Burgundy

COS 214 Final Project

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

Burgundy

System Requirements

Customer should be able to :

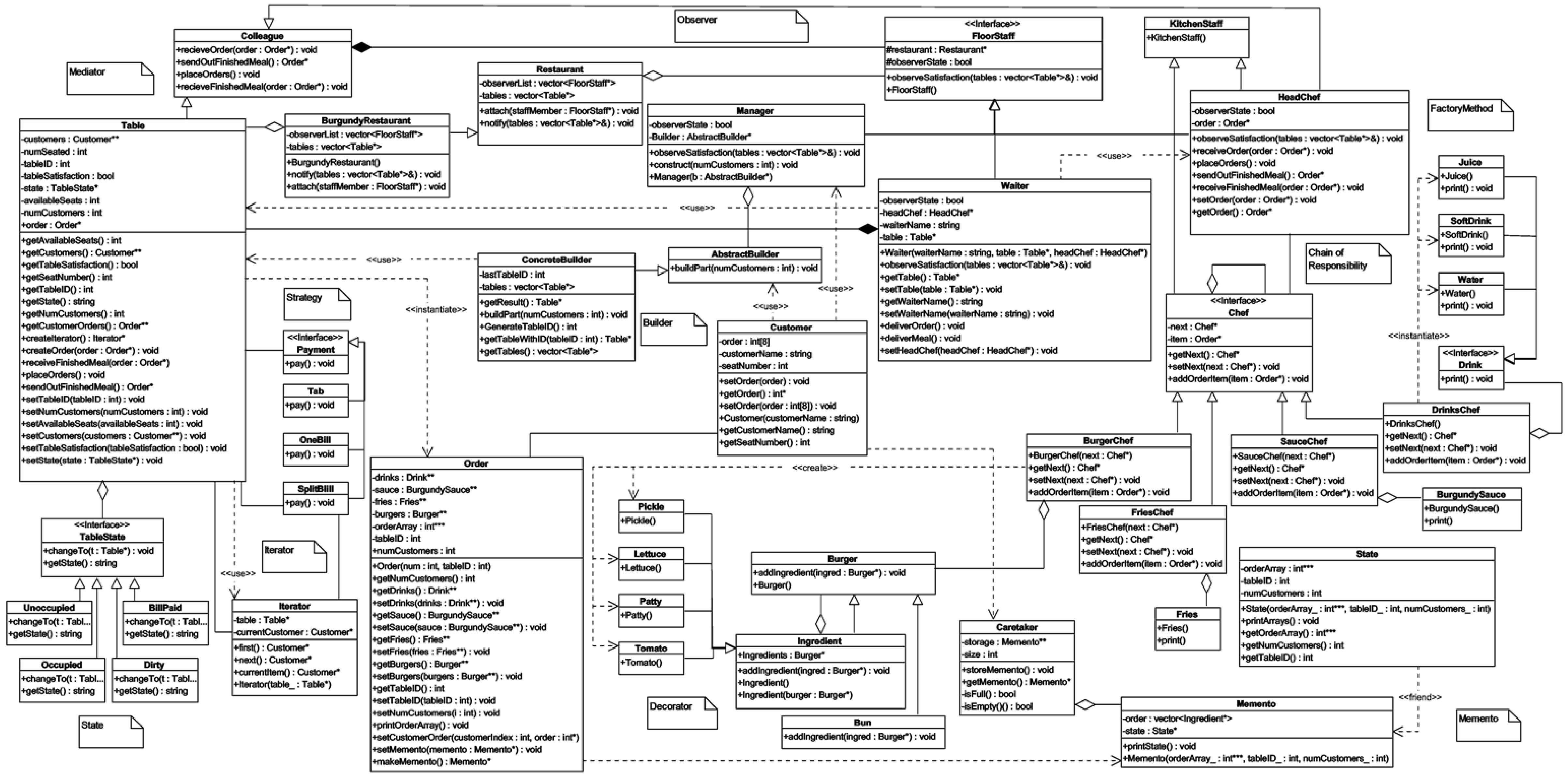
- Walk in an request to be seated
- Be assigned to a table
- Create an order and then place it by sending it to the waiter
- Specify ingredients and remove or add to the order prior to it being sent to the kitchen
- Tell a waiter to come back later if they are not ready to order
- Split the bill or pay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

Burgundy

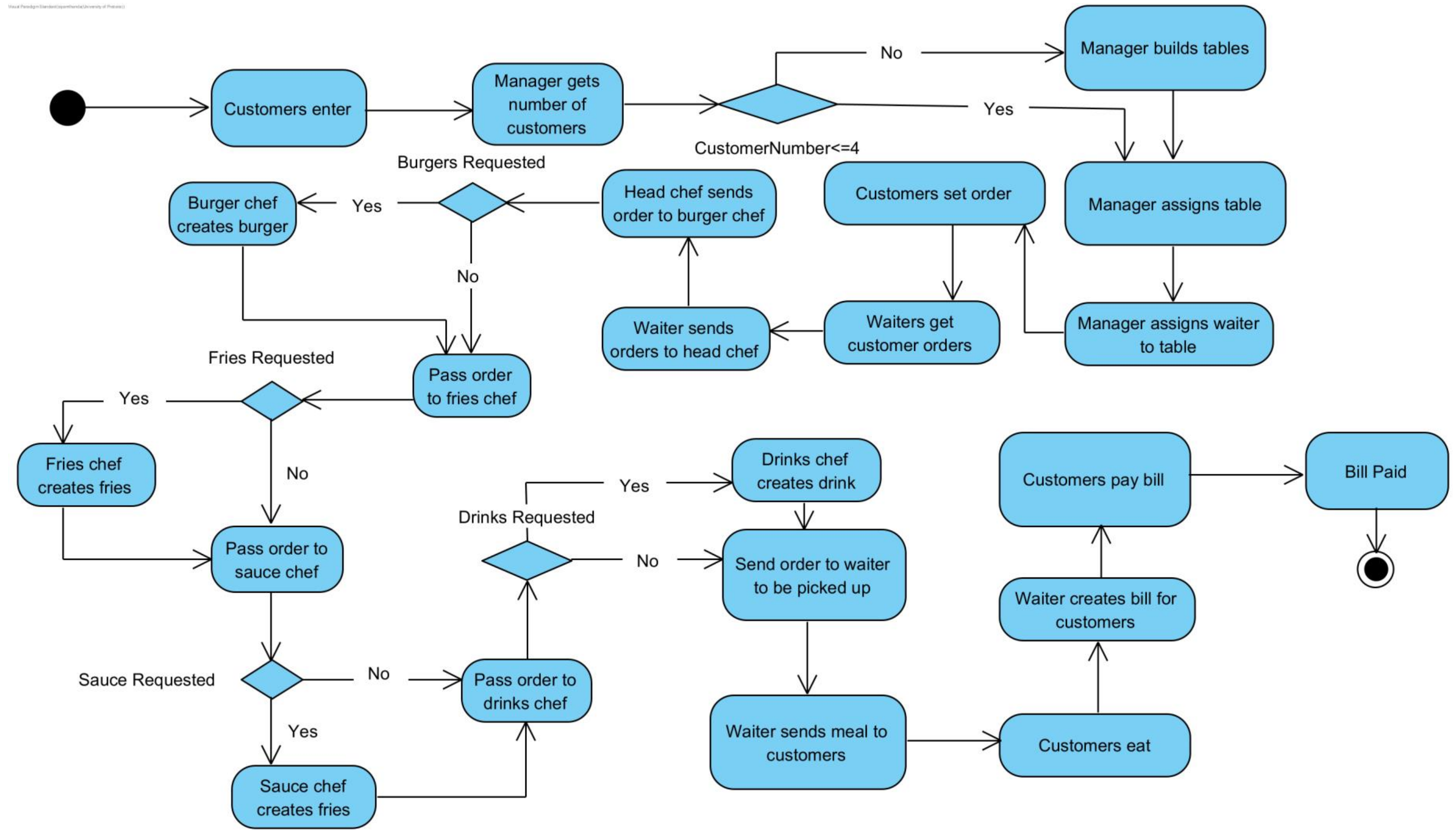
Design Patterns

- Mediator
- State
- Strategy
- Chain of Responsibility
- Memento
- Observer
- Builder
- Decorator
- Iterator
- Factory Method

Burgundy Class Diagram


Observer
Mediator
Colleague
+recieveOrder(order : Order*) : void
+sendOutFinishedMeal() : Order*
+placeOrders() : void
+recieveFinishedMeal(order : Order*) : void
Restaurant
-observerList : vector<FloorStaff*>
-tables : vector<Table*>
+attach(staffMember : FloorStaff*) : void
+notify(tables : vector<Table*>&) : void
<<Interface>>
FloorStaff
#restaurant : Restaurant*
#observerState : bool
+observeSatisfaction(tables : vector<Table*>&) : void
+FloorStaff()
KitchenStaff
+KitchenStaff()
Table
-customers : Customer**
-numSeated : int
-tableID : int
-tableSatisfaction : bool
-state : TableState*
-availableSeats : int
-numCustomers : int
+order : Order*
+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
Manager
-observerState : bool
-Builder : AbstractBuilder*
+observeSatisfaction(tables : vector<Table*>&) : void
+construct(numCustomers : int) : void
+Manager(b : AbstractBuilder*)
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*
FactoryMethod
<<use>>
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
<<use>>
Juice
+Juice()
+print() : void
SoftDrink
+SoftDrink()
+print() : void
Water
+Water()
+print() : void
<<instantiate>>
<<Interface>>
Drink
+print() : void
Chain of Responsibility
<<Interface>>
Chef
-next : Chef*
-item : Order*
+getNext() : Chef*
+setNext(next : Chef*) : void
+addOrderItem(item : Order*) : void
<<use>>
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
<<use>>
<<use>>
Builder
Strategy
<<instantiate>>
<<Interface>>
Payment
+pay() : void
Tab
+pay() : void
OneBill
+pay() : void
SplitBilll
+pay() : 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
DrinksChef
+DrinksChef()
+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
BurgundySauce
+BurgundySauce()
+print()
<<create>>
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*
<<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
State
Iterator
<<use>>
Iterator
-table : Table*
-currentCustomer : Customer*
+first() : Customer*
+next() : Customer*
+currentItem() : Customer*
+Iterator(table_ : Table*)
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
Decorator
FriesChef
+FriesChef(next : Chef*)
+getNext() : Chef*
+setNext(next : Chef*) : void
+addOrderItem(item : Order*) : void
Fries
+Fries()
+print()
State
-orderArray : int***
-tableID : int
-numCustomers : int
+State(orderArray_ : int***, tableID_ : int, numCustomers_ : int)
+printArrays() : void
+getOrderArray() : int***
+getNumCustomers() : int
+getTableID() : int
Caretaker
-storage : Memento**
-size : int
+storeMemento() : void
+getMemento() : Memento*
-isFull() : bool
-isEmpty()() : bool
<<friend>>
Memento
-order : vector<Ingredient*>
-state : State*
+printState() : void
+Memento(orderArray_ : int***, tableID_ : int, numCustomers_ : int)
Memento

Burgundy Activity Diagram



Customers enter
Manager gets number of customers
No
Manager builds tables
Yes
CustomerNumber<=4
Manager assigns table
Manager assigns waiter to table
Customers set order
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
Yes
Sauce chef creates fries
No
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

Burgundy Activity Diagram

Table of Contents

Content	Page
Research	3
Design Decisions	7
Design Patterns	15
Design Alterations	19
UML Diagrams	23
Final UML Diagrams	24
Initial Design UML Diagrams (Task 2)	29

Burgundy

COS 214 Final Project Report

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

Functional Requirements & Design Patterns

Functional Requirements & Design Patterns

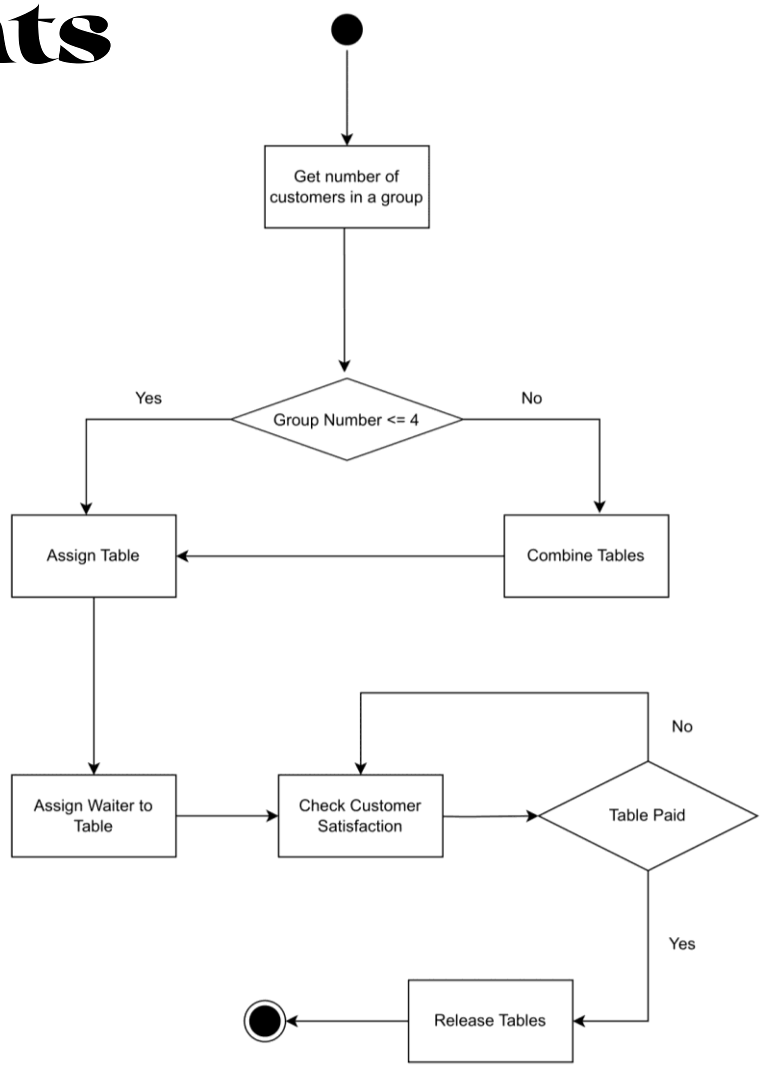
The system requirements and the corresponding design patterns used

System Requirements

Manager

Manager should be able to :

- Assign tables for customers
- Build tables which can hold a maximum of four customers
- Check on customer satisfaction



Builder

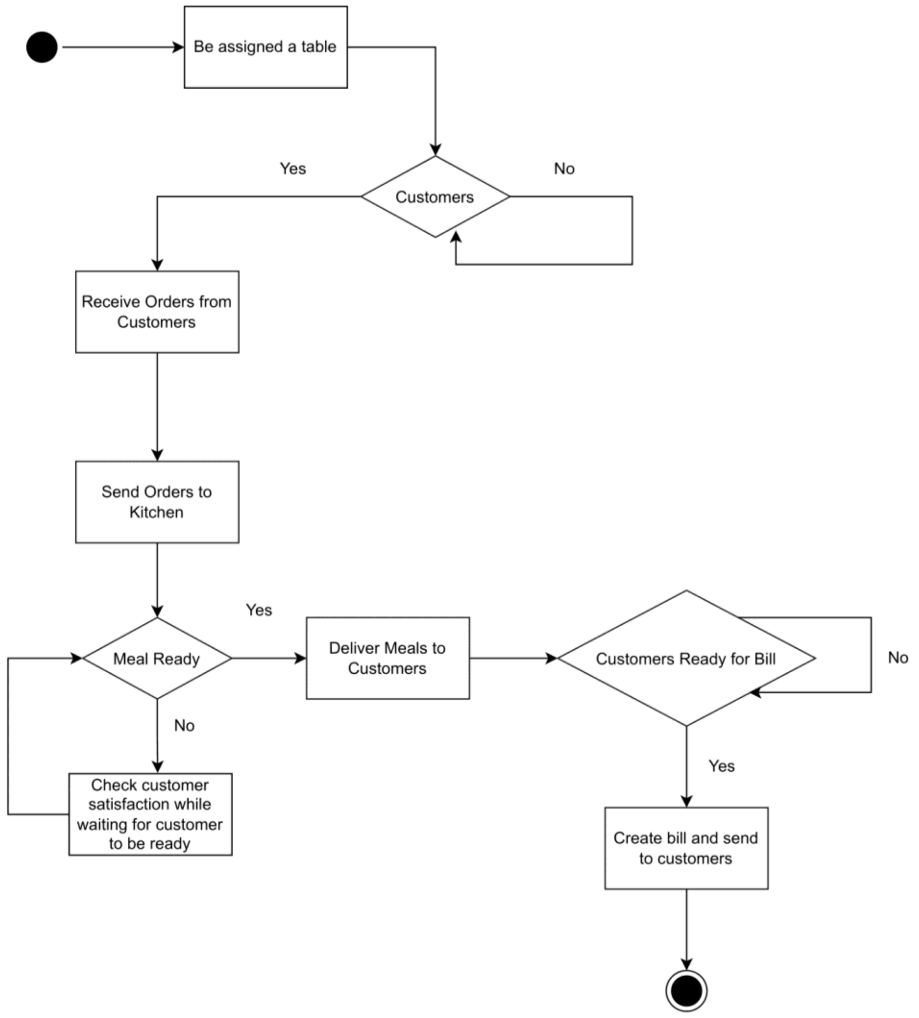
- Each table in our restaurant can seat four customers
- Should a group of customers be larger than 4 then the manager will have to combine them
- Using the builder pattern, we can separate the construction of the table object from its representation so that the same construction process can create different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

System Requirements

Waiter

Waiter should be able to :

- Be assigned a 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
- Observe customer satisfaction for the customers at their table


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

Mediator

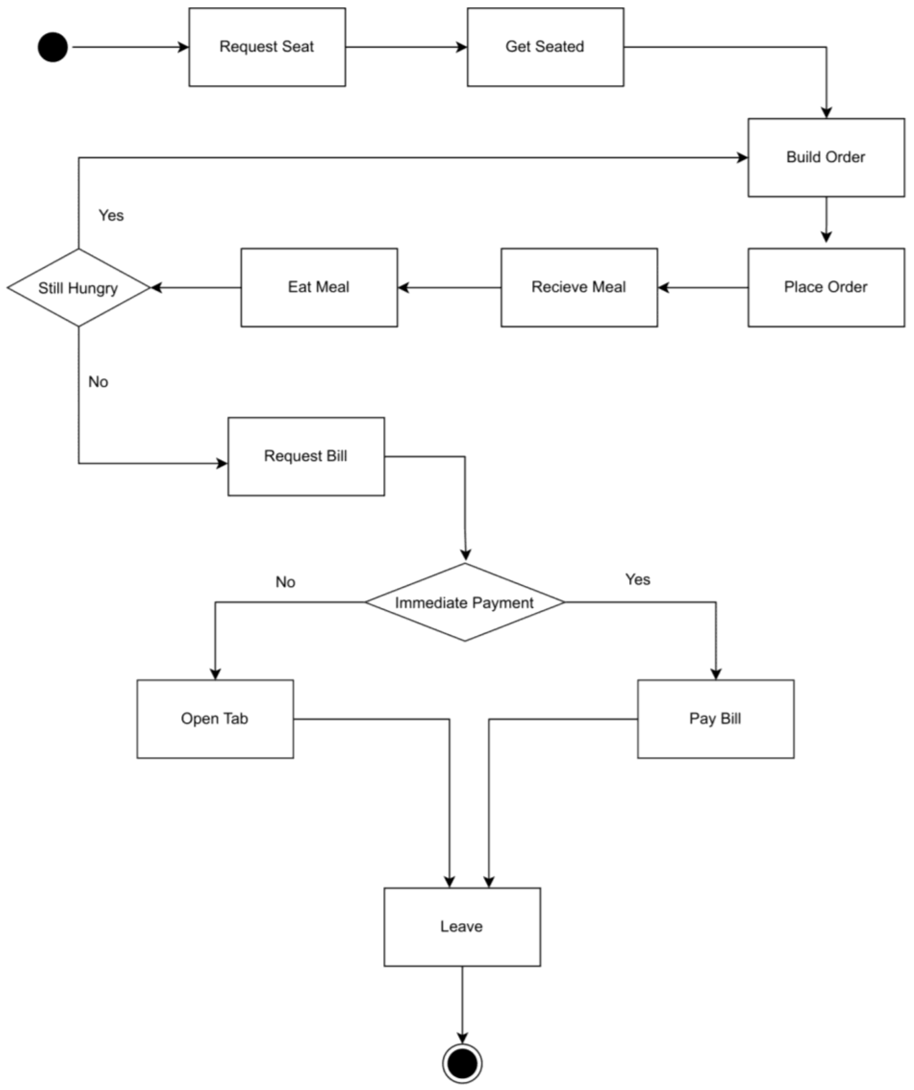
- The waiter has to facilitate communication between the head chef and the table by taking customer’s meals to and from the kitchen
- Using the mediator pattern promotes loose coupling since it keeps the table and head chef objects from referring to each other explicitly, and it lets you vary the interaction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

System Requirements

Customer

Customer should be able to :

- 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
- Split the bill or pay it in full
- Decide on payment method (by tab or pay immediately)
- Receive their order and eat it



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

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.

Iterator

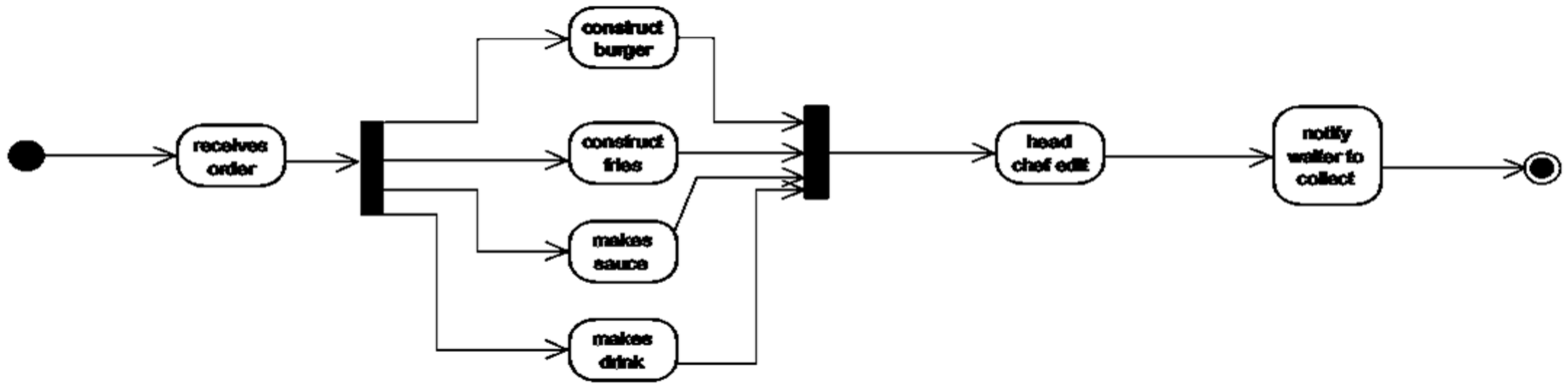
- When the meals will be paid for by splitting the bill, we iterate through the customers object sequentially without exposing its underlaying representation

System Requirements

Chef

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



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 Method

- 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

State

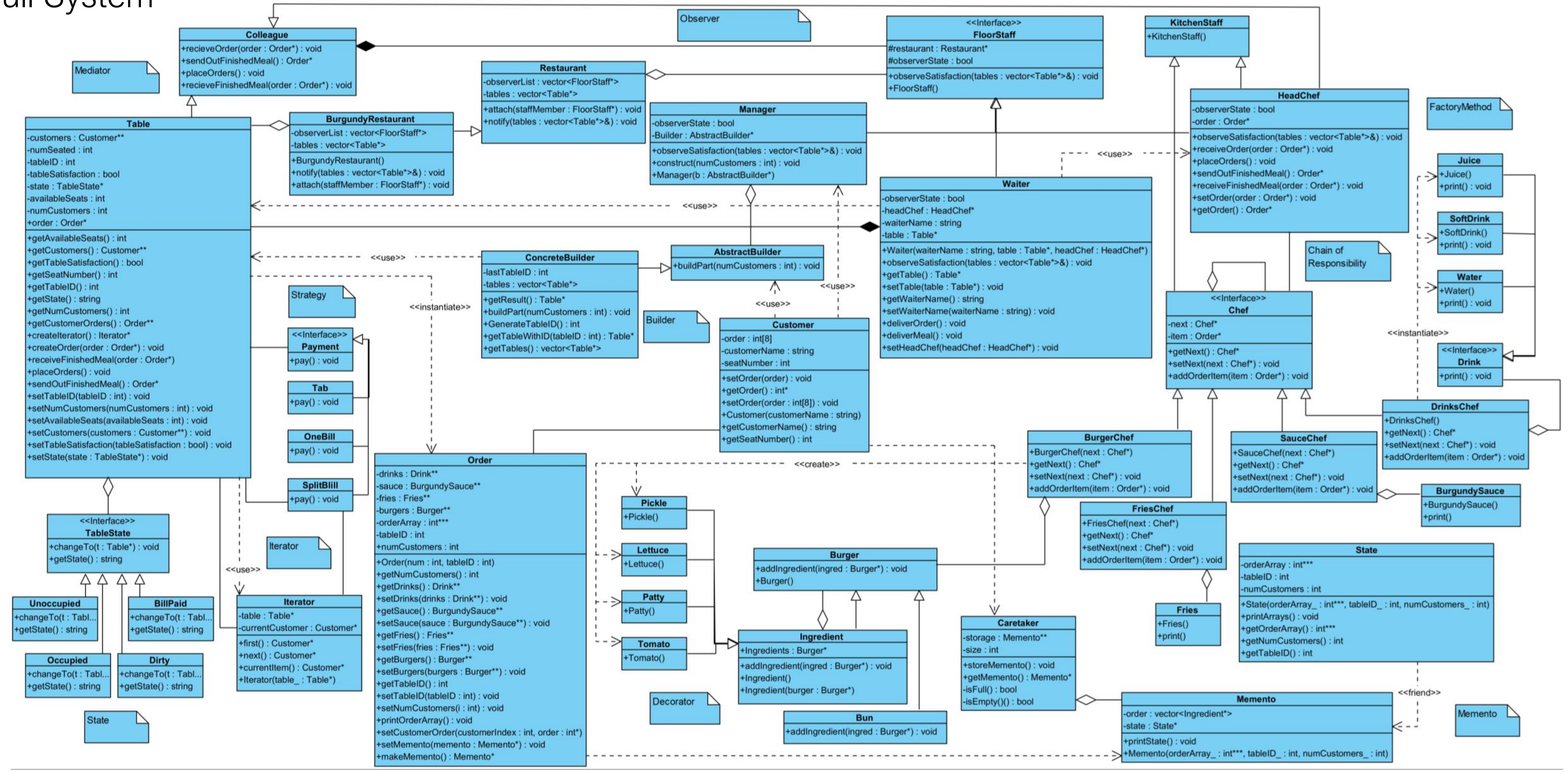
- Over time the each table may be in different states
- A table may be occupied, unoccupied, dirty or have its bill paid
- We want a table to be able to alter its behaviour depending on what its internal state is.
- The state pattern is used for exactly this.

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

Final Class Diagram

Full System


Mediator
Colleague
+recieveOrder(order : Order*) : void
+sendOutFinishedMeal() : Order*
+placeOrders() : void
+recieveFinishedMeal(order : Order*) : void
Observer
Restaurant
-observerList : vector<FloorStaff*>
-tables : vector<Table*>
+attach(staffMember : FloorStaff*) : void
+notify(tables : vector<Table*>&) : void
<<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*
FactoryMethod
Table
-customers : Customer**
-numSeated : int
-tableID : int
-tableSatisfaction : bool
-state : TableState*
-availableSeats : int
-numCustomers : int
+order : Order*
+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
Manager
-observerState : bool
-Builder : AbstractBuilder*
+observeSatisfaction(tables : vector<Table*>&) : void
+construct(numCustomers : int) : void
+Manager(b : AbstractBuilder*)
<<use>>
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
<<use>>
<<use>>
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
<<use>>
<<use>>
Chain of Responsibility
Juice
+Juice()
+print() : void
SoftDrink
+SoftDrink()
+print() : void
Water
+Water()
+print() : void
<<instantiate>>
<<Interface>>
Drink
+print() : void
Strategy
<<instantiate>>
Builder
<<Interface>>
Payment
+pay() : void
Tab
+pay() : void
OneBill
+pay() : void
SplitBlill
+pay() : 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>>
Chef
-next : Chef*
-item : Order*
+getNext() : Chef*
+setNext(next : Chef*) : void
+addOrderItem(item : Order*) : void
DrinksChef
+DrinksChef()
+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
BurgundySauce
+BurgundySauce()
+print()
<<create>>
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()
FriesChef
+FriesChef(next : Chef*)
+getNext() : Chef*
+setNext(next : Chef*) : void
+addOrderItem(item : Order*) : void
State
-orderArray : int***
-tableID : int
-numCustomers : int
+State(orderArray_ : int***, tableID_ : int, numCustomers_ : int)
+printArrays() : void
+getOrderArray() : int***
+getNumCustomers() : int
+getTableID() : int
<<Interface>>
TableState
+changeTo(t : Table*) : void
+getState() : string
Iterator
<<use>>
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*)
State
Ingredient
+Ingredients : Burger*
+addIngredient(ingred : Burger*) : void
+Ingredient()
+Ingredient(burger : Burger*)
Decorator
Bun
+addIngredient(ingred : Burger*) : void
Caretaker
-storage : Memento**
-size : int
+storeMemento() : void
+getMemento() : Memento*
-isFull() : bool
-isEmpty()() : bool
Fries
+Fries()
+print()
<<friend>>
Memento
-order : vector<Ingredient*>
-state : State*
+printState() : void
+Memento(orderArray_ : int***, tableID_ : int, numCustomers_ : int)
Memento

Git & Coding Standards

The coding standards using in implementation and the workflows we used with git

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).
- 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/