Group “CleanCode” Project Report

4.1 - Research

…to be done…

4.2 – Design decisions

Since the project entailed designing a restaurant simulator, a lot of the design decisions made were obviously inspired by restaurants themselves.

This can be broken down into “components” of a restaurant. This includes the:

* Customer
  + The customer was an obvious component of the design. This is who is able to order, be in different stages (states) of their dining experience, have a tab / bill where they can see what they ordered and the costs associated with them, be able to be visited by waiters, and give in their orders.
  + Many of the design decisions stem from the customer being able to have the experience they would in a restaurant.
* Order / Food
  + The reason for any restaurant is the food. This made it obvious that we needed a way for the customer to order.
  + The approach we took was a “build-a-burger” restaurant. This was done to allow full customisation of what the customer would like. From the burger bun to any assortment of toppings as seen in many real-life restaurants, which is nicely simulated with a decorator.
* Waiter
  + A waiter is an essential part of a restaurant, they are the communication between the customer and kitchen, as well as the tab. Thus, it was decided that the waiter would need to serve a role in many communicative aspects of the system.
* Kitchen / Chefs
  + Many restaurants’ kitchens have a line (chain) of chefs who prepare food. This is good because chefs can specialise in a specific aspect of food preparation and also it allows for a faster turnaround time of orders.
  + Our system closely simulates this by use of a Chain of Responsibilities where each chef prepares a component of an order (burger)
* Tab
  + No one would go to a restaurant where one’s orders are forgotten and misremembered when needing to pay. This raises the need to keep an accurate and storable history of what one ordered and being able to reverse / obtain a history of the tab in case of mistakes or human error by the waiter.
  + A memento closely resembles this and allows for exactly what was described.
* Reviews
  + Depending on whether or not the customer was happy with their experience, they are able to leave either a positive or negative review (rating or complaint) for the restaurant.
* Manager
  + The manager is who receives feedback (reports) from users which are either positive or negative.

4.3 – Patterns and solutions

Many of the described decisions’ problems are solved by the following design patterns:

* **Chain of Responsibilities** - Kitchen / Chefs
  + The chefs, each being a ConcreteHandler, are able to handle the request (order) each individually by passing the order along the line of chefs. This was done by passing the order from the waiter to the kitchen and allowing each chef to do their own job. This is done until the order is fulfilled where the waiter can then be notified and collect the order.
  + This solves the need for fulfilment of orders in the system and in the implementation allowed us to let different chefs be responsible for different processes in food preparation.
* **State** – Customers
  + The aforementioned stages of a customer’s dining experience was solved by using a State.
  + Without using a state, during implementation, the customer object would have become quite large had it need to handle all its states alone.
  + Due to the number of states of a customer, it is easier to model each of these states as an individual class rather than within the customer class. These states include WaitingToSit, WaitingToOrder, Default and AboutToLeave.
  + This also made implementation easier as it allowed the behaviour of the customer to change whilst not cluttering the internals of the customer class.
* **Iterator** – Waiter
  + The tables needed to be visited by the waiter. Since the tables form part of an aggregate an iterator was a good match for this.
  + The waiter is able to tend to each table sequentially without significant knowledge of the tables’ composition.
  + This further allows each customer at these tables to be visited and their orders to be taken, solving the issue of needing to access each customer individually through other means.
* **Decorator** – Orders
  + Due to the restaurant being “build-a-burger”, a decorator nicely solves the problem of needing to flexibly subclass orders. This allows for many combinations of orders without the need to implement each of these orders individually, but rather only components of the order which can be added dynamically.
* **Memento** – Tabs
  + A memento was used for the tab / bill. The tab has a memento which stores the state of the tab, which is then stored in a caretaker. This allows the tab to be restored to a previous point in order for mistakes to be undone if necessary.
  + A memento also nicely captures the state of the bill which allows a breakdown of the items and their associated costs.
* **Observer** – Waiter
  + We needed a way of communication between tables and the waiter. This was done by use of an observer. The waiter (observer) observes the tables and allows customers to be seated and see when and where a table is available.
  + In the implantation it allowed us to seat customers and do this at tables that are available.
* **Composite –** Tables
  + To be able to house the customers in the restaurant, varying table sizes were needed. The use of a composite allows this to be done quite elegantly.
  + Essentially tables are broken down further into smaller tables (RestaurantTables) which can be added / removed as needed. This allows an iterator or waiter to visit the tables as if they are an individual unit when in reality they comprise of smaller tables.
  + This also allows for customers to be seated in different sizes and variations.
* **Mediator** – KitchenMediator
  + Communication was needed between the kitchen, waiter, and commands.
  + The mediator facilitates this communication while still allowing the colleagues to be loosely coupled with each other.
  + This allows the restaurant floor and kitchen to communicate as well.
* **Command –** ConcreteMediator
  + The invoker of the commands is the Mediator. The use of command allowed us to queue the requests and, in the implementation, allowed us to flexibly add commands.
  + This also allowed us to have multiple commands such as creating orders and reviews / reports from the customer.
* **Template Method –** Reviews
  + Depending on the happiness of the customer, they are able to leave a review on the restaurant. Using the template method allowed us to just implement the action of leaving the review, while allowing the happiness of the customer to choose which review they would leave (negative or positive). This redefines the steps of the algorithm without changing its structure.
  + Essentially, creating a review has certain steps and those are changed without changing their structure (make header, make body)
* **Singleton –** Manager
  + Since there only needs to be one manager at a time, a Singleton was used to enforce this. This ensures that there is only one instance of a manager at a time and provides an access point to that instance.

4.4 – Assumptions

Best to do this together / write assumptions made about what we implemented.  
**--- Will be updated more as time goes on ---  
Customer:**- Customers cannot order once they are in the AboutToLeave state.  
- Customers can only leave a review/complaint when in the AboutToLeave state.  
  
**Decorator:**  
- The decorator will store a linked list of all the topping items.  
- The head pointer points to the first item in the linked list.  
- Creates and “order” object so that the real item can be created in the kitchen.  
**---------------------------------------------------------**

4.5 – UML diagrams

Already done, provide supporting diagrams for the above tasks