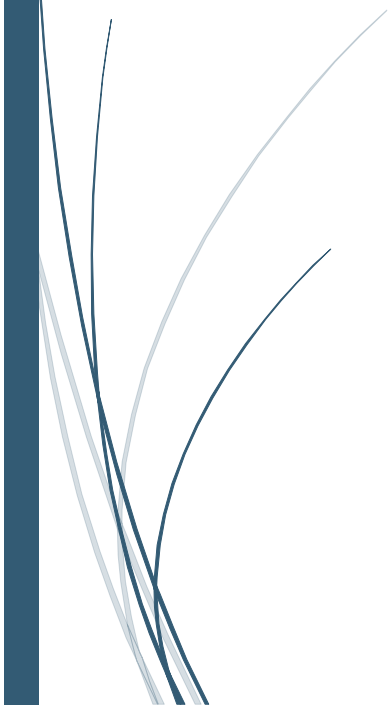
7/11/2023

# COS 214 PROJECT

Team “Concrete Pass”:

| | |
|---|---|
| Matthew Els | u21715191 |
| Dominique da Silva | u21629944 |
| Tlhalefo Dikolomela | u21507792 |
| Tinashe Austin | u21564176 |
| Christopher Katranas | u19154853 |

## Restaurant Simulator

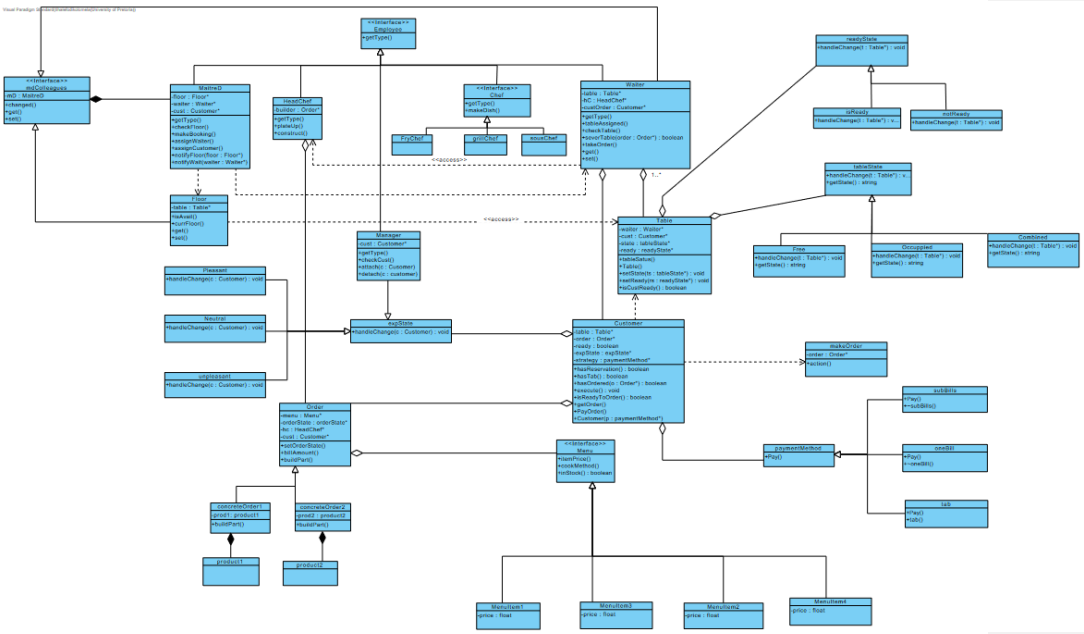
# CONTENTS

# TASK 1: PRACTICAL ASSIGNMENT

This was our initial and partial design, after a discussion:

- Mediator
  - To use for the MaitreD to allocate tables to our restaurant's customers.
- Chain of Responsibility
  - To be able to pass the order that was made from one class to another to be able get, assemble, plate and serve the meal. This would be the main focus of the system as it would dictate the flow of the program.
- Command
  - For the waiter to be able to notify and queue the orders made for the kitchen to be able to produce the meal.
- Template Method
  - Wanting to print out a customer's bill.
- Strategy
  - To be able to split the bill between multiple customers at a table.
- Façade
  - To encapsulate the system and give the client a single point of entry, even though would still be able to interact with subsystems.
- Decorator
  - To be able to add to a meal, such as additional foods or sides to the meal.
- Factory Method
  - We wanted to produce a menu and a meal using the factory method, instead of building individual meals.
- Momento
  - To store the state of the bill. The customer would be able to add to their order and once they have been completely served, the bill's state would be recalled and would be settled.
- State
  - Wanted the customer's satisfaction to ensure that various different subclasses act on the state of the customer. Such as any dissatisfaction would ensure that manager would be called.

This was our partial and initial design represented by the UML diagram that was created using Visual Paradigm.



# TASK 2: DESIGN

## 2.1 System Requirements

We are required to design and implement a restaurant simulator. The end goal of the system is the production and sale of food to customers, however there an ensemble of chaotic processes to consider.

The focus is primarily two areas namely:

- The floor.
  - The number of available tables.
  - Seating a group of customers based on whether they have made a reservation or not.
  - Having to combine tables to accommodate a large group of customers and restoring the tables to their previous state.
  - Once the group of customers has been assigned a table, the Maitre D has to assign a waiter to assist the table.
  - Should the table not be ready to place an order, the waiter should return later.
  - The table should be able to customize their order and give it to the waiter to be processed and prepared by the kitchen.
  - After the table has been served, they should be able to tip based on their satisfaction of the service.
  - The bill can either be settled (the full amount by one person, or can be distributed evenly among the customers at the particular table) or the customers can have a tab that can be settled at a later date.
- The kitchen.
  - The waiter commands the kitchen to start the preparation of an order.
  - Once the kitchen has been handed the order, the chefs can start building a meal from scratch.
  - Once the building process is done, each meal should be sent to the head chef for plating.
  - The head chef is responsible for the finishing garnishes of the order before it is sent back to the waiter to be served to the customer.

## 2.2 Activity Diagrams

**Commented [D1]:** Diagram

## 2.3 & 2.4 Identification of patterns

A combination of 10 gang of four design patterns were used to address the system requirements listed above.

1. Composite
2. Strategy
3. State
4. Observer
5. Builder
6. Chain of Responsibility

7. Decorator

8. Command

9. Template Method

10. Iterator

Commented [D2]: What is our 10th pattern???

The Composite pattern is used to model the compositions of tables to accommodate larger groups of customers.

The Strategy pattern is used to interchange the algorithm based on how the customer table would like to handle or split the bill.

The State pattern is used to be able to transition based on the internal state of the availability of the table. It is also used to model the transition based on whether the table is ready to place an order with the waiter or not.

The Observer pattern is used to model the dependency between the waiter and the customer, the waiter is notified when a table is ready to place an order.

The Builder pattern is used to model the process of building a meal of the order that was placed, from scratch and return the finished meals.

The Chain of Responsibility pattern is used to be able to pass the order request from the waiter, to the kitchen for building, the head chef and then back to the waiter to be able to serve the table.

The Decorator pattern is used to model the head chef adding additional garnished to the meal when plating the food that is going to be served to the table.

The Command pattern is used is used to model the waiter passing the order request to the kitchen.

The Template Method pattern is used

The

Commented [D3]: The last pattern and what is it being used for???

## 2.5 UML Class Diagram for system

Commented [D4]: Diagram

## 2.6 Sequence Diagrams

Commented [D5]: Diagram

## 2.7 State Diagrams

Commented [D6]: Diagram

## 2.8 Object Diagrams

Commented [D7]: Diagram

# TASK 3: IMPLEMENTATION

## 3.3 Coding Standards

We have followed C++ documentation standards. Separate header and implementation files were used (.h and .cpp files).
Comments were added using the // in our source files.

Descriptive variable have been used as well as the correct indentation.

The makefile that has been included should also be able to work should the system be expanded without having to edit this file in any way.

## 3.4 GitHub commits

Screenshot of the commits of each member:

**Commented [D8]:** Add a screenshot of all the commits made by members.

## 3.5 Doxygen comments

Our doxygen documentation can be found in our archive.

Can also be viewed at:

**Commented [D9]:** Insert the git hub doxygen link

# TASK 4: REPORT

## 4.1 Research & References

## 4.2 & 4.4 Design decisions & assumptions

We assumed that individual customers do not matter to the system. A collection of customers that are being seated at a table, will be treated as a singular object. The table will be served by the waiter and not individual customers at the table. This is due to minimize the number of unnecessary “instances” of customers, as 1 Table class represents 6 Customers.

When splitting the bill, we assumed that the value of the bill would be split evenly between all members of the particular table.

We assumed that, should a tab be the method of payment, that a tab cannot be infinite. The system set a max limit and after that the tab has to be settled.

## 4.3 & 4.5 Write up of patterns with diagrams

1. **Composite**: Composes objects into tree structures to represent part-whole hierarchies. Individual and compositions of objects are treated uniformly.

    Our table hierarchy makes use of the **Composite pattern** in order to be able to merge and decouple tables from one another for bigger groups of customers that need to be seated together.

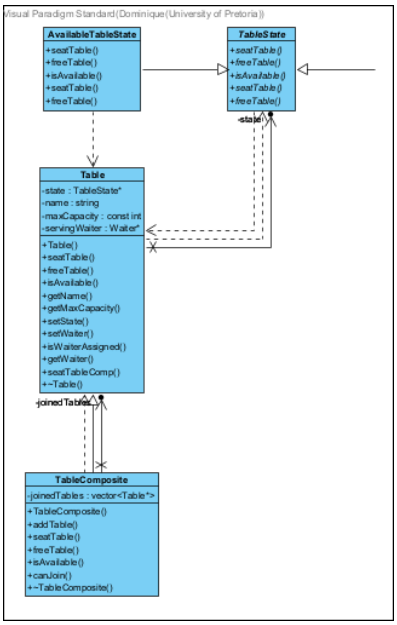
    The Component participant is the TableState. The Leaf participant is the Table, since the class consists of a singular table. TableComposite is our Composite participant as this class indicates when two or more individual tables have been joined. The TableComposite has a vector of tables that are connected, or that have been theoretically merged.

    We do this by using the methods addTable(Table*) and the freeTable() method.

**Commented [D10]:** Remember to include Harvard referencing from previous doc

**Commented [D11]:** Add onto this

**Commented [D12]:** Remember to include the diagrams




2. **Strategy**: Define a family of interchangeable algorithms, encapsulate each one, and make them interchangeable.

**Commented [D13]:** Participants?

Our Billing classes makes use of the **Strategy pattern** via the paymentMethod(). The strategy method will vary the implementation based on whether the customer is going to pay the entire bill, the bill is going to be split between the members at the table or whether a tab has been created.

Thus the Context participant is ?? which holds a pointer to an instance of the Strategy participant which is ??.

The ConcreteStrategy participants are ??

3. **State:** Allow an object to alter its behaviour when it's internal state changes. The object appears to change its class, enabling it to adapt to different situations, and effectively encapsulate the state-specific behaviour into separate state objects.

The **State pattern** is used to change the internal state of the table once the table is ready to place an order. The Table class is the Context participant, which has the reference of the current state of the table, and delegates behaviour to the State object. When a table is ready to place an order, the table switches to the readyState.

The State participant is the CustomerState class, and the derived ConcreteState participants are the readyState and the NotreadyState classes.

4. **Observer:** Define a one-to-many dependency between objects so that when one object changes its state, all its dependents are notified and updated automatically.

The **Observer pattern** is used with the State pattern in order to observe the state of the table of customers and whether they are ready to place an order with the waiter.

Observer participant is the Observer class which acts as an interface for the Waiter.

The ConcreteObserver participant is the Waiter, as the waiter observes the state of the Subject participant, which is the Table.

The Table maintains a list of waiters to notify once they are ready to order.

5. **Builder:** Separate the construction of a complex object from its representation, allowing the same construction process to create different representations.

The **Builder pattern** is used to construct our pizza menu items that have been ordered, from scratch. The Directory participant called the Director class constructs a pizza object using the Builder interface. The make(Pizza*) method is used, where a chosen menu item has been passed in as parameter that needs to be built.

Builder participant is the Builder class. ConcreteBuilder participant is the MealBuilder class. Individual components of the pizza, all have their own methods such as BuildBase() and BuildSauce().

Product being represented is the Meal class. The getResults() method that is being called in the MealBuilder class return a Meal.

6. **Chain of Responsibility:** Create a chain of objects where each object can process a request and decide whether to pass it to the next object in the chain or stop processing it.

7. **Decorator:** Attach additional responsibilities to an object dynamically, providing a flexible and reusable way to extend its functionality. This patterns allows objects to be extended with new behaviour or features without modifying the core structure.

8. **Command**: Encapsulate a request as an object, thereby allowing for parameterization of clients with requests, queueing of requests and providing support for undoable operations.

Our Kitchen makes use of the **Command pattern** by having an prepareOrder(Pizza*) method.

The Command class is the Command participant, and the OrderCommand is the ConcreteCommand. ConcreteCommand holds an instance of the type of pizza that has been requested by the waiter. The Kitchen is then called to prepare the order with the specific pizza passed in as parameter.

The Waiter is the Invoker and the Kitchen is the Receiver of the request.

9. **Template Method:** Define the skeleton of an algorithm in a method, allowing certain steps of the algorithm to be implemented by subclasses. This pattern enables the reusability of the algorithm's structure while allowing variations in the implementation of specific steps.

10.