**COS 214 - Project Report**

Cult of COS  
University of Pretoria

## **COS 214 - Project Research:**

### **Introduction:**

To accurately model our restaurant system, we used ten design patterns to represent the inner workings of a restaurant's operations. Our choice of these patterns, which include the Command, Iterator, Observer, Mediator, Strategy, Chain of Responsibility, Composite, Decorator, State, and Factory patterns, was researched and thoughtfully chosen based on their different characteristics and suitability for their roles within our simulation.

In our research, we looked into each of these design patterns, and will be discussing their unique features and how they contribute to the effective emulation of a restaurant's dynamic environment. By using these patterns, we want to create a comprehensive and functional simulation that mirrors the real-world complexities of restaurant management.

### **Command Pattern:**

The Command Pattern, as outlined by [1,263], is a powerful design pattern that facilitates the encapsulation of requests as objects. This not only enables the parameterization of clients with different requests but also allows for the queuing and logging of requests, as well as supporting undoable operations.  
The Command Pattern gives us the ability to construct a queue-based system, which aligns with the workflow of a restaurant. By representing various actions as command objects, we create a dynamic scheduling system that can efficiently manage incoming customers, table assignments, and reservations.   
The Command Pattern also allows us to cater to the wide range of characteristics of objects within our system.   
One valuable feature of the Command Pattern in our context is its support for undoable operations. For example, consider that all tables on the restaurant floor are occupied. Using the undo feature, we could return a group of waiting customers to the queue, allowing them to be idle until a table becomes available. This enhances the system's flexibility and mirrors real-world restaurant dynamics where customers may face delays.

### **Iterator Pattern:**

The Iterator Pattern plays a vital role in managing collections of objects while abstracting the underlying data structure.  
The Iterator Pattern is a behavioural design pattern that provides a standardized way to access the elements of a collection, such as a list, array, or other data structures, without exposing the underlying implementation. It separates the process of accessing elements from the specific data structure being used, which promotes flexibility and abstraction.

Abstraction of Table Access: In a restaurant, tables are an essential resource. The Iterator Pattern abstracts how tables are accessed and provides a uniform way to traverse through them. This abstraction allows us to modify the table structure without affecting the code that uses the Iterator.  
Flexibility in Traversal: The Iterator Pattern allows for flexibility in traversing the tables. Depending on the commands issued by the MaitreD, the Table Iterator class can adapt its traversal logic to match the specific requirements of the restaurant's operation.  
Separation of Concerns: The Iterator Pattern helps separate the responsibilities of accessing tables from the MaitreD's commands. This separation enhances code modularity and maintainability by isolating the logic for table iteration from the MaitreD's other tasks.  
Efficient Iteration: The Iterator Pattern can optimize the way tables are iterated, depending on the underlying data structure.

### **Observer Pattern:**

The Observer Pattern provides a clean and efficient way to handle communication and updates between components.   
Decoupling of Components: The Observer Pattern allows for a loose coupling between components, ensuring that changes in one component do not tightly bind it to others. It promotes flexibility and maintainability, as changes to one component don't ripple through the entire system.  
Event-Driven Model: The Observer Pattern is event-driven, making it well-suited for scenarios where various objects need to react to events or changes in other parts of the system. This event-driven approach simplifies the handling of more complex interactions.  
Scalability and Extensibility: The Observer Pattern accommodates scalability. If, in the future, you decide to add new features or components that require observation, the Observer Pattern allows you to extend the system without major changes.

### **Strategy and Chain of Responsibility Pattern:**

The Strategy Pattern is a behavioural design pattern that allows you to define a family of algorithms, encapsulate each one, and make them interchangeable. It enables the client to choose the appropriate algorithm to be used at runtime.

The Chain of Responsibility Pattern is a behavioural design pattern that creates a chain of objects, each of which can process a request. The request is passed along the chain until it is handled or reaches the end of the chain.

Using the Strategy and Chain of Responsibility patterns in conjunction with each other offers several advantages:  
Modularity and Extensibility:  
By combining these two patterns, you create a modular system. The Strategy Pattern allows you to encapsulate and swap out algorithms or strategies for different tasks, such as meal component preparation in our case. This ensures that your system is open to extensions without affecting the existing code. The Chain of Responsibility Pattern complements this by allowing the dynamic addition of handlers, which can be helpful if new responsibilities or chefs are introduced in a restaurant simulation.  
Flexibility and Dynamic Behaviour:  
Together, these patterns provide high flexibility. The Strategy Pattern lets you change how specific tasks are performed dynamically at runtime, adapting to different scenarios, such as varying customer orders. The Chain of Responsibility Pattern allows you to dynamically adjust the order of responsibility handlers, which can be especially useful when you need to delegate tasks based on different conditions.  
Clear Separation of Concerns: The combination of these patterns enforces a clear separation of concerns. The Strategy Pattern ensures that each chef handles a specific meal component, abstracting the details from the kitchen window. The Chain of Responsibility Pattern promotes the separation of responsibilities, ensuring that each chef is responsible for a particular component of an order. By implementing and combining the Strategy and Chain of Responsibility patterns, you create a system that not efficiently manages the expertise of chefs in preparing meal components and also allows for dynamic, adaptive, and scalable behaviour.

### **Composite and Decorator Pattern:**

The Composite and Decorator Patterns are powerful design patterns and are a good combination because:

Composite Pattern - Representing Complex Structures:  
**Hierarchy** of Menu Items: The Composite Pattern is excellent for modelling hierarchies. In a restaurant, you often have complicated menus with categories, subcategories, and individual dishes. Using the Composite Pattern, you can create a unified structure to represent all these menu items, allowing for easy navigation and organization.  
The Composite Pattern enables you to compose entire menus from smaller elements, such as dishes or submenus. This simplifies menu management, as you can treat a menu as a composite of its components, making it easier to add, remove, or change menu items.

Decorator Pattern - Extending Functionality:  
**Customizing** Dishes: Restaurants offer variations of dishes based on customer preferences. The Decorator Pattern is ideal for customizing objects dynamically. For example, decorating a basic dish with various ingredients or cooking styles

### **State and Factory:**

Incorporating the State and Factory Patterns together can be very beneficial in a system These patterns offer a complementary approach to managing the state and instantiation of objects, ensuring flexibility, maintainability, and extensibility in the software.

State Pattern - Managing Object State:  
The State Pattern is useful when an object's behaviour depends on its state, and it must transition between states while keeping its interface consistent. By using this pattern, we can represent different states of an object as distinct classes, making it easier to add or change behaviours associated with each state.

Factory Pattern - Object Creation:  
The Factory Pattern focuses on the creation of objects and encapsulates the object instantiation process. It provides a centralized and flexible way to create objects, allowing for the instantiation of different types of objects without exposing their creation logic.

State **Transition Management**: The State Pattern is excellent for managing the state and behaviour of objects, such as customers in the restaurant simulation.

Flexibility in State Configuration: In scenarios where states have variation, the Factory Pattern allows for the instantiation of state-specific objects with the desired attributes. This flexibility is beneficial when different states of an object require distinct behaviour.

**Isolation** of State and Creation Logic: The Factory Pattern separates creation logic from the rest of the code. This separation ensures that the state-specific classes evolve seperately from how they are created, making it easier to manage state transitions.

In summary, when the State and Factory Patterns are used together, they provide a structured and flexible approach to manage the state and creation of objects in a software system. This combination allows for to objects have dynamic states and to be instantiated with different configurations, promoting maintainability and extensibility.

### **References:**

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides*. Design patterns: elements of reusable object-oriented software.* Addison-Wesley, Reading, Mass, 1995.

[2] Yang, J. (2020) *Order a burger and manage a job queue using the Command Pattern*, *Ju Yang*. Available at: http://www.juyang.co/order-a-burger-and-manage-a-job-queue-using-the-command-pattern/ (Accessed: 05 November 2023).

[3] Nero, R. del (2022) *Intro to the observable design pattern*, *InfoWorld*. Available at: https://www.infoworld.com/article/3682139/intro-to-the-observable-design-pattern.html (Accessed: 05 November 2023).

[4] Korchynskyy, O. (2023) *Design patterns: Composite, decorator, facade*, *Medium*. Available at: https://medium.com/@alekpublic4/design-patterns-composite-decorator-facade-cd235c7907b0 (Accessed: 05 November 2023).

[5] Prabu, G. (2017) *Chain of responsibility and strategy pattern using task C#*, *CodeProject*. Available at: https://www.codeproject.com/Articles/1182984/Chain-of-Responsibility-and-Strategy-pattern-using (Accessed: 05 November 2023).

Assumptions and choices:

We decided that

Practical use of patterns and the problems they solve:

The singleton pattern is used for the restaurant class. There only needs to be a single restaurant instance that holds the rest of the implementation, solving the issue of a larger container for the rest of the implementation to take place within, and a single starting point for the simulation, ensuring that only one restaurant is ever created during the lifetime of the project and that in situations where a restaurant would be ‘closed’ and ‘opened’ it only creates the restaurant if it does not already exist. The singleton pattern also represents the larger whole that all the components in a restaurant form.

We make use of the command pattern for the MaitreD class, the MaitreD class being the class that controls the opening and closing of the restaurant, the introduction of customers into the restaurant and the seating at their tables, as well as the round counter, which alerts all the relevant classes that a round has passed, so those classes can react to rounds passing. This class also controls the iteration through tables to find a suitable location for customers to be seated. This pattern solved the problem of communication of rounds to the other classes as well as the creation of customers and tables and the restaurant itself. The opening and closing of the restaurant is also controlled by MaitreD.

The iterator pattern is implemented in the TableIterator class and used by MaitreD to find a table to seat a customer at, by iterating through tables until one that is available is found. This solved the problem of searching through tables to find one that a customer could be seated at.

The state pattern is implemented in the customer class to define actions to take depending on customers readiness and happiness. Each customer has a readyState (either readyToOrder or notReadyToOrder) which controls whether the customer’s order can be taken by the waiter of the table the customer is at (only once every customer at a table is ready to order can the order be placed). Each customer has a random number of rounds that they can take until they’re ready, and their state is set to readyToOrder. The customer also has a number of rounds they are happy for, which is decremented each round, and if it reaches 0, the customer happy state is set to unhappy, and if the majority of the customers at a table become unhappy, they leave the restaurant. The use of this pattern allowed us to customize the behaviour of the customer and the system based on the state of the customer and solve the problem of customer-state-dependant system behaviour.

The factory method is also implemented with Customer, to allow for a variety of types of customers with slightly differing properties to enter the restaurant, to simulate various types of customers you might find in a real-life restaurant setting. The types of customers, Normal, Wealthy, and Karen have differing default happiness properties, resulting in some getting unhappy faster than others, and creating the possibility for varying happiness states among customers.

We make use of the observer pattern with the waiter class, which can observe both the kitchenwindow and the table it is serving. The table notifies the waiter associated with it that the order can be placed, at which point the waiter places the order with the kitchenwindow and is attached to the kitchenwindow until the order has been made, when the waiter is notified that the order is completed and lets the table known that it has been served. This solved the problem of communication between waiters and tables and waiters and the kitchen and simulated the real-life interactions between tables and waiters and the kitchen and waiters.

We make use of the chain of responsibility pattern within the chef classes, wherein the meal components of menu items of an order are handled by different chefs depending on type of meal, and passed through each until the right chef is used. This allowed for the simulation of a real kitchen environment in which responsibility for different components falls on different chefs, and allowed for the simulation of the cooking process. (For example, the fry cook prepares food that should be fried etc.)

The decorator pattern is made use of with the MenuItem and Plating class, and allows for the properties of the MenuItem to be altered, in this case cost, depending on the type of plating chosen, which will affect the overall cost of the MenuItem. This allowed for the simulation of customer choices with regards meal presentation. The composite pattern is also made use of to represent a menuitem as the combination of many component parts.

We implemented the Strategy design pattern through the Chef class and its children. Each child of chef implements the cook function ‘cook’ differently, this is used for output. We did this to solve the problem of creating a unique way to output what each different chef is currently busy making. The startCooking function in KitchenWindow is used to start the process.