7/11/2023

COS 214 PROJECT

Team “Concrete Pass”:

Matthew Els	u21715191
Dominique da Silva	u21629944
Tlhalefo Dikolomela	u21507792
Tinashe Austin	u21564176
Christopher Katranas	u19154853

Restaurant Simulator

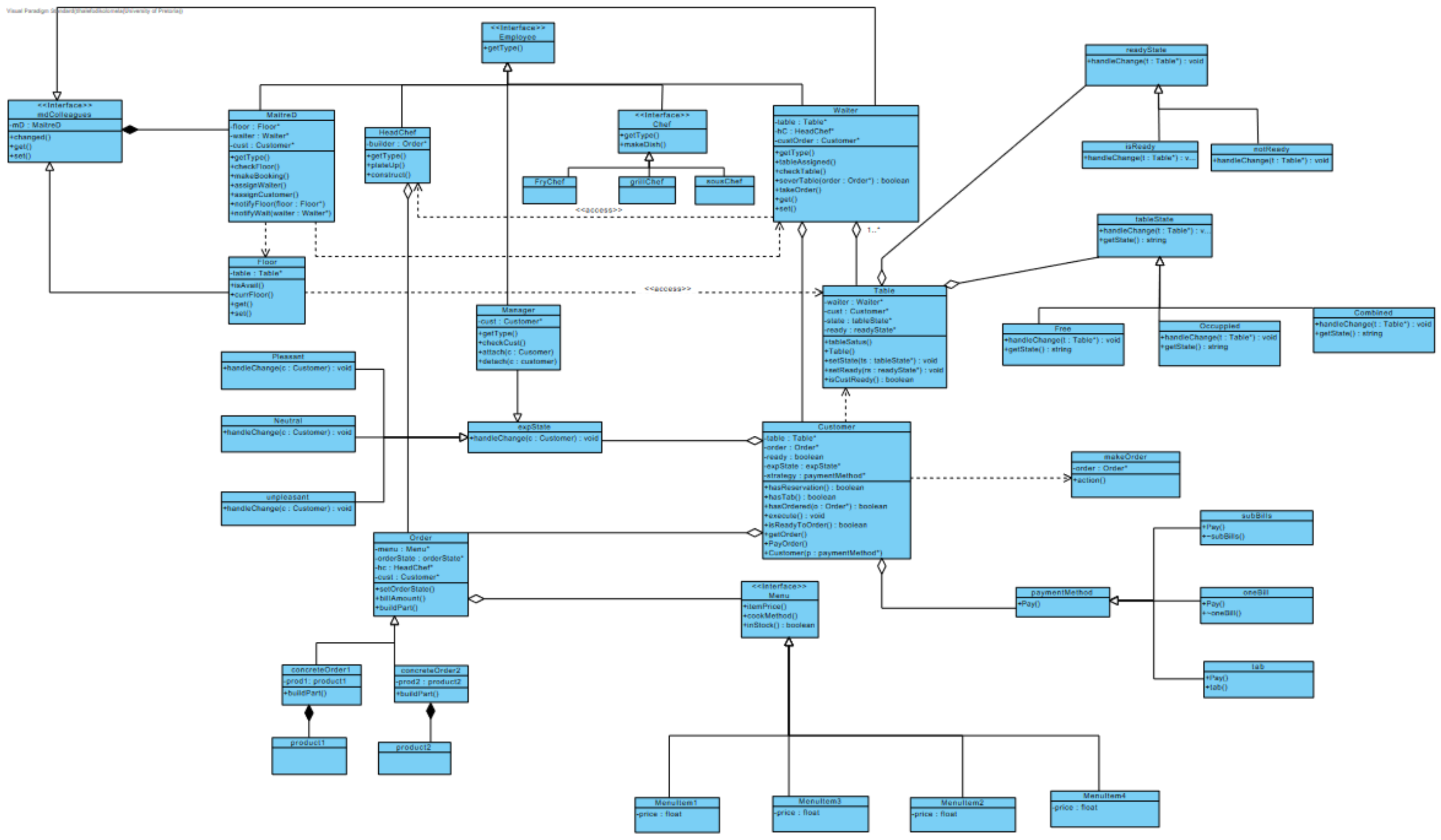
CONTENTS

TASK 1: PRACTICAL ASSIGNMENT

This was our initial and partial design, after a discussion:

- Mediator
    - To use for the MaitreD to allocate tables to our restaurant’s customers.
- Chain of Responsibility
    - To be able to pass the order that was made from one class to another to be able get, assemble, plate and serve the meal. This would be the main focus of the system as it would dictate the flow of the program.
- Command
    - For the waiter to be able to notify and queue the orders made for the kitchen to be able to produce the meal.
- Template Method
    - Wanting to print out a customer’s bill.
- Strategy
    - To be able to split the bill between multiple customers at a table.
- Façade
    - To encapsulate the system and give the client a single point of entry, even though would still be able to interact with subsystems.
- Decorator
    - To be able to add to a meal, such as additional foods or sides to the meal.
- Factory Method
    - We wanted to produce a menu and a meal using the factory method, instead of building individual meals.
- Momento
    - To store the state of the bill. The customer would be able to add to their order and once they have been completely served, the bill’s state would be recalled and would be settled.
- State
    - Wanted the customer’s satisfaction to ensure that various different subclasses act on the state of the customer. Such as any dissatisfaction would ensure that manager would be called.

This was our partial and initial design represented by the UML diagram that was created using Visual Paradigm.



TASK 2: DESIGN

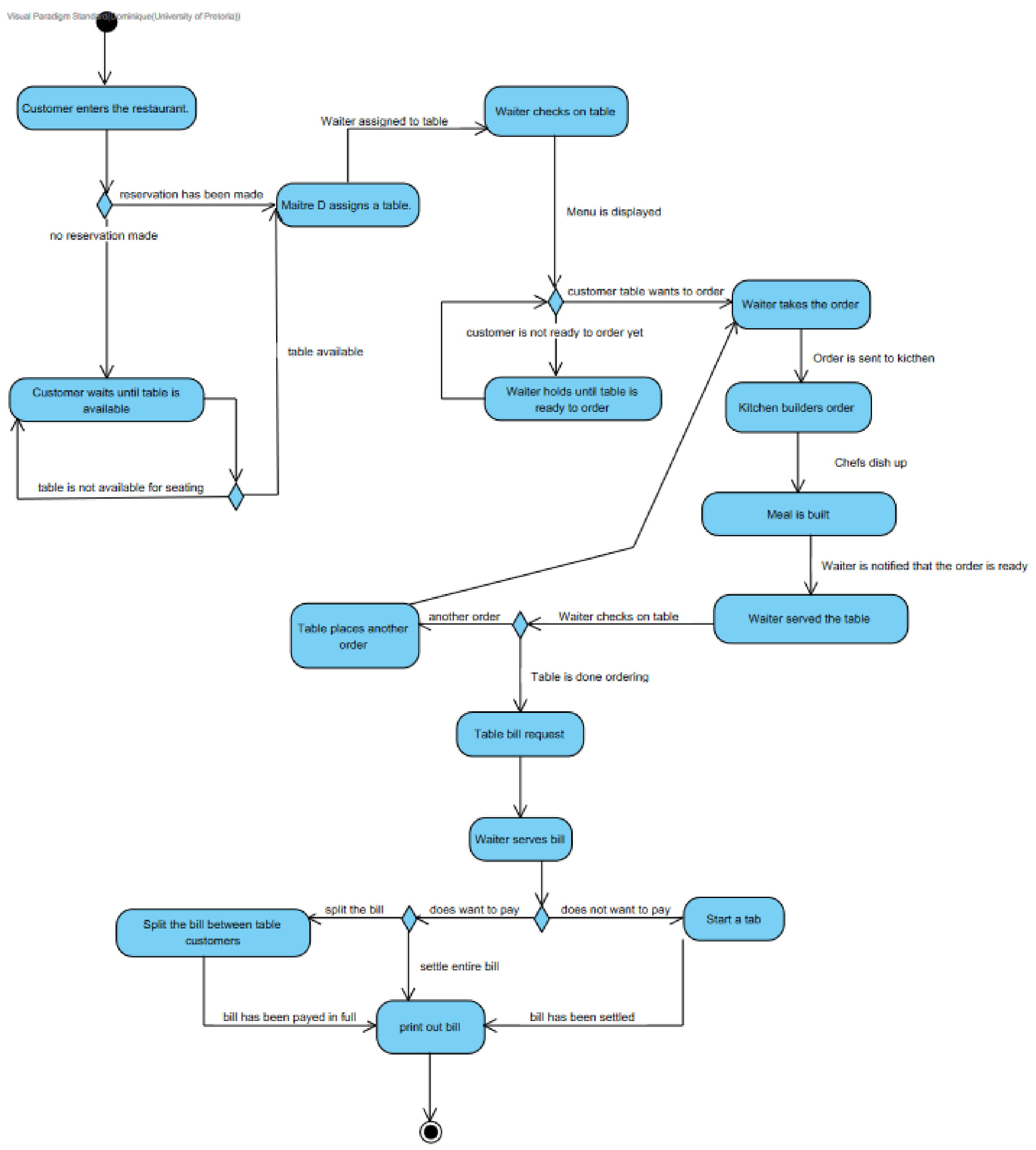
2.1 System Requirements

We are required to design and implement a restaurant simulator. The end goal of the system is the production and sale of food to customers, however there an ensemble of chaotic processes to consider.

The focus is primarily two areas namely:

- The floor.
    - The number of available tables.
    - Seating a group of customers based on whether they have made a reservation or not.
    - Having to combine tables to accommodate a large group of customers and restoring the tables to their previous state.
    - Once the group of customers has been assigned a table, the Maitre D has to assign a waiter to assist the table.
    - Should the table not be ready to place an order, the waiter should return later.
    - The table should be able to customize their order and give it to the waiter to be processed and prepared by the kitchen.
    - After the table has been served, they should be able to tip based on their satisfaction of the service.
    - The bill can either be settled (the full amount by one person, or can be distributed evenly among the customers at the particular table) or the customers can have a tab that can be settled at a later date.
- The kitchen.
    - The waiter commands the kitchen to start the preparation of an order.
    - Once the kitchen has been handed the order, the chefs can start building a meal from scratch.
    - Once the building process is done, each meal should be sent to the head chef for plating.
    - The head chef is responsible for the finishing garnishes of the order before it is sent back to the waiter to be served to the customer.

2.2 Activity Diagrams


Visual Paradigm Standard(Dominique(University of Pretoria))
Customer enters the restaurant.
reservation has been made
no reservation made
Maitre D assigns a table.
Waiter assigned to table
Waiter checks on table
Menu is displayed
customer table wants to order
customer is not ready to order yet
Waiter holds until table is ready to order
Waiter takes the order
Order is sent to kicthen
Kitchen builders order
Chefs dish up
Meal is built
Waiter is notified that the order is ready
Waiter served the table
Waiter checks on table
another order
Table places another order
Table is done ordering
Table bill request
Waiter serves bill
does want to pay
does not want to pay
Start a tab
split the bill
Split the bill between table customers
settle entire bill
bill has been payed in full
print out bill
bill has been settled
table available
Customer waits until table is available
table is not available for seating

2.3 & 2.4 Identification of patterns

A combination of 10 gang of four design patterns were used to address the system requirements listed above.

1. Composite
2. Strategy
3. State
4. Observer

5. Builder
6. Chain of Responsibility
7. Proxy
8. Command
9. Template Method
10. Facade

The Composite pattern is used to model the compositions of tables to accommodate larger groups of customers.

The Strategy pattern is used to interchange the algorithm based on how the customer is going to be seated.

The State pattern is used to be able to transition based on the internal state of the availability of the table. It is also used to model the transition based on whether the table is ready to place an order with the waiter or not.

The Observer pattern is used to model the dependency between the waiter and the customer, the waiter is notified when a table is ready to place an order.

The Builder pattern is used to model the process of building a meal of the order that was placed, from scratch and return the finished meals.

The Chain of Responsibility pattern is used to be able to pass the order request from the waiter, to the kitchen for building, the head chef and then back to the waiter to be able to serve the table.

The Proxy pattern is used to protect the customer, such that we use a proxy customer and the kitchen is not able to change any functionality of the customer when the order is being taken.

The Command pattern is used is used to model the waiter passing the order request to the kitchen.

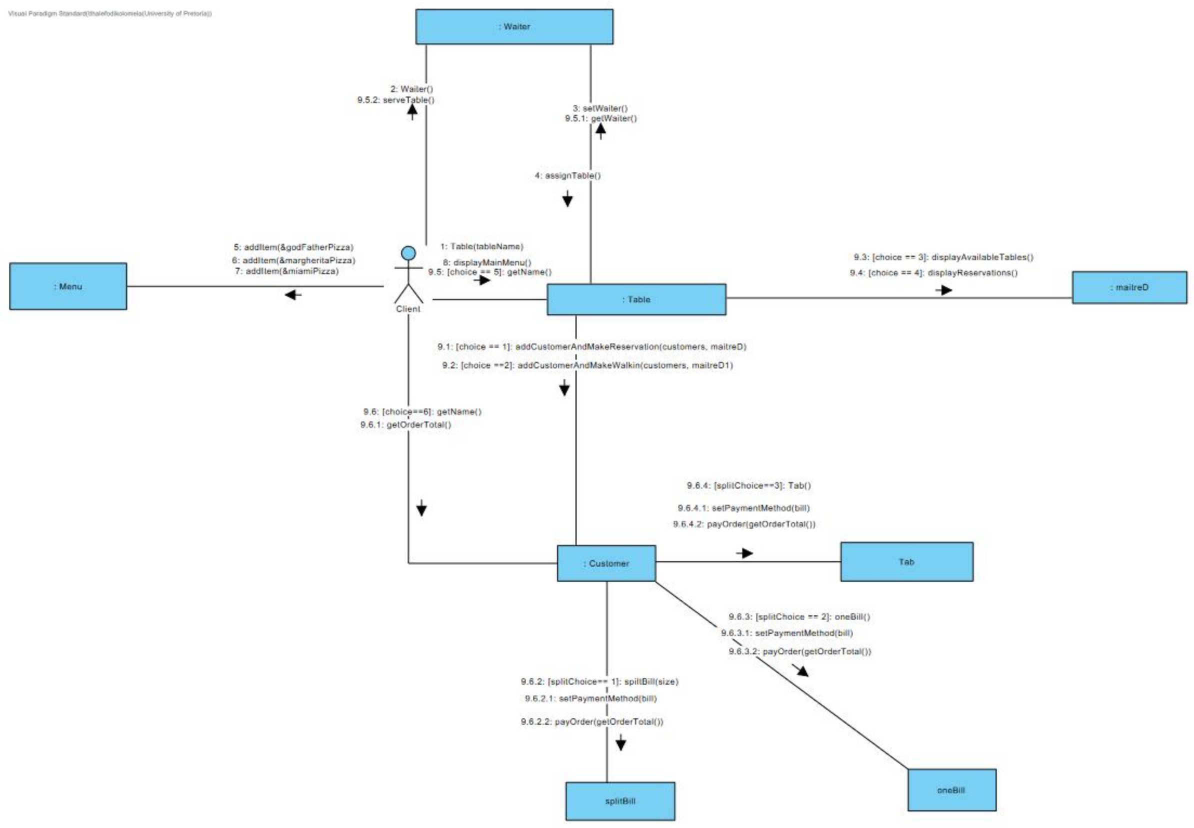
The Template Method pattern is used to model the method to be able to pay the bill. The variations of the algorithm of payment does not matter (whether a single customer pays in full, the bill is split or a tab is being payed), the bill just has to be settled.

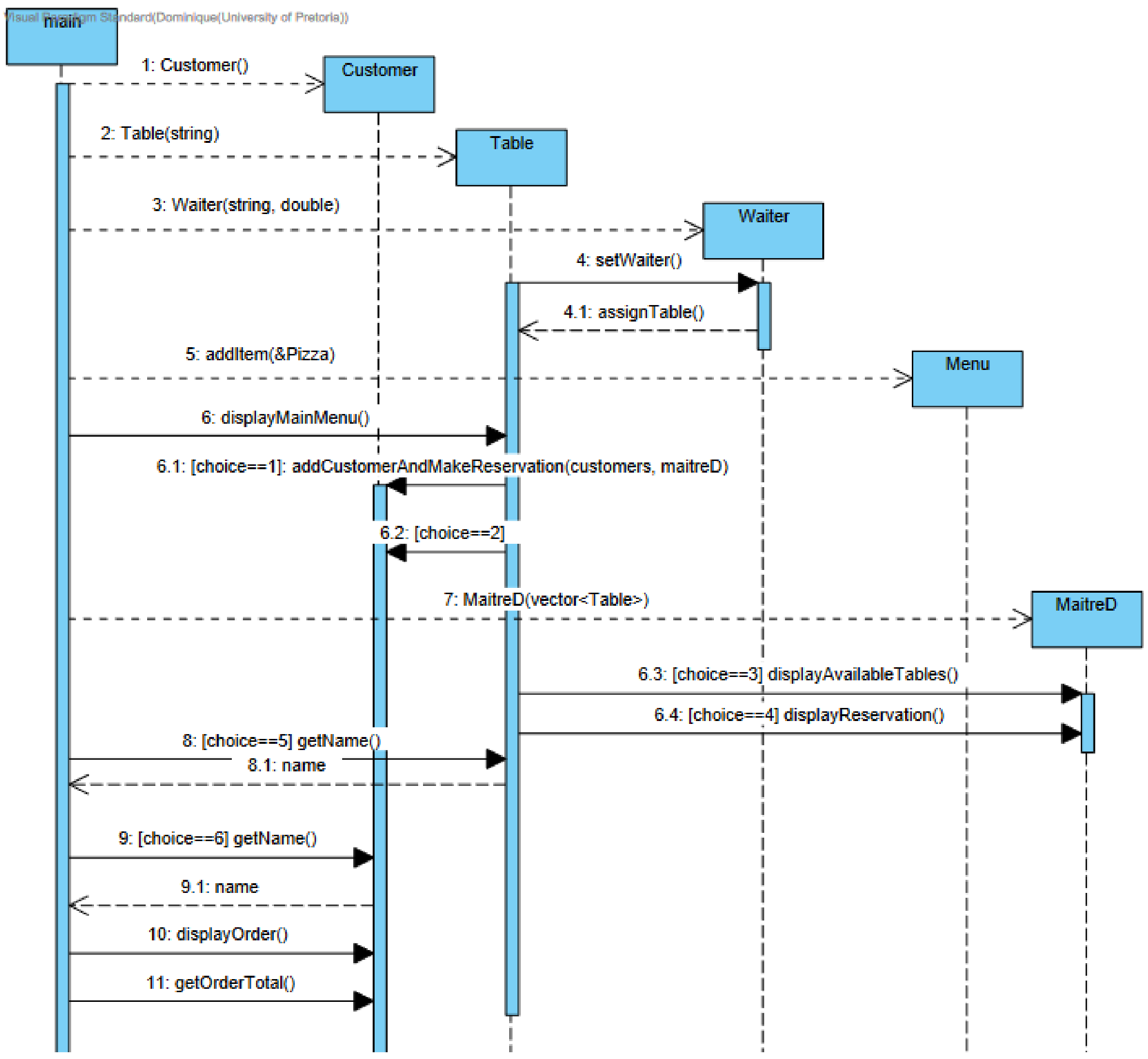
The Façade pattern is used to model the front interface for the set of interfaces that has been implemented. This just makes the overall system easier to use.

2.5 UML Class Diagram for system

* The Diagram is large. Find attached at the end of the document.

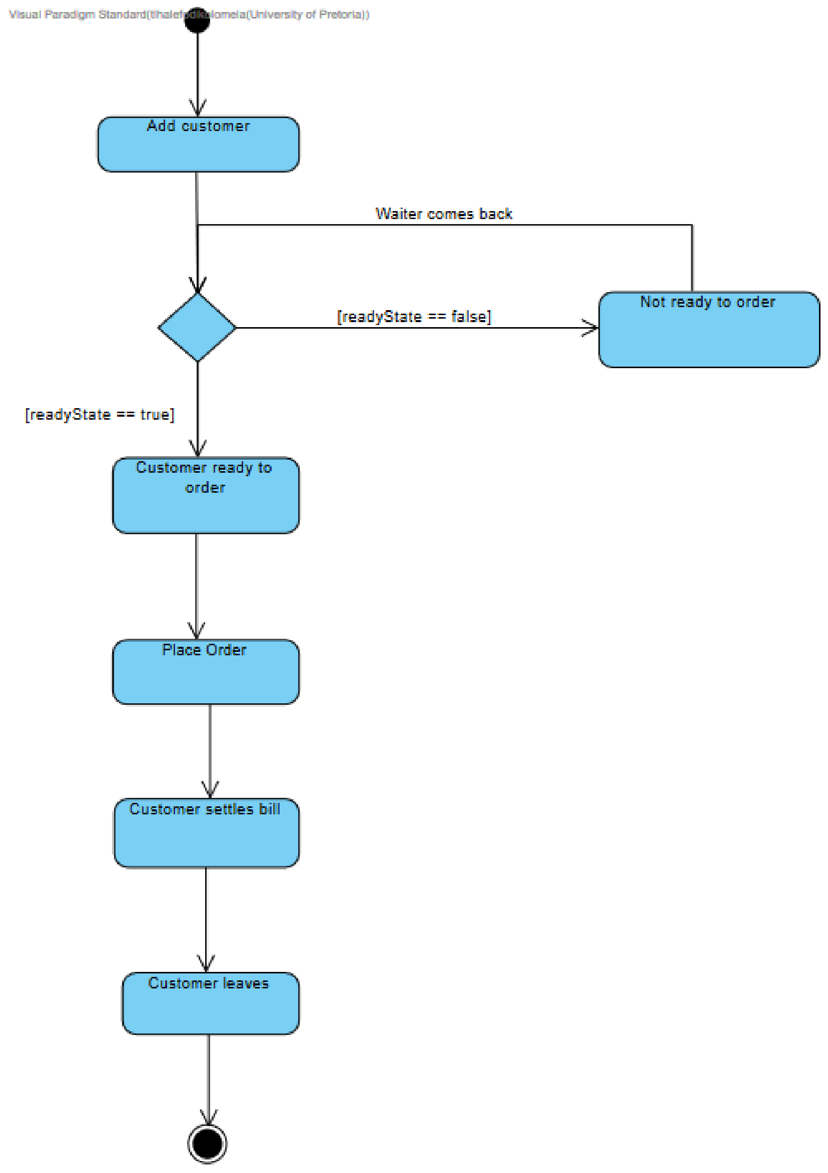
2.6 Sequence & Communication Diagrams


Visual Paradigm Standard(thalefodikolomela(University of Pretoria))
: Waiter
2: Waiter()
9.5.2: serveTable()
3: setWaiter()
9.5.1: getWaiter()
4: assignTable()
5: addItem(&godFatherPizza)
6: addItem(&margheritaPizza)
7: addItem(&miamiPizza)
: Menu
1: Table(tableName)
8: displayMainMenu()
9.5: [choice == 5]: getName()
Client
: Table
9.3: [choice == 3]: displayAvailableTables()
9.4: [choice == 4]: displayReservations()
: maitreD
9.1: [choice == 1]: addCustomerAndMakeReservation(customers, maitreD)
9.2: [choice ==2]: addCustomerAndMakeWalkin(customers, maitreD1)
9.6: [choice==6]: getName()
9.6.1: getOrderTotal()
9.6.4: [splitChoice==3]: Tab()
9.6.4.1: setPaymentMethod(bill)
9.6.4.2: payOrder(getOrderTotal())
: Customer
Tab
9.6.3: [splitChoice == 2]: oneBill()
9.6.3.1: setPaymentMethod(bill)
9.6.3.2: payOrder(getOrderTotal())
9.6.2: [splitChoice== 1]: spiltBill(size)
9.6.2.1: setPaymentMethod(bill)
9.6.2.2: payOrder(getOrderTotal())
splitBill
oneBill

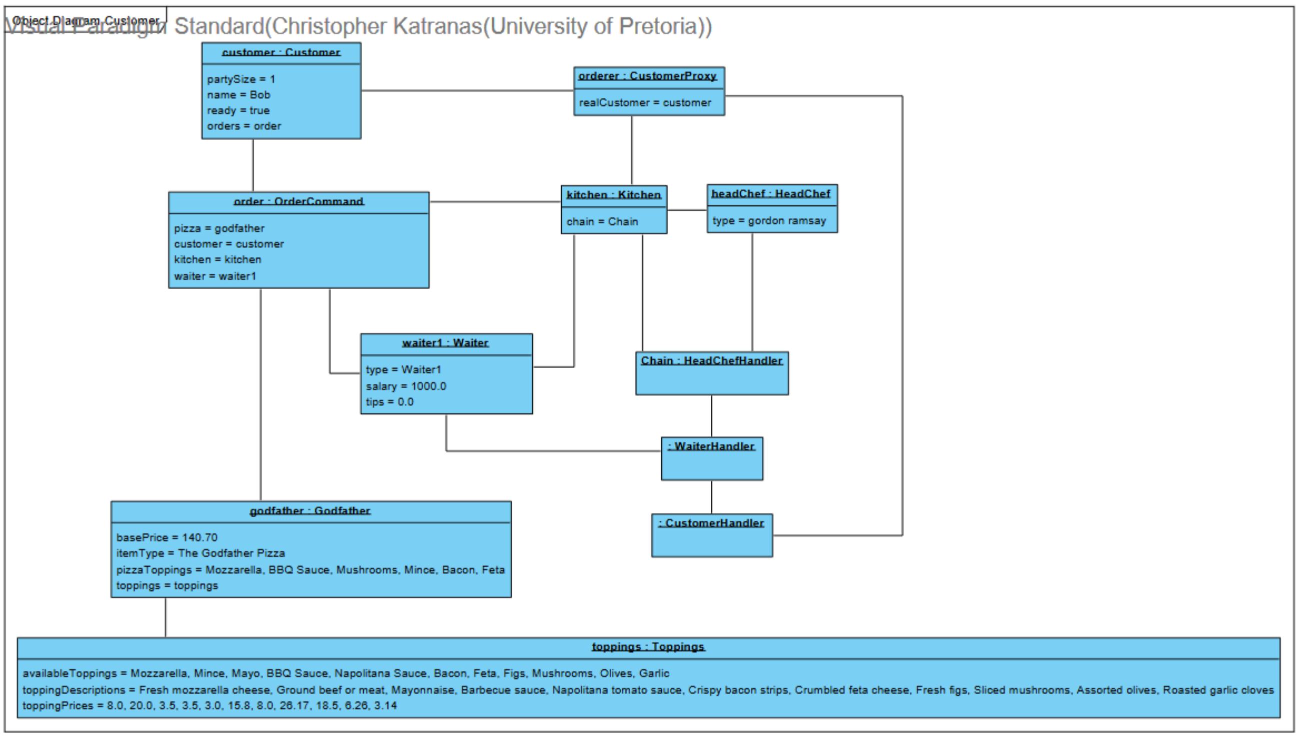


Visual Paradigm Standard(Dominique(University of Pretoria))
main
Customer
Table
Waiter
Menu
MaitreD
1: Customer()
2: Table(string)
3: Waiter(string, double)
4: setWaiter()
4.1: assignTable()
5: addItem(&Pizza)
6: displayMainMenu()
6.1: [choice==1]: addCustomerAndMakeReservation(customers, maitreD)
6.2: [choice==2]
7: MaitreD(vector<Table>)
6.3: [choice==3] displayAvailableTables()
6.4: [choice==4] displayReservation()
8: [choice==5] getName()
8.1: name
9: [choice==6] getName()
9.1: name
10: displayOrder()
11: getOrderTotal()

2.7 State Diagrams



2.8 Object Diagrams


Object Diagram Customer
Visual Paradigm Standard(Christopher Katranas(University of Pretoria))
customer : Customer
partySize = 1
name = Bob
ready = true
orders = order
orderer : CustomerProxy
realCustomer = customer
order : OrderCommand
pizza = godfather
customer = customer
kitchen = kitchen
waiter = waiter1
kitchen : Kitchen
chain = Chain
headChef : HeadChef
type = gordon ramsay
waiter1 : Waiter
type = Waiter1
salary = 1000.0
tips = 0.0
Chain : HeadChefHandler
: WaiterHandler
: CustomerHandler
godfather : Godfather
basePrice = 140.70
itemType = The Godfather Pizza
pizzaToppings = Mozzarella, BBQ Sauce, Mushrooms, Mince, Bacon, Feta
toppings = toppings
toppings : Toppings
availableToppings = Mozzarella, Mince, Mayo, BBQ Sauce, Napolitana Sauce, Bacon, Feta, Figs, Mushrooms, Olives, Garlic
toppingDescriptions = Fresh mozzarella cheese, Ground beef or meat, Mayonnaise, Barbecue sauce, Napolitana tomato sauce, Crispy bacon strips, Crumbled feta cheese, Fresh figs, Sliced mushrooms, Assorted olives, Roasted garlic cloves
toppingPrices = 8.0, 20.0, 3.5, 3.5, 3.0, 15.8, 8.0, 26.17, 18.5, 6.26, 3.14

TASK 3: IMPLEMENTATION

3.3 Coding Standards

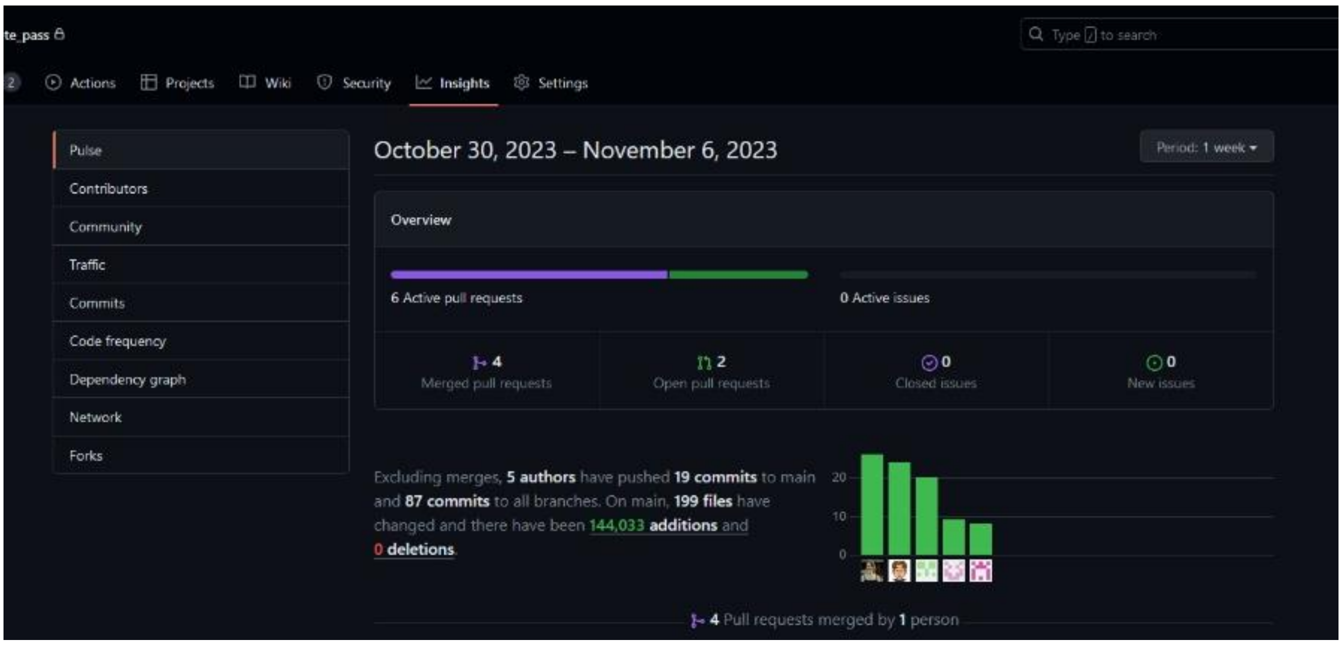
We have followed C++ documentation standards. Separate header and implementation files were used (.h and .cpp files).
Comments were added using the // in our source files.

Descriptive variable have been used as well as the correct indentation.

The makefile that has been included should also be able to work should the system be expanded without having to edit this file in any way.

3.4 GitHub commits

Screenshot of the commits of each member:



October 30, 2023 – November 6, 2023

6 Active pull requests

0 Active issues

4 Merged pull requests

2 Open pull requests

0 Closed issues

0 New issues

Excluding merges, **5 authors** have pushed **19 commits** to main and **87 commits** to all branches. On main, **199 files** have changed and there have been 144,033 **additions** and 0 **deletions**.

4 Pull requests merged by 1 person

3.5 Doxygen comments

Our doxygen documentation can be found in our archive.

Can also be viewed at:

https://github.com/MatthewEls/COS214-Concrete_pass.git

TASK 4: REPORT

4.1 Research & References

Similar project were found online and was used as reference as to what patterns to consider in our system design (as referenced below). In addition, it did serve as a guideline for us to rather pursue a simulation of the environment, rather than a game-like implementation.

GitHub. (n.d.). Grokking-OOD/object-oriented-design-case-studies/design-a-restaurant-management-system.md at master · wyaadarsh/Grokking-OOD. [online] Available at: https://github.com/wyaadarsh/Grokking-OOD/blob/master/object-oriented-design-case-studies/design-a-restaurant-management-system.md#class-diagram [Accessed 1 Nov. 2023].

Shahid, M. (2023). Restaurant-Management-System. [online] GitHub. Available at: https://github.com/mabbia706/Restaurant-Management-System/tree/master [Accessed 1 Nov. 2023].

www.c-sharpcorner.com. (n.d.). Food Delivery Application Using Design Patterns. [online] Available at: https://www.c-sharpcorner.com/article/food-delivery-application-using-with-design-patterns/.

Doyle, W. (2023). Design Patterns Explained with Food □. [online] GitHub. Available at: https://github.com/wesdoyle/design-patterns-explained-with-food/tree/main [Accessed 1 Nov. 2023].

Kothari, A. (2018). Java Builder Design Pattern Example - Java Code Geeks. [online] Examples Java Code Geeks. Available at: https://examples.javacodegeeks.com/java-development/core-java/java-builder-design-pattern-example/ [Accessed 1 Nov. 2023].

GitHub. (n.d.). COS214-Project/Docs/DefinitiveStyleGuide.md at main · Multimedia-Overachievers/COS214-Project. [online] Available at: https://github.com/Multimedia-Overachievers/COS214-Project/blob/main/Docs/DefinitiveStyleGuide.md [Accessed 3 Nov. 2023].

262588213843476 (n.d.). CodingGuidelines.md. [online] Gist. Available at: https://gist.github.com/earonesty/ccee25a56be7adeb5f670cf44e5fa479 [Accessed 3 Nov. 2023].

Software Engineering Stack Exchange. (n.d.). Creating a coding standards document. [online] Available at: https://softwareengineering.stackexchange.com/questions/196706/creating-a-coding-standards-document [Accessed 3 Nov. 2023].

webmaster (2016). Embedded C Coding Standard. [online] Barr Group. Available at: https://barrgroup.com/Embedded-Systems/Books/Embedded-C-Coding-Standard.

4.2 & 4.4 Design decisions & assumptions

The idea for our implementation was to implement a simulation of an actual restaurant, rather than the restaurant tycoon.

This was purely because it was easier to conceptualize, as we have experienced the mundane workflow of a restaurant, compared to a game.

We assumed that individual customers do not matter to the system. A collection of customers that are being seated at a table, will be treated as a singular object. The table will be served by the waiter and not individual customers at the table. This is due to minimize the number of unnecessary “instances” of customers, as 1 Table class represents 6 Customers.

When a customer arrives at the restaurant, they can choose either to make a reservation or to walk in. If they choose to make a reservation, the MaitreD context uses the reservation strategy (ReservationStrategy) to find and allocate a suitable table for the reservation. Depending on the strategy, the MaitreD collaborates with the composite structure of tables to search for available tables.Once an available table is found, the Table context transitions its state to "not available," marking it as reserved. The customer's reservation is recorded.When the reservation time comes, the MaitreD seats the customer, and the table state transitions back to "available." when a customer leaves the table.

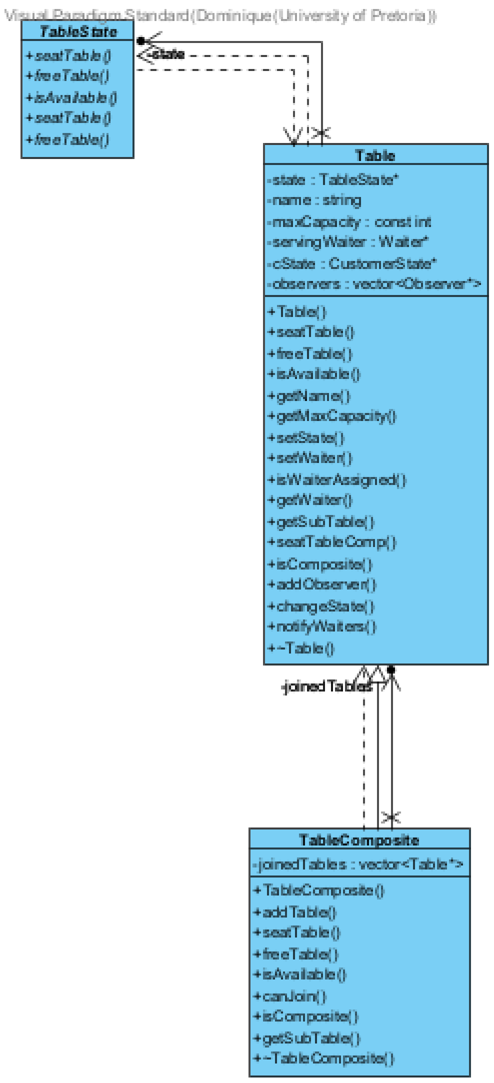
When splitting the bill, we assumed that the value of the bill would be split evenly between all members of the particular table.

We assumed that, should a tab be the method of payment, that a tab cannot be infinite. The system set a max limit and after that the tab has to be settled.

4.3 & 4.5 Write up of patterns with diagrams

1. **Composite**: Composes objects into tree structures to represent part-whole hierarchies. Individual and compositions of objects are treated uniformly.

> Our table hierarchy makes use of the **Composite pattern** in order to be able to merge and decouple tables from one another for bigger groups of customers that need to be seated together.
>
> The Component participant is the TableState. The Leaf participant is the Table, since the class consists of a singular table. TableComposite is our Composite participant as this class indicates when two or more individual tables have been joined. The TableComposite has a vector of tables that are connected, or that have been theoretically merged.
>
> We do this by using the methods addTable(Table*) and the freeTable() method.

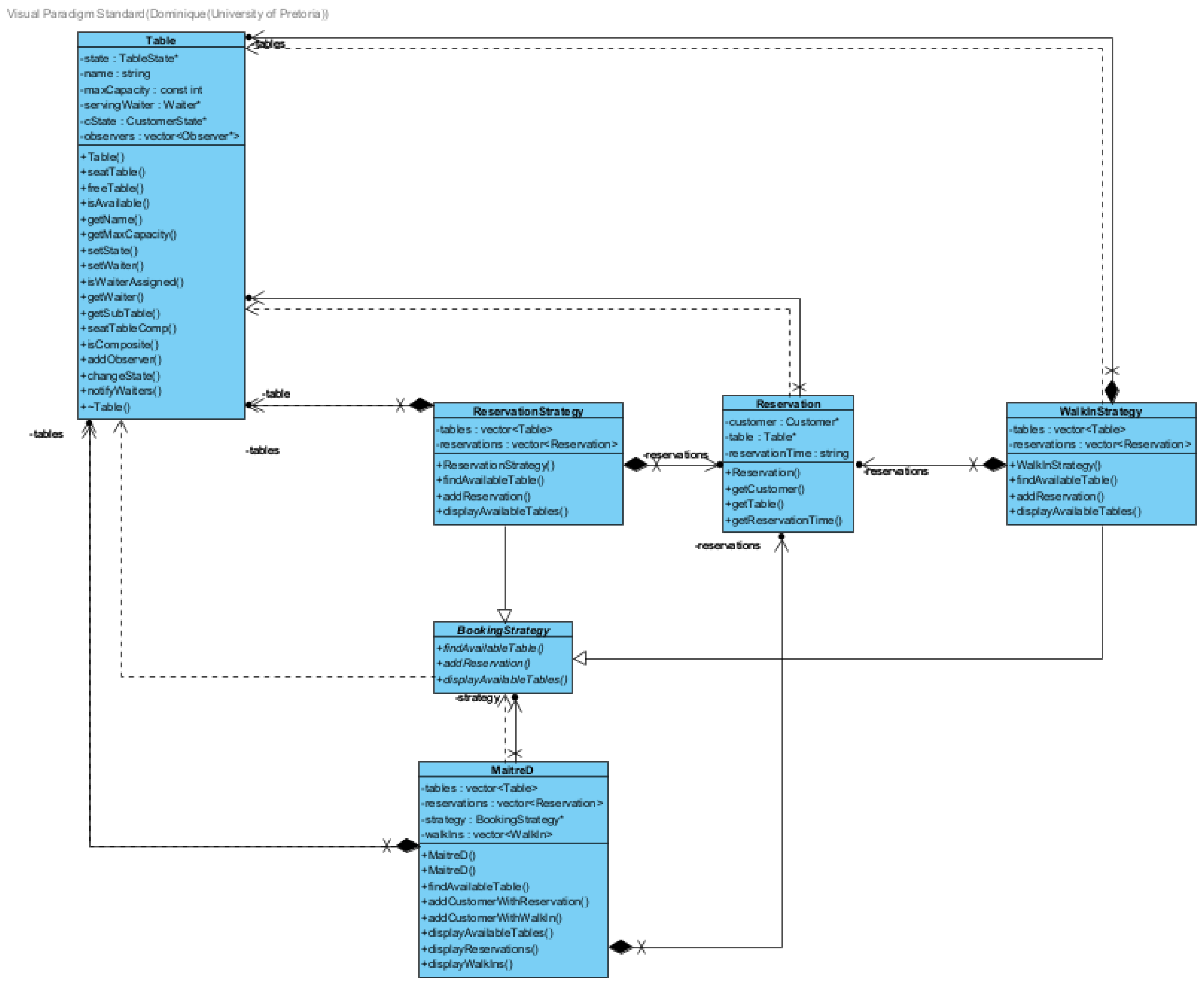


2. **Strategy**: Define a family of interchangeable algorithms, encapsulate each one, and make them interchangeable.

Our MaitreD class will make use of the **Strategy pattern**, as it can make used of either strategy to be able to seat the customers without having to know the details about whether the customer is a walk in customer, or has made a reservation, and how they are managed.

Thus the Context participant is MaitreD which holds a pointer to an instance of the Strategy participant which is the booking strategy.

The ConcreteStrategy participants are ReservationStrategy and WalkInStrategy.

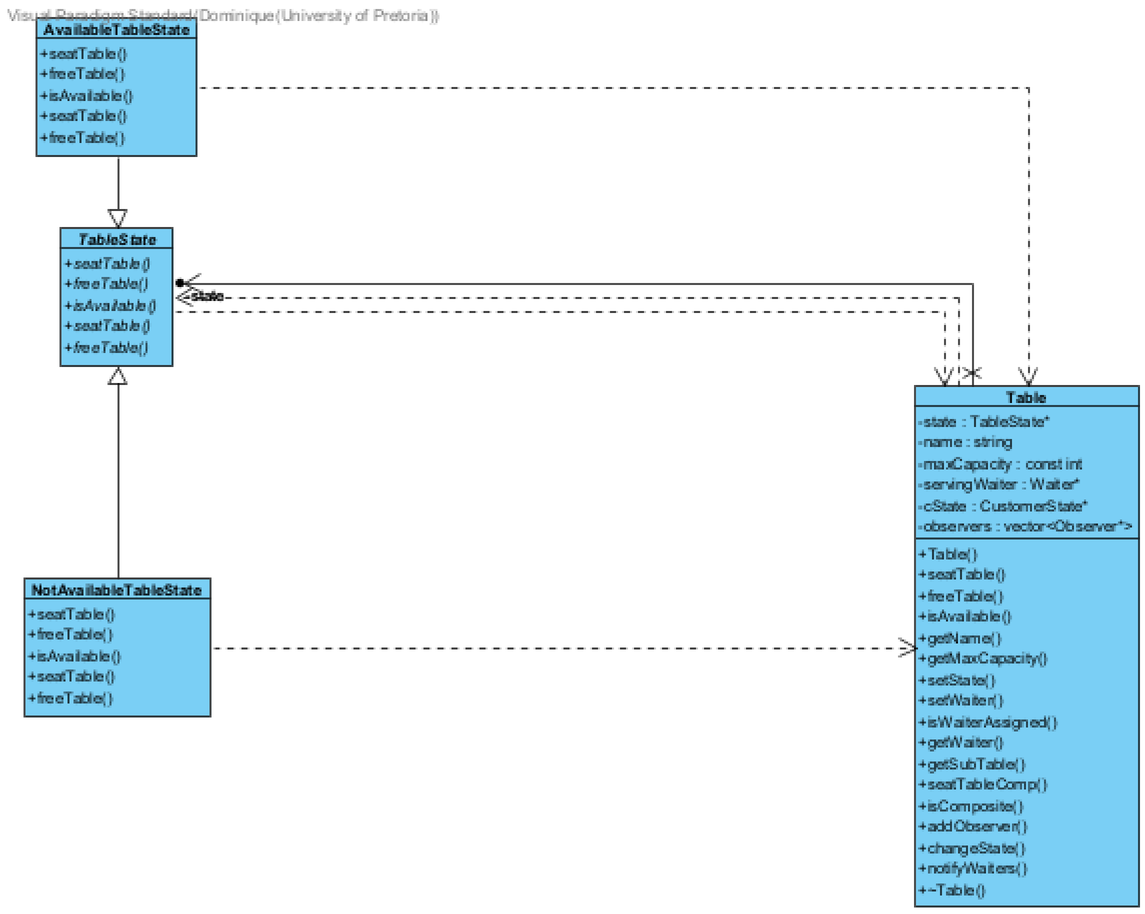


3. **State:** Allow an object to alter its behaviour when it's internal state changes. The object appears to change its class, enabling it to adapt to different situations, and effectively encapsulate the state-specific behaviour into separate state objects.

The **State pattern** is used to change the internal state of the table once the table is available or when it has been assigned. The Table class is the Context participant, which has the reference of the current state of the table, and delegates behaviour to the State object, which is called the TableState.

The derived ConcreteState participants are the AvailableTableState and the NotAvailableTableState.

Table uses the TableState objects to perform seatTable and freeTable operations, which are context specific. The Table class transitions between available and not available states.



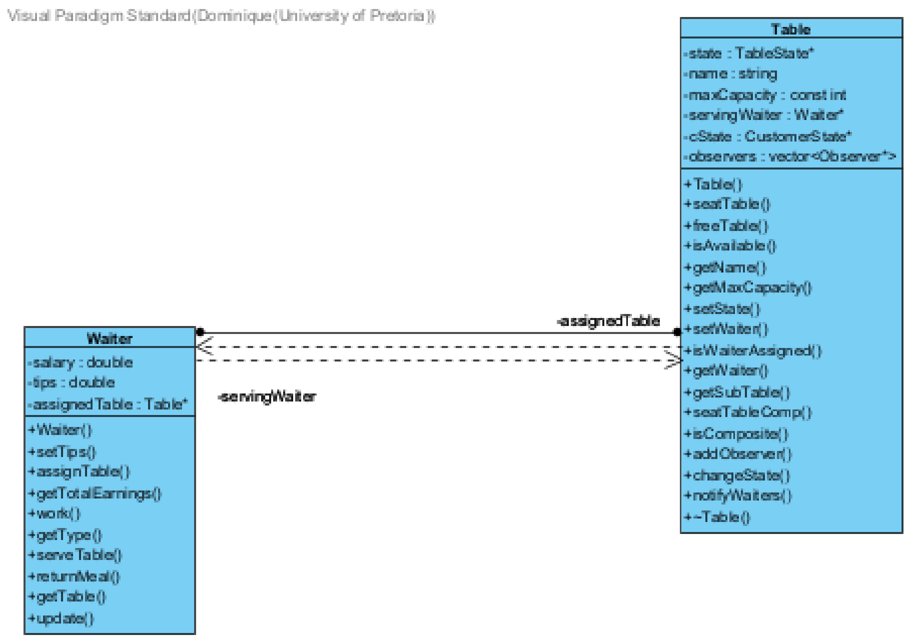
4. **Observer:** Define a one-to-many dependency between objects so that when one object changes its state, all its dependents are notified and updated automatically.

The **Observer pattern** is used with the State pattern in order to observe the state of the table of customers and whether they are ready to place an order with the waiter.

Observer participant is the Observer class which acts as an interface for the Waiter.

The ConcreteObserver participant is the Waiter, as the waiter observes the state of the Subject participant, which is the Table.

The Table maintains a list of waiters to notify once they are ready to order.

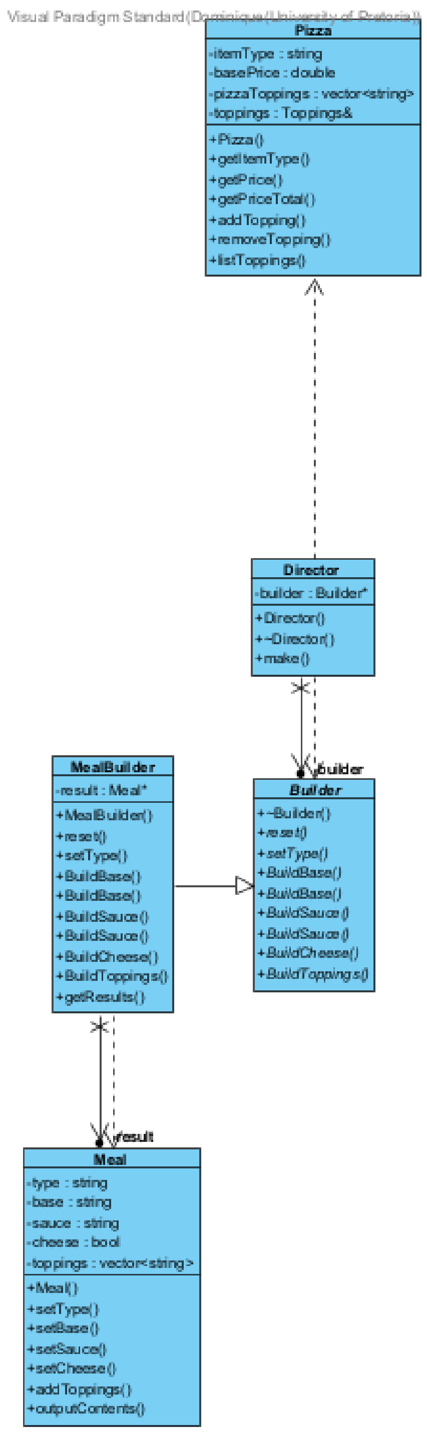


5. **Builder:** Separate the construction of a complex object from its representation, allowing the same construction process to create different representations.

The **Builder pattern** is used to construct our pizza menu items that have been ordered, from scratch. The Directory participant called the Director class constructs a pizza object using the Builder interface. The make(Pizza*) method is used, where a chosen menu item has been passed in as parameter that needs to be built.

Builder participant is the Builder class. ConcreteBuilder participant is the MealBuilder class. Individual components of the pizza, all have their own methods such as BuildBase() and BuildSauce().

Product being represented is the Meal class. The getResults() method that is being called in the MealBuilder class return a Meal.

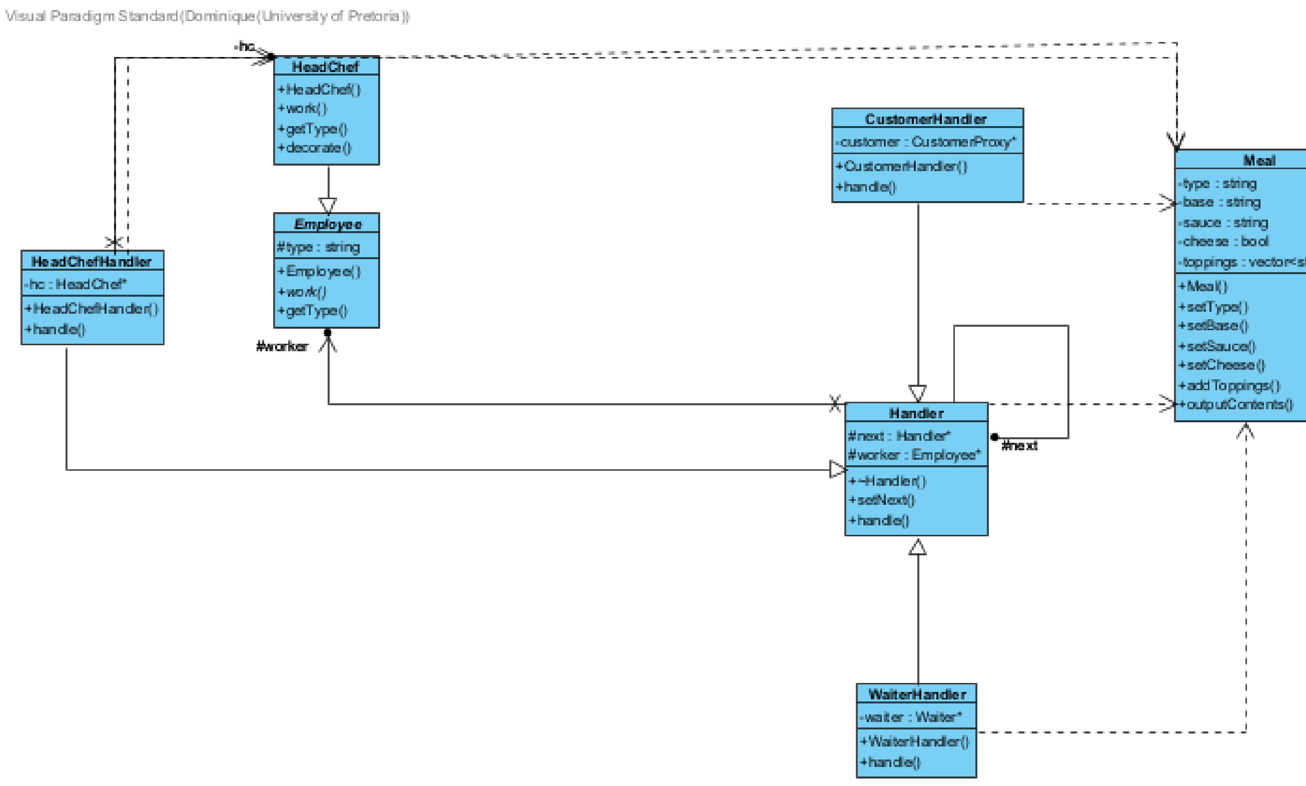


6. **Chain of Responsibility:** Create a chain of objects where each object can process a request and decide whether to pass it to the next object in the chain or stop processing it.

A variety of classes make use of the **Chain of Responsibility pattern** in order to be able to pass the order around such that the customer can be, ultimately, be served a meal.

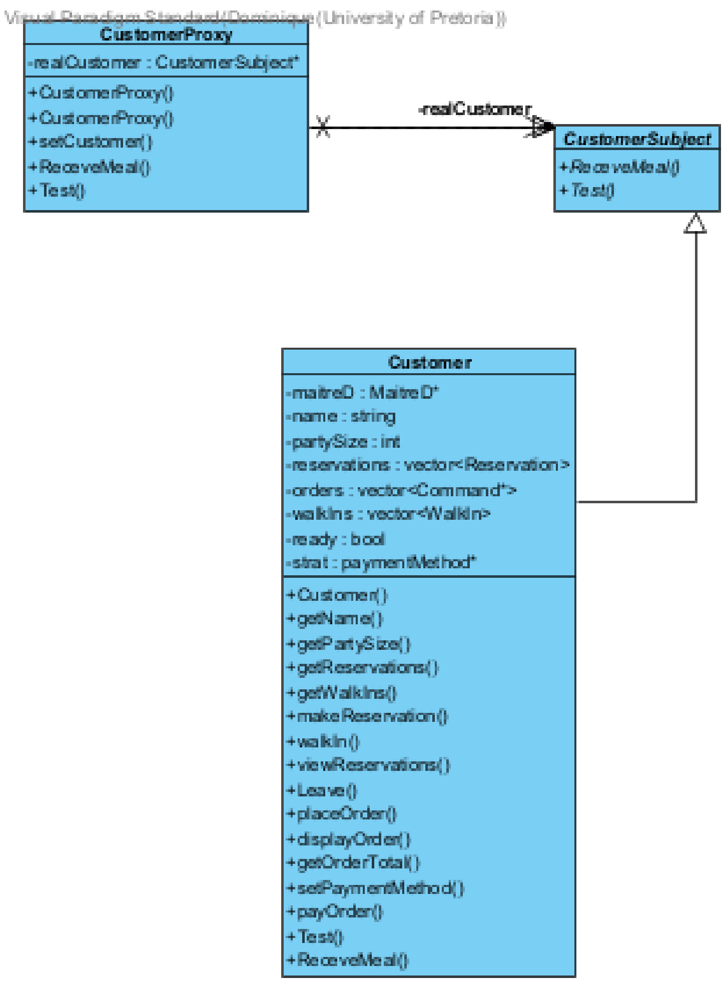
We have a main Handler class that is the Handler participant. The Handler points to a next handler, which is the class that happens to handle the order next. The ConcreteHandler participants are the CustomerHandler, WaiterHandler and the HeadChefHandler.

The order is passed from the headchef to the waiter and then to the customer. This pattern was used primarily, since it would be easier to add additional functionality for the classes to be able to handle and deliver the food, such as the head chef being able to decorate a meal.



7. **Proxy:** Provide a surrogate or placeholder for another object to control access to it. This pattern allows you to add an additional level of control and functionality, such as lazy initialization or access control, when dealing with the real object, without the client needing to be aware of it.

A surrogate customer is created as a Proxy participant. It is called the CustomerProxy, which inherits form CustomerSubject which is the Subject participant. Customer is the RealSubject. ReceiveMeal() is the request method. This is done for access control purposes as we do not want to send the actual Customer to the Kitchen, where the instance of the class can potentially be manipulated, which can cause problems within the system.

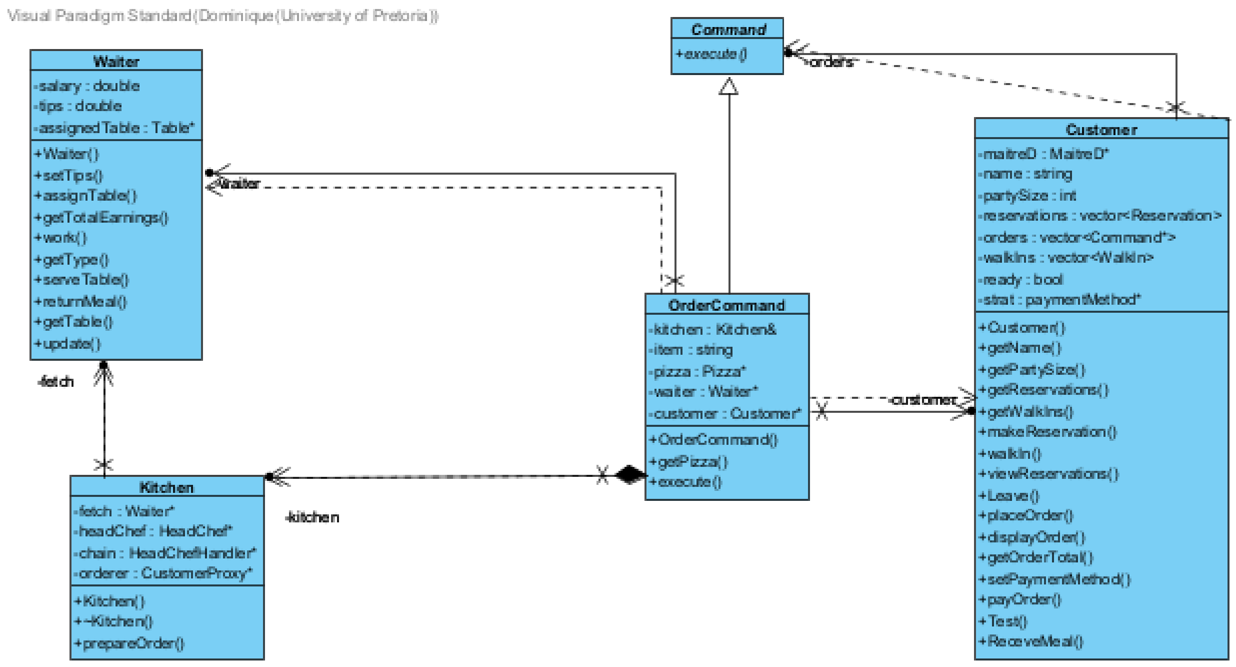


8. **Command**: Encapsulate a request as an object, thereby allowing for parameterization of clients with requests, queueing of requests and providing support for undoable operations.

Our Kitchen makes use of the **Command pattern** by having an prepareOrder(Pizza*) method.

The Command class is the Command participant, and the OrderCommand is the ConcreteCommand. ConcreteCommand holds an instance of the type of pizza that has been requested by the waiter. The Kitchen is then called to prepare the order with the specific pizza passed in as parameter.

The Waiter is the Invoker and the Kitchen is the Receiver of the request.

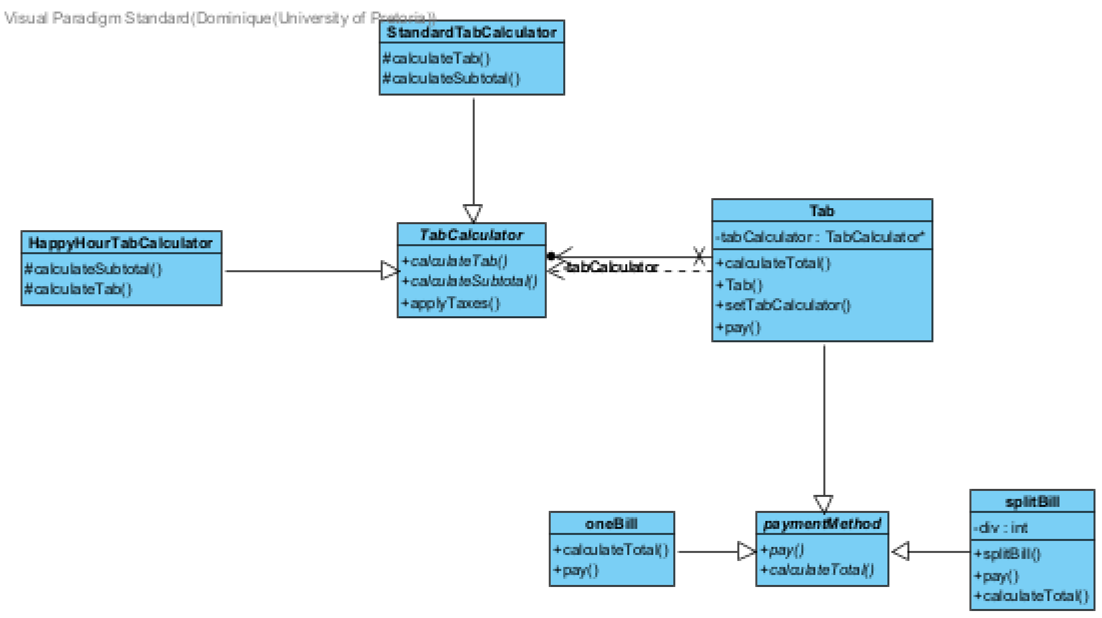


9. **Template Method:** Define the skeleton of an algorithm in a method, allowing certain steps of the algorithm to be implemented by subclasses. This pattern enables the reusability of the algorithm's structure while allowing variations in the implementation of specific steps.

The Template method design pattern is used to be able to have sub-class specific implementations of the algorithm.

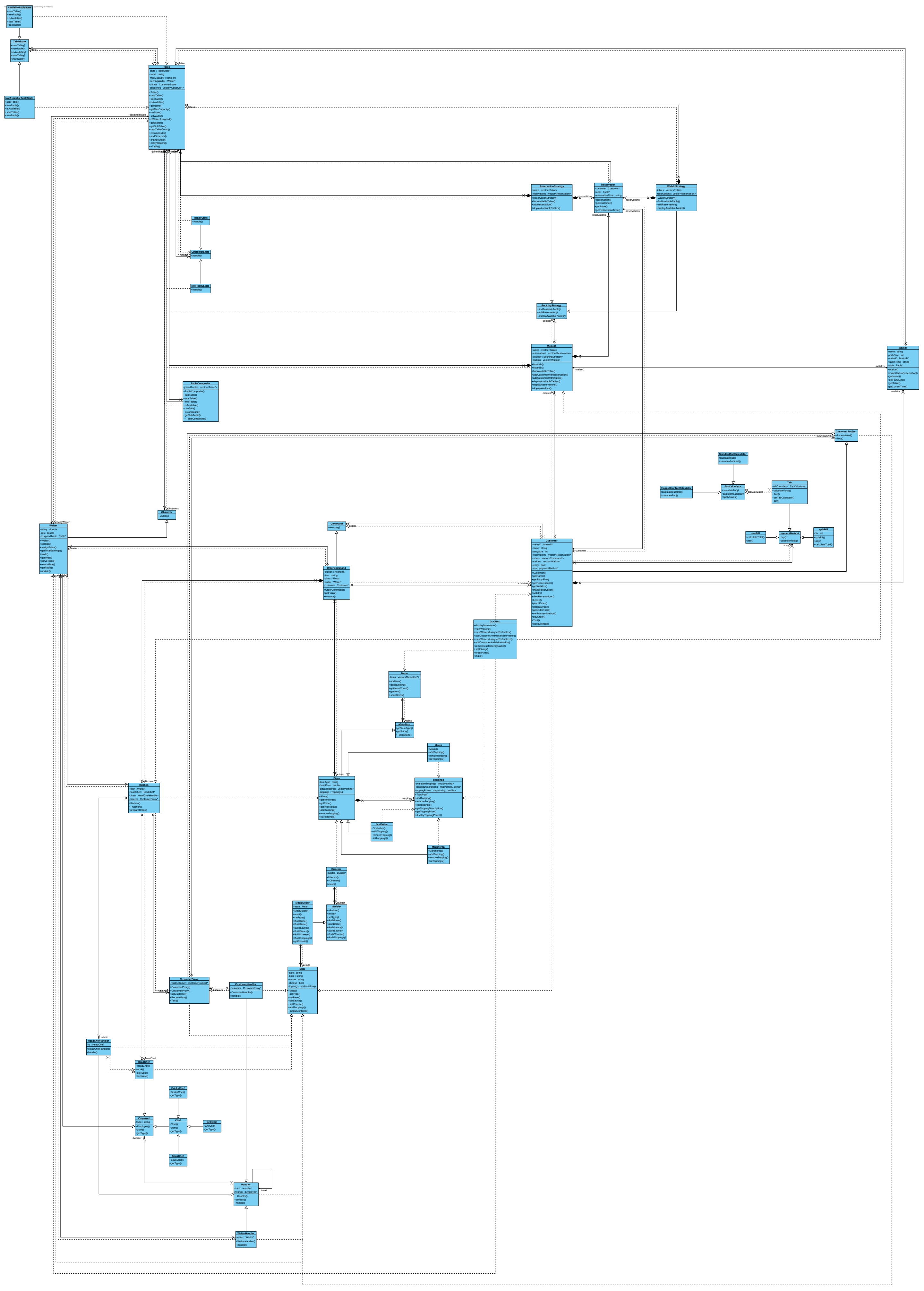
In our implementation there are two scenarios, we have the TabCalculator that is the Abstract class participant, with the ConcreteClass participants StandardTabCalculator and HappyHourTabCalculator.

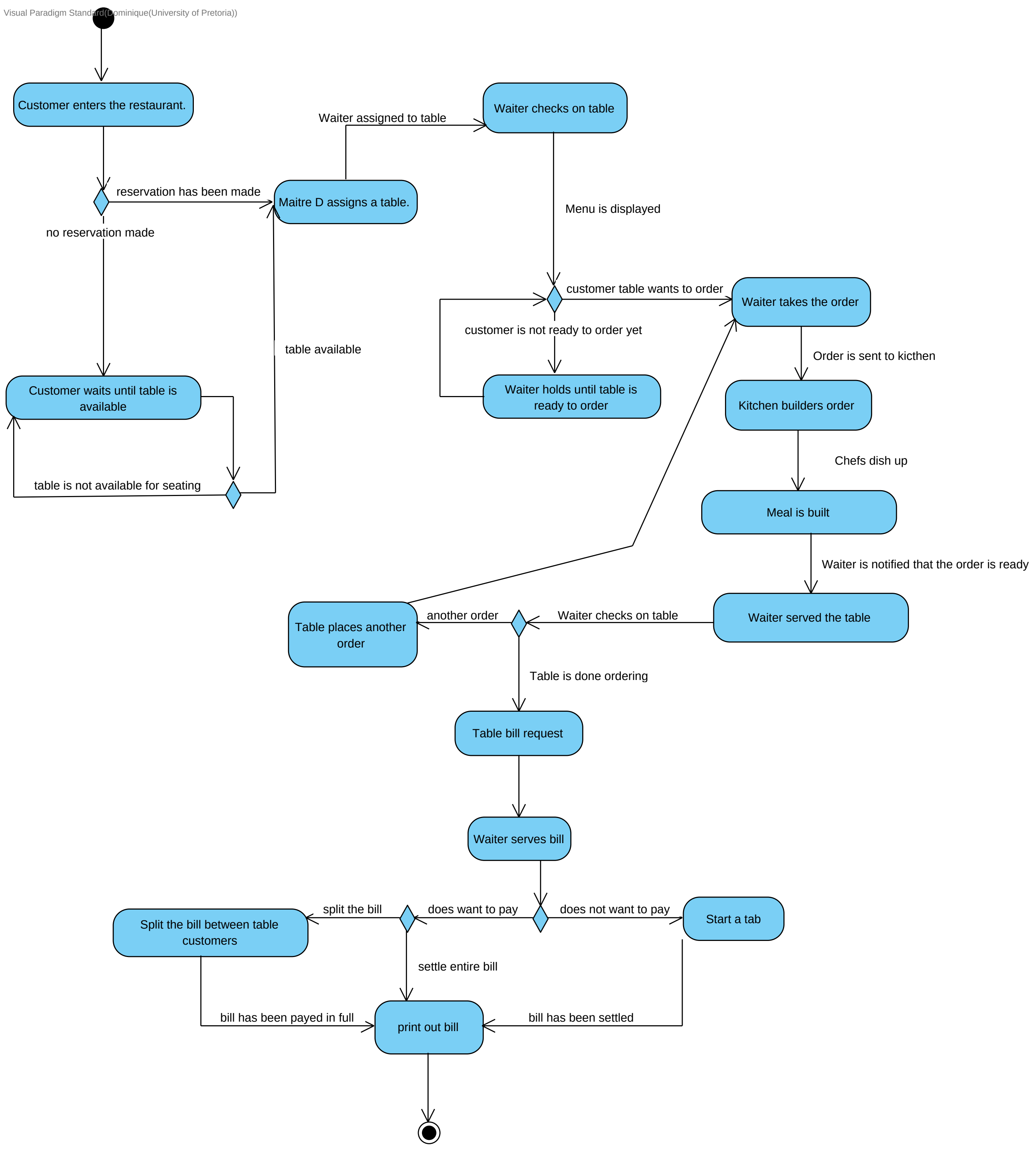
In addition we have the paymentMethod Abstract class participant with the template method calculateTotal() that is specific to the ConcreteClasses TabCalculator, oneBill and splitBill.

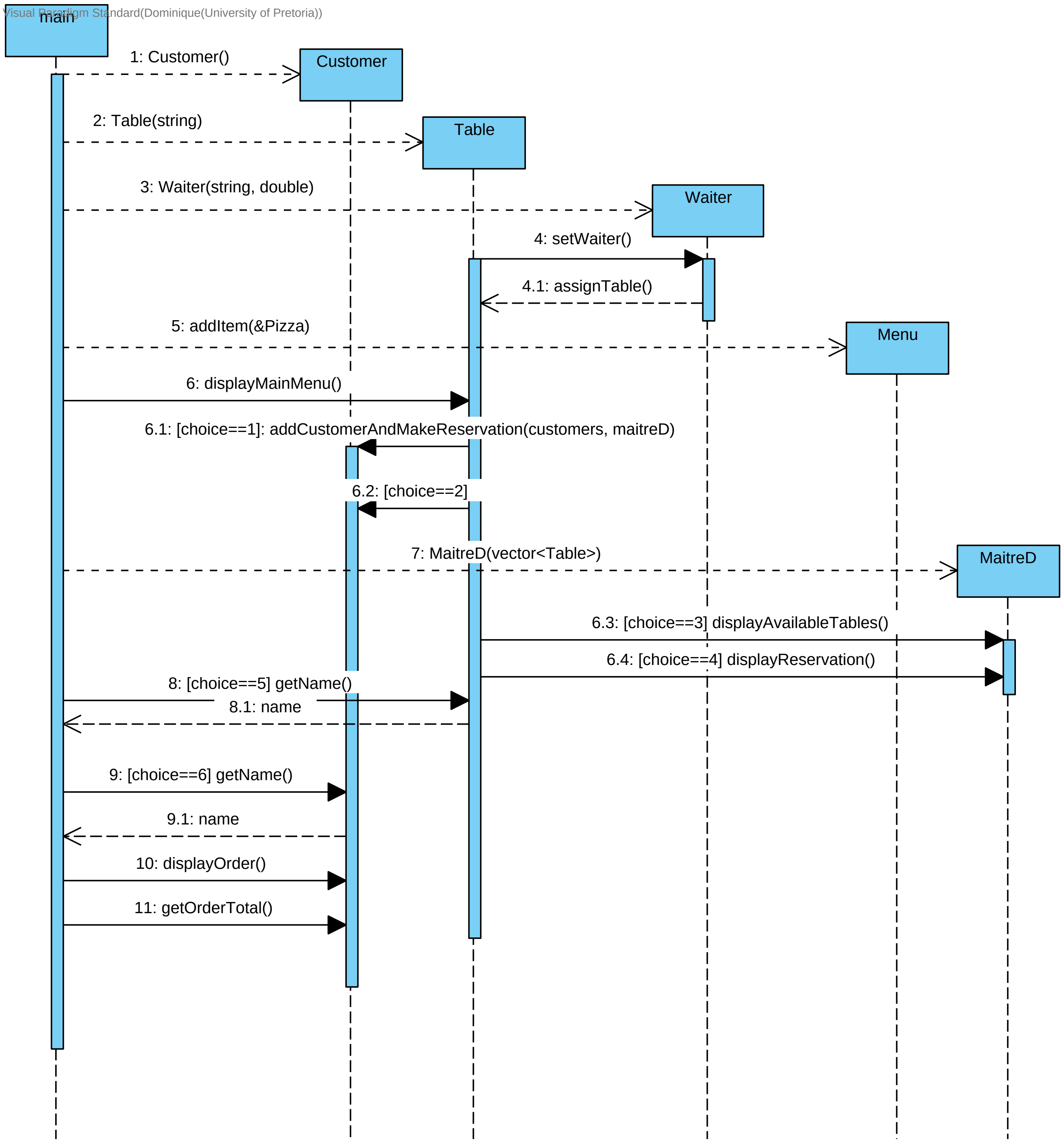


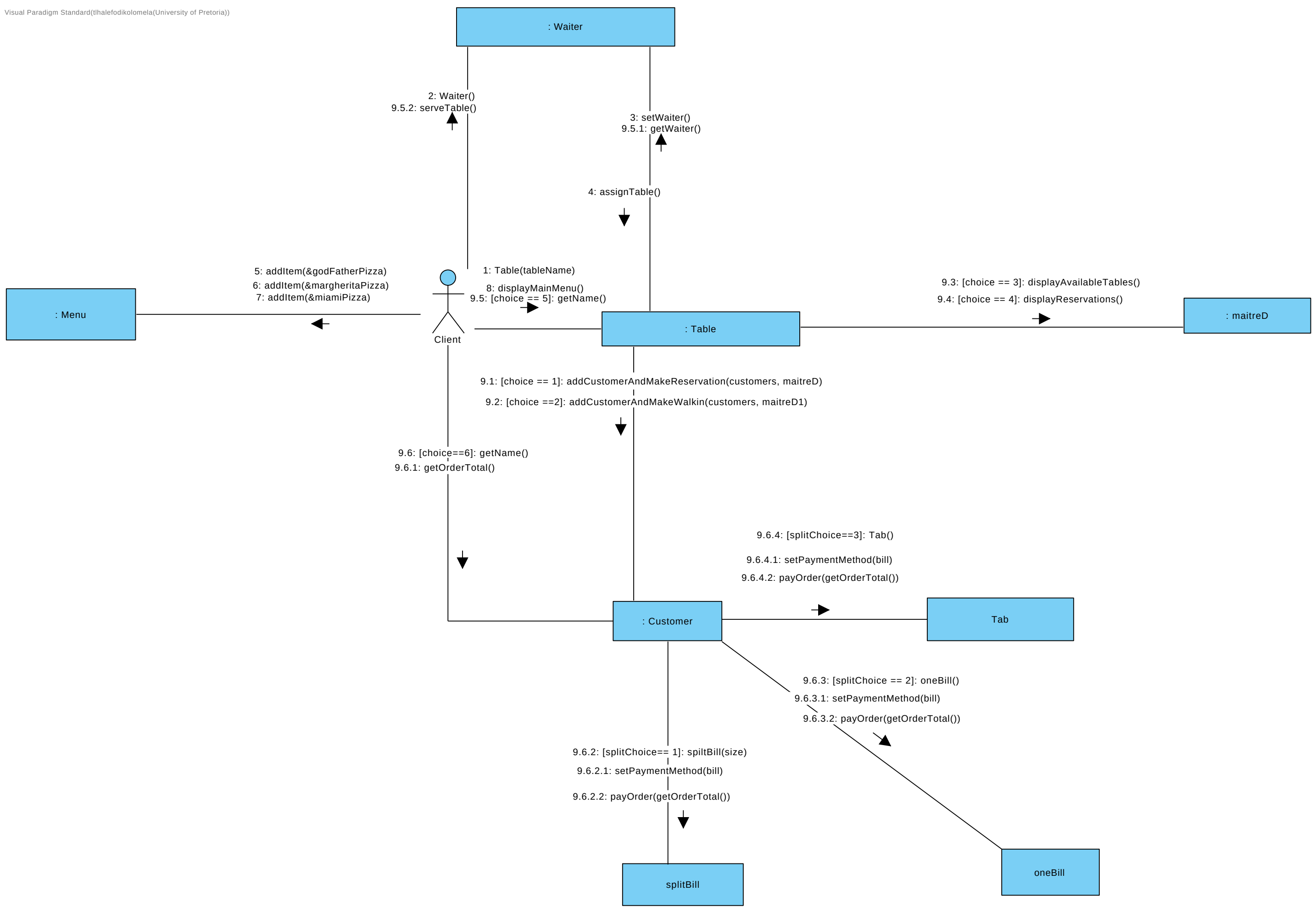
10. **Façade**: provide a unified and simplified interface to a set of interfaces in a subsystem. It serves as a higher-level entry point to make the subsystem easier to use, shielding clients from the complexities and details of the underlying components.

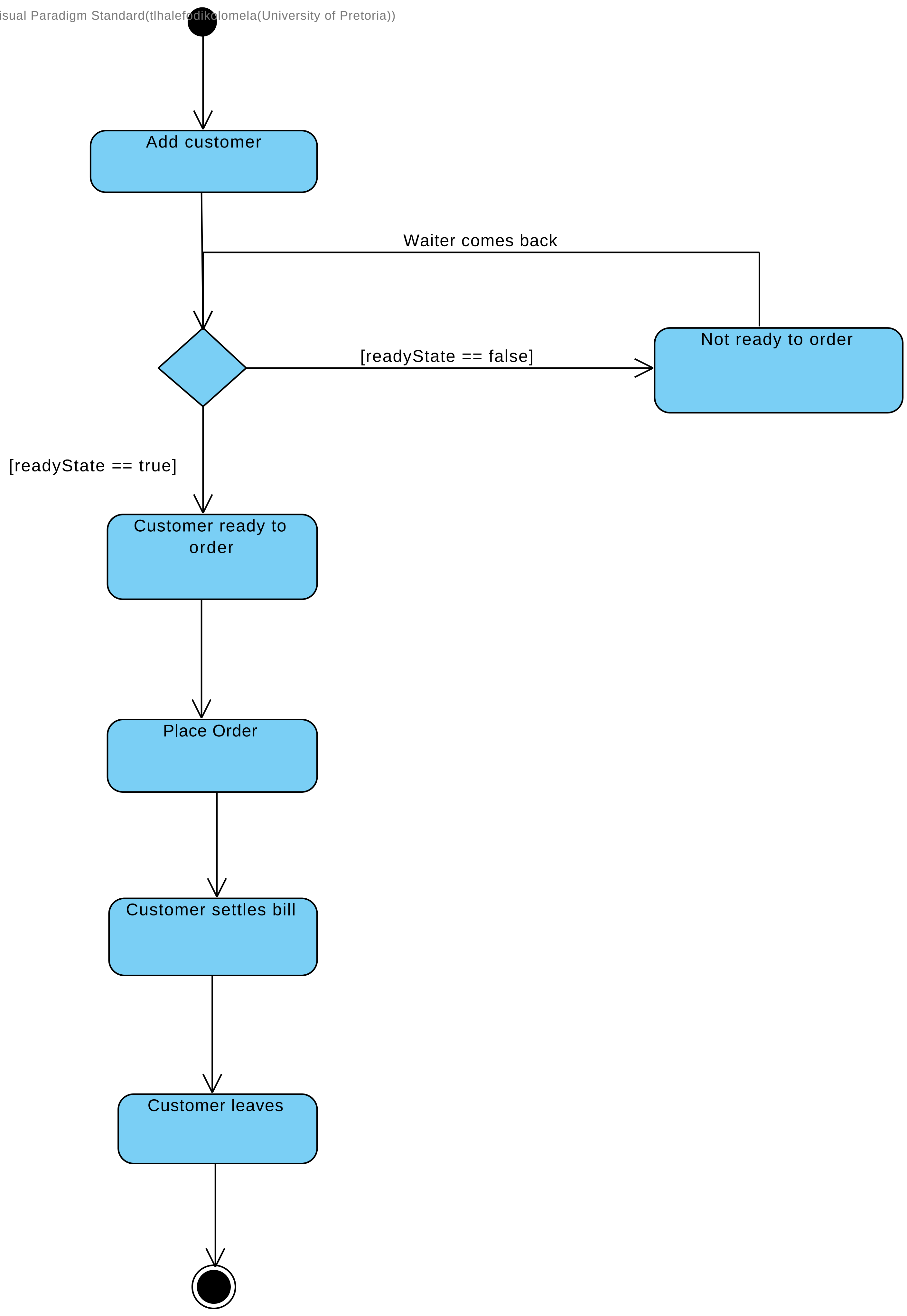
The façade participant is the entry point to the system. It will shield the client from the 10 subsystems we have currently implemented. The Façade will simply let die client call a select few methods in order to kick start the simulation, without having to call each individual methods from each of the 10 systems, thus the client still has access to each subsystem.

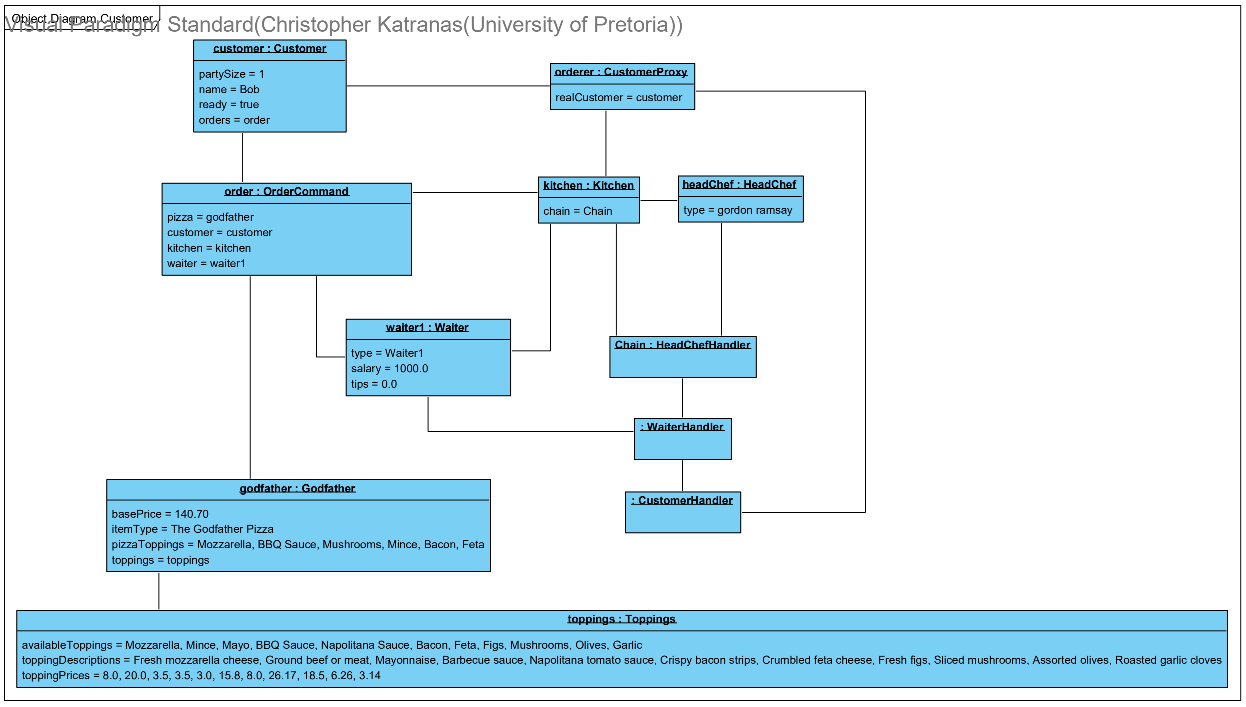

AvailableTableState
+seatTable()
+freeTable()
+isAvailable()
+seatTable()
+freeTable()
TableState
+seatTable()
+freeTable()
+isAvailable()
+seatTable()
+freeTable()
NotAvailableTableState
+seatTable()
+freeTable()
+isAvailable()
+seatTable()
+freeTable()
-state
Table
-state : TableState*
-name : string
-maxCapacity : const int
-servingWaiter : Waiter*
-cState : CustomerState*
-observers : vector<Observer*>
+Table()
+seatTable()
+freeTable()
+isAvailable()
+getName()
+getMaxCapacity()
+setState()
+setWaiter()
+isWaiterAssigned()
+getWaiter()
+getSubTable()
+seatTableComp()
+isComposite()
+addObserver()
+changeState()
+notifyWaiters()
+~Table()
-table
-tables
-assignedTable
-joinedTables
ReadyState
+handle()
CustomerState
+handle()
-cState
NotReadyState
+handle()
TableComposite
-joinedTables : vector<Table*>
+TableComposite()
+addTable()
+seatTable()
+freeTable()
+isAvailable()
+canJoin()
+isComposite()
+getSubTable()
+~TableComposite()
ReservationStrategy
-tables : vector<Table>
-reservations : vector<Reservation>
+ReservationStrategy()
+findAvailableTable()
+addReservation()
+displayAvailableTables()
-reservations
Reservation
-customer : Customer*
-table : Table*
-reservationTime : string
+Reservation()
+getCustomer()
+getTable()
+getReservationTime()
-reservations
WalkInStrategy
-tables : vector<Table>
-reservations : vector<Reservation>
+WalkInStrategy()
+findAvailableTable()
+addReservation()
+displayAvailableTables()
BookingStrategy
+findAvailableTable()
+addReservation()
+displayAvailableTables()
-strategy
MaitreD
-tables : vector<Table>
-reservations : vector<Reservation>
-strategy : BookingStrategy*
-walkIns : vector<WalkIn>
+MaitreD()
+MaitreD()
+findAvailableTable()
+addCustomerWithReservation()
+addCustomerWithWalkIn()
+displayAvailableTables()
+displayReservations()
+displayWalkIns()
-maitreD
-walkIns
WalkIn
-name : string
-partySize : int
-maitreD : MaitreD*
-walkInTime : string
-table : Table*
+WalkIn()
+makeWalkInReservation()
+getName()
+getPartySize()
+getTable()
-getCurrentTime()
CustomerSubject
+ReceveMeal()
+Test()
-realCustomer
StandardTabCalculator
#calculateTab()
#calculateSubtotal()
HappyHourTabCalculator
#calculateSubtotal()
#calculateTab()
TabCalculator
+calculateTab()
+calculateSubtotal()
+applyTaxes()
-tabCalculator
Tab
-tabCalculator : TabCalculator*
+calculateTotal()
+Tab()
+setTabCalculator()
+pay()
oneBill
+calculateTotal()
+pay()
paymentMethod
+pay()
+calculateTotal()
-strat
splitBill
-div : int
+splitBill()
+pay()
+calculateTotal()
Observer
+update()
-observers
Waiter
-salary : double
-tips : double
-assignedTable : Table*
+Waiter()
+setTips()
+assignTable()
+getTotalEarnings()
+work()
+getType()
+serveTable()
+returnMeal()
+getTable()
+update()
-servingWaiter
-waiter
Command
+execute()
-orders
OrderCommand
-kitchen : Kitchen&
-item : string
-pizza : Pizza*
-waiter : Waiter*
-customer : Customer*
+OrderCommand()
+getPizza()
+execute()
-customer
Customer
-maitreD : MaitreD*
-name : string
-partySize : int
-reservations : vector<Reservation>
-orders : vector<Command*>
-walkIns : vector<WalkIn>
-ready : bool
-strat : paymentMethod*
+Customer()
+getName()
+getPartySize()
+getReservations()
+getWalkIns()
+makeReservation()
+walkIn()
+viewReservations()
+Leave()
+placeOrder()
+displayOrder()
+getOrderTotal()
+setPaymentMethod()
+payOrder()
+Test()
+ReceveMeal()
GLOBAL
+displayMainMenu()
+viewWaiters()
+viewWaitersAssignedToTables()
+addCustomerAndMakeReservation()
+viewWaitersAssignedToTables1()
+addCustomerAndMakeWalkin()
+removeCustomerByName()
+splitString()
+orderPizza()
+main()
Menu
-items : vector<MenuItem*>
+addItem()
+displayMenu()
+getItemsCount()
+getItem()
+showItems()
-items
MenuItem
+getItemType()
+getPrice()
+~MenuItem()
Miami
+Miami()
+addTopping()
+removeTopping()
+listToppings()
-pizza
Pizza
-itemType : string
-basePrice : double
-pizzaToppings : vector<string>
-toppings : Toppings&
+Pizza()
+getItemType()
+getPrice()
+getPriceTotal()
+addTopping()
+removeTopping()
+listToppings()
-toppings
Toppings
-availableToppings : vector<string>
-toppingDescriptions : map<string, string>
-toppingPrices : map<string, double>
+Toppings()
+addTopping()
+removeTopping()
+listToppings()
+getToppingDescription()
+getToppingPrice()
+displayToppingPrices()
Godfather
+Godfather()
+addTopping()
+removeTopping()
+listToppings()
Margherita
+Margherita()
+addTopping()
+removeTopping()
+listToppings()
Director
-builder : Builder*
+Director()
+~Director()
+make()
-builder
MealBuilder
-result : Meal*
+MealBuilder()
+reset()
+setType()
+BuildBase()
+BuildBase()
+BuildSauce()
+BuildSauce()
+BuildCheese()
+BuildToppings()
+getResults()
Builder
+~Builder()
+reset()
+setType()
+BuildBase()
+BuildBase()
+BuildSauce()
+BuildSauce()
+BuildCheese()
+BuildToppings()
-result
Meal
-type : string
-base : string
-sauce : string
-cheese : bool
-toppings : vector<string>
+Meal()
+setType()
+setBase()
+setSauce()
+setCheese()
+addToppings()
+outputContents()
-kitchen
Kitchen
-fetch : Waiter*
-headChef : HeadChef*
-chain : HeadChefHandler*
-orderer : CustomerProxy*
+Kitchen()
+~Kitchen()
+prepareOrder()
-orderer
CustomerProxy
-realCustomer : CustomerSubject*
+CustomerProxy()
+CustomerProxy()
+setCustomer()
+ReceveMeal()
+Test()
-customer
CustomerHandler
-customer : CustomerProxy*
+CustomerHandler()
+handle()
-chain
HeadChefHandler
-hc : HeadChef*
+HeadChefHandler()
+handle()
-headChef
-hc
HeadChef
+HeadChef()
+work()
+getType()
+decorate()
DrinksChef
+DrinksChef()
+getType()
Employee
#type : string
+Employee()
+work()
+getType()
#worker
Chef
+Chef()
+work()
+getType()
GrillChef
+GrillChef()
+getType()
SousChef
+SousChef()
+getType()
Handler
#next : Handler*
#worker : Employee*
+~Handler()
+setNext()
+handle()
#next
WaiterHandler
-waiter : Waiter*
+WaiterHandler()
+handle()


Customer enters the restaurant.
reservation has been made
no reservation made
Maitre D assigns a table.
Waiter assigned to table
Waiter checks on table
Menu is displayed
table available
Customer waits until table is available
table is not available for seating
customer table wants to order
customer is not ready to order yet
Waiter holds until table is ready to order
Waiter takes the order
Order is sent to kicthen
Kitchen builders order
Chefs dish up
Meal is built
Waiter is notified that the order is ready
Waiter served the table
Waiter checks on table
another order
Table places another order
Table is done ordering
Table bill request
Waiter serves bill
does want to pay
does not want to pay
Start a tab
split the bill
Split the bill between table customers
settle entire bill
bill has been payed in full
bill has been settled
print out bill


main
Customer
Table
Waiter
Menu
MaitreD
1: Customer()
2: Table(string)
3: Waiter(string, double)
4: setWaiter()
4.1: assignTable()
5: addItem(&Pizza)
6: displayMainMenu()
6.1: [choice==1]: addCustomerAndMakeReservation(customers, maitreD)
6.2: [choice==2]
7: MaitreD(vector<Table>)
6.3: [choice==3] displayAvailableTables()
6.4: [choice==4] displayReservation()
8: [choice==5] getName()
8.1: name
9: [choice==6] getName()
9.1: name
10: displayOrder()
11: getOrderTotal()


: Waiter
2: Waiter()
9.5.2: serveTable()
3: setWaiter()
9.5.1: getWaiter()
4: assignTable()
: Menu
5: addItem(&godFatherPizza)
6: addItem(&margheritaPizza)
7: addItem(&miamiPizza)
Client
1: Table(tableName)
8: displayMainMenu()
9.5: [choice == 5]: getName()
: Table
9.3: [choice == 3]: displayAvailableTables()
9.4: [choice == 4]: displayReservations()
: maitreD
9.1: [choice == 1]: addCustomerAndMakeReservation(customers, maitreD)
9.2: [choice ==2]: addCustomerAndMakeWalkin(customers, maitreD1)
9.6: [choice==6]: getName()
9.6.1: getOrderTotal()
9.6.4: [splitChoice==3]: Tab()
9.6.4.1: setPaymentMethod(bill)
9.6.4.2: payOrder(getOrderTotal())
: Customer
Tab
9.6.3: [splitChoice == 2]: oneBill()
9.6.3.1: setPaymentMethod(bill)
9.6.3.2: payOrder(getOrderTotal())
9.6.2: [splitChoice== 1]: spiltBill(size)
9.6.2.1: setPaymentMethod(bill)
9.6.2.2: payOrder(getOrderTotal())
splitBill
oneBill


Add customer
Waiter comes back
[readyState == false]
Not ready to order
[readyState == true]
Customer ready to order
Place Order
Customer settles bill
Customer leaves


Object Diagram Customer
customer : Customer
partySize = 1
name = Bob
ready = true
orders = order
orderer : CustomerProxy
realCustomer = customer
order : OrderCommand
pizza = godfather
customer = customer
kitchen = kitchen
waiter = waiter1
kitchen : Kitchen
chain = Chain
headChef : HeadChef
type = gordon ramsay
waiter1 : Waiter
type = Waiter1
salary = 1000.0
tips = 0.0
Chain : HeadChefHandler
: WaiterHandler
: CustomerHandler
godfather : Godfather
basePrice = 140.70
itemType = The Godfather Pizza
pizzaToppings = Mozzarella, BBQ Sauce, Mushrooms, Mince, Bacon, Feta
toppings = toppings
toppings : Toppings
availableToppings = Mozzarella, Mince, Mayo, BBQ Sauce, Napolitana Sauce, Bacon, Feta, Figs, Mushrooms, Olives, Garlic
toppingDescriptions = Fresh mozzarella cheese, Ground beef or meat, Mayonnaise, Barbecue sauce, Napolitana tomato sauce, Crispy bacon strips, Crumbled feta cheese, Fresh figs, Sliced mushrooms, Assorted olives, Roasted garlic cloves
toppingPrices = 8.0, 20.0, 3.5, 3.5, 3.0, 15.8, 8.0, 26.17, 18.5, 6.26, 3.14