AlcodeholicsAnonymous

Group Project Report

COS 214 - 2023

Group Members

Scott Bebington	u21546216
Willie van Rooyen	u22675699
Jayson du Toit	u22571532
Ethan Groenendyk	u20743956
Charlize Hanekom	u22487222
Tim Whitaker	u22744968

Coding Standards

Naming Convention

- Camel Case
- Class names start with capital

Variables

- Names should be self-explanatory

Headers use '#pragma once' instead of '#ifndef….#define…'

- Reason: faster, less obtuse

Comments

- Make comments in header files to explain functions (not too long comments)

Brackets on the same line with function and class declarations

Other Notes:

- Use structs with classes that have no descendants or functions. Only public variables.
- Keep variables private. Update and use them using getters and setters.

Task 2: Design

2.1 Requirements

- Customers must be able to make bookings and be seated accordingly.
- Tables can be added together to seat larger parties or split up to seat more parties.
- Customers must be able to choose their meals and order it
- Customers can also add or remove certain ingredients to pre-existing meals and request specific preparation methods.
- A waiter must take the customer's order and deliver it back to the customer when it has been prepared.
- Customers must be able to decide to pay now or add to a tab.
- Customers are able to tip according to their satisfaction with the service provided.
- Communication must occur from the customer to the waiter to the chefs and back after the order has been prepared.
- The head chef must receive the order from the waiter.
- The head chef then delegates to the various category chefs according to the categories of the meals.
- The category chefs then prepare the meals delegated to them.
- When all the meals are done the head chef plates the order and notifies the waiter to take the finished order to the customer

2.2 Activity Diagrams

See Task 4.5 for activity diagrams.

2.3 Design Patterns

We will be using the following design patterns for the different systems as described:

Composite - The meals will consist of dishes and ingredients

Builder - To create the meals and get the final order product

Chain of responsibility - Handling of the order from waiter to head chef to category chefs, back to waiter and finally to the customer.

Iterator - To iterate through the tables to see availability and combine or split tables

State - For the state of the order, Ordered, Waiting, Prepared, Delivered

Observer - The waiting queue observes the Maitre D who observes the tables to see if one is available.

Mediator - The waiter will be the mediator for communication between the customer, head chef and Maitre D.

Singleton - To ensure that only one Head Chef is initialized.

Memento - To store the orders as tabs for later payment

Template - Waiter, Head Chef and Category Chefs will be children of Employee and Customer and Employee will be children of the Person class.

2.5 Complete UML Class Diagram

See Task 4.5 for UML Class Diagram

2.6 Complete Communication Diagram

See Task 4.5 for communication Diagram

2.7 Complete State Diagram

See Task 4.5 for state Diagram

2.7 Complete Object Diagram

See Task 4.5 for object Diagram

Task 4

4.1 Research (References)

A restaurant simulator aims to mimic a real life restaurant and as such one must understand the inner workings of restaurants. While we used our personal experiences of visiting restaurants, we also researched restaurant structures to see which "characters" need to do what in a restaurant for it to work efficiently (Lightspeed, 2020). We learned about *La Brigade de Cuisine* system a French organizational structure for restaurants with various chefs such as an executive chef, *chef de cuisine*, *sous chef* and line cooks or *chef de partie*. We also asked people who are in the service industry for some more background in order to see how a restaurant operates from a different perspective than a customer's.

We researched different restaurant simulator tycoon games such as Chef Life: A Restaurant Simulator, Cooking Madness and Cooking City to gain insight on how other programmers have approached this topic.

Designing and implementing such a product requires knowledge of design patterns to solve certain recurring problems in design situations (Marshall, 2023) and how to use these different patterns together in a program. For this we used the lecture slides and notes provided in the COS 214 course.

References

Kolibri Games. (2020, December 10). *Let's Build a Restaurant*. Kolibri Games. Retrieved November 6, 2023, from https://www.kolibrigames.com/blog/lets-build-a-restaurant/

Lightspeed. (2020, November 30). *Kitchen Brigade: Ultimate List of Kitchen Staff and Descriptions*. Lightspeed. Retrieved November 6, 2023, from https://lightspeedhq.com/blog/kitchen-brigade/

Marshall, L. (2023). *COS 214 Lectures*. ClickUP.

4.2 Design Decisions

For the structure of our restaurant we decided to simplify the kitchen brigade system to fulfill our needs and remove any clutter. We therefore only have a head chef and category chef.

Our restaurant simulator aims to give the user the experience of being a customer at a restaurant, therefore they choose what they want to order from the menu provided and also whether they want to add their order to a tab or pay for it immediately after enjoying the meal.

As for the workings of the floor of the restaurant, we decided that waiters and tables would have a one-to-one relationship with each other and that all the waiters will bring the orders from the customers to the head chef.

In the kitchen, the head chef will delegate the different meals and dishes to category chefs such as a meat chef, seafood chef, fry chef and drinks

4.3 Design Patterns

Design patterns explained

A **Composite** design patterns was used to create the structure of dishes that contain meals. This helped create an understandable structure with only dishes having children and not ingredients, seeing as ingredients are the lowest level.

The **Iterator** design pattern was used for interaction between the floor and the separate tables. The tables are treated as nodes in a doubly linked list with next and previous pointers. This doesn't affect how other classes interact with the Table class. The Floor class is used to add Tables to the list of tables, just how you add tables to the floor in any normal restaurant.

The **Builder** design pattern is used in our program to allow different chefs to make different parts of a final order. This is more efficient than having one chef make all the food.

The **Chain Of Responsibility** design pattern was used to move an order from the customer to the chefs and then back to the customer when it has been prepared.

The **Observer** design pattern was used by the MaitreD to observe when a table had become available and to then notify the bookings or people waiting in the queue.

The **Memento** design pattern was used to implement the tab system, creating a backup of the order and storing it in a tab which is recognised by a customer number and stored in a database.

The **State** design pattern was used to indicate the progress of the order, whether it is in an Ordered, Waiting, Prepared or Delivered state.

The **Mediator** design pattern was used to facilitate the communication between the customer and the head chef. The waiter is the mediator and is notified when the customer is ready to order, then it takes the order to the Head Chef. When the order is completed the waiter is once again notified, whereafter he delivers the order to the customer.

The **Singleton** design pattern was used to ensure that only one Head Chef is initialized as having more than one head chef would interrupt the whole kitchen process.

The **Template** design pattern includes the Person class and its children. Waiter, Head Chef and Category Chefs will be children of Employee and Customer and Employee will be children of the Person class.

Problems we faced

Our main problem we faced when implementing the design patterns was circular references. This was the case with the mediator design pattern and the chain of command design pattern. We solved these issues by removing unnecessary includes in the header files.

We had many segfault errors caused but not correctly setting values in their respective classes leading to function calls on nullptr values.

We had problems with the tab feature where the tab was constantly being cleared before the customer decided to pay.

4.4 Assumptions

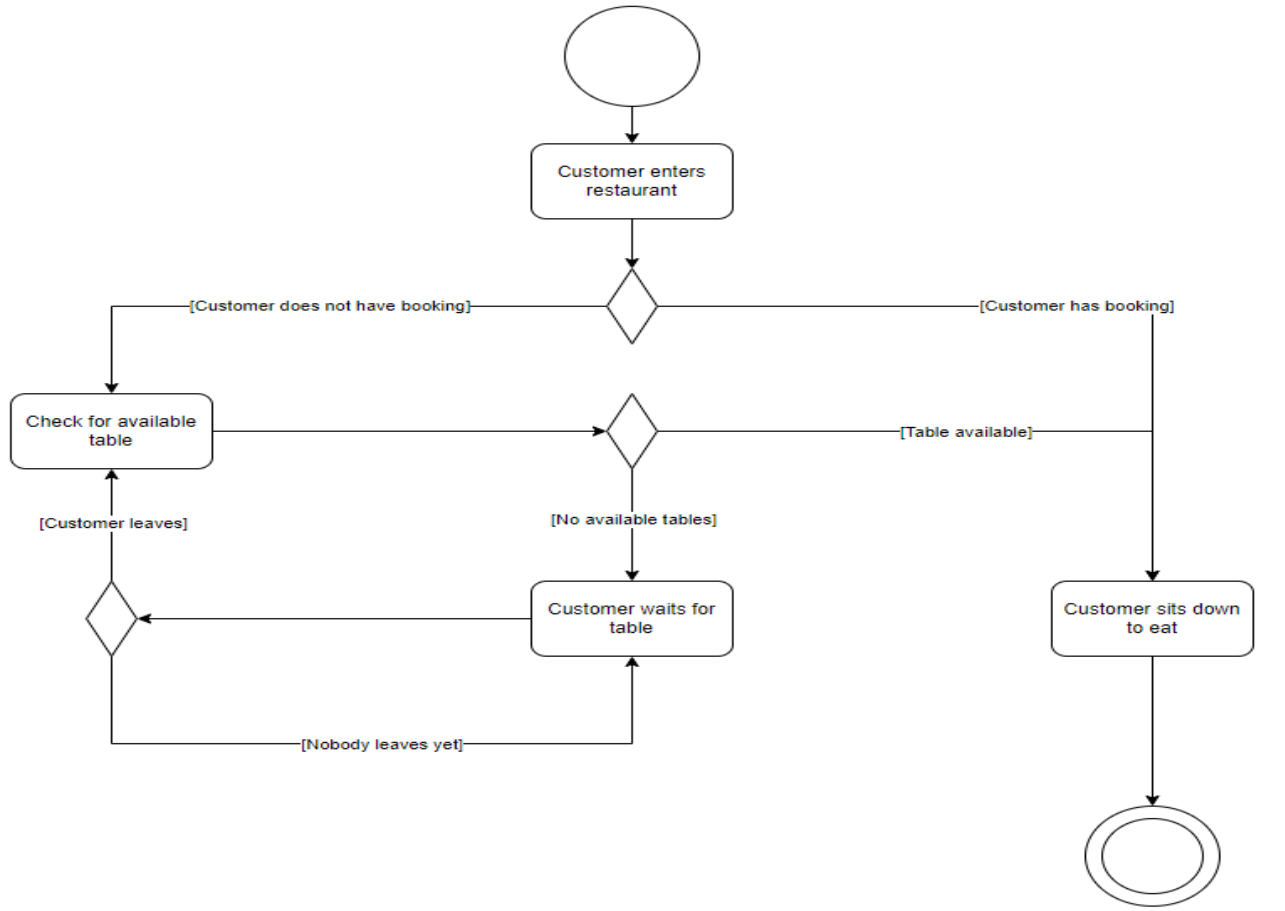
We assumed that there only exists one Head chef with different category chefs under his lead, each able to completely make their type of food. As an example the pizza chef can make all the pizzas and the drinks chef/bartender makes all the drinks.

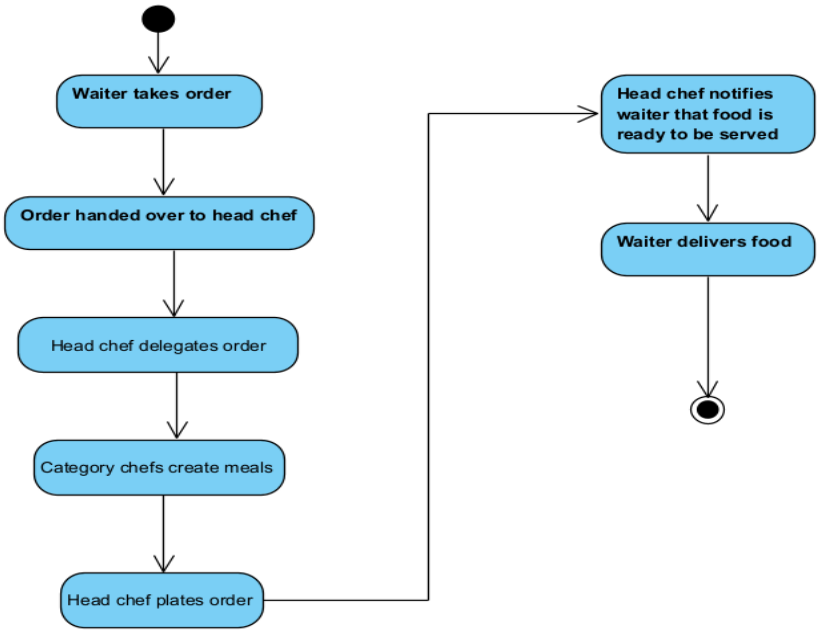
Further, we assumed that customers are able to order multiple meals and only pay at the end with either an accumulated bill or to add to their tab.

The tables where the customers are seated all only have one waiter tending to it that is notified by the customers of things like ‘ready to order’ and ‘finished eating’. All the waiters are part of a singular hive mind and act as one mediator but with different bodies. The maitre D handles all seating arrangements and bookings in the restaurant. Customers can have bookings with the maitre D seating them at booked tables.

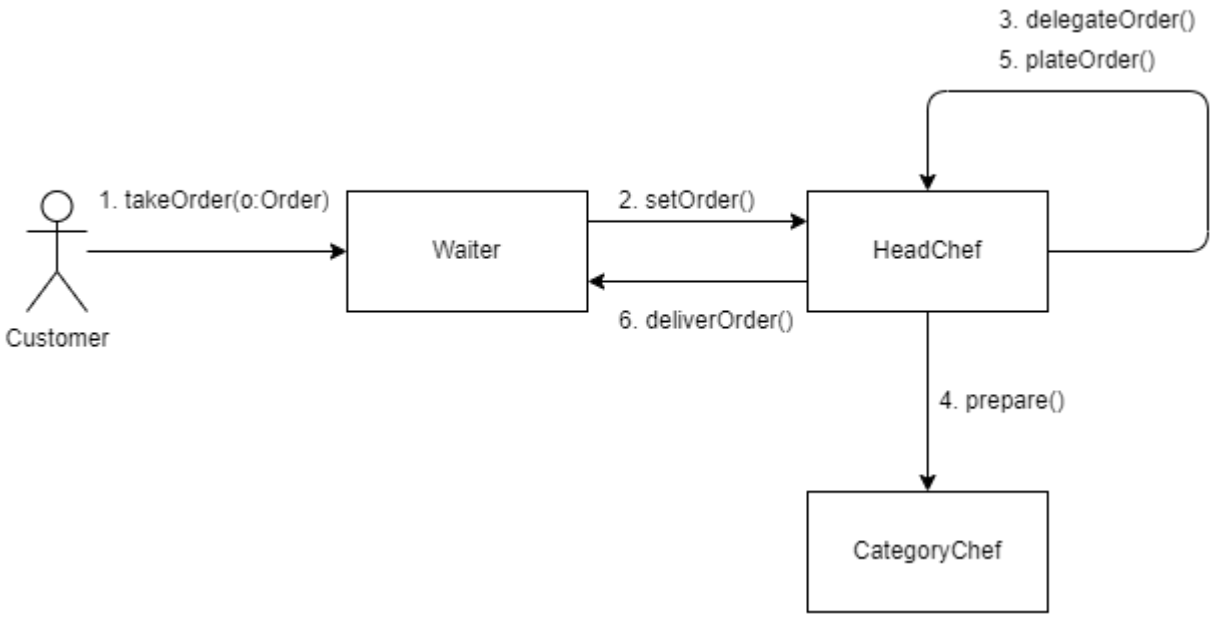
4.5 UML Diagrams

Activity Diagrams

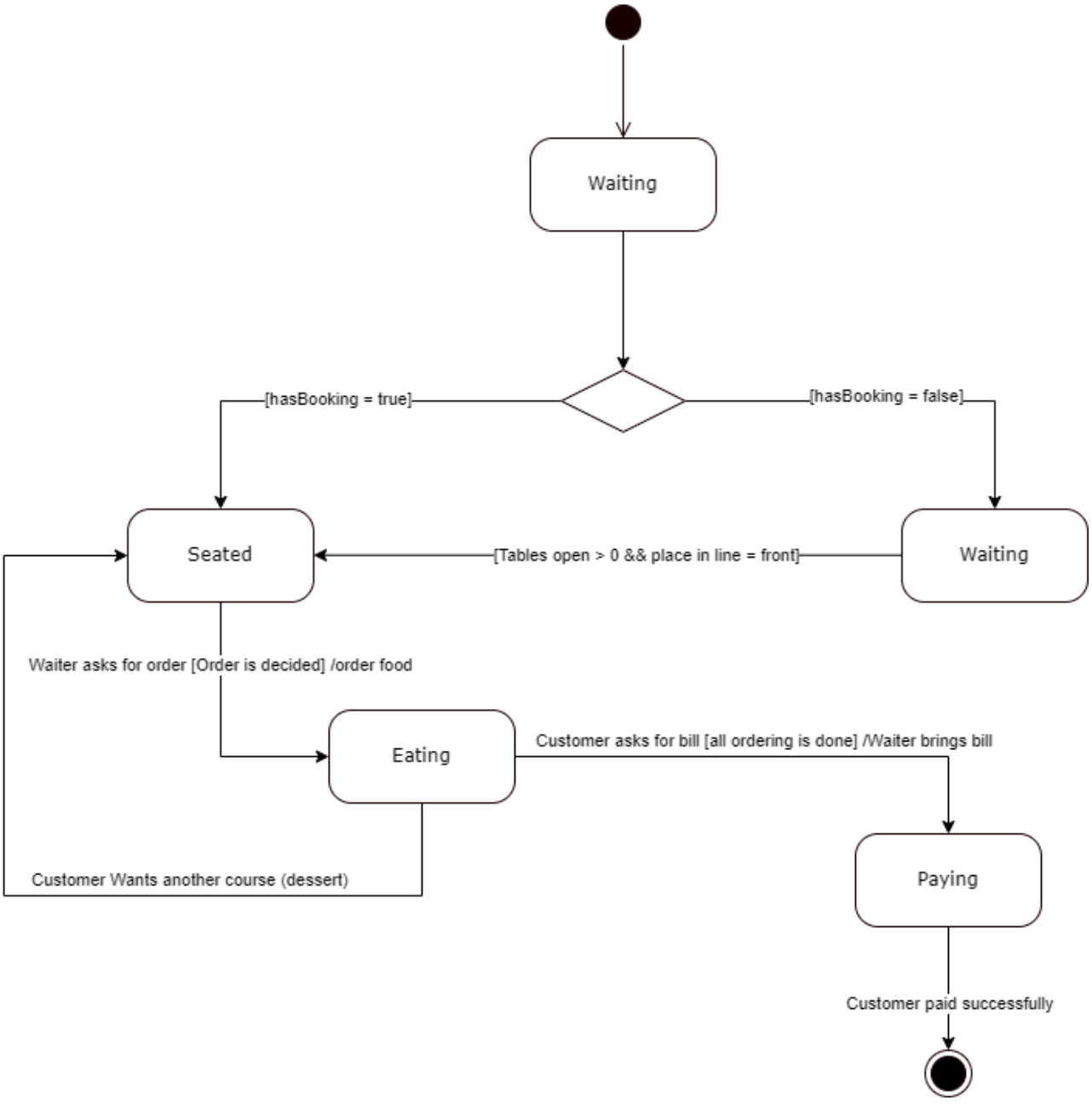




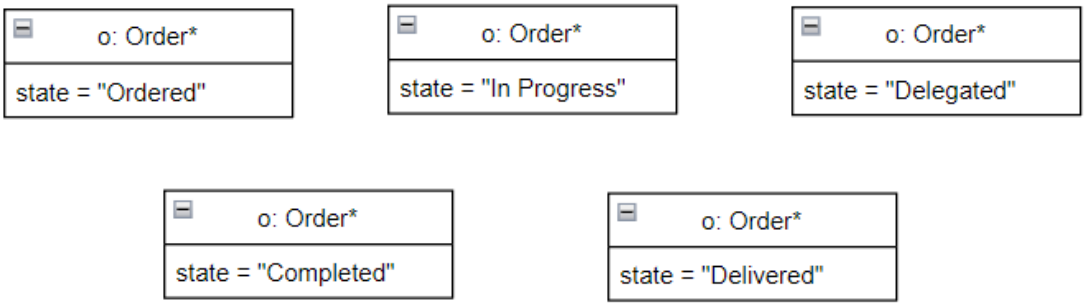
Communication diagram



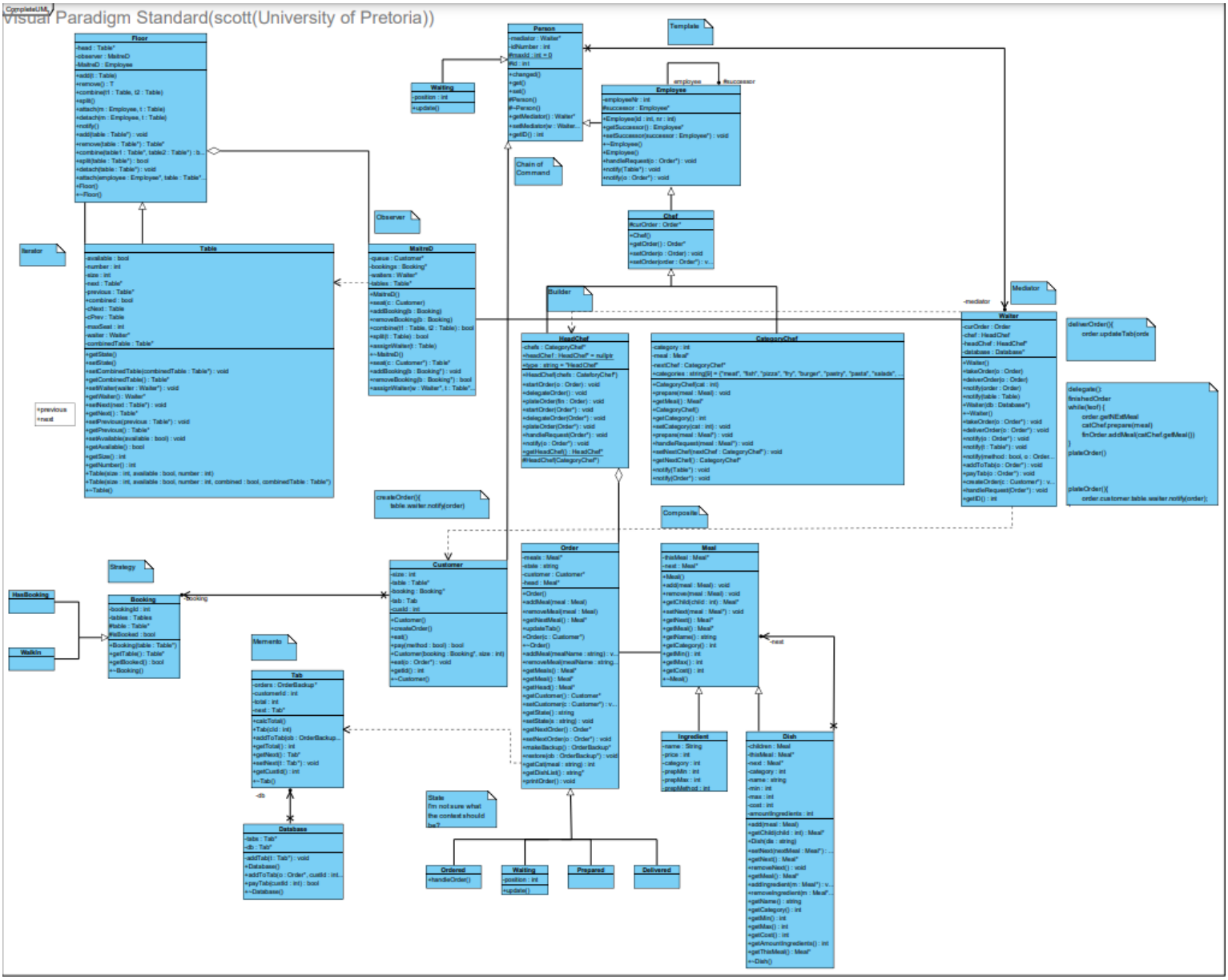
State Diagram



Object Diagram



UML Class Diagram



Sequence Diagram