Burgundy

COS 214 Final Project Report

by
Lerato Gololo · Brenden van der Mescht · Siyamthanda Ndlovu
Katlego Zondo · Sechaba Ntshehi
Group 24

Table of Contents

Research

A write-up of all the research you have done to complete this assignment including references

Coding Standards

- We researched the purpose and benefits of coding standards when beginning a collaborative project.
- Coding standards were helpful in software development as they promote consistency, readability, and maintainability of code, making it easier for us to understand other developer's work.
- Additionally, adherence to coding standards can lead to reduced bugs (for example, one of our standards was having a single return line in order to limit possible points of errors in debugging).
- We also researched what is essential to include in the standards for out project, such as naming conventions, code formatting and not including namespaces in .h and .cpp files as they make it difficult to determine the origin of functions, classes, or variables.
- Resources
  - https://dev.to/codacy/coding-standards-what-are-they-and-why-do-you-need-them-51db
  - https://www.w3schools.in/software-testing/standards

Git Standards

- Looking into the different work flows we could use while developing our project, we ended up choosing the Gitlow workflow to use.
- This was because its release-based approach is intuitive and easy to understand.
- We also appreciated that the Gitlow workflow has clear separation of branches, and dedicated branch for hotfixes, which would allow for efficient development.
- Resources
  - https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
  - https://www.abtasty.com/blog/git-branching-strategies/
  - https://www.youtube.com/watch?v=1SXpE08hvGs&pp=ygUSYysrIHVzZXIgaW50ZXJmYWNl
  - https://www.youtube.com/watch?v=RGOj5yH7evk&t=1007s&pp=ygUSYysrIHVzZXIgaW50ZXJmYWNl

User Interface

- We explored the possibility of adding a user interface for our project
- One option was using embacardo to develop an application that could run on a PC
- Another option we were considering was to use a web toolkit (the Wt web toolkit) which would allow us to a create web-based application that would have C++ be the backend language
- Resources
  - https://stackoverflow.com/questions/7894112/web-gui-for-c-console-application
  - https://www.webtoolkit.eu/wt/
  - https://www.youtube.com/watch?v=FxQTXyR4mjs&t=604s&pp=ygUSYysrIHVzZXIgaW50ZXJmYWNl

Design Decisions

Our reasoning for the design decisions we have made.

System Requirements

Customer

Customer should be able to :

- Walk in an request to be seated
- Be assigned to a table
- Create an order and then place it by sending it to the waiter
- Specify ingredients and remove or add to the order prior to it being sent to the kitchen
- Tell a waiter to come back later if they are not ready to order
- Split the bill or pay it in full
- Decide on payment method (by tab or pay immediately)
- Receive their order and eat it

System Requirements

Manager

Manager should be able to :

- Assign tables for customers
- Build tables which can hold a maximum of four tables
- Assign waiters to tables
- Check on customer satisfaction
- Know when customers have left the restaurant

System Requirements

Waiter

Waiter should be able to :

- Be assigned a table
- Observe customer satisfaction for the customers at their table
- Receive orders from customers
- Take orders to the kitchen
- Receive the meals from the kitchen and deliver them to the customers
- Create bills and send them to customers
- Notify manager when customers are done paying

System Requirements

Waiter

Chef should be able to :

- Receive orders from waiters
- Use order to construct meals
- Head chef should plate food to be sent to waiter
- Head chef also be able to observe customer satisfaction

Menu-Order System

- Table is the interfaces the waiter uses to interact with customers sitting at the table
- A table holds :
  - 1. customer
  - 2. menus
  - 3. ingredients
- The table will hold an array of customers. The table also holds an array of vectors (on array holds ingredient vectors, another will hold sauce vectors, etc)
- Array of customers map one-to-one with the vector arrays

Floor Staff Responsibilities

- Waiter
  - Takes customer order
  - Order to kitchen via the kitchen
  - Managing tables
  - Creates customer bill
  - Passes orders to customers (once completed)

- Manager
  - Checks table availability
  - Books tables per customer
  - Assigns waiters to table

- Customers
  - Book a table
  - Walk in
  - Complain
  - Split bill
  - Build an order
  - Open a tab
  - Tip waiters

Kitchen Staff Responsibilities

- Head Chef
    - Receive order from the waiter
    - Pass order to chefs to be cooked
    - Receive meal after being cooked
    - Pass the meal to waiter

- Kitchen Chefs
    - Create burgers, sauces, drinks and fries
    - Create corresponding products if the order indicates that a customer wants that product

Design Patterns

A write-up showing how all the patterns have been used, also note what problem the pattern solved in your implementation.

Chain of Responsibility

- Numerous chefs need to interact with the same order item and decide whether they have to create a product that corresponds to them
- Using a chain of responsibility allows more than one object to handle the request (the ordered food) and this is a cleaner design since the sender of the meal isn’t coupled with the object(s) that receive the request

Decorator

- Customers may order burgers but have to be able to specify what ingredients they may want in a burger (pickle, lettuce, patty or tomato)
- The customer may want any combination of ingredients
- The decorator pattern is a flexible alternative to subclassing for extending options customers have regarding what their burger is made up of

Factory

- The drinks chef returns three different types of drinks and using the factory patter, we can define an interface for creating an the drink, but lets subclasses decide which class to instantiate

Iterator

- When the meals will be paid for by splitting the bill, we iterate through the customers object sequentially without exposing its underlaying representation

Memento

- Prior to the order being sent to the kitchen to be created, the customer may wish to change their mind about their order
- This means that a customer should be able to revert its order to a previous state should they so wish
- We used the memento capture the order's state so that the object can be restored to this state later

State

- Over time the each table may be in different states
- A table may be occupied, unoccupied, dirty or have its bill paid
- We want a table to be able to alter its behaviour depending on what its internal state is.
- The state pattern is used for exactly this.

Strategy

- In our restaurant, there are multiple ways of paying for the bill
- Customers can either open a tab, split the bill, or pay the bill on their own
- The strategy pattern allows us to ddefine a family of algorithms for paying the bill, and make the algorithms interchangeable since they accomplish the same thing.

Builder

- Each table in our restaurant can seat four customers
- Should a group of customers be larger than 4 then the manager will have to combine them
- Using the builder pattern, we can separate the construction of the table object from its representation so that the same construction process can create different representations of the tables

Mediator

- The waiter has to facilitate communication between the head chef and the table by taking customer's meals to and from the kitchen
- Using the mediator pattern promotes loose coupling since it keeps the table and head chef objects from referring to each other explicitly, and it lets you vary the interaction independently.

Observer

- Each table in the restaurant can either be satisfied or dissatisfied
- The head chef, manager and waiters should all be able to observe whether the tables are satisfied or not
- Using the observer pattern we define a one-to-many dependency between the restaurant tables and the staff members so that when a table changes its state, all its observers are notified and updated automatically

Design Alterations

Any and all assumptions, alterations, and design decisions.

Assumptions

In the implementation of our system, we made the following assumptions:

- There are an infinite number of tables that customers can be seated at
- There are an infinite number of waiters that can be assigned to the tables in the restaurant
- Each customer that starts a tab is able to pay later and will do so in their own time
- Each table is assigned a single waiter

Modified Responsibilities

- Manager will no longer :
  - Check table availability
  - Book tables per customer

- Customers will no longer :
  - Book a table
  - Tip waiters

Modified/Removed Classes

Menus

- Menus will no longer be used to create the objects that will make up the order
- Chefs create own products
- This change leads to less closely coupled code

Potato

- The fries menu will no longer exist hence the potato class will have no purpose

Tomato, Chutney and BBQ Sauces

- There will no longer be different types of sauces on offered by the restaurant
- One burgundy sauce will be served

Abstract Table

- The table class will no longer inherit from a table

Abstract Iterator

- The iterator used when customers choose to split the bill will no longer have an abstract class
- This is because there is no need for a uniform interface since there is only one iterator created in the whole system

Modified/Added Classes

Kitchen Staff

- The head chef as well as all of the chefs in the chain of responsibility will all inherit from this abstract class

FloorStaff

- Staff members who observe customer satisfaction will all inherit from this abstract class in order to provide a uniform interface in the implementation of the observer pattern

Fries

- This if the object the fries chef will create and return

BurgundySauce

- This if the object the sauce chef will create and return

Colleague

- This is an abstract class that the participants of the mediator pattern will inherit from

State

- This is a class that contains the attributes in the order class that we wish to store
- This class is used in the implementation of the memento pattern allowing customers to undo their orders

TableState

- This is an abstract class that the participants of the state pattern will inherit from

UML Diagrams

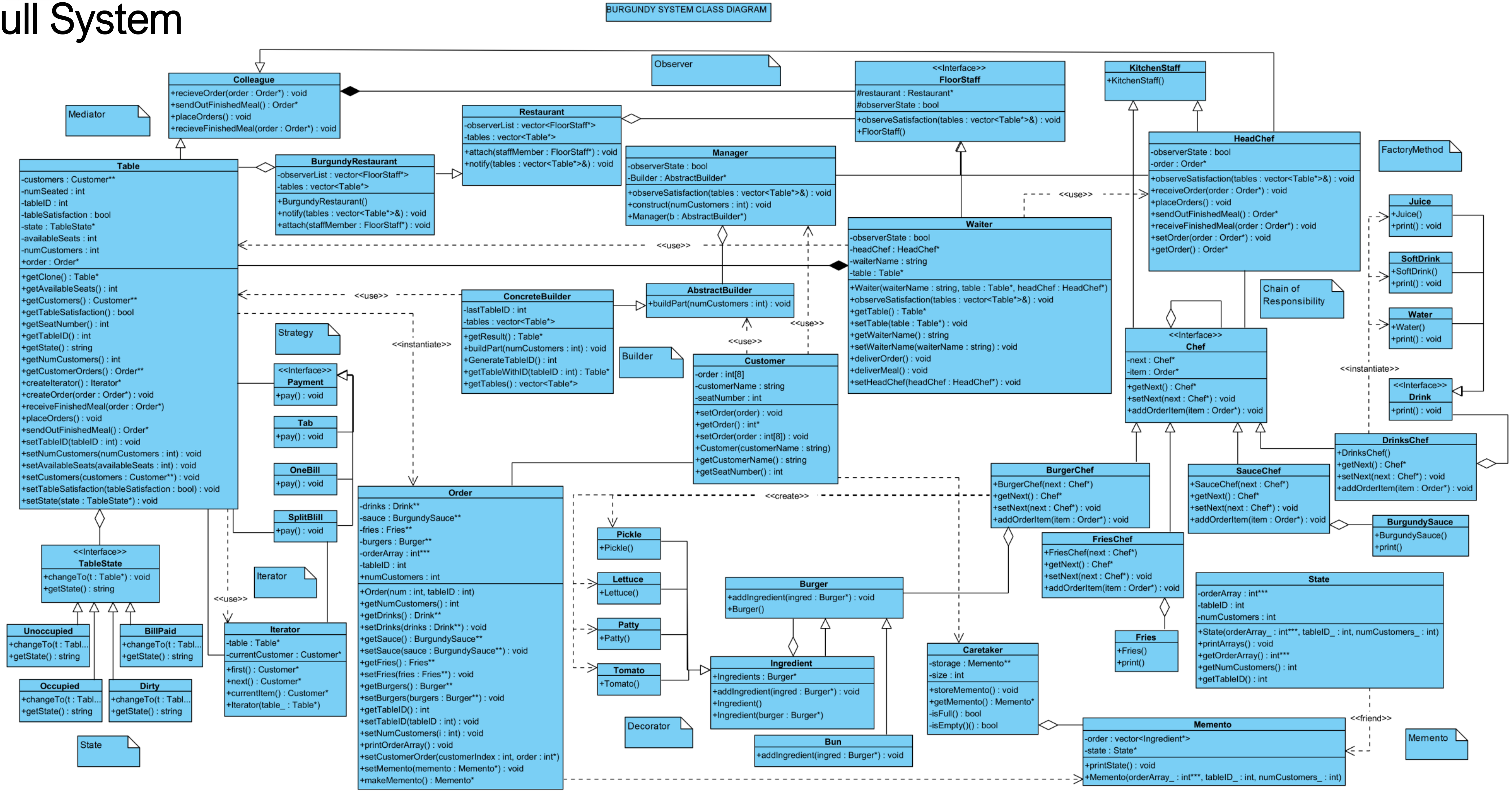
UML diagrams in support of the text presented in the report

Final UML Diagrams

UML diagrams for entire system and for critical system components

Final Class Diagram

Full System



BURGUNDY SYSTEM CLASS DIAGRAM

Colleague
+recieveOrder(order : Order*) : void
+sendOutFinishedMeal() : Order*
+placeOrders() : void
+recieveFinishedMeal(order : Order*) : void

Table
-customers : Customer**
-numSeated : int
-tableID : int
-tableSatisfaction : bool
-state : TableState*
-availableSeats : int
-numCustomers : int
+order : Order*
+getClone() : Table*
+getAvailableSeats() : int
+getCustomers() : Customer**
+getTableSatisfaction() : bool
+getSeatNumber() : int
+getTableID() : int
+getState() : string
+getNumCustomers() : int
+getCustomerOrders() : Order**
+createIterator() : Iterator*
+createOrder(order : Order*) : void
+receiveFinishedMeal(order : Order*)
+placeOrders() : void
+sendOutFinishedMeal() : Order*
+setTableID(tableID : int) : void
+setNumCustomers(numCustomers : int) : void
+setAvailableSeats(availableSeats : int) : void
+setCustomers(customers : Customer**) : void
+setTableSatisfaction(tableSatisfaction : bool) : void
+setState(state : TableState*) : void

BurgundyRestaurant
-observerList : vector<FloorStaff*>
-tables : vector<Table*>
+BurgundyRestaurant()
+notify(tables : vector<Table*>&) : void
+attach(staffMember : FloorStaff*) : void

Restaurant
-observerList : vector<FloorStaff*>
-tables : vector<Table*>
+attach(staffMember : FloorStaff*) : void
+notify(tables : vector<Table*>&) : void

Manager
-observerState : bool
-Builder : AbstractBuilder*
+observeSatisfaction(tables : vector<Table*>&) : void
+construct(numCustomers : int) : void
+Manager(b : AbstractBuilder*)

<<Interface>>
FloorStaff
#restaurant : Restaurant*
#observerState : bool
+observeSatisfaction(tables : vector<Table*>&) : void
+FloorStaff()

KitchenStaff
+KitchenStaff()

HeadChef
-observerState : bool
-order : Order*
+observeSatisfaction(tables : vector<Table*>&) : void
+receiveOrder(order : Order*) : void
+placeOrders() : void
+sendOutFinishedMeal() : Order*
+receiveFinishedMeal(order : Order*) : void
+setOrder(order : Order*) : void
+getOrder() : Order*

Waiter
-observerState : bool
-headChef : HeadChef*
-waiterName : string
-table : Table*
+Waiter(waiterName : string, table : Table*, headChef : HeadChef*)
+observeSatisfaction(tables : vector<Table*>&) : void
+getTable() : Table*
+setTable(table : Table*) : void
+getWaiterName() : string
+setWaiterName(waiterName : string) : void
+deliverOrder() : void
+deliverMeal() : void
+setHeadChef(headChef : HeadChef*) : void

ConcreteBuilder
-lastTableID : int
-tables : vector<Table*>
+getResult() : Table*
+buildPart(numCustomers : int) : void
+GenerateTableID() : int
+getTableWithID(tableID : int) : Table*
+getTables() : vector<Table*>

AbstractBuilder
+buildPart(numCustomers : int) : void

Customer
-order : int[8]
-customerName : string
-seatNumber : int
+setOrder(order) : void
+getOrder() : int*
+setOrder(order : int[8]) : void
+Customer(customerName : string)
+getCustomerName() : string
+getSeatNumber() : int

<<Interface>>
Payment
+pay() : void

Tab
+pay() : void

OneBill
+pay() : void

SplitBlill
+pay() : void

<<Interface>>
TableState
+changeTo(t : Table*) : void
+getState() : string

Unoccupied
+changeTo(t : Tabl...
+getState() : string

BillPaid
+changeTo(t : Tabl...
+getState() : string

Occupied
+changeTo(t : Tabl...
+getState() : string

Dirty
+changeTo(t : Tabl...
+getState() : string

Iterator
-table : Table*
-currentCustomer : Customer*
+first() : Customer*
+next() : Customer*
+currentItem() : Customer*
+Iterator(table_ : Table*)

Order
-drinks : Drink**
-sauce : BurgundySauce**
-fries : Fries**
-burgers : Burger**
-orderArray : int***
-tableID : int
+numCustomers : int
+Order(num : int, tableID : int)
+getNumCustomers() : int
+getDrinks() : Drink**
+setDrinks(drinks : Drink**) : void
+getSauce() : BurgundySauce**
+setSauce(sauce : BurgundySauce**) : void
+getFries() : Fries**
+setFries(fries : Fries**) : void
+getBurgers() : Burger**
+setBurgers(burgers : Burger**) : void
+getTableID() : int
+setTableID(tableID : int) : void
+setNumCustomers(i : int) : void
+printOrderArray() : void
+setCustomerOrder(customerIndex : int, order : int*)
+setMemento(memento : Memento*) : void
+makeMemento() : Memento*

Pickle
+Pickle()

Lettuce
+Lettuce()

Patty
+Patty()

Tomato
+Tomato()

Burger
+addIngredient(ingred : Burger*) : void
+Burger()

Ingredient
+Ingredients : Burger*
+addIngredient(ingred : Burger*) : void
+Ingredient()
+Ingredient(burger : Burger*)

Bun
+addIngredient(ingred : Burger*) : void

Caretaker
-storage : Memento**
-size : int
+storeMemento() : void
+getMemento() : Memento*
-isFull() : bool
-isEmpty()() : bool

Memento
-order : vector<Ingredient*>
-state : State*
+printState() : void
+Memento(orderArray_ : int***, tableID_ : int, numCustomers_ : int)

State
-orderArray : int***
-tableID : int
-numCustomers : int
+State(orderArray_ : int***, tableID_ : int, numCustomers_ : int)
+printArrays() : void
+getOrderArray() : int***
+getNumCustomers() : int
+getTableID() : int

<<Interface>>
Chef
-next : Chef*
-item : Order*
+getNext() : Chef*
+setNext(next : Chef*) : void
+addOrderItem(item : Order*) : void

BurgerChef
+BurgerChef(next : Chef*)
+getNext() : Chef*
+setNext(next : Chef*) : void
+addOrderItem(item : Order*) : void

SauceChef
+SauceChef(next : Chef*)
+getNext() : Chef*
+setNext(next : Chef*) : void
+addOrderItem(item : Order*) : void

FriesChef
+FriesChef(next : Chef*)
+getNext() : Chef*
+setNext(next : Chef*) : void
+addOrderItem(item : Order*) : void

DrinksChef
+DrinksChef()
+getNext() : Chef*
+setNext(next : Chef*) : void
+addOrderItem(item : Order*) : void

Fries
+Fries()
+print()

BurgundySauce
+BurgundySauce()
+print()

Juice
+Juice()
+print() : void

SoftDrink
+SoftDrink()
+print() : void

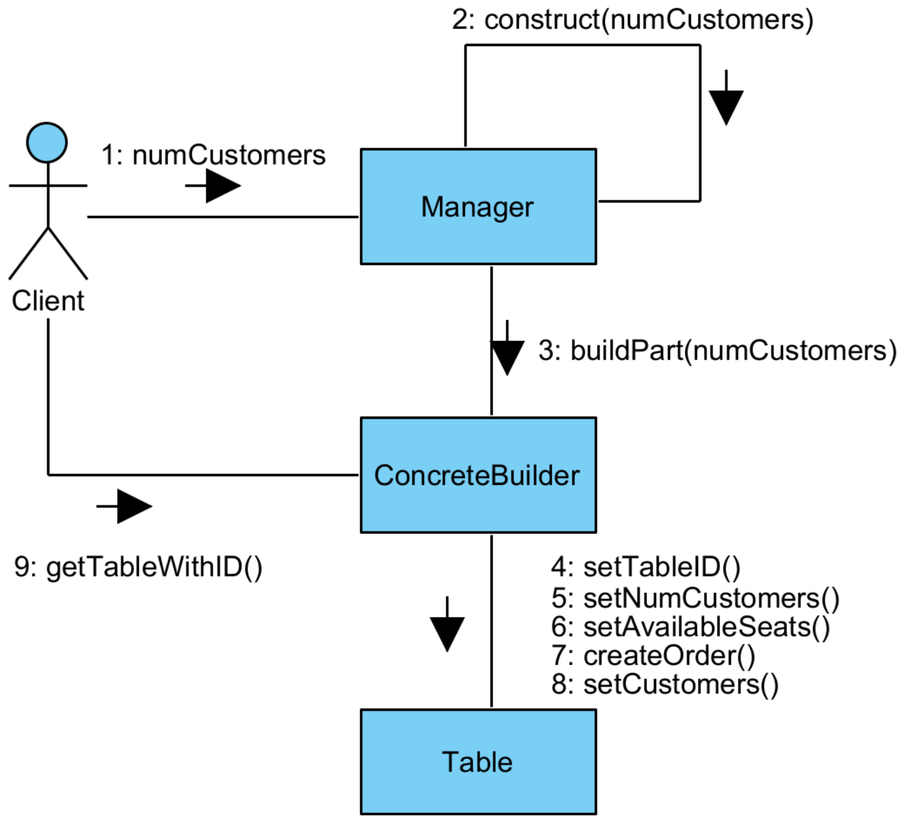
Water
+Water()
+print() : void

<<Interface>>
Drink
+print() : void

Communication Diagram

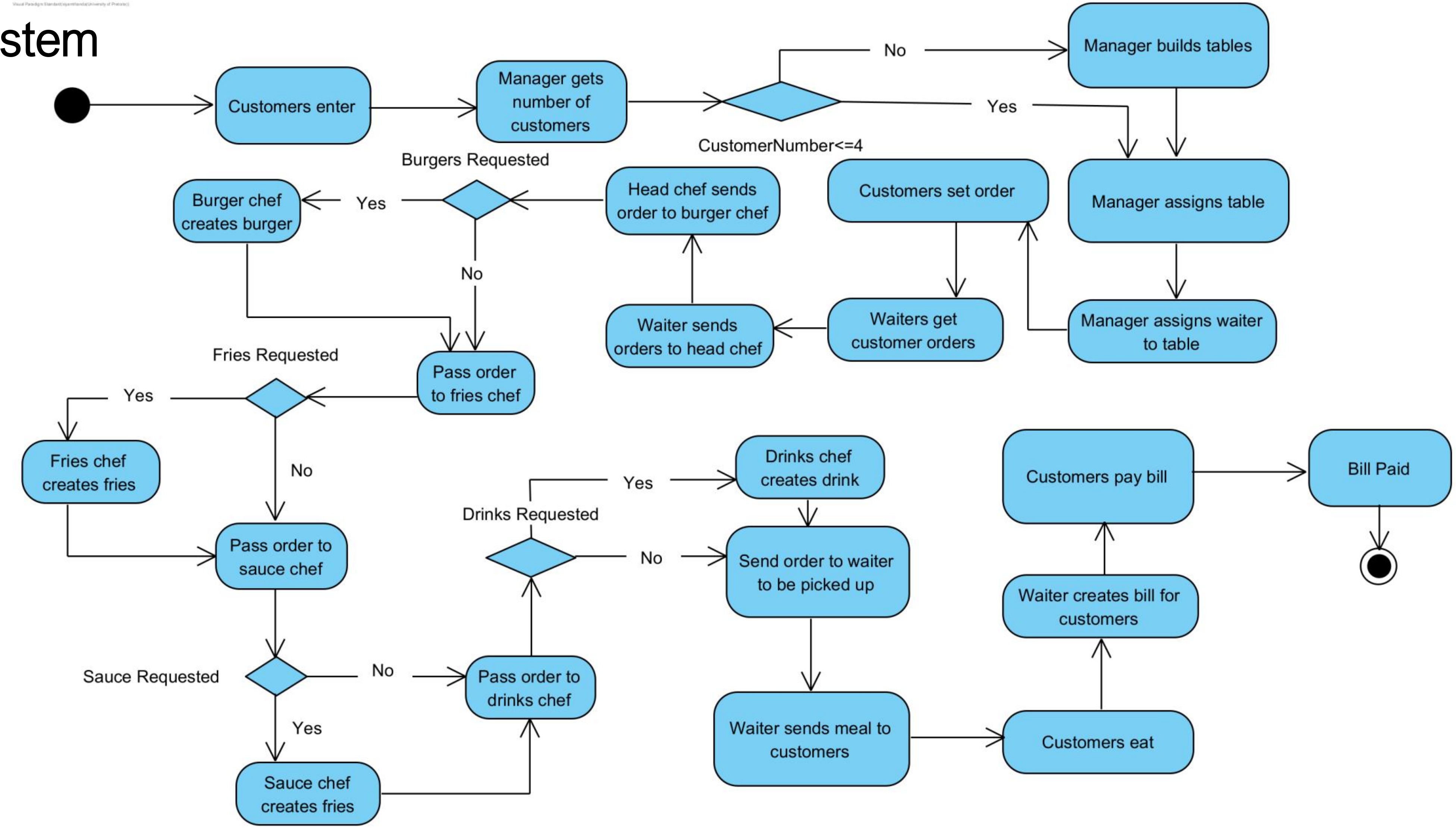
Builder

BURGUNDY SYSTEM BUILDER COMMUNICATION DIAGRAM



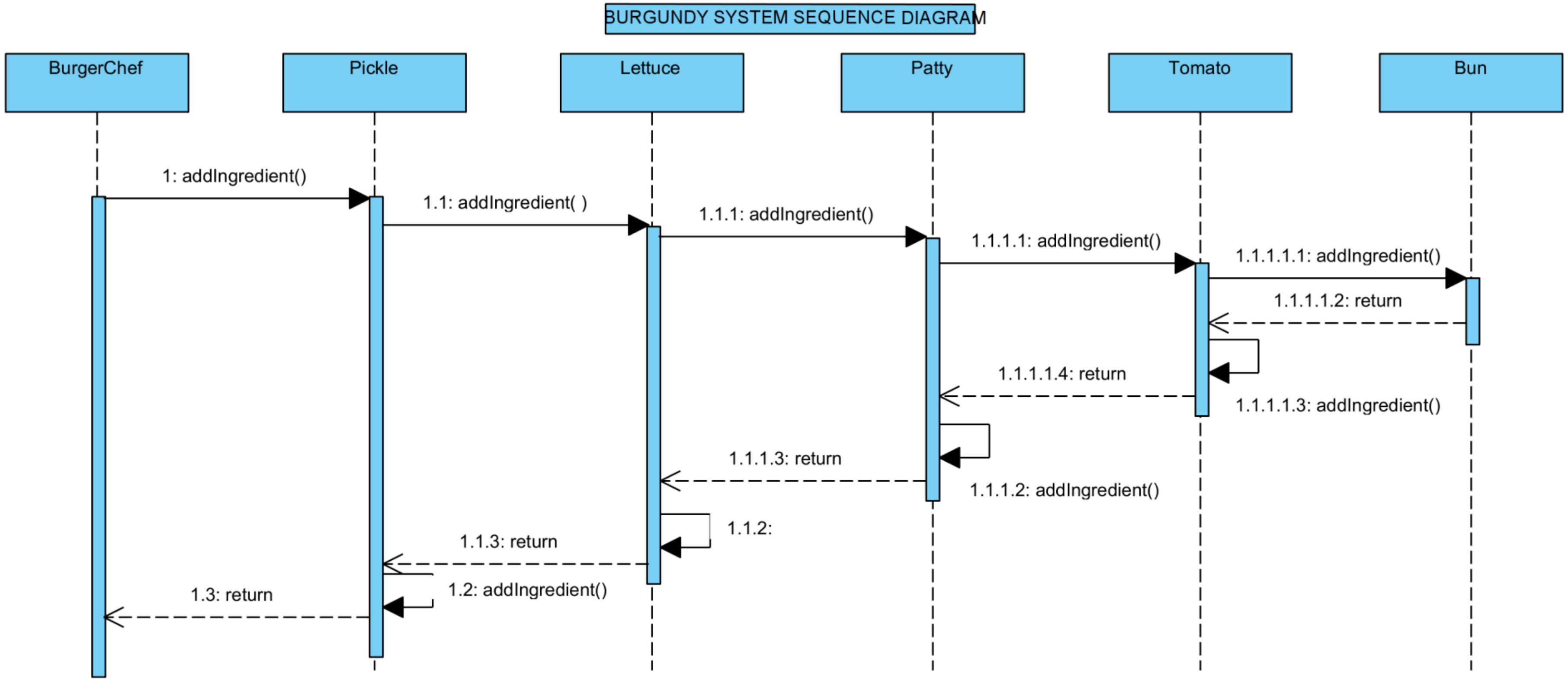
Activity Diagram

Full System


Customers enter
Manager gets number of customers
No
Manager builds tables
Yes
CustomerNumber<=4
Manager assigns table
Customers set order
Manager assigns waiter to table
Waiters get customer orders
Waiter sends orders to head chef
Head chef sends order to burger chef
Burgers Requested
Yes
Burger chef creates burger
No
Pass order to fries chef
Fries Requested
Yes
Fries chef creates fries
No
Pass order to sauce chef
Sauce Requested
No
Yes
Sauce chef creates fries
Pass order to drinks chef
Drinks Requested
Yes
Drinks chef creates drink
No
Send order to waiter to be picked up
Waiter sends meal to customers
Customers eat
Waiter creates bill for customers
Customers pay bill
Bill Paid

Sequence Diagram

Decorator

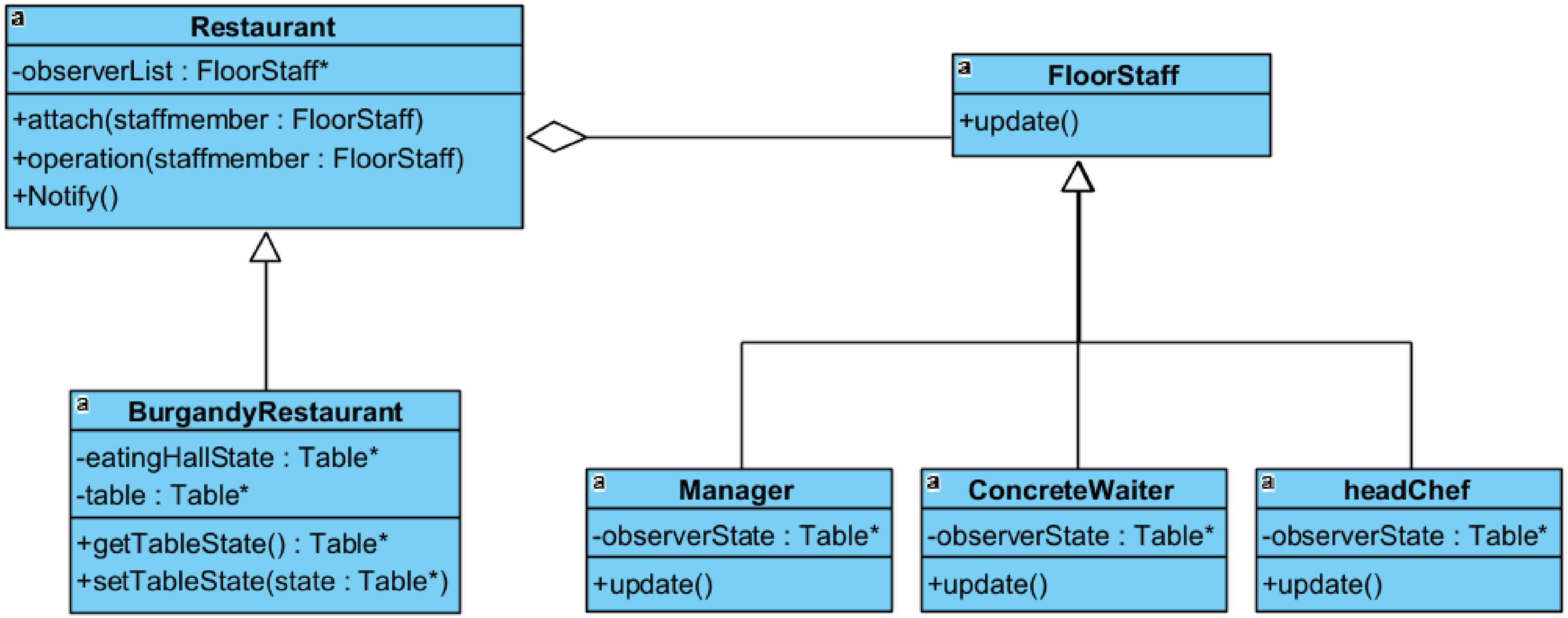


Partial UML Diagrams (Task 2)

Diagrams drawn up at the early on in the design process (Task 2)

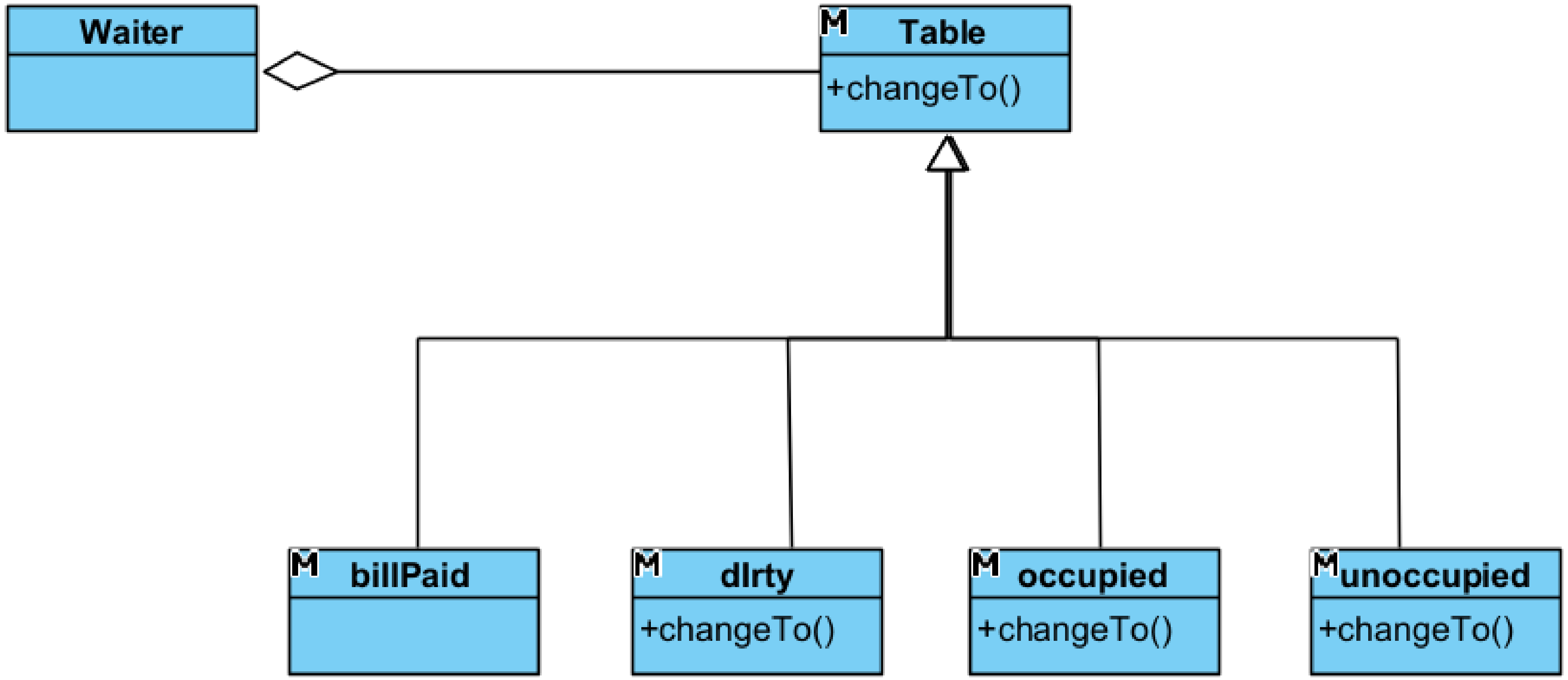
Class Diagram

Staff Observing Restaurant (Observer Pattern)



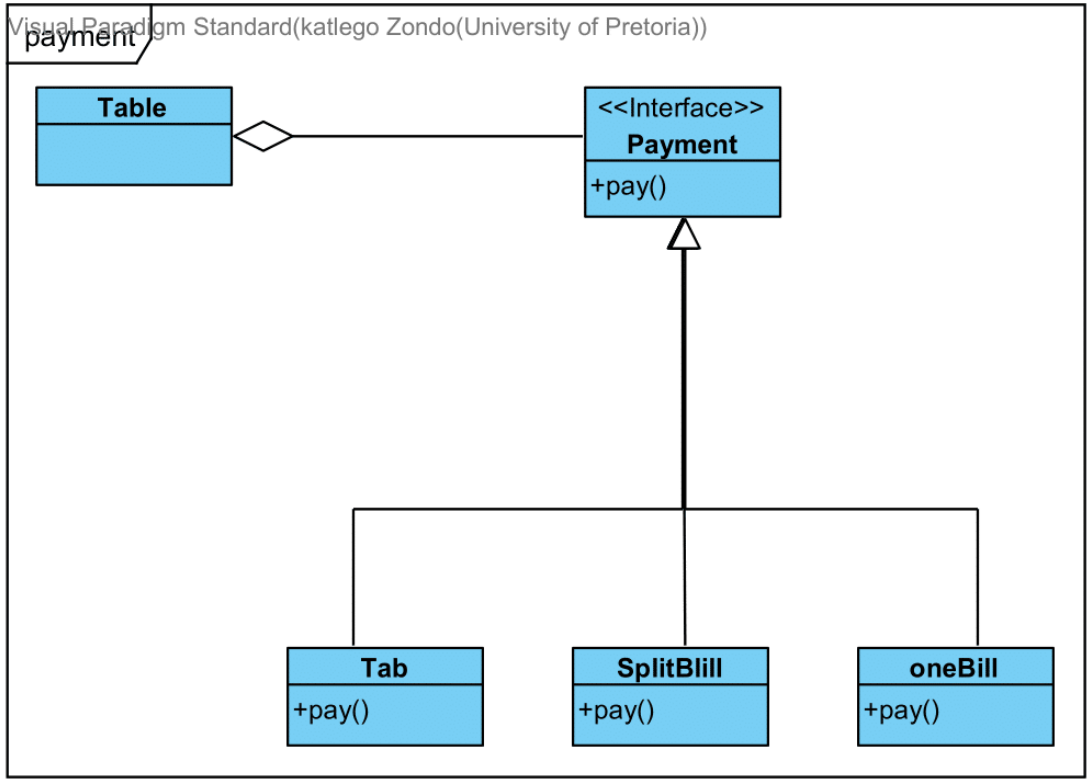
Class Diagram

Table State (State Pattern)



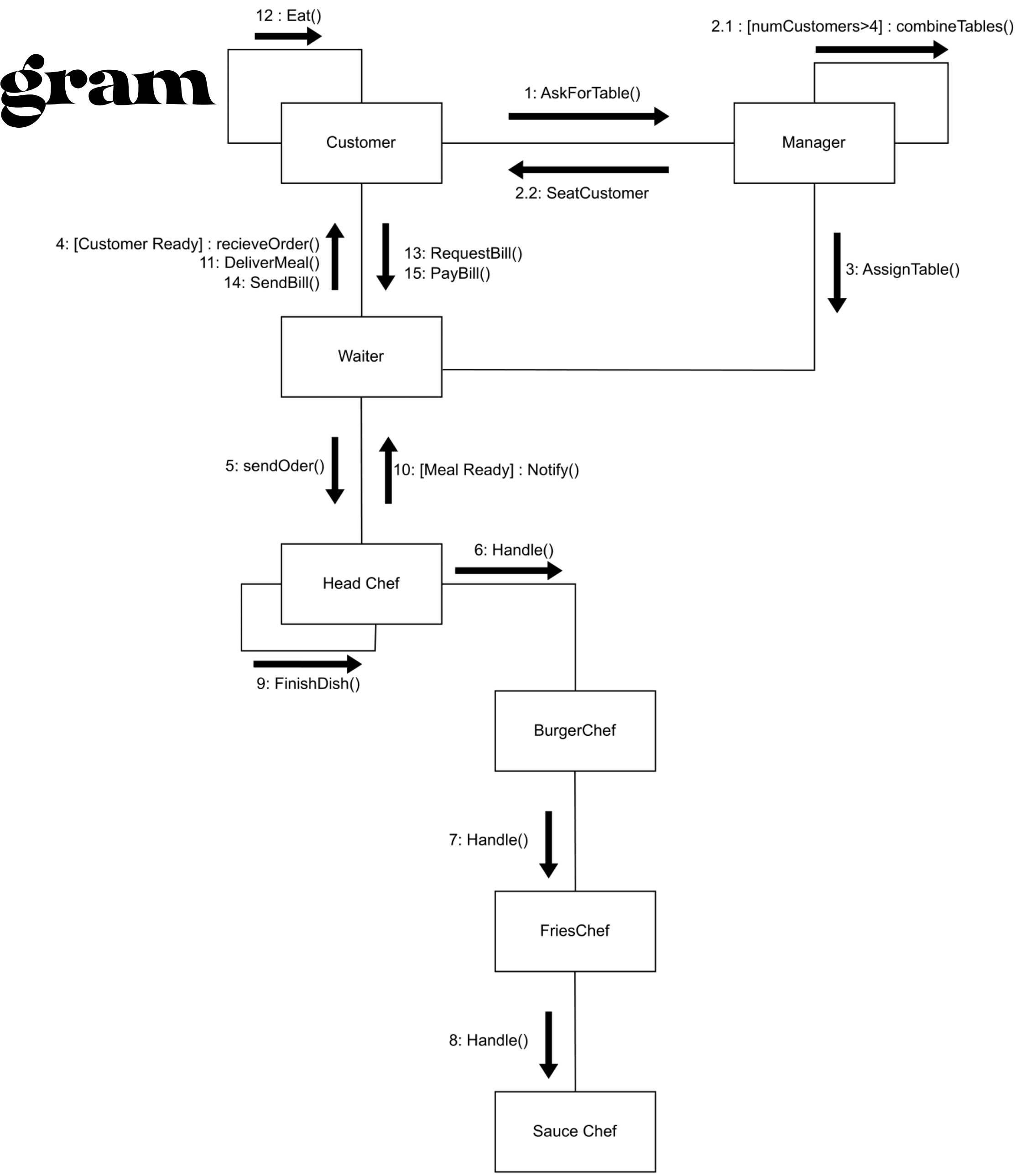
Class Diagram

Payment (Strategy Pattern)



Communication Diagram

Floor-Kitchen



12 : Eat()

2.1 : [numCustomers>4] : combineTables()

1: AskForTable()

Customer

Manager

2.2: SeatCustomer

4: [Customer Ready] : recieveOrder()
11: DeliverMeal()
14: SendBill()

13: RequestBill()
15: PayBill()

3: AssignTable()

Waiter

5: sendOder()

10: [Meal Ready] : Notify()

6: Handle()

Head Chef

9: FinishDish()

BurgerChef

7: Handle()

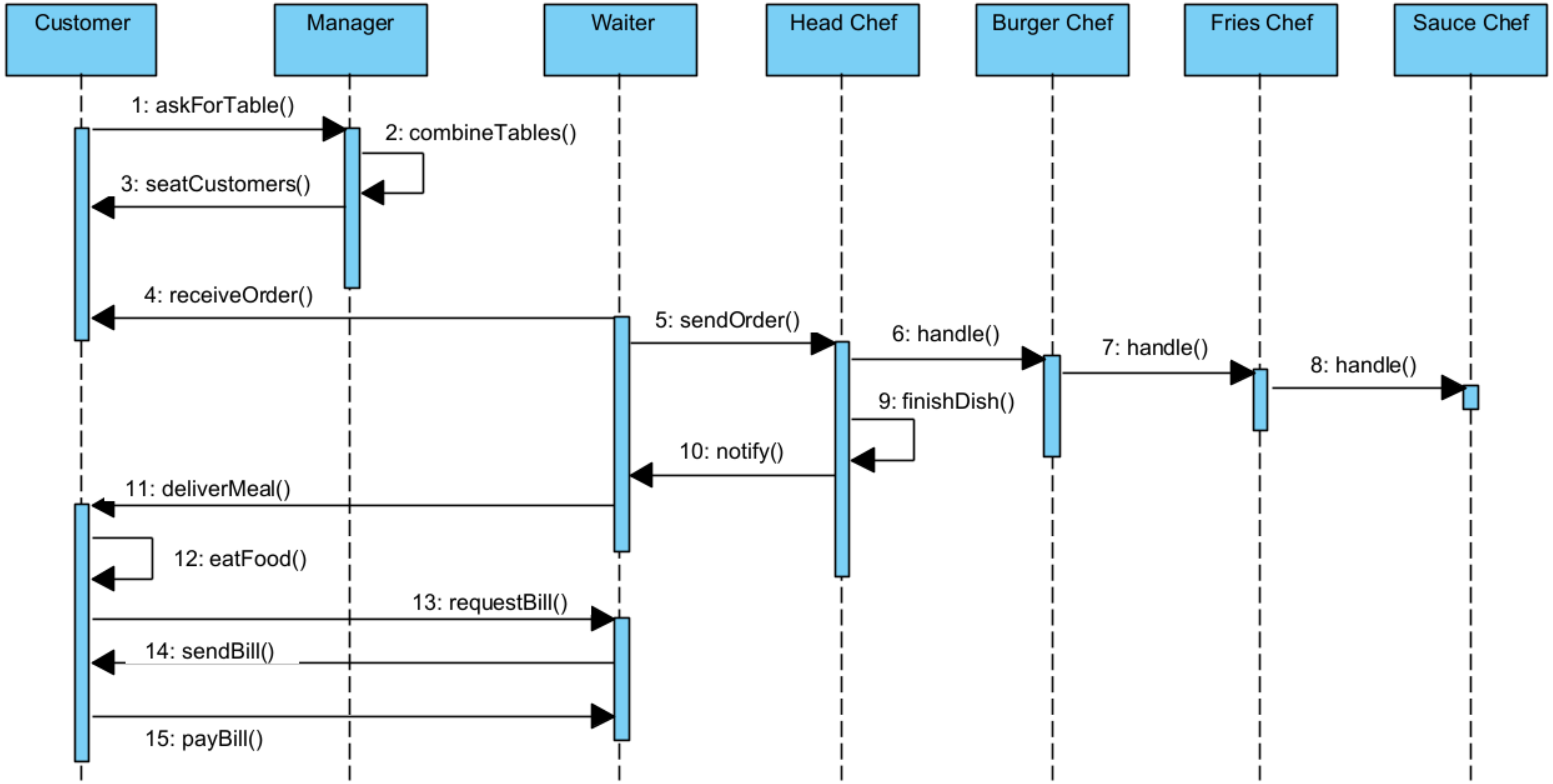
FriesChef

8: Handle()

Sauce Chef

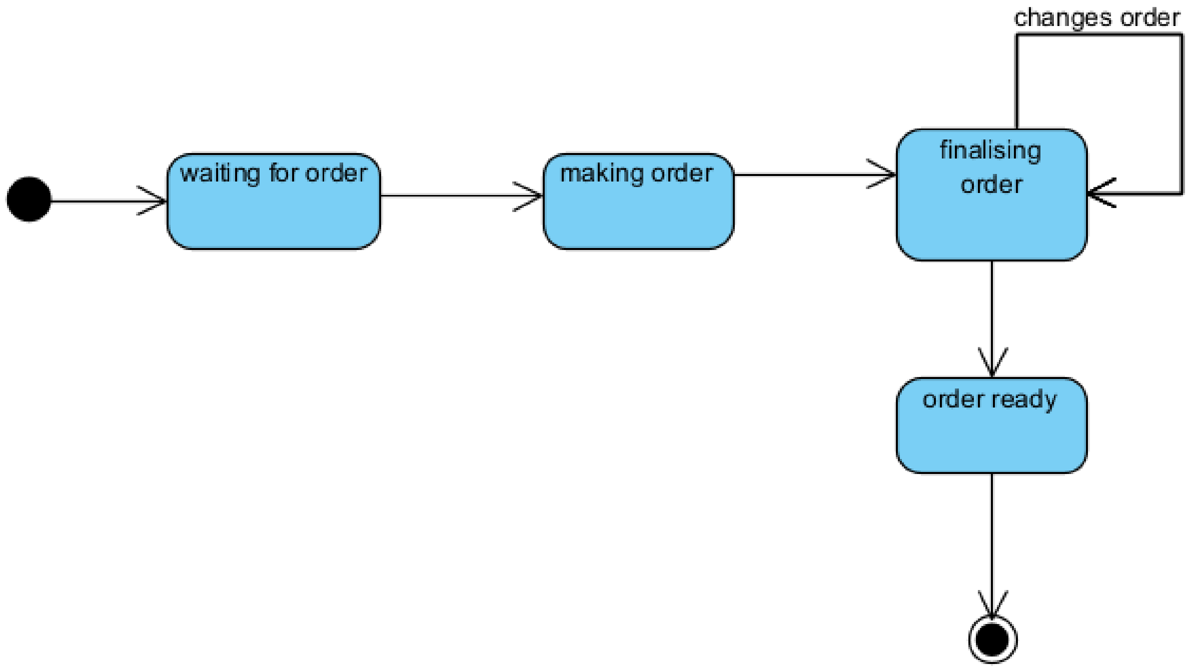
Sequence Diagram

Kitchen-Floor



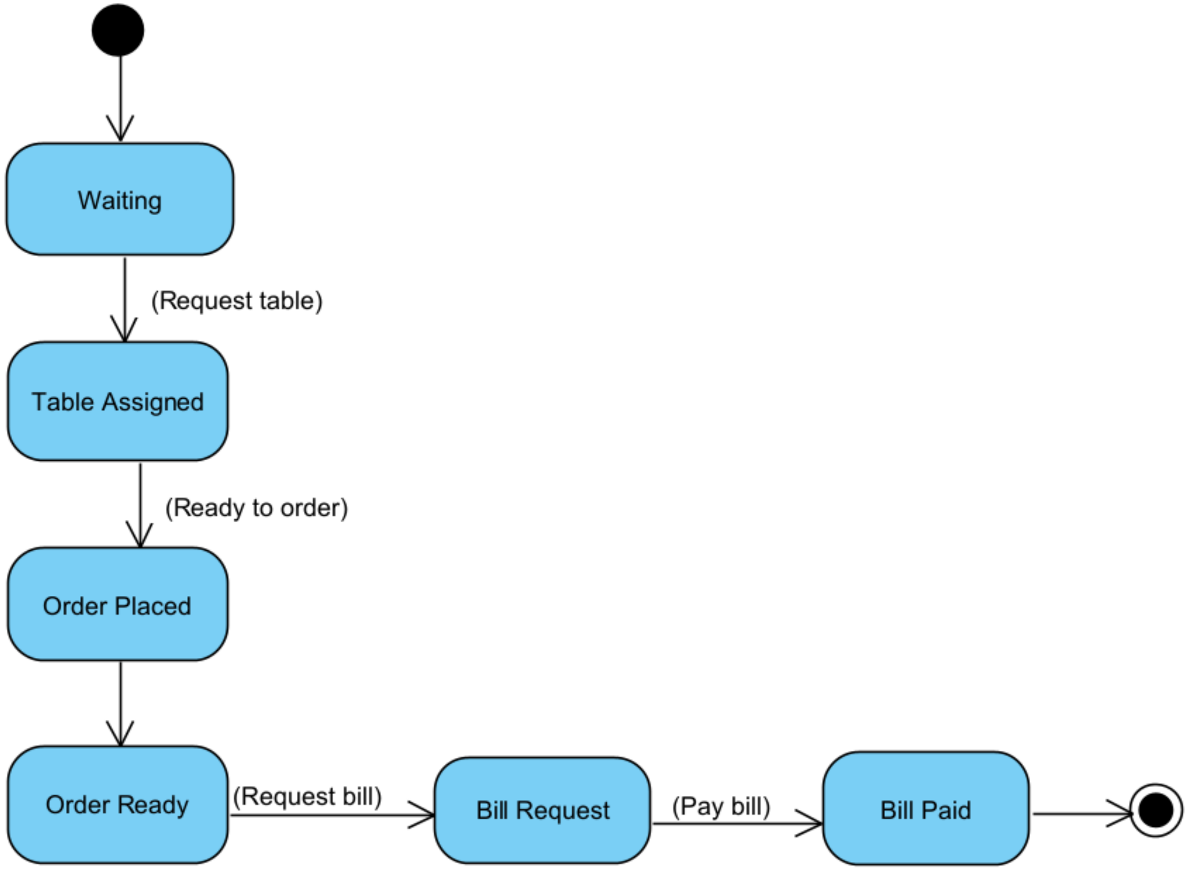
State Diagram

Chef



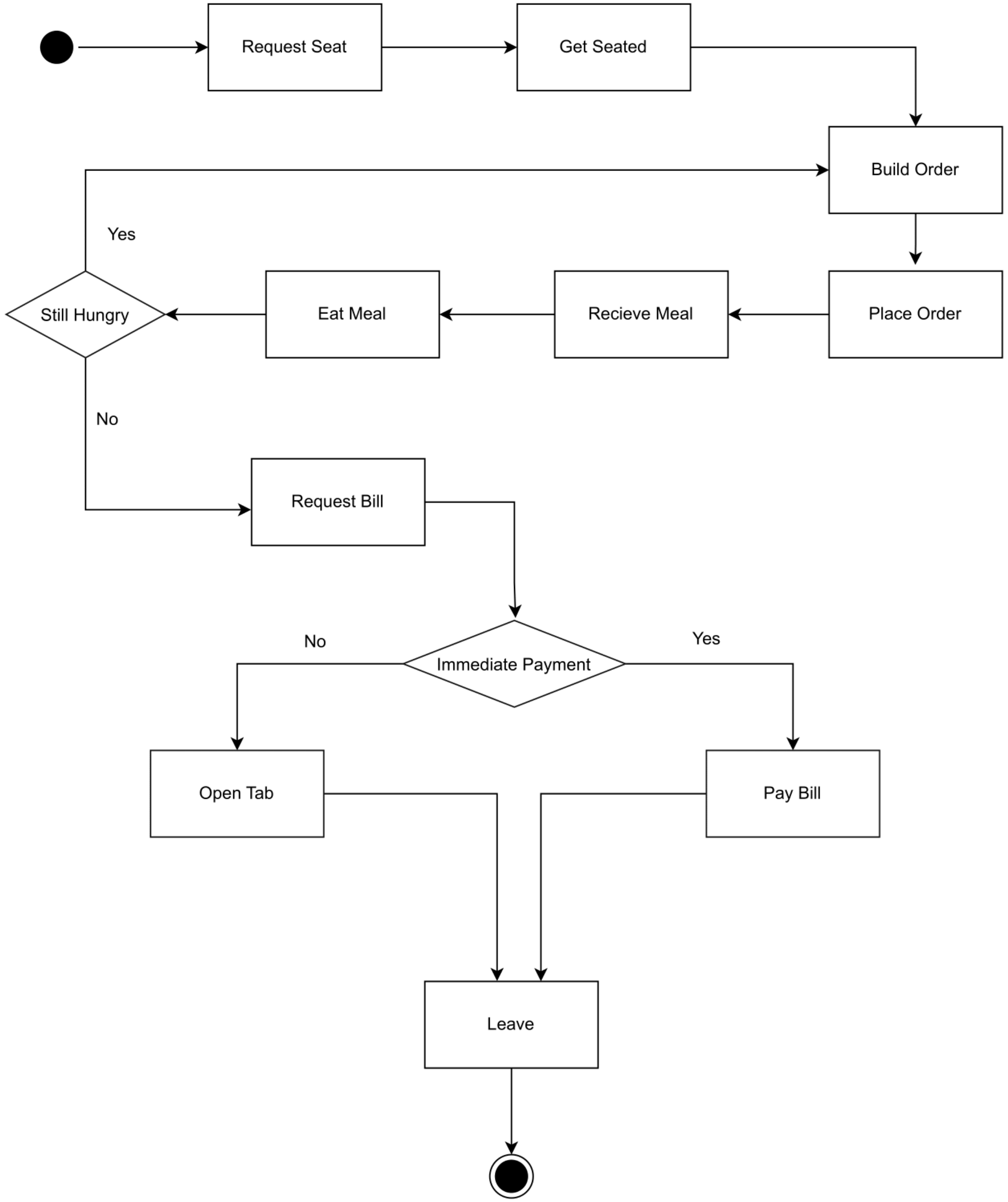
State Diagram

Customer



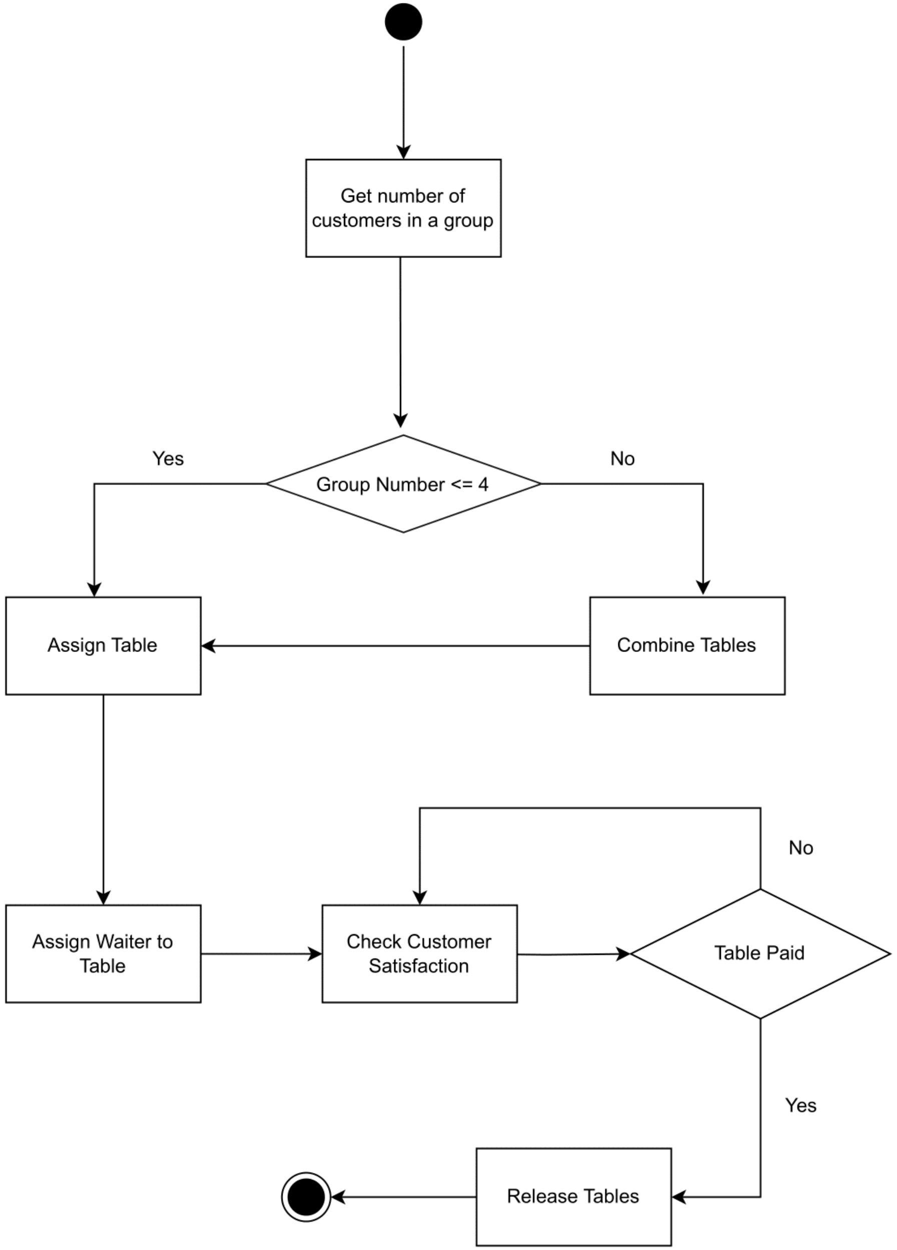
Activity Diagram

Customer



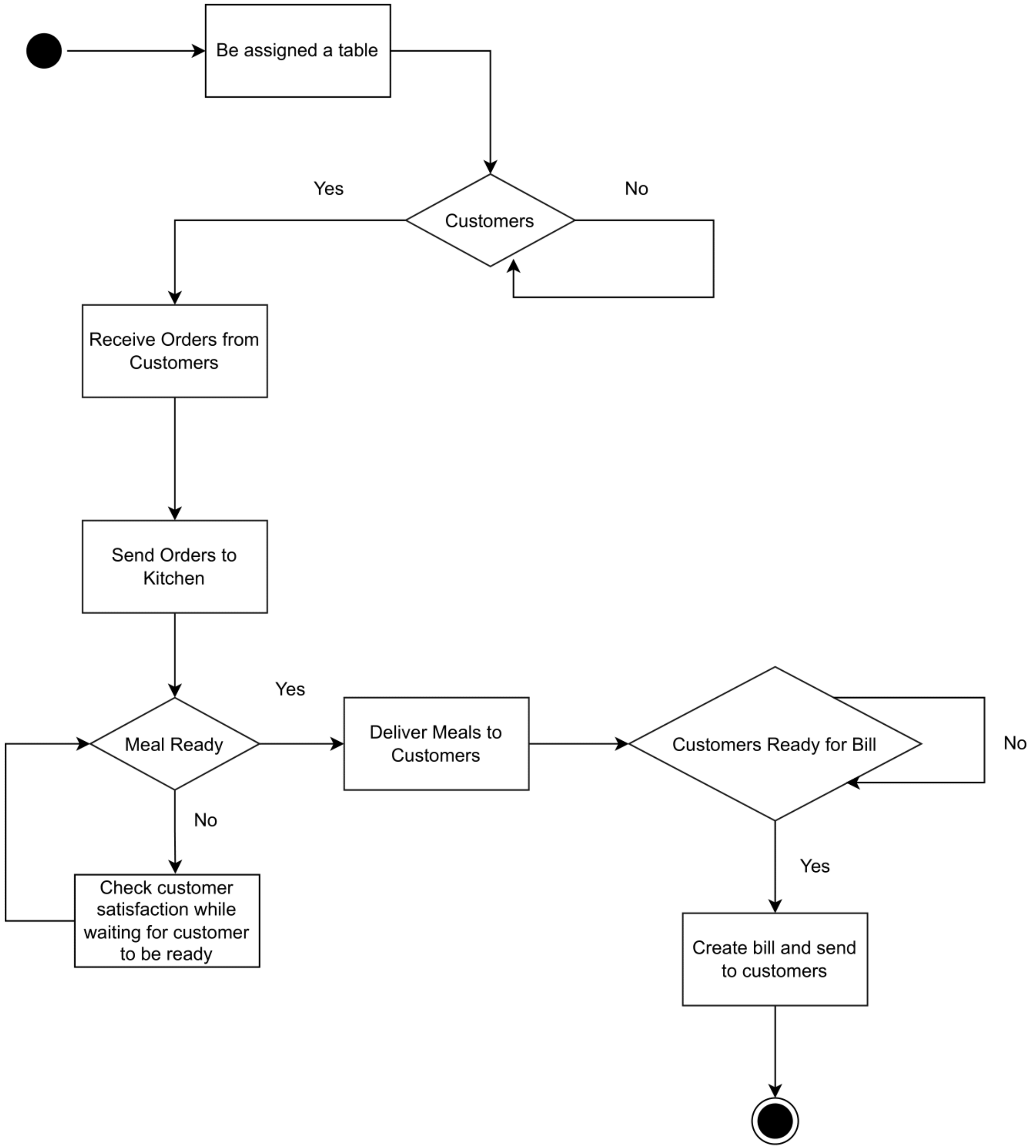
Activity Diagram

Manager



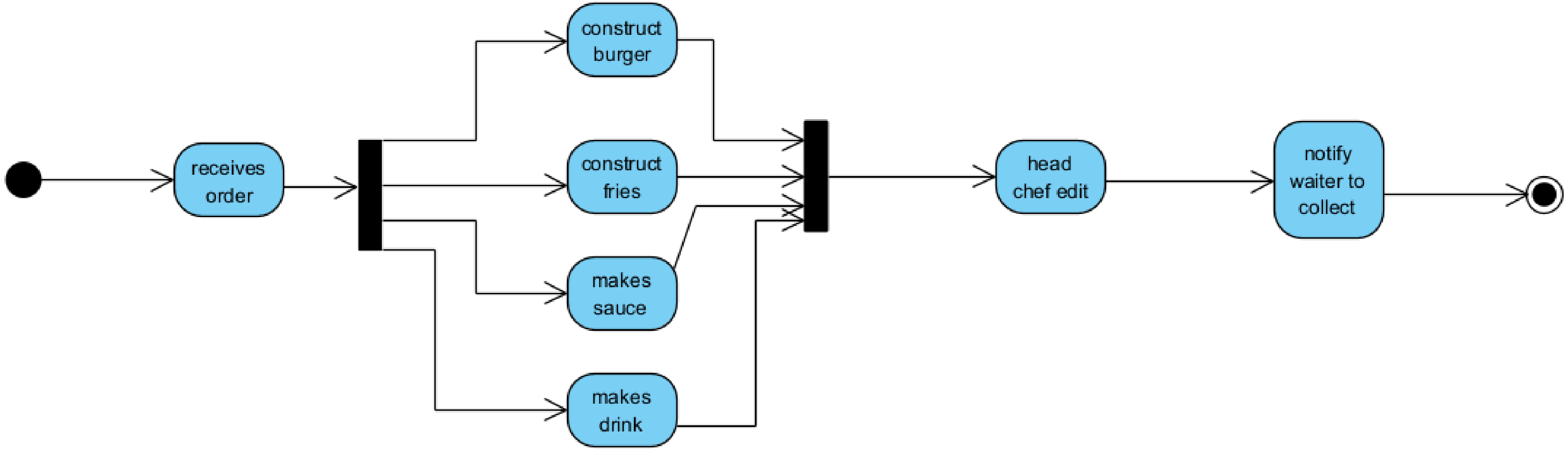
Activity Diagram

Waiter


Be assigned a table
Yes
Customers
No
Receive Orders from Customers
Send Orders to Kitchen
Yes
Meal Ready
Deliver Meals to Customers
Customers Ready for Bill
No
No
Check customer satisfaction while waiting for customer to be ready
Yes
Create bill and send to customers

Activity Diagram

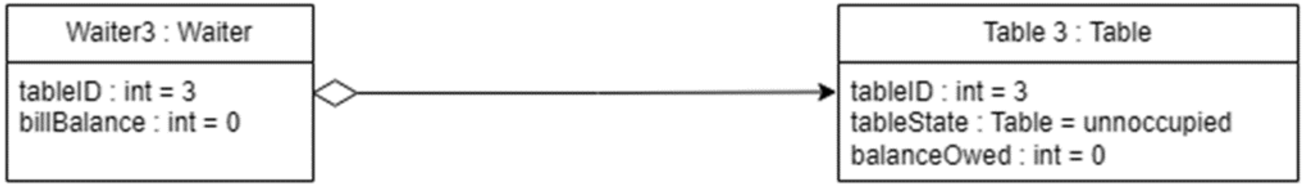
Chef



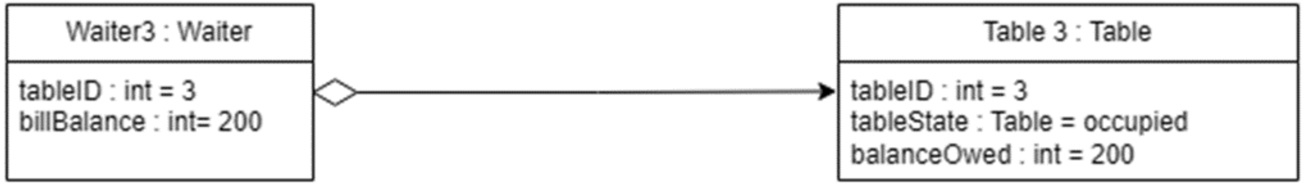
Object Diagram

Waiter

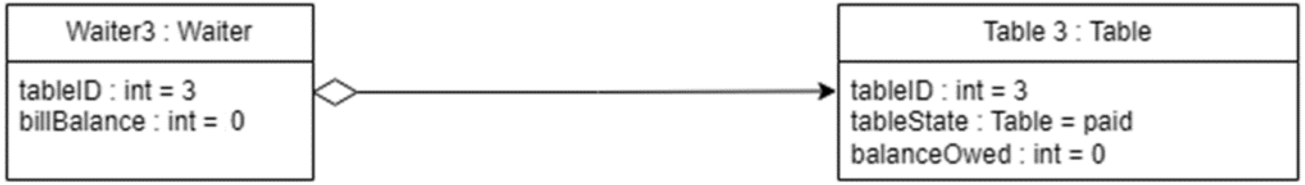
Waiter-Table Object (Waiter has been assigned a table, table is not yet occupied)



Waiter-Table Object (Waiter has created bill, table now has to pay)

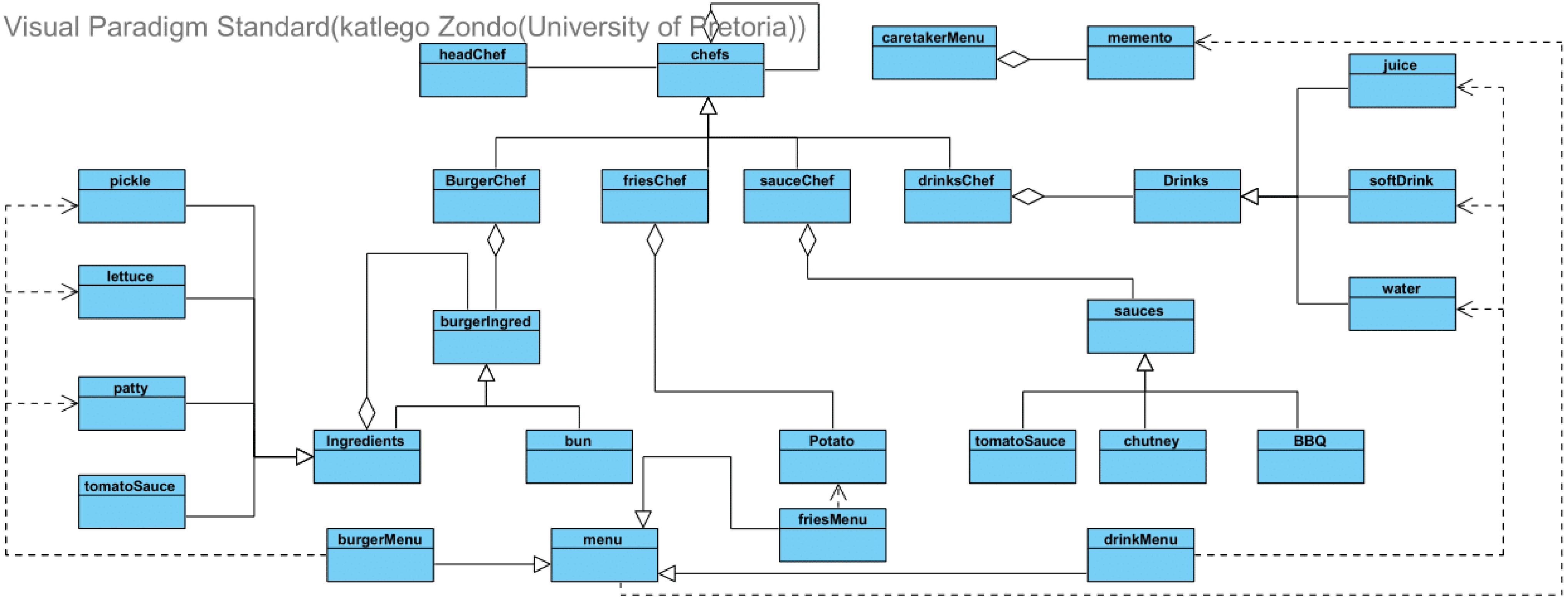


Waiter-Table Object (Waiter bill has been paid, table now has now paid bill)



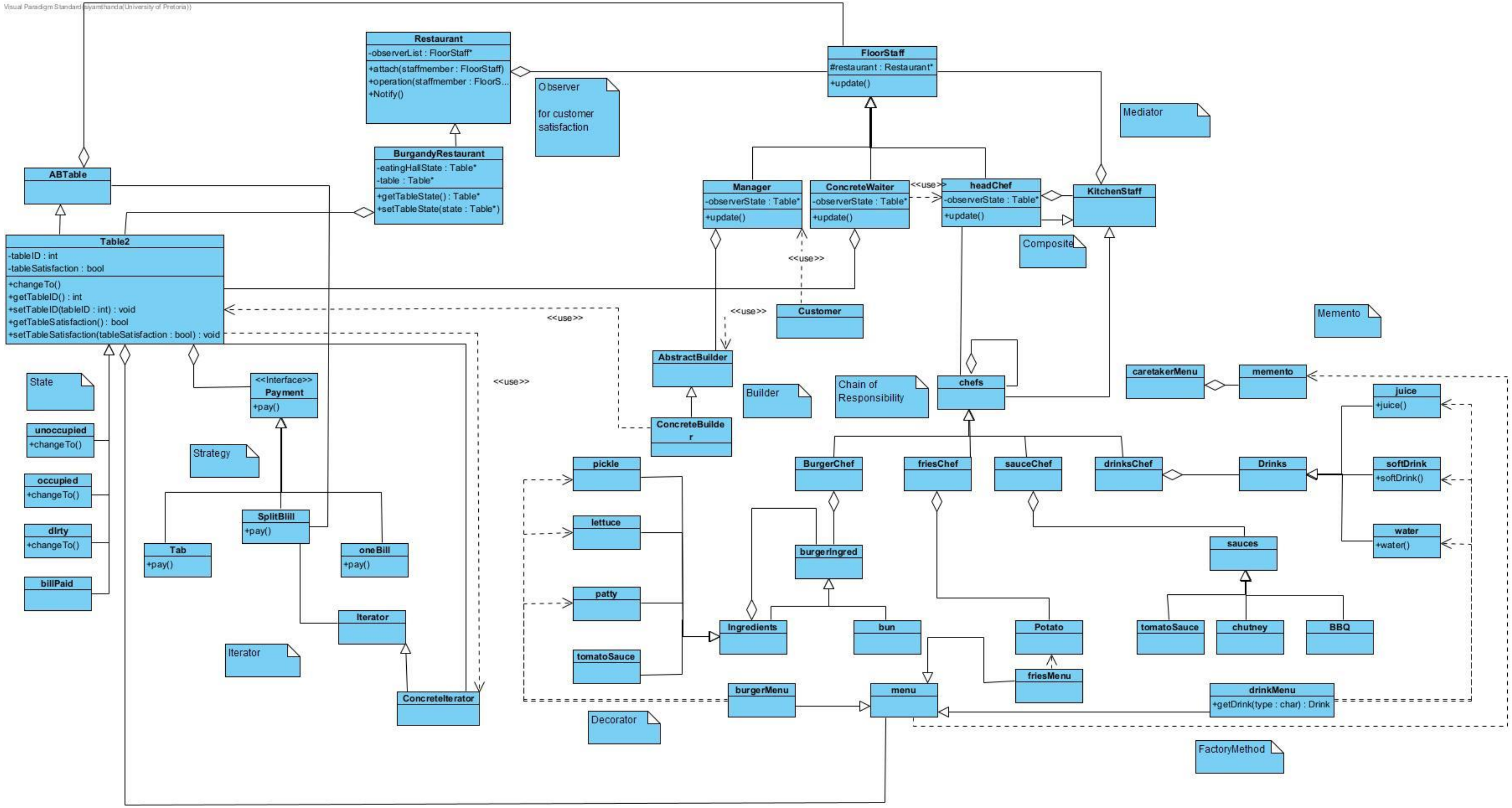
Class Diagram

Chain of Responsibility, Decorator, Factory, Memento Patterns



Class Diagram

Chain of Responsibility, Decorator, Factory, Memento, State, Strategy, Builder, Mediator, Observer, Iterator, Composite Patterns



Visual Paradigm Standard(siyamthanda(University of Pretoria))

Restaurant
-observerList : FloorStaff*
+attach(staffmember : FloorStaff)
+operation(staffmember : FloorS...
+Notify()

Observer

for customer satisfaction

BurgandyRestaurant
-eatingHallState : Table*
-table : Table*
+getTableState() : Table*
+setTableState(state : Table*)

FloorStaff
#restaurant : Restaurant*
+update()

Mediator

ABTable

Table2
-tableID : int
-tableSatisfaction : bool
+changeTo()
+getTableID() : int
+setTableID(tableID : int) : void
+getTableSatisfaction() : bool
+setTableSatisfaction(tableSatisfaction : bool) : void

Manager
-observerState : Table*
+update()

ConcreteWaiter
-observerState : Table*
+update()

<<use>>

headChef
-observerState : Table*
+update()

KitchenStaff

Composite

<<use>>

Customer

<<use>>

<<use>>

<<use>>

AbstractBuilder

ConcreteBuilder

Builder

Chain of Responsibility

chefs

Memento

caretakerMenu

memento

State

unoccupied
+changeTo()

occupied
+changeTo()

dirty
+changeTo()

billPaid

<<Interface>>
Payment
+pay()

Strategy

SplitBlill
+pay()

Tab
+pay()

oneBill
+pay()

Iterator

Iterator

ConcreteIterator

pickle

lettuce

patty

tomatoSauce

BurgerChef

friesChef

sauceChef

drinksChef

Drinks

juice
+juice()

softDrink
+softDrink()

water
+water()

burgerIngred

Ingredients

bun

sauces

Potato

tomatoSauce

chutney

BBQ

friesMenu

burgerMenu

menu

drinkMenu
+getDrink(type : char) : Drink

Decorator

FactoryMethod