COS214 - Project

An Academic Report on the Implementation of a Restaurant Simulation System

No Girls Allowed (Except Monica)

James Cooks - u21654680, Lloyd Creighton - u04954336, Quintin d'Hotman de Villiers - u21513768, Yi-Rou Monica Hung - u22561154, Qwinton Knocklein - u21669849, Chanseop Carter Shin - u21470864, Ze-Lin Daniel Zhang - u22614495

University of Pretoria

Department of Computer Science

Dr. Linda Marshall

Due Date: 07/11/2023

Date of Submission: 06/11/2023

Table of content

Introduction

In the realm of software engineering and system design, the application of design patterns is paramount in achieving solutions that are not only efficient but also maintainable and extensible. This academic report delves into the implementation of a restaurant simulation system, a complex software endeavour that mirrors the intricate workings of a real-world restaurant. Seven individuals collaboratively undertook this project, employing a range of Gang of Four design patterns to optimise the system's structure and functionality. The report provides a comprehensive analysis of the system's architecture, the design patterns employed, and the methods and technologies utilised, offering valuable insights into the practical application of software engineering principles in a group project setting.

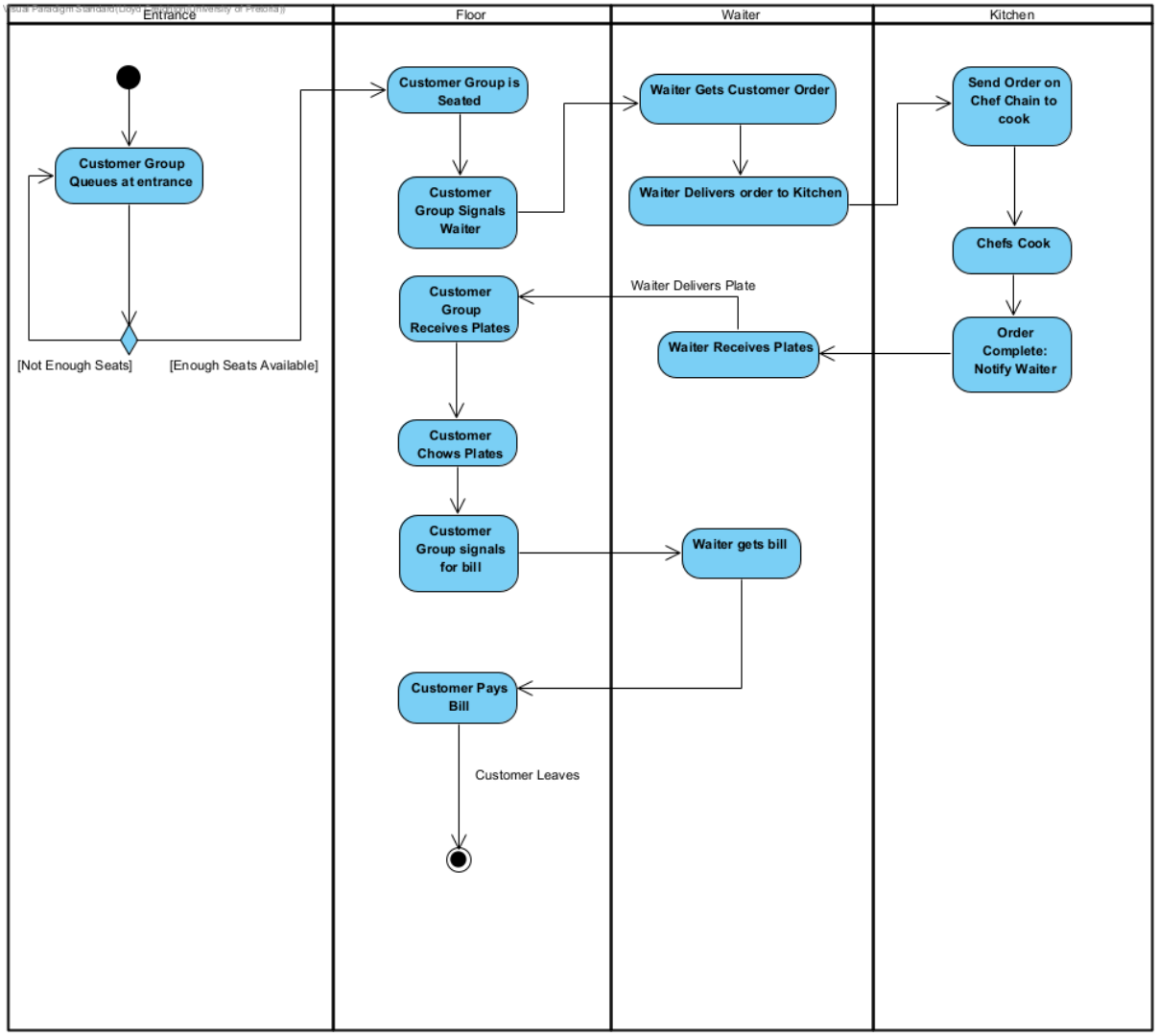
Problem statement

A system that represents a restaurant simulation is to be implemented using at least 10 of the 23 Gangs of Four design patterns. The system is supposed to implement the inner workings of a restaurant as in the real world, having a floor with tables and waiters, a kitchen with chefs and an ordering and billing system. This report describes how we, as a group, have implemented this problem using the necessary constraints.

Implementation Overview

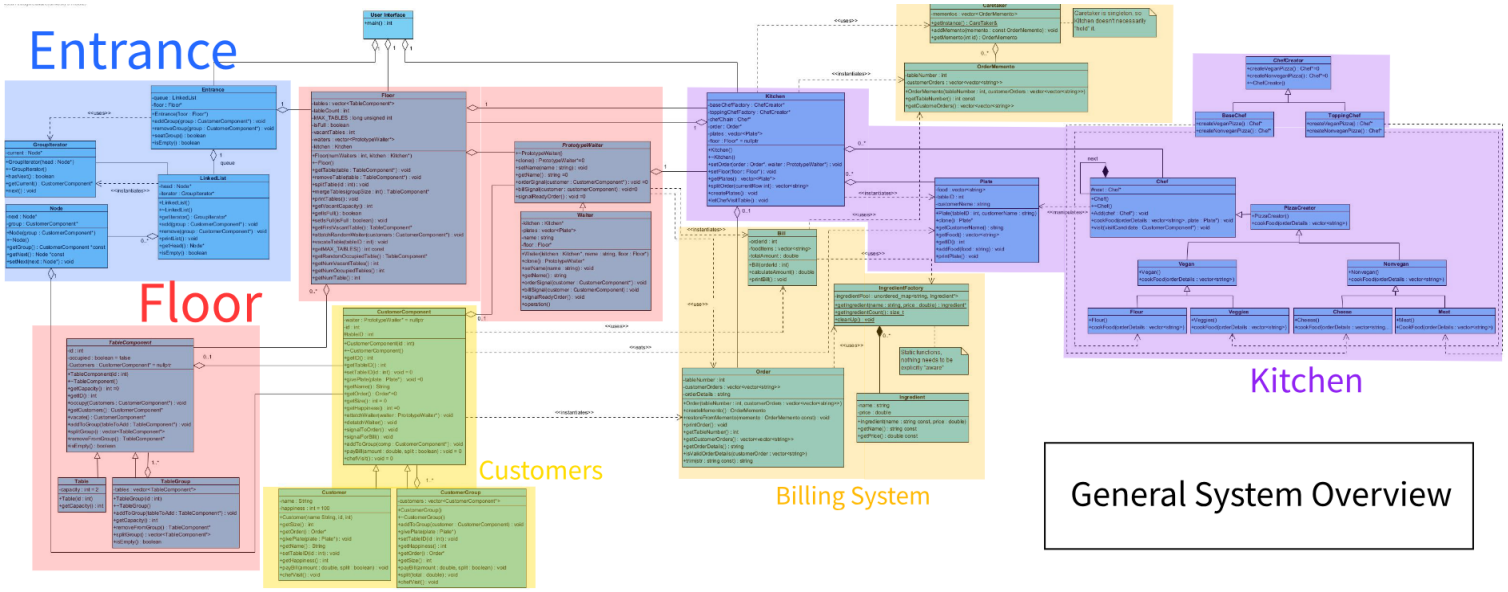
General System Overview

A high level overview of the system as a whole



The activity diagram details how the system reacts to a group of customers and their relevant actions within the restaurant. The customers get seated at a table (or, if needed, a group of tables) by the entrance, after which they will be assigned a waiter. The customers, through the waiter, will order food. The waiter will deliver the order to the kitchen, and the kitchen will start the chefs on the cooking process. Once the order is complete, the kitchen responds to the waiter with the plates for the customers. The waiter delivers the food to the customers, which they will then eat. After eating, the customers will signal for the bill, get the bill, pay the bill (by splitting or by paying all at once) and promptly be escorted from the premises.

A high level overview of the subsystems

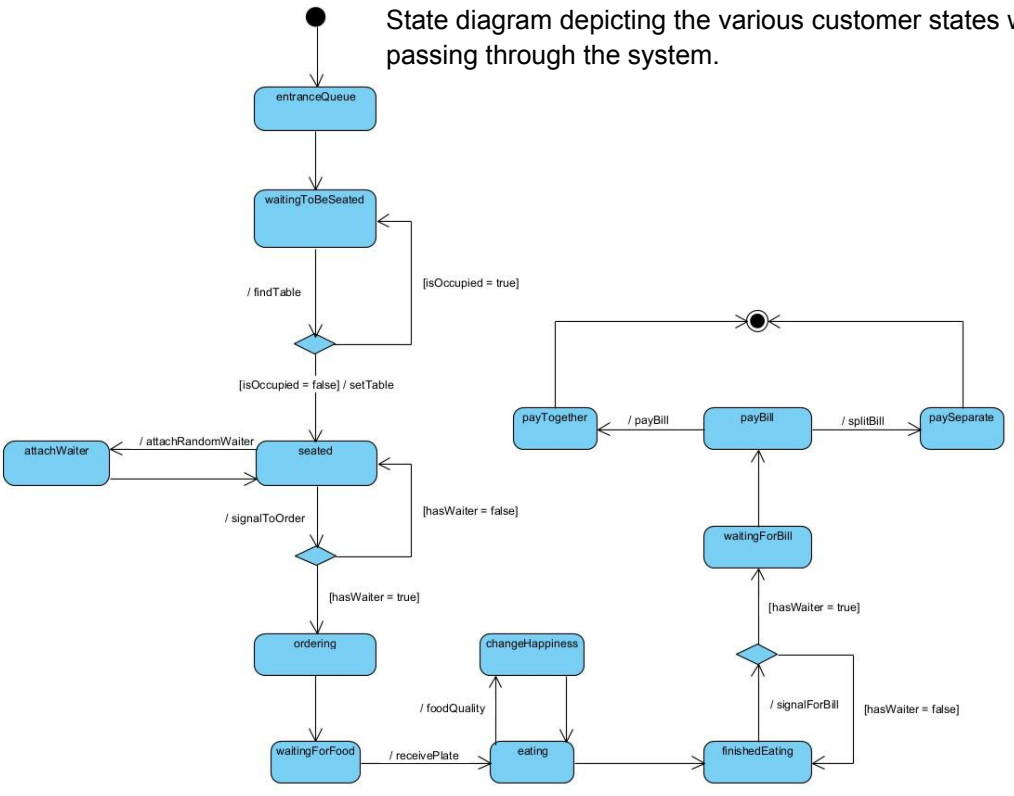


General System Overview

Customers

Customers are in essence the most important part of the system as they are the ones who get all the parts of the restaurant moving. They have operations like eating, ordering and paying. Because of this they have to be seen as a separate entity as they are involved in all parts of the system directly or indirectly.

State diagram depicting the various customer states while passing through the system.



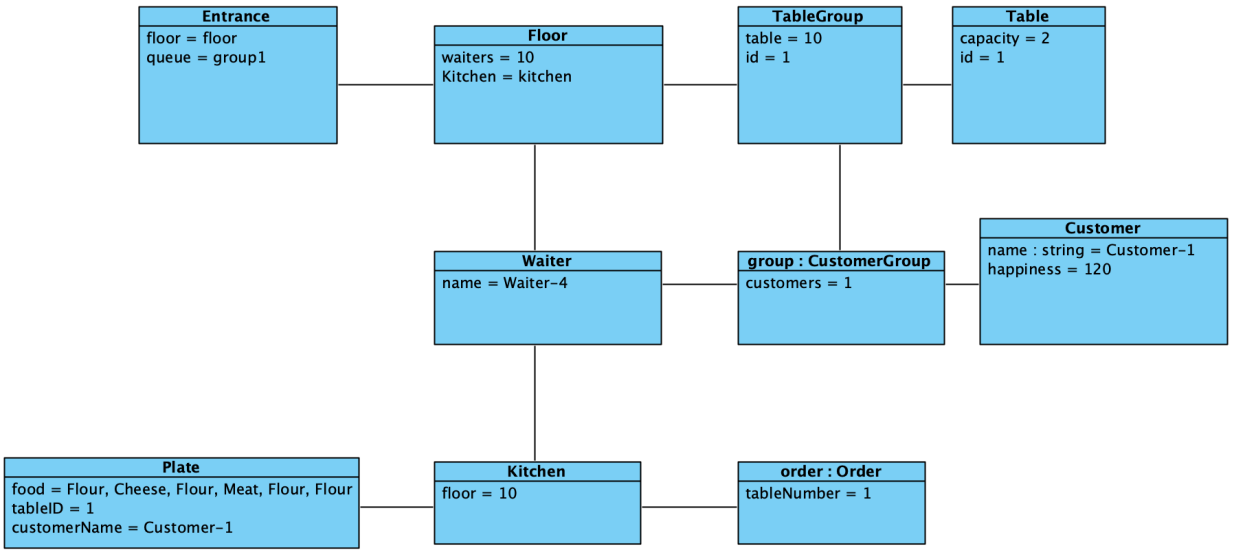
Entrance

The entrance serves as our point of entry into the restaurant that is responsible for the creation of customers from the restaurant's point of view. This allows customers to "arrive" at the restaurant and start forming a queue. The entrance is also responsible for seating customers based on the state of the floor.

Floor

The floor holds all the tables and the waiters in the restaurant. It manages how tables are grouped and split to accommodate the customers' group sizes and manages which tables are vacant or occupied. It also ensures that waiters are attached to customer groups when they are seated. When a customer group arrives, vacant tables are merged to accommodate the group's size. When a customer group leaves, the floor splits these tables back into their atomic forms.

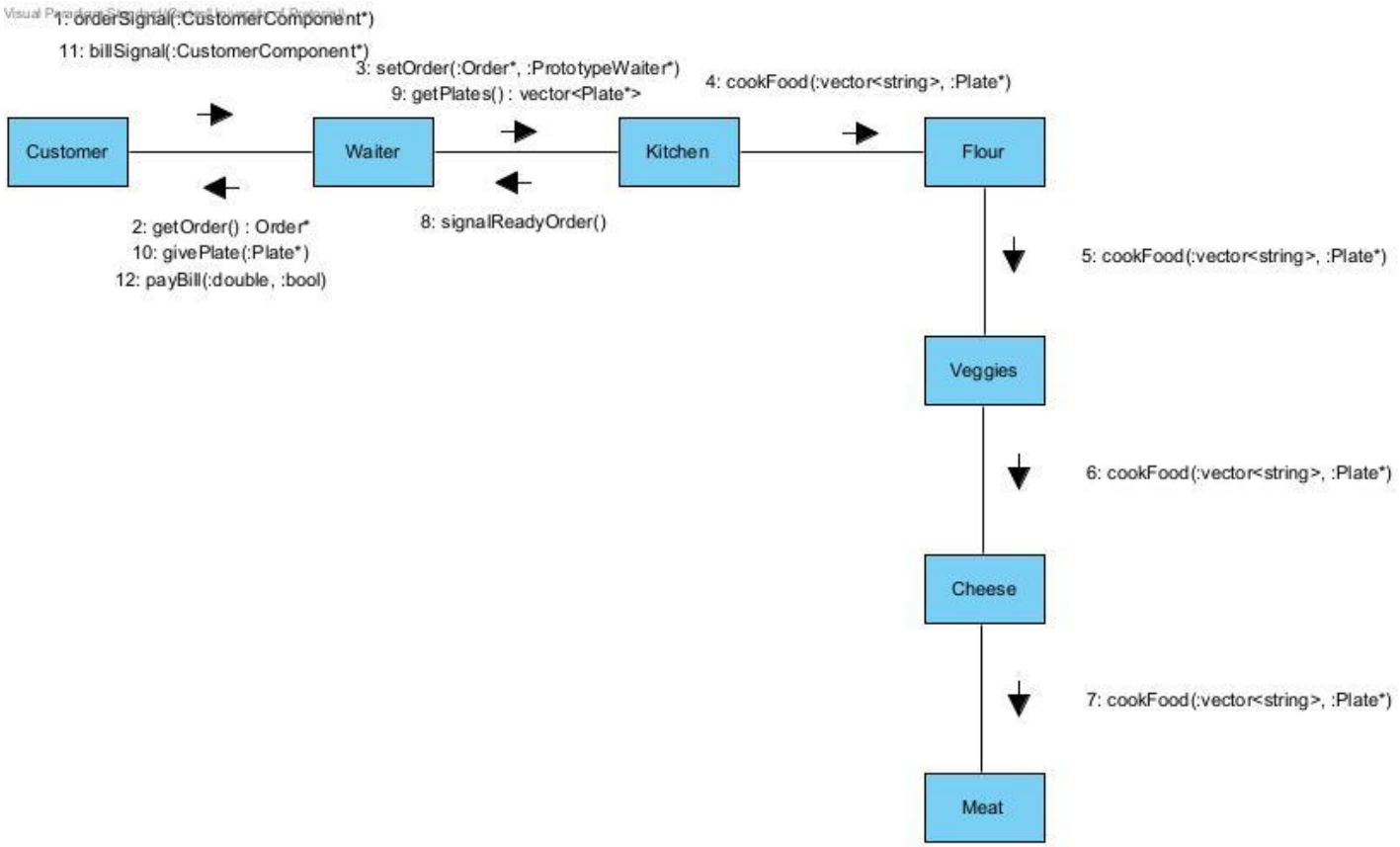
Object diagram showing a snapshot of objects in the system



Kitchen

The kitchen is where all the magic happens. Here orders are passed in from a waiter and then through a chain of chefs and workstations orders are turned into cooked food that the waiter can then return to the customers.

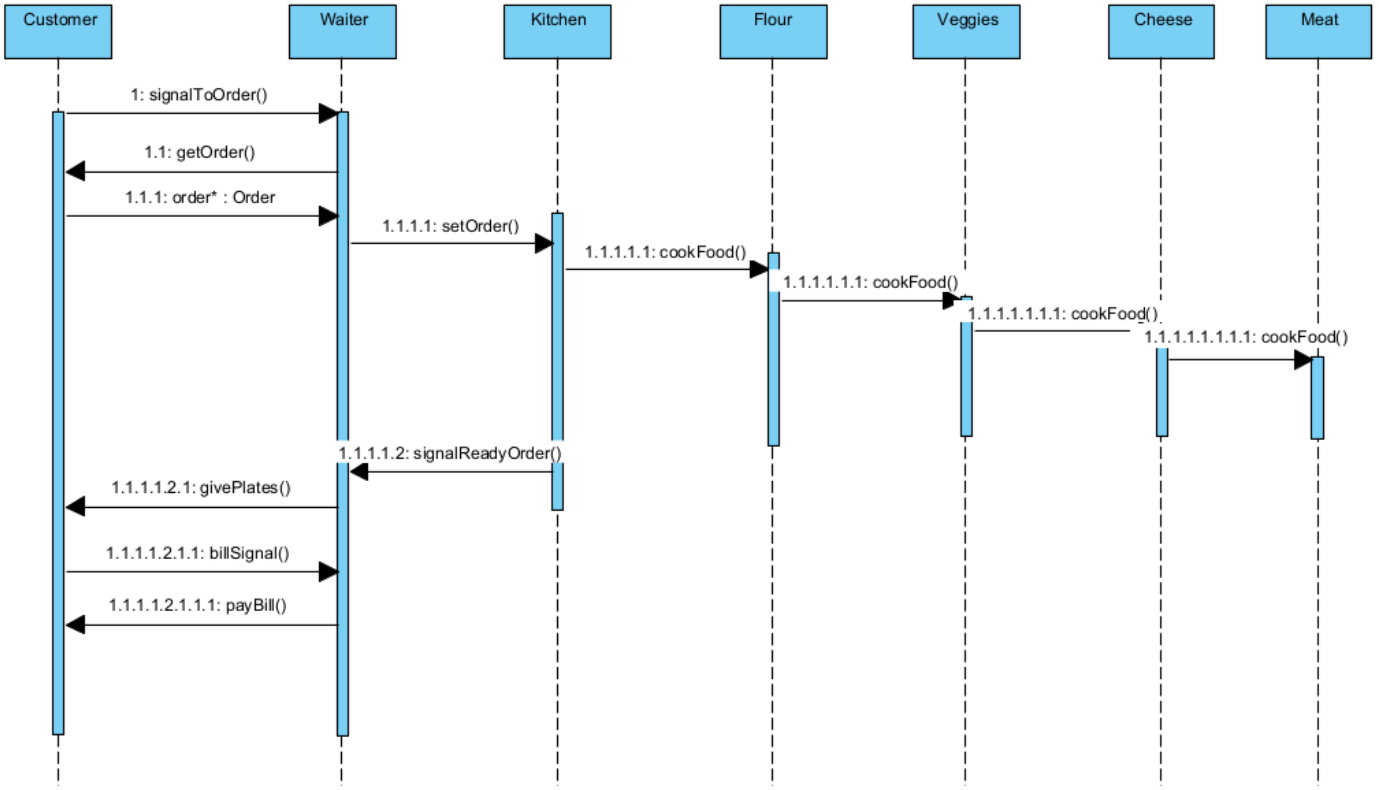
Communication diagram showing how a customer communicates through the waiter with the kitchen.



Billing system

The billing system comprises the order, ingredient, bill and order memento systems. An order is constructed using ingredients from the ingredient factory (flyweight) by the customers. Once an order is sent to the kitchen, a memento of this order is made, and after it has been processed, the order is deleted. When the customers request the bill, the bill is constructed using the memento of the order associated with the customers. The bill uses the aforementioned ingredients from the ingredient factory to get the prices of said ingredients and contribute to the total of the bill.

Sequence diagram showing a cut-out part of the system where a customer signals the waiter to order, and signals for the bill.

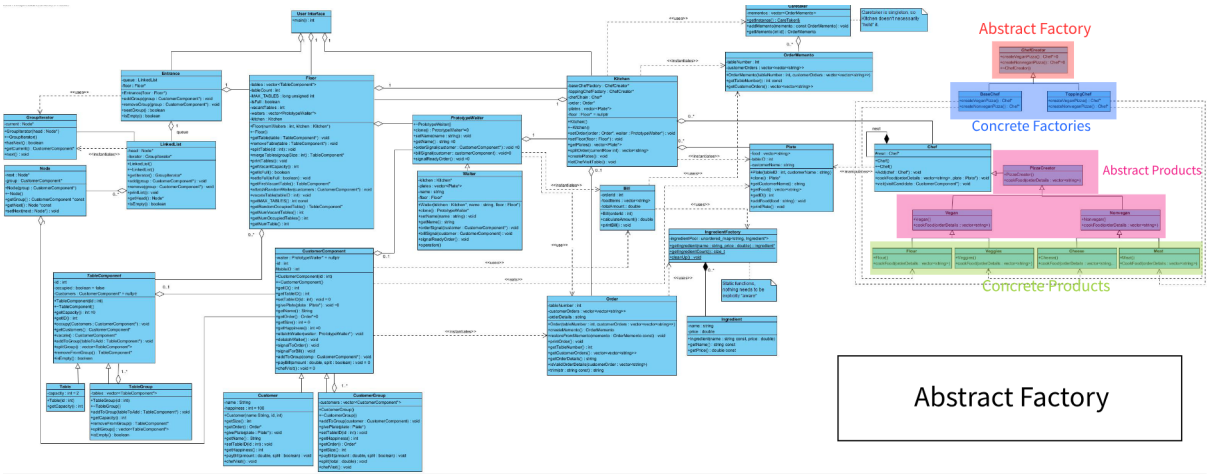


Assumptions Made

- Although we do have the code to iterate over the whole entrance and select specific customer groups based on floor availability. Upon testing however, we found that this made it impossible to serve large groups without implementing unnecessary overhead of how long customers have been waiting.
- When working with groups we only wanted to allow groups to order once and then leave the system after paying as otherwise simulations would go on for too long and become redundant and hard to read.
- CUSTOMERS LOVE EATING PLATES. This means we don’t deal with passing dirty plates back to the kitchen. We thought it was much funnier making customers deallocate their plates.

Design Patterns Used

Abstract Factory

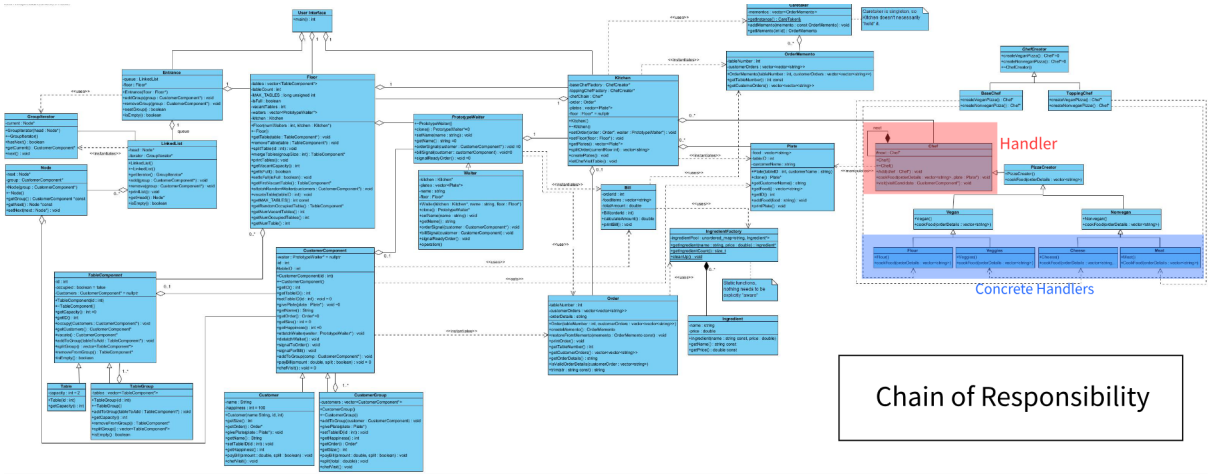


Intent:
Provide an interface for creating families of related or dependent objects without specifying the concrete classes. (Marshall and Pieterse)

How the pattern is implemented:

The abstract factory allows you to create a family of objects without specifying their concrete class. It creates various types of chefs without specifying their exact types. It also ensures that all the created objects will result in a consistent interface, this means that you can create new chefs without impacting the client code. It also allows for flexibility, if you need to add new types of chefs or modify existing chefs, you can do so respectively.
For the pizza creation, the chefs prepare for the meal and the abstract factory accommodates the creation of interrelated objects and ensures it works seamlessly.

Chain of Responsibility



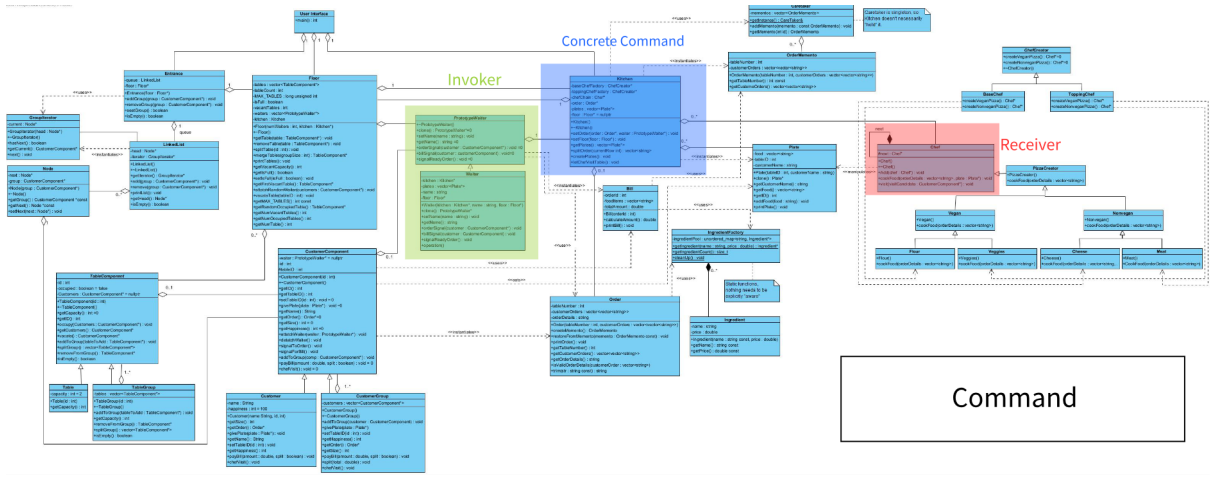
Intent:

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. (Marshall and Pieterse)

How the pattern is implemented:

The Chain of Responsibility was very useful when it came to cooking the food. Combined with the Abstract Factory to make a very versatile kitchen in which we can produce many chefs of different types and then when an order is received by the kitchen it is sent along the chain of chefs to allow each one of them a chance to handle their respective responsibility when it comes to cooking the food. By doing this we also reduced the coupling between the sender and the receiver of a request which is inline with the intent of the Chain of Responsibility.

Command



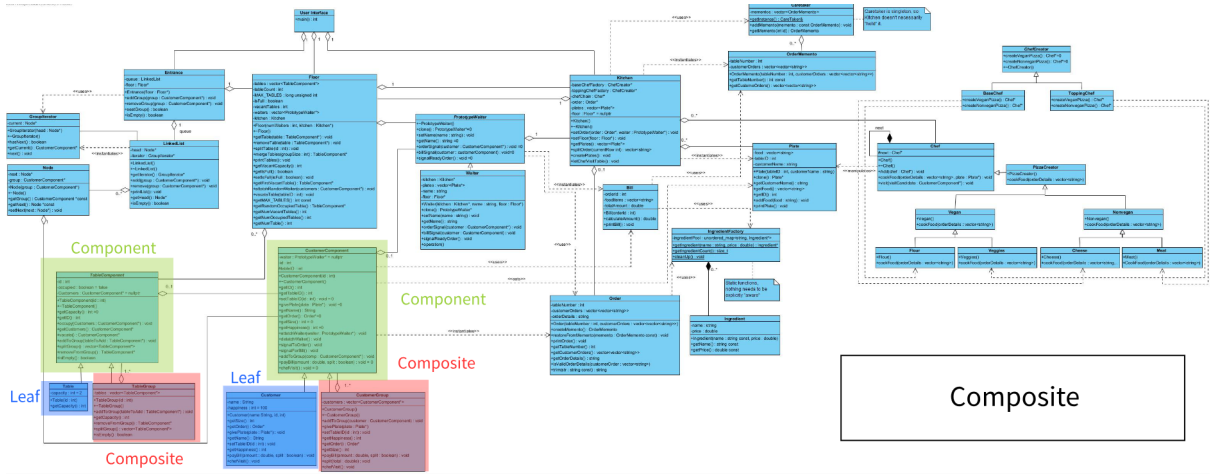
Intent:

Encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations. (Marshall and Pieterse)

How the pattern is implemented:

The command pattern is used to allow the waiter to communicate with the chefs (through the kitchen) and send through commands for the creation of customers' orders. This is very useful to decouple the waiter from the chefs as the waiter need not know of the chefs, nor how the chefs operate.The kitchen is a useful standardised interface that the waiter can use. This means that the waiter need not be aware of any changes in the kitchen, it is only aware of the fact that it sends an order and receives plates of food.

Composite



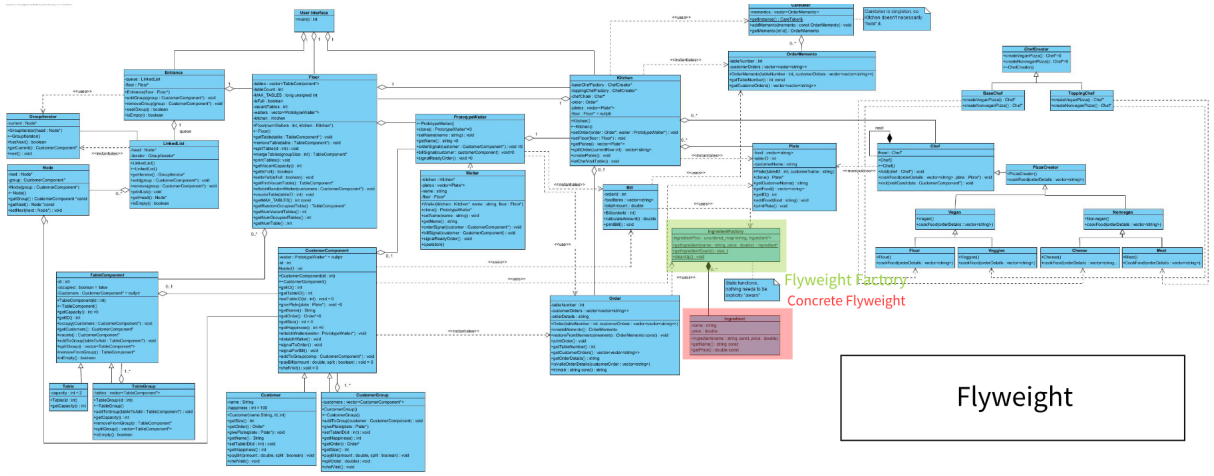
Intent:

The composite pattern creates tree structures of objects to allow for hierarchical aggregations that will be treated uniformly. (Marshall and Pieterse)

How the pattern is implemented:

Composite is used to represent our customers and tables such that we can separate the behaviours of groups of customers and groups of tables. This makes it easy to delegate to members within groups, and to do calculations (such as the size of a customer group and the capacity of a table group). Separating methods by group behaviours and atomic behaviours is also useful for methods such as separating the bill or paying all at once in a customer group.
Thus we can have our Customers and Tables be treated the same by any other class that uses them, without having to know the difference in the behaviours of groups and leaves.

Flyweight



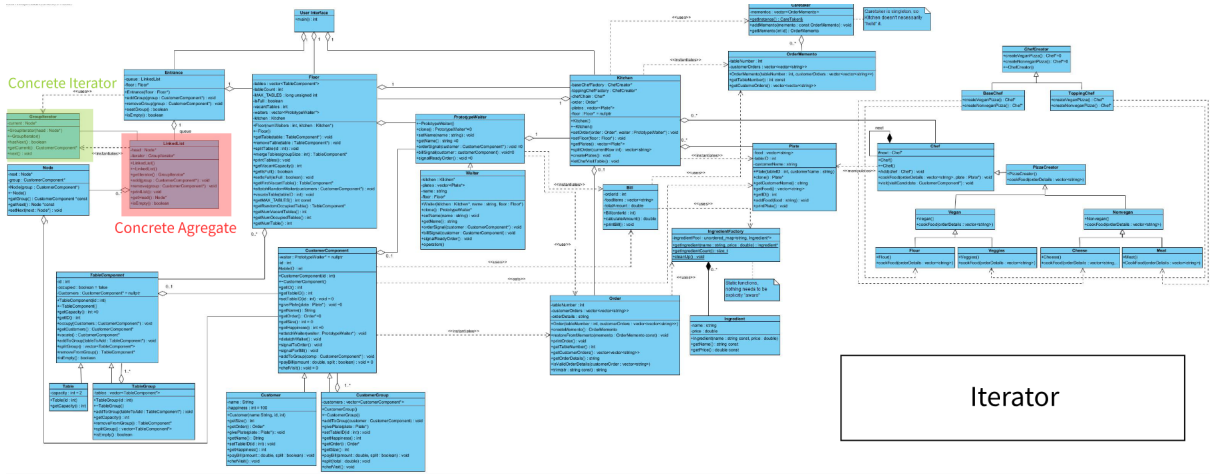
Intent:

User sharing to support large numbers of fine-grained objects efficiently. (Marshall and Pieterse)

How the pattern is implemented:

Order uses flyweight objects for billing to calculate how much each order will be per customer as well as the total of the bill for the table. Flyweight being the ingredients, which each flyweight object saves the name and the price of each ingredient. The reason that flyweight is used is so that each time the bill needs to be calculated, it is not necessary to create an entirely new ingredient object, wasting memory, and rather simply getting an already existing ingredient to get its price.

Iterator



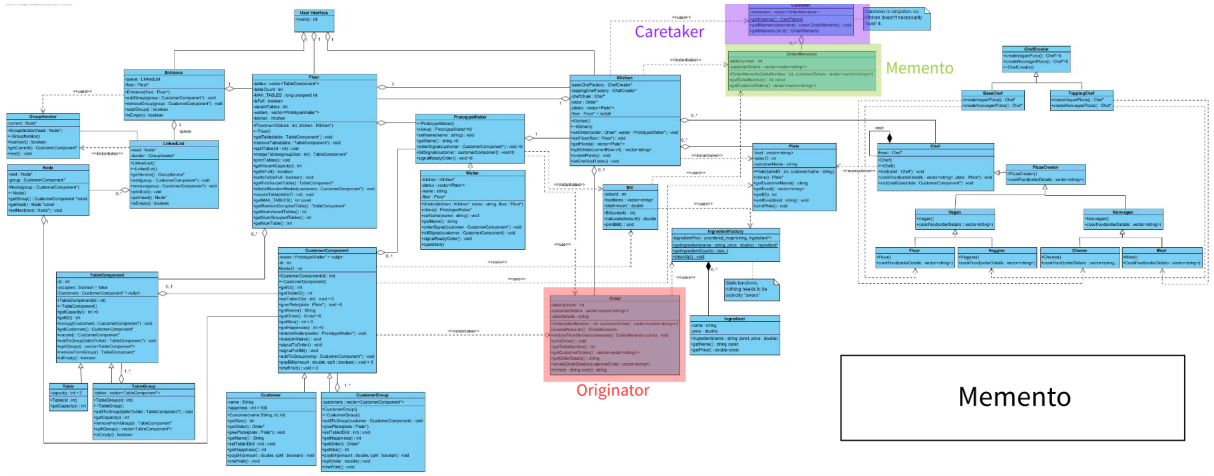
Intent:

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. (Marshall and Pieterse)

How the pattern is implemented:

The entrance is where customers who are added to the restaurant wait in a linked list. By using an iterator it separates the process of iteration through the linked list of customers from the underlying data structure which makes for a more flexible and efficient system. It also ensures that customers are seated in the correct order which simplifies needing to check this when seating the customers.

Memento



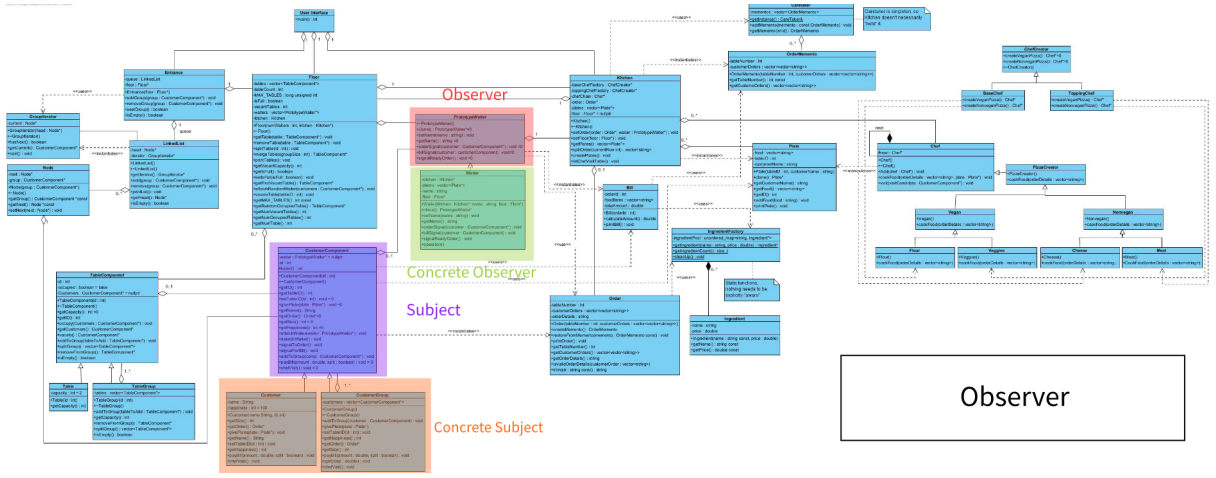
Intent:

Without violating encapsulation, capture and externalise an object's internal state so that the object can be restored to this state later. (Marshall and Pieterse)

How the pattern is implemented:

The use of memento was chosen due to being able to keep a 'record' of the orders per table, making it effective to keep track of what was ordered by each group as well as being able to calculate the bill per order. This also allows that we do not unnecessarily keep order objects around in memory, and rather keep mementos to them.

Observer



Intent:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. (Marshall and Pieterse)

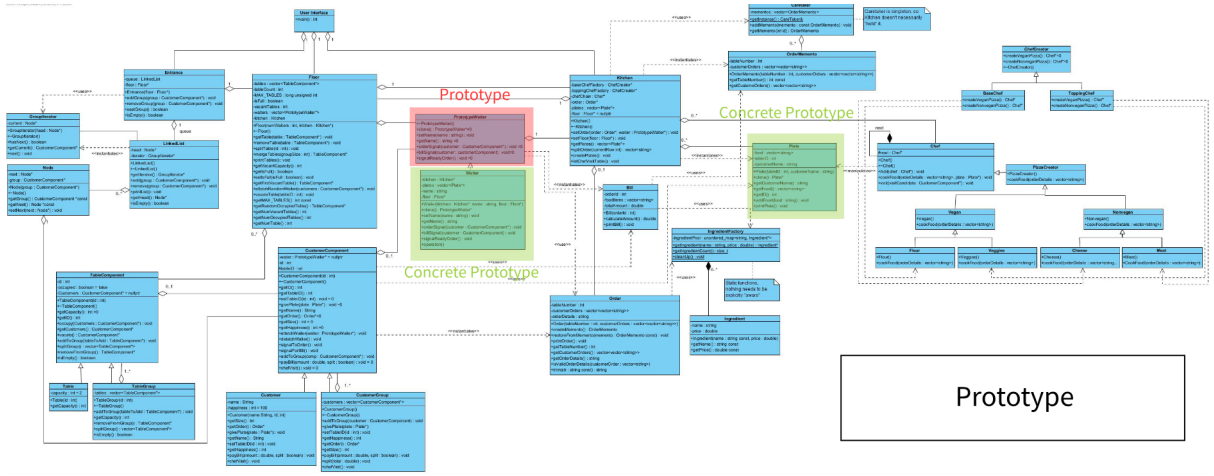
How the pattern is implemented:

The observer pattern allows for objects to attach to and detach from the objects that are to be observed. We use this in the waiter class to observe the customer objects such that customers can notify their respective waiters when they would like to order and pay the bill. This also allows us to have further expansion of waiters to add more waiter types (such as a more experienced waiter) should it be necessary.

Note on Mediator:

The waiter class also applies the mediator pattern, however not fully and to a lesser extent than the observer pattern.
Reasoning:
The customer doesn't need to interact with the kitchen directly and vice versa (in relation to orders and food). This helps with maintaining the system and makes it easier to extend the system as well. The waiter also helps control the order of operations so that there is control over the workflow. For example the waiter ensures that an order is sent to the kitchen and is responsible for delivery of the food afterwards. This would be difficult to handle if the customer and kitchen were communicating directly. Lastly, if in the future we decide to add more classes such as a manager or a Cashier they only need to interact with the waiter and not with every other class, since the waiter is a mediator it makes it easy to manage the interactions. Thus having the waiter as a mediator allows for flexibility.

Prototype



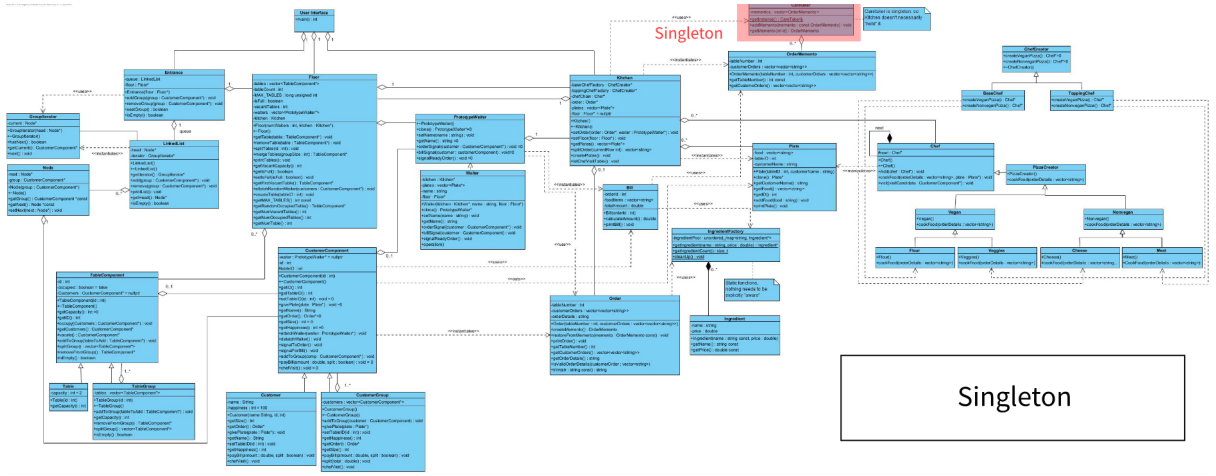
Intent:

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. (Marshall and Pieterse)

How the pattern is implemented:

We used the Prototype pattern in 2 distinct locations namely the plate and the waiters.
In the Plate class we use prototype because plates share common attributes like the table id thus it is easier to copy these plates than having to instantiate a new plate for each customer then we can simply set the things that differ for each plate like which customer the plate is assigned to.
Waiter's reasoning is quite similar in the sense that all waiters are assigned to the same floor and then we can simply set the name of each waiter.

Singleton



Intent:
Ensure a class only has one instance, and provide a global point of access to it. (Marshall and Pieterse)

How the pattern is implemented:

A singleton is used when implementing the caretaker for the memento class to ensure that there is one instance of the interface used in getting the mementos, this allows a shared interface to be used by multiple classes in the getting and setting of all mementos in the program.

Honourable mentions:

This section is patterns we believe are in our system in one way or another. Although not to the extent that we can justify including it in our official count of patterns.

Façade:

We have subsystems running like the entrance, floor and entrance, and then kitchen. The user interface interacts with these systems to produce one coherent interface for ease of use. However these systems can run on their own and produce output.

State:

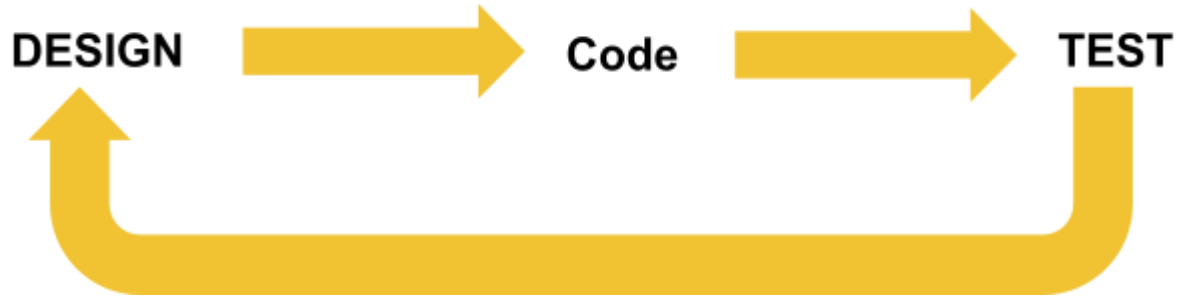
When it comes to state the relationship between the entrance and the floor lends itself to being state as before the entrance can seat people it looks at the state of the floor to determine whether it can do the operation or not. Thus the state is externalised from the entrance.

Strategy:

When it comes to customers paying the bill they can either split the bill or pay it in full. These methods of paying are interchangeable and controlled by the context of a random number that gets generated by the waiter. (The random number is for simulation purposes)

Methods and Technologies

Methods



Our approach to this project was guided by a well-defined methodology of design, code, and test. This structured workflow allowed us to maintain a systematic and iterative development process throughout the project's lifecycle.

Design: We initiated the project by thoroughly designing the system's architecture and components. This involved creating diagrams, specifying the interactions between various subsystems, and deciding on the specific design patterns to be employed. We held collaborative design sessions to ensure that all team members had a clear understanding of the system's structure and its individual components.

Code: With the design in place, we proceeded to the implementation phase. Each team member worked on their designated subsystems, following the design principles and design patterns we had outlined. Regular code reviews and discussions took place to maintain consistency and adherence to the chosen design patterns. By breaking the project into smaller, manageable parts, we ensured that coding tasks were efficient and well-structured.

Test: The testing phase was integral to our methodology. We conducted continuous testing of individual subsystems to verify that they were robust and met their intended functionality. This approach allowed us to catch and address issues early in the development process. It also ensured that every component of the system was tested thoroughly for functionality and reliability.

This methodical approach was particularly beneficial during the integration phase when combining everyone's subsystems into the overall system. The continuous testing and verification of individual parts made the integration process smoother, as we had confidence that each subsystem had been rigorously tested on its own. This methodology not only facilitated effective collaboration but also contributed to the successful completion of our restaurant simulation system.

Software

GitHub

All of the code sharing was done through a GitHub repository. Each member would have their individual branches in which they worked and committed to. Not all branches were merged to main since some code needed to be integrated between two branches to confirm that it works well together before committing to main. The integrated branch would be the branch merged with the main.

WhatsApp

A whatsapp group was created with all the group members as a way to ensure good communication amongst the team so that everyone was always informed about any changes or new information. We used this group to arrange meetings, voice questions and have short discussions about various project topics.

GitHub CodeSpaces/Liveshare extension

Liveshare was used in GitHub codespaces near the end of the project when subsystems were being integrated so that multiple people could work on the code at once, to speed up the integration. This was hosted by James, and constitutes the great discrepancy between his number of commits as opposed to the others.

Notion

Notion is a popular productivity and collaboration tool which was suggested by a team member to use. We used a Notion teamspace to plan, organise and manage our ideas and save them on this platform so that whenever someone needs to reference back to an idea or share their idea, they can always do so by posting them onto the various pages and sections created. Notion can be used on any platform which allows for convenience to access wherever and whenever.

Discord

Due to many group members living geographically far away, Discord was used extensively for having online meetings to discuss the project as well as being able to interact with each other during the coding process. It was also used to share various documents and UML diagrams.

Doxygen

Doxygen allowed us to document our code effectively, providing a clear and structured overview of the project's architecture, classes, functions, and their relationships. This documentation not only served as a comprehensive reference for the team but also facilitated seamless collaboration by ensuring that everyone understood the codebase and its functionalities.

Google Tests

We integrated Google Tests into our development workflow to ensure the reliability and correctness of our code. Google Tests provided a robust framework for writing and running unit tests, allowing us to validate individual components of our restaurant simulation independently. By conducting automated tests, we could identify and fix bugs promptly, ensuring the stability and accuracy of our program. This systematic approach to testing bolstered our confidence in the software's performance and contributed significantly to the overall quality of our simulation.

Valgrind

Valgrind was used to help identify and thoroughly eliminate any and all memory leaks in the program.

Conclusion

In conclusion, this report has provided a detailed overview of the implementation of a restaurant simulation system that incorporates at least 10 of the 23 Gang of Four design patterns. The system's objective was to replicate the inner workings of a real-world restaurant, complete with customers, tables, waiters, chefs, an ordering system, and a billing system. This project has highlighted the significance of design patterns in creating a robust and maintainable software system.

The system's implementation was thoroughly described, covering various aspects, including the high-level system overview, subsystems such as Customers, Entrance, Floor, Kitchen, and the Billing System, as well as the assumptions made during the development process.

One of the most significant contributions of this project was the effective utilisation of several design patterns, each serving a unique purpose in enhancing the system's structure and flexibility. The report provided in-depth insights into how each pattern was implemented and the specific benefits it brought to the project.

Moreover, the report discussed the methods and technologies employed during the development of the system. The team effectively leveraged version control using GitHub, facilitating collaborative development. Communication tools such as WhatsApp, Discord, and GitHub CodeSpaces/Liveshare extension enabled efficient team coordination and problem-solving. Notion served as a collaborative workspace for documenting ideas and tracking progress, while Doxygen was used to generate comprehensive technical documentation. Additionally, Google Tests and Valgrind were utilised to ensure the reliability and memory management of the software.

It is important to acknowledge that while this report highlighted the incorporation of 10 specific design patterns, other design patterns were also present in the system, albeit to a lesser extent. These patterns contributed to the overall organisation and modularity of the software.

In summary, the successful implementation of this restaurant simulation system showcases the power of design patterns in creating robust, maintainable, and extensible software. The collaborative effort of the team, combined with effective project management and the use of modern development tools, has resulted in a well-documented and functional system. This project serves as an excellent example of how the application of software engineering principles and design patterns can lead to the creation of complex, real-world systems.

References

Linda Marshall and Vreda Pieterse. *Tackling Design Patterns*, https://www.cs.up.ac.za/cs/lmarshall/TDP/TDP.html. Accessed 6 November 2023.