COS214 Practical Project

As you all know, the scenario that we have been given is to either:

A. Simulate the running of a restaurant, or

B. Gamify the restaurant and create a restaurant tycoon

(As a side note, *a tycoon (or magnate) is a wealthy, powerful person in business or industry.* Typically, a tycoon would own an entire company that consists of franchises, however, for the sake of this project, we should keep it simple and focus on the management of a single restaurant). Presented here is merely an interpretation of how we can map out which aspects of the project map to specific patterns.

Pure simulation

Singleton: *(At the time of typing this document, we haven't covered the singleton pattern but I'll provide a brief explanation). The singleton pattern allows for only one instance of a class to be created, and it is from this instance where all functions are performed. Typically, a getInstance() function is used to access the object.* Typically, a restaurant has one maitre d' hotel. As such, we can model the maitre d' hotel as a singleton. The object can then assign customers to tables which are catered for by specific waiters.

State: We can model the customer's mood as a state, of which the customer can be happy or unhappy. We can expand on this and include other states such as dissatisfied, impatient or we can rename the given states that the customer may be in.

We can use the State pattern to manage the state of customers (e.g., seated, ordering, eating, leaving) and the state of dishes (e.g., cooking, ready, served).

Strategy: Since there are different ways to prepare a meal. For example, a steak can be roasted, grilled or fried. We can assign varying modes of food preparation to various strategies. That is, each method of preparation can be a concreteStrategy.

-We could use for the types/strategies for tips eg concrete strategies may be generous tip(very satisfied), minimal tip(it was just fine)etc or no tip at all which means they have a complaint which can be handled by the next pattern(chain of responsibilities)

Chain of Responsibility: The waiter takes the order of the customer and passes that on

to the kitchen, where it is then passed on to different chefs who are in charge of different stages of meal preparation.
[Recommended by Tutor] Alternatively/additionally, the preparation process for dishes can be modeled as a Chain of Responsibility where different chefs are responsible for preparing different aspects of a dish. Each Chef would be a ConcreteHandler, while a dish would be the request being handled in the chain (Being passed down the chain until it reaches the Head Chef). An order could then be represented as a **Command** object, passed through the chain.

-We can also use this pattern for when the bill is split then depending on how much each person is contributing it can be passed on to each person. Eg if the 1st person only has R50 they can pay that and pass the bill on so that the others also pay.

-Depending on the degree of a complaint we can have different handlers handling complaints

Template Method: This comes into play when some parts of a process are the same, save for a few instructions or steps. How we can apply the template method is that certain meals can have similar preparation steps. We can model such meals by having an algorithm that matches steps which are identical while accounting for varying actions in the process.

Iterator: Should we treat the floor and/or Kitchen areas as a tilemap (Aggregate participant), the Iterator could play a role in iterating over the map. This would aid in, say, finding available tables in order for the customer(s) to navigate to them.

Observer: When a party finishes their meal, waiters can be notified of this and one of them brings the customers their bill. In this case, the waiters would be the ConcreteObservers, and each table would be a Subject. With such a scheme, different waiters can be assigned different tables to tend to via subscription.
Similarly, the same could be done when the chefs finish preparing an order- such that a waiter is notified of this and the dish(es) can be taken to the respective party.

Composite: Tables can be combined or split to accommodate various-sized groups of customers. As a result, Composite would work well in modeling this structure, treating groups of tables as one single Composite object, in which tables can be added or removed from the structure (Perhaps a Builder can be used to manage these Composites).
Alternatively, this pattern can also be used to model the bill, which can be split into ‘sub-bills’ paid by various parties at a table.

Decorator Pattern: We can use the Decorator pattern to customize dishes based on customer preferences, such as adding extra toppings or sides.

Mediator Pattern: We can use the Mediator to manage the interactions between different components, such as customers, waiters, and kitchen staff.

Builder Pattern: We can use the Builder to build parts of the game step by step. E.g. the player wishes to increase the size of their kitchen or floor add more tables to the floors and so on

Remember to separate the patterns based on which route we are going to take when implementing the project.