator pattern allows the waiter to customize the bill with additional features, such as a custom tip, without affecting the bill class. The waiter uses the bill class to create the bill object and then wraps it with one or more decorators.

The observer pattern is used to communicate between the waiter and the chef. The observer pattern defines a one-to-many dependency between objects, so that when one object changes state, all of its dependents are notified and updated automatically. In this case, the observer pattern allows the chef to notify the waiter when a dish is ready, and the waiter to deliver the order to the table. The chef is the subject or the observable, and the waiter is the observer. The chef has a list of waiters who are interested in the status of the dishes. The waiter can register or unregister himself from the chef’s list, depending on his availability.

The waiter and maitre d classes use these design patterns to emulate the waiter in a restaurant. They work together to ensure that the customers are satisfied and the restaurant runs smoothly. They are flexible and extensible, as new types of waiters, tables, chefs, and floors can be added without affecting the existing code.

Tab:

The Tab class represents a customer's tab in a restaurant. The Customer class represents the customer, linked to a Tab object, and the Floor class represents the restaurant floor, storing tabs and customers.

The design pattern implemented here is the Memento pattern. In this case, the Tab class acts as the memento, the Customer class as the originator, and the Floor class as the caretaker. The Memento pattern ensures the tab's state can be saved and retrieved without exposing its implementation details.

To create a tab, a customer object is created, associated with a floor by the maitre d. Subsequently, a tab object is created with the customer's name, linked to the customer and floor, and added to the floor's tab collection. The tab object can be accessed and modified through its methods. The

Memento pattern allows the floor to save and restore the tab's state without revealing its internal structure.

This design provides a structured way to manage customer tabs and maintain their state, crucial for a restaurant's financial operations.