Group 2

Restaurant Automation

Report 2

March 12, 2017

Project Team

Mit Patel

Raj Patel

Nill Patel

Dylan Herman

Prabhjot Singh

Moulindra Muchumari

Project Website: https://github.com/Mitbits

All team members contributed equally!

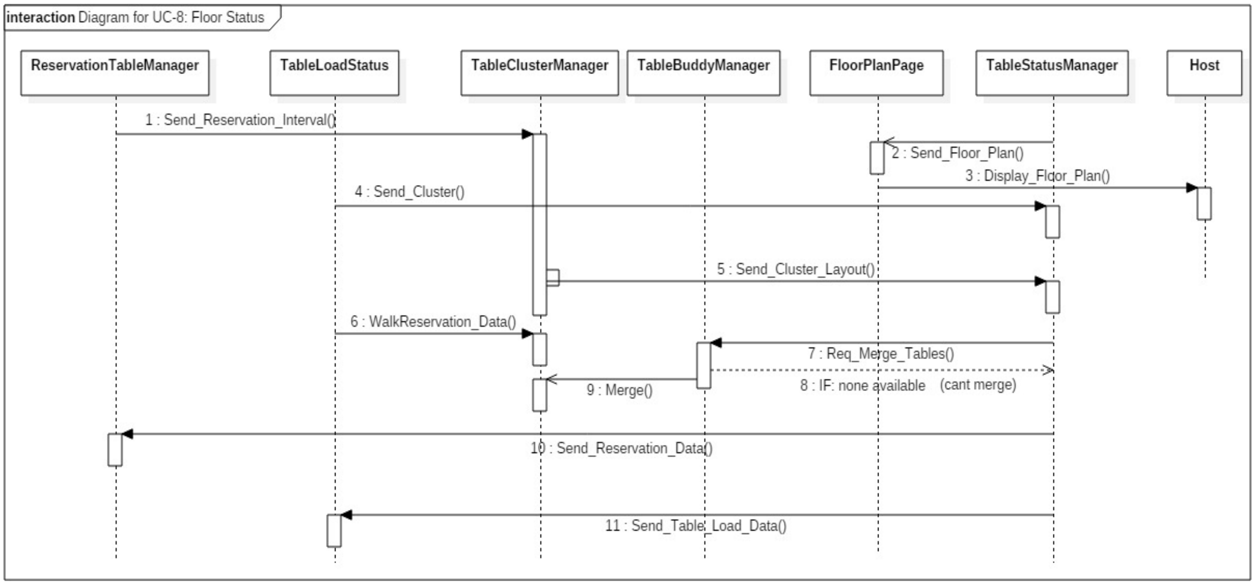
Table of Contents

Interaction Diagrams

Based on *Learning UML 2.0*, a *sequence diagram* should be used to represent a flow of messages, while a *communication diagram* is better for focusing on the connections between different participants within an interaction. The fully-dressed use cases we chose lean towards a flow of messages rather than a communications link and our system places more emphasis on the time of the interactions rather than the structural organization of the objects that send and receive messages. So it was best to use *sequence diagrams* to show our interaction diagrams, shown below:

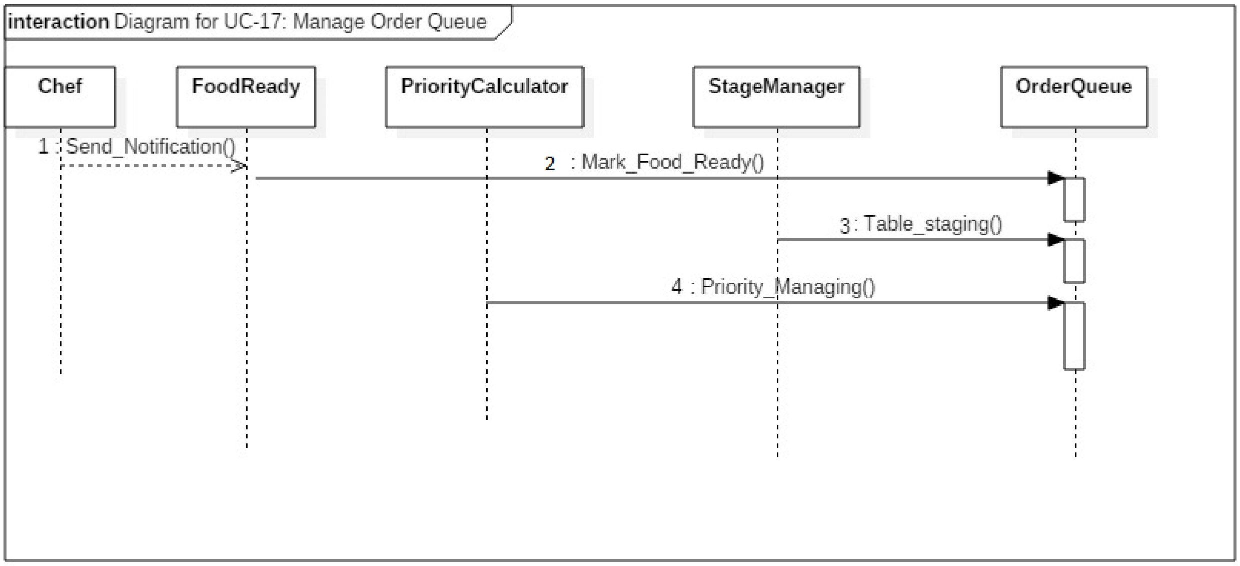
In each sequence diagram shown below, we are assuming each user is already logged into the system.

UC-8 Floor Status


interaction Diagram for UC-8: Floor Status
ReservationTableManager
TableLoadStatus
TableClusterManager
TableBuddyManager
FloorPlanPage
TableStatusManager
Host
1 : Send_Reservation_Interval()
2 : Send_Floor_Plan()
3 : Display_Floor_Plan()
4 : Send_Cluster()
5 : Send_Cluster_Layout()
6 : WalkReservation_Data()
7 : Req_Merge_Tables()
8 : IF: none available (cant merge)
9 : Merge()
10 : Send_Reservation_Data()
11 : Send_Table_Load_Data()

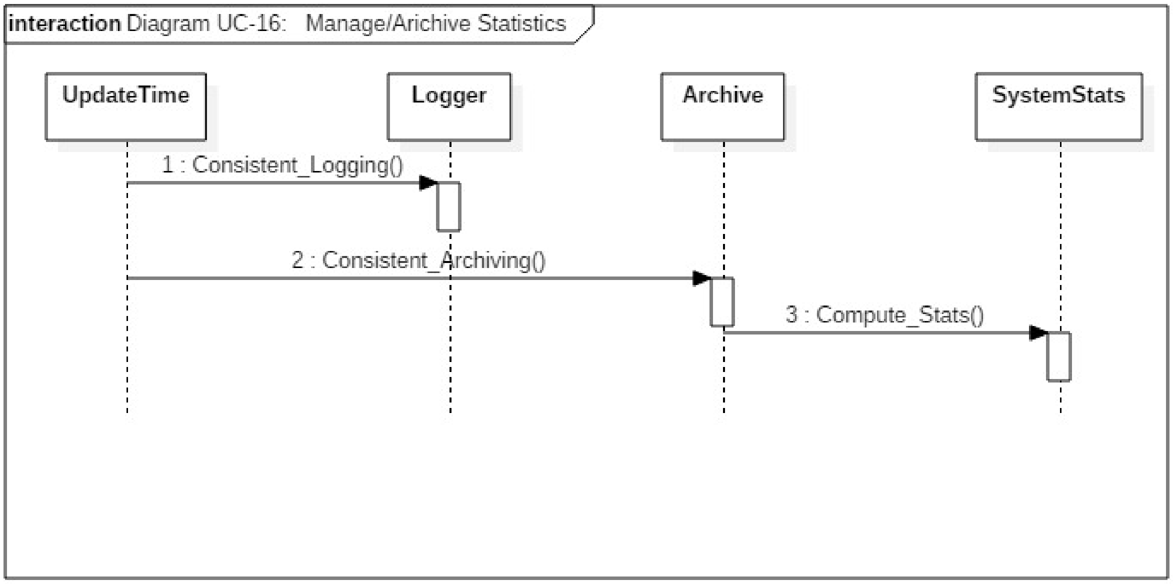
The Floor Status Use Case describes the interactions that take place for the host to be able to view an accurate display of the dining area floor plan. There are two types of tables: Reservation and Walk-in tables. Reservation tables can only be reserved within periodic intervals; this ensures there is enough time for each person to eat. However, a person can take a Reservation table through walkin if no one has reserved it at the start of the current interval; once the interval ends, the table is up for reservation again. Walkin tables are exclusive to people "walking in" only. Data is collected from past days and throughout the current day via the ReservationTableManager to decide the optimum interval for reservations, and data is collected via the TableLoadStatus to decide the optimum number of Reservation to Walk-in tables per Cluster. A Cluster is a grouping of tables of a given size. The status of each table: whether it is reserved, dirty (needs to be cleaned), taken, or free is tracked in real time. In addition, if there aren't enough tables of a certain size and type to satisfy a request, tables of a smaller size can be "buddied" up to form larger tables. All of this information is then forwarded to the floor plan display so the host can see the status of all tables in real time.

UC-17 Manage Order Queue



The Manage Order Queue use case describes the interactions that take place to order items within the queue. Items in the queue are prioritized based on the stage the items are assigned (i.e. appetizers, entrees, and desserts). In addition, a second priority calculation is computed based on the average service time it takes to process a table's order. Then, the chef marks each food item as completed.

UC-16 Manage/Archive Statistics

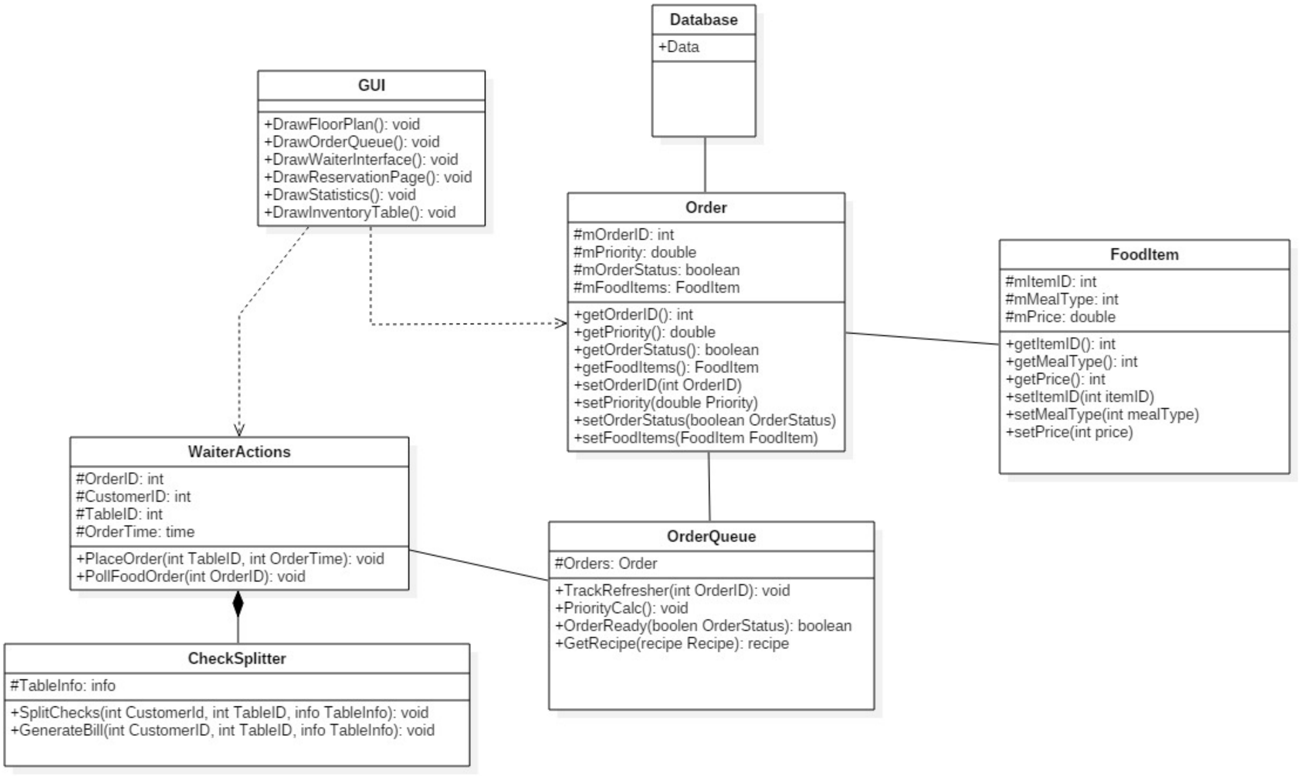

interaction Diagram UC-16: Manage/Arichive Statistics
UpdateTime
Logger
Archive
SystemStats
1 : Consistent_Logging()
2 : Consistent_Archiving()
3 : Compute_Stats()

The Manage/Archive Statistics use case describes the interaction that takes place whenever there is a new transaction. The system will get automatically updated once these transactions occur, As these values are updated, the system will periodically compute several statistics such as total profit, number of items sold, most popular dish, etc. All this information is available upon request through the system.

Interaction Diagram Summary:
We choose to design these interaction diagrams because they are the most important ones relating to our system design. The main component of the design is communication between employees and how to make it more efficient and automated. It wouldn’t be time efficient to do interactions for all of the use cases especially all the trivial ones.

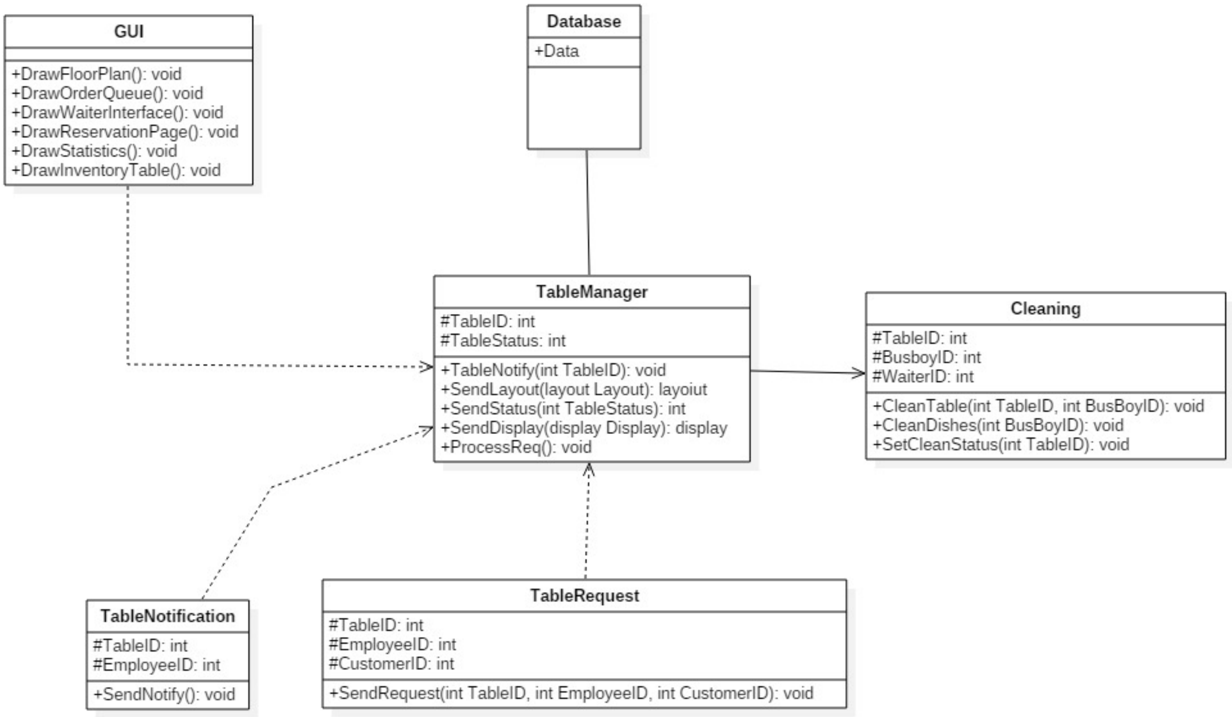
Class Diagrams and Interface Specification

Ordering Subsystem


Database
+Data
GUI
+DrawFloorPlan(): void
+DrawOrderQueue(): void
+DrawWaiterInterface(): void
+DrawReservationPage(): void
+DrawStatistics(): void
+DrawInventoryTable(): void
Order
#mOrderID: int
#mPriority: double
#mOrderStatus: boolean
#mFoodItems: FoodItem
+getOrderID(): int
+getPriority(): double
+getOrderStatus(): boolean
+getFoodItems(): FoodItem
+setOrderID(int OrderID)
+setPriority(double Priority)
+setOrderStatus(boolean OrderStatus)
+setFoodItems(FoodItem FoodItem)
FoodItem
#mItemID: int
#mMealType: int
#mPrice: double
+getItemID(): int
+getMealType(): int
+getPrice(): int
+setItemID(int itemID)
+setMealType(int mealType)
+setPrice(int price)
WaiterActions
#OrderID: int
#CustomerID: int
#TableID: int
#OrderTime: time
+PlaceOrder(int TableID, int OrderTime): void
+PollFoodOrder(int OrderID): void
OrderQueue
#Orders: Order
+TrackRefresher(int OrderID): void
+PriorityCalc(): void
+OrderReady(boolen OrderStatus): boolean
+GetRecipe(recipe Recipe): recipe
CheckSplitter
#TableInfo: info
+SplitChecks(int CustomerId, int TableID, info TableInfo): void
+GenerateBill(int CustomerID, int TableID, info TableInfo): void

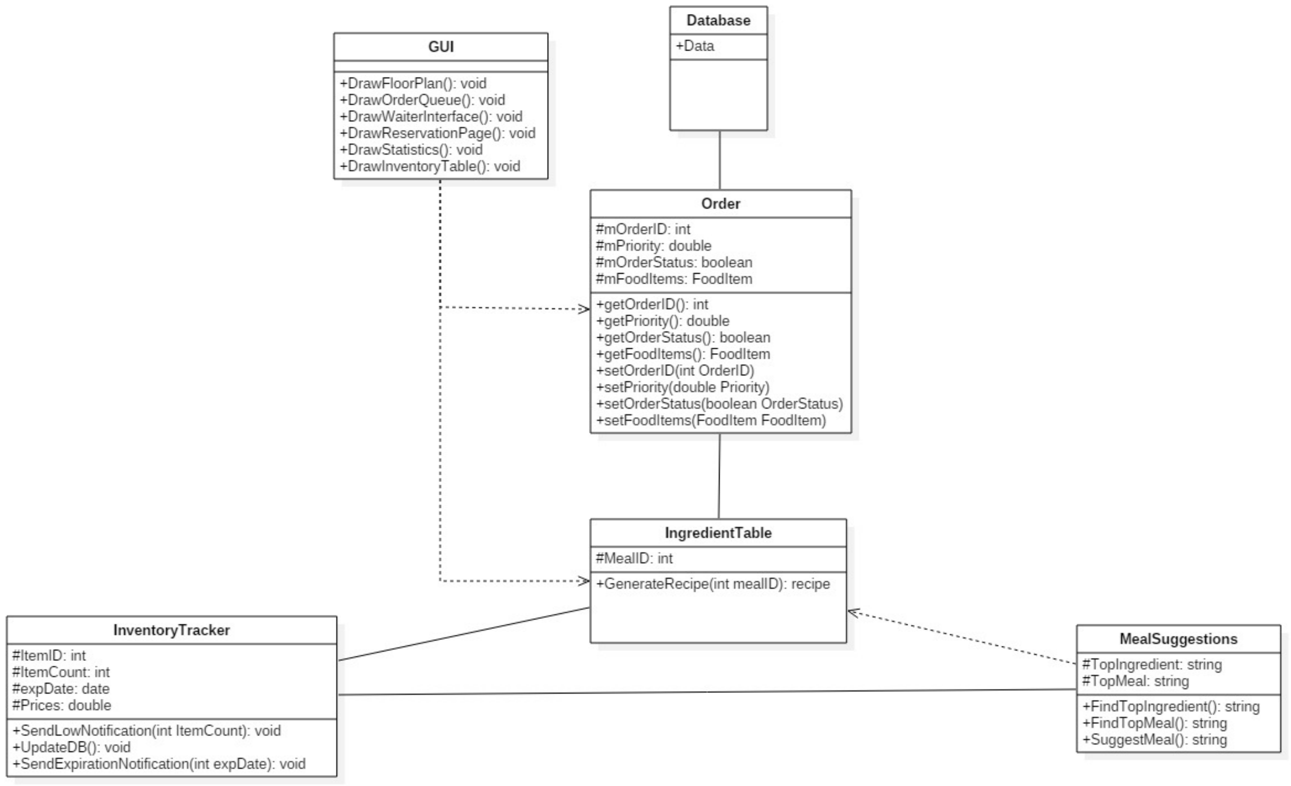
The diagram above describes the Ordering subsystem. This subsystem manages the Order queue, placing the order, paying the bill and updating the user interface with the changes. The classes are described more in detail below.

Table Request & Status Subsystem


GUI
+DrawFloorPlan(): void
+DrawOrderQueue(): void
+DrawWaiterInterface(): void
+DrawReservationPage(): void
+DrawStatistics(): void
+DrawInventoryTable(): void
Database
+Data
TableManager
#TableID: int
#TableStatus: int
+TableNotify(int TableID): void
+SendLayout(layout Layout): layoiut
+SendStatus(int TableStatus): int
+SendDisplay(display Display): display
+ProcessReq(): void
Cleaning
#TableID: int
#BusboyID: int
#WaiterID: int
+CleanTable(int TableID, int BusBoyID): void
+CleanDishes(int BusBoyID): void
+SetCleanStatus(int TableID): void
TableNotification
#TableID: int
#EmployeeID: int
+SendNotify(): void
TableRequest
#TableID: int
#EmployeeID: int
#CustomerID: int
+SendRequest(int TableID, int EmployeeID, int CustomerID): void

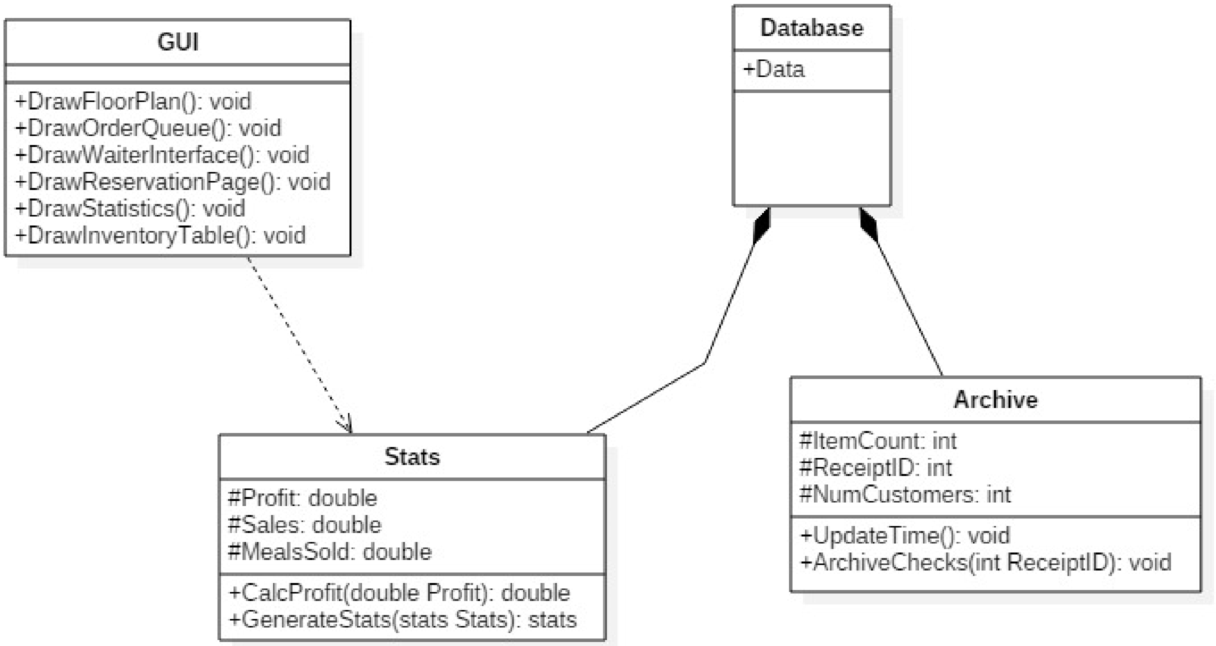
The diagram above shows the Table Request and Status subsystem. It takes care of seating customers that walk into the restaurant or reserve online. The classes are described more in detail below.

Suggestion Subsystem



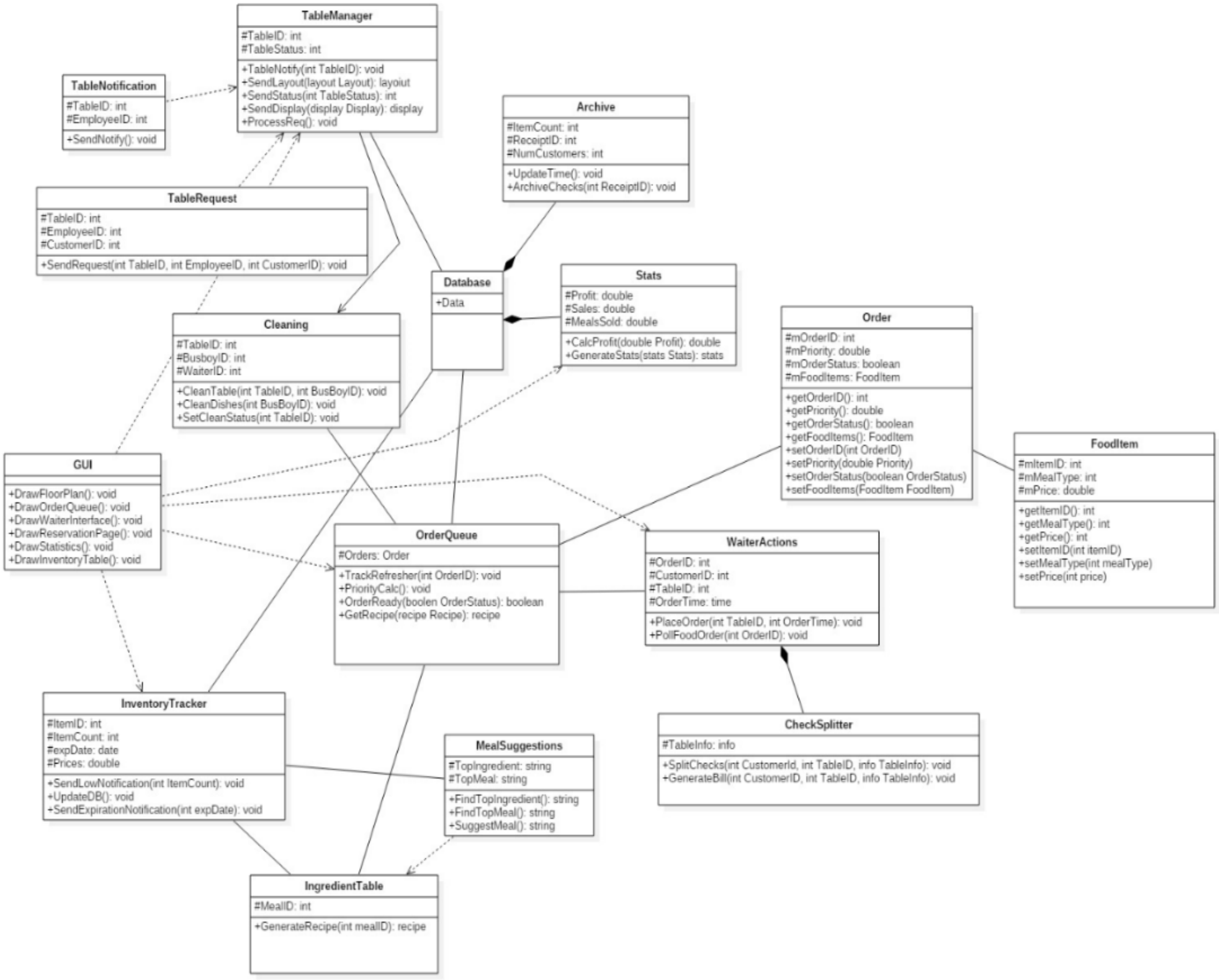
The suggestion subsystem contains classes that suggest menu items in order to maximize profit for the restaurant. It also sends notifications for low stock on items so the restaurant doesn't lose out on sales. The classes are described more in detail below.

Archive Subsystem



The archiving subsystem uses the database and previous transactions to calculate statistics and archive data. The classes, operations and attributes are described below. The database serves as a centralized point for various subsystems to store data.

Class Diagram Full


TableManager
#TableID: int
#TableStatus: int
+TableNotify(int TableID): void
+SendLayout(layout Layout): layoiut
+SendStatus(int TableStatus): int
+SendDisplay(display Display): display
+ProcessReq(): void
TableNotification
#TableID: int
#EmployeeID: int
+SendNotify(): void
Archive
#ItemCount: int
#ReceiptID: int
#NumCustomers: int
+UpdateTime(): void
+ArchiveChecks(int ReceiptID): void
TableRequest
#TableID: int
#EmployeeID: int
#CustomerID: int
+SendRequest(int TableID, int EmployeeID, int CustomerID): void
Database
+Data
Stats
#Profit: double
#Sales: double
#MealsSold: double
+CalcProfit(double Profit): double
+GenerateStats(stats Stats): stats
Cleaning
#TableID: int
#BusboyID: int
#WaiterID: int
+CleanTable(int TableID, int BusBoyID): void
+CleanDishes(int BusBoyID): void
+SetCleanStatus(int TableID): void
Order
#mOrderID: int
#mPriority: double
#mOrderStatus: boolean
#mFoodItems: FoodItem
+getOrderID(): int
+getPriority(): double
+getOrderStatus(): boolean
+getFoodItems(): FoodItem
+setOrderID(int OrderID)
+setPriority(double Priority)
+setOrderStatus(boolean OrderStatus)
+setFoodItems(FoodItem FoodItem)
FoodItem
#mItemID: int
#mMealType: int
#mPrice: double
+getItemID(): int
+getMealType(): int
+getPrice(): int
+setItemID(int itemID)
+setMealType(int mealType)
+setPrice(int price)
GUI
+DrawFloorPlan(): void
+DrawOrderQueue(): void
+DrawWaiterInterface(): void
+DrawReservationPage(): void
+DrawStatistics(): void
+DrawInventoryTable(): void
OrderQueue
#Orders: Order
+TrackRefresher(int OrderID): void
+PriorityCalc(): void
+OrderReady(boolen OrderStatus): boolean
+GetRecipe(recipe Recipe): recipe
WaiterActions
#OrderID: int
#CustomerID: int
#TableID: int
#OrderTime: time
+PlaceOrder(int TableID, int OrderTime): void
+PollFoodOrder(int OrderID): void
InventoryTracker
#ItemID: int
#ItemCount: int
#expDate: date
#Prices: double
+SendLowNotification(int ItemCount): void
+UpdateDB(): void
+SendExpirationNotification(int expDate): void
MealSuggestions
#TopIngredient: string
#TopMeal: string
+FindTopIngredient(): string
+FindTopMeal(): string
+SuggestMeal(): string
CheckSplitter
#TableInfo: info
+SplitChecks(int CustomerId, int TableID, info TableInfo): void
+GenerateBill(int CustomerID, int TableID, info TableInfo): void
IngredientTable
#MealID: int
+GenerateRecipe(int mealID): recipe

This is the full class diagram which combines the broken down subsystems in order to make it easily readable. As you can see, each subsystem interacts with each other in order to fully automate the system. If systems are disconnected in a sense, then the restaurant isn't automated to its fullest potential because a *middleman* will be required to do *heavy work* that can be replaced by the system.

Class Diagram Description

The class diagrams above are broken down into sub-systems to allow maximum readability. Below are brief descriptions for each of the classes which are followed by their attributes and operations.

TableManager is a class that contains TableID, TableStatus and uses those to notify customers of their table opening, sending the layout of the restaurant to the interface, it also processes reservation requests.

TableNotification is a class that sends the notifications to clean table, serve table, etc. to the respective employees.

TableRequest is a class that sends a request to the respective employees to complete a task for the respective tables.

Cleaning is a class that pertains to the busboy that will notify him/her when to do a certain task.

OrderQueue is a class that manages the orders and tracks refreshers along with retrieving recipe's from the ingredient table.

WaiterActions is a class that pertains to the waiter that is in charge of customer orders.

CheckSplitter is a class that automatically

IngredientTable is a class that generates the recipe for the order queue when it's requested by the order queue.

InventoryTracker is a class that keeps track of the inventory and sends notifications when items are below a certain number or if they are about to expire.

MealSuggestions is a class that will find both top ingredients and top meals to suggest meals popular and profitable meals to the manager.

Stats is a class that will calculate statistics such as profit and meals sold using information from the database

Archive is a class that consistently logs receipts along with the system time.

Database is a class that holds all the data in the system and interacts with the rest of the classes.

GUI is a class that depends on multiple classes to draw the user interface of the system.

Order is a class that specifies all the attributes of an order placed by a customer and holds all food items of that particular order.

FoodItem is an individual menu item that it a part of a customer's order.

Data Types and Operation Signatures

TableManager

Attribute	**Description**
#TableID: int	Holds the ID of the respective table
#TableStatus: int	Indicates the status of a table using numbers such as free table is 1, busy is 0

Operations	**Description**
+TableNotify(int TableID):int	Sends a notification when a table is ready for customers
+SendLayout(layout Layout): layout	Sends the table layout and any clustered groups of tables that shows the status of each table upon request
+SendStatus(int TableStatus): int	Sends the status of the table so it can be updated on the interface
+SendDisplay(display Display): display	Updates the display based on any updates to the system
+ProcessReq(): void	Process incoming reservation requests along with walk in requests

TableNotification

Attributes	**Descriptions**
#TableID: int	Holds the ID of the respective table
#EmployeeID: int	Holds the ID of the respective employee

Operations	**Descriptions**
+SendNotify(): void	Sends notifications to the respective employees.

Cleaning

Attributes	**Descriptions**
#TableID : int	Holds the ID of the respective table
#BusboyID: int	Holds the Busboy's employee ID
#WaiterID: int	Holds the Waiter’s employee ID

Operations	**Descriptions**
+CleanTable(int TableID, int BusboyID): void	Sends Busboy a notification to clean the table using the TableID
+CleanDishes(int BusboyID): void	Sends Busboy a notification to clean the dishes in the kitchen
+SetCleanStatus(int TableID): void	Busboy is able to set the status of the table to clean

TableRequest

Attributes	Descriptions
#TableID : int	Holds the ID of the respective table
#EmployeeID: int	Holds the ID of the respective employee
#CustomerID: int	Holds each customer's ID

Operations	Descriptions
+SendRequest(int TableID,int EmployeeID,int CustomerID): void	Sends a request to the system to fulfill a reservation or walk in request

InventoryTracker

Attributes	Descriptions
#ItemID: int	The ID for an item in the inventory
#ItemCount: int	The number of items corresponding to each ItemID
#expDate: date	The expiration date for the respective ItemID
#Prices: double	The price of each item for the respective ItemID

Operations	Descriptions
+SendLowNotification(int ItemCount): void	Sends notification to the manager if the item count falls below a predefined threshold
+UpdateDB(): void	Updates database with new item count every time anything from the inventory is used
+SendExpirationNotification(int expDate): void	Sends notification to the manager if the item gets closer to the expiration date

Database

Attributes	Descriptions
+data : *	Holds all the data used for the system.

* can be any type.

OrderQueue

Attributes	Descriptions
#OrderID: int	Holds the ID of the order that has been placed
#Priority: int	Holds the priority of the order
#OrderStatus: boolean	Holds the status of the order, ready or not ready
#MealType: meal	Holds the type of meal each table is on, such as appetizers, entrees etc.

Operations	Descriptions
+TrackRefresher(int OrderID): void	Tracks the refreshers for the given OrderID
+PriorityCalc(): void	Calculates the priority of the order based on factors such as time, size etc.
+OrderReady(boolean OrderStatus): boolean	Marks order as ready when it is done
+GetRecipe(recipe Recipe): recipe	Contacts the recipe book and retrieves the recipe for the requested meal(s)

IngredientTable

Attributes	**Descriptions**
#MealID: int	Holds the ID of the meal

Operations	**Descriptions**
+GenerateRecipe(int mealID): recipe	Generates the recipe for the requested mealID

Archive

Attributes	**Descriptions**
#ItemCount:int	The number of items corresponding to each ItemID
#ReceiptID: int	Holds the ID for each receipt generated at the restaurant
#NumCustomers: int	The total number of customer in a requested time period

Operations	**Descriptions**
+UpdateTime(): void	Gets the system time to display when the application was last updated
+ArchiveChecks(int ReceiptID): void	Saves the checks for future archiving use along with using the data for statistics

Stats

Attributes	Descriptions
#Profit: double	The total profit made in a requested time period
#Sales: double	The total money from sales, not including costs, over a requested time period
#MealsSold: double	The total number of meals sold over a requested time period

Operations	Descriptions
+CalcProfit(double Profit): double	Calculates the profit by using the data from the system.
+GenerateStats(stats Stats): stats	Generate several statistics summarizing the restaurant's performance

WaiterActions

Attributes	Descriptions
#OrderID: int	Holds the ID of the order that has been placed
#CustomerID: int	Holds each customer's ID
#TableID: int	Holds the ID of the respective table
#OrderTime: time	The time each order was placed

Operations	Descriptions
+PlaceOrder(int TableID,int OrderTime): void	Sends the order to the order queue along with the TableID and the time of the order
+PollFoodOrder(int OrderID): void	Sends a request to order queue to send a notification back if the food is ready

CheckSplitter

Attributes	Descriptions
#TableInfo: info	Holds the information about the table such as the size of the table and the number of people paying

Operations	Descriptions
+SplitChecks(int CustomerId, int TableID,info TableInfo): void	Splits the checks evenly between the number of people paying or by each respective order, given the TableInfo
+GenerateBill(int CustomerID,int TableID,info TableInfo): void	Generated the bill(s) based on the TableInfo, CustomerID and the TableID

MealSuggestions

Attributes	Descriptions
#TopIngredient: string	Holds the ID of the most used ingredient in the restaurant
#TopMeal: string	Holds the ID of the most ordered meal in the restaurant

Operations	Descriptions
+FindTopIngredient(): string	Looks in the database to find the top ingredient
+FindTopMeal(): string	Looks in the database to find the top meal
+SuggestMeal(): string	Uses the top ingredient and top meal along with other statistics to suggest more profitable meals

GUI

Operations	**Descriptions**
+DrawFloorPlan(): void	Uses information from TableManager and uses it to draw the floor plan of the restaurant
+DrawOrderQueue(): void	Uses information from the OrderQueue and uses it to draw the interface of the orders for the chefs in the kitchen
+DrawWaiterInterface(): void	Uses information from WaiterActions and draws the interface on the waiter’s tablet
+DrawReservationPage(): void	Draws the reservation page depending on the information in TableRequest
+DrawStatistics(): void	Draws the statistics page and displays the information from the Stats class
+DrawInventoryTable(): void	Draws the inventory table using information from the InventoryTracker

Order

Attributes	**Descriptions**
#mOrderID: int	Holds the ID of a particular Order
#mPriority: double	Holds the priority value of an Order
#mOrderStatus: boolean	Holds the order status of an Order
#mFoodItems	A list that holds all the food items within an Order

Operation	**Descriptions**
#getOrderID(): int	Gets the OrderID of an Order
+getPriority(): double	Gets the priority of an Order

+getOrderStatus(): boolean	Gets the order status of an Order
+getFoodItems(): FoodItem	Gets the list of fooditems in an Order
+setOrderID(int OrderID)	Sets the OrderID of an Order
+setPriority(double Priority)	Sets the priority of an Order
+setOrderStatus(boolean OrderStatus)	Sets the Order Status of an Order
+setFoodItems(FoodItem FoodItem)	Adds food items to the list of food items

FoodItem

Attributes	**Descriptions**
#mItemID: int	Holds the ID of each respective food item
#mMealType: int	Holds the meal type of each food item i.e Appetizers, Entrees, and Dessert
#mPrice: double	Holds the price of each food item

Operations	**Descriptions**
+getItemID(): int	Returnsthe Item ID when of an item
+getMealType(): int	Returns the meal type of a meal
+getPrice(): int	Returns price of a meal
+setItemID(int itemID)	Sets the Item ID on a item
+setMealType(int mealType)	Sets the Meal type on a meal
+setPrice(int price)	Sets the price on a meal

Traceability Matrix

	TableManager	TableNotification	Archive	TableRequest	Cleaning	Stats	GUI	OrderQueue	WaiterActions	InventoryTracker	MealSuggestions	IngredientTable	CheckSplitter	Order	FoodItem
ReservationTableManager	X														
TableLoadStatus	X														
TableClusterManager	X														
TableBuddyManager	X														
TableStatusManager	X														
TableStatusNotifier		X													
FloorPlanPage							X								
ReservationPage							X								
CleanRequest				X	X				X						
WalkinRequest				X											
SetCleanedRequest				X	X										
CleanNotify		X			X										
ReadyNotify		X							X						
Logger			X												
SystemStats						X									
Archive			X												
UpdateTime			X			X									
CheckSplitter									X				X		
PlaceOrder								X						x	
OrderQueue								X						x	
StageManager								X						x	
PriorityCalculator								X						x	
FoodChecker								X	X						
MealStageTracker							X	X	X						
FoodReady								X							
FoodPollRequest								X	X						
OrderReady								X	X					x	
InventoryContainer										X					
InventoryTracker										X					
IngredientTable												X			x
IngredientPopularity															
MealPopularity											X				
MealSuggester											X				
MealRequest										X					x

The traceability matrix describes how each the domain concepts map to the corresponding class. The explanations for the mappings are:

TableManager : The table manager is responsible for deciding what interval "Reservation" tables can be reserved in, from collected data. This is the job of the ReservationTableManager domain concept. The TableManager is also responsible for collecting data regarding the number of reservations and walkins and deciding the ratios of "Reservation" to "Walkin" tables per cluster. Therefore, the TableLoadStatus domain concept maps to the TableManager. The table Manager is also responsible for managing all the table “Clusters” per size; this is the domain concept TableClusterManager. It is also responsible for combining tables of different sizes to form larger tables when needed; this is the concept of TableBuddyManager. Lastily, it is responsible for managing all the statuses of the tables: reserved, dirty, available and in-use. This is concept of the TableStatusNotfier. We realized that all of the domain concepts that dealt with managing table weren’t distinct enough to be considered separate classes; they interconnected a lot and communicate with one-another. Thus we combined them into one class, TableManager.

TableNotification : TableNotification is responsible for managing all notifications for tables, whether a table needs to be cleaned or a waiter needs to go to that table. Thus, TableNotification takes on the concept TableNotifer. The two notifications it sends, as described above, are CleanNotify and ReadyNotify. None of these concepts are different enough that they justify splitting them into classes. The Notifications are simply messages.

Archive: We realized the Archive and Logger domain concepts were very similar, if not almost the same thing. Thus it is justified making them one class. The UpdateTime domain concept is utilized heavily by the Archive, and thus is mapped to it.

TableRequest: All of the table requests have been mapped to the TableRequest. We realized generalizing them made sense and the requests could simple be functions. They weren’t complex enough to justify them as being classes.

Cleaning: The cleaning related operations were distinct enough from everything else to justify there own class. Thus the clean related requests and notifications are mapped to another class too, Cleaning.

Stats: Statistics is a very important concept for the restaurant. The Stat class is mainly built of the SystemStatistics concept. In addition, the stats directly depend upon the time for timestamps; thus it uses UpdateTime.

GUI: The three main interfaces in the system are the FloorPlanPage, Reservation page for customers, and the WaiterActions interface for making table requests. Thus the system GUI is made up of these three.

OrderQueue: The order queue is made up lots of components that are relatively small and thus don't need to be their own classes. It manages the order in which orders are made, their priority. It also maintains state for when food and orders are ready to be picked up by the waiter. Thus it is directly connected to FoodPollRequest and OrderReady.

WaiterActions: The waiter's action are very related. The waiter asks for a table to be cleaned and is notified when one is ready and is being taken by customers. In addition, the waiter utilizes the food poll request system and its notifications to keep track of when his/her orders are ready.

InventoryTracker: The inventory tracker a concept with the same name as the class mapped to it, and also utilizes the InventoryContainer and takes in MealRequests

MealSuggestions: The meal suggestions system utilizes the different popularities to come up with suggestions and thus has those concepts mapped to it to form a class.

IngredientTable: The ingredient table mapped directly from its concept.

CheckSplitter: The check splitter mapped directly from its concept.

Order: The Order is made up from many components relating to the food items. It takes in all the information regarding each meal that the customer has requested, and the waiter inputted. It gathers this information from FoodItem, sets the priority and sends it to the OrderQueue to manage it.

FoodItem: FoodItem retrieves and sets information relating to each meal. It is associated with Order to provide the meal type.

System Architecture and System Design

Architectural Styles

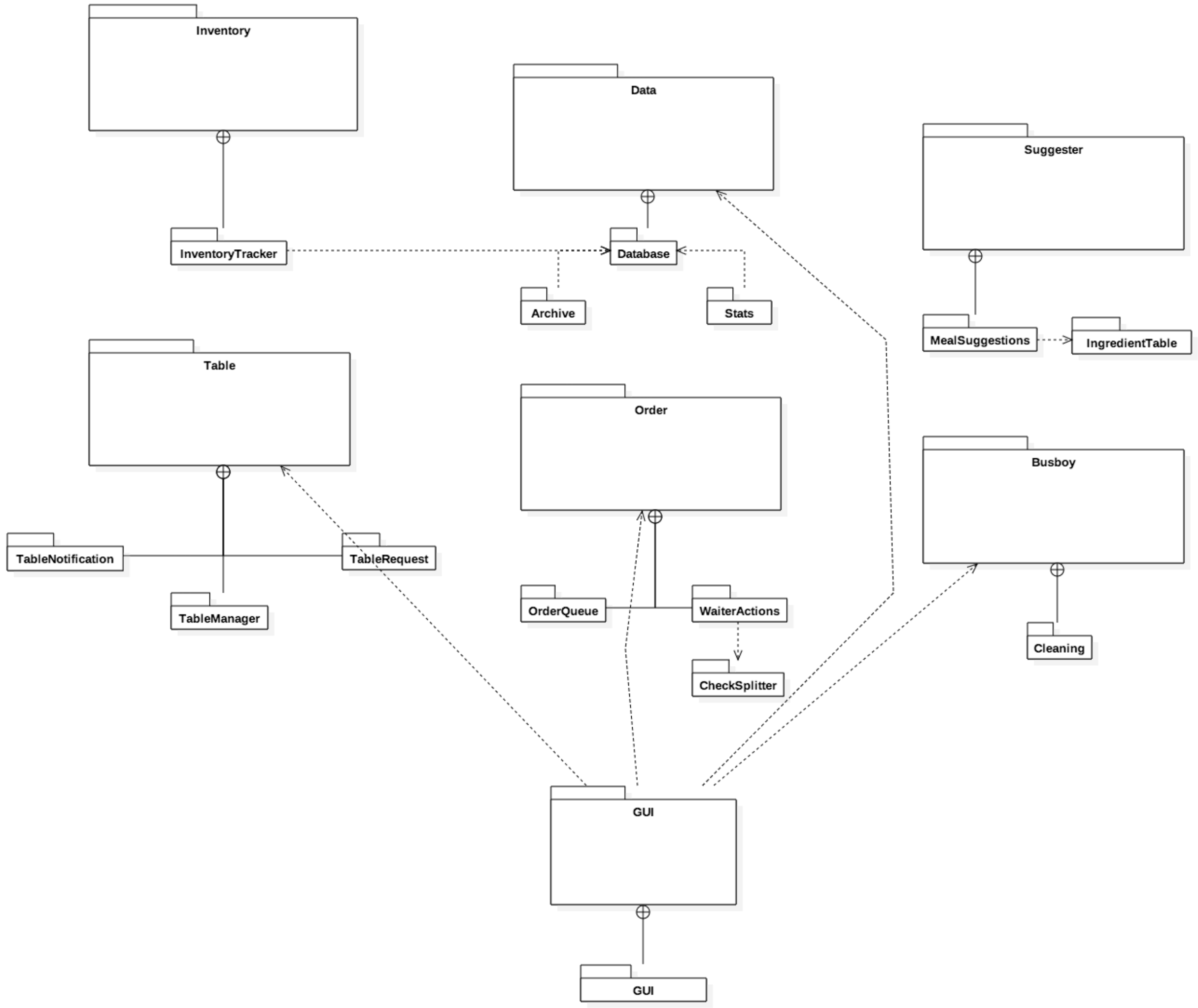
Our project follows a three-tier model which is composed of a presentation tier, a logic tier, and a data tier. Each tier is divided into smaller components based on the user interaction.

The presentation tier represents the various user interfaces depending on the user interacting with the system. The user interfaces are divided into three categories; customer, employee, and manager. The customer will be limited to viewing the reservation system, while the employee will only have access to the system specific to his or her role. In addition, the manager will have access to all features of the system, which can be configured by the application.

The logic tier deals with all of the computational aspects of our system. For instance, there will be various components that handle the floor plan to manage tables, inventory system, billing calculations and statistics, etc. The logic tier is also responsible for user authentication to allocate the appropriate permissions for a user. In an overall aspect, the logic tier handles all of the processing details of the entire system.

The data tier will represents the storage system of our application and will hold all pertinent data needed for different computational functions of the logic tier. Some types of data we will store and collect includes order history, ingredient usage, billing statistics, etc.

Identifying Subsystems



Each Package Description

1. **Inventory:** This package will keep track of the restaurant's inventory. Based on the usage of the raw materials, the Inventory package will give suggestions on when to order new inventory given the “low threshold” set by the restaurant manager. InventoryTracker class is included in this package since this is the only class the package needs to keep track of the inventory. Using the tracked data, this class makes the suggestions.

2. **Data:** This package contains our Database and a set of tools that are needed to manipulate and use this data. Our class diagram indicates that at one point other classes will interact with the database class to get some data to make their appropriate decisions. For example, the MealSuggestions class retrieves data indirectly from the database class to carry out its functional requirements. The Data package only contains classes that are

related to the storage and retrieval of data. Other classes are not required in this package because they are more related to the logical functions of the system.

3. **Suggester:** The suggester package contains the MealSuggestions and the IngredientTable classes. It carries out the responsibility of suggesting popular meals based on the data that is gathered by the system. These suggestions suggest the most profitable meals within each popularity level. Here popularity level refers the duration of the time for which the meal has been popular and is discussed in the mathematical models section. The reason for having these two classes within this package is the fact that they are essential for the package to carry out its minimum responsibilities.

4. **Table:** The table package is responsible for the overall management of the tables within the restaurant. These include managing both the walk-in and reserved tables. For instance, the tables are grouped and assigned by the TableManager class. It is also responsible for keeping track of table statuses, such as available, dirty, clean, etc. Additionally, the TableRequest class allows customers to request the assistance of waiters. The Table package includes these classes because these classes are all attributed to table operations.

5. **Order:** This package includes the following classes: WaiterActions, OrderQueue, and CheckSplitter. The WaiterActions class includes the responsibilities of the waiter such as placing an order. The orders placed is handled by the OrderQueue class, which prioritizes orders using a specialized algorithm to ensure efficient delivery of orders. Lastly, the CheckSplitter class is utilized by the WaiterActions class to split a customer's bill as specified. The Order package contains these classes because the classes are interleaved within each other, requiring each other to perform the actions required for what describes an order.

6. **Busboy:** The Busboy package contains the cleaning class which is responsible for notifying/allowing the busboy to manage the cleaning. This class is included in this package as it requires this class to carry out its responsibilities.

7. **GUI:** The GUI package consists of the GUI class which is responsible for displaying a set of controls that allow the user to interact with the system. The reason for having this class within this package is due to the fact that this package requires this class to perform the basic responsibilities that are represented by this package.

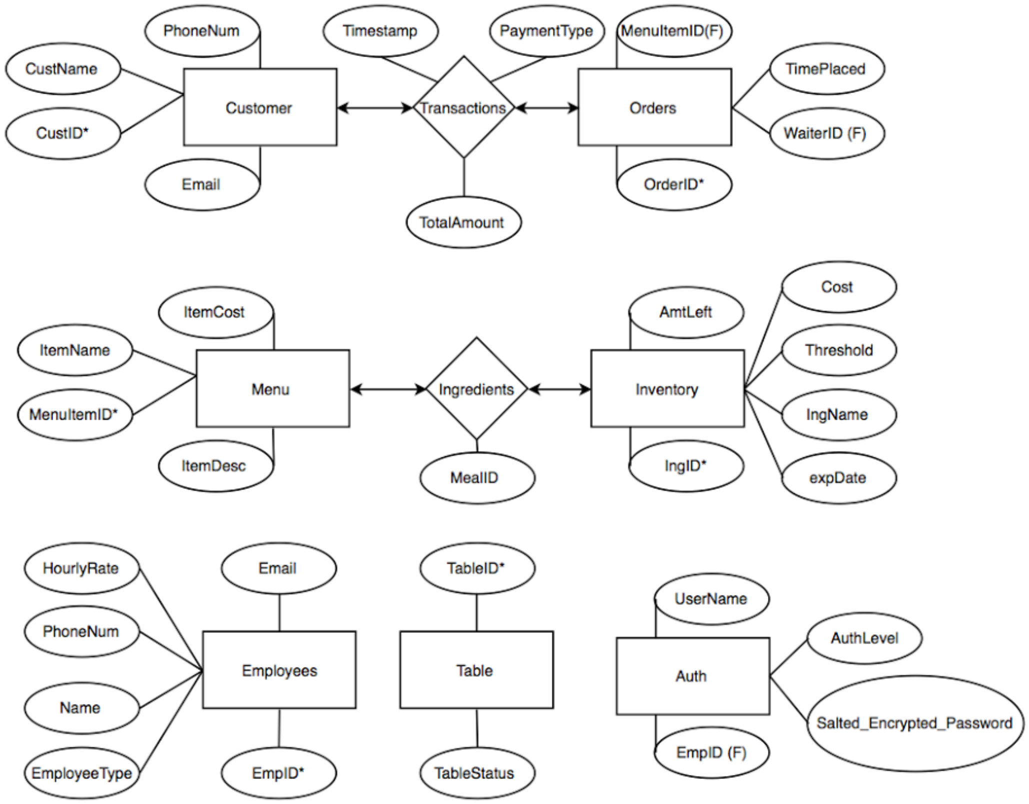
Mapping Subsystems to Hardware

Subsystems	Server	Tablet/Phone/PC
Inventory	X	
Data	X	
Suggester	X	
Table	X	X
Busboy	X	X
Order	X	X
GUI		X

The subsystems above have been allotted to either Server or Tablet/Phone/PC. We have made the hardware requirements very flexible because we want to increase the usability and mobility of our software.

Persistent Data Storage

Since our system is data driven, we are required to have persistent data storage. We intend to utilize a central database (MongoDB) for all subsystems. We decided on using a NoSQL database after designing the ER diagram. We, however, still use the diagram for reference.
The following is the ER diagram.



Network Protocol

Our application uses the widely used HTTP protocol because the application is web-based. Our user interface is accessed via an HTTP request that will display the application. The logic functions a will also be accessed via HTTP requests made by the application. All remaining forms of communication are related to the database, which are hidden away by the framework we are using. Although the exact database communication protocol is unknown, it is known that sockets are used.

Global Control Flow

Execution order: Our server is an event-driven system because a customer may place reservation request at a random type or a waiter could place on order at any time. The server responds to each request by appropriately selecting the next action to process the event. Also, an employee could be processing multiple actions at once. For example, a waiter may check the order for table 1 and take the order (submit) for table 2. So, all of the users are not restricted to a linear sequence of events.

Time dependency: Our system is both real-time and event response type. For example, all orders tracked in real-time so that customers receive their food as quick as possible. Every event between taking order and delivering it is however event response type. Each event's completion results in the start of the next event. For instance, when a customer is seated, a waiter is sent to take the order.

Concurrency: We will not directly need concurrency because each subsystem operates as a separate process. However, the underlying libraries could possibly use multiple threads (ex. server). This is not our concern because they are abstracted and a black box in our perspective.

Hardware Requirements

Tablet/Phone/Computer (Client-side)

- Modern web browser with javascript support (i.e. Google Chrome, Mozilla Firefox, Safari).
- Wireless Network Card
- High Resolution Display

Server

- 8GB RAM
- Quad-core Processor
- Network Card
- Modern Operating System that supports NodeJS (i.e Windows Server, Windows 7+, and macOS, Mac OS X, Linux distributions, etc.)

As mentioned in the mapping each subsystem section, one of the main goals of our software is to promote usability across different systems. Therefore, the minimum requirements of our hardware requires that the client-side users have a device with a modern web browser as stated above. The server-sided system requires an average amount of memory and processing power that is sufficient to keep the system running without any conflict.

Algorithms and Data Structures

Algorithms

Suggesting Meal Plan

The Meal Suggester will determine the popularity of an item based on the popularity of the item as well as the the total profit gained from the sale of that particular item. This is done by assigning levels of popularity to each item based on how long the item has been popular. Within each level the total cost and revenue of the meal item is calculated to determine the profit margin. The cost aspect of this is determined by keeping track of the total ingredients used to make a particular meal. In the end, the system will suggest items within each level of popularity based on their respective profits.

Meal Suggester Attributes

- Popularity Level
  - Level: Low, Medium, High, Exclusive
- Meal Popularity
  - Average Meals/Week - Meals/Day of last 7 days divided by DayCount
  - Average Meals/Month
  - Average Meals/Quarter

Popular Durations	**Popularity Level**
Week, Month, Quarter	**Exclusive**
Week, Month	**Medium**
Month, Quarter	**High**
Week, Quarter	**Low**
Week	**Low**
Nothing	**Low**

```
FOR EACH : POPULARITY LEVEL
{
        FOR EACH : MEAL
        {
                COST = (NUM_SOLD) * (NUM_INGREDIENTS * INGREDIENT_PRICES)
                REVENUE = (NUM_SOLD) * (MEAL_PRICE)
                PROFIT = REVENUE - COST
        }
        SORT(MEAL by PROFIT, DESCENDING)
}
```

Manage Order Queue

Node Priority:

The Order Queue is a priority based system that contains is a queue of nodes. A node groups a set of orders based on the time range they were placed. For instance, orders placed between 9PM and 10PM will belong to a specific node. The time range for each node is not static. Nodes that are created earlier have a higher priority than nodes that are created later.

Stage Priority (Within a node):

Each node consists of three stages; appetizers, entrees, and desserts. When orders are placed, all of the items are divided among the three stages. Each stage has a base priority level with appetizers having the highest priority, entrees with the second highest priority, and desserts with the least priority. If an order only belongs to less than three stages, it will have a higher priority than the base priority value of that stage.

Item Priority (Within a stage):

Within each stage, items are grouped by each table order. The priority of an item will be determined based on the average time it takes to cook the items of a stage for a table order. Individual items within a stage are prioritized in a way such that the average time for each table order remains relatively consistent. Therefore, each table order is given a fair prioritization.

```
NODE_TIME_LENGTH = some value
FOR EACH : ORDER
{
        IF : CURRENT TIME - NODE_TIME_LENGTH >= PREV_NODE_START_TIME
        {
                CURRENT_NODE = NEW NODE(PREV_START_TIME + NODE_TIME_LENGTH)
        }
        NODE = PREV_NODE
        INSERT(ORDER, NODE)
        FOR EACH: ITEM {SET BASE STAGE PRIORITY }

        IF : NUMBER OF STAGES FOR ORDER < 3
        {
                FOR EACH: ITEM { INCREASE PRIORITY OF ALL STAGES }
         }

UPDATE_PRIORITIES:
        FOR EACH : STAGE
        {
                FOR EACH : ORDER
                {
                        DETERMINE TOTAL TIME FOR ORDER
                        DETERMINE AVERAGE TIME/ITEM FOR ORDER

                        ITEM PRIORITY += RATIO OF TOTAL TIME TO AVG TIME/ITEM
                }
        }

OnItemCompleted:
        UPDATE_PRIORITIES
```

Data Structures

Our project will include several data structures to support various components.

- ➔ The OrderQueue class requires a queue that sorts FoodItem objects based on the priority calculated by our algorithm described in the algorithms section of the report.
    - ◆ Performance versus Flexibility: The order queue meets the requirement of our concept. Once all the priority values are calculated the items are linearly sorted where the first item of the queue needs to be processed before any proceeding items. Items in the middle of the queue do not need to be directly accessed, therefore, rendering a data structure such as an array irrelevant. In conclusion, a queue provides a safeguard from having too much flexibility such an array. In terms of performance, a queue versus an array would not have a big performance trade-off if at all.
- ➔ The Order class contains an array of FoodItem objects that represents individual items that a customer orders.
    - ◆ Performance versus Flexibility: An array is a simple data structure that can directly access its contents. An array can be difficult to handle if it contains too much data because sorting operations can take a long time. However, the FoodItems array of our module will not be required to store an immense amount of data in terms of practicality. For instance, a table will most likely never order 1,000,000 items. In terms of flexibility, our requirements for accessing items are not limited by the attributes of an array.
- ➔ The database contains a table that holds information about all of the meals including ingredients, prices, and quantities available.
    - ◆ Performance versus Flexibility: We chose to use a database to store meal information for primarily two reasons. One reason is that there are many attributes to “meals” that are not required to be used in every part of the program, making it inefficient to have unused information floating around in the execution environment. Moreover, by storing the information in a database, we have the flexibility to only extract data we need at a specific point in time, rather than having to work with the entire set of data.
- ➔ The database contains a table that stores archived information about past orders and inventory history.
    - ◆ Performance versus Flexibility: Information about past orders and inventory is stored in the database for a variety of reasons that are relevant to both performance and flexibility. Keeping all previous history in the execution environment is dangerous because it requires an enormous amount of memory,

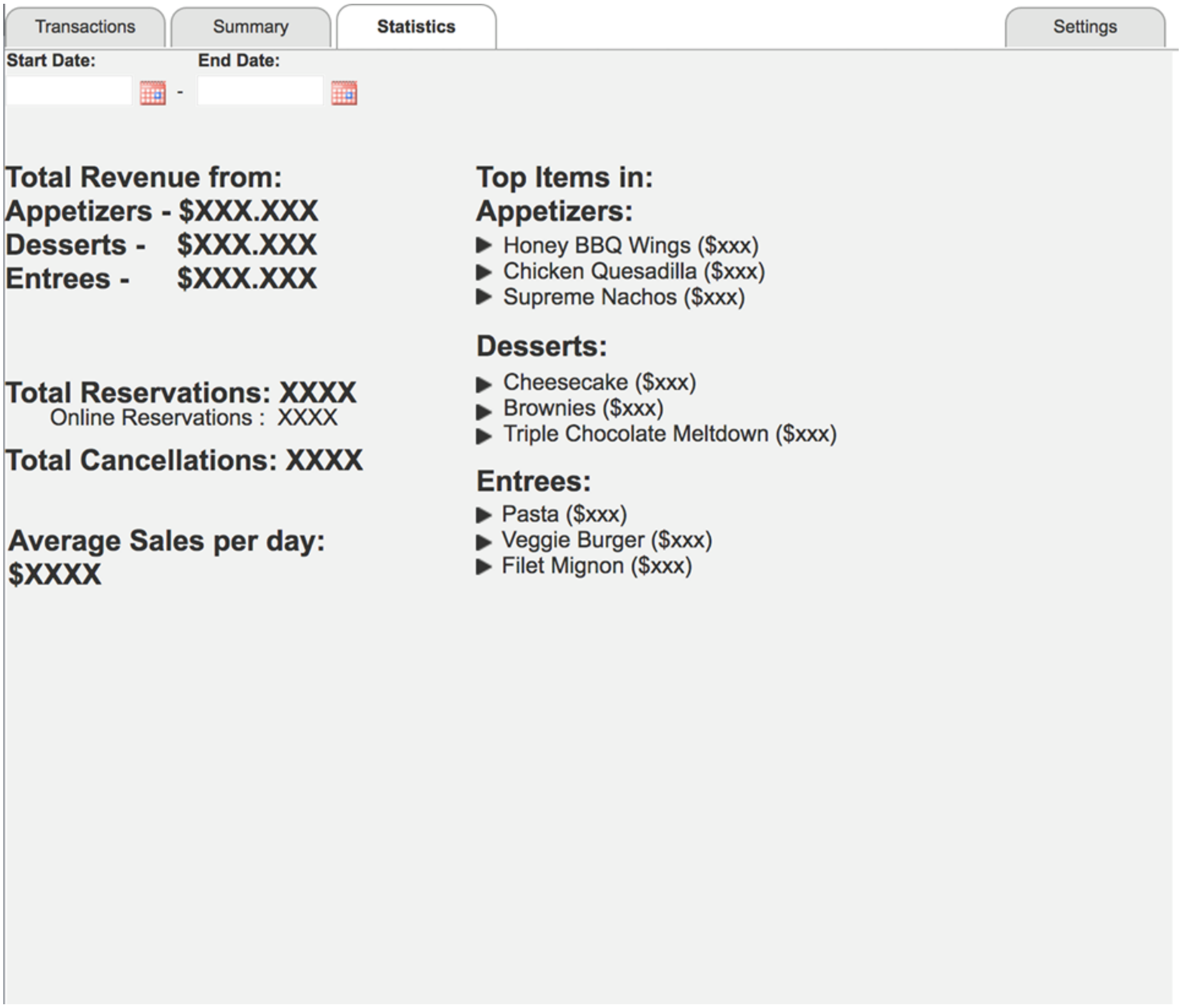
meaning it costs a lot. Also, the archive history is not volatile information, meaning that we need to save the data even when the system is not running. With respect to flexibility, the manager has the ability to view archived information for a specific range of dates/times, which a database allows with less overhead than other data structures.

User Interface and Design Implementation

In Report 1, we created wireframes and mock up designs for our three fully dressed use cases. In addition to being a restaurant automation application, our application also uses the data gathered from the system to automatically generate statistics in order to improve the restaurant. The goal of our interface is to be minimalistic and appealing while allowing ease of access to information. This will be achieved by using modern front-end frameworks such as Bootstrap or Semantic UI which allows for customization and quick templates for building interfaces. This saves us time in designing the stylesheets and instead we can focus on making sure the appealing interface is backed by a robust backend system. We have not yet implemented these interfaces, but plan to do so the following week, so there have not been implemented modifications but below we describe the interface. We also describe the ease of use and design of our planned interface below.

Manage Archive/Statistics

The archive and statistics interface will be used to display information extracted from the data automatically collected by the system. The manager is able to select a time period to view the number of transactions along with a preview of each bill generated. The preview will take up most of the screen space since the manager shouldn’t have to download a file to view it for ease of use. However, for record keeping and offline viewing the manager can print, export, all or individual bills using quick buttons displayed on the screen. In the summary tab, the system will use the collected data and generate a summary for the time period selected. It will be displayed on the screen along with a graph depicting the sales. This allows for the manager to get a quick glance at how the restaurant is performing. Lastly, in the statistics tab, the interface will display more detailed information about the restaurant with several statistics about the restaurant, giving the manager even more information about the performance of the restaurant. Although, this is data analysis, it's still collecting automated data for restaurant and helps the restaurant run more efficiently. Below is a quick mockup of the Archive/Statistics interface, only one of the three tabs is shown here:



Start Date: - End Date:

Total Revenue from:
Appetizers - $XXX.XXX
Desserts - $XXX.XXX
Entrees - $XXX.XXX

Total Reservations: XXXX
Online Reservations : XXXX

Total Cancellations: XXXX

Average Sales per day:
$XXXX

Top Items in:

Appetizers:

- Honey BBQ Wings ($xxx)
- Chicken Quesadilla ($xxx)
- Supreme Nachos ($xxx)

Desserts:

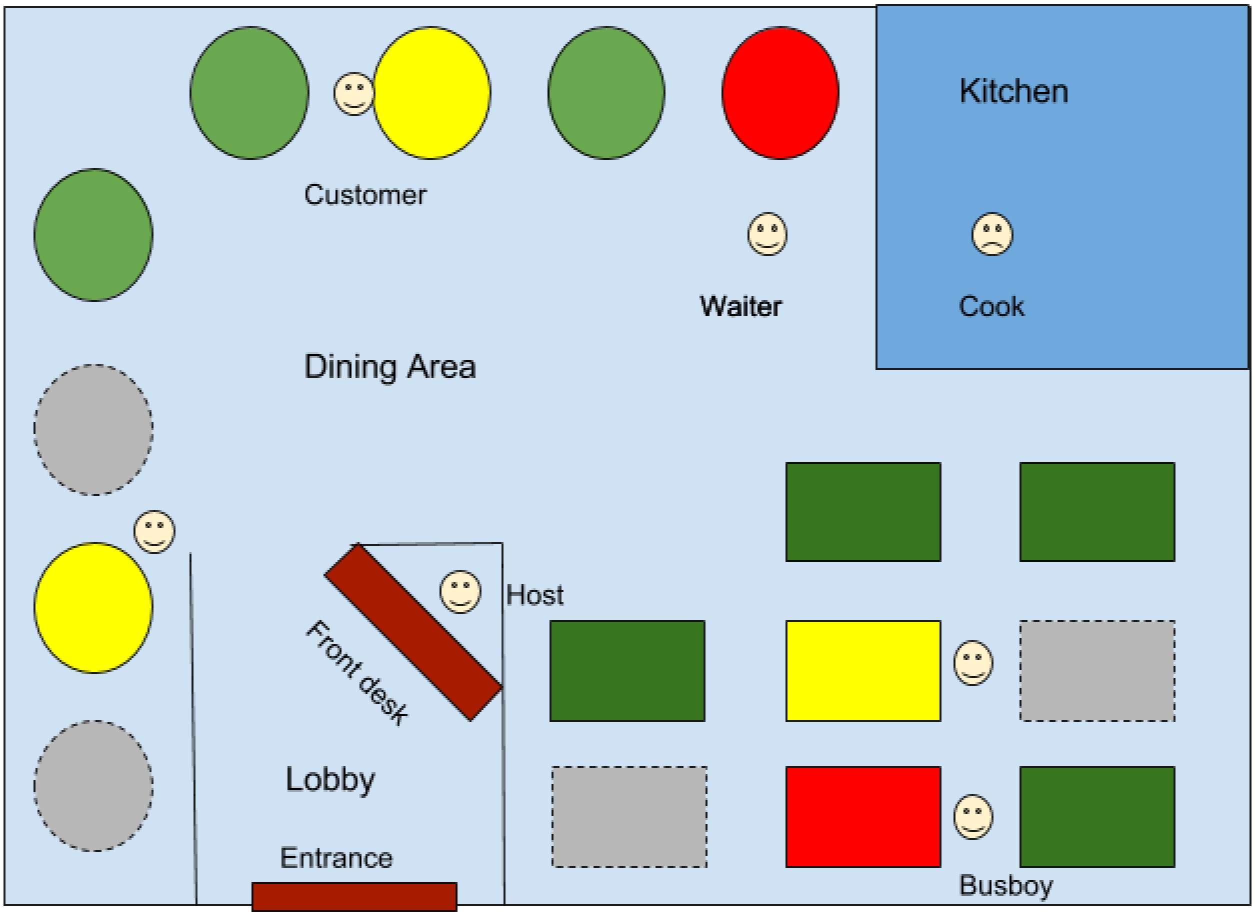
- Cheesecake ($xxx)
- Brownies ($xxx)
- Triple Chocolate Meltdown ($xxx)

Entrees:

- Pasta ($xxx)
- Veggie Burger ($xxx)
- Filet Mignon ($xxx)

Manage Floor Status

The floor status user interface is used to display the status of each table in the restaurant. It displays whether each table is available, occupied, dirty, or reserved. Each table's figure is going to contain information about the customer, so it is easy for the host to take a quick glance and seat the customer(s) at the reserved table. The host can pick a table on the interface that is free and bring up a dialog box that will allow them to enter the number of customers. This will trigger the change in the interface and display the updated information. Below is a quick mockup of the floor plan interface:

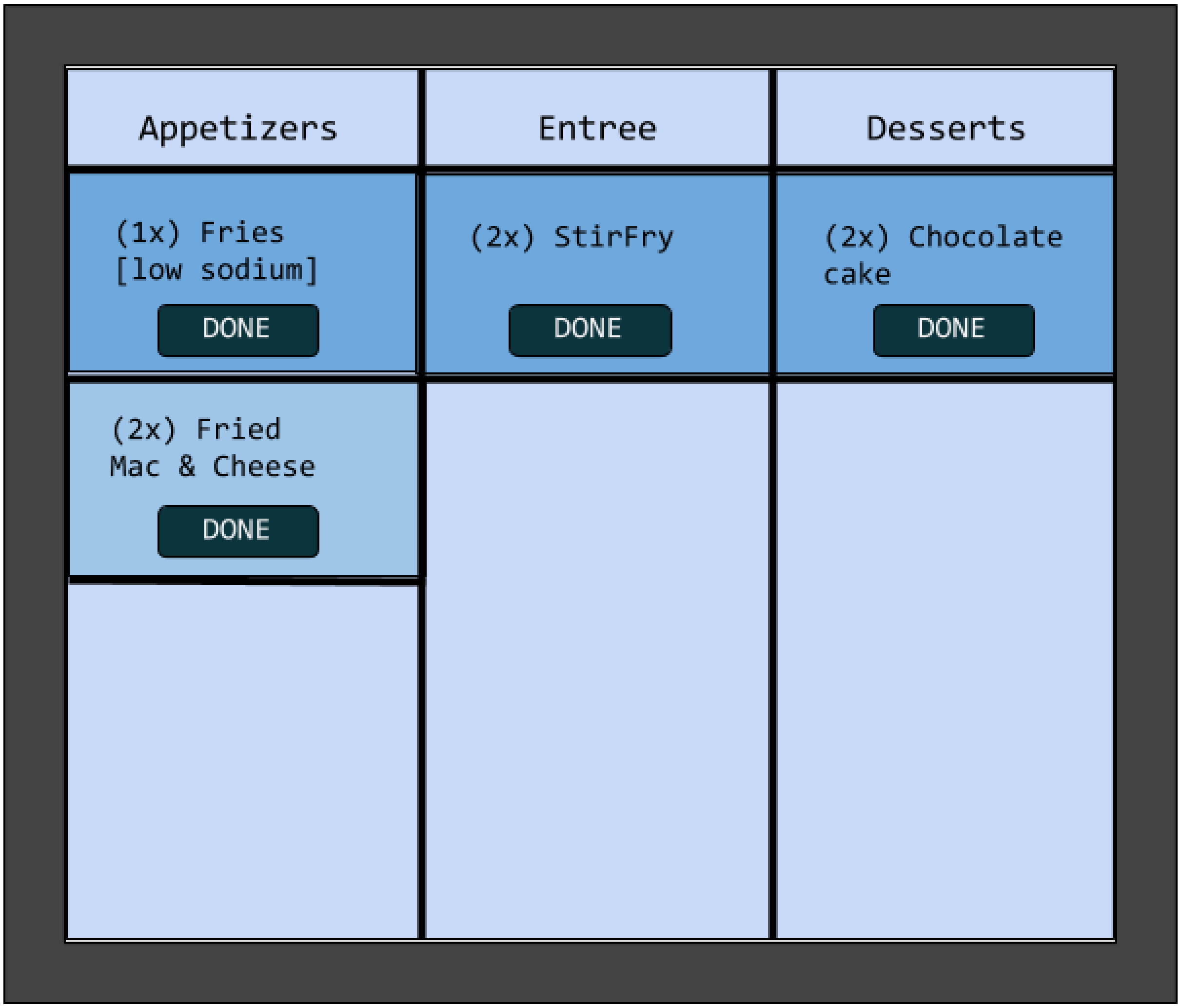


Legend:

- 🟩 Available Table
- 🟨 Occupied Table
- 🟥 Dirty Table
- ⬚ Reserved Table

Manage Order Queue

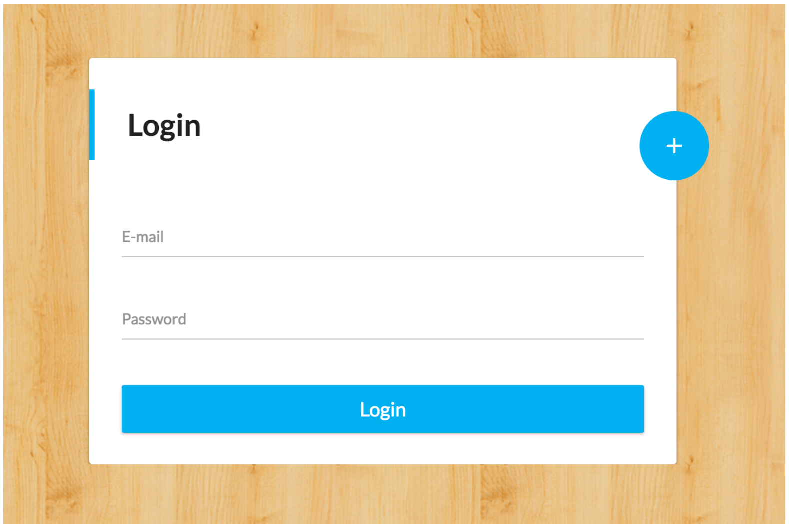
The order queue user interface is used to display the list of orders to the chefs in the kitchen. The topmost orders are the ones that need to be completed first. It is split up by the stage of the meal each table is at, such as appetizers, entrees and desserts. In order to make it an easy to use interface, we only have one button on each item that allows the chef to complete the order with just one click. The button will be big enough so it is easy for the chef to press it on a touch screen or clicking with a mouse without needing much precision. In addition to the orders being displayed, the chef can also drag around the order cards if he sees fit. The interface will also have a small alert icon when the waiter polls to check if the order is done. The order cards themselves will also change color based on the priority number calculated by algorithm so the chef can keep track of how long ago the order came in and if he needs to reorder the queue. If an item is sent back to the kitchen, it is re-added to the queue and placed at the top with highest priority allowing it to be next item the chef(s) will be making.
Below is a quick mockup for the order queue, which will be implemented later:

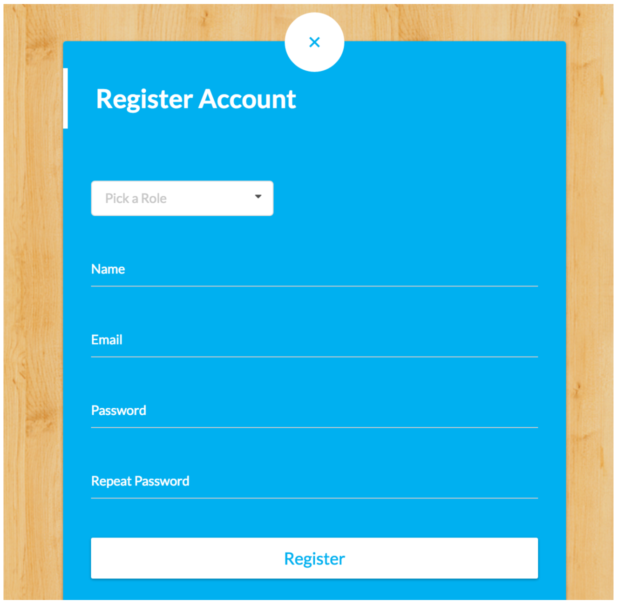


Appetizers	Entree	Desserts
(1x) Fries [low sodium]	(2x) StirFry	(2x) Chocolate cake
(2x) Fried Mac & Cheese		

Login & Register Page

The first interface we implemented is the Login and Register page, this is essential to the start of the project since you need the login page to access most of the project webpages. We used MeteorJS as a NodeJS web framework along with BootStrap and SemanticUI for the front end views for the webpage. Screenshots of the webpage are displayed below and are subject to change.

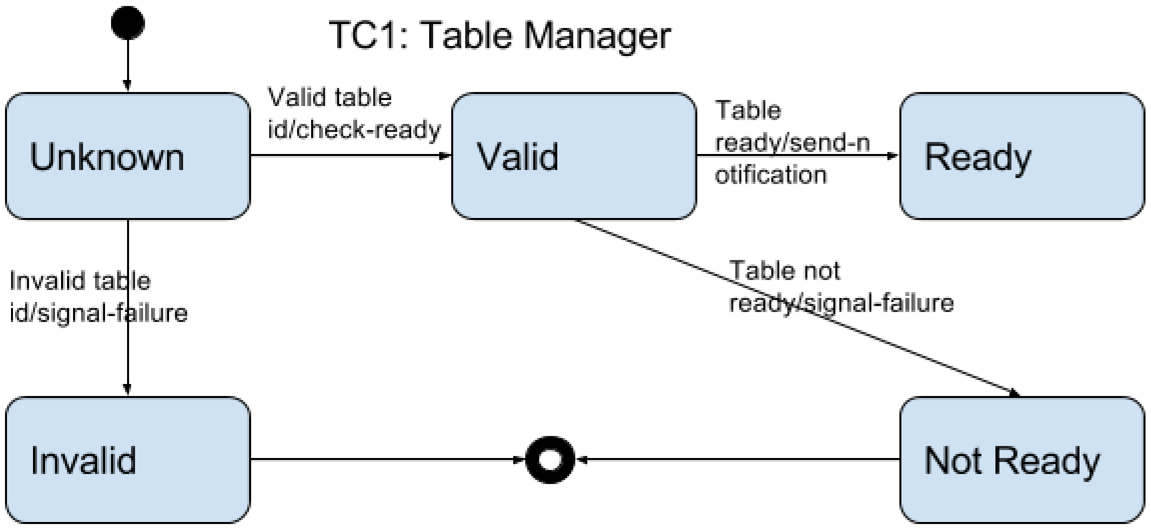




Test Cases

Table Manager

Test-case Identifier: TC-1	
Function Tested: TableNotfiy(int TableID): void	
Pass/Fail Criteria: The function passes the test if the notification of a table being ready is passed to the waiter for a table that is actually ready or responds correctly an error.	
Input Data: Input a range of valid table ids [consisting of tables that are actually ready and those which are not] and a range of invalid table ids.	
Test Procedure:	Expected Results:
Call function (Pass) with a valid table id which is ready.	A notification with this table id is sent to the waiter.
Call function (Fail) with an invalid table id or a valid table id for a not ready table.	An error is sent over the network.

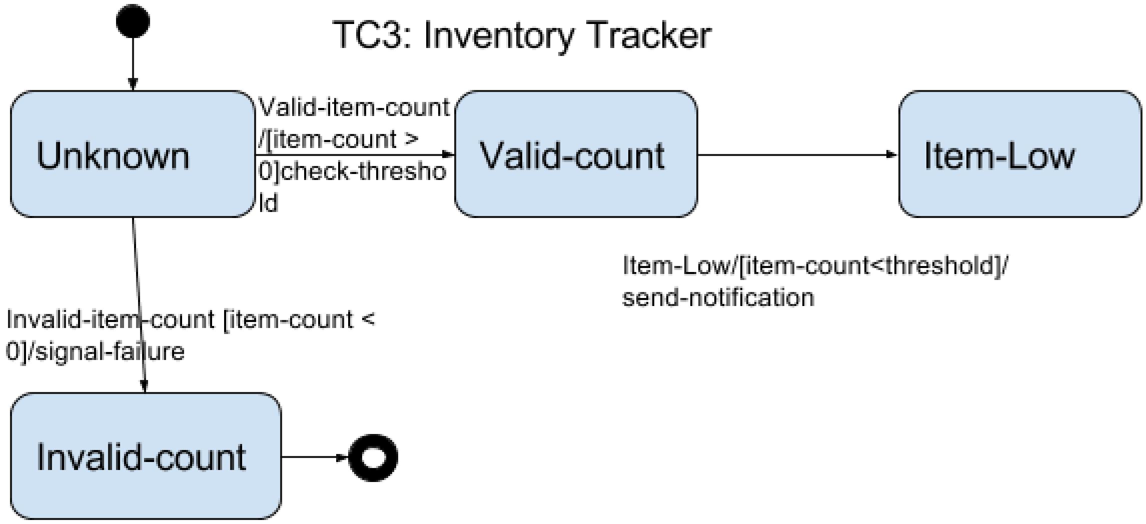

TC1: Table Manager
Unknown
Valid table id/check-ready
Valid
Table ready/send-notification
Ready
Invalid table id/signal-failure
Table not ready/signal-failure
Invalid
Not Ready

Test-case Identifier: TC-2

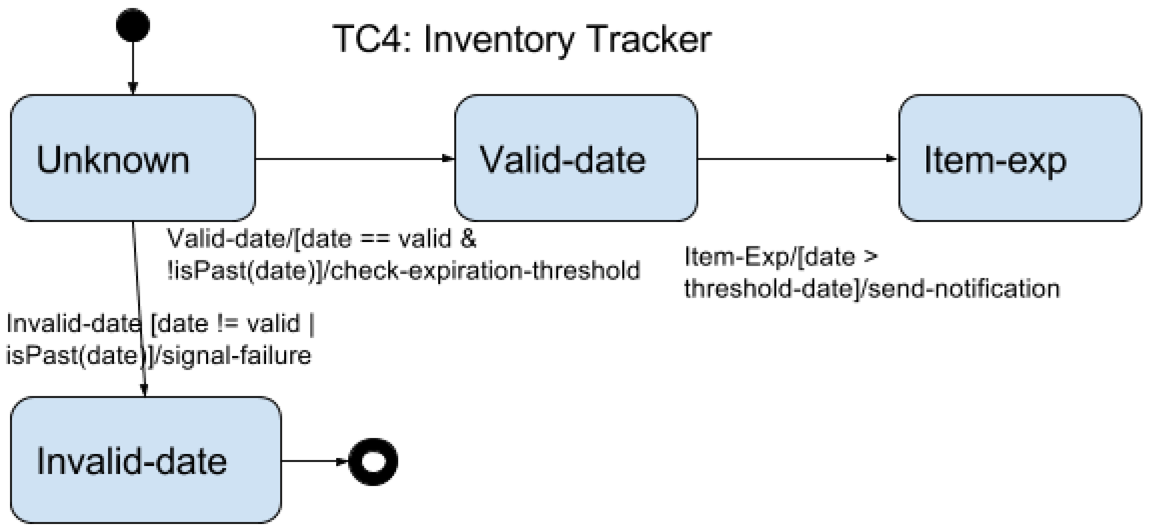
Function Tested: ProcessReq(): void	
Pass/Fail Criteria: The function passes the test if the function receives the request for a valid table over the network and performs requested status change, if such an action is valid to perform given the state of all tables.	
Input Data: Send over the network a range of requests that have valid table ids and contain actions that are valid for each of the tables and another range of requests that contain invalid table ids or valid table ids where the action requested cannot be performed.	
Test Procedure:	Expected Results:
Call function (Pass) and a request is sent to the function over the network containing a valid table id asking for a change of state of a table which can be performed.	The requested change of state of the given table is performed and seen by all of the system.
Call function (Fail) and a request is sent with an invalid table id or a request with a valid table id asking to reserve or take the table when such an action is not possible.	The function sends an error back to the sender

InventoryTracker

Test-case Identifier: TC-3	
Function Tested: SendLowNotification(int ItemCount): void	
Pass/Fail Criteria: The function passes the test if the function, given a valid item count (greater than 0) sends a notification over the network to the manager if such item count is below the stated threshold, or sends an error over the network if the ItemCount is invalid.	
Input Data: The function is called with range of valid ItemCounts and a range of invalid ItemCounts.	
Test Procedure:	Expected Results:
Call function (Pass) with a greater than zero item count.	The function sends a network message to the manager if the passed ItemCount is less than the threshold.
Call function (Fail) with an item count less than 0.	The function doesn't compare it to the threshold and simply sends an error stating so over the network


TC3: Inventory Tracker
Unknown
Valid-item-count /[item-count > 0]check-threshold
Valid-count
Item-Low
Item-Low/[item-count<threshold]/ send-notification
Invalid-item-count [item-count < 0]/signal-failure
Invalid-count

Test-case Identifier: TC-4	
Function Tested: SendExpirationNotification(int expDate): void	
Pass/Fail Criteria: The function passes the test if the function, given a valid expiration time, sends a notification to the manager containing a list of only those items that are close to expiration (given that the object knows the threshold defined as "close to expiration")	
Input Data: the function is given a range of valid expiration dates and a range of invalid expiration dates (i.e. not real dates)	
Test Procedure:	Expected Results:
Call function (Pass) is called with a valid expiration date	Sends a message over the network containing a list of items which are close expiration. Those items expirations are actually "close" to the given expiration date when using what the object was given as defined as "close to expiration".
Call function (Fail) the function was called with a date that is not real or invalid.	The function sends an error over the network.

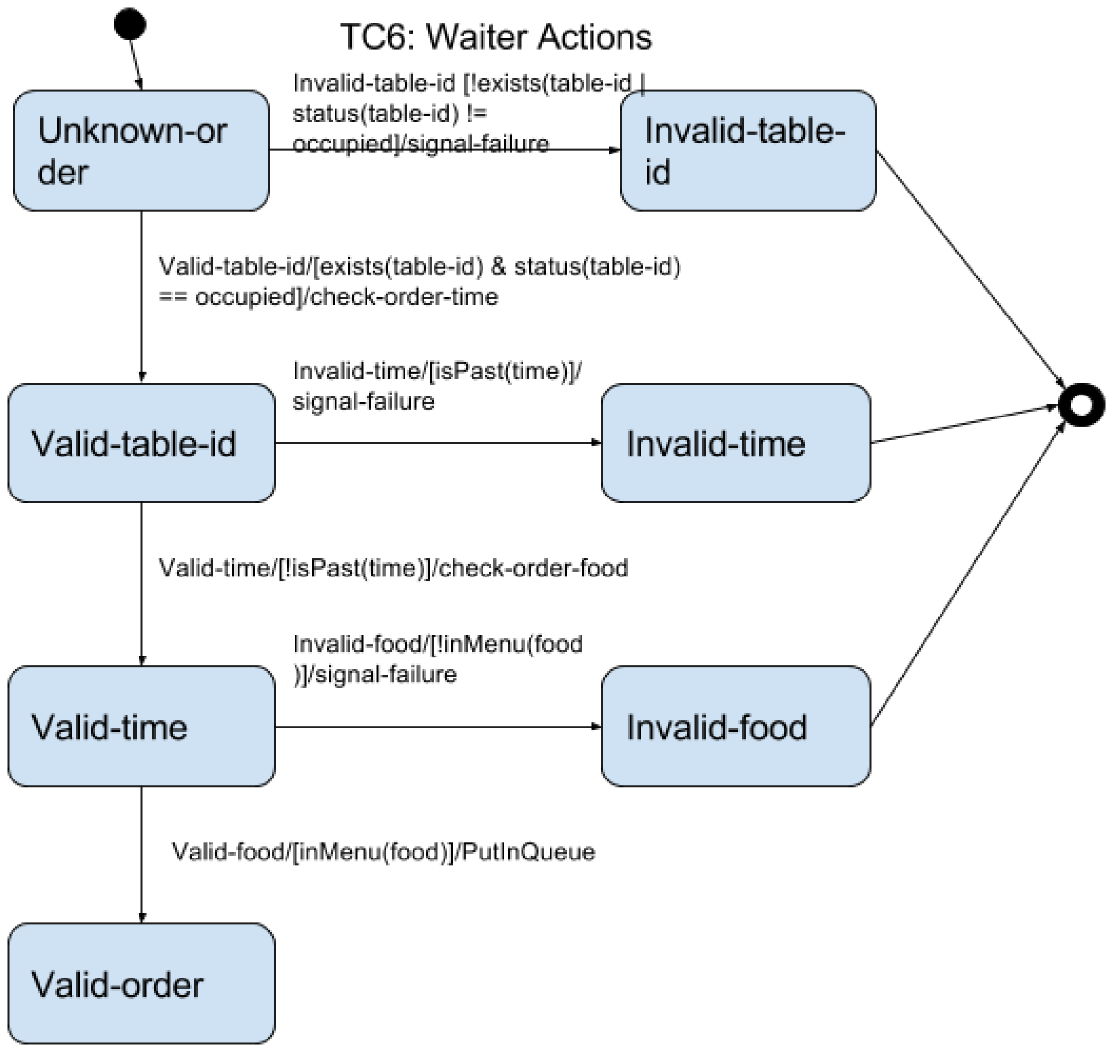

TC4: Inventory Tracker
Unknown
Valid-date
Item-exp
Valid-date/[date == valid & !isPast(date)]/check-expiration-threshold
Item-Exp/[date > threshold-date]/send-notification
Invalid-date [date != valid | isPast(date)]/signal-failure
Invalid-date

OrderQueue

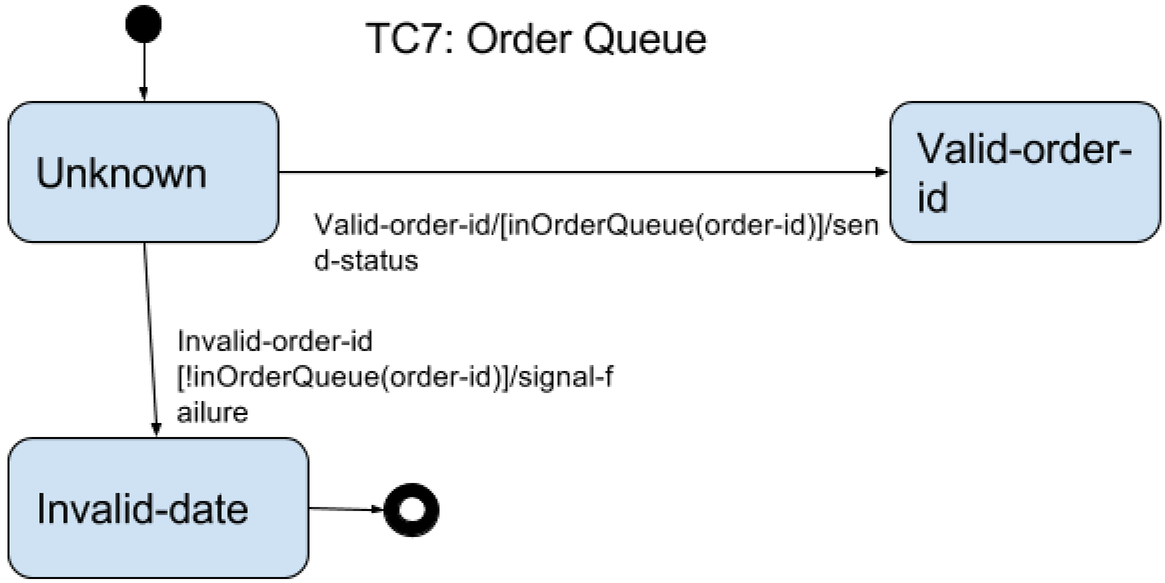
Test-case Identifier: TC-5	
Function Tested: PriorityCalc():void	
Pass/Fail Criteria: The function passes the test if it calculates the correct priority for all orders using the correct criteria	
Input Data: Input the order queue with a variety of items to see if they are prioritized correctly. Input items that test boundary cases (items that are difficult to prioritize)	
Test Procedure:	Expected Results:
Call function (Pass) when the queue has a large correct criteria.	All items are prioritized correctly using the valid list of items in it .
Call function (Fail) when the queue has no items.	No items are prioritized, the function does nothing.

WaiterActions

Test-case Identifier: TC-6	
Function Tested: PlaceOrder(int TableID, int OrderTime): void	
Pass/Fail Criteria: The function passes the test if it places all the items in the order in the order queue and they are placed in it correctly.	
Input Data: Input a range of valid tables and times and lists of food (from the network). Input range of invalid tables, times and lists of invalid foods (retrieved from the network).	
Test Procedure:	Expected Results:
Call function (Pass) with a valid table id, order time and food list(which is given through a network request).	All information is correctly placed in the order queue.
Call function (Fail) with invalid ids, times, or food list	An error is sent over the network.

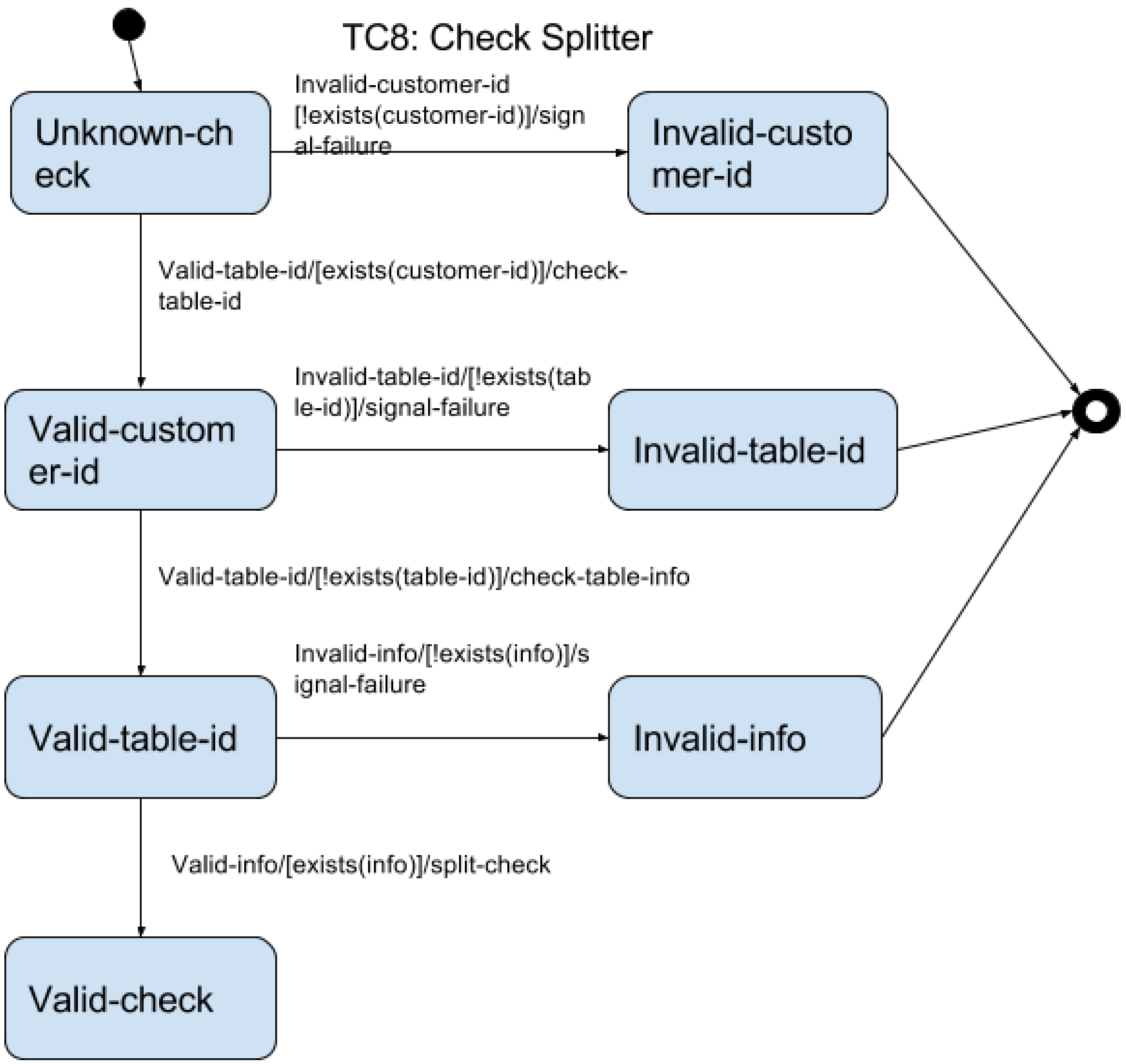

TC6: Waiter Actions
Unknown-order
Invalid-table-id [!exists(table-id | status(table-id) != occupied]/signal-failure
Invalid-table-id
Valid-table-id/[exists(table-id) & status(table-id) == occupied]/check-order-time
Valid-table-id
Invalid-time/[isPast(time)]/ signal-failure
Invalid-time
Valid-time/[!isPast(time)]/check-order-food
Valid-time
Invalid-food/[!inMenu(food)]/signal-failure
Invalid-food
Valid-food/[inMenu(food)]/PutInQueue
Valid-order

Test-case Identifier: TC-7	
Function Tested: PollFoodOrder(int OrderID): void	
Pass/Fail Criteria: The function passes the test if it makes a correct request to the order queue asking for the status of a valid order and receives a status back.	
Input Data: Input a range of valid of order ids and a invalid range of order ids	
Test Procedure:	Expected Results:
Call function (Pass) with a valid order id for that waiter.	The functions sends a valid request to the order queue over the network and receives a valid response
Call function (Fail) with an id that is not valid for that waiter.	The function sends an error over the network.


TC7: Order Queue
Unknown
Valid-order-id
Valid-order-id/[inOrderQueue(order-id)]/send-status
Invalid-order-id
[!inOrderQueue(order-id)]/signal-failure
Invalid-date

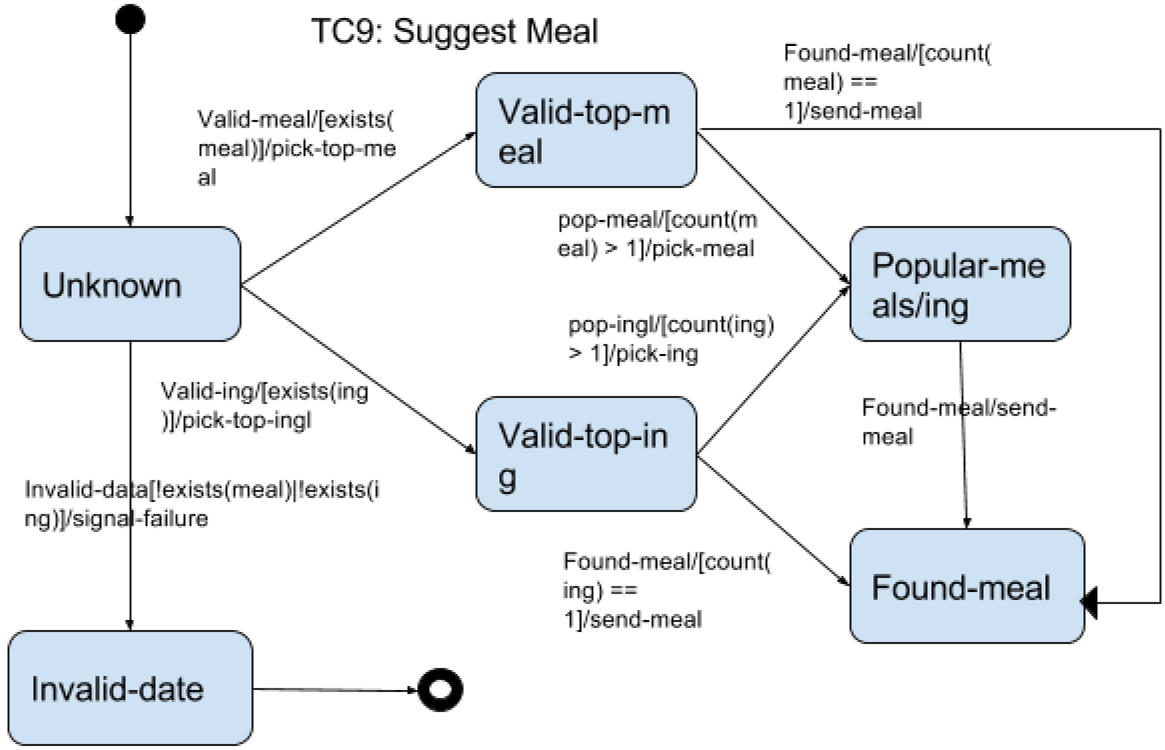
CheckSplitter

Test-case Identifier: TC-8	
Function Tested: SplitChecks(int CustomerId, int TableID, info TableInfo): void	
Pass/Fail Criteria: The function passes the test if it splits checks correctly among valid customers so that they pay for the food they are suppose to.	
Input Data: Input a range of valid customer ids, table ids and tableinfo. Input a range of invalid customer ids, table ids and invalid tableinfo.	
Test Procedure:	Expected Results:
Call function (Pass) with a valid customer id, table id, and table info	Each valid customer is given the correct amount to pay for their meal
Call function (Fail) with invalid customer id, table id, or table info.	The function should refuse to split the checks and return an error over the network.


TC8: Check Splitter
Unknown-check
Invalid-customer-id [!exists(customer-id)]/signal-failure
Invalid-customer-id
Valid-table-id/[exists(customer-id)]/check-table-id
Invalid-table-id/[!exists(table-id)]/signal-failure
Valid-customer-id
Invalid-table-id
Valid-table-id/[!exists(table-id)]/check-table-info
Invalid-info/[!exists(info)]/signal-failure
Valid-table-id
Invalid-info
Valid-info/[exists(info)]/split-check
Valid-check

MealSuggestions

Test-case Identifier: TC-9	
Function Tested: SuggestMeal(): string	
Pass/Fail Criteria: The function passes the test if it uses the correct top meal and ingredient to suggest a valid meal plan	
Input Data: Input a variety of meal statistics to the system.	
Test Procedure:	Expected Results:
Call function (Pass) with a lot of data regarding meal statistics	The function outputs a valid meal plan from the stats.
Call function (Fail) the function doesn't have much data	The function outputs that it cannot make a decision.


TC9: Suggest Meal
Unknown
Valid-top-meal
Valid-top-ing
Popular-meals/ing
Found-meal
Invalid-date
Valid-meal/[exists(meal)]/pick-top-meal
Valid-ing/[exists(ing)]/pick-top-ingl
Invalid-data[!exists(meal)|!exists(ing)]/signal-failure
pop-meal/[count(meal) > 1]/pick-meal
pop-ingl/[count(ing) > 1]/pick-ing
Found-meal/[count(meal) == 1]/send-meal
Found-meal/send-meal
Found-meal/[count(ing) == 1]/send-meal

GUI

Test-case Identifier: TC-10	
Function Tested: DrawFloorPlan(): void	
Pass/Fail Criteria: The function passes the test if it produces a correct mock of the floor plan for the host/hostess	
Input Data: Input a variety of data that could result in vastly different floor plan displays. Input invalid information about the floor plan.	
Test Procedure:	Expected Results:
Call function (Pass) multiple times iterating through different floor plans that are all valid.	All floor plans are produced correctly.
Call function (Fail) multiple times iterating through different invalid floor plans (tables that don't exists, states that are invalid, etc...).	An error should be returned for all floor plans. None should be displayed.

Test Coverage

Our system is composed of a plethora of classes. However, we decided to choose only the classes that are considered critical components of the system. These are the components that are performing lots of tasks for the system. In addition, we decided to test only the most complex methods of those classes. We believe these are the ones that require unit testing first and, if tested properly, represent how well the system performs initially. Most of the methods we will be testing interact with the database in someway. These are important to test since it is a data-driven system. Some of these tests, test the different connections in the system. After these tests are completed, we will develop tests for the smaller and less complex components and the other connections in the system. These current test cases cover multiple stages in our integration strategy.

The testing of algorithms used throughout the system is covered by the current test cases. These major algorithms include the order priority calculation (PriorityCalc) and the meal suggestion algorithm. We have also included the test of one of the most important interfaces in our system, the floor plan. The floor plan will require iterated testing utilizing a lot of different floor plans, as described in its test case.

Integration Testing Strategy

We chose to use bottom-up testing as our integration testing strategy. Bottom-up testing is mainly useful for two main reasons. First, it is simpler to unit-test modules that are not dependent on other modules. Since bottom-up testing starts with these modules, it is easier to isolate bugs by this method. Second, bottom-up testing is useful for projects that rely on a base strongly. In our project, all modules are highly connected but rely strongly on the database (it is a data-driven system). So, we start with testing the database, then we do bottom-up testing for each module.

Each module will be tested starting with the part that communicates with the database to the frontend. This fixes deeper and potentially more dangerous bugs first. Then, we test the connections between each module. This tests the entire system and should detect most of the bugs. Finally, we will do black box testing by forgetting our implementation. This final step allows us to see if our solution has fulfilled all the use cases.

Project Management and Plan of Work

Merging Contributions

Our team used Google Docs and Github as our tools for collaboration. Google Docs was used to merge our work on different parts of the report and format it properly. Github was used as a central repository for all our files.

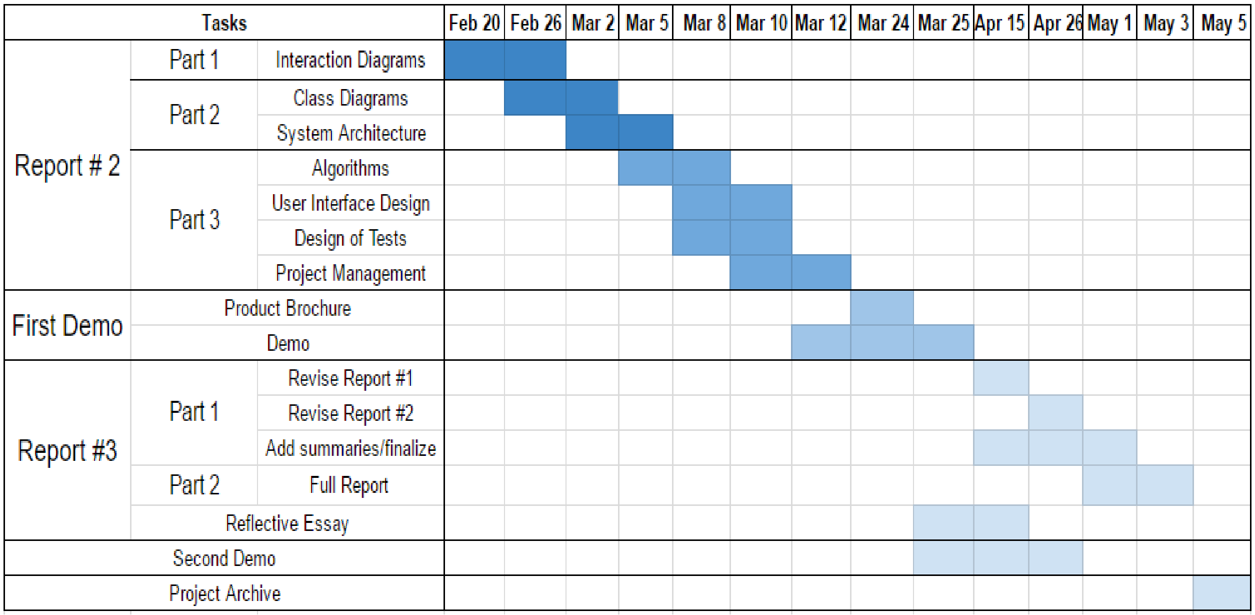
Problems/Issues Encountered:

We had some trouble meeting properly for the last few weeks because of the hectic midterm schedule of each of our members. However, we were able to efficiently delegate tasks beforehand to compensate for the missing member(s).

Project Coordination and Progress Report

Our team has aggressively tackled the development of this project to complete it before the deadline. We chose to use MeteorJS as our full stack development toolbox. This allowed us to not worry about integrating different pieces of backend and frontend. Instead, we focused on building the frontend of the project. So far we have been able to make HTML+CSS files for our various frontends (chef, waiter, etc). Then, we pivoted to build the backend. We have built an ER model of the database. Now we are focusing on the server programming and connecting the frontend and the database.

Plan of Work



Tasks			Feb 20	Feb 26	Mar 2	Mar 5	Mar 8	Mar 10	Mar 12	Mar 24	Mar 25	Apr 15	Apr 26	May 1	May 3	May 5
Report # 2	Part 1	Interaction Diagrams	■	■												
	Part 2	Class Diagrams		■	■											
		System Architecture			■	■										
	Part 3	Algorithms				■	■									
		User Interface Design					■	■								
		Design of Tests					■	■								
		Project Management						■	■							
First Demo	Product Brochure									■						
	Demo								■	■	■					
Report #3	Part 1	Revise Report #1										■				
		Revise Report #2											■			
		Add summaries/finalize										■	■	■		
	Part 2	Full Report												■	■	
	Reflective Essay										■	■				
Second Demo											■	■	■			
Project Archive																■

Note: The different colors indicate that there is a deadline for submission is between them.

Breakdown of Responsibilities

To make sure that our team is productive and efficient we divided up into 3 groups of two. Since we have completed an initial design of our system together, our next few weeks would be to build prototype of the system. Since we have not completely created detailed algorithms, we will try to build the UI and use basic/naive algorithms according to the concepts in the domain model. We created the tasks by grouping similar use cases in a subsystem and assigning each subsystem to a subteam. As each feature is developed by the sub-teams they will be performing tests on it so its not left until last minute.

Teams:

Dylan Herman and **Moulindra Muchumari** are responsible for Cleaning, Billing, and Manager subsystems.

Completed Tasks:

- Created preliminary UI design of the Manager's data and statistics view
- Planned out the use cases and domain models for the management subsystem
- Design what data points are to be collected from the restaurant and possible statistical measurements that can be analyzed from the data.
- Implement the UI design for the manager's console and make changes accordingly
- Create the split check interface

Current Tasks:

- Create/list out possible settings that the manager should manage
- Implement the archiving and statistics system
- Keep attending weekly meetings

Future Tasks:

- Design and Implement the various statistics
- Create the database with necessary authorization/privileges for manager data

Mit Patel and **Raj Patel** are responsible for the Ordering subsystem.

Completed Tasks:

- Created the UI design for the chef's Order Queue
- Planned out the use cases and domain models for the Order Queue subsystem
- Created the mathematical models for managing order queue

- Implement the UI design for the chef's Order Queue
- Create UI design for the waiter's tablet to place orders and receive notifications

Current Tasks:

- Implement the Order Queue subsystem using the mathematical model and domain analysis
- Keep attending weekly meetings

Future Tasks:

- Create a sample menu/ menu categories
- Design/Implement the order queue algorithm

Nill Patel and **Prabhjot Singh** are responsible for Seating and Inventory Management subsystems.

Completed Tasks:

- Created an interactive UI design for the floor plan using different colors for the status
- Keep attending weekly meetings
- Planned out the use cases and domain models for the Floor Plan subsystem
- Created the mathematical models for suggesting meal plans
- Implement the floor plan UI (only the non-interactive part)
- Create the reservation interface frontend
- Design the UI for raw materials (ingredients) interface that displays the list of materials and their amounts

Current Tasks:

- Implement the floor plan UI (the interactive part)
- Implement the food suggestion subsystem using the mathematical model and domain analysis
- Keep attending weekly meetings

Future Tasks:

- Design/Implement the reservation algorithm

Design Principles

To create our project design, we followed the principle of GRASP. The “controller” in our case is represented by the database. This is because the database is used in each subsystem to relay information between other subsystems. We don’t have alternative assignments because we did not decide on any significant trade offs between two similar designs. The choices we made were most optimal for our project design. For example, we chose to have a main controller to handle all information that is passed around each class rather than having all the classes depend on each other. Therefore, individual components in our system are not fully dependent on other components, meaning if one part of the system fails, other parts will still be running. Furthermore, we also embodied the principle of high cohesion by making sure that each class had its own well defined task. As a result, the reusability of our project is increased.

Improvements from Past Projects

In comparison to previous projects, the main intellectual contribution from our team would be the aspect of collecting a lot of data and using that data to determine trends that would help the overall efficiency and profitability of the restaurant. An instance of this can be seen in our inventory system. This system takes into account not only the ingredient usage but also keeps track of which ingredients are popular alongside the meals that are popular and uses them to determine suggestions that maximize profit as well as customer satisfaction. Simply keeping track of meal popularity is not enough since multiple meals may be popular just because they have a popular ingredient. It is important to take this into account so restaurants can make more of the popular meal that maximizes their profit and not have to make all popular meals.

Another instance where our project differs from others is the Order Queue Management system. Ordering meals to be cooked and pushed out to the customer is more complex than simply first-in, first-out ordering. Things that must be taken into account are: how large is a given order and how much time on average does it take to make a meal in that order? What type of meal is it, appetizer, entree, or dessert? What "stage" of the meal is one table on vs another? Simple first-in, first-out might put the table who orders all their meals at once over a table who just order one meal a little later than that table. Both tables should be able to get at least one meal before they get their second. We have taken into account these complexities and have devised mechanisms for ordering the meals in a fair fashion something other projects have not done.

All team members managed the project equally!

References

- "What's is the difference between include and extend in use case diagram?" N.p., n.d. Web. 5 Feb. 2017. <http://stackoverflow.com/questions/1696927/whats-is-the-difference-between-include-and-extend-in-use-case-diagram>.

- Marsic, Ivan. "Software Engineering Project Report." Software Engineering Project Report - Requirements. N.p., n.d. Web. 19 Feb. 2017. <http://www.ece.rutgers.edu/~marsic/Teaching/SE/report1.html>.

- Russ Miles and Kim Hamilton: *Learning UML 2.0,* Reilly Media, Inc. 2006.

- StarUML
  -Program used to create Diagrams

- Axure
  -Program used to create wireframes for the UI

- Meteor
  -Web framework written using NodeJS used for our project
  https://www.meteor.com/

- Semantic-UI
  -User Interface library used for our project
  https://semantic-ui.com/

- GRASP (object-oriented design). N.p., n.d. Web. 12 Mar. 2017. <https://en.wikipedia.org/wiki/GRASP_%28object-oriented_design%29>.

- BootStrap 3
  User Interface library used for our project
  http://getbootstrap.com/
- MongoDB
  -Collection Based Database
  https://www.mongodb.com/