COS214 Report - Flyweight Fighters

GitHub Repository

```
https://github.com/GeekGurusUnion/214-Project.git
```

[Alternate link]: https://github.com/GeekGurusUnion/214-Project/

Running the Program

There are no pre-requisites to run this program.

Building

```
make
```

Run

```
make run
```

Unit Tests

Testable files: `unittest_Facade.cpp`, `unittests_Dawie.cpp`, `unittests_Iwan.cpp`, `unittests_Dawie.cpp`, `unitTests_Tiaan.cpp`, `unitTests_Xavier.cpp`

```
# Example: Testing unittest_Facade.cpp on the GTest/GMock framework
make test INCLUDE_TEST=unittest_Facade.cpp
```

Leak Checks

On MacOS:

```
make leaks
```

On Windows/Linux:

```
make valgrind
```

Introduction

This report presents the design of a restaurant simulator in C++, focusing on the utilization of various design patterns learned in class. The primary objective of this assignment was to create a comprehensive simulation of restaurant operations, containing both the floor and kitchen aspects. The simulation is intended to mimic the complexities of a restaurant environment, including customer management, order processing and kitchen operations, all while demonstrating the application of design patterns covered in the coursework.

In this report, we will address the following components:

Research and References:

We will discuss the research conducted during the project, which includes restaurant management practices, the design patterns learned in our coursework, as well as research on coding and Git standards, specifically the Git Flow strategy. We will also provide references for the sources used to inform our design decisions and relative practices.

Design Decisions and Design Patterns:

We will explain why we decided to use certain design patterns to create the restaurant simulation. This section will provide insights into the thought process behind our design choices and the specific design patterns that were employed to address various challenges within the restaurant simulation. This section will also note the problems each pattern solves in our implementation. To enhance the understanding of our design and architecture, we will incorporate UML diagrams that

visually depict the relationships and structures of components and design patterns within the restaurant simulator.

Diagrams:

This section serves as a visual supplement to our report, providing a collection of essential UML (Unified Modeling Language) diagrams that offer a comprehensive depiction of the design and architecture of our restaurant simulator.

Through this report, we aim to offer a comprehensive overview of our restaurant simulation, emphasizing the role of design patterns learned in our coursework in achieving a realistic and efficient model of restaurant operations. The subsequent sections will delve into each of the components in greater detail, providing a view of the project's research design and its alignment with the principles of design patterns.

Research and References

Coding Standards

1. Organizational

1.1. Use a version control system

- Never keep files checked out for long periods (small incremental updates (see Git Standards 1.1. for more info)).
- Ensure that checked-in code doesn't break the build (GitHub Actions doesn't do everything).

1.2. Code Reviews

- Peer-review other's work so that you understand what is going on and ensure their code isn't breaking your code.
- Make a GitHub issue to describe the problem and assign the responsible person to the issue.

1.3. File Names

Filenames should be all lowercase and can include underscores.

Examples of acceptable file names:

`my_useful_class.cpp` `myusefulclass.cpp`

2. Design Style

2.1. Give entity *one* cohesive responsibility

- For each entity, focus on one thing at a time.
- Give each entity (variable (member), class, function) one well-defined responsibility.
- As the entity grows, the scope increases, but it should not diverge.

2.2. KISS (Keep It Simple Software)

- Correct is better than fast.
- Simple is better than complex.
- Clear is better than cute.
- Safe is better than insecure

2.3. Minimize global and shared data

Sharing causes contention. Avoid shared data, like global data. This increases coupling which reduces maintainability.

2.4. Ensure resources are owned by objects

Never allocate more than one resource (pointer) in a single statement. This eases the process of memory deallocation.

2.5. Optimize for the reader, not the writer

More time is spent reading code than writing it.

3. Coding Style

3.1. Use `const` proactively

`const` (immutable) variables are easier to understand and to track. It's safe and checked at compile time.

3.2. Declare Variables as locally as possible

Variables introduce state, and you should have to deal with as little state as possible, with lifetimes as short as possible.

3.3. Always initialize variables

This is a common source of C++ hotfixes. Initialize variables upon definition.

3.4. Avoid long functions

Excessively long functions and nested code blocks are often caused by failing to give one function one cohesive responsibility (As explained in **2.1**).

3.5. Minimize Definitional Dependencies

Don’t be over-dependent: Don’t #include a definition when it is not needed (or included by its parent anyways).

3.6. Always write internal #include guards

Prevent unintended multiple inclusions by using #include guards.

3.7. Don't use `using namespace std;`

Rather use a using-declaration which lets you use cout/cin/string without qualification

```
using std::cout;
cout << "Values:";
```

4. Functions and Operators

4.1. Order parameters according to their value, pointer or reference

Distinguish among input, output, and input/output parameters, and between value and reference parameters.

4.2. Avoid overloaded operators as far as possible

Overload operators only for good reasons. It's easy to misuse operator overloading and cause confusion among fellow coders.

5. Class Design and Inheritance

5.1. Use design patterns!

Because we should XD!

5.2. Each `new` should be coupled with an `delete`

Basic thought for memory deallocation.

5.3. The `#define` guard

As an example, the file `foo/src/bar/baz.h` in project `foo` should have the following guard:

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_

...

#endif  // FOO_BAR_BAZ_H_
```

5.4. Declaration Order

Within each section, prefer grouping similar kinds of declarations together, and prefer the following order:

- Types and type aliases (typedef, using, enum, nested structs and classes, and friend types)
- (Optionally, for structs only) non-static data members
- Static constants
- Factory functions
- Constructors and assignment operators
- Destructor
- All other functions (static and non-static member functions, and friend functions)
- All other data members (static and non-static)

6. Construction, Destruction and Copying

6.1. Define and initialize member variables in the same order

Agree with your constructor's parameters: member variables are initialized in the order they are declared.

7. Error Handling and Exceptions

Prefer using exceptions over `cout`. This keeps the output clean.

Sources

http://micro-os-plus.github.io/develop/sutter-101/

https://google.github.io/styleguide/cppguide.html

https://stackoverflow.com/questions/1452721/why-is-using-namespace-std-considered-bad-practice

https://refactoring.guru/design-patterns

https://www.cs.up.ac.za/cs/lmarshall/TDP/TDP.html

Git Standards

1. Repository Rules

- Fork the Organization's repo to work and test code locally - **fork ONLY the `main` branches**
- Reduce the frequency of pull requests unless the advancement is impeded by a required feature from a particular team member.

1.1. Basic Workflow

1. Make sure to fetch latest updates from the organization's repo onto your forked repo by running

```
git fetch upstream && git merge upstream/dev
```

1. **Make sure you are working in `origin/main`**
2. Commit frequently to the respective branch, but also narrow-down commits to provide clarity
3. After a night's work, create a pull request to merge onto organization's repo such that other members can sync with your latest changes.
4. **Make sure your pull request directs to the correct branch.**

Example workflow: When working on the `dev` branch, use `git fetch upstream/dev`. You can make your own branches in your forked repo as needed. When creating a Pull Request, select the `dev` branch on the Organization side (see image above). This applies to all branches (e.g. working with `hotfix` branch causes a PR to be directed to the organization's `hotfix`-branch)

2. Committing Code

- Make atomic commits of changes, even across multiple files, in logical units. That is, as much as possible, each commit should be focused on one specific purpose.
- As much as possible, make sure a commit does not contain unnecessary whitespace changes. This can be checked as follows:

```
$ git diff --check
```

3. Commit Messages

For consistency, try and use the imperative present tense when creating a message. Examples:

- Use "Add tests for" instead of "I added tests for"
- Use "Change x to y" instead of "Changed x to y"

4. Branching

4.1. Main Branches

Our main repository will have `main` as the evergreen branch.

4.2. Supporting Branches

To aid in ease of tracking new features, a few sub-branches have been added:

- Feature branches
- Dev branches
- Hotfix branches

These branches may have a limited lifetime and will be removed eventually.

4.3. Feature Branches

4.3.1. Naming Convention

Feature Branches must be named `feature-<featureClassification>` (referred to as `feature-id`).

4.3.2. Merging Feature Branches

These feature-branches must be merged onto the `dev`-branch.

```
$ git checkout -b feature-id dev    // creates a local branch for the new feature
$ git push origin feature-id        // makes the new feature remotely available
```

And for actually merging onto `dev`-branch:

```
$ git merge dev
```

When development on the feature is complete, the lead (or engineer in charge) should merge changes into master and then make sure the remote branch is deleted.

```
$ git checkout dev                  // change to the master branch
$ git merge --no-ff feature-id      // makes sure to create a commit object during mer
ge
$ git push origin dev               // push merge changes
$ git push origin :feature-id       // deletes the remote branch
```

4.4. Hotfix Branches

4.4.1. Naming Convention

Feature Branches must be named `hotfix-<hotfixClassification>` (referred to as `hotfix-id`).

4.4.2. Merging Hotfix Branches

These hotfix-branches must be merged onto the `main`-branch.

```
$ git checkout -b hotfix-id master          // creates a local branch for the new hotf
ix
$ git push origin hotfix-id                   // makes the new hotfix remotely availab
le
```

And for actually merging onto the `main`-branch:

```
$ git merge master                    // merges changes from master into hotfix branch
```

When development on the hotfix is complete, [the Lead] should merge changes into master and then make sure the remote branch is deleted.

```
$ git checkout master                                 // change to the master branch
$ git merge --no-ff hotfix-id       // makes sure to create a commit object during mer
ge
$ git push origin master                              // push merge changes
$ git push origin :hotfix-id                             // deletes the remote branch
```

Sources

https://gist.github.com/digitaljhelms/4287848

https://gist.github.com/digitaljhelms/3761873

System Requirements

The floor

1. Enable a customer to request to be seated and if available give them a table and show them to said table.
2. A waiter should take orders, Customer should be able to ask the waiter to come back later.
3. The waiter should pass orders to the kitchen, and when ready, deliver them to the table.
4. A bill Should be presented at the end of the meal. The Bill should be splitable into sub-bills. allowing for more than one payment from the party.
5. Tables must be combinable or splitable to be able to fit various sized parties
6. Restaurant should be walk in based.
7. Waiters will be assigned tables that they are responsible for.
8. Customers should be able to tip or complain.
9. Customers should be able to fully customize their orders from an available list of customizations.

The Kitchen

1. Waiter should pass an order to the kitchen
2. The system should have a head chef that acts as a central point of communication for the kitchen
3. Kitchen should notify the waiter when the order is completed.
4. Different chefs should be responsible for different parts of the preparation process

5. The dish should travel between multiple stations to be finished. Before being handed back t the head chef.

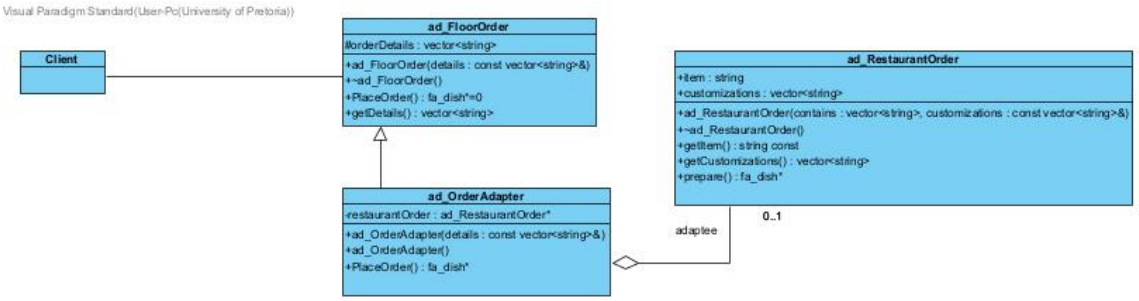
Design Decisions and Design Patterns

In this section, we will dive into the reasoning behind the design decisions of the restaurant simulation, specifically focusing on the use of design patterns and what problems they solved in our implementation. To achieve a well-structured and efficient implementation, we employed several design patterns and each of these patterns was selected to address specific challenges in the restaurant simulation. UML diagrams that visually depict the relationships and structures of components and design patterns within the restaurant simulator will be provided.

Adapter

- **The Problem**:
  - The kitchen requires a an approach to process the "raw" orders from waiters.
- **The Solution**:
  - Implement an Adapter Pattern to convert written orders into **2** structured vectors: Item and Customizations.
- **Abstraction**:

  Similar to a waiter entering a written order slip into a system. Which then sends the re-structured order to the kitchen.
- **Methodology**:
  - Using the Object Adapter variant of the Adapter Pattern. Translate an `ad_FloorOrder` (a list representation of an order) into an `ad_RestaurantOrder` (2 structured vectors) using the `ad_OrderAdapter`.
- **Reasoning:**
  - Ensures consistent order formatting arrives at the kitchen
  - Decouples the order-taking process from the order preparation process.Singleton
  - **The Solution**:



- Ensure that the kitchen has only one head chef that manages the orders and delegates tasks to chefs.
- Implement a Singleton Pattern to create and ensure a single instance of the `si_headChef`.

Singleton

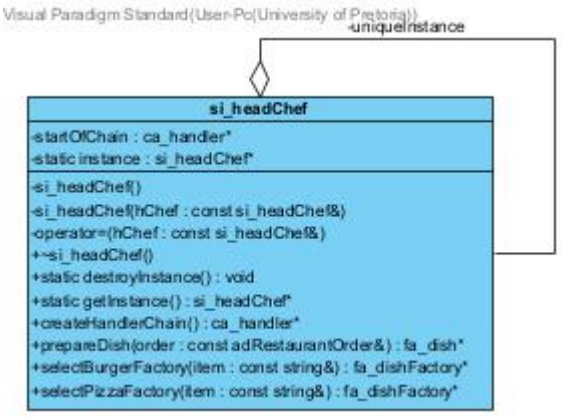
- **The Problem:**
  - The kitchen requires a single head chef that acts as a central point of communication between all kitchen staff.
- **The Solution:**
  - Implement a Singleton Pattern to create and ensure a single instance of the si_headChef.
- **Abstraction**:

  Similar to a normal restaurant environment. There should be only one head chef. To ensure efficient operations within the kitchen. This is achieved by delegating the different stages of food prep to various personnel within the kitchen.

Similar to a normal restaurant environment. There should be only one head chef. To ensure efficient operations within the kitchen. This is achieved by delegating the different stages of food prep to various personnel within the kitchen.

- **Methodology**:
  - `si_headChef` is made a singleton to centralize kitchen management, creating a single point of entry for order handling and dish preparation.
  - Private constructors and static methods enforce a single instance.
  - The `si_headChef` delegates dish creation to the relevant line chefs. It then delegates customizations to the customization handlers.
- **Reasoning**:

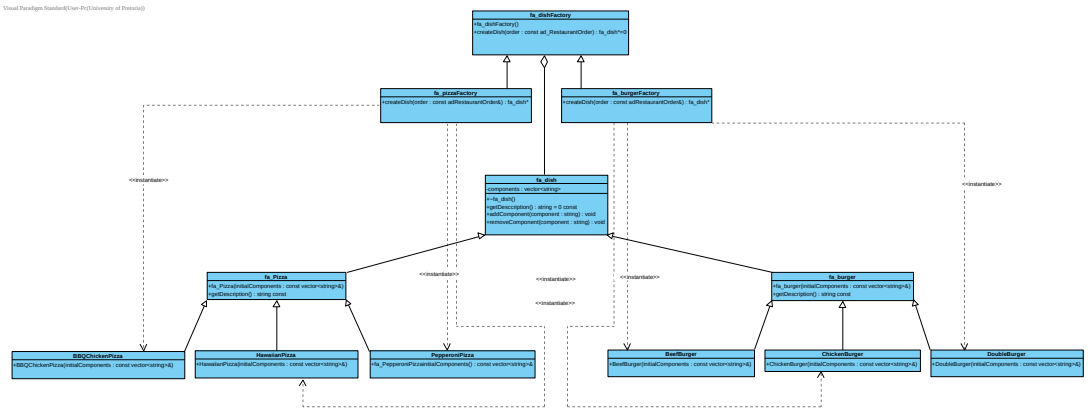
- A single instance avoids conflicts in kitchen management and ensures that all orders are processed uniformly.
- Having one head chef leads to efficient kitchen operations.



Factory Method

- **The Problem**:
  - The restaurant needs to be able to create different types of dishes like pizzas and Burgers whilst allowing for the addition of new dish types in the future.
- **The Solution**:
  - Use a Factory method Pattern to handle the creation of different dish types.
- **Abstraction**:
  - A chef has a specialty, Which allows it to create a specific type of dish. Before handing it back to the head chef.
- **Methodology**:
  - An abstract base class `Dish` defines the interface for the dishes.
  - Concrete classes like `Pizza` and `Pasta` inherit from `Dish` and implement specific methods of construction.
  - A `DishFactory` provides the ability to `createDish` which instantiates and returns the appropriate `Dish` These classes act as chefs.
- **Reasoning**:
  - By centralizing the dish creation process the complexity associated with managing multiple creation points throughout the application is minimized.

- The pattern promotes loose coupling between the code that requests a new dish and the code that knows how to make it. which leads to a more robust system.



Chain of Responsibility

- **The Problem**:
  - The restaurant needs to handle customizations. such as adding or removing certain ingredients. Based on customer preference
- **The Solution**:
  - Using the Chain of Responsibility pattern, pass the customization requests along a chain of handlers. Until one of them handles the request.
- **Abstraction**:
  - The head chef passes a completed order to the customizations chai. This is a chain of chefs where the dish passes from chef to chef and each chef determines if they are capable of adding or removing a certain ingredient which they are responsible for.
- **Methodology**:
  - The `ingredientHandler` is an abstract class for defining a method of handling ingredient customizations.
  - Specific handlers such as `AddingredientHandler` and `RemoveingredientHandler` implement the `ingredientHandler` to address particular customizations. These handlers allow for variation in construction. Where the system specifies what ingredient a handler adds or removes.

- Handlers determine their capability to process a request with the `canHandle` method, and they process the request with the `handle` method if it's within their responsibility.
- The head chef is responsible for constructing the customization chain.

- **Reasoning**:
  - This dynamic creation and management of handlers make the system flexible allowing for addition or removal of handlers without drastic changes to the system.
  - New handlers can be introduced and integrated into the chain without affecting existing handlers.

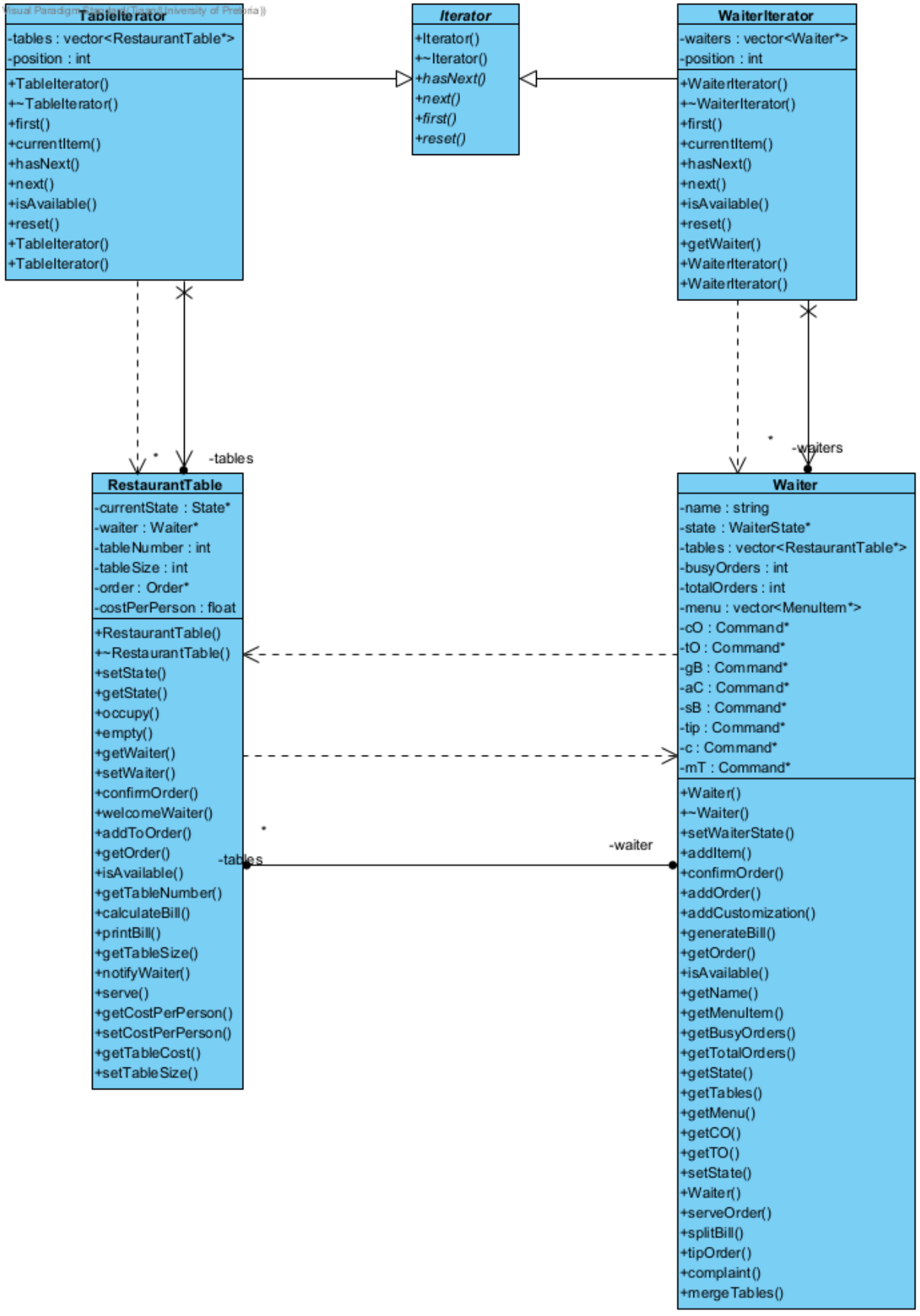
Design Decisions for the Floor Operations

Iterator Pattern:

The foundation of our design for the floor side involved the use of the Iterator pattern. This pattern enabled us to manage waiters and tables systematically. We had two different Iterators:

1. `WaiterIterator`
2. `TableIterator`

Waiters needed to execute various commands such as taking orders, customizing orders, confirming orders to be sent to the kitchen and generating a bill for tables. The Iterator pattern facilitated the traversal of both waiters and tables, allowing us to efficiently manage orders, check table availability, and match waiters with suitable tables. This pattern was instrumental in ensuring the orderly flow of operations on the restaurant floor.

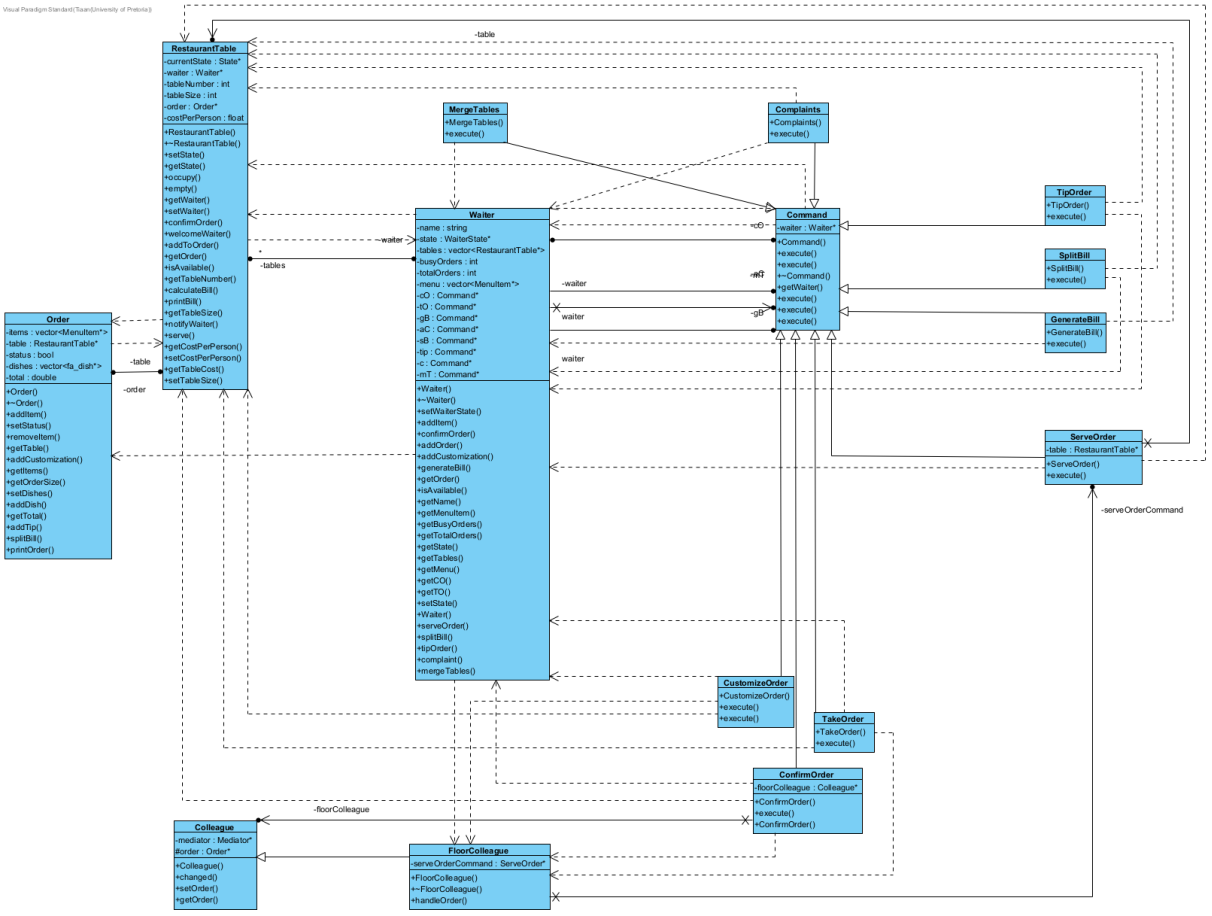

TableIterator
-tables : vector<RestaurantTable*>
-position : int
+TableIterator()
+~TableIterator()
+first()
+currentItem()
+hasNext()
+next()
+isAvailable()
+reset()
+TableIterator()
+TableIterator()
Iterator
+Iterator()
+~Iterator()
+hasNext()
+next()
+first()
+reset()
WaiterIterator
-waiters : vector<Waiter*>
-position : int
+WaiterIterator()
+~WaiterIterator()
+first()
+currentItem()
+hasNext()
+next()
+isAvailable()
+reset()
+getWaiter()
+WaiterIterator()
+WaiterIterator()
-tables
-waiters
RestaurantTable
-currentState : State*
-waiter : Waiter*
-tableNumber : int
-tableSize : int
-order : Order*
-costPerPerson : float
+RestaurantTable()
+~RestaurantTable()
+setState()
+getState()
+occupy()
+empty()
+getWaiter()
+setWaiter()
+confirmOrder()
+welcomeWaiter()
+addToOrder()
+getOrder()
+isAvailable()
+getTableNumber()
+calculateBill()
+printBill()
+getTableSize()
+notifyWaiter()
+serve()
+getCostPerPerson()
+setCostPerPerson()
+getTableCost()
+setTableSize()
Waiter
-name : string
-state : WaiterState*
-tables : vector<RestaurantTable*>
-busyOrders : int
-totalOrders : int
-menu : vector<MenuItem*>
-cO : Command*
-tO : Command*
-gB : Command*
-aC : Command*
-sB : Command*
-tip : Command*
-c : Command*
-mT : Command*
+Waiter()
+~Waiter()
+setWaiterState()
+addItem()
+confirmOrder()
+addOrder()
+addCustomization()
+generateBill()
+getOrder()
+isAvailable()
+getName()
+getMenuItem()
+getBusyOrders()
+getTotalOrders()
+getState()
+getTables()
+getMenu()
+getCO()
+getTO()
+setState()
+Waiter()
+serveOrder()
+splitBill()
+tipOrder()
+complaint()
+mergeTables()
-tables
-waiter

Command Design Pattern:

Each waiter was equipped with the Command design pattern to execute a series of tasks effectively. The Command pattern allowed us to encapsulate specific actions into objects, providing a way for waiters to take orders, confirm orders, customize orders and generate bills. We designed four distinct commands for waiters:

1. `TakeOrder`: This command allowed a waiter to take a customer's order, selecting items from the restaurant's menu and building the order.
2. `ConfirmOrder`: After taking an order from a table, the waiter could confirm the order, which notified the mediator to pass it through to the kitchen for preparation.
3. `CustomizeOrder`: In addition to the standard order-taking, this command enabled waiters to accommodate customer preferences by customizing the order. For example, customers could request specific preparations, such as adding Extra Cheese or Removing Garlic, and the waiter would ensure these preferences were added onto the order.
4. `Generate Bill`: After the customers had finished their meals, waiters used this command to calculate the total bill for the table. The generated bill was then presented to the customers.
5. `complaints`: After the customers had finished their meals, the waiters use this command to receive and complaints that the clients might have.
6. `mergeTables`: In the case that the table is not big enough to fit all the customers, there is an option to merge the tables in order to accommodate all of the customers.
7. `splitBill`: Before paying the bill, the customers has the choice to split the bill between the rest of the customers that is seated at the table.
8. `tipOrder` : When presented with the bill, the customer has a choice to add a tip to the bill which would then be paid to the waiter.

This modular approach enhanced the flexibility of the waiters, allowing them to interact with tables and orders in a structured and extensible manner. The Command pattern played a pivotal role in orchestrating these essential tasks within the restaurant floor operations, promoting modularity and maintainability.



State Design Pattern:

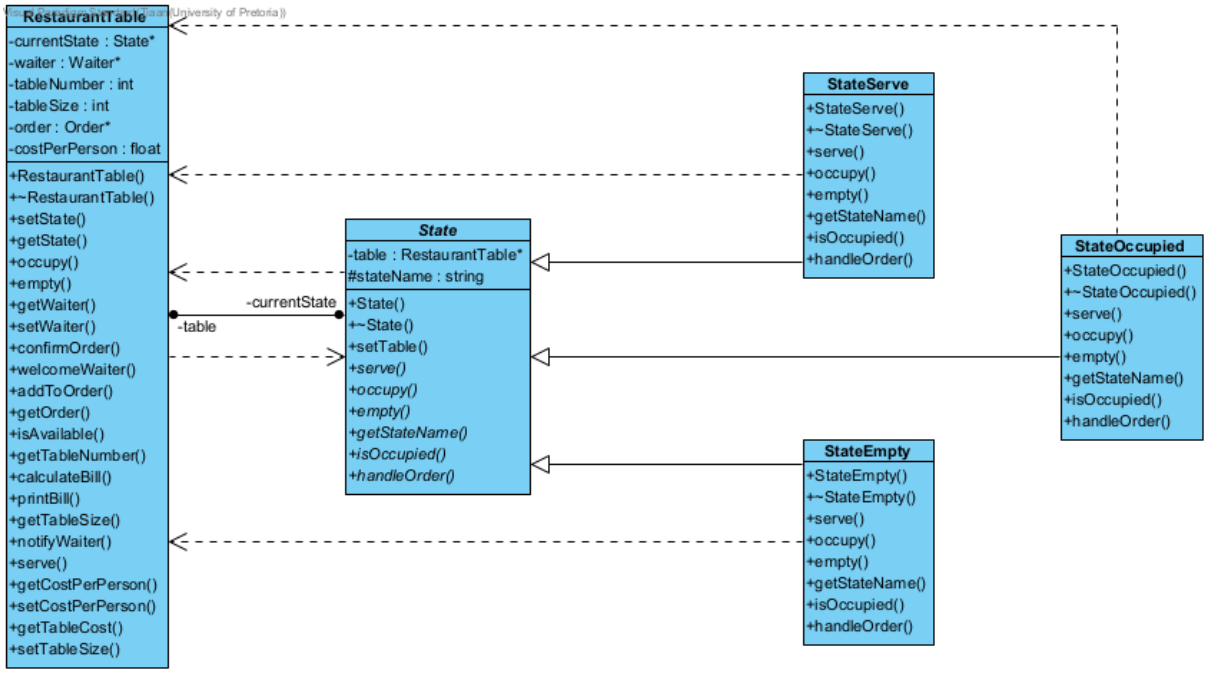
The State Design Pattern played a crucial role in maintaining the state of tables and determining the availability of waiters to take orders. Tables could transition between three distinct states:

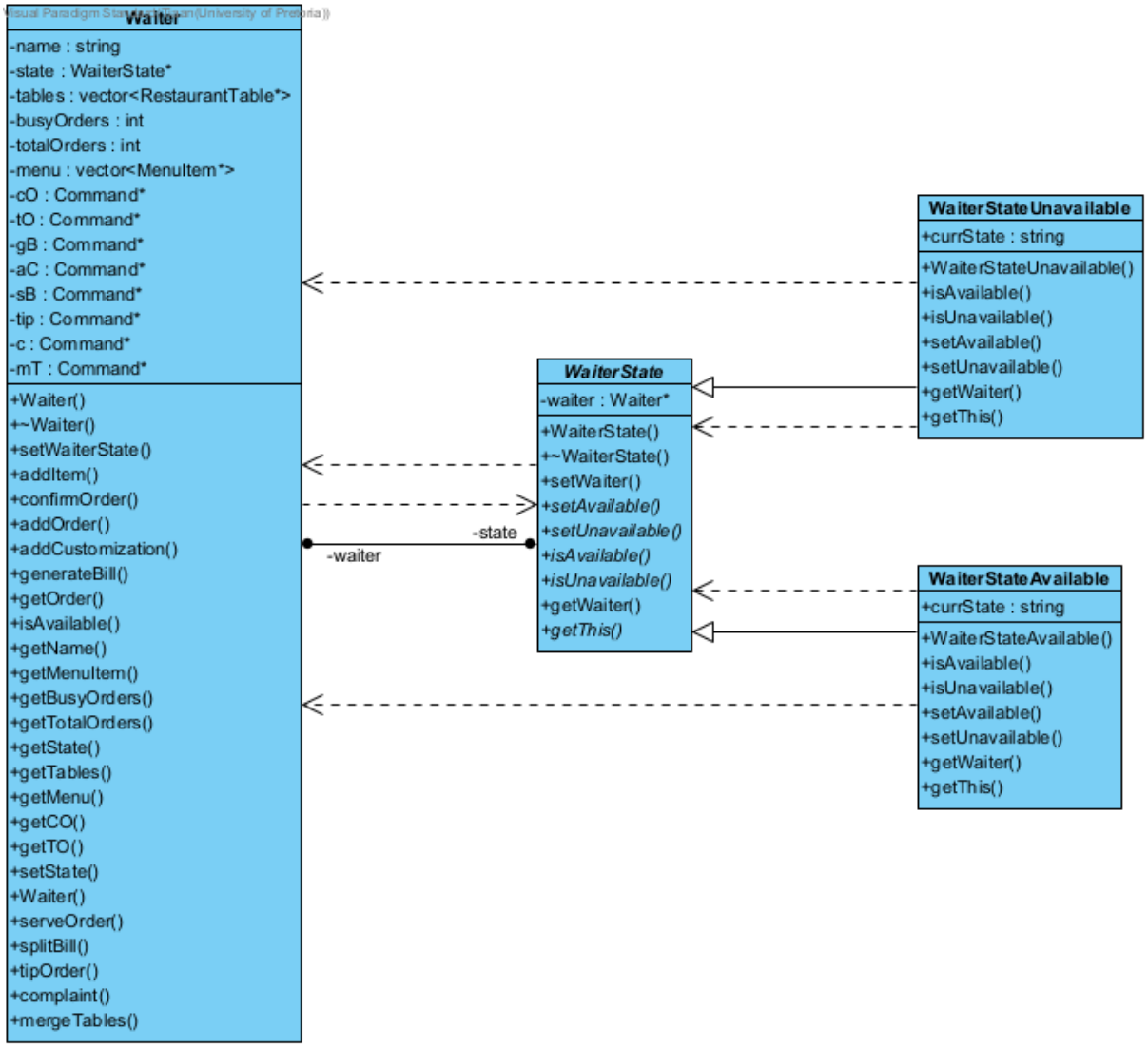
1. `Empty`: The Empty state signified a vacant table.
2. `Occupied`: The Occupied State indicated a table is currently in use.
3. `Served`: The Served State implied that the table had been served.

For waiters, the state pattern allowed them to indicate their availability:

1. `WaiterStateAvailable`: Indicating a waiter is available to take or add more orders.
2. `WaiterStateUnavailable`: Indicating a waiter is unavailable to take more orders.

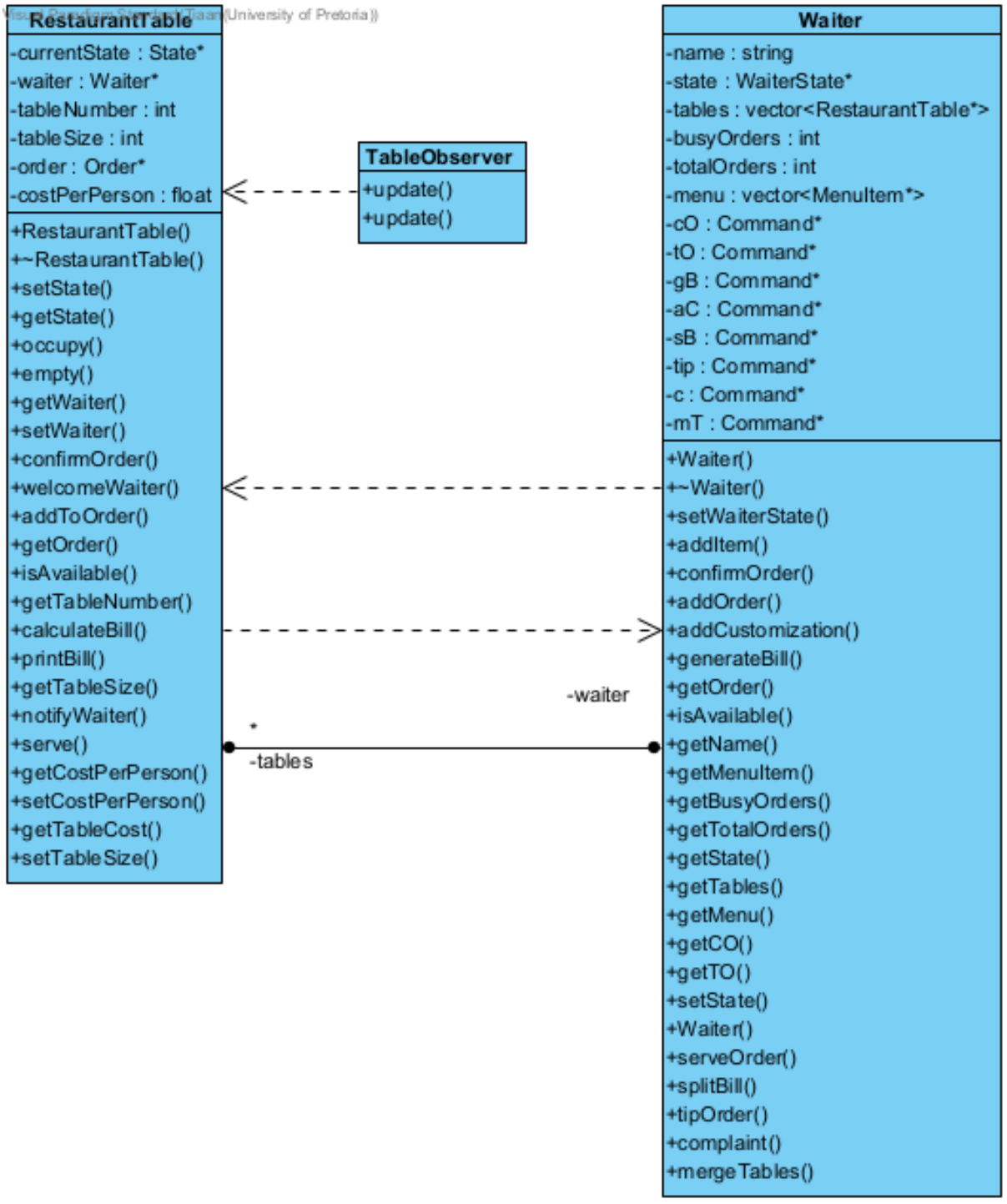
By using the State pattern, we achieved a flexible and organized system for managing the dynamic nature of tables and waiters in the restaurant.


RestaurantTable
-currentState : State*
-waiter : Waiter*
-tableNumber : int
-tableSize : int
-order : Order*
-costPerPerson : float
+RestaurantTable()
+~RestaurantTable()
+setState()
+getState()
+occupy()
+empty()
+getWaiter()
+setWaiter()
+confirmOrder()
+welcomeWaiter()
+addToOrder()
+getOrder()
+isAvailable()
+getTableNumber()
+calculateBill()
+printBill()
+getTableSize()
+notifyWaiter()
+serve()
+getCostPerPerson()
+setCostPerPerson()
+getTableCost()
+setTableSize()
-currentState
-table
State
-table : RestaurantTable*
#stateName : string
+State()
+~State()
+setTable()
+serve()
+occupy()
+empty()
+getStateName()
+isOccupied()
+handleOrder()
StateServe
+StateServe()
+~StateServe()
+serve()
+occupy()
+empty()
+getStateName()
+isOccupied()
+handleOrder()
StateOccupied
+StateOccupied()
+~StateOccupied()
+serve()
+occupy()
+empty()
+getStateName()
+isOccupied()
+handleOrder()
StateEmpty
+StateEmpty()
+~StateEmpty()
+serve()
+occupy()
+empty()
+getStateName()
+isOccupied()
+handleOrder()



Observer Pattern:

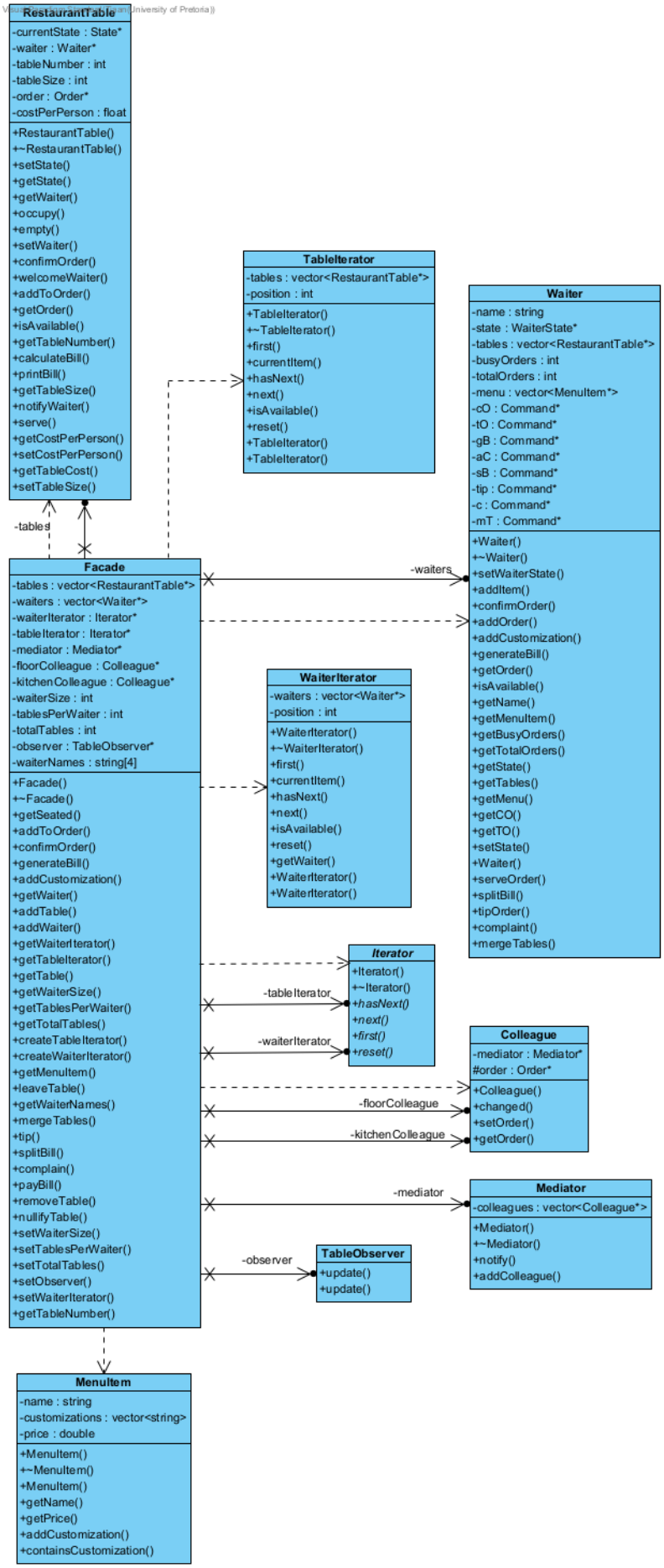
The Observer Pattern played a vital role in facilitating communication between tables and waiters. When a table wanted to place an order, the Observer Pattern was employed to notify the respective waiter. This notification served as a trigger for the waiter to initiate the order-taking process. The Observer Pattern ensured that relevant parties were informed of events, such as a table wanting to order an item. It provided a mechanism for loosely coupled communication, enhancing the responsiveness of the system to customer requests.



Facade Pattern:

While we had a clear idea of which design patterns to use for specific tasks within the restaurant floor operations, integrating them into a cohesive system presented challenges. The multitude of patterns, each serving a distinct purpose, could potentially lead to complex interactions. To streamline and simplify the interaction of these patterns, we introduced the Facade pattern. The Facade provided a unified

interface, creating a single point of entry for all the different design patterns employed in the floor side of the restaurant simulator. It allowed us to simulate the floor operations seamlessly by encapsulating the intricate interactions between patterns. The Facade pattern played a pivotal role in making the complex structure of the floor operations more manageable and coherent.


RestaurantTable
-currentState : State*
-waiter : Waiter*
-tableNumber : int
-tableSize : int
-order : Order*
-costPerPerson : float
+RestaurantTable()
+~RestaurantTable()
+setState()
+getState()
+getWaiter()
+occupy()
+empty()
+setWaiter()
+confirmOrder()
+welcomeWaiter()
+addToOrder()
+getOrder()
+isAvailable()
+getTableNumber()
+calculateBill()
+printBill()
+getTableSize()
+notifyWaiter()
+serve()
+getCostPerPerson()
+setCostPerPerson()
+getTableCost()
+setTableSize()
TableIterator
-tables : vector<RestaurantTable*>
-position : int
+TableIterator()
+~TableIterator()
+first()
+currentItem()
+hasNext()
+next()
+isAvailable()
+reset()
+TableIterator()
+TableIterator()
Waiter
-name : string
-state : WaiterState*
-tables : vector<RestaurantTable*>
-busyOrders : int
-totalOrders : int
-menu : vector<MenuItem*>
-cO : Command*
-tO : Command*
-gB : Command*
-aC : Command*
-sB : Command*
-tip : Command*
-c : Command*
-mT : Command*
+Waiter()
+~Waiter()
+setWaiterState()
+addItem()
+confirmOrder()
+addOrder()
+addCustomization()
+generateBill()
+getOrder()
+isAvailable()
+getName()
+getMenuItem()
+getBusyOrders()
+getTotalOrders()
+getState()
+getTables()
+getMenu()
+getCO()
+getTO()
+setState()
+Waiter()
+serveOrder()
+splitBill()
+tipOrder()
+complaint()
+mergeTables()
-tables
-waiters
Facade
-tables : vector<RestaurantTable*>
-waiters : vector<Waiter*>
-waiterIterator : Iterator*
-tableIterator : Iterator*
-mediator : Mediator*
-floorColleague : Colleague*
-kitchenColleague : Colleague*
-waiterSize : int
-tablesPerWaiter : int
-totalTables : int
-observer : TableObserver*
-waiterNames : string[4]
+Facade()
+~Facade()
+getSeated()
+addToOrder()
+confirmOrder()
+generateBill()
+addCustomization()
+getWaiter()
+addTable()
+addWaiter()
+getWaiterIterator()
+getTableIterator()
+getTable()
+getWaiterSize()
+getTablesPerWaiter()
+getTotalTables()
+createTableIterator()
+createWaiterIterator()
+getMenuItem()
+leaveTable()
+getWaiterNames()
+mergeTables()
+tip()
+splitBill()
+complain()
+payBill()
+removeTable()
+nullifyTable()
+setWaiterSize()
+setTablesPerWaiter()
+setTotalTables()
+setObserver()
+setWaiterIterator()
+getTableNumber()
WaiterIterator
-waiters : vector<Waiter*>
-position : int
+WaiterIterator()
+~WaiterIterator()
+first()
+currentItem()
+hasNext()
+next()
+isAvailable()
+reset()
+getWaiter()
+WaiterIterator()
+WaiterIterator()
Iterator
+Iterator()
+~Iterator()
+hasNext()
+next()
+first()
+reset()
-tableIterator
-waiterIterator
Colleague
-mediator : Mediator*
#order : Order*
+Colleague()
+changed()
+setOrder()
+getOrder()
-floorColleague
-kitchenColleague
Mediator
-colleagues : vector<Colleague*>
+Mediator()
+~Mediator()
+notify()
+addColleague()
-mediator
TableObserver
+update()
+update()
-observer
MenuItem
-name : string
-customizations : vector<string>
-price : double
+MenuItem()
+~MenuItem()
+MenuItem()
+getName()
+getPrice()
+addCustomization()
+containsCustomization()

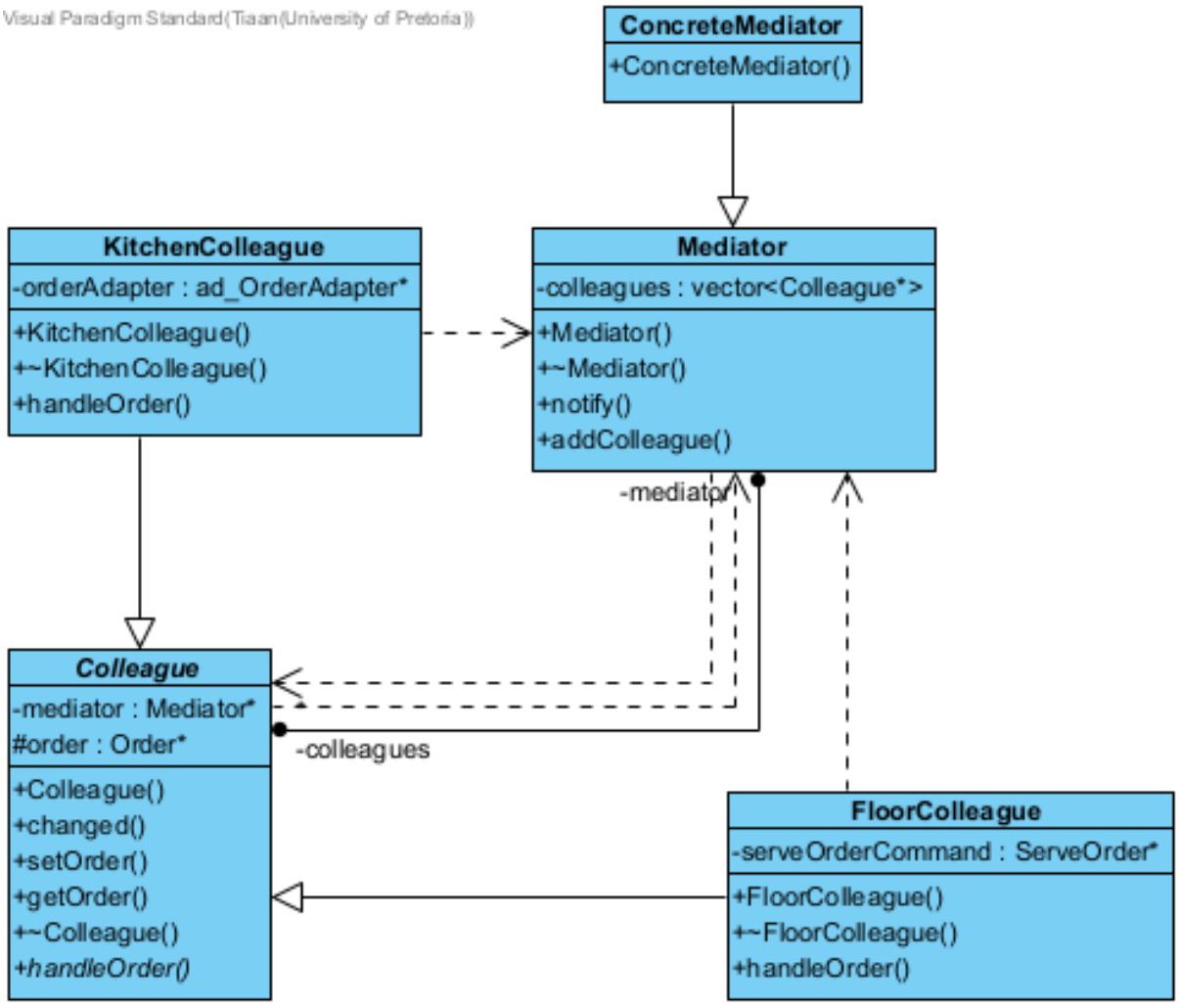
The Mediator:

While we were busy with our initial design for the restaurant, one of the challenges we encountered was the need to link the two separate sections of the restaurant: the Floor and the Kitchen. Each of these sections had distinct responsibilities, with the Floor being responsible for taking orders from customers and managing tables and waiters, while the Kitchen focused on order preparation and cooking.

The problem that needed to be addressed was how to ensure the smooth flow of orders, from the moment a customer placed an order on the Floor to its preparation in the Kitchen and ultimately its delivery back to the customers. To tackle this issue, we introduced the Mediator Pattern into our design.

The Mediator Pattern acted as a pivotal solution to the problem, serving as the bridge that connected the FloorColleague (representing the Floor) and the KitchenColleague (representing the Kitchen). By mediating the interactions and communication between these two distinct parts of the restaurant, the Mediator Pattern enabled the FloorColleague to pass orders to the KitchenColleague for preparation. Once an order was complete in the kitchen, it was efficiently transmitted back to the FloorColleague for delivery to the customers.

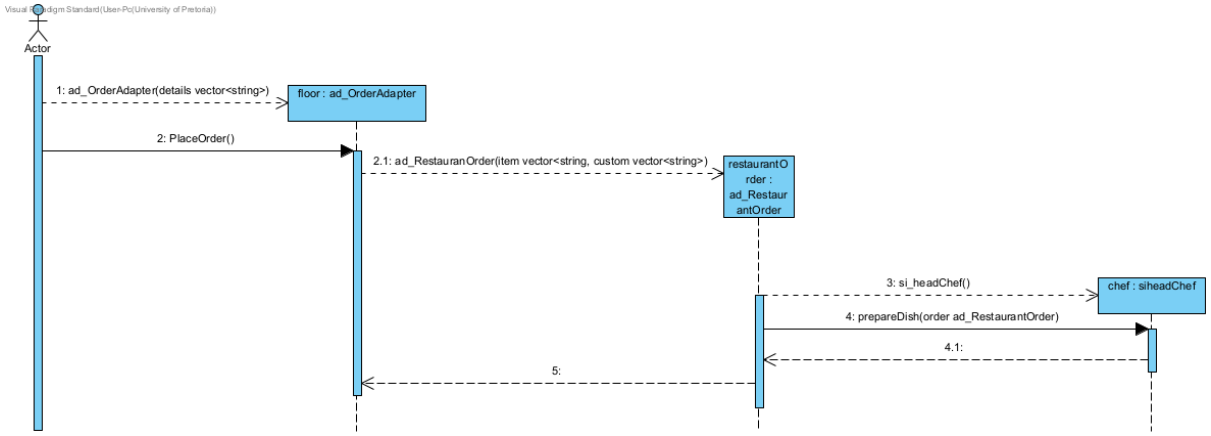
This implementation of the Mediator Pattern not only addressed the challenge of linking the two separate sections but also ensured that orders could be processed and completed in a coordinated and organized manner. It played a critical role in streamlining the order flow within the restaurant simulator.



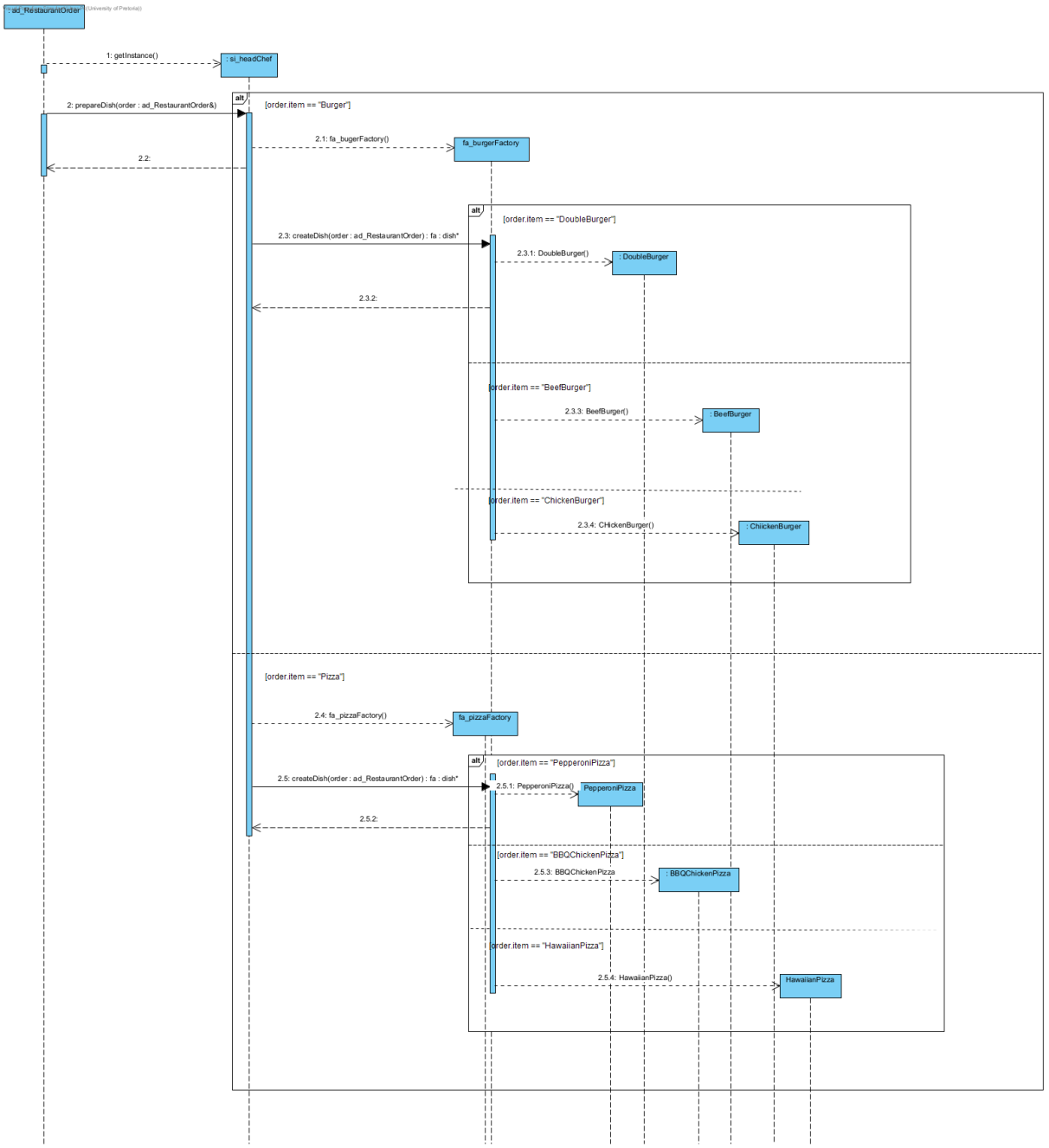
Diagrams

Sequence Diagrams

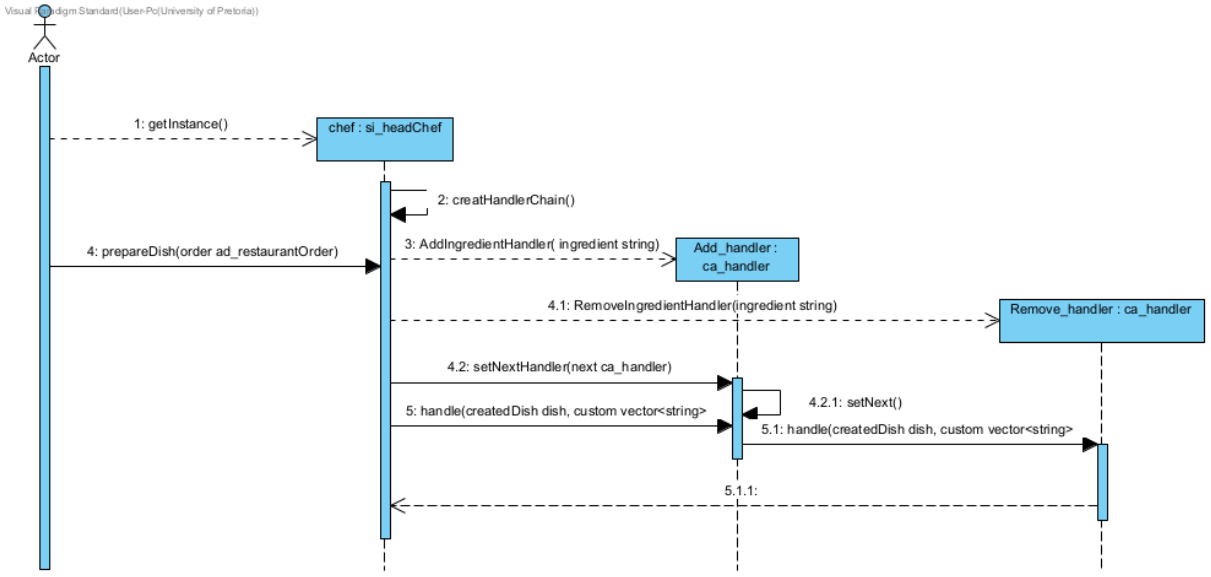
Interaction between Adapter and head chef



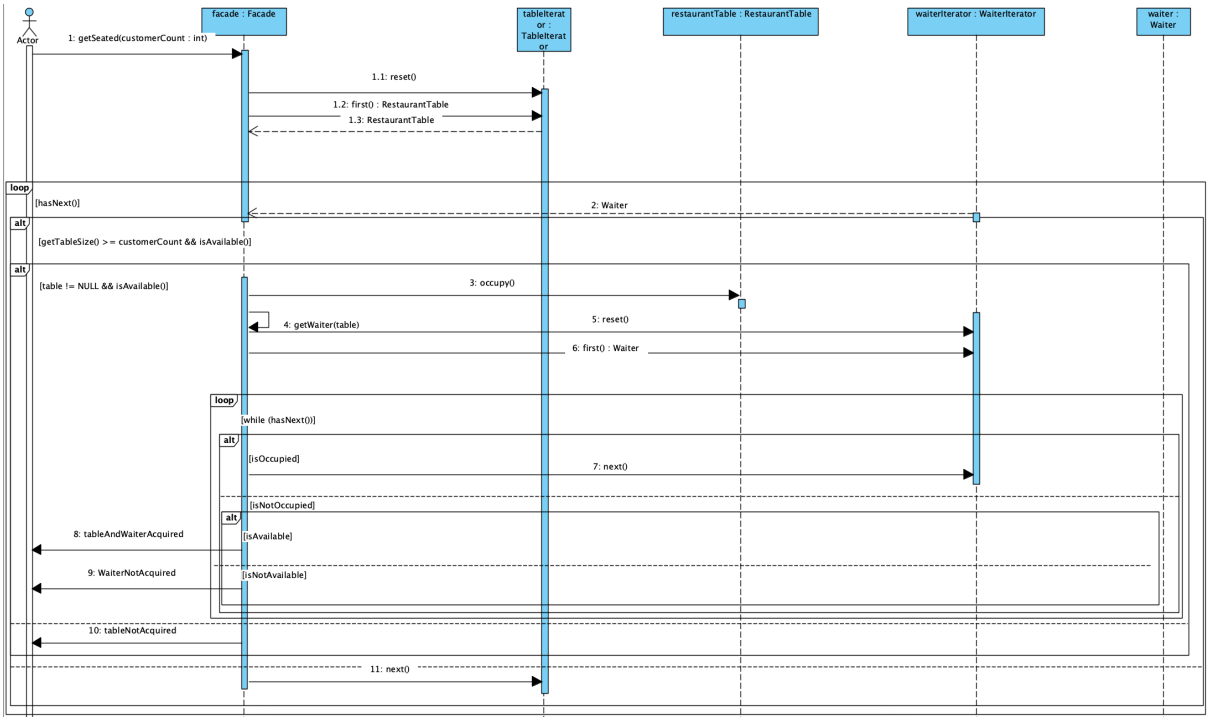
Interaction between head chef and Factory method



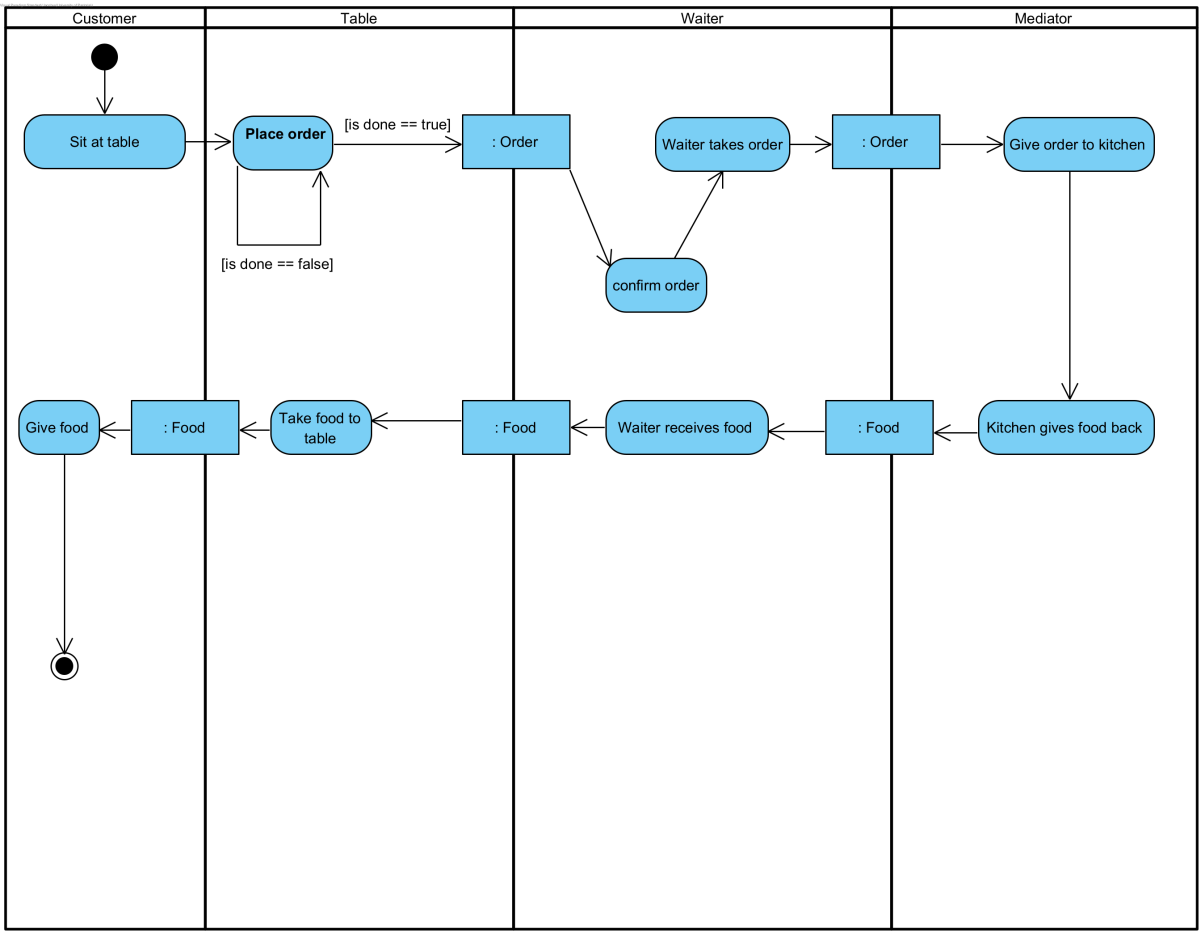
Interaction between head chef and chain of command



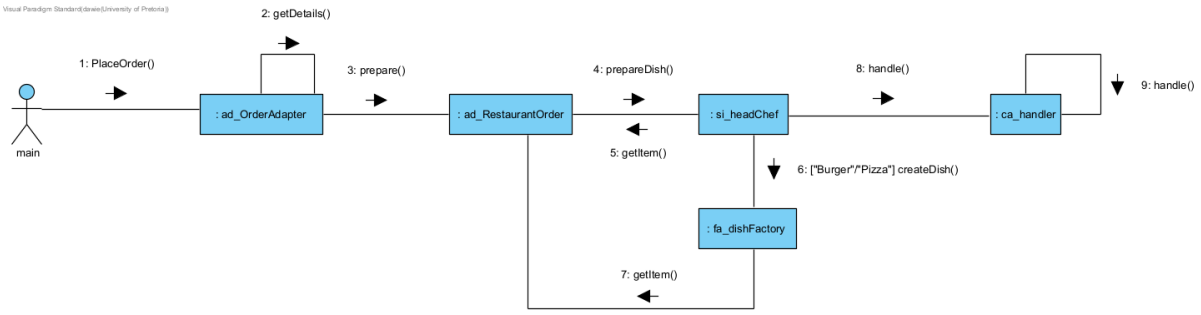
getSeated and getWaiter Sequence Diagram



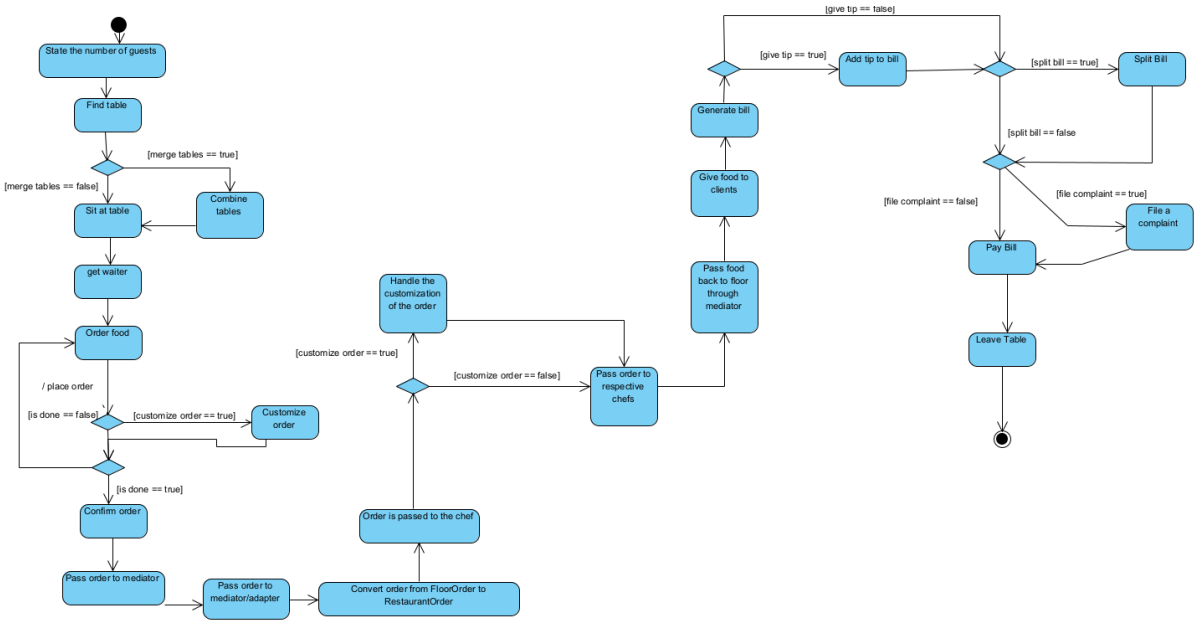
Activity Diagram - Floor



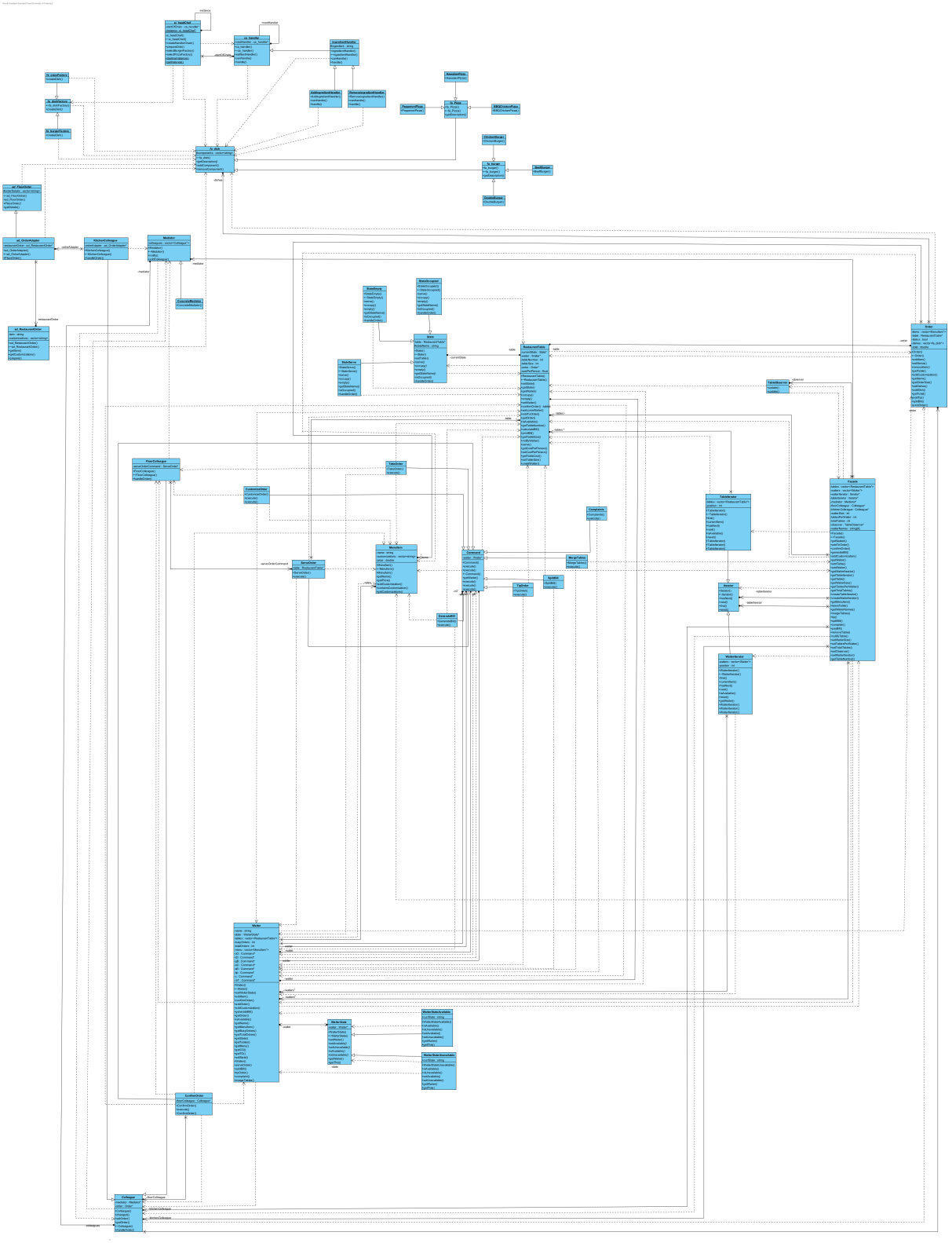
Communication Diagram



State Diagram



Full Class Diagram



Team Members

Members:

Tiaan Pouwels (u21675229@tuks.co.za)

Dawie Reyneke (u21438112@tuks.co.za)

Xavier Reynolds (u20526254@tuks.co.za)

Keith Homan (u21473103@tuks.co.za)

Jacobus Smit (u21489476@tuks.co.za)

Iwan de Jong (u22498037@tuks.co.za)