COS 214 - Project Research:

Cult of COS
University of Pretoria

Introduction:

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

In the following sections, we will delve into each of these design patterns, discussing their unique features and how they contribute to the effective emulation of a restaurant's dynamic and multifaceted environment. By leveraging these patterns, we aim to create a comprehensive and functional simulation that mirrors the real-world complexities of restaurant management and operations.

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.

First and foremost, the Command Pattern provides us with the ability to construct a queue-based system, which aligns seamlessly with the workflow of a restaurant. By representing various actions as command objects, we create a dynamic scheduling system that can efficiently manage the influx of customers, table assignments, and reservations.

Furthermore, the Command Pattern allows us to cater to the diverse characteristics of objects within our system. For instance, certain customers may possess a priority feature, which implies that they should be prioritized over others when it comes to table allocation. By leveraging this pattern, we can define different concrete commands tailored to these specific scenarios, ensuring that the system behaves in a way that reflects the nuanced nature of restaurant operations.

One of the most valuable features of the Command Pattern in our context is its support for undoable operations. In the restaurant simulation, this functionality proves invaluable. For example, consider a scenario where all tables on the restaurant floor are occupied. Using the undo feature, we can efficiently return a group of waiting customers to the queue, allowing them to be in an idle state until a table becomes available. This not only enhances the system's flexibility but also mirrors real-world restaurant dynamics where customers sometimes face delays and are accommodated accordingly.

Iterator Pattern:

The Iterator Pattern is a fundamental design pattern that 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 details. 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 setting, 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 or data source 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 duties.
Efficient Iteration: The Iterator Pattern can optimize the way tables are iterated, depending on the underlying data structure. In our system the iterator pattern allows one to create priority queueing based on some sort of underlying characteristic that an object may hold.

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. Promotes flexibility and maintainability, as changes to one component don't ripple through the entire system.
Event-Driven Model: The Observer Pattern is inherently event-driven, making it well-suited for scenarios where various entities need to react to events or changes in other parts of the system. This event-driven approach simplifies the handling of complex, asynchronous 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 modifications.

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 in a software system offers several advantages:
Modularity and Extensibility:
By combining these two patterns, you create a highly modular system. The Strategy Pattern allows you to encapsulate and swap out algorithms or strategies for different tasks, such as meal component preparation in your case. This modularity 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 or rearrangement of handlers, which can be advantageous as new responsibilities or chefs are introduced in a restaurant simulation.
Flexibility and Dynamic Behaviour:
Together, these patterns provide a high degree of 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 prioritize tasks or delegate responsibilities based on changing 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 efficiently, 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 aspect of the order. This division of labour makes the codebase more maintainable and comprehensible. By integrating

the Strategy and Chain of Responsibility patterns, you create a software system that not only efficiently manages the diverse expertise of chefs in preparing meal components but also allows for dynamic, adaptive, and scalable behaviour. This combination empowers your system to accurately simulate the dynamic environment of a restaurant kitchen while maintaining clean, modular, and extensible code.

Composite and Decorator Pattern:

The Composite and Decorator Patterns are powerful design patterns that, when used in conjunction, can provide significant benefits to your restaurant simulation system. Here are reasons why they are a good combination:

Composite Pattern - Representing Complex Structures:
Hierarchy of Menu Items: The Composite Pattern is excellent for modelling hierarchical structures. In a restaurant system, you often have complex 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 modify menu items.

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

State and Factory:

Incorporating the State and Factory Patterns together, even outside the context of a restaurant, can be highly beneficial in various software systems. These patterns offer a complementary approach to managing the state and instantiation of objects, ensuring flexibility, maintainability, and extensibility in the software. Here's why combining these two patterns is advantageous:

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 employing this pattern, you 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 or configurations 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 your restaurant simulation. However, objects often need to transition between states dynamically. The Factory Pattern can be used to create state-specific objects, ensuring that the correct state objects are instantiated when a transition occurs.

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

Isolation of State and Creation Logic: The Factory Pattern isolates the creation logic from the rest of the code. This separation ensures that the state-specific classes can evolve independently of how they are created, making it easier to manage complex 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 is not limited to a restaurant simulation but can be applied in various contexts where objects have dynamic states and need 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).