2017

Zest-Ware – Report 2 Part 1

ALEXANDER DEWEY | AMA FREEMAN | DWAYNE ANTHONY
FAHD HUMAYUN | NATHAN MORGENSTERN | RAPHAELLE MARCIAL
SHEHPAR SOHAIL

IVAN MARSIC | *Professor – Software Engineering*

Table of Contents

Individual Contributions Breakdown

Part 1

Subgroup Members	***Customer***			***Manager***		***Kitchen***	
	Nathan M.	***Shehpar S.***	***Fahd H.***	***Raphaelle M.***	***Ama F.***	***Alex D.***	***Dwanye A.***
Interaction Diagrams	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*

Part 2

Subgroup Members		Customer			Manager		Kitchen	
		Nathan M.	*Shehpar S.*	*Fahd H.*	*Raphaelle M.*	*Ama F.*	*Alex D.*	*Dwanye A.*
Class Diagrams & Specifications	*Class Diagrams & Descriptions*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*
	Signatures	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*
	Traceability Matrix	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*
System Architecture & System Design	*Architectural Styles*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*
	Package Diagram	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*
	Mapping to Hardware	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*
	Global Control Flow	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*
	Hardware Requirements	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*

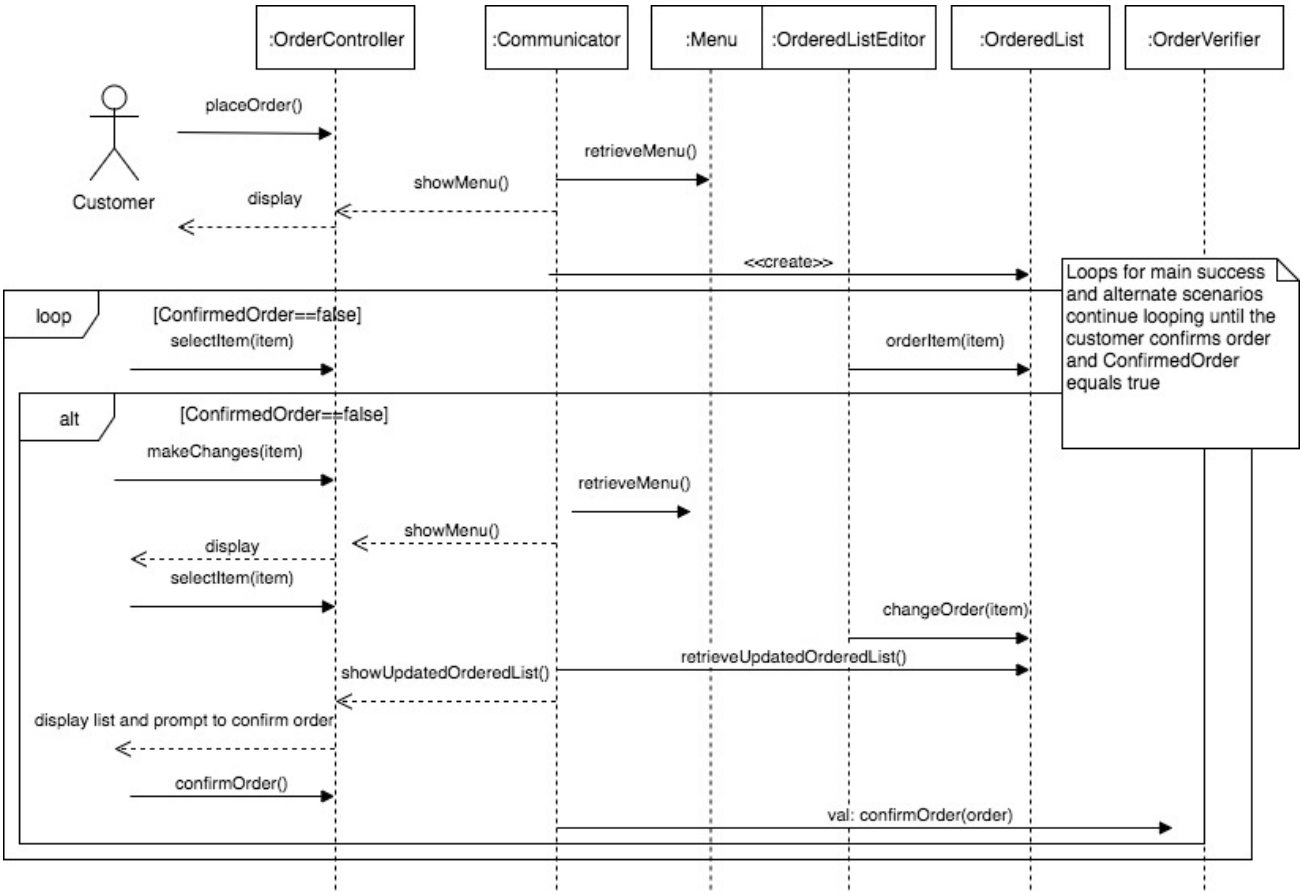
Part 3

Subgroup Members		***Customer***			***Manager***		***Kitchen***	
		Nathan M.	***Shehpar S.***	***Fahd H.***	***Raphaelle M.***	***Ama F.***	***Dwanye A.***	***Alex D.***
Akgorithms and Data Structures	***Algorithms***	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*
	Data Structures	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*
User Interface Design and Implementation		*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*
Design of Tests	***Listing Test Cases***	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*
	Test Coverage of tests	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*
	Integration Testing Strategy	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*	*14.29 %*

Part 1: Interaction Diagrams

Customer

Use Case: OrderItems (UC – 23)

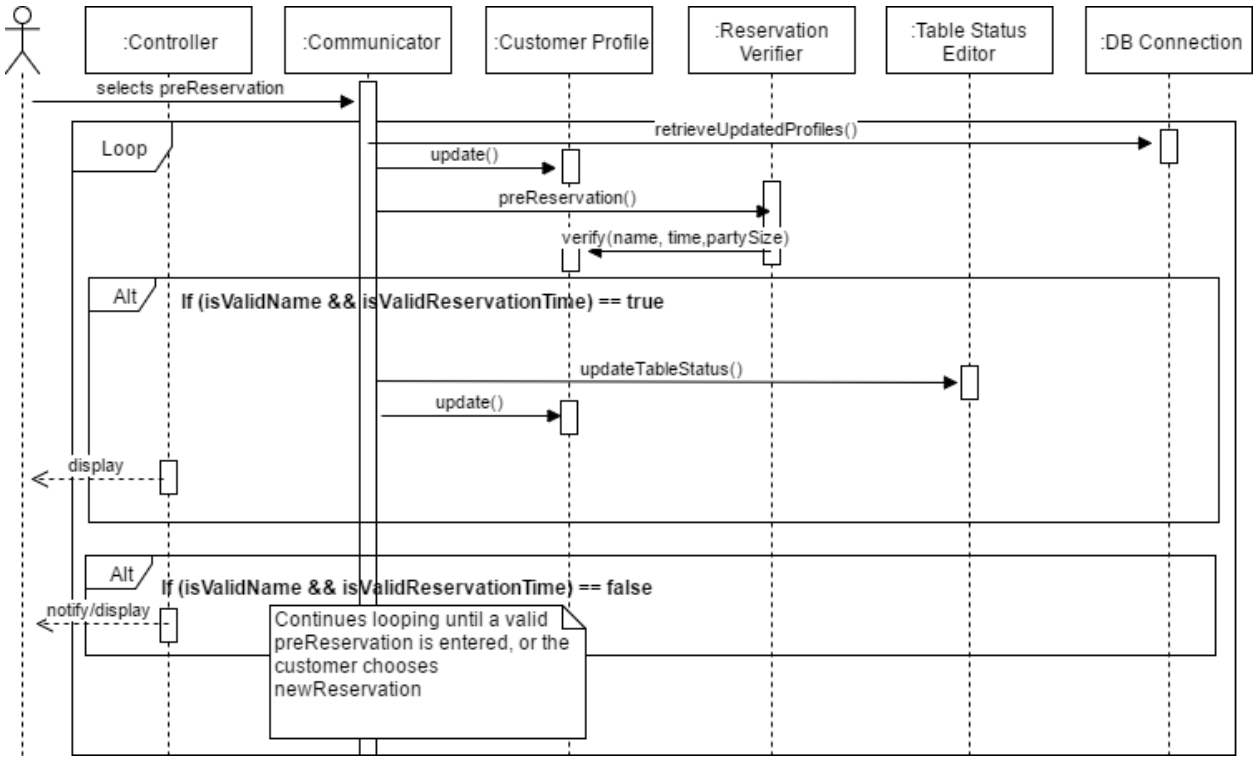


Sequence diagram for the customer placing an order on the CTS

A sequence diagram that represents when a customer wants to place an order. The Communicator first retrieves the menu through the $retrieveMenu()$ method. The Controller then displays the menu to the customer. The Communicator creates an ordered list, where all the items ordered by the customer will be listed. The customer selects the item they want to order. The Ordered List Editor lists that item to the ordered list through the $orderItem(item)$ method. This process continues until the customer is satisfied and confirms their order listed on the Ordered List. When the customer confirms their order, $ConfirmedOrder$ will equal true and the order will be placed.An alternate scenario will be

when the customer wants to make changes to the items listed in the ordered list. The Communicator will again retrieve the menu and the Controller will display the menu to the customer. The customer will select the item they want to add/remove. The Ordered List Editor will change the order accordingly on the Ordered List through the $changeOrder(item)$ method. The Communicator will retrieve the updated list through the $retrieveUpdatedOrderedList()$ method. The Controller will display the Ordered List to the customer and prompt the customer to confirm their order. This process will continue until the customer confirms their order and $ConfirmOrder$ equals true.

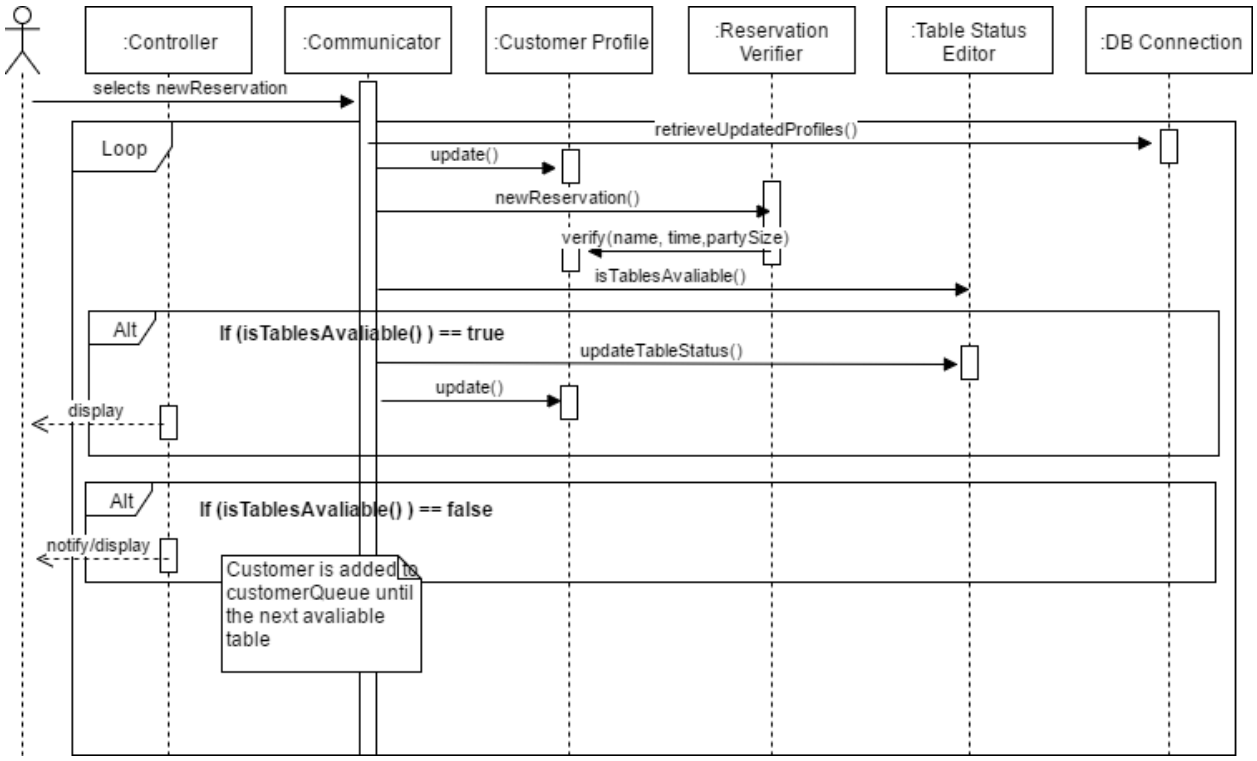
Use Case: PrePreservation(UC-16)



Sequence diagram for the customer selecting pre-reservation option on the CWS

A sequence diagram that executes when the customer selects pre-reservation. In the loop, the Communicator constantly refreshes and get the update profiles by calling the $retrieveUpdateProfiles()$ on the DB Connection. The communicator then calls $preReservation()$ on the Reservation Verifier. The Reservation Verifier calls $verify()$ with the three parameters for *name, time, and party size.* If the reservation is valid the Communicator calls $updateTableStatus()$ on the Table Status Editor and the Controller shows/displays the status to the customer. Alternatively, if the reservation is invalid, it continues to loop until a valid reservation is entered, or new-reservation (by tapping back) is selected.

Use Case: NewReservation(UC-17)

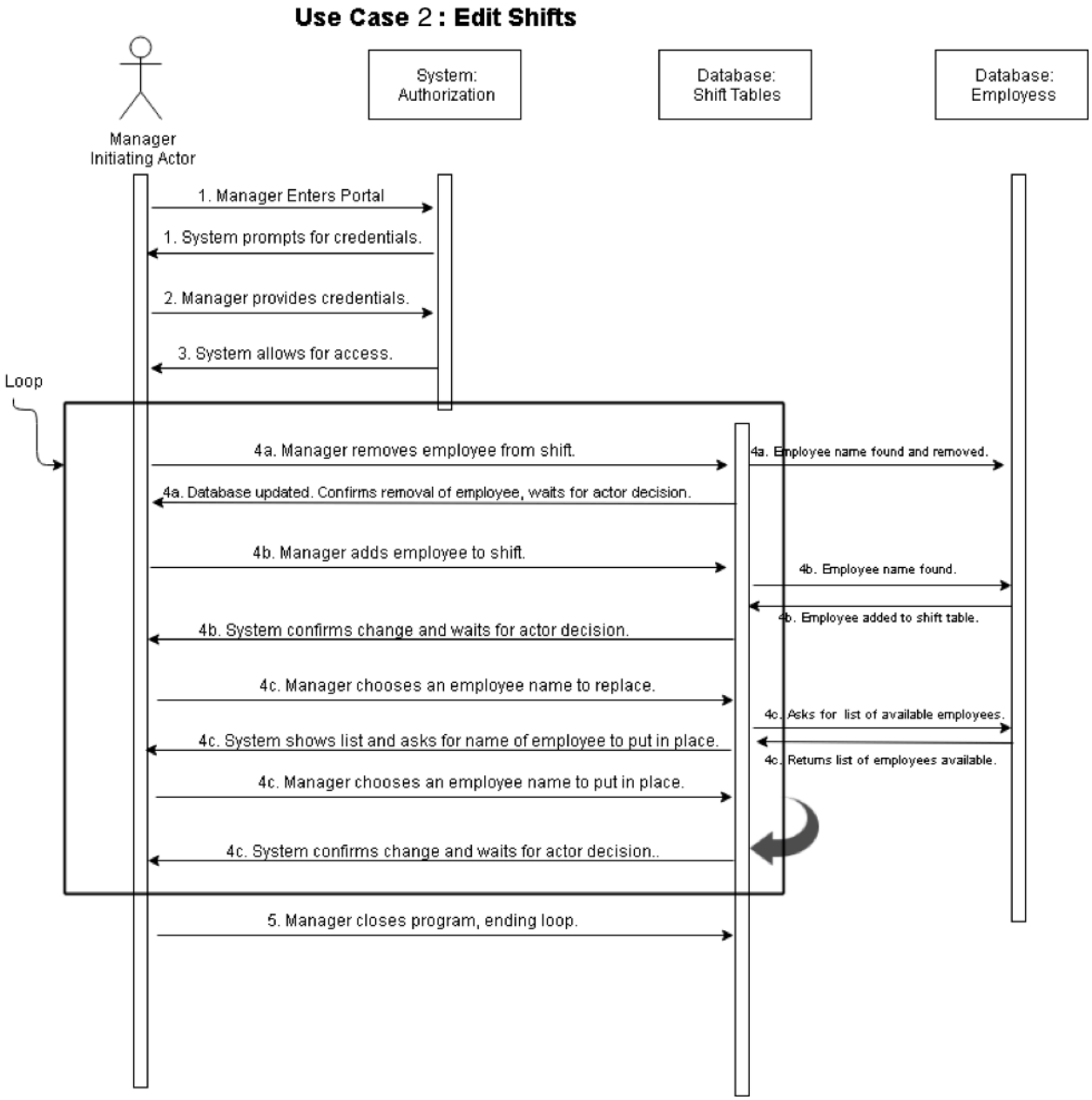


Sequence diagram for the customer selecting new reservation option on the CWS

A sequence diagram that executes when the customer selects new-reservation. In the loop, the Communicator constantly refreshes and get the update profiles by calling the $retrieveUpdateProfiles()$ on the DB Connection. The Communicator then calls $newReservation()$ on the Reservation Verifier. The Reservation Verifier calls $verify()$ with the three parameters for *name, time, and party size.* The Communicator then calls $isTableAvailable()$ on the Table Status Editor to check the availability of the table. If the reservation is valid the Communicator calls $updateTableStatus()$ on the Table Status Editor. If the table is available, then the method $isTableAvailable()$ will return true, and the Communicator will update the status by calling the method $updateTableStatus()$ on Table Status Editor. If the table is not available, then the method $isTableAvailable()$ will return false and the customer will be added to the queue for the availability of table.

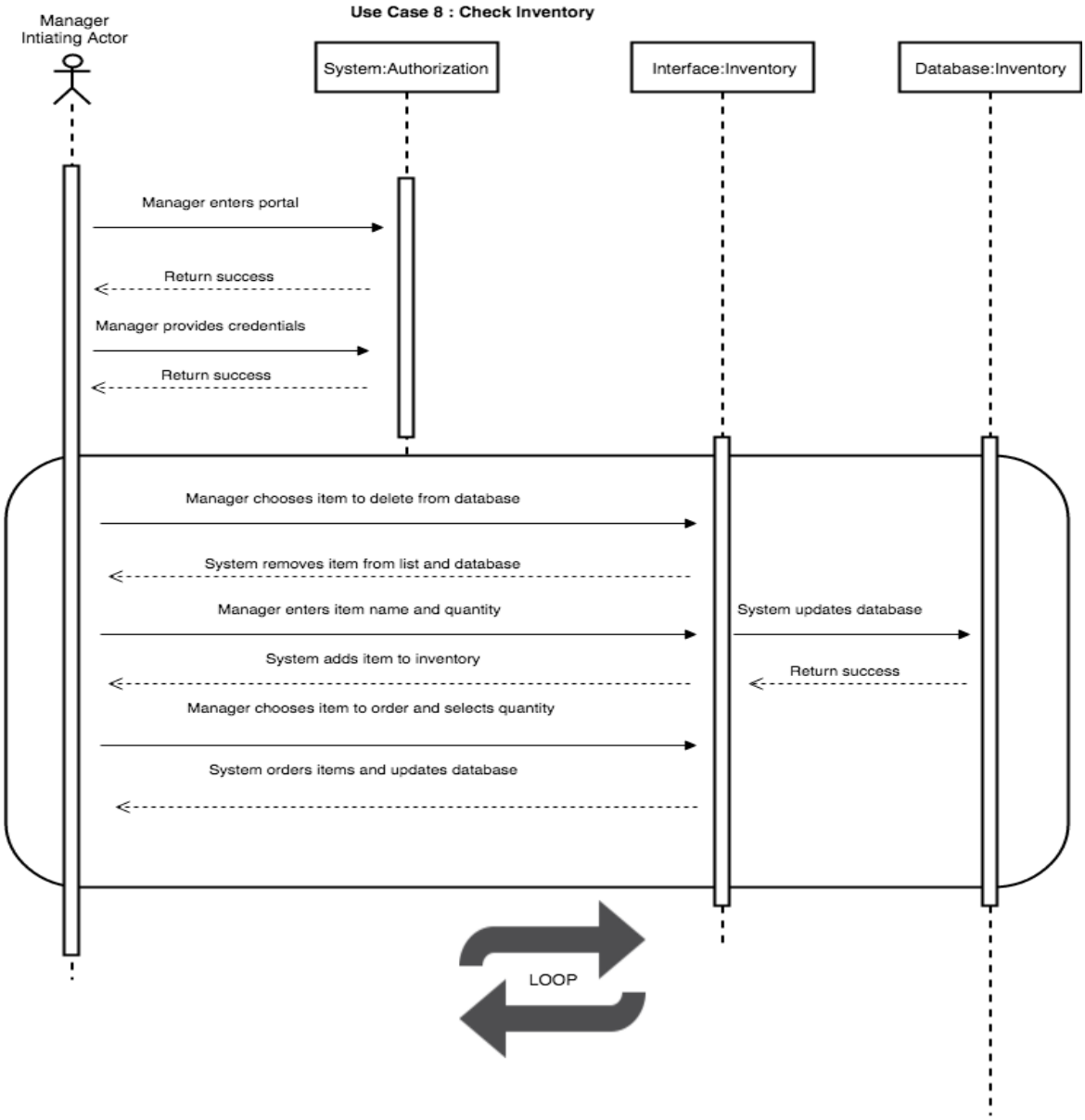
Manager

Use Case: EditShifts(UC-2)



One of the tasks of a manager is editing the shifts of employees if any changes are required. Deleting, adding, and changing the conditions of a shift requires, at most, three interactions per task. The transactions are quick; they are not difficult for the manager to execute and give visual results for the manager to see changes. Mistakes by the manager are also easily fixing in less than three interactions.

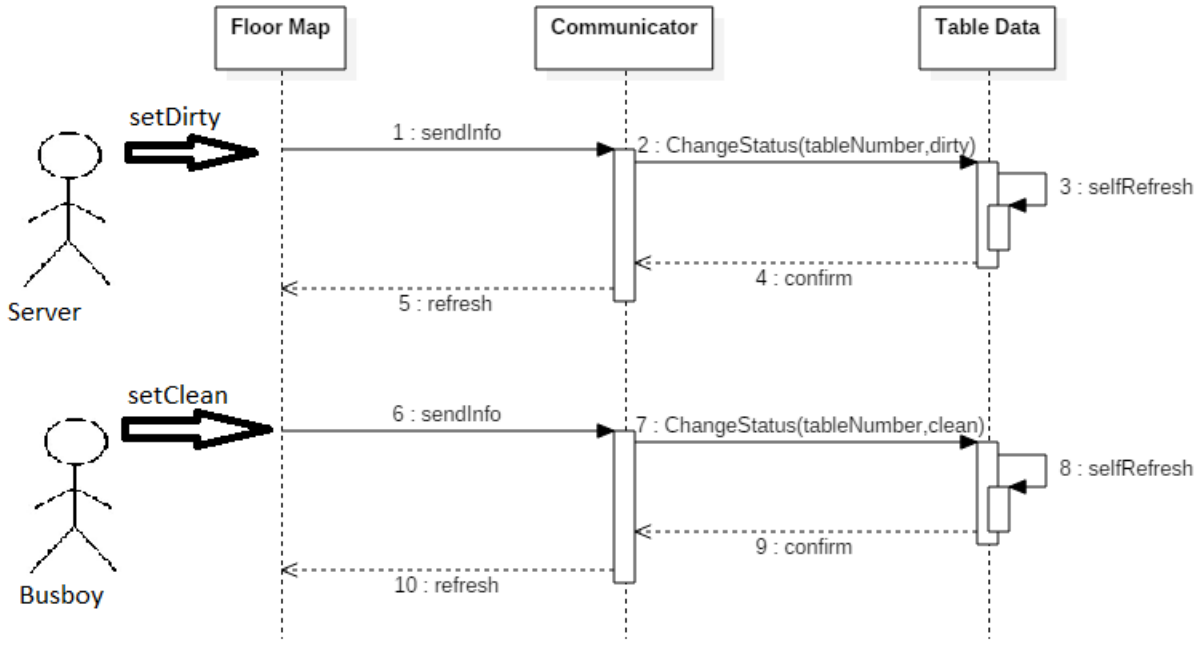
Use Case: CheckInventory(UC-8)



The manager enters their portal, and enters their credentials to access the restaurant's inventory. Here they can delete, add, and order items as they please. These changes will be reflected in the inventory database.

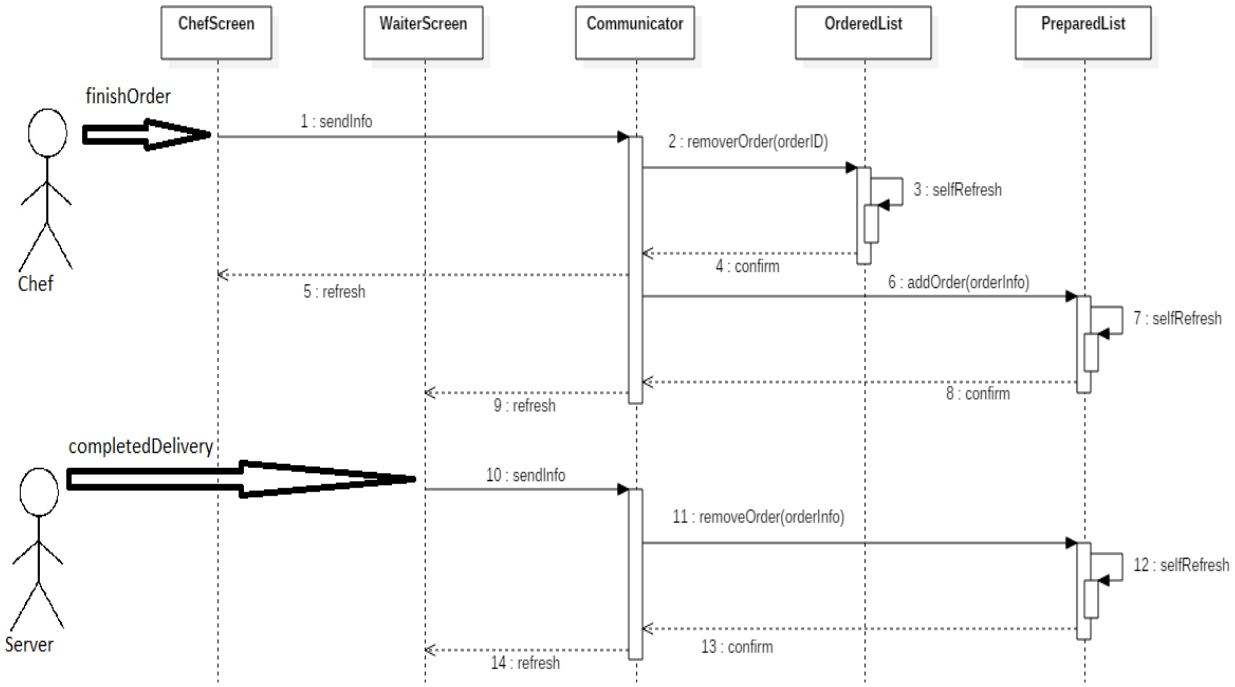
Kitchen

Use Case: CleanTable



An important task of the restaurant staff is to clean the table when the guests leave. To maintain an organized and up-to-date layout of the restaurant, the staff must input the current status of a table when they are finished doing certain things to maintain the table. In this situation, the server must input when the table is dirty and the busboy must input when he has cleaned the table. Any mistakes made when changing the status can be quickly set right at any time.

Use Case: DeliverFood

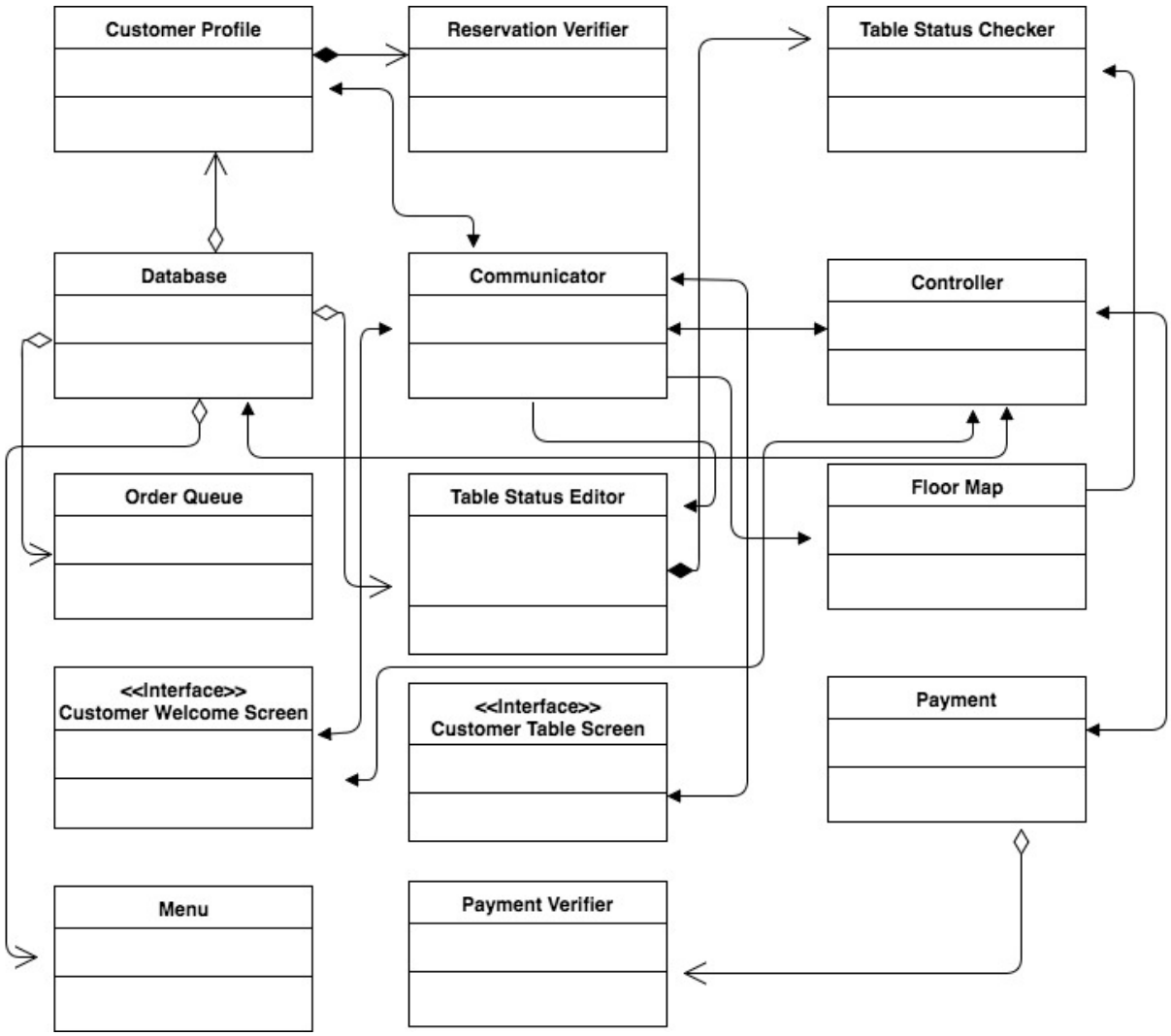


One of the most important tasks of the kitchen staff is to deliver a guest's food to them when its completed. When the chef has completed an order, he confirms it is done on his interface, and the order is then sent to the server interface. When the server has delivered the food, he confirms it and takes it off of the server interface's queue.

Part 2.2: Class Diagram and Interface Specification

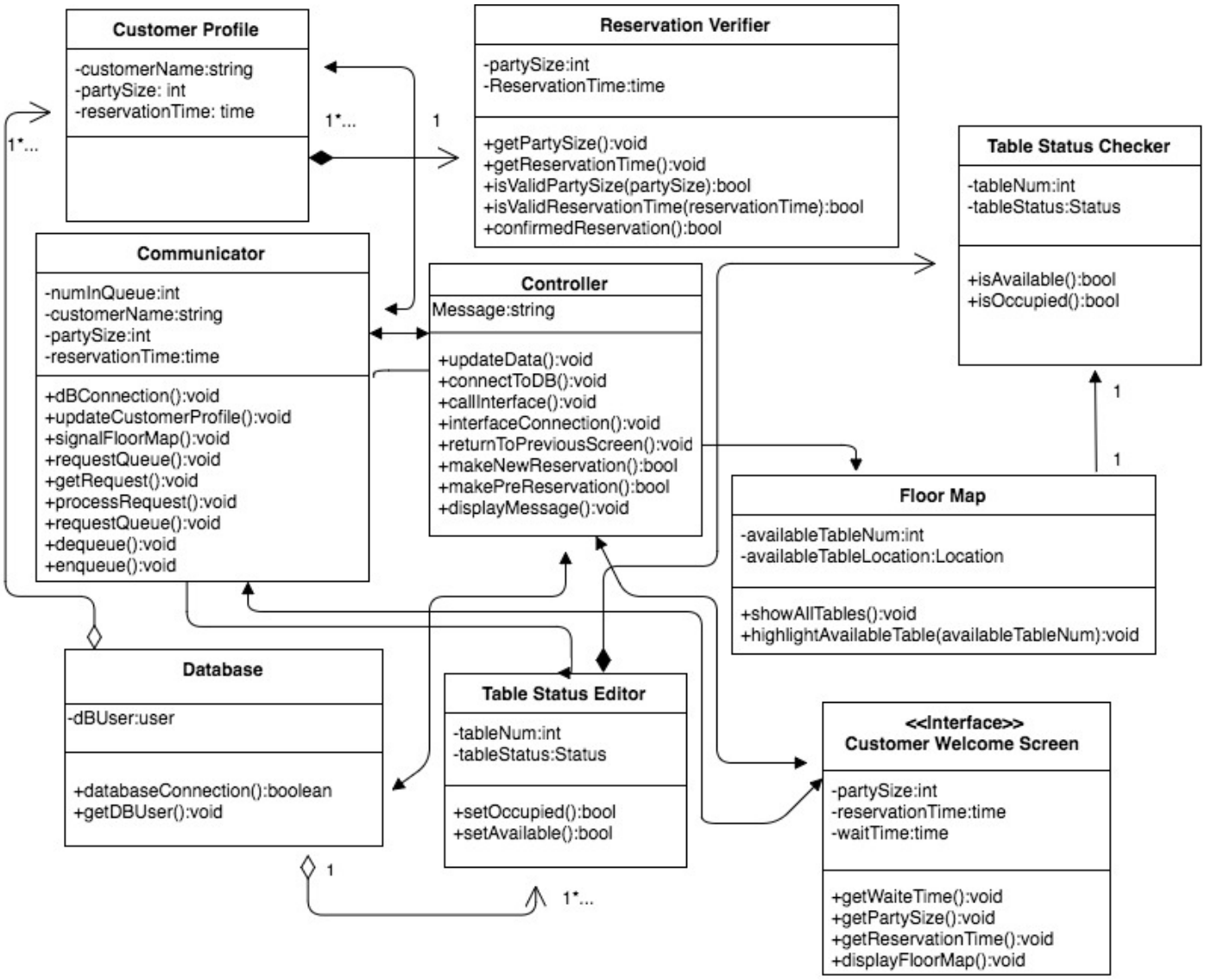
2.2.a. Class Diagrams

2.a.i. Customer



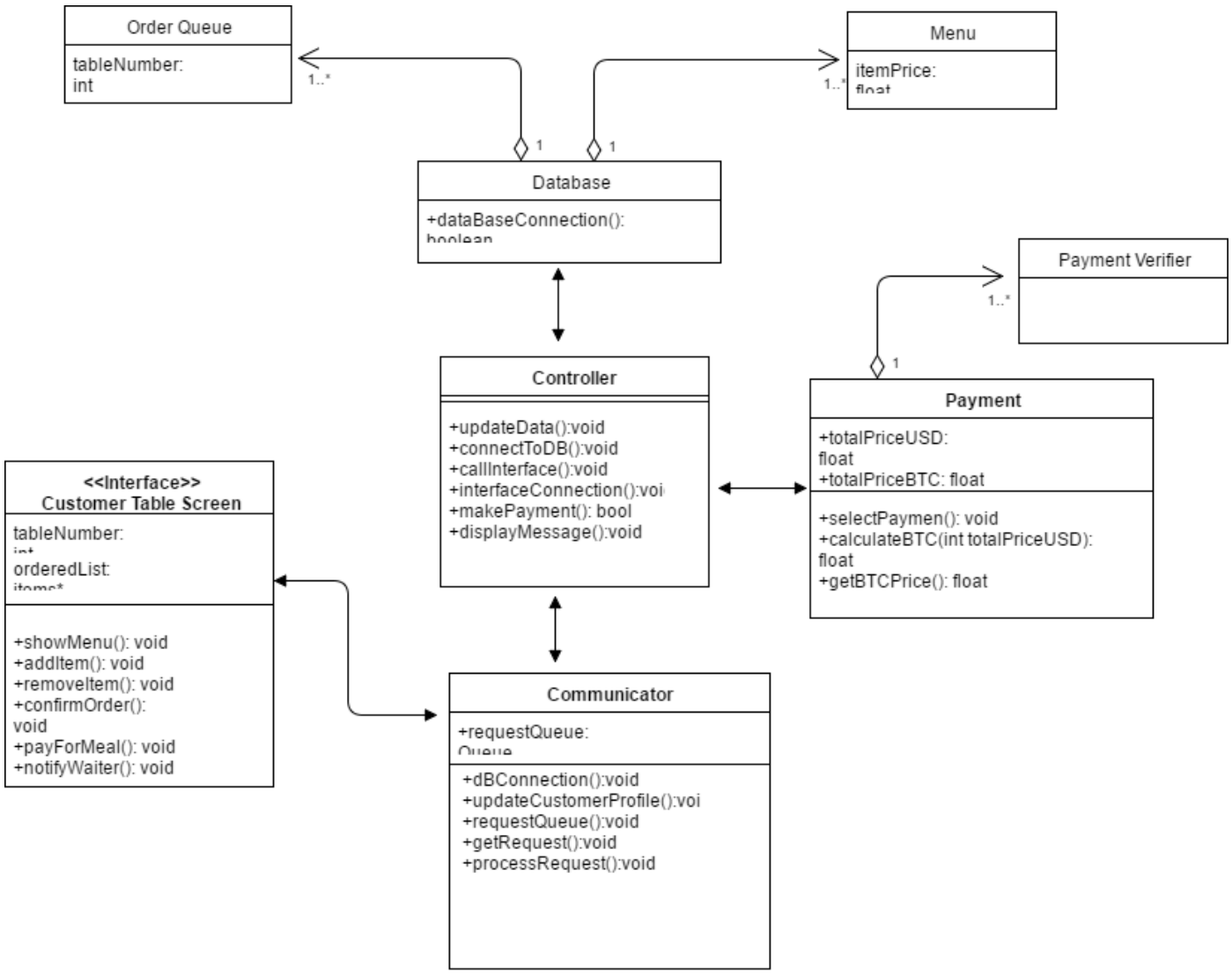
Pre-Reservation and New-Reservation (UC-16 & UC-17):

Below shows the class diagram for the fully-dressed use cases 16 and 17. This is based on the overview class diagram because the relationships between the classes are the same. However, the class diagram below is simply a partial diagram pertaining to the reservation process, for both pre-reservations and new-reservations.

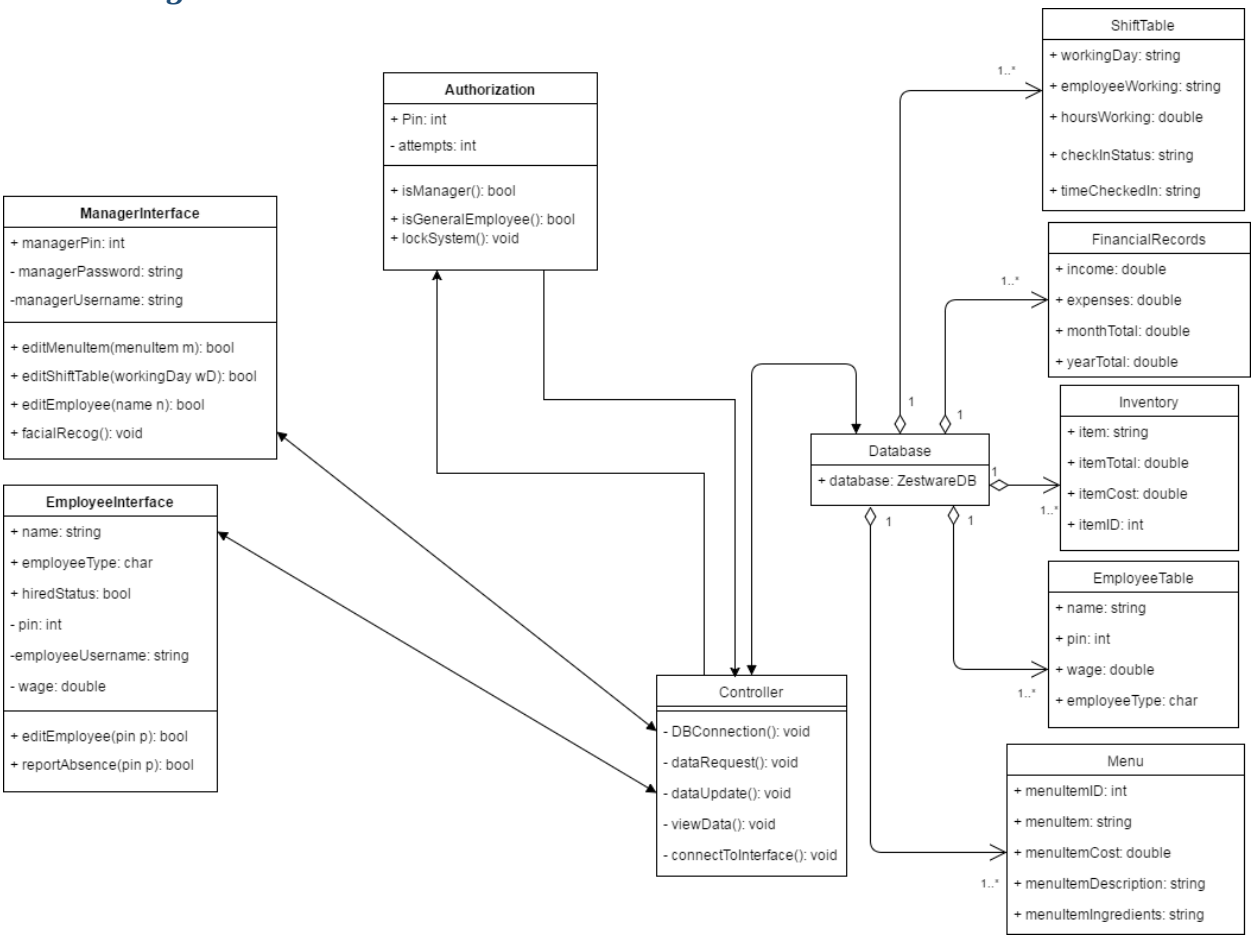


OrderItems and CardPayment (UC-23 &UC-30):

Below shows the class diagram for the fully-dressed use cases 23 and 30. This is based on the overview class diagram because the relationships between the classes are the same. However, the class diagram below is simply a partial diagram pertaining to the ordering of items and card payment processes.



2.a.ii. Manager



Class Descriptions

ManagerInterface **–**The Manager Interface is the manager's main form of completing their responsibilities. Here, they can change the menu, alter shifts, and edit employee information.

EmployeeInterface **–**The Employee Interface allows all employees to access information about themselves, and make any necessary changes. They may also use this tool to let the manager know that they will not be able to work a certain shift.

Controller **–**The controller is the layer that allows information to flow between the interfaces and the restaurant database.

Authorization-The authorization class determines the administrative powers of an employee. If the employee is a manager, they will have access to editing shift tables and other managerial actions. If the employee is not a manager, they will only be authorized to view the shift table but can request changes directly to the manager.

ShiftTable **–**The shift table is included in the database, and shows which employees are working in a day.

FinancialRecords **–**The financial records are included in the database which allows the manager to view expenses and income. They can also narrow this information down to a certain time period.

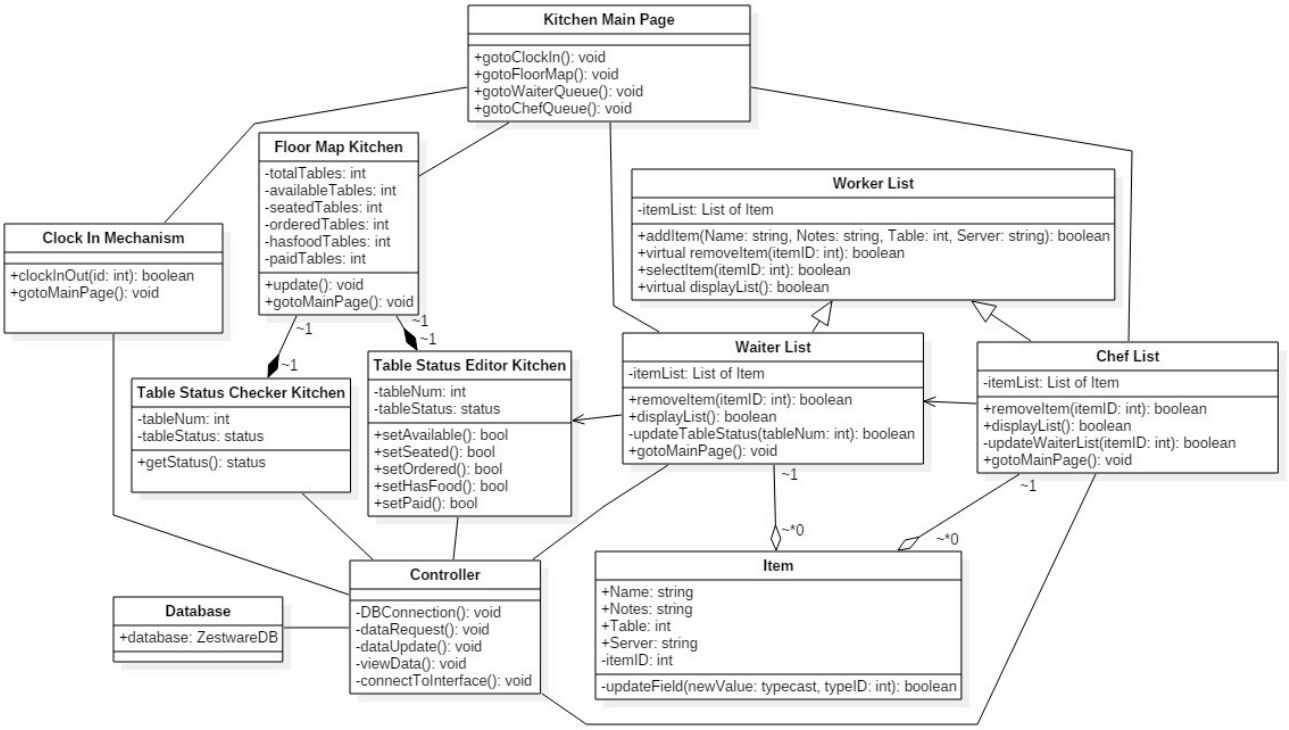
Inventory **–** The inventory is included in the database, and shows the amount of food items the restaurant currently holds. Furthermore, it has the price of an additional unit of that each item.

EmployeeTable **–** The employee table is included in the database, and contains information about each employee.

Menu **–** The menu is included in the database, and contains information about each dish currently served by the restaurant.

Database **–** The database is the central hub for accessing information about restaurant operations.

2.a.iii. Kitchen



Class Descriptions

Worker Main Page- The worker main page is the first thing users will interact with when they enter the kitchen part of the software. It allows the user to navigate to clock in, the floor map, the chef queue, or the waiter queue. Of course, the user can also go back to interface selection from here as well.

2.2.b. Data Types and Operation Signatures

2.b.i. Customer

Pre-Reservation and New-Reservation (UC-16 & UC-17):

Customer Profile:

The Customer Profile is in charge of holding information related to the customer in order to identify the customer and their reservation.

Attributes:	***Description:***
-customerName:string	*The customer's name that the reservation is under*
-partySize:int	*The party size of the customer*
-reservationTime:int	*The time the reservation is requested for*

Reservation Verifier:

The Reservation Verifier is in charge checking if the customer successfully made a new reservation. It depends on the Customer Profile because if there is no customer profile, then the reservation verifier cannot verify the reservation.

Attributes:	***Description:***
-partySize:int	*The party size of the customer*
-reservationTime:int	*The time the reservation is requested for*

Methods:	***Description:***
+getPartySize():void	*This method retrieves the party size the customer enters*
+getReservationTime():void	*This method retrieves the reservation time the customer made a reservation for*
+isValidPartySize(partySize):bool	*This method checks if the party size the customer entered is valid*
+isValidReservationTime(reservationTime):bool	*This method checks if the reservation time the customer entered is valid*
+confirmedReservation():bool	*This method returns true if the reservation made by customer is successful*

Floor Map:

The Floor Map is responsible for displaying the floor map of the restaurant (displaying all tables) and highlighting the available table that is assigned to the particular customer.

Attributes:	***Description:***
-availableTableNum:int	*The number of the available table*
-availableTableLocation:Location	*The location of the available table*

Methods:	***Description:***
+showAllTables():void	*This method displays all the tables of the restaurant on the floor map*
+highlightAvailableTable(availableTableNum):void	*This method highlights the table assigned to the specific customer*

Table Status Checker:

The Table Status Checker is responsible for checking the status of the tables.

Attributes:	***Description:***
-tableNum:int	*The number of the table*
-tableStatus:Status	*The status of the table specified by tableNum*

Methods:	***Description:***
isAvailable():bool	*This method returns true if table is available*
isOccupied():bool	*This method returns true if table is occupied*

Table Status Editor

The Table Status Editor is responsible for editing the status of the tables in the restaurant.

Attributes:	***Description:***
-tableNum:int	*The number of the table*
-tableStatus:Status	*The status of the table specified by tableNum*

Methods:	***Description:***
setAvailable():bool	*This method returns true if table is set to available*
setOccupied():bool	*This method returns true if table is set to occupied*

Communicator:

The communicator is responsible for communicating with the other parts of the system.

Attributes:	***Description:***
-numinQueue:int	*The number of people in queue*
-customerName:string	*The name of the person who the reservation is under*
-partySize:int	*The party size of the customer*
-reservationTime:time	*The time the reservation is for*

Methods:	***Description:***
+dBConnection():void	*This method determines the connection with the database*
+updateCustomerProfile():void	*This method updates the customer profile*
+signalFloorMap():void	*This method signals the floor map for the next customer in queue to be seated*
getRequest():void	*This method get requests from other parts of the system*
processRequest():void	*This method processes requests from other parts of the system*
dequeue():void	*This method removes the next customer to be seated from the queue*
queue():void	*This method adds a customer who is waiting to be seated to the queue*

Database:

The database is responsible for storing restaurant data on the server.

Attributes:	***Description:***
-dBUser:user	*The user utilizing the database*

Methods:	***Description:***
+dBConnection():void	*This method determines the connection with the database and returns true if successfully connected*
+getDBUser():void	*This method retrieves who is using the database*

Customer Welcome Screen (Interface):

The Customer Welcome Screen is responsible for displaying the floor map, the availability of the tables, and an estimated wait time for the customers.

Attributes:	***Description:***
-waitTime:time	*The time the customer has to wait for next available table that fits their party size*
-partySize:int	*The party size of the customer*
-reservationTime:int	*The time the reservation is requested for*

Methods:	***Description:***
+getPartySize():void	*This method retrieves the party size the customer enters*
+getReservationTime():void	*This method retrieves the reservation time the customer made a reservation for*
+getWaitTime():void	*This method retrieves estimated wait time for next available table*
+displayFloorMap():void	*This method displays the floor map on the screen*

Controller:

The controller is responsible for coordinating tasks between interface and database.

Attributes:	***Description:***
-Message:string	*The messages displayed to the customer*

Methods:	***Description:***
+updateData():void	*This method retrieves the party size the customer enters*
+connectToDB():void	*This method retrieves the reservation time the customer made a reservation for*
+callInterface():void	*This method retrieves estimated wait time for next available table*
+returnToPreviousScreen():void	*This method displays the floor map on the screen*
+makeNewReservation():void	*This method allows customer to make new-reservation*
+makePreReservation():void	*This method allows customer to make pre-reservation*
+displayMessage():void	*This method displays messages on the screen*

OrderItems and CardPayment (UC-23 &UC-30):
Customer Table Screen:

The CTS is the main way the customer will interact with the Restaurant personnel, from here they are given many options from ordering

Attributes:	***Description:***
tableNumber: int	*This is the location of the customer table*
orderedList: items	*An itemized list of the ordered items for the kitchen staff and also for calculating the total price later.*

Methods:	***Description:***
showMenu(): void	*This method shows the menu items including food, drinks, side orders, and desserts*
confirmOrder(): void	*Confirms the items on the order list, and adds it to the order queue for processing in the kitchen*
addItem(): void	*Adds a menu item on the current order list*
removeItem(): void	*Removes an item on the current order list*
PayForMeal(): void	*Allows the customer pay for the items they have ordered.*
NotifyWaiter(): void	*Allows the customer to contact the waiter for assistance.*

Payment:

This class processes payments for the customer, if the payment is in Cash, the waiter will be notified. Otherwise the customer will be able to pay directly at the CTS using this class.

Attributes:	***Description:***
totalPriceUSD: float	*The total price of the meal in USD*
totalPriceBTC: float	*The price of the meal in Bitcoin*

Methods:	***Description:***
+selectPayment(): void	*Allows the customer to select between Credit Card, Cash, or Bitcoin payment*
-calculateBTC(int totalPriceUSD): float	*Calculates the equivalent price in BTC*
+getBTCPrice(): float	*Finds the current price of Bitcoin for converting from USD*

Controller:

The controller is responsible for coordinating tasks between interface and database, the methods are the same as the use cases of pre and new reservation with the addition of makePayment() method..

Methods:	***Description:***
+makePayment(): void	*This method allows the customer to begin the payment process for their order*

2.b.ii. Manager

Manager Interface

The Manager Interface is the manager's way of handling their tasks, and is also known as the Manager Portal. After logging in, they will be able to access information that is only available to them and not other employees.

Attributes	***Description***
+managerPin: int	*Pin number consisting of 4 integers used when logging into the Manager Portal*
-managerPassword: string	*A password is necessary for accessing more secure data such as inventory*
-managerUsername: string	*Alias for the manager when logging into the Manager Portal*

Methods	***Description***
+editMenuItem(menuItem m): bool	*Used for editing a specific item on the menu*
+editShiftTable(workingDaywD): bool	*Allows for editing employee shifts on a given day*
+editEmployee(name n): bool	*Allows for the manager to editing an employee's information*
+facialRecog(): void	*Used as an alternate authentication to the Manager Portal and other data that requires manager authentication*

Employee Interface

The Employee Interface is used by all employees as a way of viewing their information and reporting their absences.

Attributes	***Description***
+name: string	*Name of the employee is used as an identifier*
+employeeType: char	*Categorizes the employee according to their role (waiter, chef, busboy, manager)*
+hiredStatus: bool	*Hiring status of the employee. This is only modified when the employee starts or ends their time at the restaurant*
-pin: int	*Pin number consisting of 4 integers used when logging into Zest-Ware as an employee*
-employeeUsername: string	*An alias for the employee when logging into Zest-Ware*
-wage: double	*Amount of money the employee is making on an hourly basis*

Methods	***Description***
+editEmployee(pin p): bool	*Used by an employee to edit their own information*
+reportAbsence(pin p): bool	*Allows for an employee to report their absence*

Controller
The controller establishes the connection between the interfaces and the database. Any modifications made using the interface is updated in the database, vice versa.

Methods	***Description***
-DBConnection(): void	*Attempts to establish a connection for the database*
-dataRequest(): void	*Retrieves data from the database*
-dataUpdate(): void	*Updates data in database and interfaces*
-viewData(): void	*Allows the employee to view data from the database through their interface*
-connectToInterface(): void	*Attempts to establish a connection for the interfaces*

Authorization
Determines the viewing and editing power of the individual logging into the system.

Attributes	*Description*
+Pin: int	*Pin associated with employee logging in.*
+attempts: int	*Number of attempts user has before system locks*

Methods	*Description*
-isManager(): bool	*Determines if the employee is a manager.*
-isGeneralEmployee(): bool	*Determines if the employee is a general manager.*
-lockSystem(): void	*Locks the system from user*

Shift Table
The shift table contains information pertaining to employee shifts.

Attributes	***Description***
+workingDay: string	*Name of a specific work day*
+employeeWorking: string	*Names of employees working on a given day*
+hoursWorking: double	*Number of hours an employee is working*
+checkInStatus: string	*Shows whether the employee is in, out, or at lunch.*
+timeCheckedIn: string	*Date and time that the employee checked in.*

Financial Records
This is used by the manager to view all finances for the restaurant.

Attributes	***Description***
+income: double	*Amount of money generated*
+expenses: double	*Amount of money spent*
+monthTotal: double	*Net amount of money made in a monthly period*
+yearTotal: double	*Net amount of money made in a yearly period*

Inventory
The inventory shows all relevant information to food items.

Attributes	***Description***
+item: string	*Name of the item*
+itemTotal: double	*Quantity of item given in pounds*
+itemCost: double	*Cost of buying an additional unit of a certain item*
+itemID: int	*A more systematic identifier given to each item in the form of 6 integers*

Employee Table
The employee table contains information to all employees

Attributes	***Description***
+name: string	*Name of the employee*
+pin: int	*The PIN of the employee used to log in*
+wage: double	*Amount of money the employee is making on an hourly basis*
+employeeType: char	*Categorizes employee based on their role*

Menu
The menu shows the current list of all dishes offered by the restaurant

Attributes	***Description***
+menuItemID: int	*An more systematic identifier given to each dish in the form of 6 integers*
+menuItem: string	*Name of each dish*
+menuItemCost: double	*Cost of each dish*
+menuItemDescription: string	*Description of each dish*

Database
The database contains all the restaurant's information.

Attributes	***Description***
+database: ZestwareDB	*The database utilized in Zest-Ware*

2.b.iii. Kitchen

Methods:	***Description***
+gotoClockIn(): void	*This is used to navigate to the clock in screen.*
+gotoFloorMap(): void	*This is used to navigate to the floor map screen.*
+gotoWaiterQueue(): void	*This is used to navigate to the waiter's queue screen.*
+gotoChefQueue(): void	*This is used to navigate to the chef's queue screen.*

Clock In Mechanism- Provides user with a simple numberpad interface in order to clock in or out using their employee ID.

Methods:	***Description***
+clockInOut(id: int): boolean	*This method communicates with the server to toggle the employees status between clocked in and clocked out if the employee enters a valid ID.*
+gotoMainPage(): void	*This method navigates the user back to the kitchen main page.*

Floor Map Kitchen- This is very similar to the customer floor map, however it incorporates many more statuses for a table to have in order to provide workers with more relevant information

Attributes:	***Description***
-totalTables: int	*An integer that holds the total number of tables in the restaurant.*
-avaiableTables: int	*An integer that keeps track of the number of available tables.*
-seatedTables: int	*An integer that keeps track of the number of tables that have customers that have not ordered anything yet.*
-orderedTables: int	*An integer that keeps track of the number of tables that have customers that are waiting on an order.*
-hasfoodTables: int	*An integer that keeps track of the number of tables that have customers that have received all of their orders.*
-paidTables: int	*An integer that keeps track of the number of tables that have customers that have paid and are leaving.*

Methods:	***Description***
+update(): void	*This visually updates the floor plan using the current table information.*
+gotoMainPage(): void	*This method navigates the user back to the kitchen main page.*

Table Status Checker Kitchen- This is very similar to the customer status checker and is also responsible for checking the status of the tables. The difference is that it is designed to support the multiple statuses a table can have in kitchen mode.

Attributes:	***Description***
-tableNum: int	*An integer that hold the designated table number of the table being checked.*
-tableStatus: status	*A data type that holds the status of the table being checked.*

Methods:	***Description***
+getStatus(): status	*A method that returns the status of the table being checked.*

Table Status Editor Kitchen- This is very similar to the customer status checker and is also responsible for editing the status of the tables. The difference is that it is designed to support the multiple statuses a table can have in kitchen mode.

Attributes:	***Description***
-tableNum: int	*An integer that hold the designated table number of the table being checked.*
-tableStatus: status	*A data type that holds the status of the table being checked.*

Methods:	***Description***
+setAvailable(): bool	*Sets the status of the table being edited to available.*
+setSeated(): bool	*Sets the status of the table being edited to seated.*
+setOrdered(): bool	*Sets the status of the table being edited to ordered.*
+setHasFood(): bool	*Sets the status of the table being edited to has food.*
+setPaid(): bool	*Sets the status of the table being edited to paid.*

Worker List- The parent class of waiter list and chef list. Provides the basic framework for these two subclasses.

Attributes:	***Description***
-itemList: List of Item	*This list holds all the items currently being processed.*

Methods:	***Description***
+addItem(Name: string, Notes: string, Table: int, Server: string): boolean	*A method that adds an item onto itemList.*
+virtual removeItem(itemID: int): boolean	*A virtual class used as a place holder for a function that will remove an item from the list*
+selectItem(itemID: int): boolean	*Used by the user to specify which item is to be added/removed.*
+virtual displayList(): boolean	*A virtual class used as a place holder for a function that will visually display and refresh the list.*

Chef List- Provides the interface and data required for the chef to see what needs to be prepared. When an item is complete, it can be removed from the chef's list which automatically adds it to the waiter list.

Attributes:	***Description***
-itemList: List of Item	*This list holds all the items currently being processed.*

Methods:	***Description***
+removeItem(itemID: int): boolean	*Removes an item from the list and refreshes the display.*
+displayList	*A function that will visually display and refresh the list.*
-updateWaiterList(itemID: int): boolean	*Called when an item is removed from the list. The function automatically send the item to the waiter list.*
+GotoMainPage(): void	*This method navigates the user back to the kitchen main page.*

Waiter List- Provides the interface and data required for the servers to see what food needs to be delivered.

Attributes:	***Description***
-itemList: List of Item	*This list holds all the items currently being processed.*

Methods:	***Description***
+removeItem(itemID: int): boolean	*Removes an item from the list and refreshes the display.*
+displayList	*A function that will visually display and refresh the list.*
-updateTableStatus(tableNum: int): boolean	*Called when an item is removed from the list/ delivered to the table in order to update the table status.*
+GotoMainPage(): void	*This method navigates the user back to the kitchen main page.*

Item- The Data structure used to keep all the relevent data of a food order.

Attributes:	***Description***
+name: string	*The menu name of the food item.*
+notes: string	*Any extra comments/directions the guests might provide for preparing their food.*
+table: int	*The number of the table that this item was ordered from*
+server: string	*The name of the server who is primarily responsible for the table that ordered this item*
-itemID: int	A number given to the order so the system can recognize it easily.

Methods:	***Description***
-updateField(newValue: typecast, typeID: int): boolean	*A method used to update any one of the fields of the item.*

2.2.c. Traceability Matrix

2.c.i. Customer

	Software Classes													
Domain Concepts	Customer Welcome Screen	Customer Table Screen	Customer Profile	Reservation Verifier	Table Status Checker	Table Status Editor	Floor Map	Menu	Payment	Payment Verifier	Order Queue	Controller	Communicator	Database
Customer	x	x					x	x						
Kitchen Staff								x			x			
Manager			x					x						x
Waiter	x	x					x	x						
New reservation	x		x	x	x	x						x	x	x
Ordering Food		x							x		x	x	x	x
Paying for order		x					x		x	x		x	x	x
Checking in			x	x		x	x					x	x	x

A lot of our classes were made directly from the domain concepts. A few classes needed to be added such as Order Queue and Payment. These classes allow for the customer's order to be sent to the kitchen, and for the customer to be able to pay for their meal via Cash, Card, or Bitcoin, respectively.

2.c.ii. Manager

		Software Classes									
Domain Concepts		*ManagerInterface*	*EmployeeInterface*	*Controller*	*Authorization*	*ShiftTable*	*FinancialRecords*	*Inventory*	*EmployeeTable*	*Menu*	*Database*	*Authorization*
EmployeeProfile			**X**	**X**	**X**	**X**			**X**		**X**	**X**
ManagerCredentials		**X**	**X**	**X**	**X**	**X**			**X**		**X**	**X**
Item		**X**		**X**				**X**			**X**	**X**
ShiftCalendar		**X**	**X**	**X**	**X**	**X**					**X**	**X**
FinanceTable		**X**		**X**	**X**		**X**				**X**	**X**
Menu		**X**		**X**						**X**	**X**	
Notifier		**X**		**X**				**X**			**X**	
Archiver		**X**	**X**	**X**							**X**	
PostProcessor		**X**		**X**							**X**	
SearchRequest		**X**	**X**	**X**							**X**	

The classes were evolved by the domain concepts to ensure the connection between all roles of the restaurant. Each class can be deduced based on the attribute description of each concept. An example of this is the FinanceTable domain concept, as the manager accesses all finances which is held in the restaurant database. This corresponds to a manger interface, financial records from a database, and a controller to access the information from the database and show it to the manager through the interface.

2.c.iii. Kitchen

Domain Concepts		*Worker Main Page*	*Clock In Mechanism*	*Floor Map Kitchen*	*Table Status Checker Kitchen*	*Table Status Editor Kitchen*	*Chef List*	*Waiter List*	*Item*	*Controller*	*Database*
Bus Boy		***X***	***X***	***X***	***X***	***X***		***X***		***X***	
Chef		***X***	***X***				***X***		***X***	***X***	
Hostess		***X***	***X***	***X***	***X***	***X***				***X***	
Waiter		***X***	***X***	***X***	***X***	***X***		***X***	***X***	***X***	
Customer							***X***	***X***	***X***		***X***
Manager			***X***	***X***					***X***	***X***	***X***

The classes came from the domain concepts by evaluating what the software should need to do in order to provide easy communication between everybody in the restaurant. Each class is designed to be an aid to the actions of the people in the restaurant. For example, the chef list class is intended to provide a line of communication from the customer to the chef that indicates when something has been ordered. It also serves as a line of communication from the chef to the waiter that indicates when food is ready to be delivered.

Part 2.3: System Architecture and System Design

2.3.a. Architectural Styles

The system is composed of multiple architectural styles. Each section will define the types of styles utilized.

Client and Server

The design is based on the client-server architecture, because, client-server is prevailing model for network programming. Zest-Ware has a lot of communication between client applications and server, for example, the interface communicating with communicator which in turn is linked to the controller (as can be seen in the class diagrams) and so on. So, almost every application requires some sort of communication between clients uses client-server architecture. Each entity on the network in Zest-Ware assumes the role of either a client or a server. A database server (Amazon Web Server, AWS) is the main server (web server) that is providing database services to other clients, and then that client can act as a server to the other client. To provide a set of resources or functionalities to the clients' requests the server is always active. For example, for making reservation the server (AWS) needs to be updated frequently to provide information to the client (CWS interface [this is an indirect link, which is linked via communicator, table status editor, floor map]) on its request. So, for new reservation the server would send data to the client by giving the information about the customers already in the reservation list/queue for specific table, data, and time. Using this information would validate the reservation of a new customer. And, for pre-reservation upon the client's request the server would provide the customer profiles. Similarly, the client (CTS) would be requesting the server for the menu items, availability of items, changes made to menu, etc. For the payment through bitcoin, the server in this case would be the internet (online network communication with online servers). The communication between client and server is never direct with each other, they are only capable of indirect communication through the server. Clients in Zest-Ware are located on CTS, CWS, KSS, and Manager's computer, while the servers are located on different computer systems like the database, online servers etc. They are loosely coupled systems communicating through a message-passing mechanism.

Zest-Ware is a 3-tier architecture: client layer, business layer, and data layer. The reason of business layer is for validation of data, calculations, data insertion etc. and acts as an interface between the client layer and data access layer. As mentioned earlier for reservation the validation of data is needed so a direct communication between client and server would create chaos at times that is why an interface in between them the business layer (application server) is needed. Similarly, for making changes in data the application server would handle the flow and updates of data between the data source (AWS) and the client applications.

For the program to be constantly updating and to keep track of restaurant data, database interactions are necessary. The manager needs to keep track of the restaurant's income, employees, and inventory. All this data will be kept in the database, which will act as the server in which the program will interact with. The authorization class also depends on the database to provide the viewing and editing accesses of the individual logging in. All data will be kept in the database for easy access and printing.

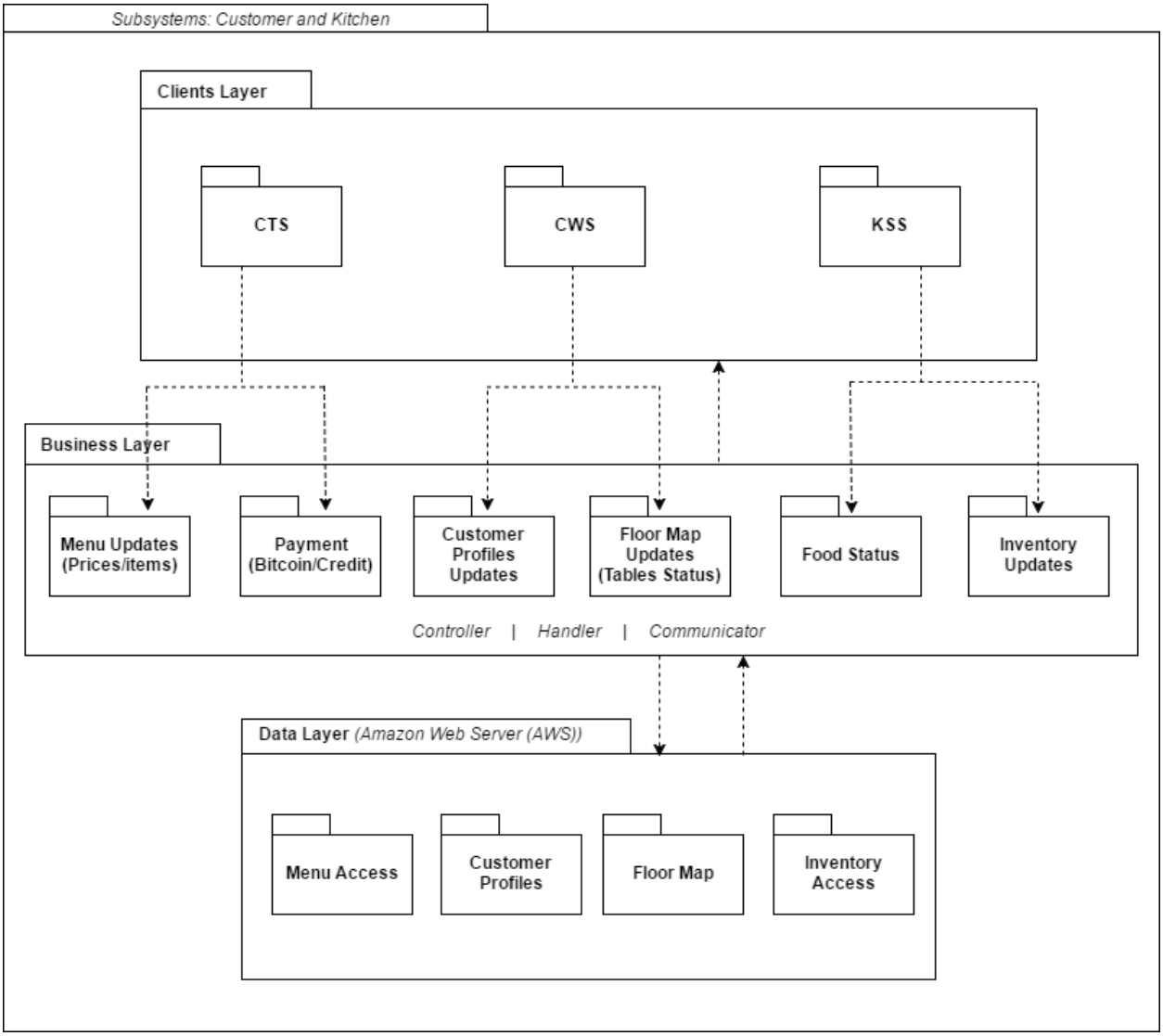
Domain Driven and Object Oriented Design

Employees, customers, and the manager are "business objects" (Microsoft, 2009). Each type of object has specific duties and attributes referring to their place as an object of the restaurant. All employees have check in times, a wage, check out times, and the ability to view the shift table. Customers provide the restaurant with income (or expenses) and feedback. The manager has higher administrative powers through methods that allow the manager object to edit shift tables, employee data, and to order items through the inventory interface.

Message Bus (Microsoft, 2009).

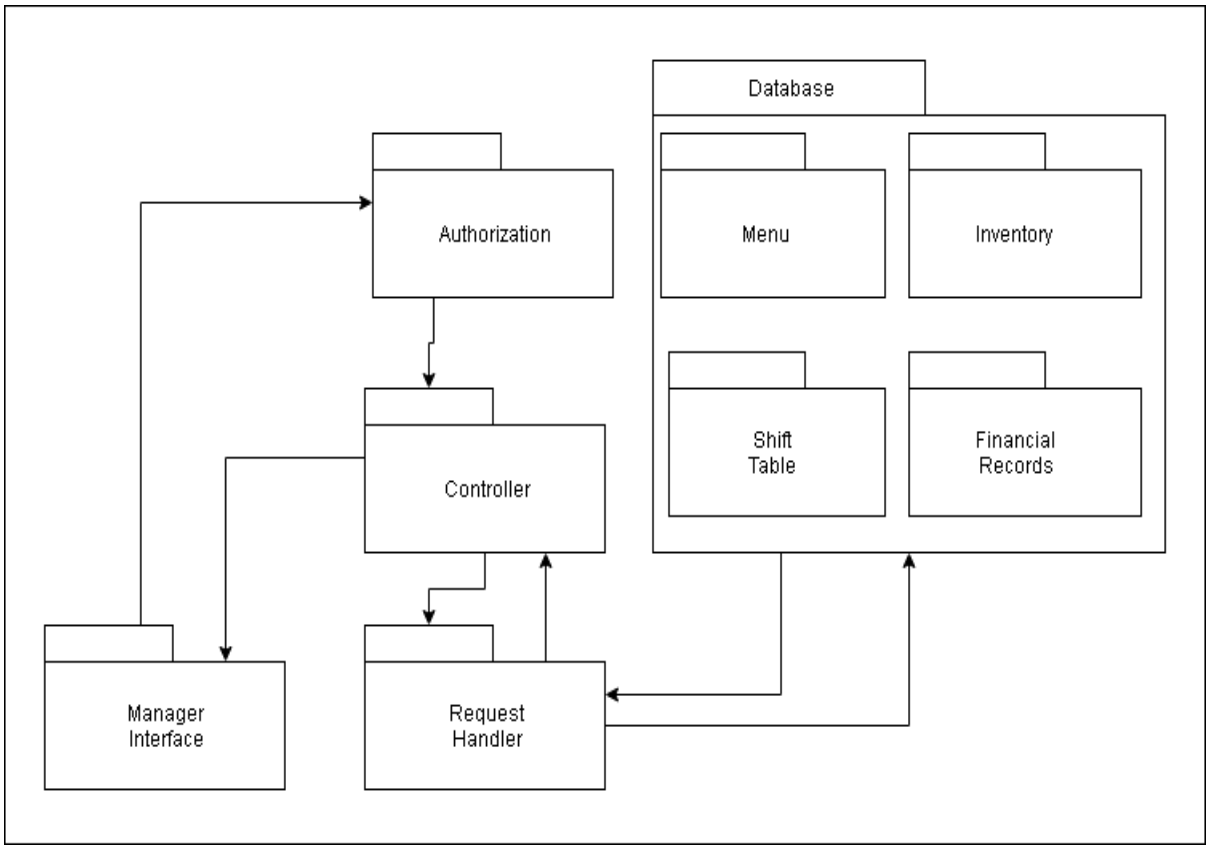
The system responds to humans using update messages for certain events. Event messages from the system include, but are not limited to, incoming food messages indicating that the food is ready for the table. When the chef has finished preparing the food, another employee can change the food status to ready and the customer may pick up their order. Another case occurs when a customer has finished using their table. After payment, the busboy is alerted that the table is vacant for cleaning, and after cleaning the busboy can alert that the table is cleaned and ready for use.

2.3.b. Identifying Subsystems



Package Diagram: Subsystems in the Customer and Kitchen part

The interfaces are CTS, CWS, and KSS mentioned in the Clients Layer. As mentioned in the architecture design they don't communicate directly with the server (AWS), so, there is a Business layer in between them i.e. the controller, handler, communicator etc. that would act as the link between them. The entities in business layers repeatedly get updated information from the data layer, and whenever the entity in client layer request's the business layer provides or takes care of it. The data storage, data access etc. are developed and maintained as independent modules on separate platform i.e. the AWS.



Package Diagram: Manager Subsystem Diagram

Specifically, for the manager, all manager tasks are collected from the manager's interface. Once the individual has been authorized to access various data, the controller sends requests to the request handler. When the manager wants to edit any of the data in the database, the request handler makes the changes via methods and confirms the changes with the manager.

2.3.c. Mapping Subsystems to Hardware

The mapping of subsystems to hardware is simple, and will be deployed on multiple computers and tablets. Each back-end/front-end part of the software is somewhat analogous to the placement of the subsystems in the restaurant. The central computer, which will be exclusively accessed by the manager, will hold the interface along with the handler, authorization, database, and controller while the stationary tablets placed on each table will hold the manager interface, authorization, handler, and controller. This holds for the tablets used by the busboy and chef.

The interface CWS runs on a big touch screen computer installed at the entrance of the restaurant. The interface CTS runs on small touch screen tablets installed on each table. The interface KSS runs on a big touch screen computer installed in the kitchen. These act as clients. While, the server is web server i.e. Amazon Web Server. There are some individual clients inside the CTS, that use client as web browser, e.g. entertainment section in the CTS. The other form of client is for the bitcoin payment where a QR code is displayed on the CTS, and the bitcoin application installed on the customer's phone acts as a client to send the bitcoin to the displayed public address, which in return the CTS acts as a client to Zest-Ware's server (AWS) where the transaction can be verified.

2.3.d. Persistent Data Storage

To keep data on the restaurant's various attributes, a relational database will be implemented. Many of the operations of a restaurant can be well defined but are also dependent on attributes of each other. For example, when a customer orders food, the request is sent to the chef to prepare the food. For the customer interface to be updated to indicate that the food is ready, an employee must respond through the system that the food is ready for pickup. The order was initiated by the employee and updated by the actions of the employee overlooking the process. The order is also archived for viewing by the manager. All the past orders made are stored in the database and can be easily totaled to compare to the daily expenses of the restaurant. The program will use simple SQL statements to provide the manager with information regarding the financial status of the restaurant. Below are various persistent objects related to the restaurant which must be archived for future analysis.

Persistent Objects

Employees– Employee objects which include waiters, busboys, managers, and chefs.

Customer Surveys – Feedback surveys collected from customers after their visit and meal.

Financial Records of the Restaurant – All income and expenses of the restaurant.

The Menu – Menu of the restaurant and the major ingredient associated with each item.

Order Status – Status of a current order.

The Inventory – Stock of items used by the restaurant such as food stuffs, furniture, toiletries, etc.

Shift Tables – Time tables of employees of the restaurant.

Restaurant Floor Map – Unique floor map of the restaurant using the software.

Table, Food, and Employee Working Status – Statuses of various objects. Employees checking in, checking out, or going for lunch. Tables either vacant of occupied. Food either ready or in the process of cooking.

2.3.e. Network Protocol

Client Server Pair

The program uses the concept of a client-server pair. Employees or entities of the restaurant may need to access data from tables within a database. The client server pair ensures that communication is quick and reliable between restaurant entities and the database.

Java JDBC

This is a Java API that can access data within the database for viewing or manipulation. The program will be written using Java, therefore using this API will be beneficial and convenient. The steps of the interaction will go as follows:

1. When a specific method is called requiring access to the database, a connection to the database will be established.
2. Within the method, SQL queries will be sent to request or change information.
3. The queries will be processed and changes will be made or data pulled to be displayed to the client.

2.3.f. Global Control Flow

Execution Orderness:

The system has both procedure-driven and event-driven models. The procedure-driven is for running the loops and updating status. Where, the program must repeatedly check for the update in status of tables, food, etc. so, it is procedure-driven because the program doesn't have to wait for any user input. While, some parts of the program are modeled where it must wait for an event to occur or for user input, e.g. when making a reservation, checking for reservations, placing an order, etc. they are all event-driven, because the user is interacting with the system, and the system must wait for the next input by the user in order to execute the next instruction. So, the flow of execution in Zest-Ware is both based on the program-driven and event-driven models.

Time Dependency:

The system is real-time type and event-response type. Zest-Ware is real-time type because when a specific operation is done e.g. order being place, order being prepared, call for assistance, payment, make reservation etc. then for these Zest-Ware needs to respond within milliseconds or microseconds. If compared to non-real-time system where a system response is not guaranteed within any timeframe then this would cause a problem, e.g. if the order is placed at a table then the system needs to receive data, process it, and return the results as soon as possible (like send the information to the KSS). Zest-Ware is also periodic real-time for checking the status of food prepared, the status of tables, etc. Now, it could be a loop running that keeps waiting for a change in status of table, or payments made etc. So, Zest-Ware is an event-response type because for certain operation it must react according to them e.g. there are different interfaces for different screens, so, for each interface selected the system should respond and display the specific interface, e.g. for the screen on entrance when the CWS is selected then the system would display the interface of CWS, and for the tables when the CTS is selected then the system would display the interface of CTS. Another example of event-driven in Zest-Ware is when customer is entering the name, party size, etc. for making a reservation, so, the system runs an event loop that keeps waiting for such activities.

Concurrency:

As mentioned earlier, Zest-Ware will be working on multiple systems, and the system of Zest-Ware has multiple threads running on each system. Now, each system can be running and executing a thread at the same time, which is why it is concurrent. Concurrency in the system can occur when multiple orders are in the process of being placed at the same time, so, threads of the system are occurred at the same time on different CTS. Synchronization is not used when the orders are being placed, because each process needs to access the menu at the same time, so that other customers can place order and does not have to wait for one customer to finish processing and then the thread is available for another customer to process it. Though, Synchronization is used when the thread which is repeatedly updating is being accessed e.g. when a user is trying to access the inventory database, or table status (floor mapping system) etc. then the system needs to make the user wait for a while if the inventory database is currently being accessed by another process, or if the status of table is being changed (which means it is accessed by another process/thread at the same time as the user tries to access).

2.3.g. Hardware Requirements

Tablet Requirements		
	Minimum	***Recommended***
Screen Size	*13”*	*15”*
Display Type	*Touch Screen (Colored)*	*Touch Screen (Colored)*
Display Format	*4:3 Standard LCD*	*16:9 TFT LCD*
Resolution	*800 x 600*	*1024 x 768*
Processor	*1 GHz*	*1.5 to 2 GHz*
RAM	*512 MB*	*1 GB*
Storage	*16 GB*	*32 GB*
Network Connectivity	*IEEE 802.11a Wi-fi*	*IEEE 802.11n Wi-fi*

Touch Screen Computer Minimum Requirements		
	Minimum	***Recommended***
Screen Size	*17”*	*20” to 22”*
Display Type	*Touch Screen (Colored)*	*Touch Screen (Colored)*
Display Format	*4:3 Standard LCD*	*16:9 TFT LCD*
Resolution	*1024 x 768*	*1280 x 1024*
Processor	*1.8 GHz*	*2 to 2.4 GHz*
RAM	*1 GB*	*2 GB*
Storage	*20 GB*	*40 GB*
Network Connectivity	*IEEE 802.11a Wi-fi*	*IEEE 802.11n Wi-fi*

Part 3.4: Algorithms and Data Structures

4.a. Algorithms

4.a.i. Customer

Assignment of Tables Algorithm:

When a customer makes a new-reservation or pre-reservation, they will be requested to enter their name and party size by the Customer Welcome Screen (CWS). The customer's party size will be stored in a queue, considering first in first out order. The algorithm will search for an available table that suits the party size and the floor map will highlight the table assigned to the customer. The list of available tables will be known with the help of the Customer Table Screen (CTS) and the busboy's screen. As a customer successfully makes a payment and the busboy has marked the table ready for the next customer through his screen, the table status will change to available for the specific table (using the table's identification number). The algorithm will utilize two arrays that will help it determine which tables are available. The first array will store the identification numbers of the available tables and the second array will store the identification numbers of the tables that are occupied. The algorithm will check the first array with the next customer's party size. If there is an available table that fits the party size, then the algorithm will assign that table to the customer. It will send the table identification number assigned to the customer to the floor map so it can highlight the table on its screen. However, if there is no available table that fits the party size, then the customer will be told to wait in the waiting area and another algorithm would be used, which will calculate the estimated wait time that fits the customer party size accordingly.

Estimate Wait Time Algorithm:

As tables at the restaurant become occupied and the customers are requested to wait in the waiting area, they are shown an estimated wait time on the Customer Waiting Area Screen (CWAS) for the next free table. An array will be needed to store the reservation/check-in time of the customer in order for an algorithm to calculate the estimated wait times. The algorithm will calculate an average dine time for customers and a buffer will be added for larger party sizes. In our system, small party sizes (1-2 people) will be shown the average dine time as the estimated wait time with no buffer added. For medium party sizes (3-4 people), there will be a small buffer added to the average dine time, and larger party sizes (5 and above) will have more buffer added to the average dine time to calculate the estimated wait time. For example, if a customer of party size two had checked in fifty minutes ago and the average dine time our system has stored for small party size is one hour, the customer will be shown the estimated wait time as ten minutes on the CWS.

Total Meal Cost Algorithm:

After a customer has selected the meals they want to purchase and confirmed their order on the Customer Table Screen (CTS), an algorithm must be utilized that calculates the total cost. This algorithm will include a linked list with the items that the customer has ordered and add up the cost of each element, including the tax. This algorithm will help calculate the total cost of the food that each customer has purchased.

Payment through Bitcoin Algorithm:

If the customer decides to pay for their meal via Bitcoin, an algorithm is needed to calculate the price of the meal in BTC. The bitpay API automatically calculates the total price in BTC that the customer should pay using an updated exchange rate of the passed currency parameter. On the ZestWare end of the software, it is our responsibility to provide a total cost that is accurate and includes the tax to be collected. The functioning of the total meal cost algorithm was discussed above.

4.a.ii. Manager

The algorithms used by the management implementation of Zest-Ware are not complex, but are essential in implementing use case functions. For our manager and employee interfaces, there are not any algorithms implemented since these tools hold mostly static information. As stated in our controller class, there will be functions to establish a connection to the database, and functions to transfer data between the interface and database. In order to implement these functions, there will be algorithms used to sort and modify the information. For our inventory, we will use sorting algorithms to identify items which are low in stock. The FinancialRecords class will calculate averages in a certain period to categorize expenses and income. Furthermore, the food cost formula from the mathematical model section will be utilized in this class.

Clock In Algorithm:

The program will keep track of employees clocking in and out of the restaurant. Using a basic clock in algorithm and PHP and Java to access the database, all employee in and out statuses will be tracked easily. The process starts when the employee is prompted for their ID number when they encounter the desktop or tablet interface. Once they log in, the PHP file will access the database and search for the employee. If the employee does not exist, access is denied and the system will alert the user. If they do exist, the date and time of clocking in will be recorded and displayed to the user. They are also entered into an array of employees currently in the active status in the restaurant. In other words, they are currently clocked in and ready to work. The logged in interface will be displayed to the user. On the interface, the employee will have the options to punch out or go to lunch. As of now, the employee status is punched in, and this is also recorded in the database. Selecting punch out will remove the employee name from the array of active employees and change their status to out. The date and time of punching out will also be recorded. When “go to lunch” is selected, the employee name will be moved to another array for employees on lunch break. When the employee returns, they log in with their ID, select “back from lunch,” and return to their duties. The system will record the date and time that the employee has returned and will remove the employee name from the lunch array and place the employee name back into the active employee array. From observation, the arrays of active and at lunch employees will be very small. This is desired. Rather than continuously going back and forth to the database for the current employees, the small arrays will be easier to sort through and display. In effect, the arrays will be sorted using insertion sort. Insertion sort, is an in place algorithm that will not require any extra memory; the algorithm is also stable and can sort a list as employees punch in. This makes the algorithm desirable over others in terms of giving an immediate response to a manager looking to see who is in for work.

Financial Record Calculator Algorithm:

There will be times where the manager wishes to view the financial status of the restaurant. The financial tracker will be very simple. For the current scope of the project,

the restaurant will have a balance of money owned. When customers purchase food, the amount owed will be directly added to the restaurant balance. When the manager wishes to make a purchase, the amount will be deducted from the restaurant balance. When employees need to be paid on the pay date, the amount will also be deducted from the restaurant balance. Rather than using complicated financial algorithms first hand, it is imperative that the money transaction system works on a smaller scale first.

4.a.iii. Kitchen

addItem(itemID: int, Notes: string, Table: int, Server: string) : boolean

Description:

This function is to be used when an order comes from the customer into the kitchen, or when the kitchen staff completes an item and it is added to the waiter list. This algorithm assures that an item is added to the list safely and completely. When it is called, it adds another item entry onto the end of the linked list that has the attributes that are entered in the function handle. The function returns a "true" if it succeeds and "false" otherwise.

Pseudocode:

```
//itemList is a linked list of item classes
addItem(itemID: int, Notes: string, Table: int, Server: string) : boolean{

        if(argument in handle invalid){
                return false
        }
        else{
                new Item tempItem

                tempItem.updateField(itemID, itemID)
                tempItem.updateField(Name, Name)
                tempItem.updateField(Table, Table)
                tempItem.updateField(Server, Server)
                tempItem.updateField(Notes, Notes)

                itemList.add(tempItem)
                if(itemList denies the addition){
                        return false
                }
                else{
                        return true
                }
        }
}
```

removeItem(itemID: int) : boolean

Description:

This function is used when an order is completed by the kitchen staff or a waiter delivers an order to a guest. The algorithms main function is to remove any item selected and have the linked list mend itself back together. When it is called, it removes an entry and has the pointer that was pointing to it point to the item that the removed item was pointing to. The function returns a "true" if it succeeds and "false" otherwise.

Pseudocode:

```
//itemList is a linked list of item classes
removeItem(itemID: int) : boolean{

        if(itemID not found in linked list){
                return false
        }
        else{
                find item with the particular ID
                itemList.removeItem(index)

                if(itemList denies the removal){
                        return false
                }
                else{
                        return true
                }
        }
}
```

updateField(newValue: typecast, typeID: int): boolean

Description:

This function is used to rewrite the attributes of an item. The algorithm detects which field should be changed via the second argument in the handle, and then changes it to the first argument. It returns “true” if the update is successful and “false” if the update fails.

Pseudocode:

```
updateField(newValue: typecast, typeID: int): boolean {

        switch (typeID){

        case Name:
                self.Name = newValue
                return true

        case Notes:
                self.Notes = newValue
                return true

        case Table:
                self.Table = newValue
                return true

        case Server:
                self.Server = newValue
                return true

        case itemID:
                self.itemID = newValue
                return true

        default:
                return false
        }
}
```

suggestCookingOrder(newItem : item) : void

Description:

After basic the basic workings of the software is successfully implemented, we would like to work on a mechanic that lets the chef ask the software what the most efficient order to prepare the items might be. This would add a new field in the item list that contains the menu item's approximate cook time. Basically, the function would use the approximate cook time, which table order it, and the order in which the items are ordered to determine the resulting sequence. It also will acknowledge the fact that multiple things can be cooked at once to generate the sequence. The overall method needs to be perfected with some simulation of orders coming in and practice, but we have a general idea of how it would work. Ideally, this will update for every new item added to the chef list.

Pseudocode:

```
suggestCookingOrder(newItem : item) : void{

        if(newItem and the last 2 items are from the same table){
                do NOT allow these items to be seperated in the list, overriding the other
rules
        }

        if(newItem ordered by same table as item in the last 5 spots on the list){
                put newItem after the other item
        }

        else if(newItem has longer cook time than one of the last 5 items){
                put newItem before this other item
        }

        else if(newItem has cook time less than 1 minute){
                put newItem at top of the list
        }

        else{
                put newItem to the end of the list
        }
}
```

4.b. Data Structures

4.b.i. Customer

For the system to implement the algorithms and other features, certain complex data structures must be used. Data structures are important in our software system because they make it easier to store and sort data. As the system operates, the information within these complex data structures will need to be retrieved and utilized in performing a specific task. The software system mainly uses sorted arrays because they have better performance since they can access information quickly. We also used priority queues because elements can be removed and added from the ends and linked lists because they do not have a fixed size.

Data structure needed for assignment of tables:

In order to designate tables to the customers at the restaurant, two arrays and one queue must be used. One of the arrays will hold the identification numbers of tables that are free to be used by customers. This array will be sorted in ascending order. Similarly, another array will be used that will hold information about tables that are unavailable to the incoming customers. Each element in this array will represent the identification number of a table within the restaurant. This array will also be in ascending order. Through the use of these two arrays, algorithms can easily search which tables are available and compare the customers' party sizes with the total number of people that can be seated at each corresponding table. As the table statuses change from available to occupied and vice versa, the identification numbers of the corresponding tables will be removed from one of the arrays and added to the other array, indicating the status change. In addition, a queue will be used for storing the customer's party sizes. It will be a way to determine which customers arrived first to the restaurant because the queue will be first in first out order (FIFO). As more customers visit the restaurant, their party sizes will be added to the end of the queue and will not be matched with an available table until the party sizes in front have been handled accordingly.

Data structure needed for estimating wait time:

As customers enter the restaurant and are added to the end of the queue, another sorted array will be needed in order to calculate the estimated wait time. This array will have elements in ascending order that represent the time a customer has made a reservation for or has checked in at. The elements of this array can be compared to the average dine in times corresponding to the party size to determine the estimated wait time.

Data structure needed for ordering and total meal cost:

When a customer selects food from the menu on the Customer Table Screen (CTS), these items and their costs will be stored in a linked list. A linked list is suitable to use in this case because a customer can delete a dish anywhere in the list, unlike an array. In an array, there is less flexibility and more difficult to add or remove items in any position. Arrays have fixed sizes and to increase their sizes utilizes a lot of storage and time. Thus, a linked list works well for the software system in terms of ordering because a customer has the option to make changes to the order and the item can be located anywhere in the list. In addition, this linked list will help determine the total meal cost. When customers have confirmed their orders, all the elements in the linked list will be summed together to determine the total cost of the customer's meal.

4.b.ii. Manager

For the manager portion of the software, all information is stored in the database, so there is no need for other forms of data structures.

Employee Arrays:

As mentioned above, there will be two arrays for general employees of the restaurant. The arrays will include all employees and sort based on ID number and then alphabetical order by last name. The first array contains all active employees currently working. Whether they are a chef in the back cooking, a busboy cleaning, or a waitress tending to a table, the manager will easily be able to view who is in and is doing what. The second array will display employees who have gone to lunch. Once again this array will be sorted by ID number, then alphabetical order by last name. The manager will be able to see who is currently out easily and quickly thanks to the real time capability of insertion sort. Lastly, if desired the manager may sort by employee position (such as C for chef or B for busboy) or they may sort by first name. There will be multiple options for the convenience of the manager.

Inventory Shopping Cart:

When the manager must purchase items for the benefit of the restaurant, they access the manager inventory check portal. In the case that a manager wishes to purchase multiple items, or even just one item, a general shopping cart is needed in order for the manager to review the purchases. For this case, a Linked List of inventory items will be implemented. Each item will have its own identifier as a part of the linked list. When an item is added to the cart, the system will prompt the manager for a quantity. The manager provides the quantity and this is changed in the attributes of the item. In the case that the manager wishes to change the amount later, they can choose the item from the cart interface, update the amount, and the system will change it. When removing an item, the linked list will simply point the tail of the item before the deleted item to the head of the next item. The implementation will be simple and not difficult for the program to run.

4.b.iii. Kitchen

The item lists for the worker list classes are going to be implemented as linked lists. In the case of addItem, it does not really matter how the list is implemented. However, for removeItem, linked lists would probably be the best to use since its a little more difficult to remove items from the middle in other listing data structures.

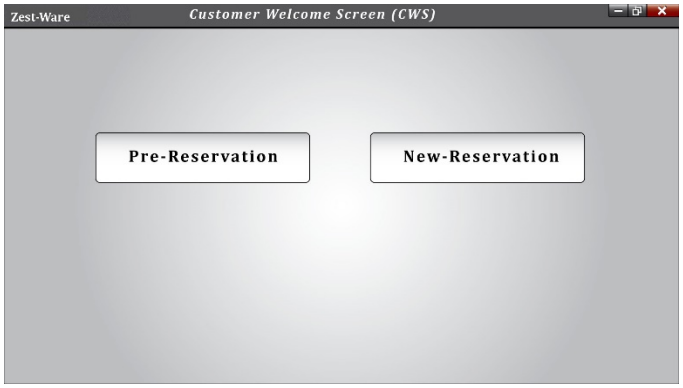
For the data structure holding the data of each table, a simple array will work the best. The tables in the array just need to be accessed in order to change their attributes and there rarely needs to be an addition or removal of a table. So, even the simplest data structures will suffice in this situation.

Part 3.5: User Interface Design and Implementation

5.i. Customer

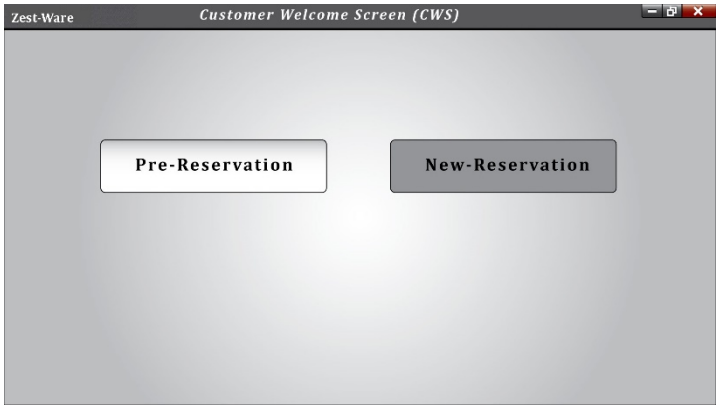
New-Reservation

The CWS will display two options one for pre-reservation and the other for new-reservation. The image below shows it:



If the user wants to make new reservation, then the user will tap on the new-reservation, so, this means the $tapCounter = 1$ (this is only used here for explaining and keeping count of how many taps would each function or operation need).

The image below shows the first tap:

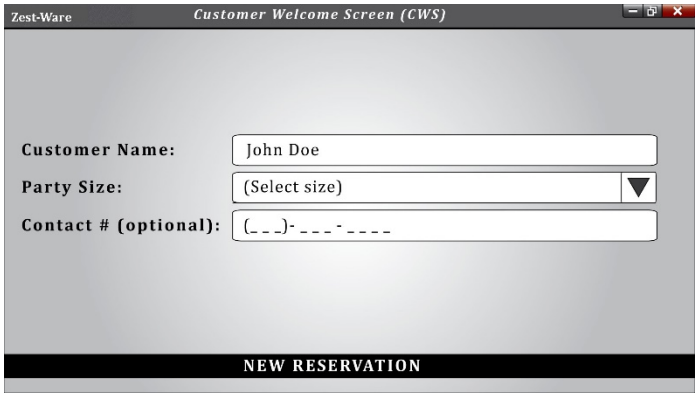


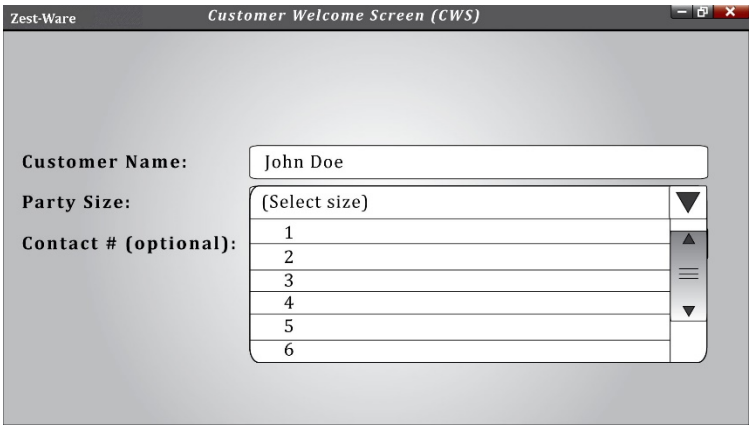
After the first tap on the new-reservation the CWS would prompt the user for input of name, party size, and contact # (optional). When the user taps on the column for Customer Name, the system would display a keyboard through which the user will enter their name, as shown in the image below:

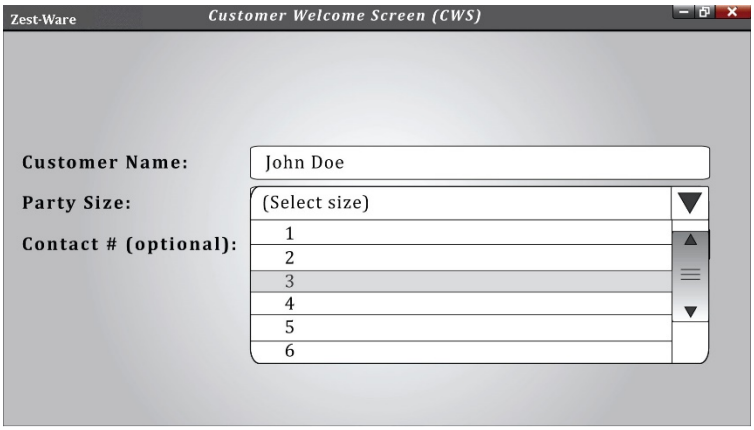
This means one tap for selecting the column $tapCounter = 2$, and the taps on keyboard depending upon the length of the name of the customer. This can be reduced in such a way that the system directly displays the keyboard, so the customer does not have to tap on the column to bring up the keyboard, keeping the $tapCounter' = 1$.

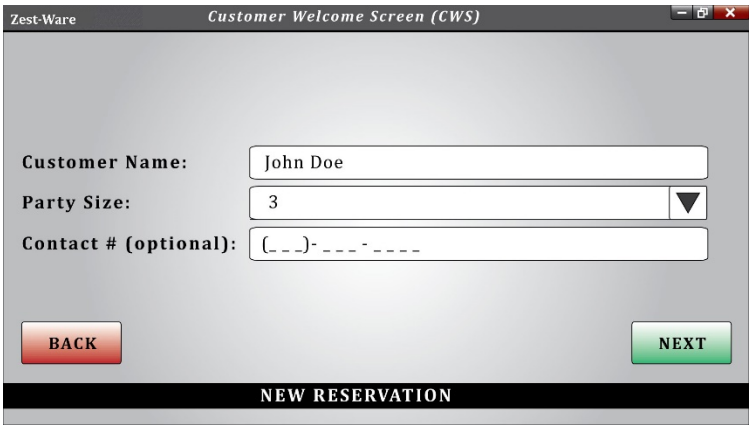
When the customer is finished entering their name they would tap the enter key on the keyboard to let the system know. So, this would make the total number of taps as $tapCounter' = 2 + x$, where, x is the number of taps of the customer's name.

The scroll down of the party size column would bring a drop-down menu from which a customer can select the party size, this would already require less number of taps compared to the tapping of the column, then system displaying keyboard, and then typing in the number of their party size. The images below show how the party size can be selected.





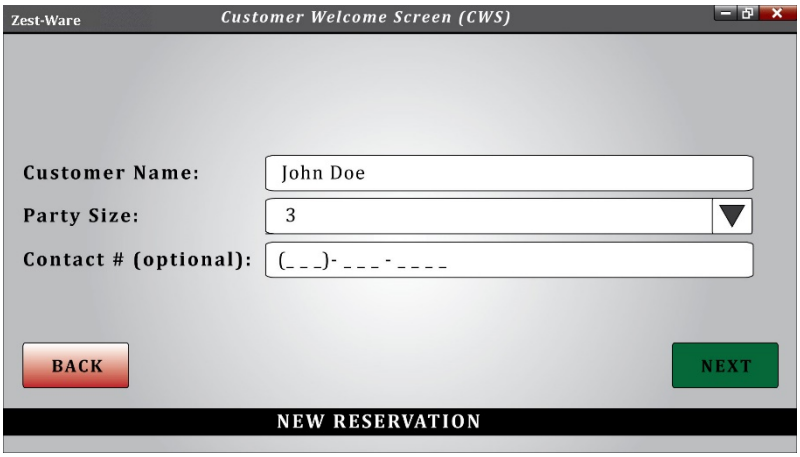


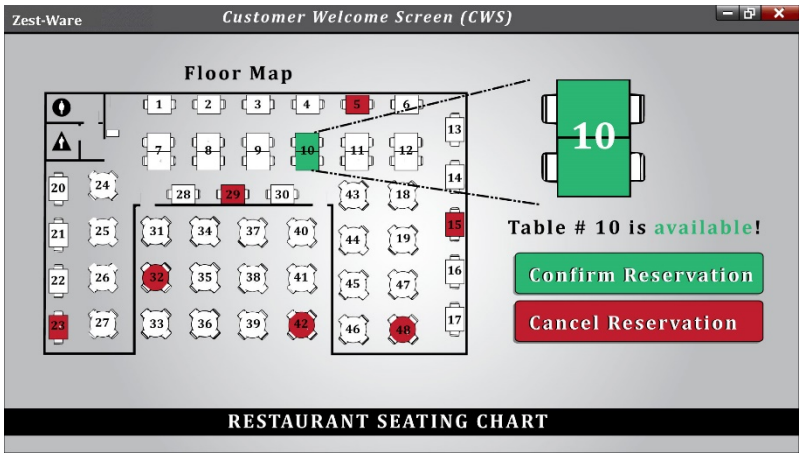


So, to keep track of the taps, this would be one tap for the drop-down, then scroll up and down for the drop-down menu, and a tap for selecting the appropriate number. So, the best or least number of taps would be $tapCounter' = 5 + x$.

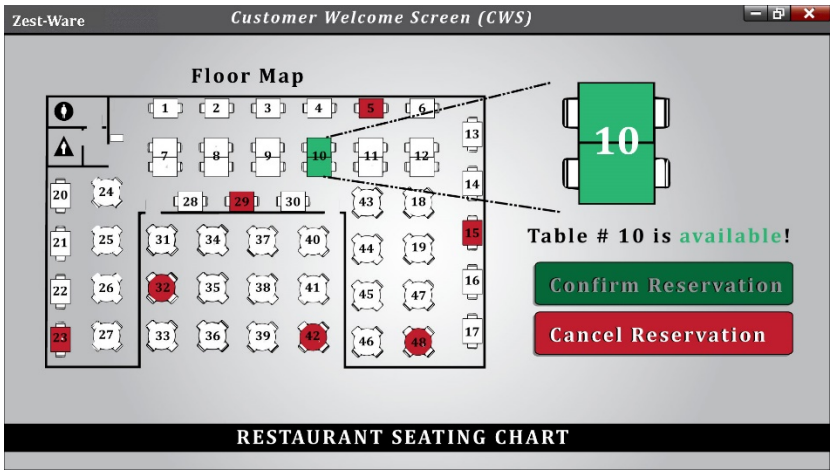
As, the contact # is optional so we can skip that part for now (in general the contact # would take 12 number of taps. 10 taps for the 10 digits contact number, 1 tap for selecting the column, and 1 tap when done).

When the user is done entering the necessary information, the system would be waiting for the next response from the user, so the user would tap NEXT, and the system would check for the availability of tables, and would display a floor map, the images below show these steps:

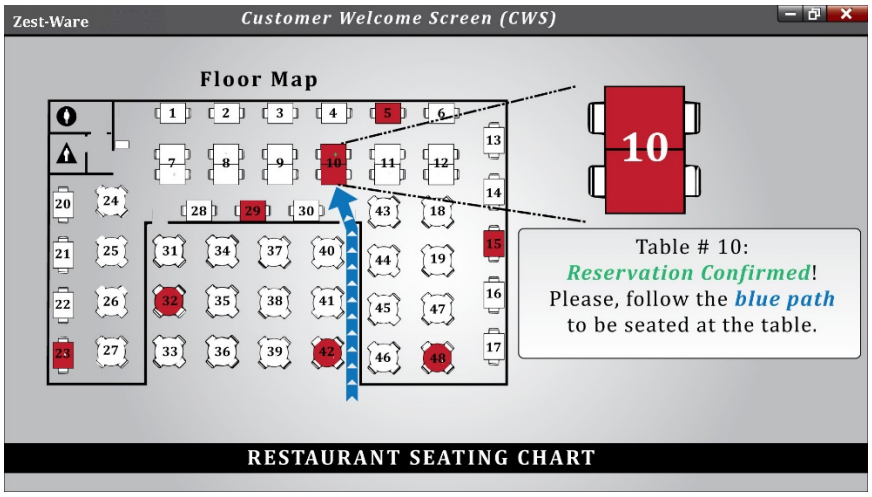




So, $tapCounter' = 6 + x$, at this stage. The system would display the available table, and would wait for the user to confirm reservation. So, $tapCounter' = 7 + x$ including the tap for Confirm Reservation shown below:



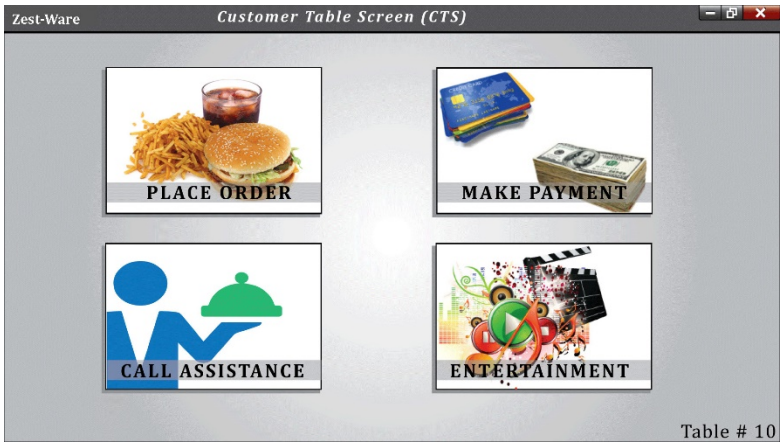
After the tap of the confirm reservation the system would display the path to the table by displaying the floor map shown in the image below, and the user wouldn't have to interact with the CWS by tapping anything, the screen would change back to pre-reservation and new-reservation after few seconds.

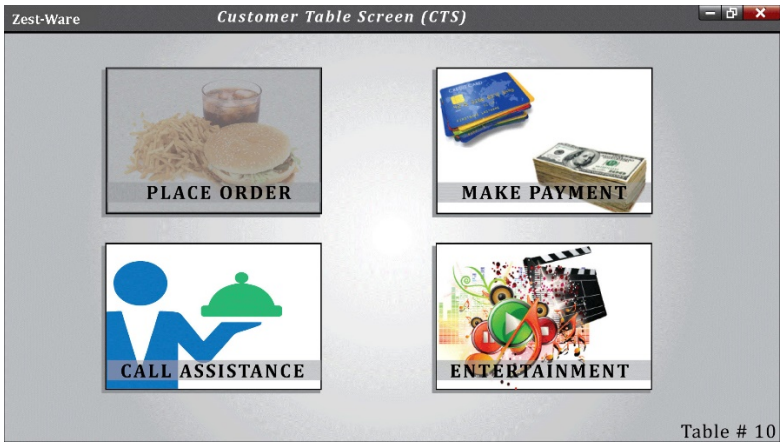


So, the number of taps it took was 7 plus whatever taps the customer's name took.

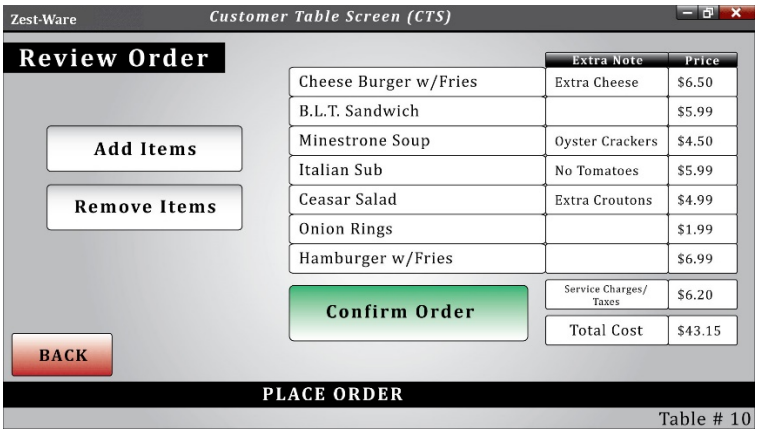
Placing Order

The CTS displays an icon for placing order the user would need one tap to go into the place order, the tap and the icons are shown in the two images below:



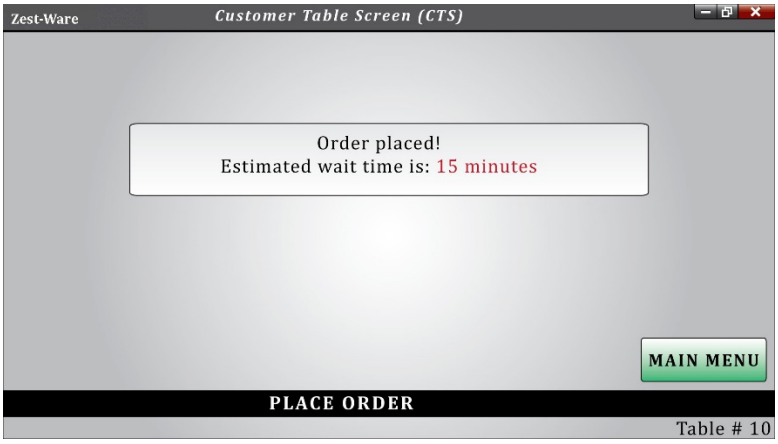


So, the $tapCounter = 1$ at this point. The CTS will display the icons inside the place order, which include: food, drink, extras, etc. The user will continue tapping on the icons and menu items for placing an order, and by tapping BACK button/icon to go back to the previous menu and choose different options. This procedure is totally customer dependent like it depends on what the customer is ordering, how much they are ordering, so will assume x number of taps for this entire procedure of selecting menu items, adding/removing items. When the customer is done with their order, the system would display the review of their order and assume the customer doesn't want to make any more changes, or even if they are then it will be added to the x number of taps. The next two images show the place order menu and the review order screen:



So, at this point the $tapCounter = 1 + x$. There would be one more tap for the confirmation of order, the system would display the estimated wait time, and the customer would be done with placing of order in $2 + x$ taps. Where, x is the number of taps customer took while adding/removing items. In general, it would take like 2 to 3 taps for placing one item in the order from the menu (this is only for the x).

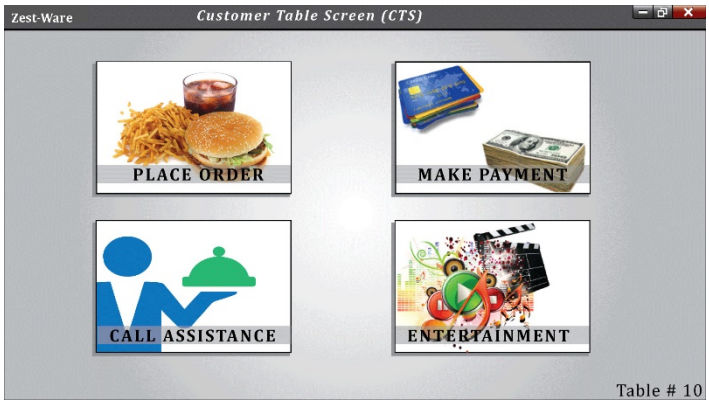
The image below shows the message displayed by the system after order has been placed.

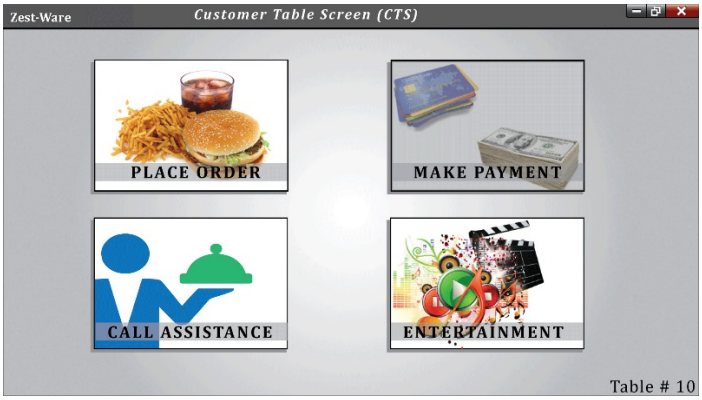


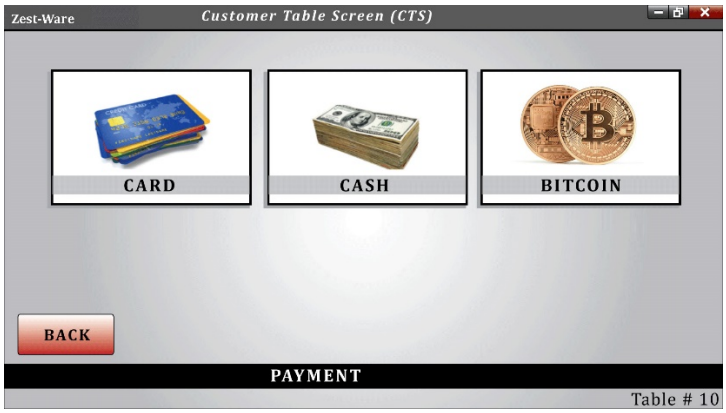
There has been little or less modification in the user interface for making new-reservation and placing order, as it was already optimized to have the least user effort and best ease-of-use.

Bitcoin Payment

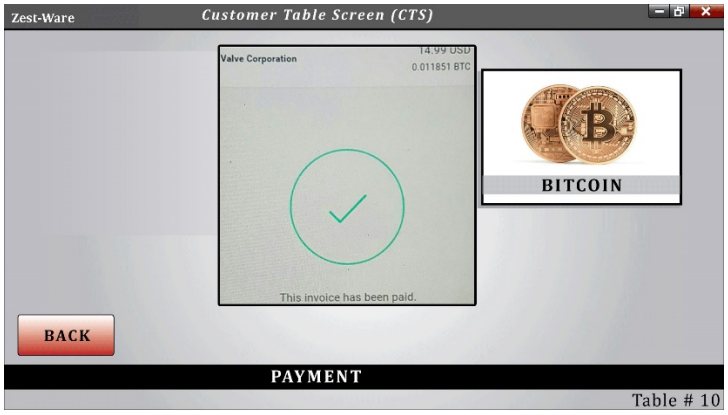
To make the payment via bitcoin, the user/customer would tap on the make payment icon on the CTS, then three options would be displayed i.e. card, cash, and bitcoin. After the bitcoin payment, the QR code would pop up, the customer would use their bitcoin client to scan the code. When the payment is successfully received on the bitpay server it will show as a green check mark. The system would display payment complete. These would require 3 taps and the scanning of QR code. This is illustrated in the images below:






Zest-Ware
Customer Table Screen (CTS)
CARD
CASH
BITCOIN
BACK
PAYMENT
Table # 10


Zest-Ware
Customer Table Screen (CTS)
bitpay
Awaiting Payment...
14:38
Valve Corporation
14.99 USD
0.011851 BTC
Scan
Copy
BITCOIN
BACK
PAYMENT
Table # 10


Zest-Ware
Customer Table Screen (CTS)
Valve Corporation
14.99 USD
0.011851 BTC
This invoice has been paid.
BITCOIN
BACK
PAYMENT
Table # 10

5.ii. Manager

Our initial screen mock-ups from the first report have not changed since we believe they optimize user effort. The following are the use cases we will be implementing for the first demo:

UC - 02 EditShifts
UC - 04 ShowSurveyData
UC - 08 CheckInventory
UC - 14 AccessPortal (will also show employees accessing the portal)

5.iii. Kitchen

For the GUI of the kitchen aspect of the software, the initial designs do not need any significant modifications. Besides just making the graphics more sleek and professional looking, the actual structure and substance of the GUI is very simple and pretty easy to navigate. Every action that needs to be performed in this section of the software only takes three or four taps/clicks, so at this time there is not really anything that we could change.

Part 3.6: Design of Tests

6.i. Customer

a. Specified Tests

PreReservation:

Test-case Identifier:	C-TC-1
Use Case Tested:	UC-16, main success scenario(Valid pre-reservation)
Pass/fail Criteria:	A valid name and party size receiving "Thank you"(PASS) Invalid name or party size displayed "Thank you" (FAIL)
Input Data:	Customer name:string, party size: int

Test Procedure:	**Expected Result:**
Step 1. Customer selects the pre-reservation icon on the CWS	The CWS changes to the next screen where the user is prompted to enter their Name and Party Size
Step 2. The customer inputs their name and party size correctly into the CWS and presses the "Next" icon	The Customer- after being verified as having an existing reservation in the database - is welcomed by the CWS to the Restaurant and the screen displays the location of their table
Step 3. The customer hits confirm and walks to their table	The CWS says "Thank you" before timing out in approximately 30 seconds or when it senses touch input.

Test-case Identifier: C-TC-1	
Use Case Tested: UC-16, alternate success scenario (Valid pre-reservation but incorrect input)	
Pass/fail Criteria: A valid name and party size receiving “Thank you”(PASS) Invalid name or party size displayed “Thank you” (FAIL)	
Input Data: Customer name:string, party size: int	
Test Procedure:	**Expected Result:**
Step 1. Customer selects pre-reservation icon on the CWS	The CWS changes to the next screen where the user is prompted to enter their Name and Party Size
Step 2. The customer inputs their name and party size incorrectly into the CWS and presses the “Next” icon	The Customer- which is not able to be verified by the CWS- is prompted to enter their information again into the CWS or “Back” to make a new reservation
Step 3. The customer inputs their name and party size a second time-this time correctly-into the CWS and presses “Next”	The Customer is welcomed by the CWS to the Restaurant and the screen displays the location of their table
Step 4. The customer hits confirm and walks to their table	The CWS says “Thank you” before timing out in approximately 30 seconds or when it senses touch input.

NewReservation:

Test-case Identifier:	C-TC-2
Use Case Tested:	UC-17, main success scenario(Open seats)
Pass/fail Criteria:	Seats are available: CWS says "Thank you" (PASS) Seats are unavailable: CWS says "Thank you"(FAIL)
Input Data:	Customer name:string, party size:int

Test Procedure:	**Expected Result:**
Step 1. Customer selects the new-reservation icon on the CWS	The CWS changes to the next screen where they are prompted to enter their name
Step 2. The customer inputs their name and party size into the CWS and hits the "next" icon	The software will check to see if there are any available tables. If there are the CWS will display all the available tables to the customer
Step 3. The customer selects a table on the CWS that is highlighted in green to represent its available status	The CWS will prompt the user to verify that the selected table is the one they want, they will have an option to select "confirm" or "cancel"
Step 4. The customer chooses "confirm"	The CWS says "Thank you" and notifies the customer they may sit at Table X before timing out in 30 seconds or when it senses touch input.

BitcoinPayment:

Test-case Identifier:	C-TC-3
Use Case Tested:	UC-40, main success scenario(Customer makes payment)
Pass/fail Criteria:	If the tab is successfully pulled(PASS) The tab is not found in database(FAIL)
Input Data:	amount: Currency

Test Procedure:	**Expected Result:**
Step 1. Customer selects "Make a payment" icon on the CTS	The CTS changes to the next screen where the customer has the following options: "Card", "Cash", or "Bitcoin"
Step 2. The customer chooses "Bitcoin" option	The current tab is pulled from the database, and the total fiat price is used with the bitpay API. Once the tab is successfully pulled from our database and the API call is made, the User will be prompted with a QR code to scan.
Step 3. The customer opens their Bitcoin client, and scans the QR code on the CTS. The payment is then sent to the displayed address.	As soon as the payment is received the bitpay server verifies and then displays a green check mark to the user letting them know the payment was successful.

OrderItems:

Test-case Identifier: C-TC-4	
Use Case Tested: UC-23, main success scenario(Customer orders from menu)	
Pass/fail Criteria: Customer orders: orderedList saved to DB (PASS) Customer orders: orderedList not saved to DB (FAIL)	
Input Data: orderedList: items	
Test Procedure:	**Expected Result:**
Step 1. Customer selects "Place order" on the CTS	The CTS moves to the next screen where the customer has the options "Food", "Drinks", "Side Orders", "Desserts", and "Specials"
Step 2. Customer Selects "Food" icon and adds a food item	The food item should be added to the orderedList which stores the customers order
Step 3. Customer selects back, then presses the "Drink" icon and adds a drink to their order	The drink item should be added to the orderedList with the food item they selected.
Step 4. Customer goes back to the main ordering screen and chooses the place order button	The customer is taken to the Review Order page of the CTS where they are shown the food item and drink item they chose with the total cost.
Step 5. Customer selects the "Confirm Order" button to confirm and send the order to the kitchen for processing	The orderedList should be saved and sent to the database, where the kitchen is able to pull for processing. The CTS should say "Order placed" and say processing before providing an estimated wait time.

a. Test Coverage:

The current tests are meant to cover the cases that can be presented to the Client themselves, and do not yet require the components of the other modules to be integrated. The goal of the first demo is to show the client that progress has been made, and give them an idea of our vision for their Software goals. The tests have thus been designed in such a way to show the Client the most important aspects of the Software. The traceability matrix was also helpful to make sure we are covering the most important classes in our test cases.

b. Integration Testing Strategy:

We have divided the ZestWare software system into 3 modules that will together create the software system – customer, kitchen, and manager. The customer module is focused on the customer centric activities and is responsible for designing and implementing the Customer Table Screen(CTS), Customer Welcome Screen(CWS), the Customer Waiting Area Screen(CWAS), the floor map, Customer surveys, implementation of Bitcoin payments using Bit pay API, and potentially credit cards using stripe payment. The kitchen is focused on anything kitchen or food related and includes the Chef, Waiter, and Busboy. The kitchen module is responsible for the Kitchen Screen(KS), Waiter and Busboy interface, Notification to the busboy of clean and dirty table, and . Finally, the manager module is focused on making the day to day responsibilities of the manager easier so that they can focus on sales. The manager component is responsible for the Employee Login by facial recognition, an inventory tracking system, making sure the manager is only able to access the login portal, and saving the Survey data.

The most sensible type of integration testing for our group will be bottom up integration testing. This type of integration testing starts by forming a hierarchy that puts the units without any dependencies first. As a group, we have decided the most efficient way to implement this software was to divide the work into equal modules that can be first demonstrated on their own. The bottom up integration testing method is the most appropriate for this approach because each subgroup will be able to test their own components. As the hierarchy progresses, we will meet at a point where our components will need to be integrated together - this final integration of components will form the complete Zestware system.

The nonfunctional system requirements will be able to be tested on their own for each module. For example, a nonfunctional requirement of the Chef is to be able to remove an item from the queue in 2 clicks or less. So, when designing the interface, the Kitchen module will test their interface to adhere to this user story. Likewise, each module will verify their programs are operating system independent. Once the system is integrated as a whole, there will be further testing to verify each of the modules adheres to its nonfunctional and system requirements.

6.ii. Manager

a. Specified Tests

AccessPortal:

Test-case Identifier:	M-TC-1
Use Case Tested:	UC-14, main success scenario
Pass/fail Criteria:	The test passes if the user enters the appropriate login credentials, using less than the maximum allowed number of unsuccessful attempts
Input Data:	Manager username, Manager PIN

Test Procedure:	**Expected Result:**
Step 1. Type in an incorrect PIN and a valid username	System deploys error message indicating failure; records unsuccessful attempt in the database; prompts the user to try again
Step 2. Type in the correct PIN and an invalid username	System deploys error message indicating failure; records unsuccessful attempt in the database; prompts the user to try again
Step 3. Type in the correct PIN and valid username	System allows user to access portal; records successful attempt in the database

ShowSurveyData:

Test-case Identifier:	M-TC-2
Use Case Tested:	UC-04, main success scenario
Pass/fail Criteria:	The test passes if the user can see all customer surveys after accessing the Manager Portal, and can filter out surveys based on time/ratings
Input Data:	Select "Survey Data" from Manager Portal

Test Procedure:	**Expected Result:**
Step 1. Include no customer survey data	System deploys error message indicating failure; Tells user that there are no surveys on file
Step 2. Include survey data	System indicates success by showing user data; Data is automatically sorted by most recent entries
Step 3. Select "Sort By" and choose "Rating"	System indicates success by showing sorted data; Data is now sorted from highest rating to lowest rating

EditShifts:

Test-case Identifier:	M-TC-3
Use Case Tested:	UC-02, main success scenario
Pass/fail Criteria:	The test passes if the user enters the appropriate login credentials, finds the specified employee, and changes the date and time that the employee is working.
Input Data:	Manager username, Manager PIN, Employee Name (First or Last) or Employee ID number

Test Procedure:	**Expected Result:**
Step 1. Type in a correct PIN and a valid username	System allows Manager access to the shifts calendar.
Step 2. Choose the name of the employee working on a specific day.	System displays employee and the date and time they work.
Step 3. Manager enters a new date and time for the employee to work from drop down menu.	System responds by updating employee working info and confirms with manager that the employee's work time has been updated.

CheckInventory:

Test-case Identifier:	M-TC-4
Use Case Tested:	UC-08, main success scenario
Pass/fail Criteria:	The test passes if the user chooses multiple items, stores them in a cart, and finally "purchases" the items. The system will confirm purchase. The test fails if: items could not be purchased but restaurant can afford the items; items total is above restaurant savings total and items were purchased; etc.
Input Data:	Select "Inventory Check" from Manager Portal

Test Procedure:	**Expected Result:**
Step 1. Manager selects the Inventory check portal.	System displays inventory check interface.
Step 2. Manager chooses items and item quantity to add to "cart."	System continuously adds items to the cart after manager has chosen quantity.
Step 3. Manager chooses "Purchase" button.	System prompts if purchases are "ok." Once confirmed, system confirms that items are purchased, records purchase in database, and deducts purchase total from restaurant balance.

b. Test Coverage

The test cases are aimed at highlighting new features which help with manager's responsibilities. In doing so, we will be showing how the classes in the manager portion interact with each other. For example, our implementation that involves checking inventory connects the authorization, manager interface, database, controller, and inventory classes. This will be used to convince the client that the manager part of Zest-Ware is able to work independently of other aspects of the software.

c. Integration Testing Strategy:

Due to the structure of our subgroups, integration should not be difficult amongst each part. All data will be sourced from a shared database, where each subgroup will utilize the necessary tables. If another group needs access to another table, it will not be difficult to implement this in the final product. For example, the manager will require access to customer surveys. The customer group will be collecting post-meal surveys from customers for the benefit of the restaurant. Without interrupting or changing the data of the surveys, the manager subgroup will be able to access data for viewing only, thus avoiding any potential issues within the database medium.

6.iii. Kitchen

a. Specified Tests

Clean Table:

Test-case Identifier: K-TC-4 **Use Case Tested:** **Pass/fail Criteria:** A valid name and table number "Thank you" (Pass) A invalid name or table number ("Invalid entry") **Input Data:** Busboyr name: string, tableNumber int	
Test Procedure:	**Expected Result:**
Step 1. Waiter selects clean table icon on the CWS	The CWS changes to the next screen where the user is prompted to enter their Name and table number
Step 2. Busboy accepts the request and begins to clean the table	The Busboy- which is not able to be verified by the CWS- is prompted to enter their information again into the CWS or "Back" to reenter their credentials
Step 3. Busboy confirms that the table has been cleaned and is ready for the next customer.	The Busboy is welcomed by the CWS to the Kitchen and the screen displays the location of the table needed to be cleaned .

Deliver Meal:

Test-case Identifier: K-TC-1 **Use Case Tested:** UC-35, Deliver **Pass/fail Criteria:** The meal is removed from the waiter list and the table the meal goes to has its status updated if that was the last item on the waiter list that table "Thank you" (Pass) A invalid name or table number ("Invalid entry") **Input Data:** fooditem: (item is a custom class in our software)	
Test Procedure:	**Expected Result:**
Step 1. The waiter delivers the food in real life and selects to remove the item. Step 2. The software goes through the list and checks if there is any more items for that table	The item is removed and the software begins making checks for the table status If there are more orders that table, the table is still awaiting order. Otherwise the table is still waiting for more food. .

Prepare Meal:

Test-case Identifier: K-TC-2 **Use Case Tested:** UC-36, Prepare Meal **Pass/fail Criteria:** The meal is removed from the waiter list and the table the meal goes to has its status updated if that was the last item on the waiter list that table "Thank you" (Pass) A invalid name or table number ("Invalid entry") **Input Data:** fooditem: (item is a custom class in our software)	
Test Procedure:	**Expected Result:**
Step 1. The waiter delivers the food in real life and selects to remove the item. Step 2. The software goes through the list and checks if there is any more items for that table	The item should automatically appear on the chef's list when the signal is sent from the customer side If that is the first item, the table is now waiting for food If it is not that is supposed to already by it status The item is removed from the list and sent to waiter list for delivery .

Part 3.7: Project Management and Plan of Work

7.a. Merging the contributions from individual team members:

Compiling the final copy of this report was not too difficult compared to report 1. When report 1 was being created, every subgroup uploaded multiple files in the same folder. The name of the files were not very descriptive and thus resulted in confusion as to which file was the final copy to be submitted. However, the procedure we followed for report 2 was different. Every subgroup created mini folders within the main folder and removed old files as modified versions of the files were uploaded. In report 1, the formatting of the tables and the spacing/indexing was not consistent. The customer subgroup had to spend a lot of additional time to format everyone's work to make it more consistent and cohesive. In report 2, to ensure uniform formatting, the subgroups used one style of table formatting and inserted their information. From doing this, less time was needed to compile the final copy of the report from everyone's work. We also decided to write the information out in separate word documents and upload them in the folders on the drive instead of google docs. This is because google docs makes unnecessary changes the documents and causes the report to not be consistent in terms of formatting.

7.b. Project coordination and progress report:

Customer Module:

The use cases that our group aims to complete for demo:

- UC-16 (preReservation)
- UC-17 (newReservation)
- UC-23 (orderItems)
- UC-30 (cardPayment)
- UC-40 (bitcoinPayment)

What is already function, what is currently being tackled?

We are currently looking into the bitpay API and stripe API for setting up bitcoin and credit card transactions. We have also discussed building on top of a previous project as has been suggested to us. The project from 2013 gravy Express meets some of our goals as it is operating system independent so it is an ideal candidate. Furthermore, as a group we have created an AWS account for hosting our website/project.

Manager Module:

The use cases that our group aims to complete for demo:

- UC - 02 EditShifts
- UC - 04 ShowSurveyData
- UC - 08 CheckInventory
- UC - 14 AccessPortal (will also show employees accessing the portal)

Kitchen Module:

The use cases that our group aims to complete for the demo:

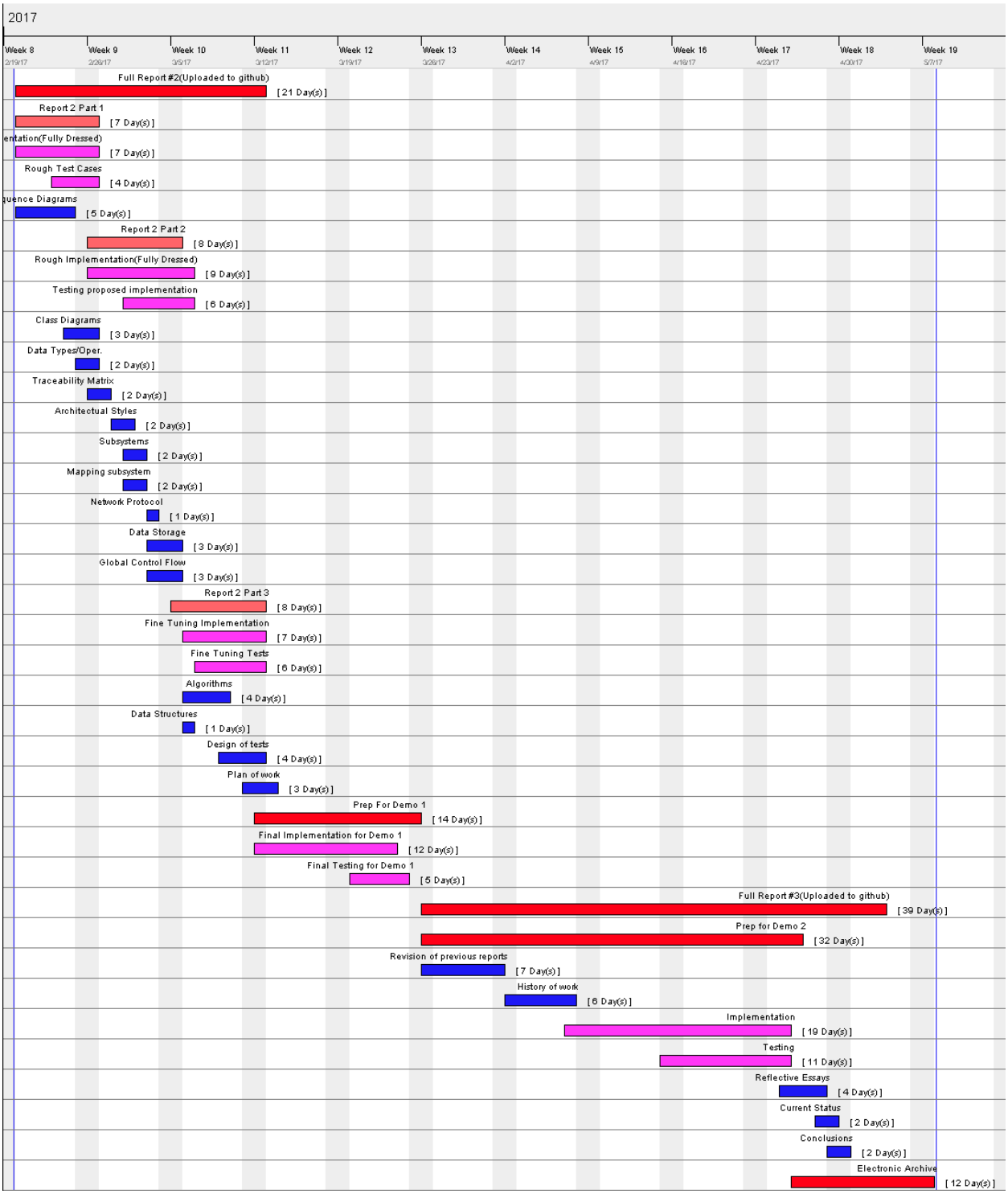
- UC-35 (DeliverMeal)
- UC-36 (PrepareMeal)
- UC-37 (CancelMeal)
- UC-38 (CleanTable)

What is already functioning, what is currently being tackled?

Currently, we are working on learning how to properly integrate AWS into our part of the project since neither of us has much experience with use. We need this functionality for many things obviously, but perhaps the most important is to receive order information from the customer end. After this is tackled, the chef and waiter lists are going to developed for the first demo. Then, the modified floor map designed especially for kitchen use is going to be implemented followed by the clock-in and schedule viewer mechanic.

7.c. Plan of work:

Name	Begin date	End date
Full Report #2(Uploaded to github)	2/20/17	3/12/17
Report 2 Part 1	2/20/17	2/26/17
Rough Implementation(Fully Dressed)	2/20/17	2/26/17
Rough Test Cases	2/23/17	2/26/17
Sequence Diagrams	2/20/17	2/24/17
Report 2 Part 2	2/26/17	3/5/17
Rough Implementation(Fully Dressed)	2/26/17	3/6/17
Testing proposed implementation	3/1/17	3/6/17
Class Diagrams	2/24/17	2/26/17
Data Types/Oper.	2/25/17	2/26/17
Traceability Matrix	2/26/17	2/27/17
Architectual Styles	2/28/17	3/1/17
Subsystems	3/1/17	3/2/17
Mapping subsystem	3/1/17	3/2/17
Network Protocol	3/3/17	3/3/17
Data Storage	3/3/17	3/5/17
Global Control Flow	3/3/17	3/5/17
Report 2 Part 3	3/5/17	3/12/17
Fine Tuning Implementation	3/6/17	3/12/17
Fine Tuning Tests	3/7/17	3/12/17
Algorithms	3/6/17	3/9/17
Data Structures	3/6/17	3/6/17
Design of tests	3/9/17	3/12/17
Plan of work	3/11/17	3/13/17
Prep For Demo 1	3/12/17	3/25/17
Final Implementation for Demo 1	3/12/17	3/23/17
Final Testing for Demo 1	3/20/17	3/24/17
Full Report #3(Uploaded to github)	3/26/17	5/3/17
Prep for Demo 2	3/26/17	4/26/17
Revision of previous reports	3/26/17	4/1/17
History of work	4/2/17	4/7/17
Implementation	4/7/17	4/25/17
Testing	4/15/17	4/25/17
Reflective Essays	4/25/17	4/28/17
Current Status	4/28/17	4/29/17
Conclusions	4/29/17	4/30/17
Electronic Archive	4/26/17	5/7/17


2017
Week 8
2/19/17
Week 9
2/26/17
Week 10
3/5/17
Week 11
3/12/17
Week 12
3/19/17
Week 13
3/26/17
Week 14
4/2/17
Week 15
4/9/17
Week 16
4/16/17
Week 17
4/23/17
Week 18
4/30/17
Week 19
5/7/17
Full Report #2(Uploaded to github) [21 Day(s)]
Report 2 Part 1 [7 Day(s)]
entation(Fully Dressed) [7 Day(s)]
Rough Test Cases [4 Day(s)]
quence Diagrams [5 Day(s)]
Report 2 Part 2 [8 Day(s)]
Rough Implementation(Fully Dressed) [9 Day(s)]
Testing proposed implementation [6 Day(s)]
Class Diagrams [3 Day(s)]
Data Types/Oper. [2 Day(s)]
Traceability Matrix [2 Day(s)]
Architectual Styles [2 Day(s)]
Subsystems [2 Day(s)]
Mapping subsystem [2 Day(s)]
Network Protocol [1 Day(s)]
Data Storage [3 Day(s)]
Global Control Flow [3 Day(s)]
Report 2 Part 3 [8 Day(s)]
Fine Tuning Implementation [7 Day(s)]
Fine Tuning Tests [6 Day(s)]
Algorithms [4 Day(s)]
Data Structures [1 Day(s)]
Design of tests [4 Day(s)]
Plan of work [3 Day(s)]
Prep For Demo 1 [14 Day(s)]
Final Implementation for Demo 1 [12 Day(s)]
Final Testing for Demo 1 [5 Day(s)]
Full Report #3(Uploaded to github) [39 Day(s)]
Prep for Demo 2 [32 Day(s)]
Revision of previous reports [7 Day(s)]
History of work [6 Day(s)]
Implementation [19 Day(s)]
Testing [11 Day(s)]
Reflective Essays [4 Day(s)]
Current Status [2 Day(s)]
Conclusions [2 Day(s)]
Electronic Archive [12 Day(s)]

7.d. Breakdown of responsibilities:

List of the names of modules and classes that each team member is currently responsible for developing, coding, and testing:

The customer subgroup (Nathan Morgenstern, Fahd Humayun, and Shehpar Sohail) will be responsible for developing, coding, and testing the following classes/modules:

- **Customer Profile-**Shehpar Sohail is responsible for developing, coding, and testing for this class/module
- **Reservation Verifier-**Fahd Humayun is responsible for developing, coding, and testing for this class/module
- **Table Status Checker-**Nathan Morgenstern is responsible for developing, coding, and testing for this class/module
- **Communicator-**Shehpar Sohail is responsible for developing, coding, and testing for this class/module
- **Database-**Fahd Humayun is responsible for developing, coding, and testing for this class/module
- **Controller-**Nathan Morgenstern is responsible for developing, coding, and testing for this class/module
- **Order Queue-**Shehpar Sohail is responsible for developing, coding, and testing for this class/module
- **Table Status Editor-**Fahd Humayun is responsible for developing, coding, and testing for this class/module
- **Floor Map-**Nathan Morgenstern is responsible for developing, coding, and testing for this class/module
- **Customer Welcome Screen-**Shehpar Sohail is responsible for developing, coding, and testing for this class/module
- **Customer Table Screen-**Fahd Humayun is responsible for developing, coding, and testing for this class/module
- **Payment-**Nathan Morgenstern is responsible for developing, coding, and testing for this class/module
- **Payment Verifier-**Shehpar Sohail is responsible for developing, coding, and testing for this class/module
- **Menu-**Fahd Humayun is responsible for developing, coding, and testing for this class/module

The Manager subgroup (Ama Freeman, Raphaelle Marcial) will be responsible for developing, coding, and testing the following classes/modules:

- **Employee Interface** - Raphaelle Marcial is responsible for developing, coding, and testing for this class/module.
- **Manager Interface** - Ama Freeman is responsible for developing, coding, and testing for this class/module.
- **Authorization -** Both Raphaelle Marcial and Ama Freeman are responsible for developing, coding, and testing the this class/module.
- **Shift Table -** Raphaelle Marcial is responsible for developing, coding, and testing for this class/module.
- **Inventory** - Ama Freeman is responsible for developing, coding, and testing for this class/module.
- **Controller -** Both Raphaelle Marcial and Ama Freeman are responsible for developing, coding, and testing the this class/module.
- **Database -** Both Raphaelle Marcial and Ama Freeman are responsible for developing, coding, and testing the this class/module.

The Kitchen subgroup (Dwayne Anthony, Alex Dewey) will be responsible for developing, coding, and testing the following classes/modules:

- **Waiter List-** Alex Dewey is responsible for developing, coding, and testing for this class/module
- **Chef List-** Alex Dewey is responsible for developing, coding, and testing for this class/module
- **Clock-in-** Dwayne Anthony is responsible for developing, coding, and testing for this class/module
- **Employee Schedule View-** Dwayne Anthony is responsible for developing, coding, and testing for this class/module
- **Database-** Dwayne Anthony is responsible for developing, coding, and testing for this class/module
- **Controller-** Dwayne Anthony is responsible for developing, coding, and testing for this class/module
- **Kitchen Table Status Editor-** Alex Dewey is responsible for developing, coding, and testing for this class/module
- **Kitchen Floor Map-** Alex Dewey is responsible for developing, coding, and testing for this class/module

Who will coordinate the integration?

For the first demo, it is our goal to be able to present the components of each individual module stand alone. The final integration of the software should be completed by demo 2. The entire team is responsible for the integration as each subgroup will be working with the database, github repository, and integration testing of their features. The github repository and e-Archive will be managed by Nathan Morgenstern although contributions will be made to subgroup branch by all team members.

Who will perform and integration testing? (The assumption is that the unit testing will be done for each unit by the student who developed that unit):

Integration testing will be done using the bottom up testing method as discussed. Therefore, each module-Customer, Manager, Kitchen - will be responsible for writing their own integration tests for their respective classes and features. The integration will ideally be split up evenly in each module, and will be revised case by case. The final integration testing of the ZestWare system as a whole will be split evenly between the entire group(ideally).

References

Anon, 2017. Unified Modeling Language. *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Unified_Modeling_Language#Interaction_diagrams [Accessed February 25, 2017].

Anon, 2017. Sequence diagram. Wikipedia. Available at: https://en.wikipedia.org/wiki/Sequence_diagram [Accessed February 25, 2017].

Miles, R. & Hamilton, K., 2006. Learning UML 2.0: Beijing: O'Reilly.

http://www.angelpos.ca/15-Touch-Screen-Monitor_p_9.html

http://www.touch4.com/touch-screen-kiosks/specsheets/specsheet_gp2_v4.pdf

Bandarpalle, S. et al., 2015. Food EZ.

Microsoft (2009). Chapter 3: Architectural patterns and styles. . Retrieved from https://msdn.microsoft.com/en-us/library/ee658117.aspx

JDBC architecture (the Java™ tutorials > JDBC(TM) database access > JDBC introduction). (1995). Retrieved March 5, 2017, from https://docs.oracle.com/javase/tutorial/jdbc/overview/architecture.html

Lesson: All about sockets (the Java™ tutorials > custom networking). (1995). Retrieved March 5, 2017, from http://docs.oracle.com/javase/tutorial/networking/sockets/index.html

Reserved, A. R., & Terms. (2012, October 23). Objectivity/DB programming concepts - persistent objects. Retrieved March 5, 2017, from http://support.objectivity.com/learning/objectivity/10_2_1/faq/concepts/persistent-objects

http://www.ece.rutgers.edu/~marsic/books/SE/book-SE_marsic.pdf page 143

https://bitpay.com/api

Patel, P. et al., Auto-Serve,

Cohen, Y. et al., GravyXpress: A Restaurant Management Software,

Anon, 2017. Unit testing. Wikipedia. Available at: https://en.wikipedia.org/wiki/Unit_testing [Accessed March 11, 2017].

2017. Insertion Sort. Wikipedia. Available at:

https://en.wikipedia.org/wiki/Insertion_sort [Accessed March 11, 2017].