Restaurant Automation & Optimization

Report 3

May 2, 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

[X*] Represents References

Overview of System Features

Features Implemented

1. Inventory (Managers View)
    a. Automatically tracks the restaurant's inventory, and notifies appropriate personnel when certain items are running low.
    b. Manages inventory orders, and updates the inventory database accordingly.
2. Order Queue (Chef's View)
    a. Handles the priority in which the order items are presented to the chef.
    b. Allows the chef to check order items when they are done, and to add more cook time to an order if it is behind schedule.
    c. Keeps track of the actual time taken to cook each order item. Also, automatically updates the cook time for that item to represent a more accurate cook time value.
3. Reservation (Customer's View)
    a. Allows the customer to place reservations through the restaurant's website.
    b. Sends the customer a confirmation email for all successful reservations
4. Floor Plan (Host's/Waiter's/Busboy's View)
    a. Shows the host/waiter/busboy a live view of all the tables current statuses and sizes.
    b. Assists host in determining the relative seatings of the customers that walk in based on the information presented to the host from the floor plan.
    c. Allows the waiter to select the table they are serving.
    d. Allows the busboy to see which tables need to be cleaned.
    e. Enables waiter/host to merge tables to create a larger table to satisfy the need of a larger party of people.
5. Menu Suggester (Managers View)
    a. Suggests to the manager, with the use of the data collected from previous orders, a list of menu items to keep and which ones to get rid of.
6. Statistics (Manager's View)
    a. Presents the manager with an overview of various of statistics pertaining to past orders, time spent by customer in restaurant, number of reservations over time, and many more that can be used by manager to make informed business decisions.
7. Waiter (Waiters View)
    a. Helps the waiter manage tasks such as ordering food, completing bills, tracking and table statuses for the tables he/she is serving.

Even though we implement a good amount of features for our project there are numerous features that we have not implemented. Features like employee portals, employee wage system, etc. were not implemented because for our project we wanted to focus more on complex and new features that have not been implemented by previous groups in the past. We decided to focus on the above features because we wanted to spend most of our time on attributes that would make our project stand out in comparison to previous projects along with going above and beyond just restaurant automation. Spending time on features that have not been implemented in the past would allow us to provide a greater intellectual contribution to the project at hand.

Summary of Changes

Project Objectives

From the beginning of the semester the main goal was to create an application that would automate a restaurant's operations. However, as the semester progressed and we tried to implement the different use cases that we had come up with, we realized that we wanted to make an application that was more than just restaurant automation. We came up with ideas that seemed to not only automate the restaurant, but optimized the overall restaurant business. It seemed as if that coming up with the novel idea was more of a process then something that we came up with overnight. With each progression our project objective evolved into something that was truly novel. So one major change that we have in this report since the past two reports is the inclusion of not only automation within our project but also the aspect of optimization.

Instead of listing all the different changes of the sections in one list we decided to add the list of changes for each section within its own respective section. So this way, the reader can read the section and see how we evolved in that particular section instead of a quick summary in the beginning. Instruction on how to view the changes are mentioned below

Summary of Individual Section Changes

If you want to see the changes made to each section of the report follow the instructions below:

1) Customer Statement of Requirements - Refer to the subsection “Changes from past reports” of this section.
2) Glossary of Terms - Refer to the subsection “Changes from past reports” of this section.
3) User Stories - Refer to the subsection “Key Revisions” of this section
4) Functional Requirements Specification - Refer to subsection “Implemented Use Case Summary” of this section.
5) Domain Analysis - Refer to the subsection “Changes from past reports” of this section.
6) Interaction Diagrams - Refer to the subsection “Changes from Previous Interaction Diagrams” of this section.
7) Class Diagram - Refer to the subsection “Changes from past reports” of this section.
8) System Architecture and System Design - Refer to the subsection “Changes from past reports” of this section.
9) Algorithms and Data Structures - Refer to the subsection “Changes from past reports” of this section.
10) User Interface and Design Implementation - Distinction between the previous reports and the new changes are made by using two sections. One is called “User Interface and Design Implementation” and the other one is called “Final User Interface and Design Implementation.”
11) Design of tests - Refer to the subsection “Changes from past reports” of this section.

Customer Statement of Requirements

Running a restaurant with pen and paper in the age of technology is getting very inefficient. In order to keep up with the competition and take advantage of the opportunities provided by new technologies. We would like to have a system that will integrate all the employees and allow for seamless tracking of each task. Automation decreases cost by hiring less people and helps to remove the human error that causes customer dissatisfaction.

Customers don’t have a way to express feedback regarding their experience

Sometimes our customers may have a bad experience at our restaurant that can lead to a loss of customers. If we are aware of such negative experiences, we can prevent them from occurring in the future, thus increasing customer satisfaction. Currently, submitting complaints at our restaurant comes at a cost of time to both the manager and the customer. Usually, our customers do not want to go out of their way to waste time submitting a complaint. Moreover, spending time attending to customer complaints comes at an inconvenience to the manager because he or she must temporarily stop overseeing restaurant operations.

Imagine a full time employee at an office decides to take their one-hour lunch break at our restaurant across the street. The customer arrives at our restaurant, waiting to be seated. A waiter unwelcomingly greets the customer. Usually, such an incident would not be worthy of complaining about to the manager. The customer is seated, and the waiter takes the customer's order. The food arrives, but the customer receives the wrong item. The waiter sighs and brings the food back to the kitchen. Fifteen minutes later, the customer’s food finally comes out. After finishing his/her food the customer pays the bill and leaves the restaurant in a hurry because they had to go back to work. The customer was unable to give the restaurant’s manager feedback about the poor service they received and most likely wouldn’t go back there to eat.

We would prefer a solution that would allow us to easily receive customer feedback without coming at a cost of time to us or the customer. Also, we would like to be able to view and save customer feedback for personal records to prevent any feedback from being missed.

There is a communication issue between the host and the customer to notify the customer when he or she can be seated

When customers arrive at our restaurant, they want to spend less time waiting, and more time dining with their friends and/or family. During busy hours, wait times can be as long as one to two hours long, meaning that the lobby will be crowded. In such circumstances, it is difficult for the host/hostess to notify a customer that their table is ready above all the noise and ruckus. Miscommunication errors can lead to inefficiency in the seating system, thus increasing overall wait time and customer unhappiness.

Assume a family of six people visit our restaurant at 6PM on a Friday night to enjoy a family dinner. The family enters the restaurant and the host informs them that the wait time will be one hour and 30 minutes to be seated. The family decides to take advantage of the long wait time by visiting the mall across the street. One hour later, an unexpected amount of tables were vacant, making room for the family of six. Unfortunately, the family was not present at the time they were at the top of the queue. The host spent 5 minutes looking for the family of six, unable to find them. As a result, the host assigned the table to the next customers in line. The family arrived 10 minutes later, only to learn they had to wait an extra hour.

We would prefer to have a system that will let us contact our customers when there is a wait time. The customers should be able to keep in contact with us to know if they would like to keep their reservation.

Managing customer checks both efficiently and in an organized manner

Using the old fashioned system to generate bills and keep records is cumbersome, inefficient, and unorganized. The current system requires our waiters to manually calculate subtotals for a customer's bill, which can be time consuming and prone to errors. Moreover, keeping records of all checks for future references requires extra work and responsibility for the manager of our restaurant.

Imagine a party of 15 walks into our restaurant, and they order a lot of food. When it comes time for the bill, the waiter has to start manually adding up each order to get the total for the bill. This can take quite some time because the waiter has to make sure the bill is accurate. After the bill is paid, the waiter adds the bill to the daily stack of bills which will then need to be logged.

One possible solution that could help us solve this is a system that will help us remove the manual calculation of bills and store the data on a computer. This will help save time for both the customer and the restaurant.

Inefficiency with table management & cleaning

A common problem we find in our restaurant is the inability to keep track of unoccupied or dirty tables, which leads to inefficiencies in seating our customers. This means we lose potential revenue because our customers experience longer wait times and are not satisfied.

When a customer comes into our restaurant they first approach the host to be seated. The host has to look at the whiteboard and see which tables are available and seat the customer(s) accordingly. At busy times in our restaurant, the customer may have to wait a significant amount of time for a table to become vacant. However, the wait time is longer than it should be because the whiteboard doesn't get updated instantly when the busboy cleans the table.

A solution we believe would solve this problem is to have a system that allows us to view the dining area in realtime. It would display when tables are vacant, occupied or dirty using different colors and patterns to distinguish between the three statuses. This would allow us to view instantaneous updates about the status of the table and decrease wait time and increase customer satisfaction, when compared to the traditional whiteboard method.

Splitting checks & generate bills

When parties finish their meal, there are instances where they forget to mention splitting checks in the beginning and inform the waiter of splitting checks when the total check has already been calculated. In such a case, the waiter would have to ensure the orders of each individual and split the checks accordingly. In such a situation the waiter would have to go back to the cashier and inform them to print individual checks for each customer. Situations like these usually become a hassle for both the waiter as well as the customers. This leads to customer dissatisfaction, adds to the wait time of the customer(s) which in turn makes that table unavailable for a longer period time for the next customer(s).

A possible solution to this should allow us to split the checks whenever necessary, so this way we don't end up wasting time and resources and can seat the incoming party as soon as possible.

Wasted time in relaying the order to the kitchen

In our restaurant, the waiter has to jot down the order on a notepad for the kitchen and take a carbon copy to the cashier. This is very inefficient in the sense that the waiter will have to make multiple trips between the table, kitchen and the cashier. Doing so for multiple tables, takes away time from serving our customer's needs. The kitchen notifies the waiter when the order is ready by ringing a bell, but the waiter is not going to know whether the bell was rung for his order or a different one. This wastes the time of our waiters and causes them to run around for no reason. Also, we had previously lost or damaged the paper copies. Sometimes we have issues reading the waiter's handwriting.

A possible solution is to make this entire process electronic. The waiter could take the order on a tablet or some device instead of jotting down on a notebook. Then, the order could be sent electronically to the kitchen and the cashier. Also, the same system could notify the waiter when his order is ready. We think this would really decrease the amount of trips to be taken by the waiter. Moreover, we think it would be easier for us to process the bills.

Prioritizing and managing the flow of orders in the kitchen

Our kitchen is typically a chaotic place. Most of chaos results from us having to keep track of all the orders and which of our waiters they can from. However, not only do orders have to be served in a first-in first-out fashion, but the time of delivery made needs to dynamically change based on multiple factors such as: fairness, is our restaurant having a slow day and they want to keep as many customers in the restaurant as possible to appear "busy", does a customer want their food to be held until they finish their appetizer?, and etc. Thus managing the kitchen is more complex than simply queueing up orders as they come. That's why adding automation can create order in our kitchen.

A solution we believe would solve this problem is to have a system that automatically keeps track of orders as they come in. However, in our restaurant the orders that come first are not always the orders that get dished out first. We would need to have multiple order queues to make sure all tables get the food in a fair manner (appetizers come first, etc...). We would also like the system to automate the delegation of tasks (which cooks cook what). These multiple queues would also be help for customers that push orders back; they should be put behind other orders that have not been dished out yet.

Managing and keeping track of all the raw materials in the kitchen

Our restaurant has problems with tracking raw materials in the kitchen. We usually order the raw materials for the next week so we receive them on Sunday before the week starts. Since we can't know exactly how much we will use, we order more than required and throw away extra materials at the end of the week. We lose a lot of money and materials because of this. We tried to keep track of the raw materials every day by noting them on paper. But, our cooks found it difficult to note down what they used while cooking. Sometimes we were wrong with our estimate and we ran out of materials during the week.

We would like a solution that electronically deducts materials whenever the cooks finish an order. Then, the system could notify us to if any materials are running low, so we could order them on the go.

Planning the menu and activities for the day

Many restaurants plan activities the night before to streamline cooking for the next day. This allows them to serve more customers and reduce the waiting time for customers to receive their food. A lot of restaurants do not plan these activities properly or allocate enough time to finish them the night before. Hence, this reduces productivity next day and increases stress and unhappiness in the kitchen. Moreover, this might directly affect the revenue of the restaurant and decrease customer satisfaction.

A solution is to combine the planning activities with the cooking activities, so that planning for the next day happens concurrently with serving customers. This solution might work because the planning activities can be completed when the restaurant might not be busy. Also, this simplifies managing the kitchen for the Chef by allowing one system to assign activities for the cooks.

On many days, we are unprepared for cooking items in the kitchen because we did not prepare the required materials properly the day before. We would like a solution that will help us properly prepare the required materials for the next day to improve the efficiency in our kitchen.

Cleaning dishes

Busboys in our restuarant perform two main tasks: clean tables and clean dishes. They have be in both the kitchen and the dining area and it causes a lot of running around. It is also hard for the chef to find a busboy to clean dishes. This decreases the efficiency of the chef because he needs to be in the kitchen to manage the cooks. Some restaurants mitigate this problem by having some busboys stay in the kitchen as dishwashers. However, this is an inefficient solution because the dishwashers are not needed when there are no dishes to be washed and the busboys don't need to be in the dining area when there are no dirty tables. Moreover, this requires us to hire more people for this solution.

We would like to have a system that will tell the busboys when the dishes or tables need to be cleaned. The chef's should be able to contact any of the busboys that are free. Overall we want less clutter in the dining area and kitchen.

Changes from past reports

The customer statement of requirements did not change much since the beginning of the semester since what the customer expected from the application has not changed. However, throughout the duration of this semester we have only focused on the requirements that we thought were the most useful to the customer and promoted both automation and optimization for their restaurant. Focusing on requirements of priority ensured our time was spent addressing important problems. We also focused on the requirements that were not previously done, so we could challenge ourselves and innovate new features.

Glossary of Terms

Manager - Responsible for managing employees, overseeing the daily operations of the restaurant, and ensuring profitability of the restaurant.

Kitchen - The area where the chef(s) cooks and prepares food. Also, the busboy(s) transports and cleans dishes in this area.

Chef - A professional cook that prepares food for the restaurant.

Customer - A person that purchases food or service at the restaurant.

Waiter/Waitress - Responsible for taking orders and completing requests for customers to ensure customer satisfaction.

Host/Hostess - Manages the restaurant lobby and makes seating arrangements for customers .

Busboy - Cleans dishes and clears tables for the restaurant.

Dining Area - The area where customers eat their meals.

Cashier - A waiter responsible for completing customer transactions.

Split Check - A check that is customized based on all the items purchased by one individual.

Lobby - The area where the customer checks into the restaurant and waits for a table.

Exclusive - An item that has been popular for the duration of a week, a month, and a quarter is classified as an exclusive item.

High - An item that has been popular for the duration of a quarter or a quarter and a month is given the “high” popularity level.

Medium - An item that has been popular for the duration of a week or a week and a month is given the “medium” popularity level.

Low - An item that has been popular for the duration of a week or a week and a quarter is given the “low” popularity level. Items that have not been popular for any of the durations are given the default “low” level as well.

Technical Terms

Graphical User Interface - A visual interface that allows workers at the restaurant to interact with the system

Order Queue - lists out food orders to the chef based on first come first serve

Database - A storage device that archives daily restaurant sales, inventory, and feedback.

Floor Plan - An interface that is used to show which tables are currently busy, free ,or dirty.

Notification - A way to alert employees so they can run the restaurant more efficiently.

Reservation - A system that allows customers to claim a table at a specific time before going to the restaurant

Stage - Represents the three categories of meals used in our order queue system i.e appetizer, entree dessert

Walk-in - When customers come into the restaurant without requesting reservation prior to coming in

CookTime - Represents the time that it takes to cook an order item

Item Priority - Represents the priority value that is used to order food items in the order queue used by the chef

Menu Suggestion - An insight provided to the manager indicating to either remove or keep the item on the menu.

Inventory Threshold - Predetermined low limit for each ingredient in the inventory, which is used to determine if it needs to be restocked.

Manual Merge - Merging tables together by selecting them manually

Changes from past reports

There have not been any major changes within this section other than a couple of additions of new terms to make sure that the reader understands some of the terminology that we came up with as our project evolved.

User Stories

High (core) Priority
Medium Priority
Low Priority

Waiter

Table 1.1

Identifier	User Story	Size
ST-W-1	As a waiter, I can remotely send customer orders directly to the chef	6
ST-W-2	As a waiter, I can choose to split the check multiple ways or generate one check.	5
ST-W-3	As a waiter, I can receive notifications when the food is ready for my table.	4
ST-W-4	As a waiter, I can receive notifications when a table requests my assistance.	3
ST-W-5	As a waiter, I can see how long each order will take approximately based on the data collected from previous orders.	5

Table 1.01

Customer

Identifier	User Story	Size
ST-C-1	As a customer, I can remotely make a reservation for the restaurant.	4
ST-C-2	As a customer, I can receive remote notifications that indicate when my table is available.	2
ST-C-3	As a customer, I can alert the waiter to request assistance.	3

Table 1.02

Host

Identifier	User Story	Size
ST-H-1	As a host I shall be able to see a floor plan of the dining room with which seats are available.	7
ST-H-2	As a host I shall be able to mark tables as reserved or available.	3
ST-H-3	As a host I should be able to see real time updates of the expected time a table should be done.	6

Table 1.03

Busboy

Identifier	User Story	Size
ST-B-1	As a busboy I shall be immediately notified when a table needs to be cleaned	3
ST-B-2	As a busboy I shall be notified when I should go to the kitchen to clean dishes	3
ST-B-3	As a busboy I shall be able to mark tables as cleaned	2
ST-B-4	As a busboy I shall be notified when to bring more refreshers to tables.	3

Table 1.04

Manager

Identifier	User Story	Size
ST-M-1	As a manager, I can see the status of every item in the inventory. Such as if an item is running low, it will give me an indication.	6
ST-M-2	As a manager, I can modify accounts and permission for each employee and add or remove new ones as necessary.	5
ST-M-3	As a manager, I can view all the transactions that took place in a given day.	3
ST-M-4	As a manager, I can view statistics on the sales in a given day.	4

Table 1.05

Chef

Identifier	User Story	Size
ST-CF-1	As a chef, I can view orders that have been placed by the customers	4
ST-CF-2	As a chef, I can notify the waiter that the order is done	4
ST-CF-3	As a chef, I can manage the inventory, and notify the manager when items are running low	7
ST-CF-4	As a chef, I can prioritize and manage orders in the queue	6
ST-CF-5	As a chef, I can view the status of each table	3
ST-CF-6	As a chef, I can alert the busboy to clean the dishes	2
ST-CF-7	As a chef, I can make the use of raw materials more efficient using the data from the system	7
ST-CF-8	As a chef, since I pre-make dishes, I can get information on the most popular dishes of the past day, so I can decide which dishes to prep before tomorrow's opening.	6

Table 1.06

Key Revisions

Features Not Implemented

- **ST-W-4, ST-C-3**: Decided not to leave tablet devices at each table to keep minimize the cost to the restaurant, and avoid theft.
- **ST-H-3**: One of the data processing algorithms we decided to exclude given the project time frame and the lower significance of this algorithm compared to the other algorithms.
- **ST-B-2, ST-B-4, ST-CF-6** This feature was not too significant for us to focus on implementing given the project time frame. The busboy functionality is incorporated into the floor status, making up for the omission of these user stories.
- **ST-CF-5**: We decided not to allow the chef to view the status of each table because he only needs to focus on the operation of the order queue. Viewing the status of each table is irrelevant to a chef's job description.

Features Implemented but with modifications

- **ST-W-2** : Splitting checks between multiple people was not implemented, but waiter can still generate one check for that order.
- **ST-W-3, ST-CF-2**: The chef marks the items as complete however, there is no notification implemented for the waiter to see when they're done.
- **ST-C-2** : Instead of receiving a notification when a table is available for the customer, the customer gets an email after reserving their table.
- **ST-CF-3, ST-M-1** : These two user stories can go together since they are both accomplishing the same task but just for different users. So Chef and Manager can both access and modify the inventory. They also can see when the item is below the threshold. The chef doesn't notify manager about low threshold, the system does.
- **ST-M-2**: Any employee can make their account using the roles. Manager or anyone else does not have the ability to change the permissions or remove the accounts, once they're created.
- **ST-CF-8**: The data is available to the manager, so we did not directly make the previous day's data available to the Chef. So the data is there but not made into a user interface for the chef.

Many of the user stories are specific to certain types of users of the system. Therefore, some user stories overlap, making it repetitive to implement similar things multiple times. Instead, we implemented the more useful user story, and made the data available system-wide. Hence, any future developers can easily implement the omitted/modified user stories. As a result, we were able to allocate our time to more distinct features.

Functional Requirements Specification

Stakeholders

Restaurant Owners/Managers - Restaurant owners/Managers would primarily have interest in our system because it provides a method of automation to optimize the efficiency of the restaurant. In turn, the owners would benefit from a rise in profit and a decrease in wasted resources (i.e. time, money, etc).

Customers - Customers benefit from the system in two ways without being the primary users of the system. The customers benefit from the improved service generated by the system itself (i.e. food tracking, instant service requests, etc). Also, the reservation system provides an easy way for customers to make reservations at a particular date and time without having to interact with a person.

Restaurant Employees - Restaurant employees benefit from this system because many of their daily tasks and responsibilities are automated by the system, thus increasing coordination, reducing human error, and reducing wasted resources.

Actors & Goals

Customer
Initiating and Participating
Role: Actor visiting the restaurant to order food and or drinks.
Goal: Customer makes reservations, places orders, and calls waiter.

Waiter
Initiating and Participating
Role: Employee of restaurant that is in charge of taking care of customers.
Goal: Waiter places the order, get notification when food is ready or when the customer requests assistance, and prepares the bill.

Host
Initiating and Participating
Role: Employee that greets customers in the lobby area.

Goal: Uses floor plan to assign tables to customers and manages reservations.

Chef

Initiating and Participating

Role: Employee that prepares all the food for customers in the kitchen

Goal: Gets orders from the waiter and prepares cooking. Notifies waiter when food is ready. Prepares food for the next day based on previous statistics. Notifies Busboy when dishes are full/dirty.

Busboy

Initiating and Participating

Role: Employee that is responsible for keeping the restaurant area clean

Goal: Busboy gets notified when he/she has to clean the tables and dishes.

Manager

Initiating and Participating

Role: Employee that oversees every other employee and manages the entire restaurant

Goal: Gather data from restaurant transactions and manage all employee accounts

Use Cases

Casual Description

Our group's user stories accommodate the causal description requirement. However, to make the mapping between the two more clear here is a table that shows the mapping

Use Cases	User Stories
UC-1	ST-W-1
UC-2	ST-W-2
UC-3	ST-W-3, ST-CF-2
UC-4	ST-W-4, ST-C-3
UC-5	ST-W-5
UC-6	ST-C-1
UC-7	ST-C-2
UC-8	ST-H-1, ST-H-3
UC-9	ST-H-2
UC-10	ST-B-1
UC-11	ST-B-2
UC-12	ST-B-3
UC-13	ST-B-4
UC-14	ST-M-1
UC-15	ST-M-2
UC-16	ST-M-3, ST-M-4
UC-17	ST-CF-1, ST-CF-4

UC-18	ST-CF-3
UC-19	ST-CF-5
UC-20	ST-CF-7
UC-21	ST-CF-8

Table 1.07

Use Case Diagram

We Split up the Use Case Diagram so it will be easier to trace and understand.

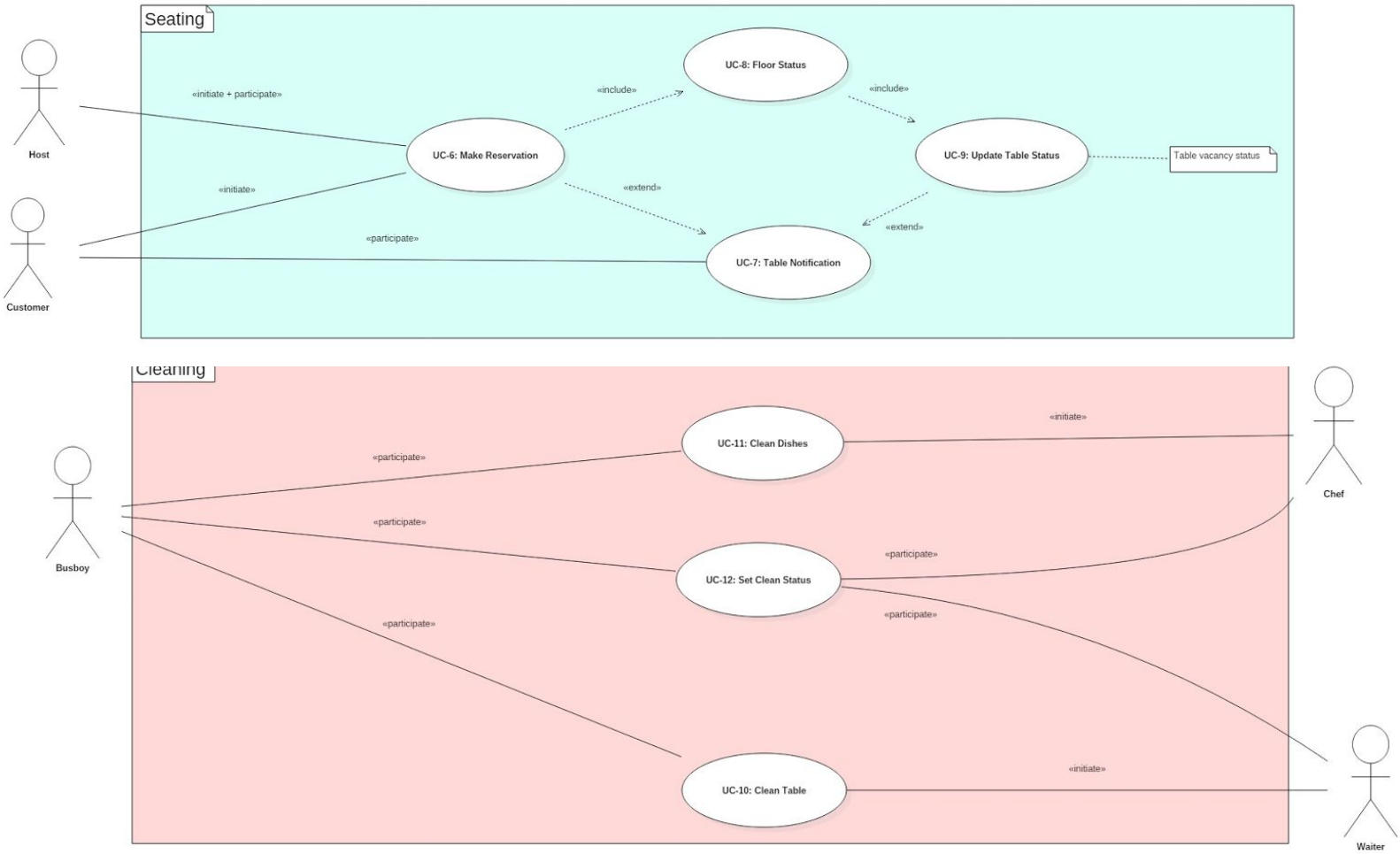


Figure 1.01

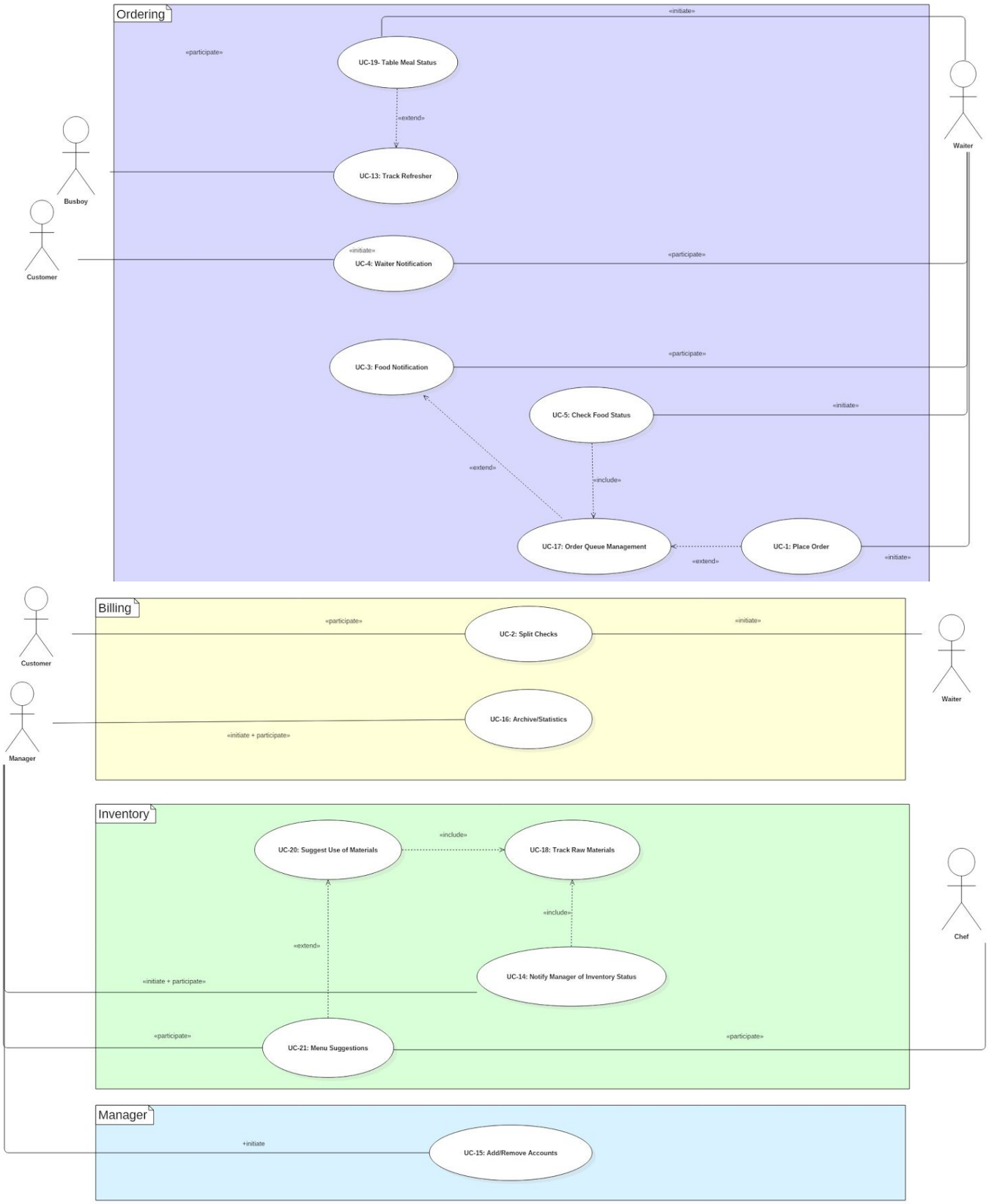
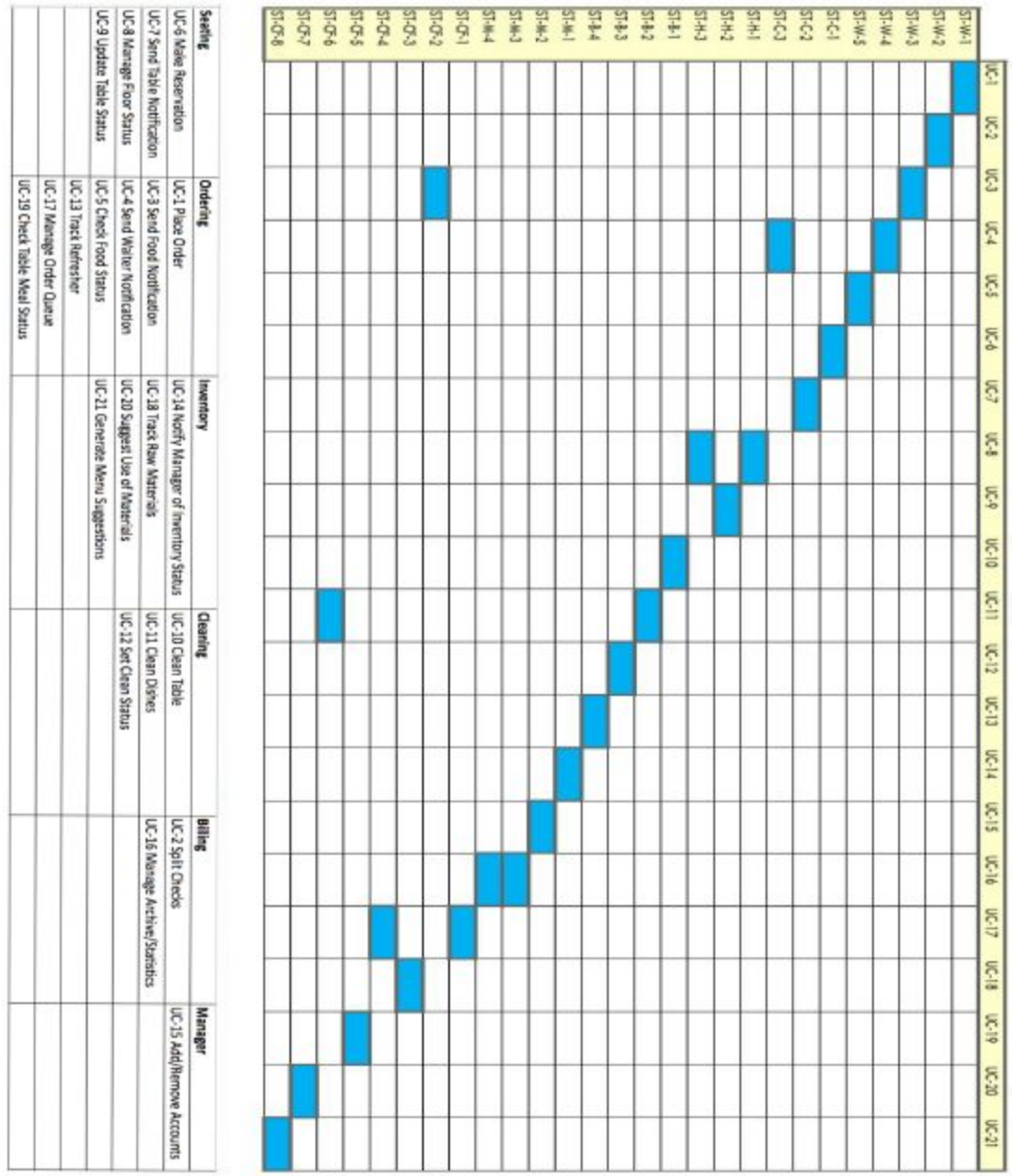

Ordering
«initiate»
«participate»
UC-19- Table Meal Status
«extend»
Waiter
Busboy
UC-13: Track Refresher
Customer
«initiate»
UC-4: Waiter Notification
«participate»
«participate»
UC-3: Food Notification
UC-5: Check Food Status
«initiate»
«extend»
«include»
UC-17: Order Queue Management
UC-1: Place Order
«extend»
«initiate»
Billing
Customer
«participate»
UC-2: Split Checks
«initiate»
Waiter
Manager
UC-16: Archive/Statistics
«initiate + participate»
Inventory
UC-20: Suggest Use of Materials
«include»
UC-18: Track Raw Materials
Chef
«include»
«extend»
UC-14: Notify Manager of Inventory Status
«initiate + participate»
«participate»
UC-21: Menu Suggestions
«participate»
Manager
+initiate
UC-15: Add/Remove Accounts

Figure 1.02

Traceability Matrix Diagram



Seating	Ordering	Inventory	Cleaning	Billing	Manager
UC-6 Make Reservation	UC-1 Place Order	UC-14 Notify Manager of Inventory Status	UC-10 Clean Table	UC-2 Split Checks	UC-15 Add/Remove Accounts
UC-7 Send Table Notification	UC-3 Send Food Notification	UC-18 Track Raw Materials	UC-11 Clean Dishes	UC-16 Manage Archive/Statistics	
UC-8 Manage Floor Status	UC-4 Send Waiter Notification	UC-20 Suggest Use of Materials	UC-12 Set Clean Status		
UC-9 Update Table Status	UC-5 Check Food Status	UC-21 Generate Menu Suggestions			
	UC-13 Track Refresher				
	UC-17 Manage Order Queue				
	UC-19 Check Table Meal Status				

Figure 1.03

The traceability matrix (figure 1.2) above shows how the use case's map to the user stories previously described. It helps figure out which use cases play a crucial role in this system.

Fully-Dressed Description

Use Case UC-8 Manage Floor Status
Related User Stories: ST-H-1
Initiating Actor: Host, Busboy, Waiter
Actor's Goal: Uses floor plan to manage tables and reservations for customers
Participating Actors: Host, Busboy, Waiter
Preconditions: ● Database has all tables and its corresponding attributes ● List of upcoming reservations
Postconditions: ● Each table shows correct status and attributes
Flow of Events for Main Success Scenario: -> 1) User logs into system <- 2) Login success status sent back -> 3) If successful, a request is sent for an updated floor plan view <- 4) An updated view of the floor plan is returned with the updated statuses -> 5) Seat customer at an appropriate table <- 6)Updates the table statuses appropriately -> 7) Waiter serves the customer and checks off order as completed when they are done <- 8) The table statuses are updated appropriately -> 9) Busboy cleans the tables and sets the table as cleaned when done <- 10) The table statuses are updated to display the new status of the table

Table 1.08

[4*]Sequence Diagram Use Case UC-8

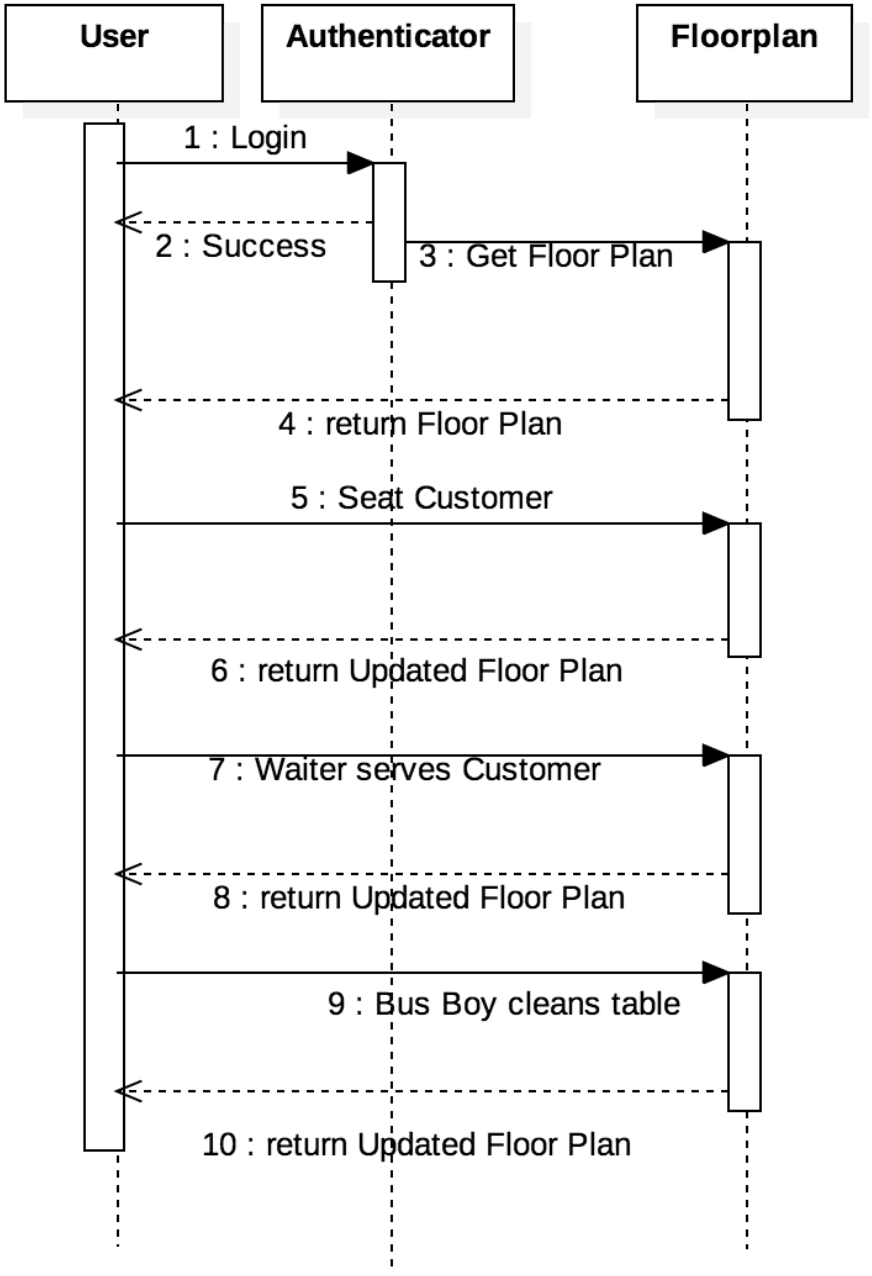


Figure 1.04
User represents the participating actors: Host, Waiter, Busboy.

Use Case UC-17 Manage Order Queue
Related User Stories: ST-CH-1, ST-CH-4
Initiating Actor: Chef
Actor's Goal: The chef manages items sent to the order queue by starting to cook items, marking items as completed, and adding extra time to a started item if needed.
Participating Actors: Waiter
Preconditions: ● The order placed by the waiter was successfully added to the order queue. ● Other orders may or may not exist in the order queue. ● The order and individual order items within an order are not already in the order queue.
Postconditions: ● The chef completes an order, marking each item and order as completed. ● The order item is removed from the order queue.
Flow of Events for Main Success Scenario: ->1) Waiter places an order. <- 2) The order is prioritized into order the queue based on the stage of the meal (i.e. appetizers before main meal). -> 3) The chef starts to work on an a particular meal by tapping on the respective row. -> 4) The timer runs out and the item is marked as completed.
Flow of Events for Alternate Success Scenario (1): ->1) Waiter places an order. <- 2) The order is prioritized into order the queue based on the stage of the meal (i.e. appetizers before main meal). -> 3) The chef starts to work on an a particular meal by tapping on the respective row. -> 4) The chef presses the completed icon to finish the item before the timer runs out.
Flow of Events for Alternate Success Scenario (2): ->1) Waiter places an order. <- 2) The order is prioritized into order the queue based on the stage of the meal (i.e. appetizers before main meal). -> 3) The chef starts to work on an a particular meal by tapping on the respective row. -> 4) The chef taps the timer icon to add more time to cook the meal. <- 5) The chef completes cooking the item.
Flow of Events for Alternate Success Scenario (3): ->1) Waiter places an order.

<- 2) The order is prioritized into order the queue based on the stage of the meal (i.e. appetizers before main meal). -> 3) The chef starts to work on an a particular meal by tapping on the respective row. -> 4) The chef marks an item as completed. -> 5) The chef *undos* the completed item because the chef did not actually finish cook the item. -> 6) The chef completes cooking the item.

Table 1.09

Sequence Diagram Use case UC-17

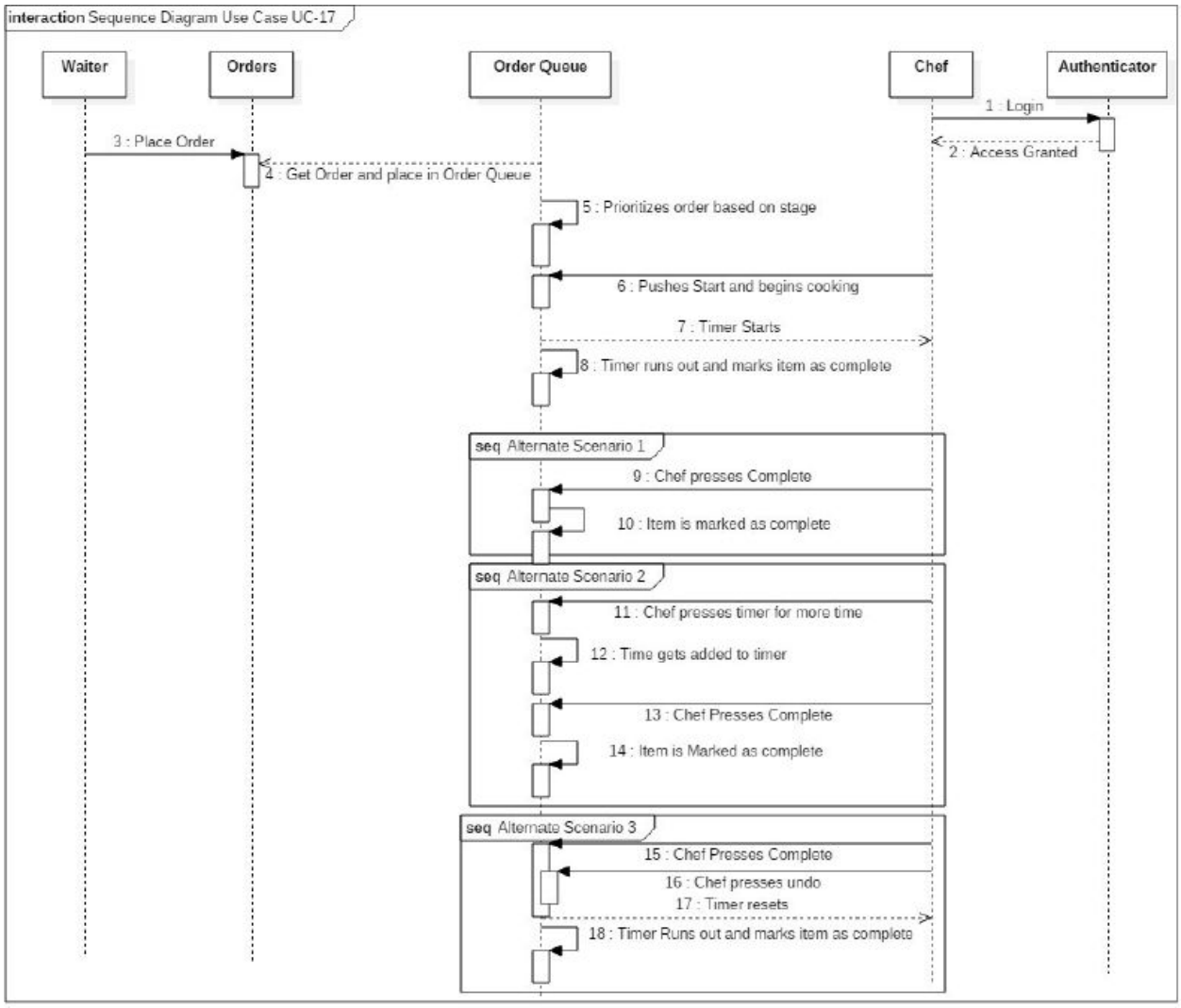


Figure 1.05

Use Case UC-16 Manage Archive/Statistics
Related User Stories: ST-M-3, ST-M-4
Initiating Actor: Manager
Actor's Goal: Manager analyzes statistics and data to get an insight on financial decisions to make to improve the restaurant's business.
Participating Actors: None
Preconditions: ● There must be existing data in the database
Postconditions: ● Each statistics gives accurate sales, averages, and sums ● Manager is able to make sound decisions about the decisions
Flow of Events for Main Success Scenario: -> 1) Manager logs in <- 2) Login Authenticated -> 3) Manager selects data he/she wants to view <- 4) System provides statistics on select option

Table 1.10

Sequence Diagram: Use Case UC-16

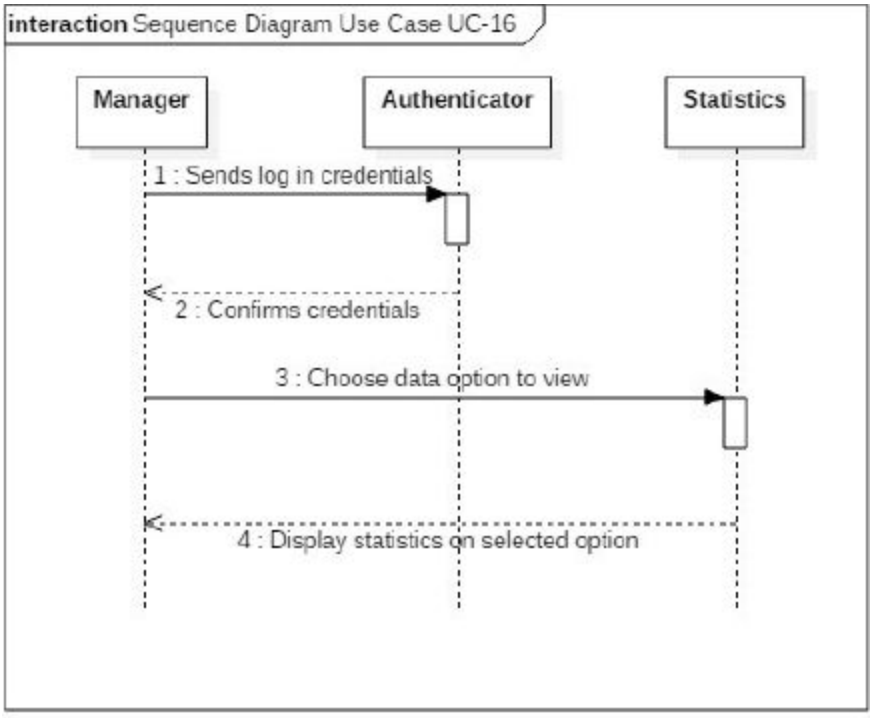


Figure 1.06

Use Case UC-18 Track Raw Materials
Related User Stories: ST-CF-3
Initiating Actor: Manager
Actor's Goal: Keep track of raw materials in the inventory. Update the inventory values as the items are used. Find out if an item is running low
Participating Actors: None
Preconditions: ● There must be existing data for each item in the inventory in the database
Postconditions: ● Inventory values are updated ● Get a low threshold warning
Flow of Events for Main Success Scenario: -> 1) Manager logs in <- 2) Login Authenticated <- 3) Inventory is displayed -> 3) When a chef completes an item, the ingredients are subtracted from the inventory. <- 4) Updated inventory is displayed

Flow of Events for Alternate Success Scenario (1):
-> 1) Manager logs in
<- 2) Login Authenticated
<- 3) Inventory is displayed
-> 3) When a chef completes an item, the ingredients are subtracted from the inventory
<- 4) If the item is below threshold, update inventory with below threshold warning

Table 1.11

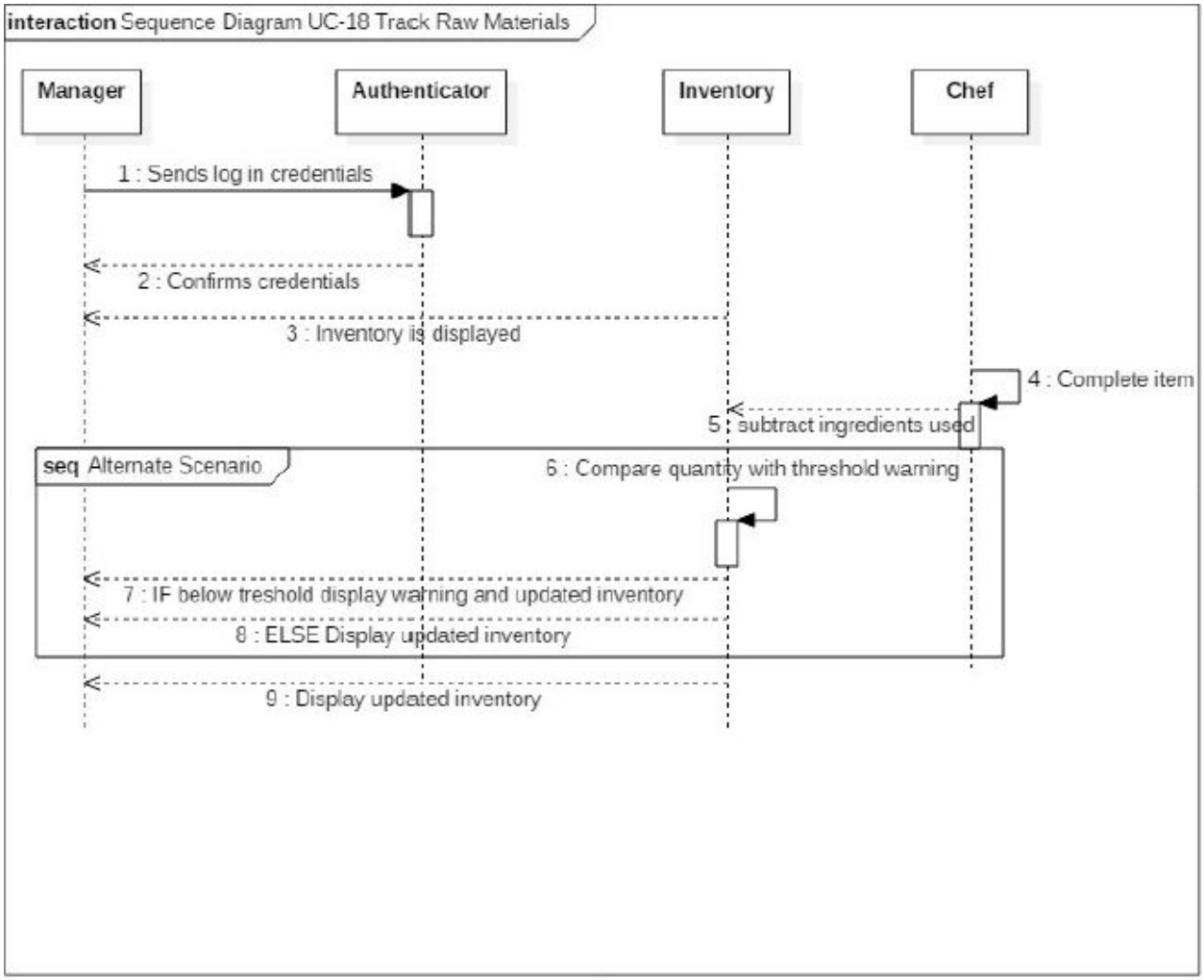


Figure 1.07

Use Case UC-21: Menu Suggestions
Related User Stories: ST-CF-8
Initiating Actor: Manager
Actor's Goal: Use menu suggestions to decide, which items should remain and be removed from the menu based on both profit and popularity.
Participating Actors: None
Preconditions: ● The database should contain orders for at least one quarter (92 days). ● The menu item database should be populated with at least one item. ● The inventory database should be populated with ingredients required by each existing menu item.
Postconditions: ● The manager will be presented with a breakdown of the popularities specified for each menu item.
Flow of Events for Main Success Scenario: -> 1) Manager logs into system <- 2) System verifies login <- 3) System performs menu suggestion algorithm -> 4) Manager analyzes menu suggestion results -> 5) Manager decides that current menu is satisfactory
Flow of Events for Alternate Success Scenario (1): -> 1) Manager logs into system <- 2) System verifies login <- 3) System performs menu suggestion algorithm -> 4) Manager analyzes menu suggestion results -> 5) Manager decides to remove certain items from the current menu

Table 1.12

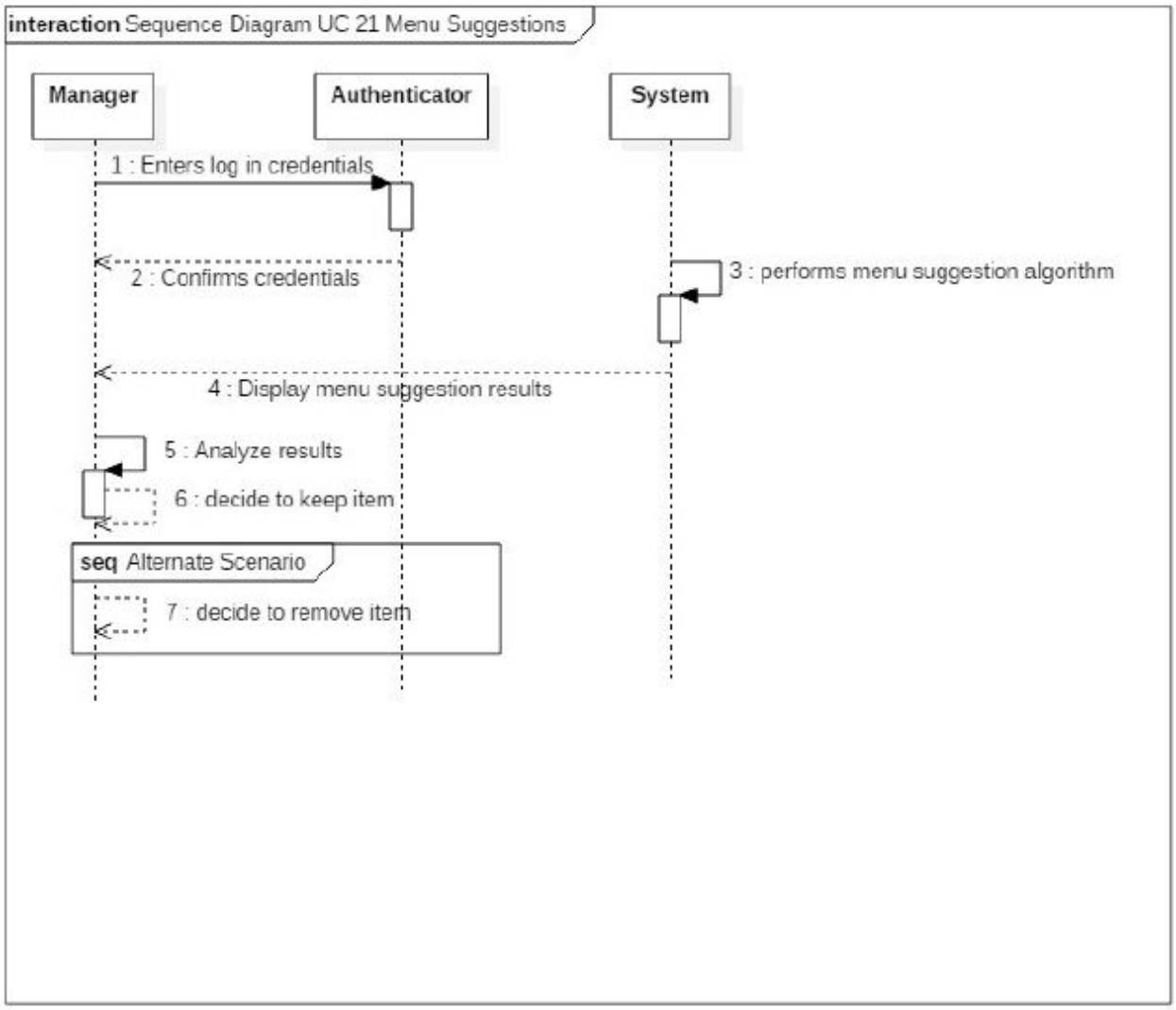

interaction Sequence Diagram UC 21 Menu Suggestions
Manager
Authenticator
System
1 : Enters log in credentials
2 : Confirms credentials
3 : performs menu suggestion algorithm
4 : Display menu suggestion results
5 : Analyze results
6 : decide to keep item
seq Alternate Scenario
7 : decide to remove item

Figure 1.08

Use Case UC-1 Place Order
Related User Stories: ST-W-1
Initiating Actor: Waiter
Actor’s Goal: To take a customer's order and send it to the order queue
Participating Actors: Customer
Preconditions: ● Customer is seated at the table and ready to order ● All menu items are loaded in the database
Postconditions: ● Customer's order will be in the system and pushed out to the order queue where the chef can see it
Flow of Events for Main Success Scenario: -> 1) Customer gives the waiter their order and the waiter places the order into the system <- 2) Order is placed into system successfully -> 3) Customer is done with food and asks the waiter for the bill -> 4) Waiter generates bill <- 5) Bill successfully paid by the customer

Table 1.13

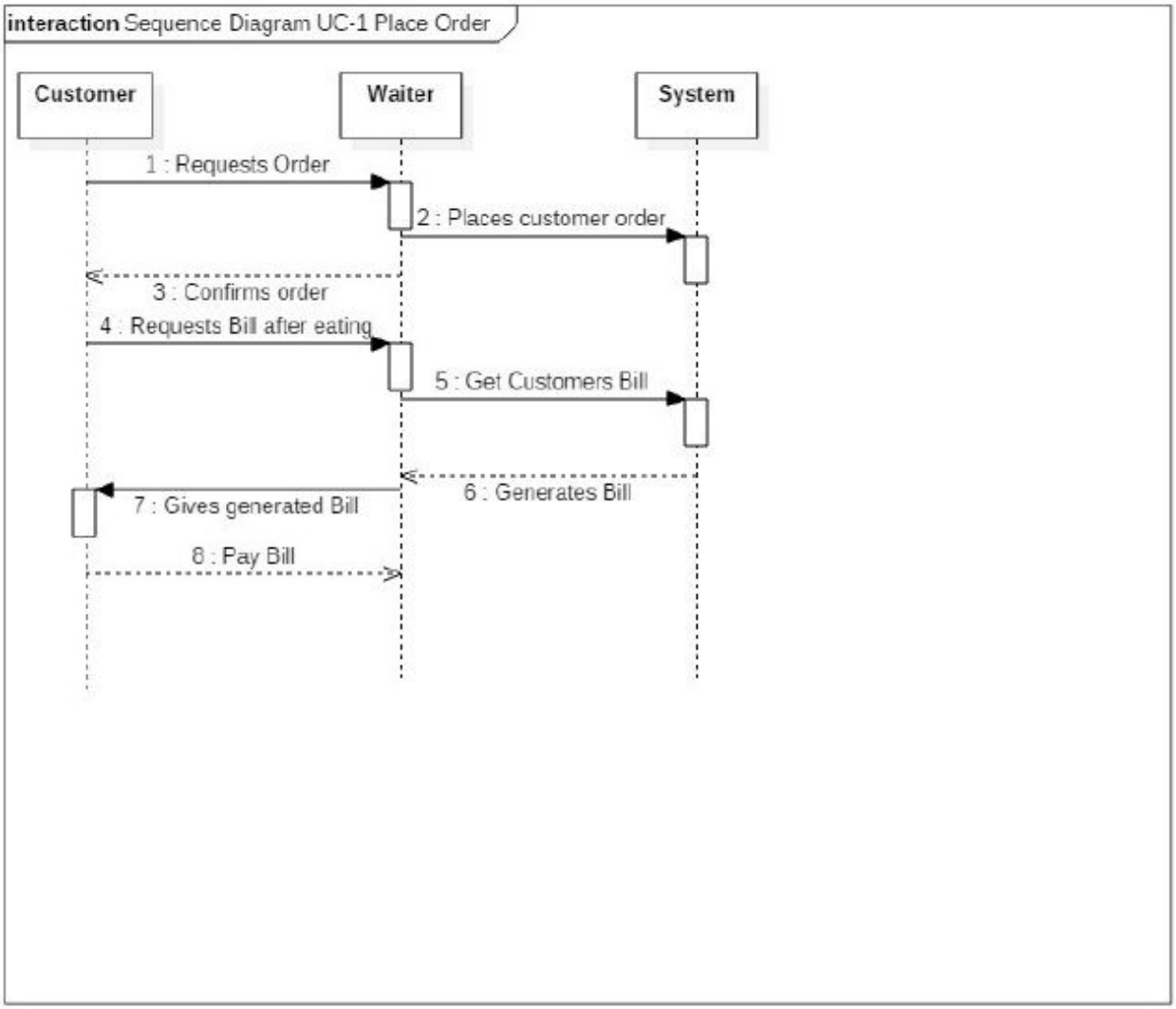


Figure 1.09

Use Case UC-6 Make Reservation
Related User Stories: ST-C-1
Initiating Actor: Customer
Actor's Goal: Make a reservation with relevant information at the restaurant
Participating Actors: None
Preconditions: ● Customer has access to internet or phone to make the reservation

Postconditions: ● A table is assigned to the reservation
Flow of Events for Main Success Scenario: -> 1) Customer submits a reservation request <- 2) System processes request and sends a confirmation
Flow of Events for Alternate Success Scenario (1): -> 1) Customer submits a reservation request <- 2) System declines request because there are no reservations available at that time

Table 1.14

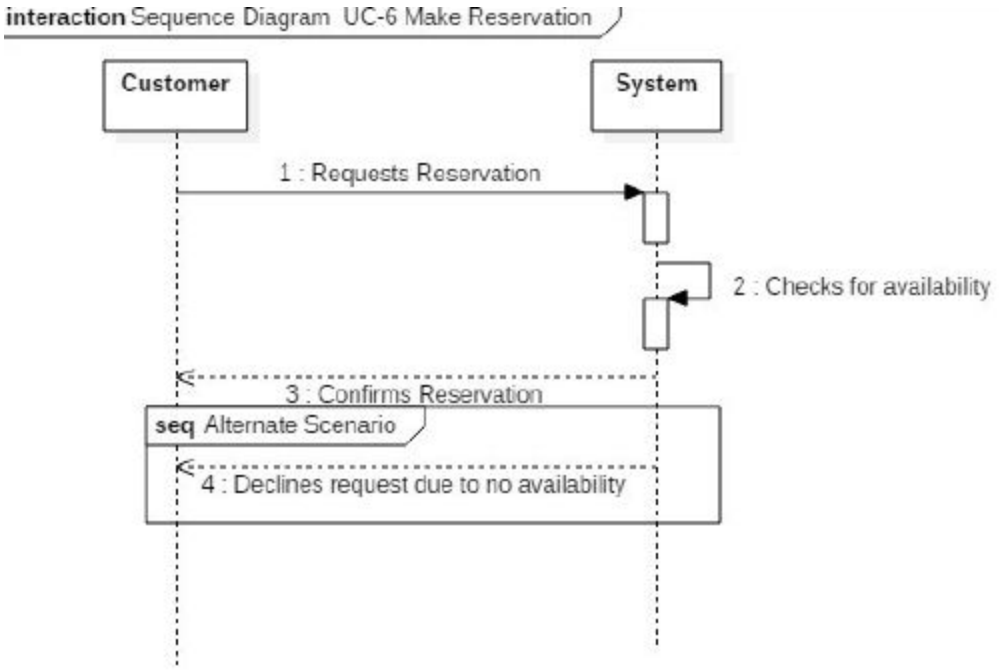


Figure 1.10

Implemented Use Case Summary

The dark-bulleted use cases below incorporate the light-bulleted uses cases in terms of how our system is implemented. When we started off this project we had multiple use cases for every single action taken by the user. However, as we implemented the application we realized that these individual actions could be grouped into larger use cases that would cover many of the smaller original use cases. Hence, we covered many small and similar uses cases in one general use case for the fully-dressed descriptions.

- UC-8 - Manage Floor Status
    - UC-9 - Update Table Status
    - UC-10 - Clean Table
    - UC-12 - Set Clean Status
- UC-6 - Reservation
    - UC-7 - Table Notification
- UC-18 - Track Raw Materials,
    - UC-14 - Notify Manager of Inventory Status
- UC-1 - Place Order
    - UC-2 - Split Checks

The above list mentions most of the use cases that we have covered in our implementation of the project for this semester. There are still multiple use cases that we did not cover because we wanted to focus on the more important use cases for the semester. However, in the future if this project were to go into production for an actual restaurant then those trivial use cases could easily be implemented.

User Effort Estimation using Use Case Points

PF = 28 hours per Use Case
Weight:
1-2 Easy
3-4 Normal
5-6 Hard

UC	Use Case Weight
UC -1: Place Order	3
UC -2: Split Checks	1
UC -3: Send Food Notification	2
UC -4: Send Waiter Notification	3
UC -5: Check Food Status	2
UC -6: Make Reservation	3
UC -7: Send Table notification	3
UC -8: Manage Floor Status	4
UC -9: Update Table Status	2
UC -10: Clean Table	2
UC -11: Clean Dishes	2
UC -12: Set Clean Status	2
UC -13: Track Refresher	1
UC -14: Notify Manager of Inventory Status	2
UC -15: Add/Remove Accounts	3
UC -16: Manage Archive/Statistics	4
UC -17:Order Queue Management	6
UC -18: Track Raw Materials	2
UC -19: Check Table Meal Status	1

UC -20: Suggest Use of Materials	1
UC -21: Generate Menu Suggestions	3

Table 3.1
UUCP = 52

Technical Factor	Weight	Perceived Complexity	Calculated Factor (Weight x Perceived Complexity)
1	1	2	2
2	.5	3	1.5
3	.5	1	0.5
4	1	5	5
5	.5	1	0.5
6	1	4	4
7	1	3	3
8	2	6	12
9	.5	.5	0.25
10	.5	0	0
11	.5	0	0
12	.5	.5	0.25
13	.5	1	0.5
14	.5	4	2
15	1	0	0
16	2	5	10
17	2	6	12
18	.5	3	1.5
19	.5	2	1

20	.5	3	1.5
21	1	4	4

Table 3.2

$Total\ Complexity\ Factor = 61.5$

$TCF = .6 + (61.5 * .01)$

$TCF = 1.215$

Environmental Factor	Weight	Perceived Impact	Calculated Factor (Weight x Perceived Impact)
1	1	2	2
2	.5	1	0.5
3	.5	3	1.5
4	1	4	4
5	.5	3	1.5
6	1	2.5	2.5
7	1	2	2
8	2	2	4
9	.5	.5	0.25
10	.5	1	0.5
11	.5	1	0.5
12	.5	2	1
13	.5	1	0.5
14	.5	4	2
15	1	2	2
16	2	1	2
17	2	1.5	3
18	.5	1	0.5

19	.5	2	1
20	.5	3	1.5
21	1	2	2

Table 3.3

$Environmental\ Factor\ Total\ =\ 34.75$

$ECF\ =\ 1.4\ +\ (34.75 * (-\ .03))$

$ECF\ =\ 1.4\ -\ 1.0425$

$ECF\ =\ .3575$

$UCP\ =\ UUCP\ *\ TCF\ *\ ECF$

$UCP\ =\ 52\ *\ 1.215\ *\ .3575$

$UCP\ =\ 22.586$

$Duration\ =\ UCP\ *\ PF$

$Duration\ =\ 22.586\ *\ 28$

$Duration\ =\ 632.4\ Hours\ =\ 26.35\ days$

Domain Analysis

Domain Model

The colors in each row represent the following

DELETIONS
INSERTIONS

Deletions - things we have removed since previous reports
Insertions - things we have added since previous reports

Concept Definitions:

UC- 8: Manage Floor Status, UC-9: Update Table Status, UC-7: Send Table Notification, UC-6: Make Reservation

Responsibility Description	Type	Concept Name
Authenticates and sets permissions for the user logging in.	D	Authenticator
Container for user's authentication data, such as pass-code, timestamp, permissions, etc.	K	Key
Collects data about reservations	D	ReservationTableManager
Actively collects table request data for a given table size cluster.	K	TableLoadStatus
Groups tables by clusters and manages the reservation interval and number of reservation to walkin tables for each table cluster	D	TableClusterManager
Pair up tables to make them a bigger size based on availability	D	TableBuddyManager
Manages status of table (clean, reserved, in-use, free) and takes state changes	D	TableStatusManager

Notifies relevant personnel of changes in table status	D	TableStatusNotifier
Displays total floor plan	D	FloorPlanPage
Displays reservation options	D	ReservationPage
Waiter requests the busboy to clean a specific table	K	CleanRequest
Host requests a table to be marked for walkin	K	WalkinRequest
Waiter sends notification when table is done and needs to be cleaned	K	SetCleanedRequest
Busboy receives a notification to clean a specific table	K	CleanNotify
Notify waiter when the table is ready	K	ReadyNotify

Table 1.16

UC-16: Manage Archive/Statistics, UC-2: Split Checks

Responsibility Description	Type	Concept Name
Authenticates and sets permissions for the user logging in.	D	Authenticator
Logs all daily sales of the restaurant	D	Bookkeeper
Extracts and calculates statistics from several data sets such as meals, sales, and profit for the requested time period	K	SystemStats
Container for past inventory count, transactions, and statistics	K	Archive
Updates and displays system time	D	UpdateTime
Pays the bill to the restaurant	D	BillPayer

Table 1.17

UC-17: Manage Order Queue, UC-1: Place Order, UC-3: Send Food Notification, UC-5: Check Food Status, UC-19: Check Table Meal Status

Responsibility Description	Type	Concept Name
Customer order is put into the food queue	D	PlaceOrder
Container for customer orders. Acts as controller and marks food as out for cooking and keeps track of food until it is fully cooked.	K	OrderQueue
Notification for when food is ready	D	FoodReady
Divides meals into stages and moves meals between stages based on time since order.	D	StageManager
Manages queue priority based on the time the order came in, the meal type, and the average time to make the meal in the order.	D	PriorityCalculator
Queries the order queue system to check if the food is ready or not.	D	FoodChecker
Relevant participants can track a table's meal stage.	K	MealStageTracker
a request for food to be marked as done	K	FoodDone
polls on behalf of the waiter to ask when parts of his order are ready (one per waiter order)	D	FoodPollRequest
Sent to waiter when his order is ready	K	OrderReady
Sent to waiter when his order item is ready	K	OrderItemReady
Calculates a new cook time for a menu item based on the actual time it takes for a chef to complete an order item	D	UpdateCookTime
Records the number of times an item has been ordered in its lifetime.	D	MenuItemTracker

Table 1.18

UC-14: Notify Manager of Inventory Status, UC-18: Track Raw Materials, UC-20: Suggest Use of Materials, UC-21: Generate Menu Suggestions

Responsibility Description	Type	Concept Name
Contains the state of the restaurant's current inventory.	K	InventoryContainer
Gets sent of list of ingredients and subtracts them from the inventory and forms an items list for the order which contains which "batches" of items to use	D	InventoryTracker
Contains the mapping of all meals to their respective ingredients. Processes meal requests from the chef.	D	IngredientTable
Keeps track of ingredient popularity.	K	IngredientPopularity
Keeps track of meal popularity.	K	MealPopularity
Based on meal popularity, meal prices,ingredient prices, and profitability it suggests a list food items to keep on the menu	D	MenuSuggester
Sent by the chef containing the meal to be made by him	K	MealRequest
Contains the list of items to be used to make the meal.	K	ItemsList
Sent to manager when the low threshold of an inventory item has been reached.	K	ItemLowNotifcation

Table 1.19

When considering all the use cases we came up with we avoided the domain modeling concepts for some of them because they weren’t core features of the project.

Association Definitions:

Floor Plan associations:

Concept pair	Association description	Association name
FloorPlanPage↔ TableStatusManager	TableStatusManager passes the status of all tables so that FloorPlanPage can display a visual to the host	send floor plan info
ReservationPage↔ TableStatusManager	TableStatusManager passes reservation table info for ReservationPage to display to customer	send reservation info
TableStatusManager↔ TableClusterManager	TableClusterManager passes the types and size of the tables and the interval for reserve table to TableStatusManager	send cluster layout
TableStatusManager↔ TableBuddyManger	TableStatusManager requests help from TableBuddyManager to see if it's possible to merge tables. If it is possible, then the tables are merged. Otherwise, the request is denied.	Request and merge tables
TableClusterManager↔ TableBuddyManager	TableBuddyManager sends a request to TableClusterManager to merge tables to form a new one and move around clusters	merge tables
TableStatusManager↔ ReservationTableManager	TableStatusManager passes data on the number of reserved tables and stats so that ReservationTableManager can store the data.	send reservation data
TableStatusManager↔ TableLoadStatus	TableStatusManager passes data about the request "load" for tables so that TableLoadStatus can determine the future number of required reserved vs walkin tables.	send table load data
TableLoadStatus↔ TableClusterManager	TableLoadStatus sends data to TableClusterManager so it can calculate the number of reserved vs walkin tables in each cluster	send walk/reservation data
ReservationTableManager ↔ TableClusterManager	ReservationTableManager sends the suggested reservation request interval to	send reservation interval

	TableClusterManager	
TableStatusManager↔ TableStatusNotifier	TableStatusManager sends the required notification to StatusNotify so it can be passed to the relevant personnel	send request to notify
TableStatusNotifier↔ CleanNotify	TableStatusNotify forms a CleanNotify to be sent out to the busboy	send clean notify
TableStatusNotifier↔ ReadyNotify	TableStatusNotify forms a ReadyNotify to be send to the waiter to alert his/her table is ready	send ready notify
CleanRequest↔ TableStatusManager	CleanRequest tells TableStatusManager to mark table as "needs cleaning"	send clean request
WalkinRequest↔ TableManager	WalkInRequest tells TableStatusManager to find available walkin table and mark it as in use	send walkin request
SetCleanedRequest↔ TableStatusManager	SetCleanedRequest sent by busboy to mark a table as available after it has been cleaned	send set-cleaned request

Table 1.20

Archiving/Logging Associations:

Concept pair	Association description	Association name
BillPayer↔Archive	Paid bills/orders are archived in the Archive database.	archiving
BillPayer↔Bookkeeper	The daily bills are logged by the Bookkeeper	logging
Archive↔SystemStats	Utilizes data from Archive to form stats for a request period	statistics forming
UpdateTime↔Bookkeeper	Logger uses UpdateTime to get consistent time stamps	consistent logging
UpdateTime↔Archive	Archive uses UpdateTime to get consistent time stamps for archiving	consistent archiving

Table 1.21

Order Management Associations:

Concept pair	Association description	Association name
PlaceOrder ↔ OrderQueue	PlaceOrder retrieves the orders from the waiter which get pushed into the order queue.	Order request
StageManager ↔ OrderQueue	StageManager provides the data about the “meal stage” of each table so that OrderQueue can fairly distribute orders so that no table gets all of eats meal "stages" before another.	Table Staging
PriorityCalculator ↔ OrderQueue	PriorityCalculator utilizes data to calculate the meal priorities in the queue, which is utilized by QrderQueue to prioritize orders.	Priority Managing
OrderQueue ↔ OrderReady	A full order that has been pushed out of the order queue is marked as done.	Order Complete
OrderQueue ↔ OrderItemReady	An individual order item that has been pushed out of the order queue is marked as done.	Order Item Complete
OrderQueue ↔ UpdateCookTime	The cook time is updated through UpdateCookTime each time an order item is completed.	Cook Time Container
StageManager ↔ MealStageTracker	StageManager provides the meal stage status for each table to MealStageTracker which provides an interface for viewers to know what meal stage a table is at.	Stage viewing
FoodReady↔ OrderQueue	Food that has been pushed out of the order queue is marked cooking and waits to be marked as done. FoodDone notifies the queue to mark the food as done cooking.	send food done
FoodPollRequest↔ Orderqueue	The FoodPollRequest (per waiter) knows all the stages of a waiter's order and continuously polls to ask if food is ready for food related to a given stage from the OrderQueue.	poll food done
FoodPollRequest↔	FoodPollRequest sends an OrderReady	send order ready

OrderReady	notification to the waiter when all the food related to one of his order's stages is ready.	
PlaceOrder↔ FoodPollRequest	FoodPollRequest gets the PlaceOrder request so they can poll the order queue for the food in the proper order.	add order

Table 1.22

Inventory Associations:

Concept pair	Association description	Association name
IngredientTable↔InventoryTracker	IngredientTable sends a list of ingredients for InventoryTracker to subtract from the inventory	inventory request
InventoryTracker↔InventoryContainer	InventoryTracker tells InventoryContainer what items to subtract out. InventoryTracker pulls items from each item's inventory and forms a list.	subtract and fetch items
InventoryTracker↔Items List	InventoryTracker forms a list from the items it pulled from the InventoryContainer and sends the list with which items to be used for an order item.	send items list
MealRequest↔Ingredient Table	The chef sends a MealRequest for the meal he is making to the IngredientTable who maps the meal item to its ingredients	send meal request
InventoryTracker↔ItemLowNotifcation	When InventoryTracker sees an item is below its "low" threshold it sends a ItemLowNotification to the manager saying which items to buy more of.	notify low items
IngredientTable↔MealPopularity	MealPopularity keeps a count of which items are requested to IngredientTable to keep track of most popular items.	track meal popularity
InventoryTracker↔IngredientPopularity	IngredientPopularity tracks which items are requested to InventoryTracker and keeps track of most popular ingredients.	track ingredient popularity
MealSuggester↔MealPopularity	MealSuggester checks MealPopularity to see which meals are popular and profitable to suggest to advise on which	Reads meal data

	items to remove and keep from the menu.	
MealSuggester↔IngredientPopularit	MealSuggester checks MealPopularity to see which ingredient are popular to form an optimal meal plan for the next week.	Read ingredients data

Table 1.23

Attribute Definitions:

Concept	Attributes	Attribute Descriptions
Authenticator	-Role	Manager, Waiter, Chef, Host, etc.
ReservationTableManager	-NumOfReservationTables	Number of reservation tables available
TableLoadStatus	-TableSizeRequests	Number of requests for the respective table size
TableClusterManager	-ReservationNum -WalkInNum	-Number of reservations vs walk ins for a specific table
TableBuddyManager	-NewTableSize	-Adds tables of smaller size together to create a new table size
TableStatusManager	-TableStatus	-Holds the status of the table, such as clean, reserve, in-use, etc.
TableStatusNotifier	- StatusToSend	-Holds the status of a table
FloorPlanPage	- Tables	-Holds table statuses, size, and ID
ReservationPage	- Name - Email - Size - Date	-Holds the name, and email of the person who wants the reservation and the size and date of reservation
CleanRequest	-TableID	-Holds the ID of the table that the busboy needs to clean.
WalkinRequest	-TableID -CustomerID	-Holds the ID of the table that will be marked as walk in and

		the customer occupyng it
CleanNotify	-TableDirty	-Is true if the respective table is dirty, and if true sends a notification to the busboy
SetCleanedRequest	-TableID	-Holds the ID of the table that the busboy has cleaned
ReadyNotify	-TableID	-Holds the ID of the table that the waiter needs to serve
BookKeeper	-ReceiptID	-Unique identification number for each receipt generated
SystemStats	-MealsSold -Sales -Profit	-Total numbers of meals sold in a given time period -Total number of sales in a given time period -Total profit in a given time period
Archive	-InventoryCount	-Keeps all the counts in the inventory per day
UpdateTime	-SystemUpdateTime	-Time the system was last updated
BillPayer	-PriceOfBill	-Holds the price of the bill the customer needs to pay
PlaceOrder	-CustomerOrder	-List of items customer ordered
FoodPollRequest	-OrderID -OrderStatus	-Uses orderID to check if the OrderStatus
OrderQueue	-OrderQueue	-Queue of all the orders coming in to the kitchen
FoodReady	-IsFoodReady	-Holds the value received from FoodChecker about the status of the food
MealRequest	-MealName	-Container for the recipe of

		the meal requested
OrderReady	-IsOrderReady	-Updates the value of the order in the database to be completed
UpdateCookTime	-NewCookTime	-The cook time to update the cooktime in the database with
MenuItemTracker	-NumTimesSold	-List of times each item has been sold
OrderItemReady	-IsOrderItemReady	-Updated within an order to keep track of an order status
StageManager	-TimeSinceOrder -NumOfMealsPerStage	-How long it has been since the order was placed -Number of meals in each stage
ItemLowNotification	-NotifyLow	-Alerts when certain items are low
PriorityCalculator	-MealType -AverageTime	-Type of meal (appetizers, entrees, desserts) -Average time to make meal based on previous data
ItemsList	-ShowList	-Shows what items will be needed for the meal
FoodChecker	-FoodStatus -OrderID	-Status of the order based on the order identification
MealStageTracker	-MealStage -TableID	-The stage of the meal(appetizers,entrees, dessert) the respective table is
InventoryContainer	-ListIngredients -Prices -ExpirationDates	-Keeps a tabular format of the list of ingredients -List of prices associated with each ingredient -The expiration date of the ingredients
InventoryTracker	-LowThreshold	-The low thresholds of each ingredient

IngredientPopularity	-TopIngredients	-List of each ingredient used and how often it was used and sorted from high to low
IngredientTable	-RecipeBook	-List of ingredients used for each recipe
MealPopularity	-TopMeals	-List of how many times a meal was ordered and sorted from high to low
MenuSuggester	-SuggestedMenuItems	-List of meals being suggested based on collected data

Table 1.24

Old Traceability Matrix

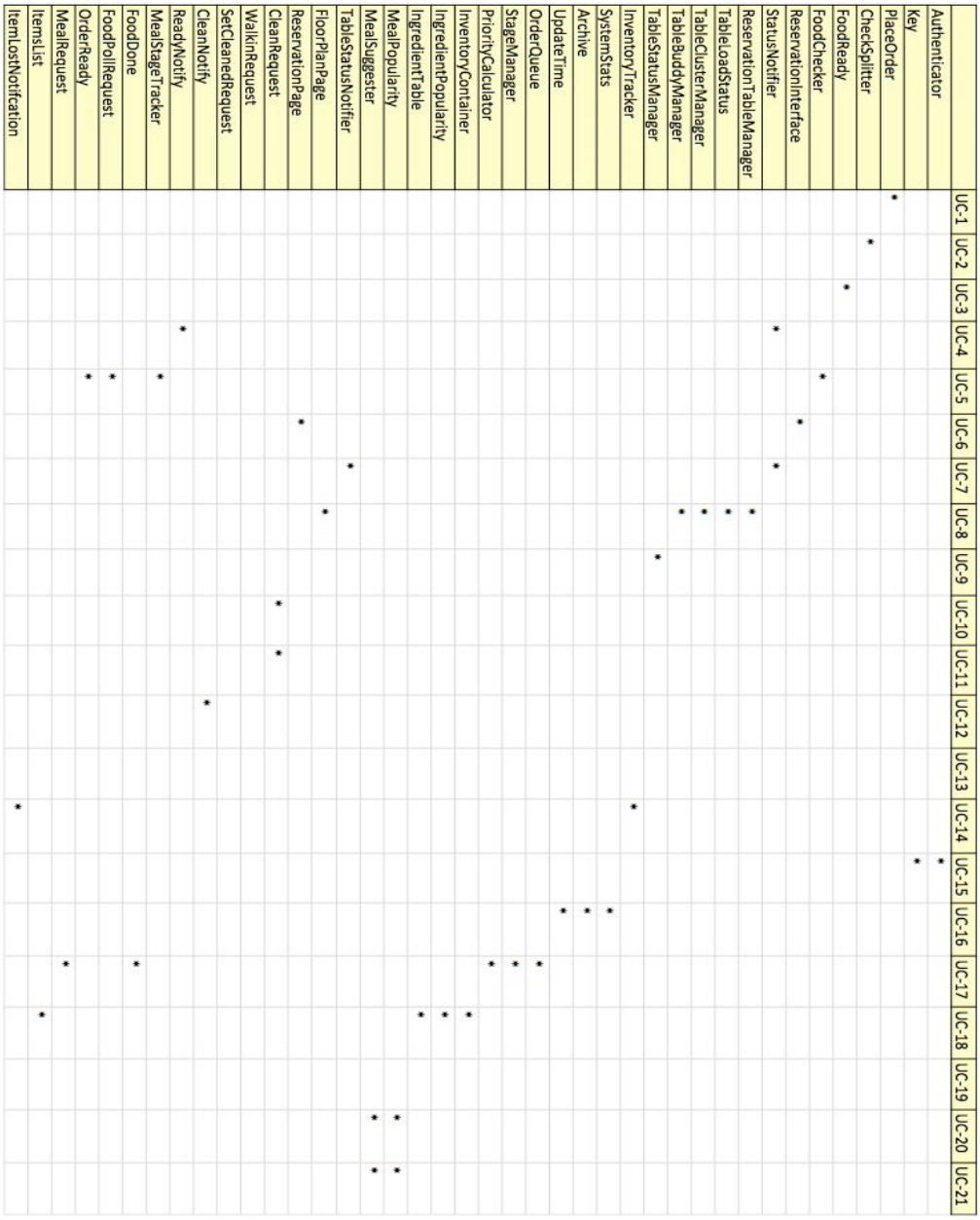


Figure 1.11

New Traceability Matrix

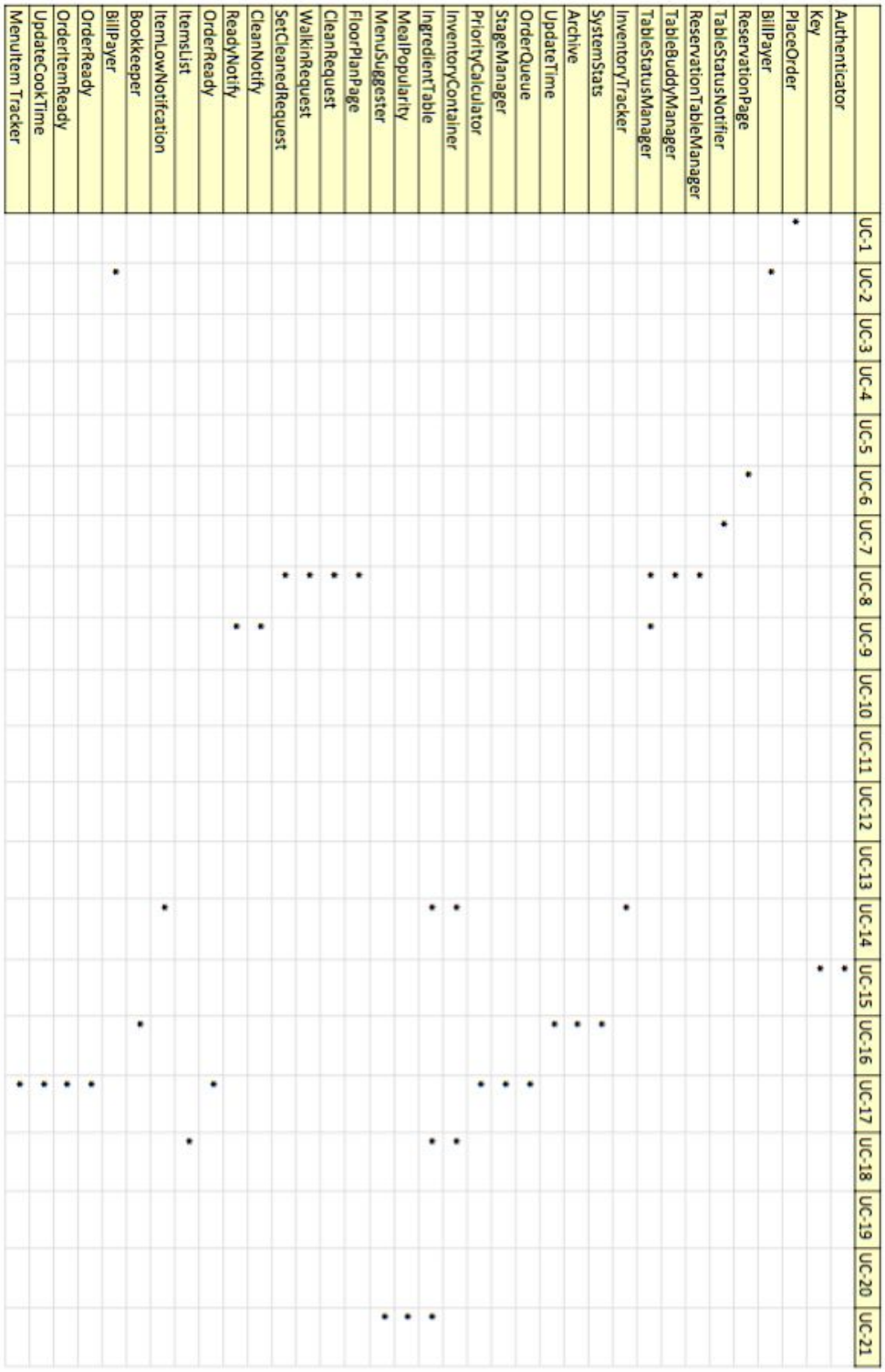


Figure 1.12

The traceability matrix above shows how each domain concept is connected to a use case. The domain concept might partly implement a use case or a domain concept might implement multiple use cases completely.

Domain Model Diagram

Below is the Domain models for the chosen use cases. It displays the interaction with each respective subsystem

Old Domain Model for Floor Plan

::Domain Model for UC- 8: Manage Floor Status, UC-9: Update Table Status, UC-7: Send Table Notification, UC-6: Make Reservation

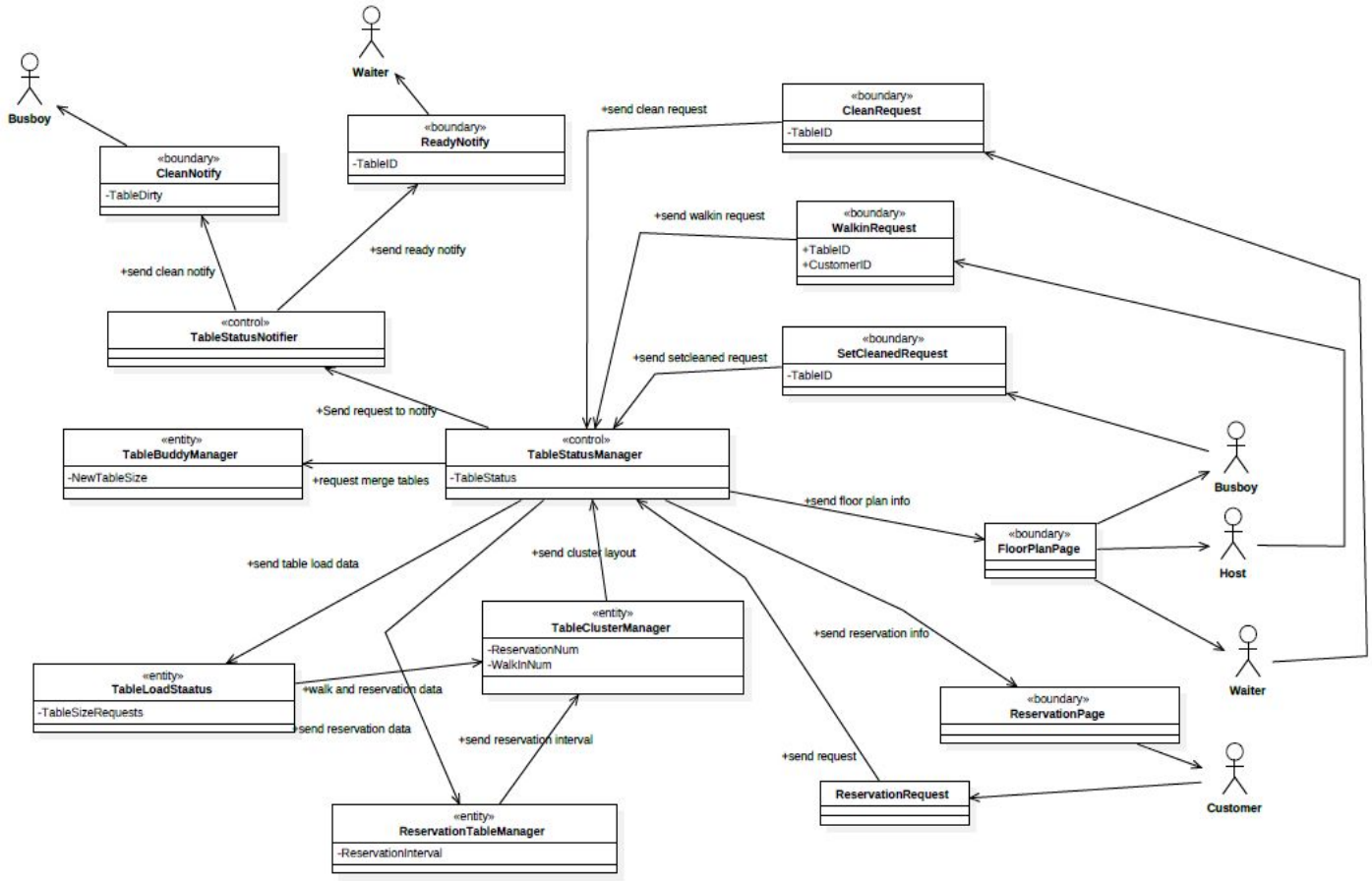


Figure 1.13

New Domain Model for Floor Plan

::Domain Model for UC- 8: Manage Floor Status, UC-9: Update Table Status, UC-7: Send Table Notification
UC-6: Make Reservation

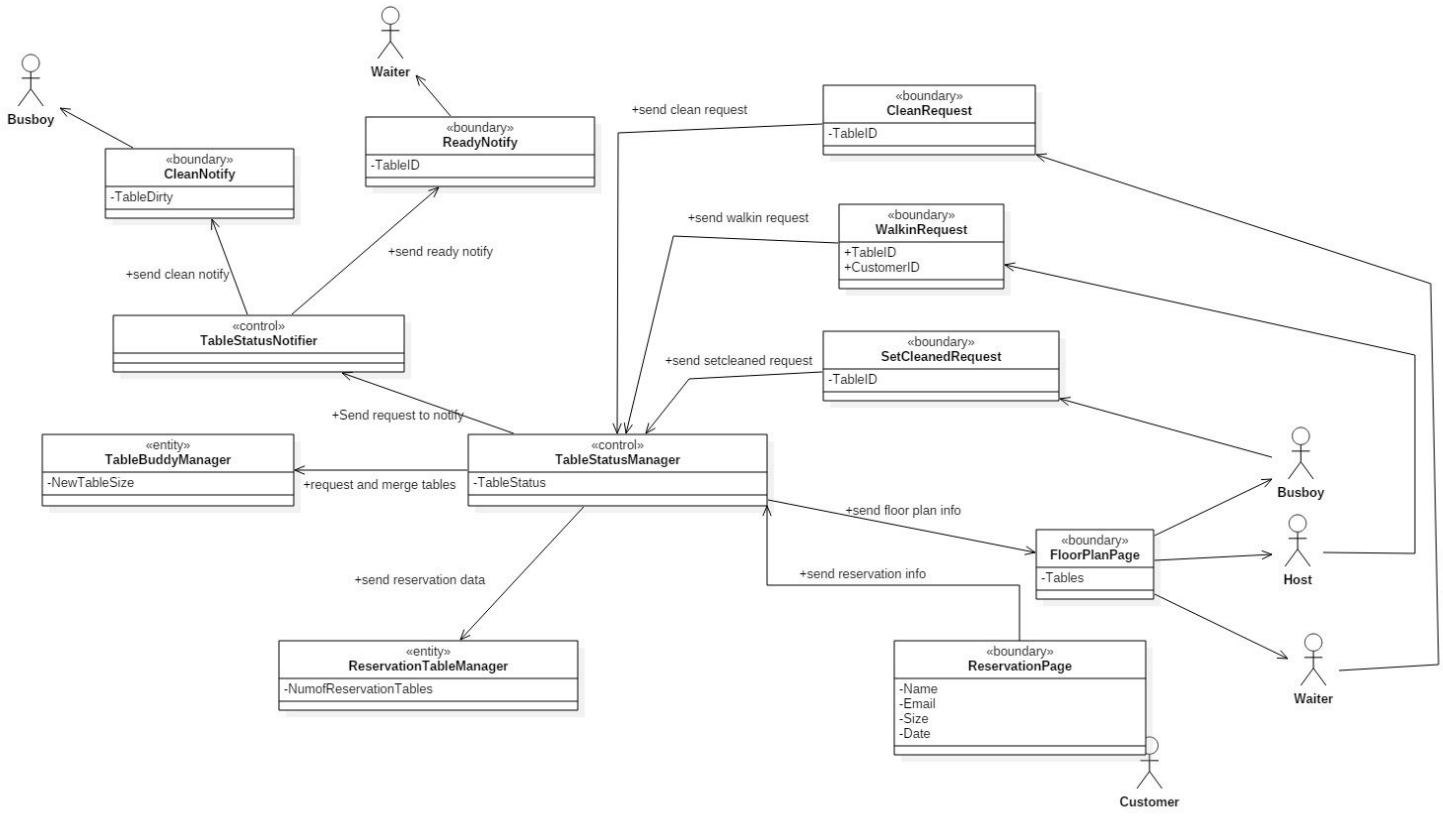


Figure 1.14

The domain model above visually represents the interacting entities associated with the floor plan subsystem. Overall this subsystem deals with the management of seating a customer and relaying information regarding the changes in the status of the tables within the floor plan to the busyboy, host and waiter. This abstraction shows how the different actors participate within this subsystem to get information or update the information within the system. For example, a customer can make a reservation which can be seen by the host, who in turn seats the customer and marks the table as taken after which the waiter can claim that table, serve the customer and then notify the bus boy by marking the table as dirty. Details of the associations between the different concepts can be found in the association tables in this section.

Old Domain Model for Managing Order Queue

::Domain Model for UC-17: Manage Order Queue, UC-1: Place Order, UC-3: Send Food Notification, UC-5: Check Food Status, UC-19: Check Table Meal Status

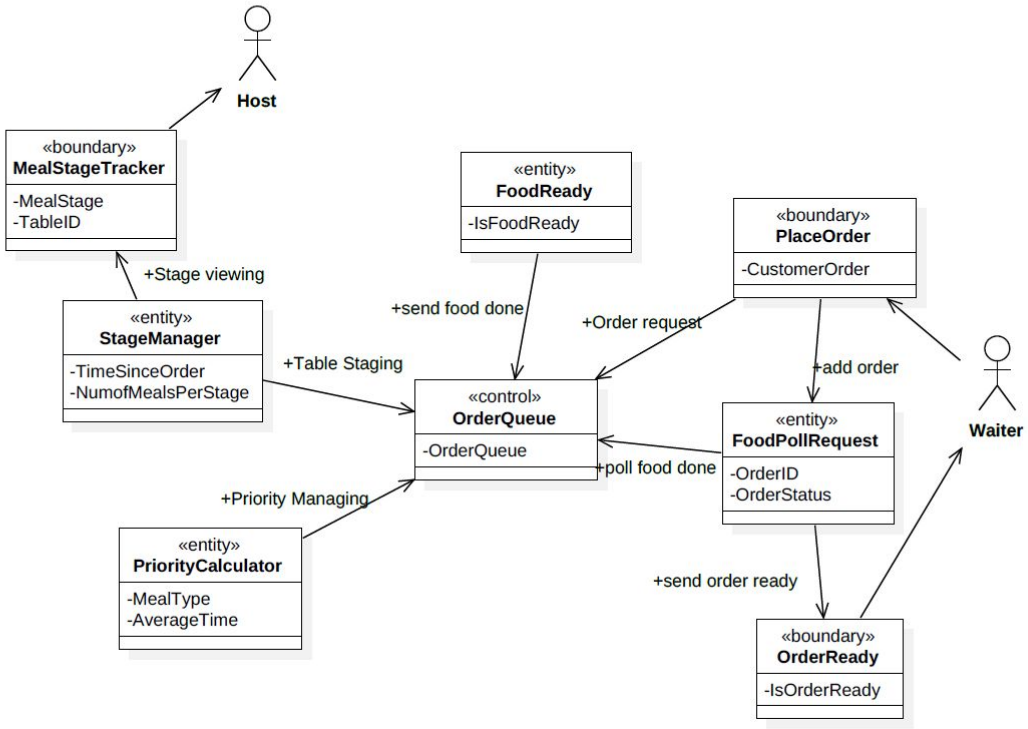


Figure 1.15

New Domain Model for Managing Order Queue

::Domain Model for UC-17: Manage Order Queue, UC-1: Place Order, UC-3: Send Food Notification, UC-5: Check Food Status, UC-19: Check Table Meal Status

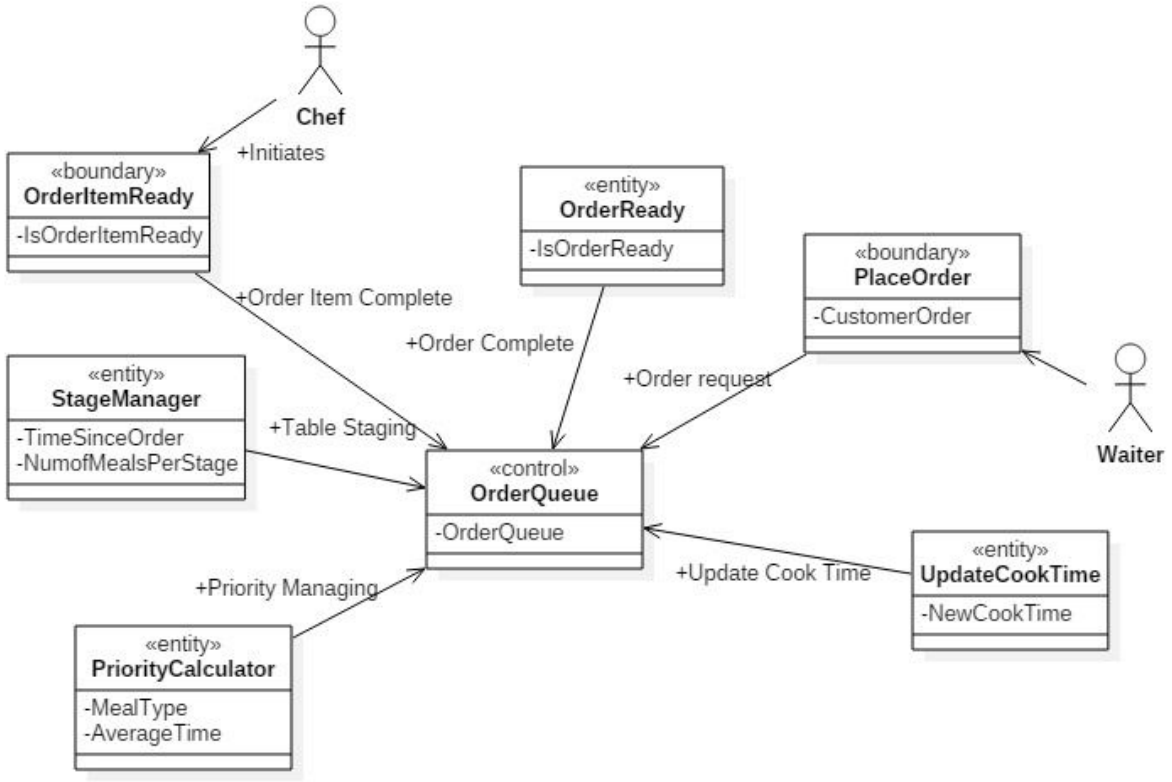


Figure 1.16

The domain model above visually represents the interacting entities associated with the order queue subsystem. Overall this subsystem deals with the aspect of placing an order and making sure that information is relayed to the chef in the kitchen. The chef views the item in the order queue as an itemized list that is determined by the priority calculator. Whenever the chef finishes an item the time it took for him to make the item is recorded and updated within the database. When the chef finishes all items within an order the order is marked as completed and saved.

Old Domain Model of Managing Inventory

::Domain Model for UC-14: Notify Manager of Inventory Status, UC-18: Track Raw Materials, UC-20: Suggest Use of Materials, UC-21: Generate Menu Suggestions

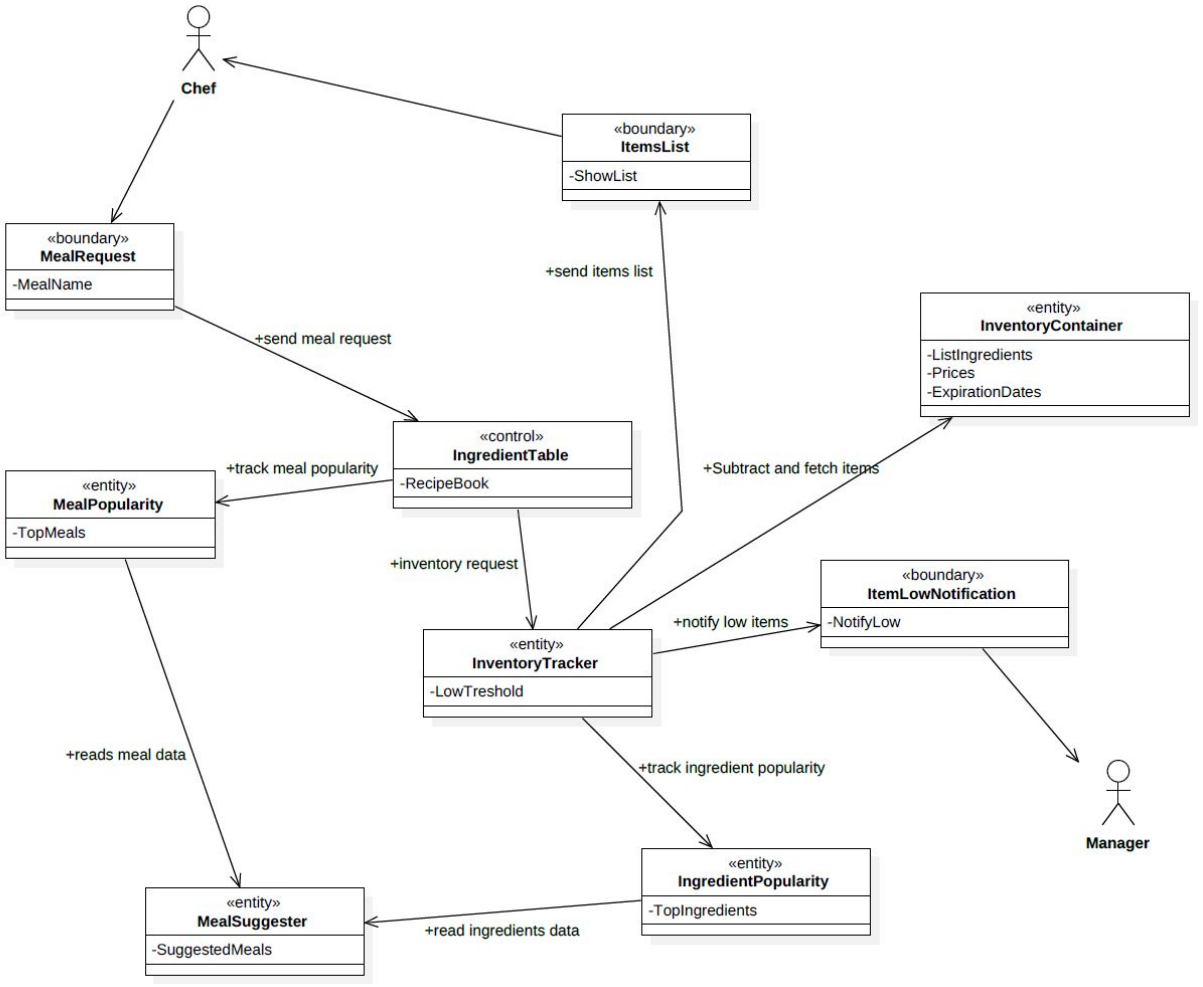


Figure 1.17

New Domain Model of Managing Inventory

::Domain Model for UC-14: Notify Manager of Inventory Status, UC-18: Track Raw Materials, UC-20: Suggest Use of Materials, UC-21: Generate Menu Suggestions

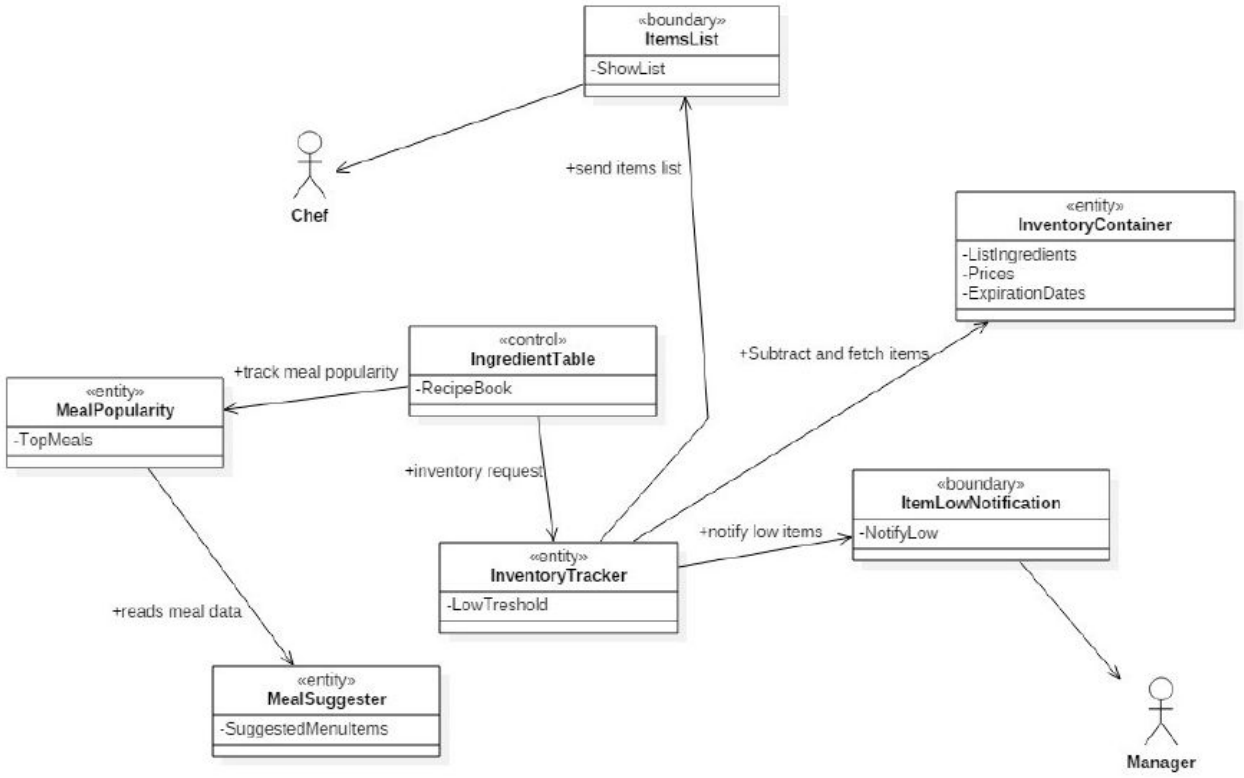


Figure 1.18

The domain model above visually represents the interacting entities associated with the inventory subsystem. Overall this subsystem deals with the aspect of keeping track of the raw materials being used in the restaurant and updating the manager on the corresponding changes. Each time the chef completes an item, the item list automatically updates itself and subtracts the corresponding ingredients from the inventory container. This information is updated each time an order item is completed and the manager gets a live view of the updated quantity of each inventory item. Whenever an item is running low the manager would get a notification indicating that he/she has to replenish his stock. During this process the count of how often each item is sold is also kept track of and used to suggest meal based on popularity. The details on how this is done can be found in this section...

Old Domain Model for Managing Archive and Statistics

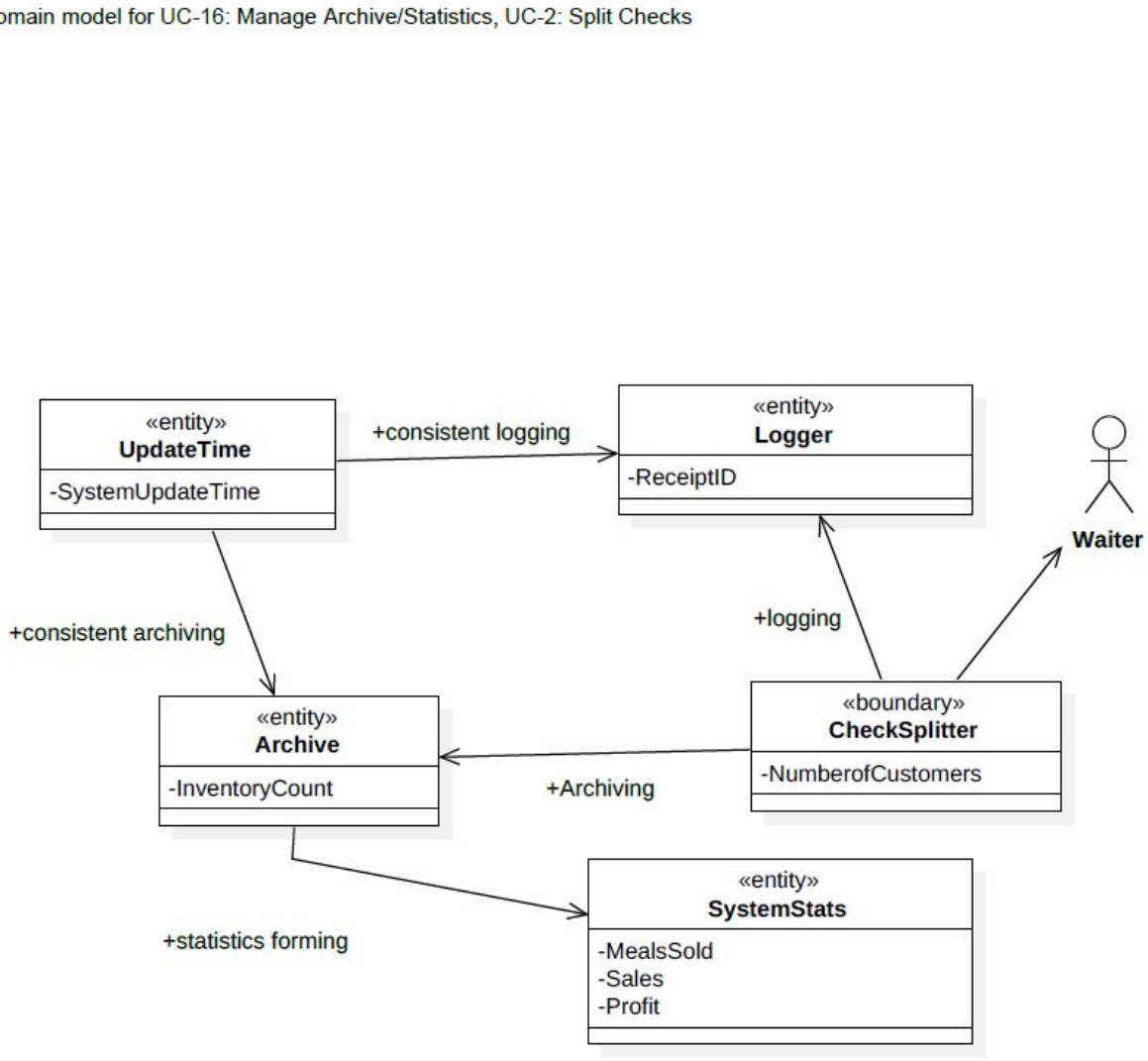


Figure 1.19

New Domain Model for Managing Archive and Statistics

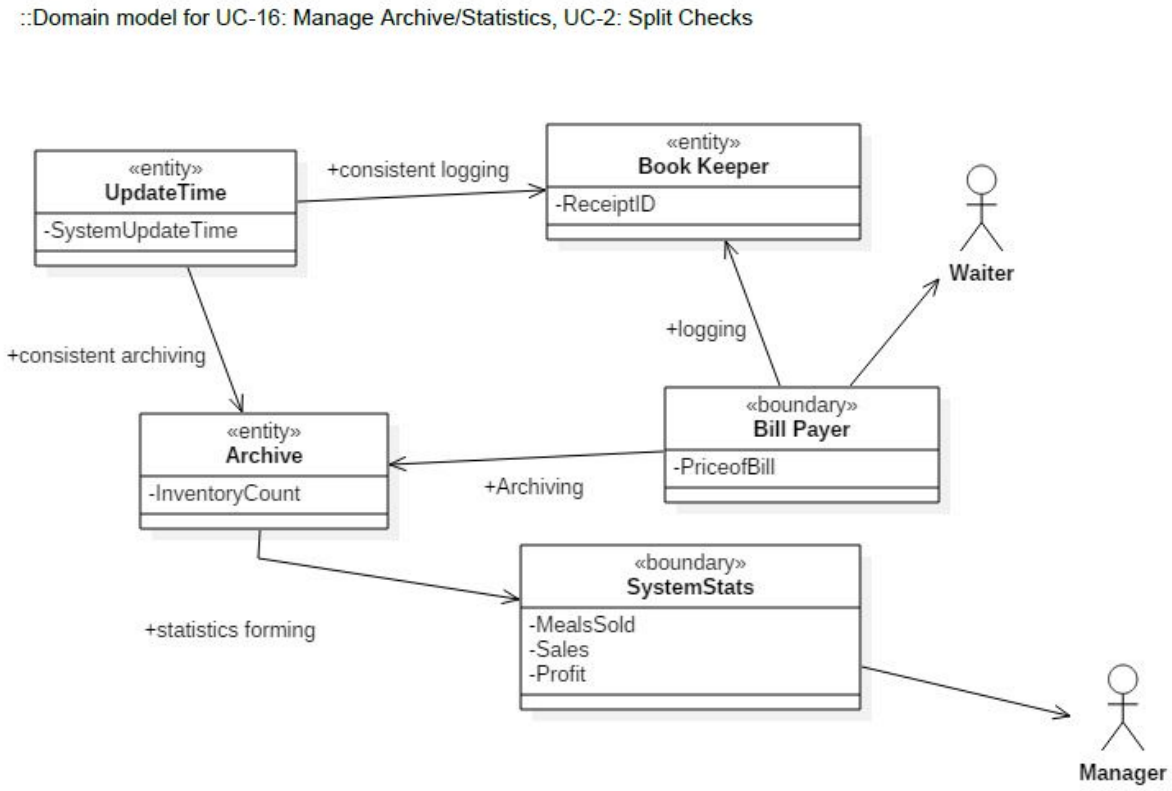


Figure 1.20

The domain model above visually represents the interacting entities associated with the Archive and Statistics subsystem. Overall this subsystem deals with the aspect of keeping track of all the data associated with the different orders of the restaurant. All the data is stored within the database and used in the calculation of the system statistics. The way these statistics are calculated is shown within this section. The statistics calculated will be shown to the manager and can be used to by the manager to make wise business decisions.

System Operation Contracts

Operation	**Manage Floor Status**
Preconditions	- User is authenticated by the authenticator using his key
Postconditions	- FloorPlanPage shows the floor plan for all tables and shows the status of each table - Customer can make a reservation using the reservation page.

Table 1.25

Operation	**Manage Order Queue**
Preconditions	- Customer's valid order is placed through PlaceOrder - OrderQueue and PriorityCalculator is initialized. (MealType and AverageTime are either calculated from previous collected data or initial input).
Postconditions	- The status of an orderItem and an Order is updated to completed when a chef completes an order. - The chef can complete an order or add time via the OrderQueue interface. - OrderQueue manages priority and cooking time for each order based on calculations by PriorityCalculator

Table 1.26

Operation	**Manage Inventory**
Preconditions	- User is authenticated by the authenticator using his key - The ItemsList and IngredientTable are correctly configured.
Postconditions	- The ItemLowNotification is sent when an item from the ItemsList reaches the LowThreshold - MealSuggester shows a list a suggested menu items. - InventoryContainer holds information for all items in the inventory.

Table 1.27

Operation	**Manage Archive/Statistics**
Preconditions	- Manager is authenticated by the authenticator using his key - BookKeeper has recorded the daily sales of the restaurant(at least one sale) - Archive contains past data of the restaurant
Postconditions	- SystemStats analyzes the data and displays statistics.

Table 1.28

Mathematical Model

Suggesting Meal Plan

One of the primary automatic data processing algorithms implemented in our system is the Menu Suggestion algorithm. The menu suggestion algorithm is designed to help the business manager make beneficial decisions by suggesting items on the menu that are both popular with customers and profitable to the restaurant. Therefore, the business manager can adjust the restaurant menu to remove items that are not popular or profitable, and keep items that make profit and customers content. For instance, an item might be making the manager a large chunk of profit in one week alone. However, that item may not sell often over a longer time period such as a month. Trends such as these need to be identified, so that the manager can make the best judgement on how to operate the restaurant.

Overall, the meal suggestion algorithm remained the same as mentioned in the past reports. However, a few changes were made to make the algorithm feasible and operate as expected in the system environment. To clarify, system environment represents the structure of our database collections and the way the code is written overall. The algorithm needs to be able to adapt to its surroundings in order to function properly. Hence, a pseudocode version of our algorithm is provided below, highlighting the important calculations. All changes made to the algorithm will be described at the end of the **Suggesting Meal Plan** section.

How does the algorithm work?

First off, the algorithm (Figure 1.29) is explained assuming that there is existing data in the database for at least the length of one quarter of a year. Specifically, the existing data are customer orders placed by waiters. Moving on, three separate time periods are allotted; *last week* (7 days), *last month* (30 days), *last quarter* (~92 days). For the three time periods, a set of existing orders is pulled from the database as previously mentioned. Next, profit, revenue, and cost are separately calculated for each item on the menu based on the orders in each time period. Therefore, each menu item will have three sets of cost, revenue, and profit (for *last week*, *last month*, and *last quarter*). Next, each set is sorted by profit in descending order. At this point, item profits are calculated for each item, but they're values are not yet relevant in terms of popularity. To determine relevant items in the sorted lists, the **top five profit makers** are taken from **each time period**. Then a popularity is assigned to each item in the menu based on which top-five time period(s) that menu item occurs in, if any. Table 1.29 shows the different possible combinations and their respective popularity levels. For instance, if an item falls within the *Exclusive* popularity, the manager is making a good profit and customers are satisfied with that item. Therefore, the algorithm takes both profitability and customer satisfaction into account, making it a unique feature to optimize and improve business decisions.

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

}

FOR EACH : MENU_ITEMS
{
	CHECK(ALL SORTED_ITEMS_BY_PROFIT)
	{
		Find the combinations of menu item occurrences in the first five items of
		EACH SORTED_ITEMS_BY_PROFIT

		Assign popularity to a menu item based on Table 1.29
	}
}
```

Figure 1.29 - A pseudocode version of the meal suggestion algorithm.

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

Table 1.29 - A list of combinations mapped to popularity levels explained in the text.

The primary changes made to the meal suggestion algorithm is that only the top five profit makers are considered in choosing a rank other than low. If each item was considered in the

calculation, then the characteristic bond between customer satisfaction and profitability would be lost. Items that are not too profitable would be considered, therefore, making the data less valuable.

Manage Order Queue

Node Priority:
The Order Queue is a priority based system that contains 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
```

Figure 1.30

Given the time constraint we were only able to implement a partial section of the algorithm. However, we implemented a sufficient amount of the algorithm to ensure that the order queue is still efficient. A full implementation of the algorithm would only increase the efficiency a little more. Therefore, the tradeoff of not completing the algorithm was not detrimental for our project because we were able to focus on other unique features.

Changes from past reports

In comparison to previous reports we had to add multiple domain concepts that were not there initially but at the same time we had to remove many of the domain concepts that we decided not to use. This happens to be the case because over the duration of the project we have evolved our idea of what the project should be like in the end. Not implementing certain features while adding some modified features lead to a new set of domain concepts, associations, and attributes. To clarify how our project has evolved for this section we highlighted any deletions or insertions using the below legend.

DELETIONS
INSERTIONS

Deletions - things we have removed since previous reports
Insertions - things we have added since previous reports

Interaction Diagrams

[3*]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.

OLD UC-8 Floor Status

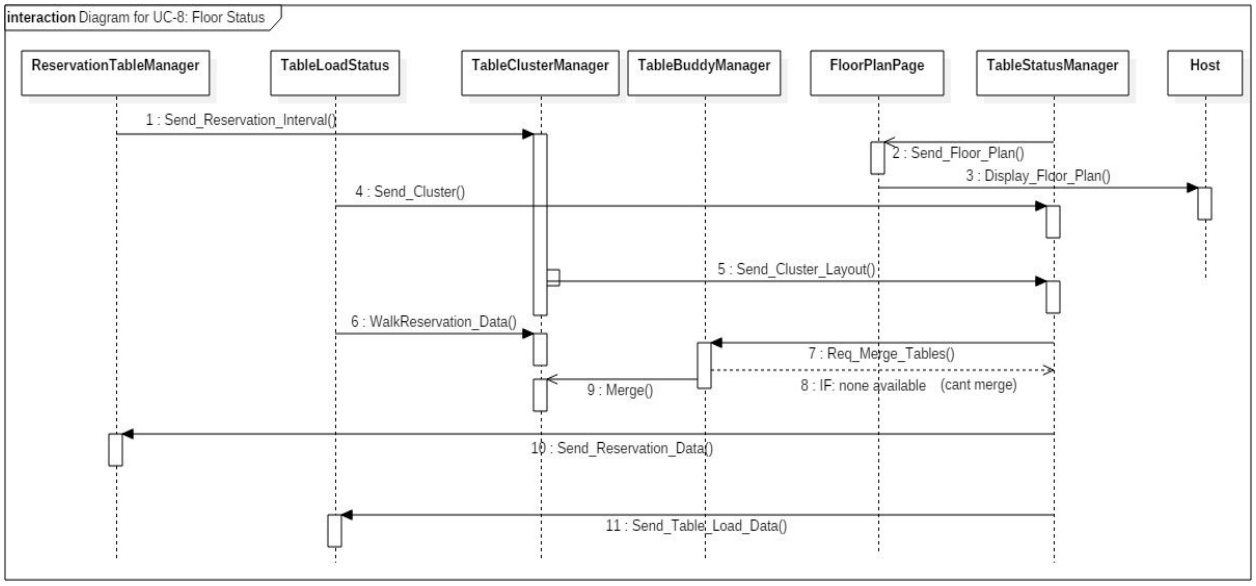


Figure 1.31

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.

NEW UC-8 Floor Status

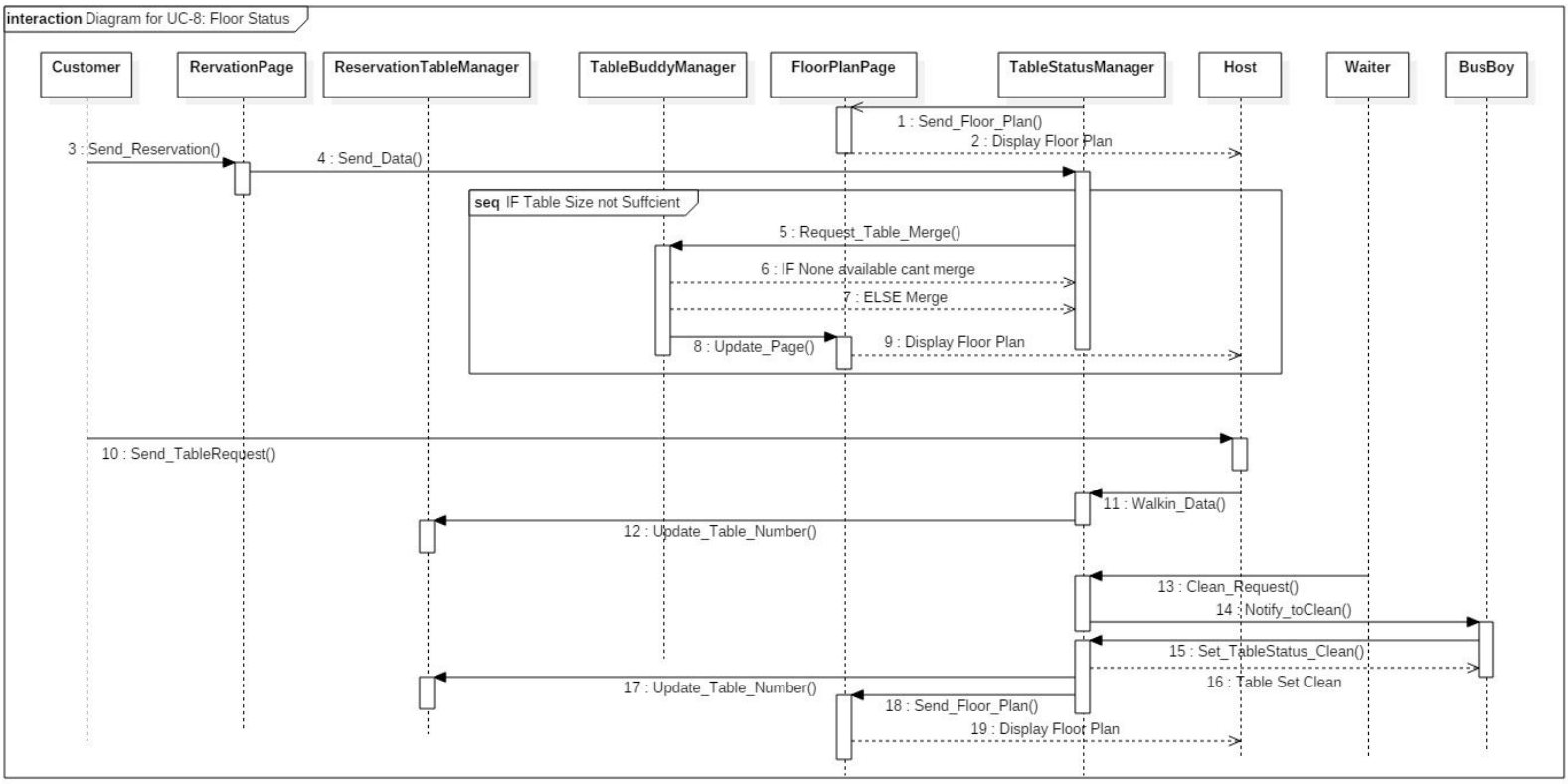


Figure 1.32

The Floor Status Use Case interaction diagram shows what occurs whenever there is a change to a table. In this interaction diagram we see it starts off with the host viewing a the Floor Plan before any interaction occurs. Next, we see that a customer sent a reservation through the reservation page. This reservation data is then sent to the TableStatusManager, which determines whether the reservation can be made or not. If the reservation is possible, the table will automatically merge if needed and be displayed onto the Floor Plan. After this occurs, we can see that another customer walks into the restaurant asking the host to be seated. The host then takes the customer's request and submits the data to the TableStatusManager, which will update the number of available reservations to the ReservationTableManager. After the customer is seated and done eating, the waiter requests the table to be clean. The Busboy is notified from this request and sets the table as clean after cleaning the table. Once the table is clean, the table on the Floor Plan is updated as well. For this interaction diagram, we followed the **Pub-Sub Pattern**. In the interaction diagram, all of the “Table manager” objects would behave as the publisher since they know where the events occurred from. The subscribers would be the Host, Waiter, Busboy, and the FloorPlanPage. This is because they are the ones that are interested in the events and receives the information.

OLD UC-17 Manage Order Queue

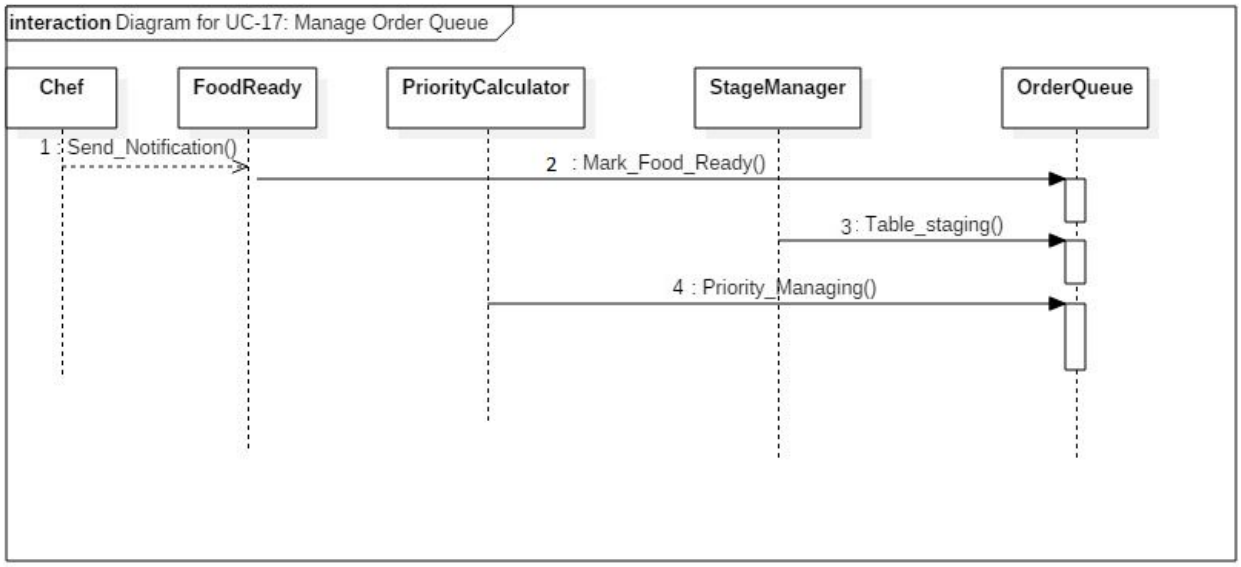


Figure 1.33

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.

NEW UC-17 Manage Order Queue

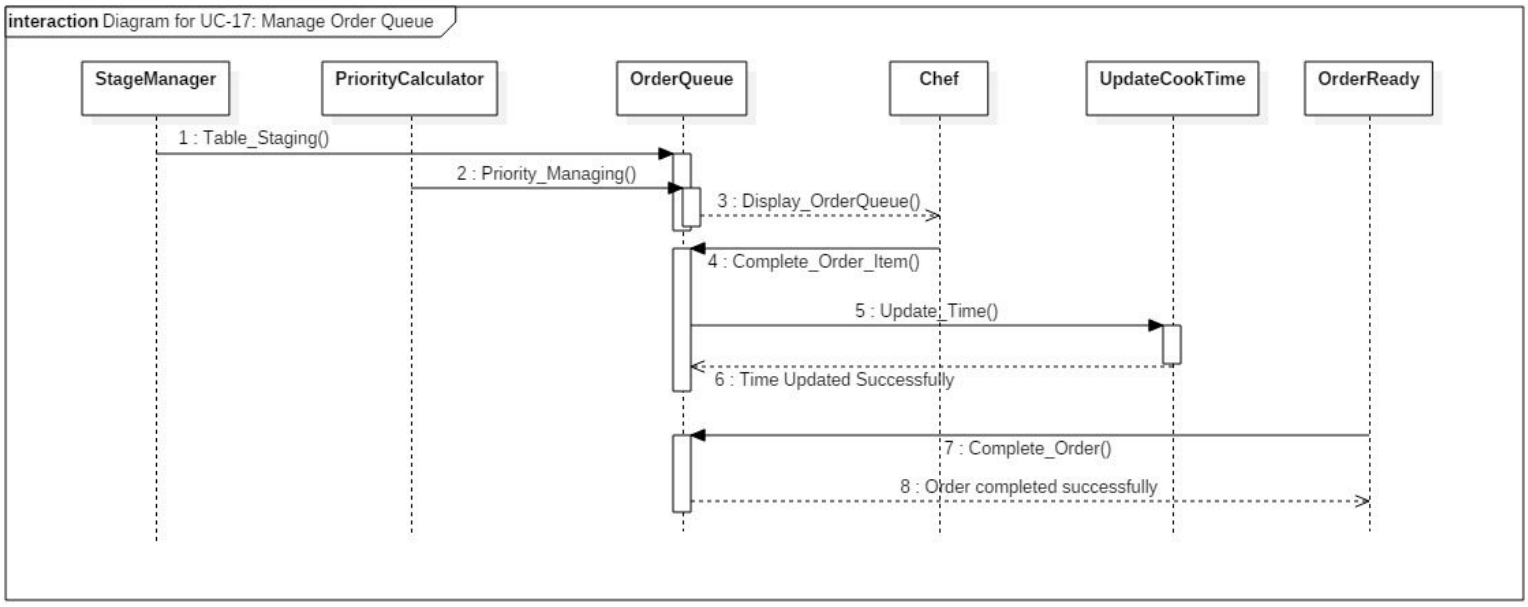


Figure 1.34

The Manage Order Queue use case comes into play after the waiter places an order. 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. A chef can complete an order item which will take the time spent making the order item and use that to update the cookTime for an item. Once all the OrderItems in an Order have been completed, OrderReady will mark the Order as ready. For this interaction diagram, we followed the **Delegation Pattern**. In the interaction diagram, the OrderQueue is the that collects objects and works to provide service for the chef.

OLD UC-16 Manage/Archive Statistics

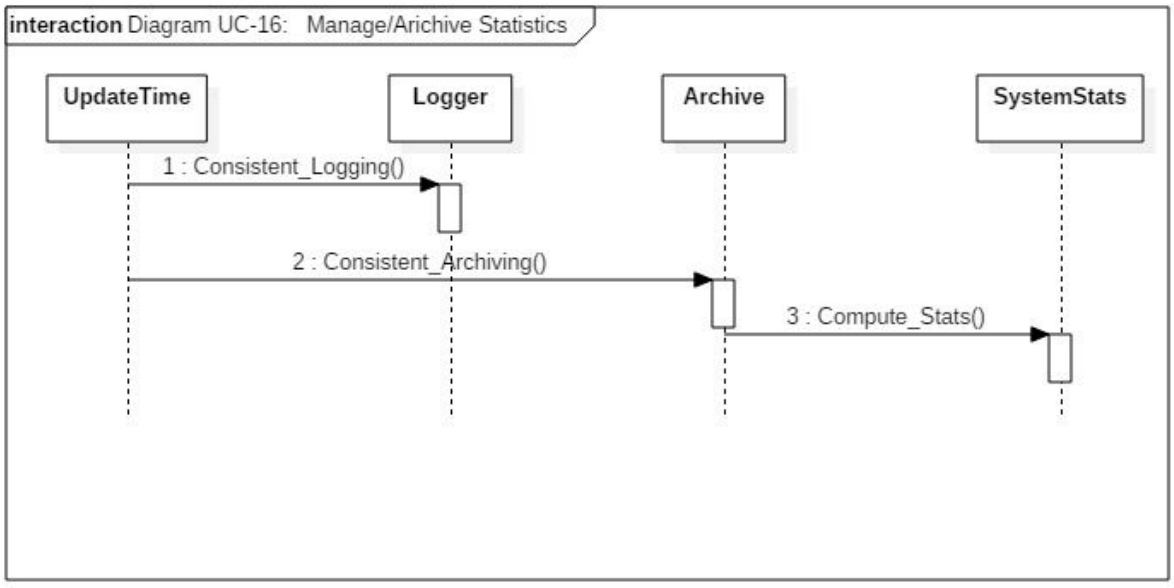


Figure 1.35

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.

NEW UC-16 Manage/Archive Statistics

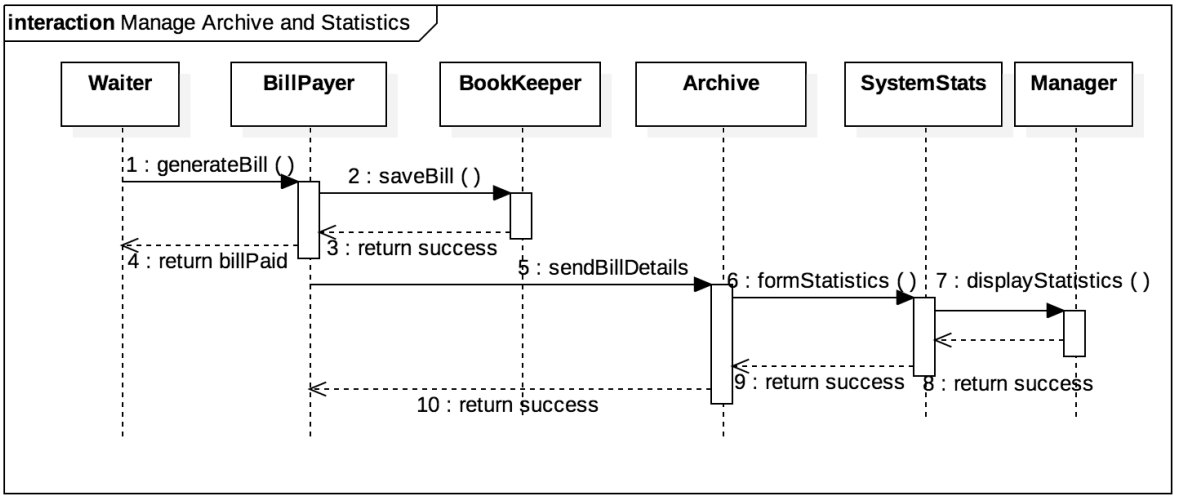


Figure 1.36

The Manage/Archive Statistics Use Case describes the interactions that take place whenever there is a transaction. The system takes the details of transactions and extracts information to form statistics. For example SystemStats can take all the food items of the completed orders, from the Archive, and calculate the profit for each item. These statistics can now be displayed to the manager, allowing him to make optimal decisions for his restaurant. In addition to profit, the system will also be calculating statistics such as, number of items sold, most popular dish, most profitable dish, number of reservations, etc then can further be used to make wise business decisions. For this diagram we followed the design of the **pub-sub pattern**. With this design, there is high cohesion and low coupling as well. We split up the the communication responsibilities and made sure that there wasn't a single object doing all of the work. The subscribers in this case are manager and waiter, and the rest represent the publishers.

NEW UC-18 Track Raw Materials

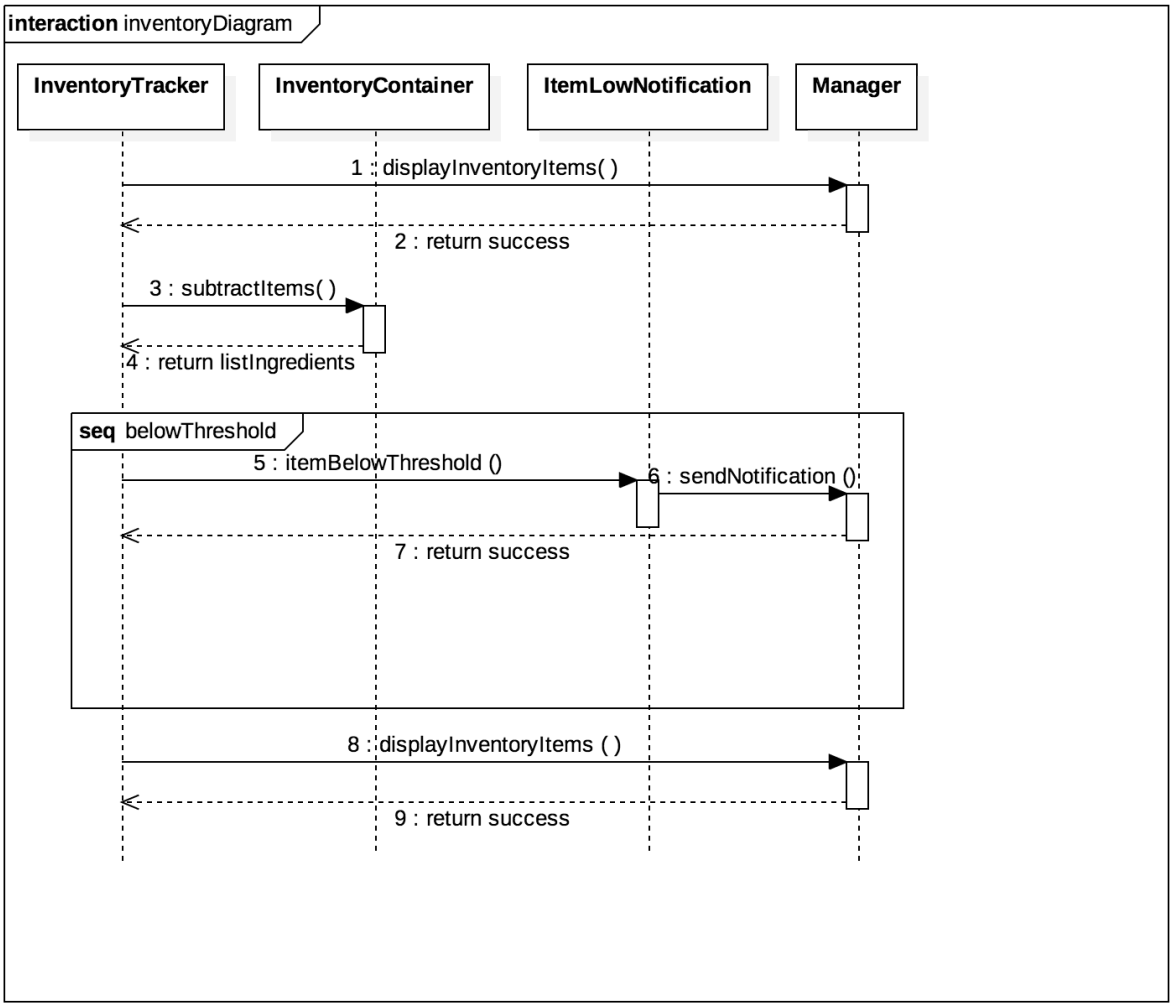


Figure 1.37

The Track Raw Materials use case describes the interactions in the inventory subsystem. The InventoryContainer holds all the details for each inventory item. Every time an item is used, the InventoryTracker automatically subtracts the respective amount from the InventoryContainer. If an item goes below the predetermined threshold value, ItemLowNotification will update the Manager to let him know that the item is running low and should be replenished soon. As the item counts get updated, the display to the manager is also updated so he is able to see the information as it is changing. For this diagram we followed the **Command Pattern** design, where the InventoryTracker is the "command". It can execute an action and knows who the receiver is. It can also undo an action if it is reversible and needed.

NEW UC-21 Menu Suggestions

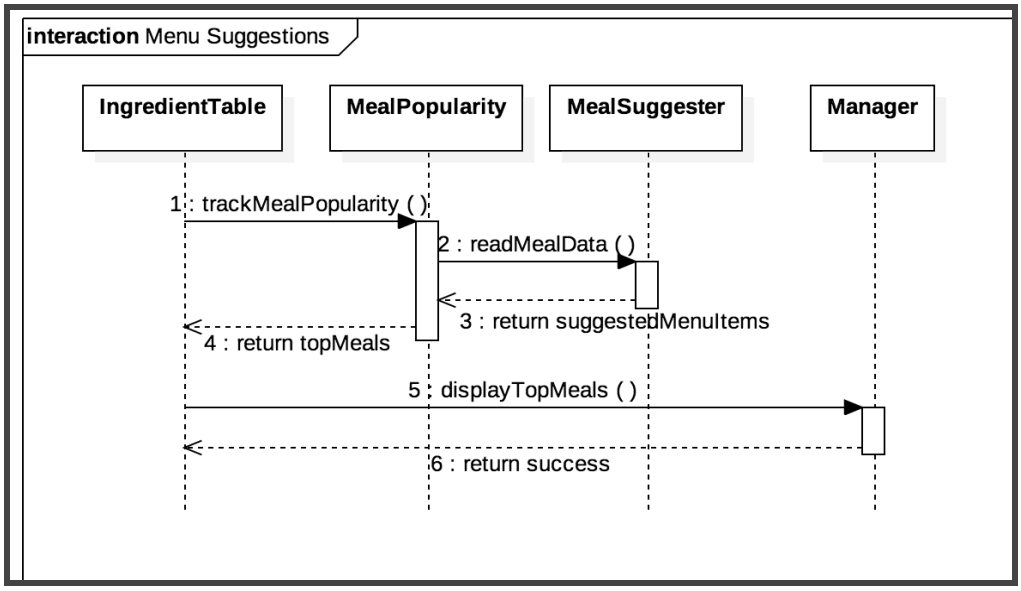


Figure 1.38

The Menu Suggestions use case describes the background work done by the system to figure out the popular and profitable items. Menu Suggestions are calculated using an algorithm described in figure 1.49. In the interaction diagram, the IngredientTable sends a request to MealPopularity to track the meals. Which then reads the data from Meal Suggester, generated by the algorithm. The data is then returned, so it it can be properly displayed for the Manager to allow him to make decisions to optimize his restaurant. For this diagram we used the **pub-sub pattern** and **delegation pattern**. The delegator is the IngredientsTable who takes objects from the MealPupularity and MealSuggester and provides a service. The receiver of this service would be the Manager who is also a subscriber.

NEW UC-1 Place Order

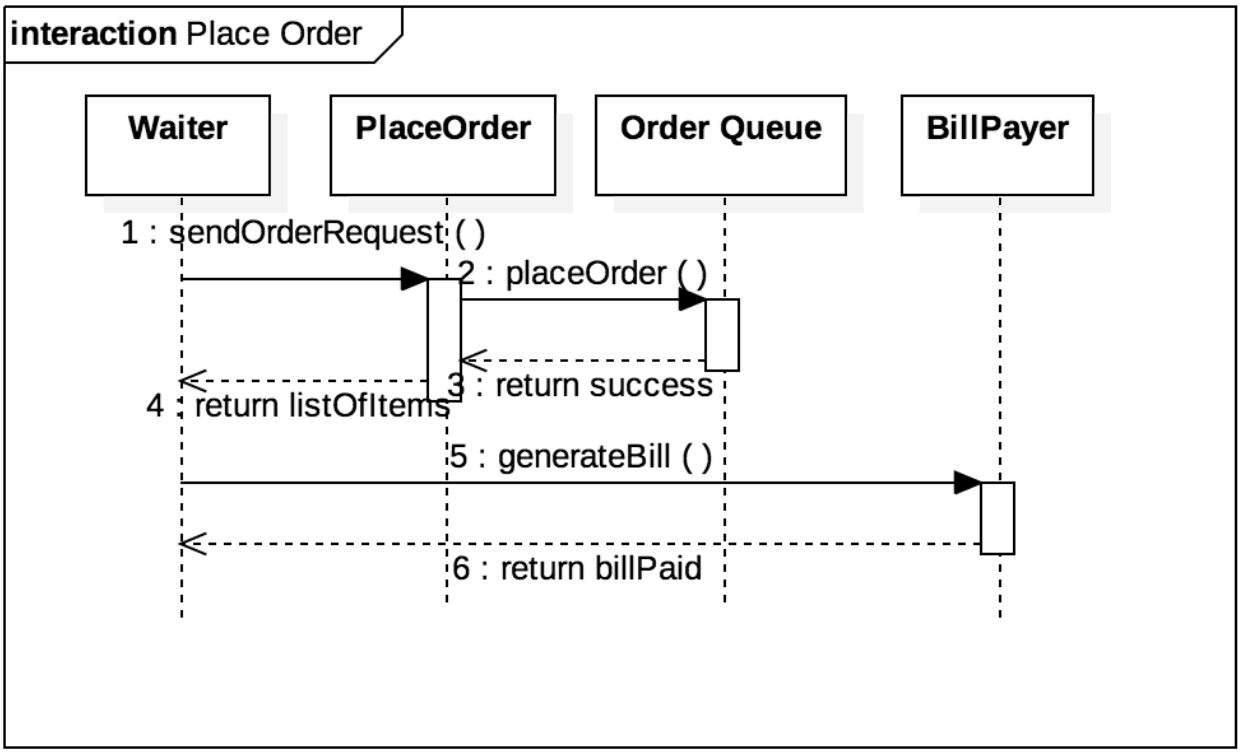


Figure 1.39

The Place Order use case describes the interaction of submitting and paying for an order. The waiter sends an order request to the place order system, which validates the order before it is submitted to the order queue. Once the order is complete, the waiter is able to request BillPayer to generate the Bill and once it is paid, it returns to the waiter saying it has been paid. For this diagram we used the **pub-sub pattern** and **delegation pattern**. The delegators and the publishers are PlaceOrder and OrderQueue. The subscribers are the waiter and the Billpayer since they process the received events.

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.

Changes from Previous Interaction Diagrams

As we continued working on our project, we focused on the non trivial features along with taking the feedback from previous reports into account. For this report, we redesigned our interaction diagrams based on the updated domain models. In our previous diagrams, we did not provide feedback for some of our function calls that required it, so we took that into account and made sure that the diagrams were giving feedback. We made sure that our diagrams were self-explanatory and therefore anyone with no knowledge of our system should be able to understand how the interactions of each use case works. Beneath all our diagrams we made sure to explain in further detail how each interaction took place so that if certain topics are unclear from the diagram then the reader can reference the description.

Class Diagram and Interface Specification

OLD Class Diagram Full

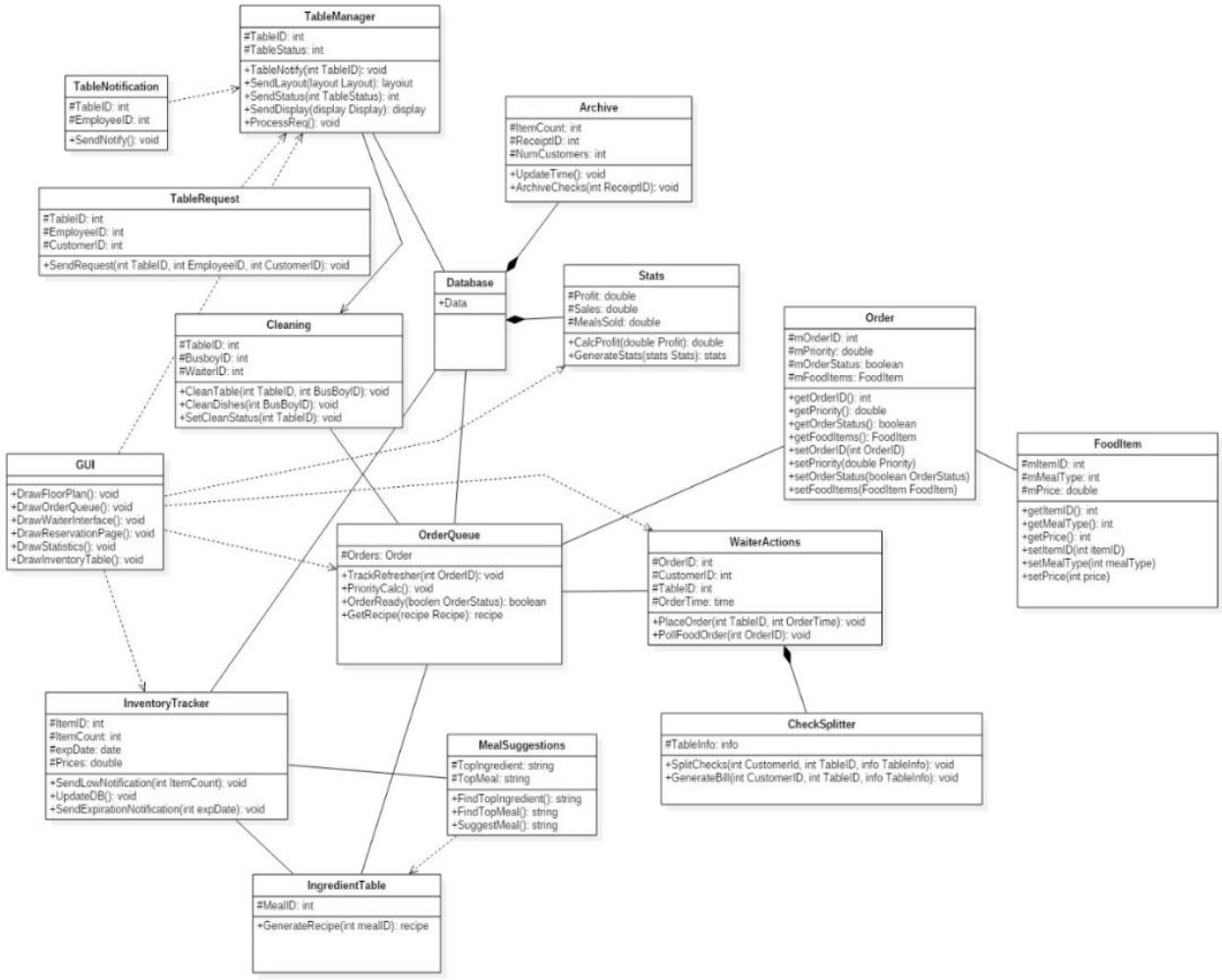


Figure 1.40

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 Full

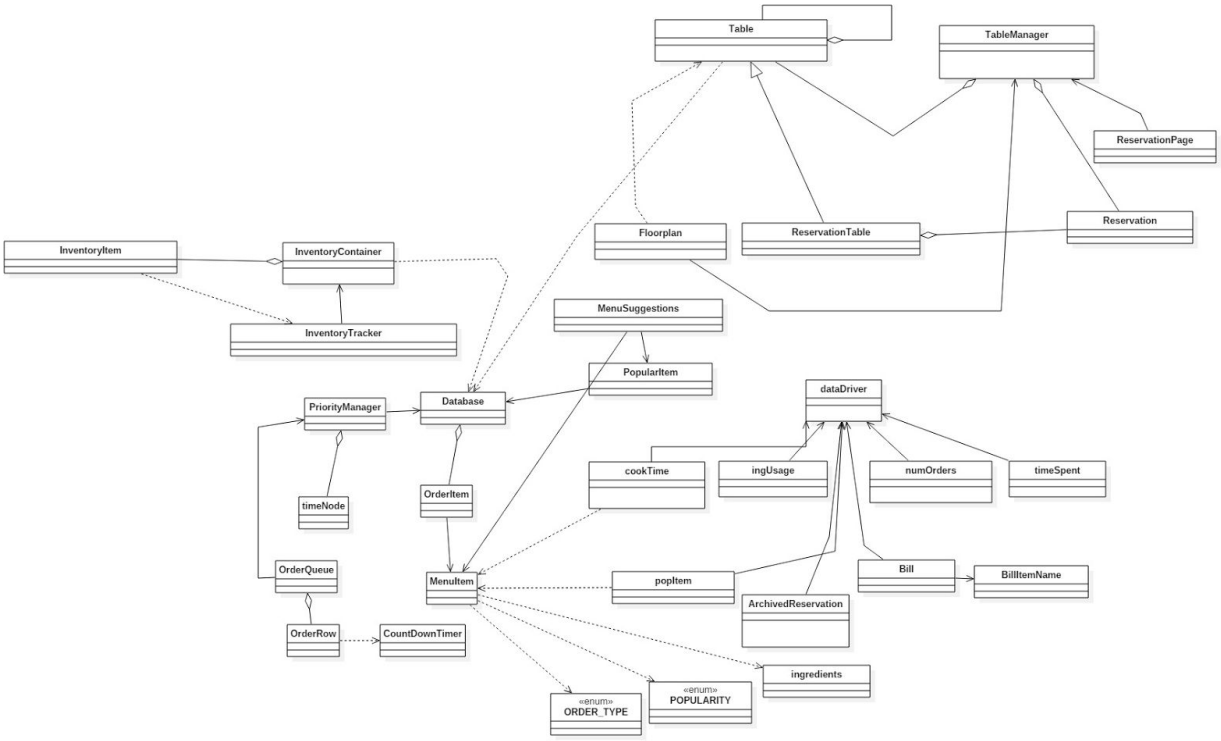


Figure 1.41

Class Diagram for Manage Order Queue

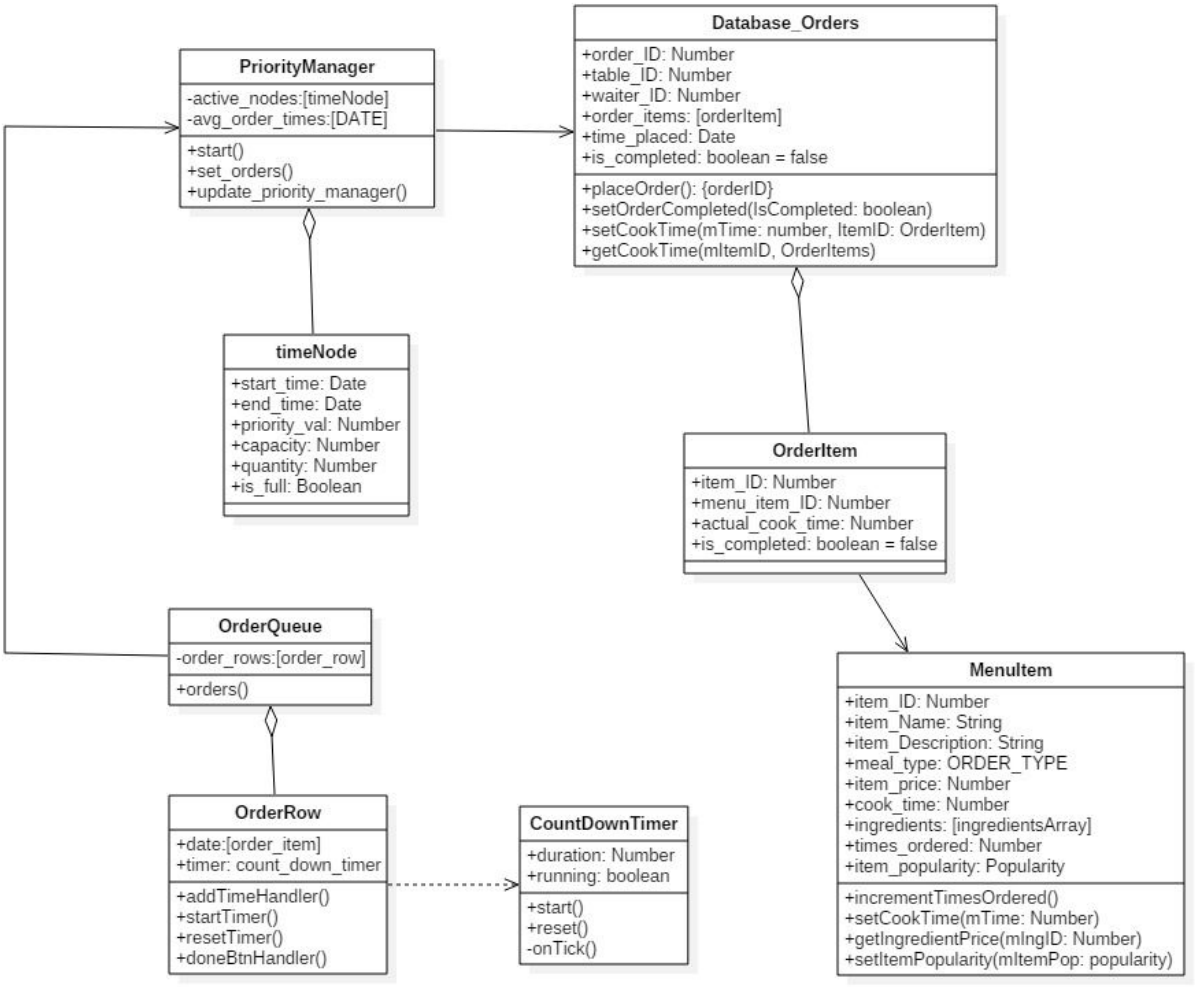


Figure 1.42

Class Diagram for Manager

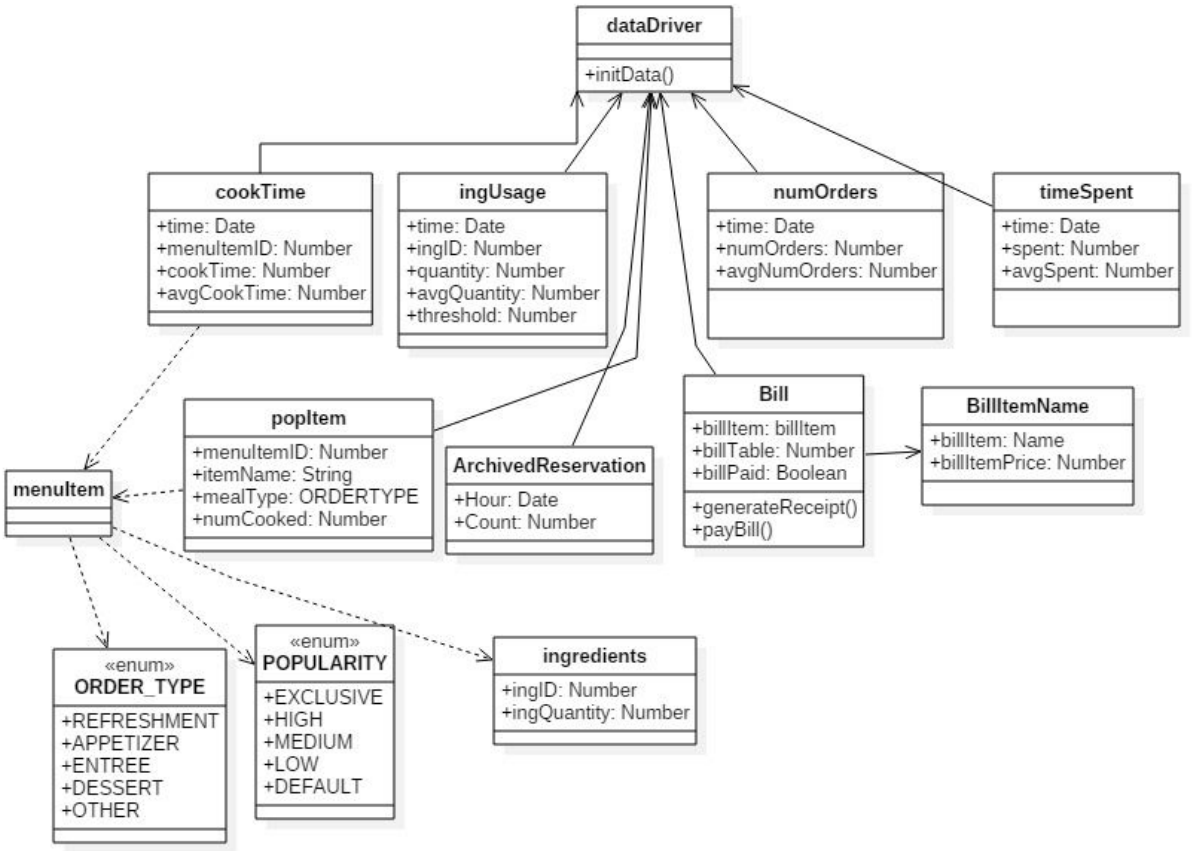


Figure 1.43

Class Diagram for FloorPlan

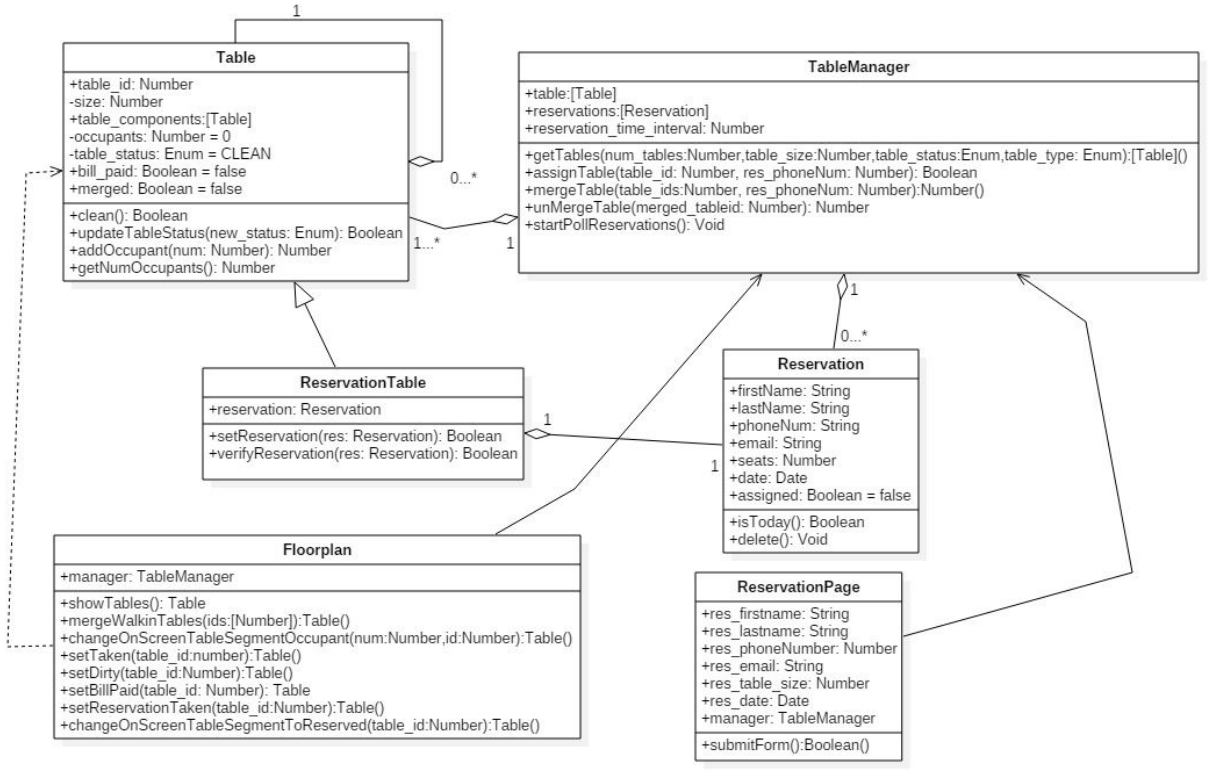


Figure 1.44

Class Diagram for Menu Suggestions

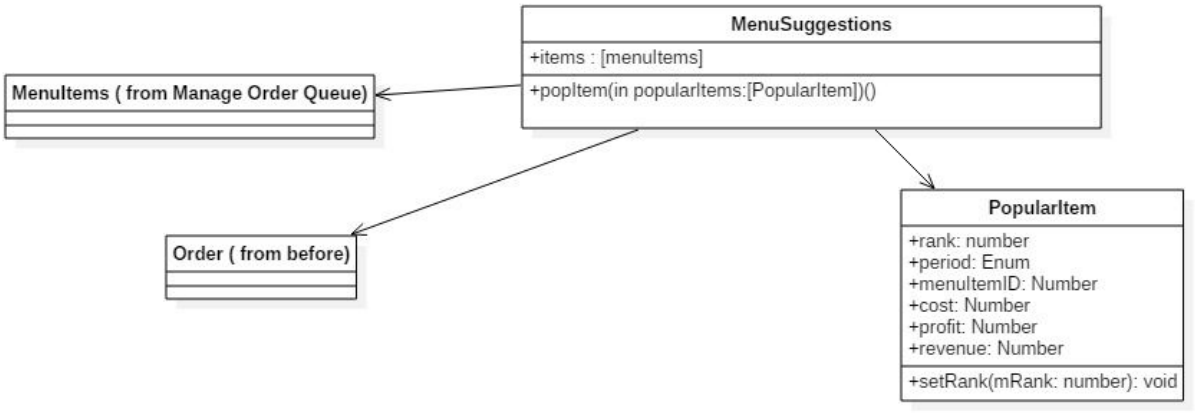


Figure 1.45

Class Diagram for Tracking Raw Materials

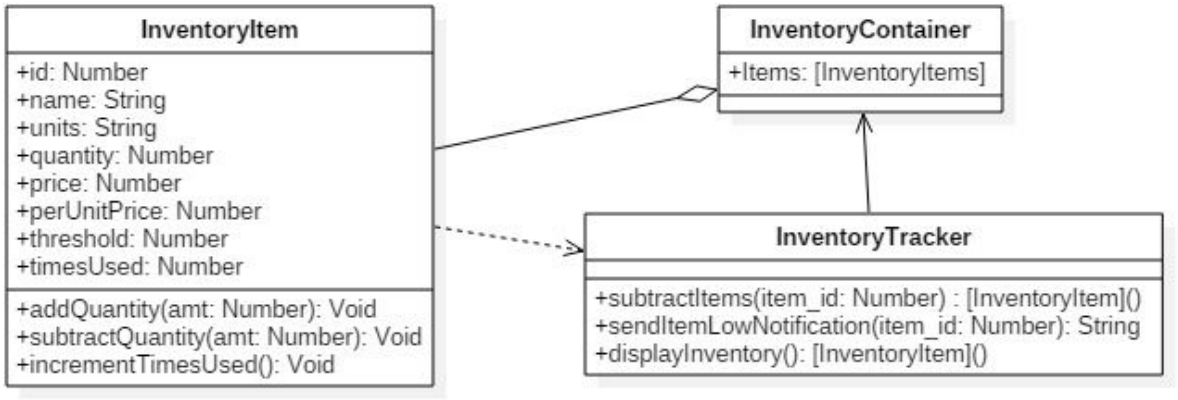


Figure 1.46

Data Types and Operation Signatures

PriorityManager

Attribute	**Description**
-active_nodes	Nodes in the order queue currently being cooked
-avg_order_times	Average cook times calculated for each active node

Table 1.29

Operations	**Description**
+start	Initializes the priorities based on collected data
+setOrders	Adds new orders to the priority manager
+updatePriorityManager	Calculates new priorities

Table 1.30

TimeNode

Attribute	Description
+start_time	Time the chef started cooking the item
+end_time	Time the chef finished cooking the item
+priority_val	Calculated priority value used in the order queue sorting
+capacity	How many items can be in cooked at once
+quantity	Number of items to be cooked
+is_full	Is true if quantity is capacity

Table 1.31

OrderQueue

Attribute	Description
-orderRows	Represents an individual row in the order queue.

Table 1.32

Operations	Description
+orders	Retrieves unprocessed orders from the Orders collection.

Table 1.33

OrderRow

Attribute	Description
+data	Contains the details from the order such as name, cook time, etc.
-timer	Timer object that is responsible for count down timer

Operations	Description
+doneBtnHandler	Contacts the server if the chef finishes the item
+addTimeHandler	Adds more time if chef needs more time
+startTimer	Timer starts running
+endTimer	Time is paused

Table 1.34

CountDownTimer

Attribute	Description
+duration	Number of seconds to be countdown
+running	True if the timer is running

Table 1.35

Operations	Description
+start	Timer starts running and duration is decremented
+reset	Timer is paused and countdown is reset to original duration
+onTick	Calls itself every second to decrement duration

Table 1.36

MenuItem

Attribute	Description
+item_ID	Unique ID given to menu items
+item_name	The name of the menuItem
+item_description	Descriptions of what the menu item contains

+meal_type	The type of meal the menu item is
+cook_time	The amount of time it takes to cook the meal
+ingredients	The ingredients that are used to create the menu items
+times_ordered	Counts the number of times the menu items will be ordered
+item_popularity	Ranks the menu items to see which item is most used

Table 1.37

Operations	**Description**
+incrementTimesOrdered	Increments the amount an item has been ordered by 1
+setCookTime	Sets the cook time of an item with a new time
+getIngredientPrice	Gets the price of the selected menu item
+setItemPopularity	Sets the rank of a selected item for how frequently its used

Table 1.38

OrderItem

Attribute	**Description**
+item_id	The ID number of an item within an order.
+menu_item_id	The corresponding menu item ID of an order item
+actual_cook_time	The cook time it takes to cook a menu item.
+is_completed	Indicates if orderItem has been completed

Table 1.39

Order

Attribute	Description
+order_ID	The ID of an individual order
+table_ID	The table ID that the order belongs to
+waiter_ID	The waiter ID of the waiter who placed the order
+order_items	An array of orderItem containing all the ordered items at a table
+time_placed	The data and time the order was placed
+is_completed	Indicates if the order has been completed

Table 1.40

Operations	Description
+placeOrder	Places an order
+setOrderCompleted	Set isCompleted attribute to completed
+setCookTime	Sets the actual cook time attribute for an orderItem
+getCookTime	Retrieves the actual cook time attribute for an orderItem

Table 1.41

MenuSuggestions

Attribute	Description
-pop_items	Table containing the menu_item_ids of items which their ranks

Table 1.42

Operations	Description
+itemsRank	Ranks the items based on the ranking

	algorithm

Table 1.43

PopularItem

Attribute	**Description**
-rank	The rank of an item on the menu suggestion leaderboard
+period	The popularity period of a popularItem
+menu_item_ID	The corressponding menu item ID of the popularItem
+cost	The total cost of a popularItem in the specified period
+profit	Thetota profit of a popularItem in the specified period
+revenue	The total revenue of a popularItem in the specified period

Table 1.44

Operations	**Description**
+setRank	Sets the rank attribute of a popularItem

Table 1.45

DataDriver

Operations	**Description**
+initData	Initializes/cleans the data in each data table

Table 1.46

CookTime

Attribute	Description
+time	Current time the data point was measured
+menu_item_id	ID of the menu item of which cook time was measured
+cook_time	Cook time measured
+avg_cook_time	Average cook time up to this point from beginning of measurement

Table 1.47

IngUsage

Attribute	Description
+time	Current time at which data point was created
+ing_id	ID of the ingredient
+quantity	Quantity of the ingredient left
+avg_quantity	Average quantity of the ingredient in store
+threshold	Low threshold of the ingredient (input by the manager)

Table 1.48

NumOrders

Attribute	Description
+time	Time at which the data point was created
+num_orders	Number of orders in the order queue at this point

+avg_num_orders	Average number of orders today

Table 1.49

TimeSpent

Attribute	**Description**
+time	Time at which the data point was created (taken every hour)
+spent	Time spent by a customer at the restaurant (
+avg_spent	Average time spent by customers at this time

Table 1.50

PopItem

Attribute	**Description**
+menu_item_ID	ID of the popular menu item
+item_name	Name of the popular menu item
+meal_type	Meal Type of the popular menu item
+num_cooked	Number of Orders of the popular menu item (this is a temporary storage for the week)

Table 1.51

ArchivedReservation

Attribute	**Description**
+hour	Represents a group of reservations made within an hour of the day (1-24)
+count	The number of reservations made in an hour period.

Table 1.52

Bill

Attribute	Description
+bill_items	All the items ordered by a table to be factored into the bill
+bill_table_number	The table ID that the bill belongs to
+bill_paid	If the bill has been paid or not

Table 1.53

Operations	Description
+generateBill	Generates the bill for a table
+payBill	Pay the bill and update database components

Table 1.54

BillItemName

Attribute	Description
+bill_item_name	Name of the item on the bill
+bill_item_price	Price of the item on the bill

Table 1.55

Ingredients

Attribute	Description
+ing_ID	Ingredient ID
+ing_name	Ingredient's name
+ing_quantity	Quantity of ingredient left in store

Table 1.56

Popularity

Attribute	Description
+EXCLUSIVE	Highly popular item (usually this week,

	month and quarter)
+HIGH	Popular item (this week and month)
+MEDIUM	Popular this week
+LOW	Not popular
+DEFAULT	Initial status

Table 1.57

OrderType

Attribute	**Description**
+REFRESHMENT	Menu item of type refreshment
+APPETIZER	Menu item of type appetizer
+ENTREE	Menu item of type entree
+DESSERT	Menu item of type dessert
+OTHER	Unclassified menu item

Table 1.58

Object Constraint Language (OCL) Contracts

Floor Plan

```
Table
        context Table inv:
                self.table_id > 0
                self.size > 0
        context Table::clean():Boolean
                pre:
                        self.table_status == DIRTY
                        self.occupants > 0
                        self.bill_paid == true
                post:
                        self.table_status == CLEAN
                        self.bill_paid == false
                        self.occupants == 0
```

```
        context Table::updateTableStatus(new_status: Enum):Boolean
                pre:
                        self.table_status != new_status
                        new_status == CLEAN || new_status == DIRTY || new_status ==
TAKEN || new_status == RESERVED
                post:
                        self.table_status == new_status
        context Table::addOccupant(num: Number):Number
                pre:
                        num > 0
                post:
                        self.occupants + num <= self.size
        context Table::getNumOccupants():Number:
                pre:
                        self.occupants > 0
TableManager

        context TableManager inv:
                self.reservation_time_interval > 0
                self.tables.isEmpty() == false
        context TableManager::getTables(num_tables: Number,table_size: Number,
table_status: Enum, table_type: Enum): [Table]
                pre:
                        self.num_tables > 0
                        self.table_size > 0
                        self.table_type == WALKIN || self.table_type == RESERVED
        context TableManager::assignTable(table_id: Number, res_phoneNum:
Number):Boolean
                pre:
                        self.tables.exists(table | table.table_id == table_id) == true
                        self.reservations.exists(res | res.phoneNum == res_phoneNum)
                        self.tables[table_id].occupants == 0
                        self.reservations[res_phoneNum].assigned == false
                        self.reservations[res_phoneNum].date -  now.date <
self.reservation_time_interval
                post:
                        self.tables[table_id].reservation.phoneNum == res_phoneNum
                        self.reservations[res_phoneNum].assigned == true
                        self.tables[table_id].occupants ==
self.reservations[res_phoneNum].seats

context TableManager::mergeTable(table_ids: [Number], res_phoneNum: Number): Number
```

```
        pre:
                table_ids.size() >= 2
                table_ids.forAll(id | self.tables.exists(table | table.table_id == id) == true) ==
true
                self.reservations.exists(res | res.phoneNum == res_phoneNum) == true
        post:
                self.tables.exists(table | table.table_components.intersection(table_ids) ==
table_ids) == true
                self.table_ids.forAll(id : self.tables[id].merged == true) == true
                merged_table = self.tables.select(table |
(table.table_components.intersection(table_ids) == table_ids))
                merged_table.occupants ==  self.reservations[res_phoneNum].seats
                merged_table.size == self.reservations[res_phoneNum].seats

context TableManager::unMergeTable(merged_tableid: Number) :Number
        pre:
                self.tables.exists(table | table.table_id == merged_tableid) == true
                self.tables[merged_tableid].table_components.isEmpty() == false
                components = self.tables[merged_tableid]. table_components
        post:
                components.forAll(id | self.tables[id].merged == false)
                self.tables.exits( table | table.table_id == merged_tableid) == false
//all reservations with dates less than reservation_time_interval from now, should be assigned
a table
context TableManager::startPollReservations(): Void
        pre:
                self.reservations.isEmpty() == false
        post:
                self.reservations.select(res | res.date - now.date <
self.reservation_time_interval) .forAll(now_res | now_res.assigned == true) == true

ReservationTable
        context ReservationTable::setReservation(res: Reservation): Boolean
                pre:
                        TableManager.reservations.exists(res_t | res_t == res) == true
                post
                        self.reservation == res
        context ReservationTable::verifyReservation(res: Reservation): Boolean
                pre:
                        TableManager.reservations.exists(res_t | res_t == res) == true
                post:
                        //returns true if reservation has valid time
```

Reservation

context Reservation: **inv**
self.seats > 0

context Reservation::isToday():Boolean
pre:
TableManager.reservations.exists(res | res == self) == true
post:
//return true if reservation is today
context Reservation::delete():Boolean
pre:
TableManager.reservations.exists(res | res == self) == true
post:
TableManager.reservations.exists(res | res == self) == false

ReservationPage
context ReservationPage::submitForm():Boolean
pre:
self.res_phoneNumber != null
self.res_firstName != null
self.res_lastName != null
self.res_email != null
self.res_date != null
post:
self.manager.select(res| res.phoneNum ==self.res_phoneNum).size()
== 1

FloorPlan
context FloorPlan: **inv**
self.manager != null
context FloorPlan::showTables(): [Table]
pre:
self.manager.tables.isEmpty() == false
post:
//returns list of tables
context FloorPlan::mergeWalkinTables(ids: [Number]):Table
pre:
ids.isEmpty() == false
post:

```
                    self.manager.tables.exists(table |
table.table_components.intersection(ids) == ids) == true

        context FloorPlan::changeOnScreenSegmentOccupants(num: Number, id:
Number):Table
                pre:
                    self.manager.tables.exists(table | table.table_id == id) == true
                    num > 0
                post:
                    self.manager.tables[id].occupants + num =<
self.manager.tables[id].size

        context FloorPlan::setTaken(table_id: Number): Table
                pre:
                    self.manager.tables.exits(table | table.table_id == table_id) == true
                    self.manager.tables[table_id].table_status != TAKEN
                post:
                    self.manager.tables[table_id].table_status == TAKEN

        context FloorPlan::setDirty(table_id: Number): Table
                pre:
                    self.manager.tables.exits(table | table.table_id == table_id) == true
                    self.manager.tables[table_id].table_status != DIRTY
                post:
                    self.manager.tables[table_id].table_status == DIRTY
        context FloorPlan::setBillPaid(table_id: Number):Table
                pre:
                    self.manager.tables.exits(table | table.table_id == table_id) == true
                    self.manager.tables[table_id].bill_paid == false
                post:
                    self.manager.tables[table_id].bill_paid == true
        context FloorPlan::setReservationTaken(table_id: Number):Table
                pre:
                    self.manager.tables.exits(table | table.table_id == table_id) == true
                    self.manager.tables[table._id].table_type == RESERVATION
                    self.manager.tables[table_id].table_status == RESERVED

                post:
                    self.manager.tables[table_id].table_status == TAKEN
        context FloorPlan::changeOnScreenTableSegmentToReserved(table_id: Number):
Table
                pre:
```

```
                    self.manager.tables.exits(table | table.table_id == table_id) == true
                    self.manager.tables[table._id].table_type == RESERVATION
                    self.manager.tables[table_id].table_status != RESERVED

            post:
                    self.manager.tables[table_id].table_status == RESERVED
```

Table 2.1

MenuSuggestions

```
MenuSuggestions:
        context MenuSuggestions::popItem(popularItems:[PopularItem]):Void
                pre:
                        popularItems != null
                        popularItems.forAll(item: item.rank>0) == true
                post:
                        //returns the list of menu suggestions
PopularItem
        context PopularItem::SetRank(mRank: Number): Void
                pre:
                        mRank > 0
                post:
                        self.rank ==mRank
```

Table 2.2

Manager

```
Bill:
        context Bill:generateReceipt()
                pre:
                        self.billItem != null
                        self.billTable != 0
                        self.billPaid == true
                post:
                        //returns receipt
        context Bill:payBill()
                pre:
                        self.billItem == null
                post
                        self.billItem != null
```

Table 2.3

Tracking Raw Materials

```
InventoryItem:
context InventoryItem::addQuantity(amt:Number):Void
        pre:
                self.id != null
                self.name != null
                old_quantity = self.quantity
        post:
                self.quantity = amt+old_quantity

context InventoryItem::subtractQuantity(amt:Number): Void
        pre:
                self.id!=null
                old_quant = self.quantity
        post:
                self.quantity = old_quantity - amt

context InventoryItem::incrementTimesUsed():void:
        pre:
                self.id!=null
                old_times = self.timesUsed
        post:
                self.times_use = old_times +1


InventoryTracker:

context InventoryTracker::subtractItems(item_id:Number):[InventoryItem]
        pre:
                InventoryContainer.items[item_id].id != null
                old_quant = InventoryContainer.items[item_id].quantity
        post:
                InventoryContainer.items[item_id].quantity =old_quant -1

context InventoryTracker::sendItemLowNotifications(item_id:Number):String
```

```
        pre:
                InventoryContainer.items[item_id].id != null
                InventoryContainer.items[item_id].quantity <
InventoryContainer.items[item_id].threshold
        post:
                //sends notification

context InventoryTracker::displayInventory():[InventoryItem]
        pre:
                Inventory.items.size() != 0

        post:
                //returns inventory
```

Table 2.4

Manage OrderQueue

```
PriorityManager:
        context: PriorityManager::start()
                pre:
                        self != null
                post:
                        //starts priority manager

        context PriorityManager::set_orders()
                pre:
                        self.!=null
                post:
                        //prints orders

        context  PriorityManager::update_priority_manager()
                pre:
                        self!=null
                post:
                        //updated

Database_orders

        context Database_ordersr::placeOrder(itemId)
                pre:
                        itemId!=0
                post:
```

```
                    //order placed
        context Database_orders::setOrderCompleted(itemId)
                pre:
                    itemId!=0
                post:
                    self.order_items.exits(item:item.itemId==itemId)==false
        context Database_orders::setCookTime(mTime:number, itemID:OrderItem)
                pre:
                    mTime != 0
                    itemId != 0

                post:
                    //cook time set
```

Table 2.5

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

Figure 2.6

Changes from past reports

Most of our class diagram changes are updated based on the implementation of our actual code. Before, our class diagram was based off of conceptual ideas and the small amount of code we wrote. Now our class diagrams very closely represent our actual system code. Before implementing, we were able to write constraints using OCL. This allowed us to program with less errors related to end cases. In report two, it was very hard to come up with a class structure that made sense. However, once we wrote the code, we were able to get the classes directly.

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

Package Diagram

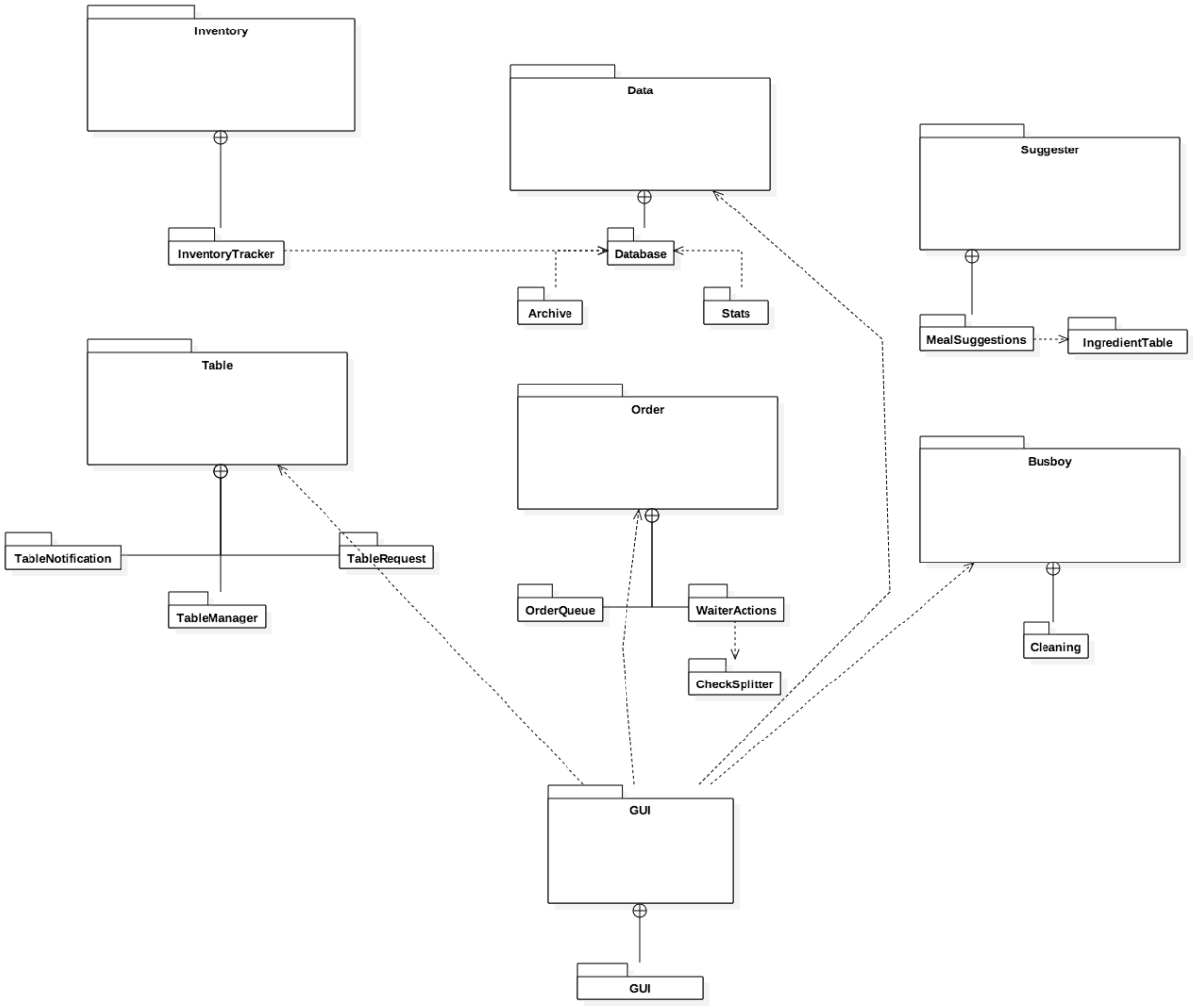


Figure 1.47

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 indications 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. Every time an order is completed, the system takes the ingredients defined for each menu item and subtracts the respective items from the inventory counts. This allows real time updates to the inventory, so manager or another employee doesn’t

have to go in count inventory every time. The suggestor can use the information in inventory, ingredient table and order to generate suggestions using the algorithm defined in figure 1.49.

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. 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 BillPayer. 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 BillPayer 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

Table 1.59

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

[10*]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. The following is the ER diagram.

ER Diagram

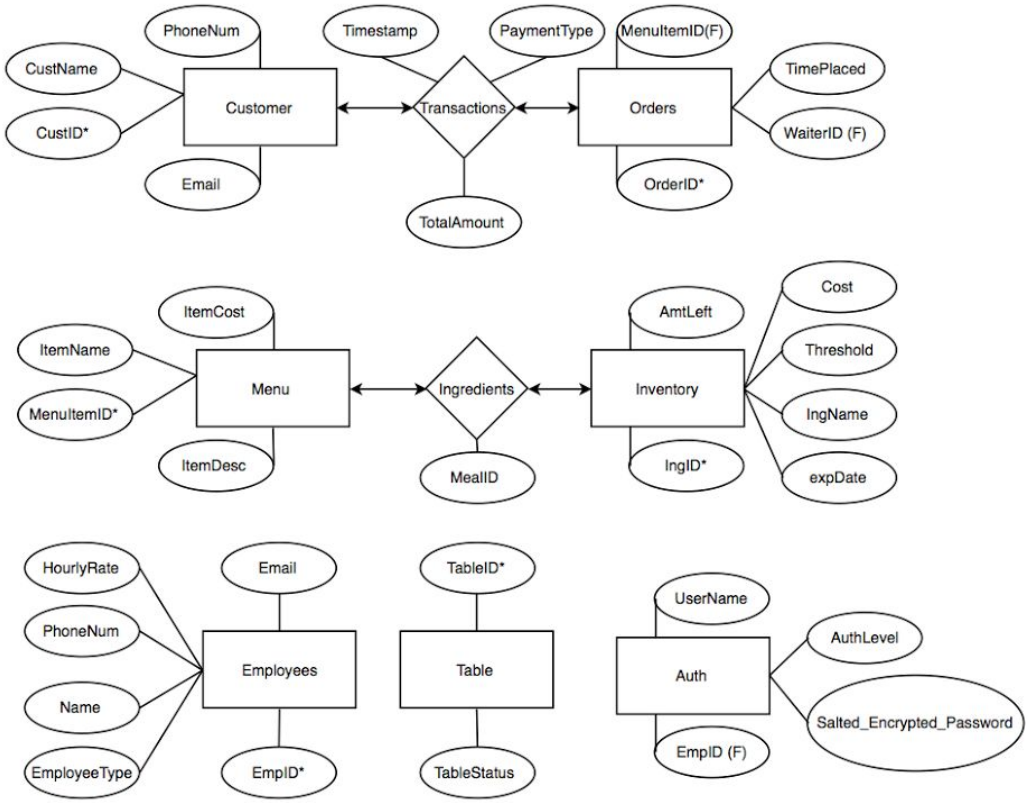


Figure 1.48

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
- Credit/Debit Card Reader

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.

Changes from past reports

Overall, this section did not undergo major changes from previous reports since we haven't changed any of the subsystems in a major way. Slight modifications would include the change of class names and the particular way in which they are executed. Meaning, what the packages do is still the same however how they carry out their tasks might be slightly modified. In terms of the hardware requirements there haven't been many changes except for the addition of a credit/debit reader that would facilitate the transactions of the restaurant.

Algorithms and Data Structures

Algorithms

Suggesting Meal Plan

One of the primary automatic data processing algorithms implemented in our system is the Menu Suggestion algorithm. The menu suggestion algorithm is designed to help the business manager make beneficial decisions by suggesting items on the menu that are both popular with customers and profitable to the restaurant. Therefore, the business manager can adjust the restaurant menu to remove items that are not popular or profitable, and keep items that make profit and customers content. For instance, an item might be making the manager a large chunk of profit in one week alone. However, that item may not sell often over a longer time period such as a month. Trends such as these need to be identified, so that the manager can make the best judgement on how to operate the restaurant.

Overall, the meal suggestion algorithm remained the same as mentioned in the past reports. However, a few changes were made to make the algorithm feasible and operate as expected in the system environment. To clarify, system environment represents the structure of our database collections and the way the code is written overall. The algorithm needs to be able to adapt to its surroundings in order to function properly. Hence, a pseudocode version of our algorithm is provided below, highlighting the important calculations. All changes made to the algorithm will be described at the end of the **Suggesting Meal Plan** section.

How does the algorithm work?

First off, the algorithm (Figure 1.49) is explained assuming that there is existing data in the database for at least the length of one quarter of a year. Specifically, the existing data are customer orders placed by waiters. Moving on, three separate time periods are allotted; *last week* (7 days), *last month* (30 days), *last quarter* (~92 days). For the three time periods, a set of existing orders is pulled from the database as previously mentioned. Next, profit, revenue, and cost are separately calculated for each item on the menu based on the orders in each time period. Therefore, each menu item will have three sets of cost, revenue, and profit (for *last week*, *last month*, and *last quarter*). Next, each set is sorted by profit in descending order. At this point, item profits are calculated for each item, but they're values are not yet relevant in terms of popularity. To determine relevant items in the sorted lists, the **top five profit makers** are taken from **each time period**. Then a popularity is assigned to each item in the menu based on which top-five time period(s) that menu item occurs in, if any. Table 1.60 shows the different possible combinations and their respective popularity levels. For instance, if an item falls within the *Exclusive* popularity, the manager is making a good profit and customers are satisfied with that

item. Therefore, the algorithm takes both profitability and customer satisfaction into account, making it a unique feature to optimize and improve business decisions.

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

}

FOR EACH : MENU_ITEMS
{
	CHECK(ALL SORTED_ITEMS_BY_PROFIT)
	{
		Find the combinations of menu item occurrences in the first five items of
		EACH SORTED_ITEMS_BY_PROFIT

		Assign popularity to a menu item based on Table 1.60
	}
}
```

Figure 1.49 - A *pseudocode* version of the meal suggestion algorithm.

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

Table 1.60 - A list of combinations mapped to popularity levels explained in the text.

The primary changes made to the meal suggestion algorithm is that only the top five profit makers are considered in choosing a rank other than low. If each item was considered in the calculation, then the characteristic bond between customer satisfaction and profitability would be lost. Items that are not too profitable would be considered, therefore, making the data less valuable.

Manage Order Queue

Node Priority:
The Order Queue is a priority based system that contains 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
```

Figure 1.50

Given the time constraint we were only able to implement a partial section of the algorithm. However, we implemented a sufficient amount of the algorithm to ensure that the order queue is still efficient. A full implementation of the algorithm would only increase the efficiency a little more. Therefore, the tradeoff of not completing the algorithm was not detrimental for our project because we were able to focus on other unique features.

Menu Item Cook Time

This data processing algorithm, although simple, helps make the cook time more accurate as order items are completed.The cook times are kept track of using the timers. Each order item in the order queue is associated with a timer which is activated when the chef clicks on the item. The timer starts counting down time as the chef is making the item. When the chef completes an order item, the algorithm averages the actual time taken to cook an item with the cook time that was previously calculated and stored in the database. It continues this process for each order that is completed and updates the database accordingly.

One beneficial outcome of this algorithm is greater customer satisfaction. The better time approximation allows waiters to inform customers on an accurate estimation of the cook time of a meal. This algorithm is prone to some skewed data collection, but can be fixed in future updates. For example bad data can come from when the chef checks off an item as done when he/she is not actually done with the item. A way to fix this issue can be addressed is by making

the use of a nonlinear smoothing function to reduce the impact of the data outliers. Another solution would be to set a “low threshold” value that could be used to make sure that the data is within an expected range of values thus preventing it from large-scale errors.

```
IF(ITEM COOKED ALREADY)
{
        ACTUALTIMETAKEN = GET ACTUAL COOKTIME
       NEW COOKTIME = (ACTUALTIMETAKEN + PREVIOUSCOOKTIME) / 2
       UPDATE COOKTIME
}
```

Figure 1.51

Statistics

The data shown in the Statistics page was randomly generated. All the time series graphs have data for a year. We used different algorithms to generate these graphs. However, in the actual product, the data would be collected from the system. The main statistic we used to analyze the data was the moving average. This allows the restaurant owner to understand actual trends by filtering day to day fluctuations in the data. We defined the moving average subset size to be the entire set collected so far.

```
CALC_MOVING_AVERAGE(PREV_MOVING_AVG, NUMBER_OF_ITEMS, NEW_VALUE) {
         MOVING_AVG = (NUMBER_OF_ITEMS * PREV_MOVING_AVG + NEW_VALUE) /(NUMBER_OF_ITEMS + 1)
}
```

Figure 1.52

Reservation/Floor Plan

Reservations are made utilizing the online customer reservation system. Each reservation is then pushed into a queue. The "Table Manager" object periodically checks, asynchronously, for reservations that are less than or equal to two hours from now. Thus, reservations are not assigned to tables on creation. If this were not done, a reservation hours or days from now would block a table from being used by any customers until then.

If the reservation is less than or equal to two hours from now, the Table Manager searches for a possible Table to assign them too.

The floor plan system is controlled by an object utilizing a hash table called the "Table Manager". The table manager assigns tables to reservations and monitors and controls the status of all tables. Tables are assigned to both reservations and walkins utilizing the "Buddy algorithm." The table manager keys into the hash table using the requested party size; if the size is odd, it is increased by one. If there does not exist a table large enough to accommodate the

part, tables of a small size are "buddied-up" to form a large table. This can be done manually, by the manager, or automatically for reservations. The tables are then broken up when the party is finished. The benefit of the "Buddy Algorithm" is that it attempts to accommodate larger parties.

```
PUSH_RESERVATION(RESERVATION){
      TABLE_MANAGER.RESERVATIONS.PUSH(RESERVATION)
}

MANUAL_MERGE(SIZE,TABLE_TYPE,TABLE_STATUS){
     HASH_TABLE = TABLE_MANAGER.T_HASH_TABLE
     RETURN BUDDY(HASH_TABLE,SIZE,TABLE_TYPE,TABLE_STATUS)
}

MANUAL_ASSIGN(PARTY_SIZE){
    TABLES =TABLE_MANAGER.T_HASH_TABLE[PARTY_SIZE]
    FOR TABLE IN TABLES{
      IF(TABLE IS CLEAN AND TABLE IS WALKIN){
          TABLE.OCCUPANTS = PARTY_SIZE
          TABLE.STATUS  = TAKEN
      }
    }
}

START_POLL_RESERVATIONS(){

               RESERVATIONS_QUEUE  = TABLE_MANAGER.RESERVATIONS
               TABLE_HASH_TABLE    = TABLE_MANAGER.T_HASH_TABLE
               WHILE TRUE {
                     CURRENT_TIME = TIME.NOW()
                     FOR ALL RESERVATIONS IN RESERVATIONS_QUEUE{
                            IF (RESERVATION IS NOT NOW OR RESERVATION IS NOT ASSIGNED){
                                     CONTINUE
                             }
                            TIME_DIFFERENCE = ABSOLUTE_VALUE(RESERVATION.TIME- CURRENT_TIME)
                           IF( TIME_DIFFERENCE <= 2){
                                 PARTY_SIZE = (PARTY_SIZE %2) ? PARTY_SIZE : PARTY_SIZE+1
                                 TABLES =
FIND_TABLE(TABLE_HASH_TABLE[PARTY_SIZE],RESERVATION,CLEAN)
                                 IF (NO TABLES){
                                     TABLES =
BUDDY(TABLE_HASH_TABLE,PARTY_SIZE,RESERVATION,CLEAN)
                                 }
                                 ASSIGN(TABLES,RESERVATION)
                           }
           }
}
           BUDDY(HASH_TABLE,SIZE,TABLE_TYPE,TABLE_SIZE){
                     SEARCH_SIZE = SIZE/2
                     WHILE SEARCH_SIZE > 1{
                       TABLES_LIST = HASH_TABLE[SEARCH_SIZE]
                         IF (SIZEOF (TABLES) > 1){
                              MERGED_TABLE = NEW TABLE(TABLES_LIST)
                              return MERGED_TABLE
```

```
                        }
                      }
                    }
                    RETURN 0
          }
         ASSIGN(TABLE,RESERVATION){
            TABLE.RESERVATION = RESERVATION
            RESERVATION.ASSIGNED = TRUE
            RESERVATION.TABLE = TABLE
            TABLE.OCCUPANTS = RESERVATION.PARTY_SIZE
            TABLE.STATUS = TAKEN
         }
```

Figure 1.53

Automated Inventory

The automated inventory algorithm, although simple, plays a crucial role in our system. It takes away the man hours spent on taking inventory and automates that problem. Now the manager or any other employee doesn't have to go in every so often to take inventory. It also allows the manager to quickly see which item is running low so he can replenish the stock.

```
// SUBTRACTING ITEMS AS THEY ARE USED
NUMINGREDEINTS = NUMBER OF INGREDIENTS IN THE ORDER BEING COMPLETED
FOR NUMINGREDIENTS {
    FIND INGREDIENT IN INVENTORY AND SUBTRACT THE AMOUNT USED IN THE ORDER
    INCREMENT THE NUMBER OF TIMES THIS INGREDIENT WAS USED
}

// ITEM LOW NOTIFICATION
IF ITEM'S QUANTITY < ITEM'S THRESHOLD {
    NOTIFY MANAGER OF LOW STATUS
}
```

Figure 1.54

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.

- ➔ The reservation system utilizes a queue to assign reservations on a "first-come, first-serve" basis. The system then periodically searches for the first reservation in the queue that is at least two hours from the current time.
    - ◆ Performance versus Flexibility: The Queue data structure maps naturally to this problem domain. Reservations are using assigned to customers who request them first. This structure is flexible because it reservations can be easily appended to the end of the list as they are created. However, it suffers from some performance issues. The queue must be traversed since the structure is ordered based on reservation creation time, not the time associated with the reservation. This must be done to find reservations that are less than two hours from now and thus must be assigned a table. However, the total size of the queue remains relatively small as reservations because reservations of the same time are constrained by the number of tables available.
- ➔ The floor plan system utilizes hash table with chaining. Where the keys are table sizes and the values are chained lists of tables of that size. This structure is called the "Table Manager".
    - ◆ Performance versus Flexibility: The hashtable data structure allows for, on average, constant time access. In addition, this structure is flexible since adding a table is done simply by appending it to the list of tables of the same size. The chaining is done since tables of the same size map to the same bin.

Changes from past reports

For the algorithms that were here from the previous reports, minor changes have been added to the way they are implemented and those changes are contained within the explanations of their algorithms. However, four new algorithms have been added and they are the menu item cook time, statistic, reservation/floorplan, and automated inventory algorithms. These four algorithms were added as our project evolved and as we started moving towards not only restaurant automation but optimization as well. These algorithms add novelty to the overall project and help make it stand out in comparison to previous projects. The details of how these algorithms are implemented are mentioned in the text above.

User Interface and Design Implementation

*This shows our development of the UI from the initial mock-ups

Preliminary Design

*The following designs are mockup wireframes created in the beginning of the project. The next section will show the fully developed user interfaces

The manager will be able to see all the checks for the given date, and be able to export them if he needs to. From this page the manager can navigate to the summary or statistics tab to view more information.

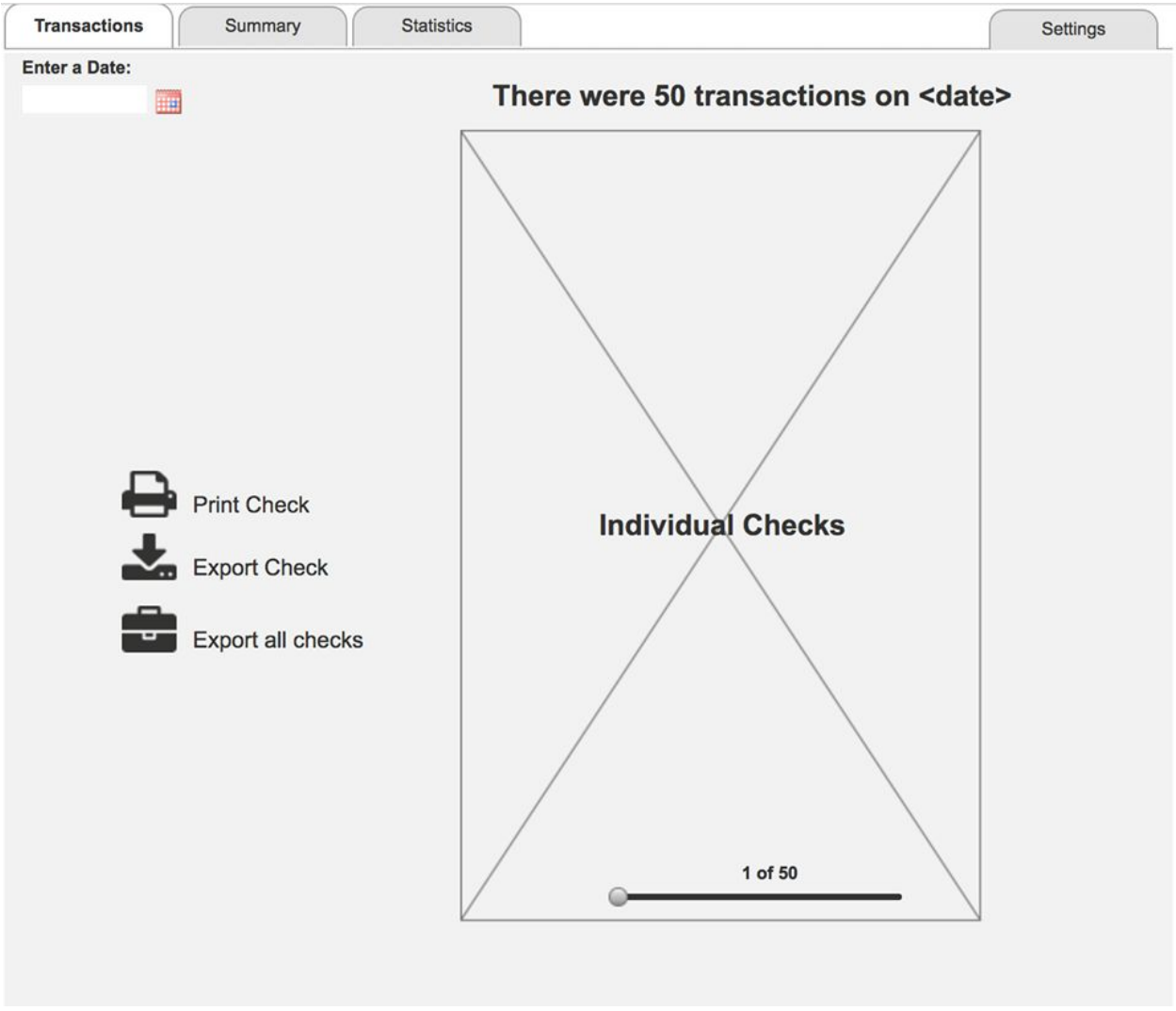


Figure 1.55

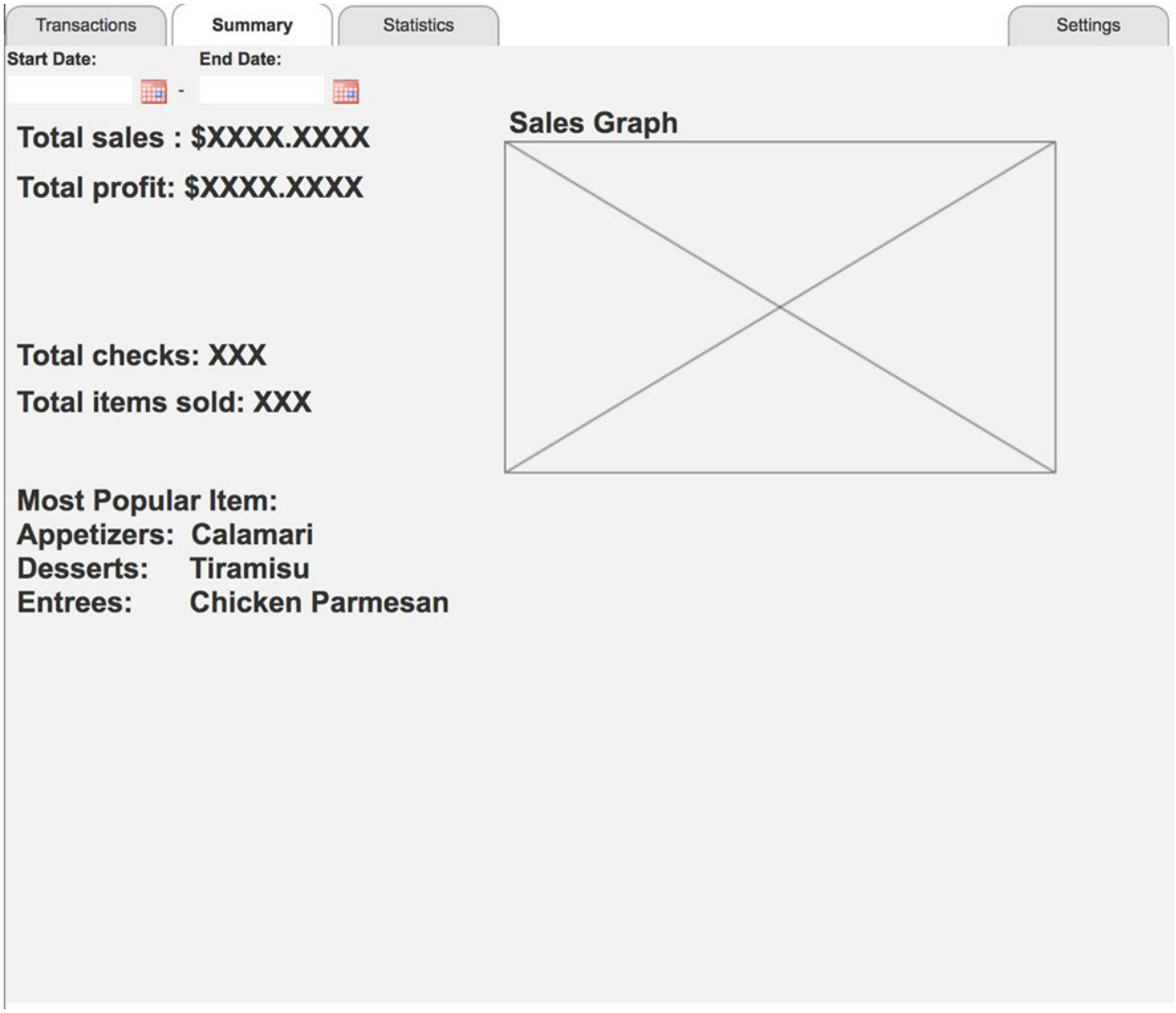

Transactions
Summary
Statistics
Settings
Start Date:
End Date:
Total sales : $XXXX.XXXX
Total profit: $XXXX.XXXX
Sales Graph
Total checks: XXX
Total items sold: XXX
Most Popular Item:
Appetizers: Calamari
Desserts: Tiramisu
Entrees: Chicken Parmesan

Figure 1.56

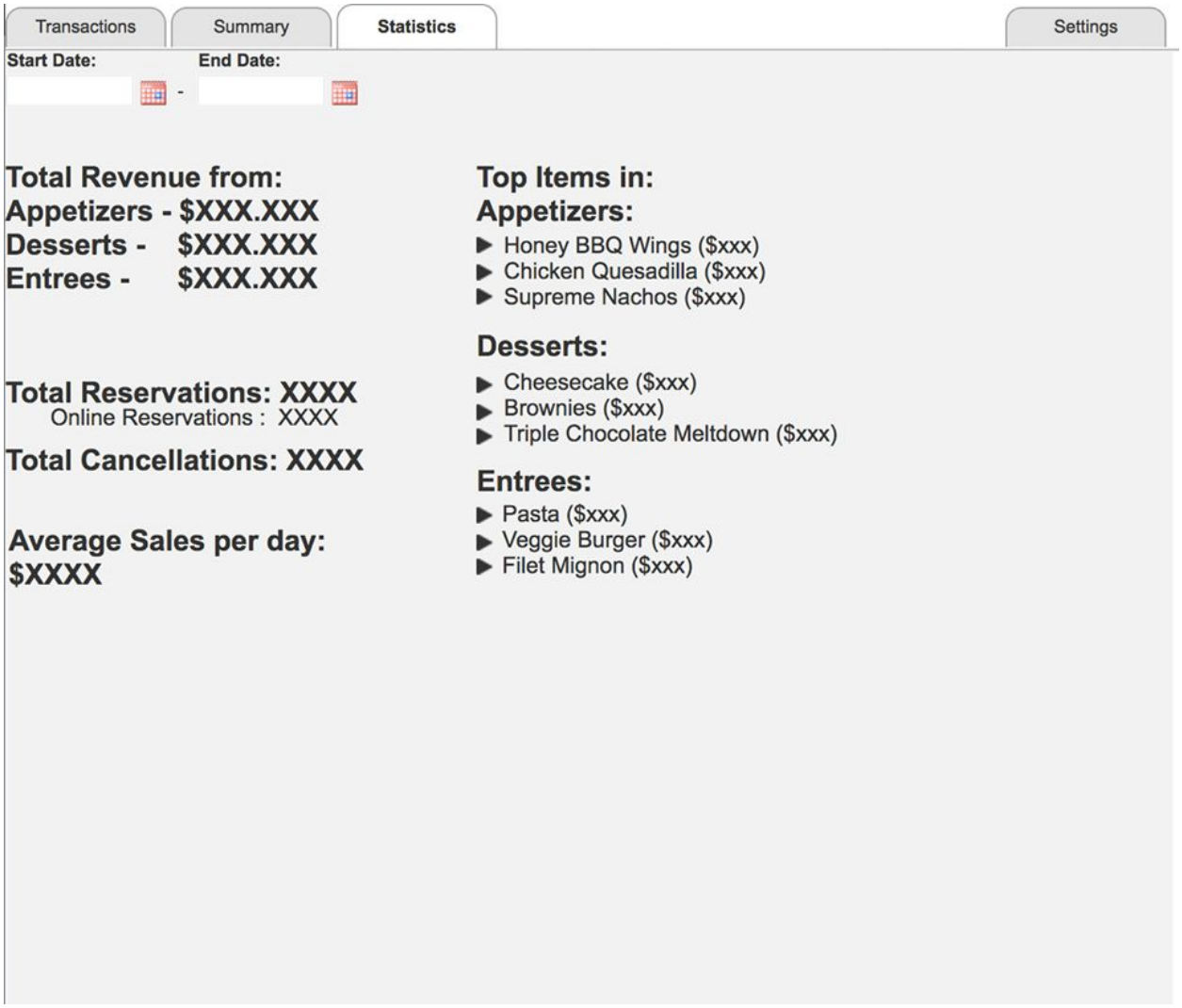

Transactions
Summary
Statistics
Settings
Start Date:
End Date:
Total Revenue from:
Appetizers - $XXX.XXX
Desserts - $XXX.XXX
Entrees - $XXX.XXX
Total Reservations: XXXX
Online Reservations : XXXX
Total Cancellations: XXXX
Average Sales per day:
$XXXX
Top Items in:
Appetizers:
Honey BBQ Wings ($xxx)
Chicken Quesadilla ($xxx)
Supreme Nachos ($xxx)
Desserts:
Cheesecake ($xxx)
Brownies ($xxx)
Triple Chocolate Meltdown ($xxx)
Entrees:
Pasta ($xxx)
Veggie Burger ($xxx)
Filet Mignon ($xxx)

Figure 1.57

The Chef will be able to see which items on the list he/she should get done first. The chef could also modify the queue if needed.

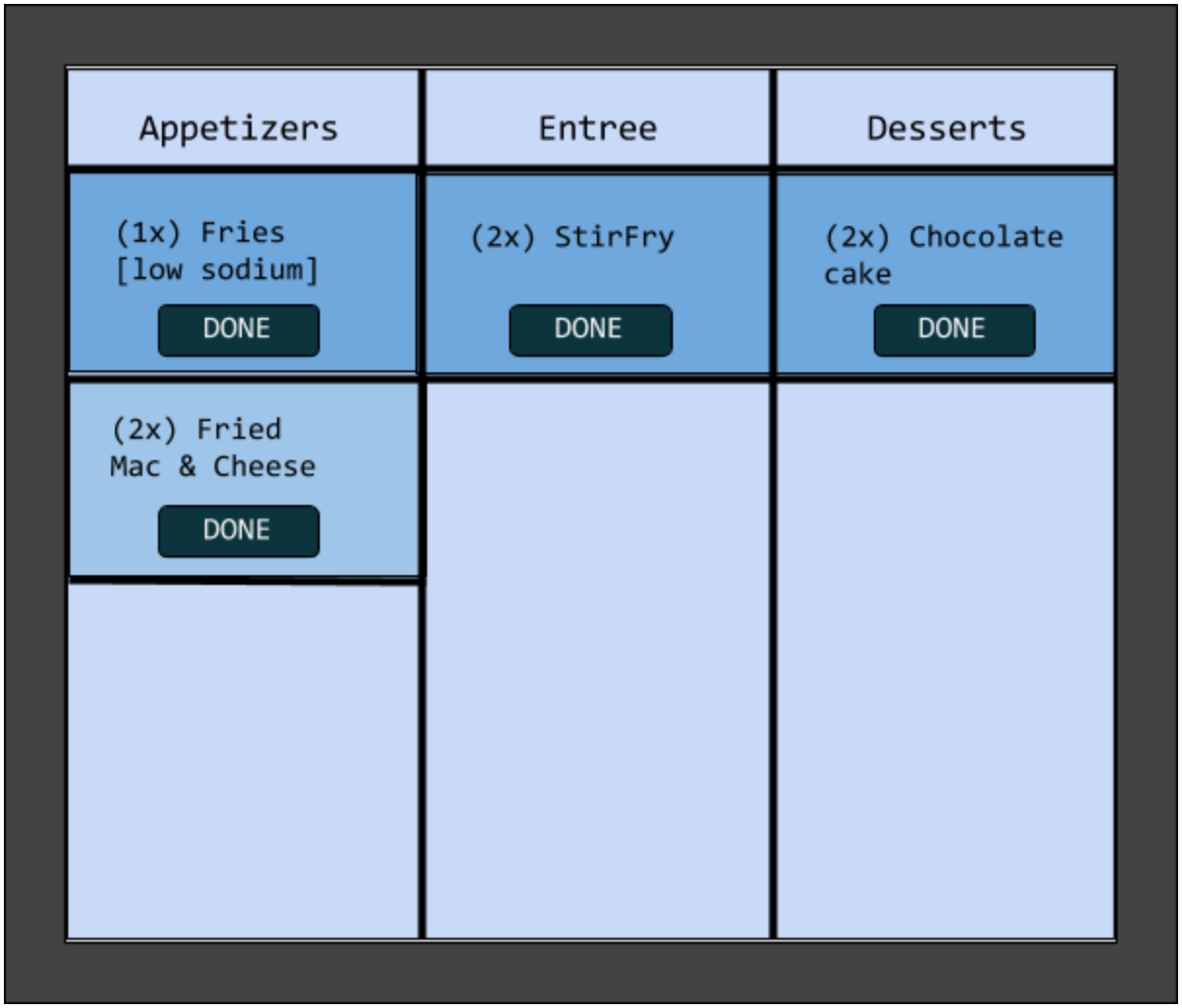
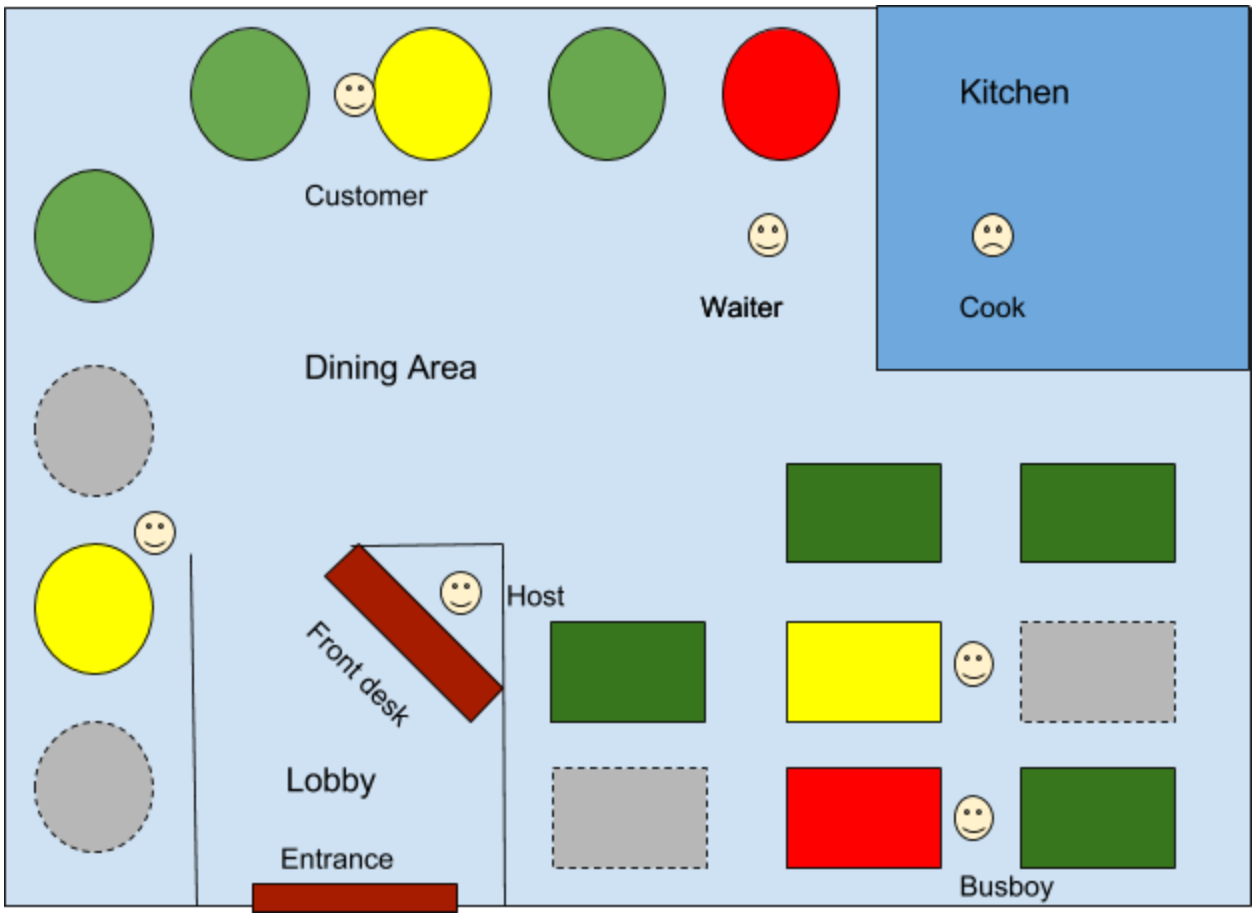


Figure 1.58

Floor Plan



Legend:

Available Table

Occupied Table

Dirty Table

Reserved Table

Figure 1.59

Final User Interface and Design Implementation

[7*][9*] In the previous section, we showed the wireframes and mock up designs created for our original three fully dressed use cases. However, as the project evolved, the interfaces and the implementation also evolved. In addition to being a restaurant automation application, our application also optimizes the restaurant by using data collected from the system. The data is analyzed by algorithms, graphs, and statistics in order to improve the restaurant. In order to achieve complex interface design, we used modern front-end frameworks such as BootStrap, jQuery, and Semantic UI which allowed easy customization and quick templates for building professional and intuitive interfaces. In the following sections, we will showcase our interfaces and describe how they play an important role in making the application easy to use.

Inventory Tracking

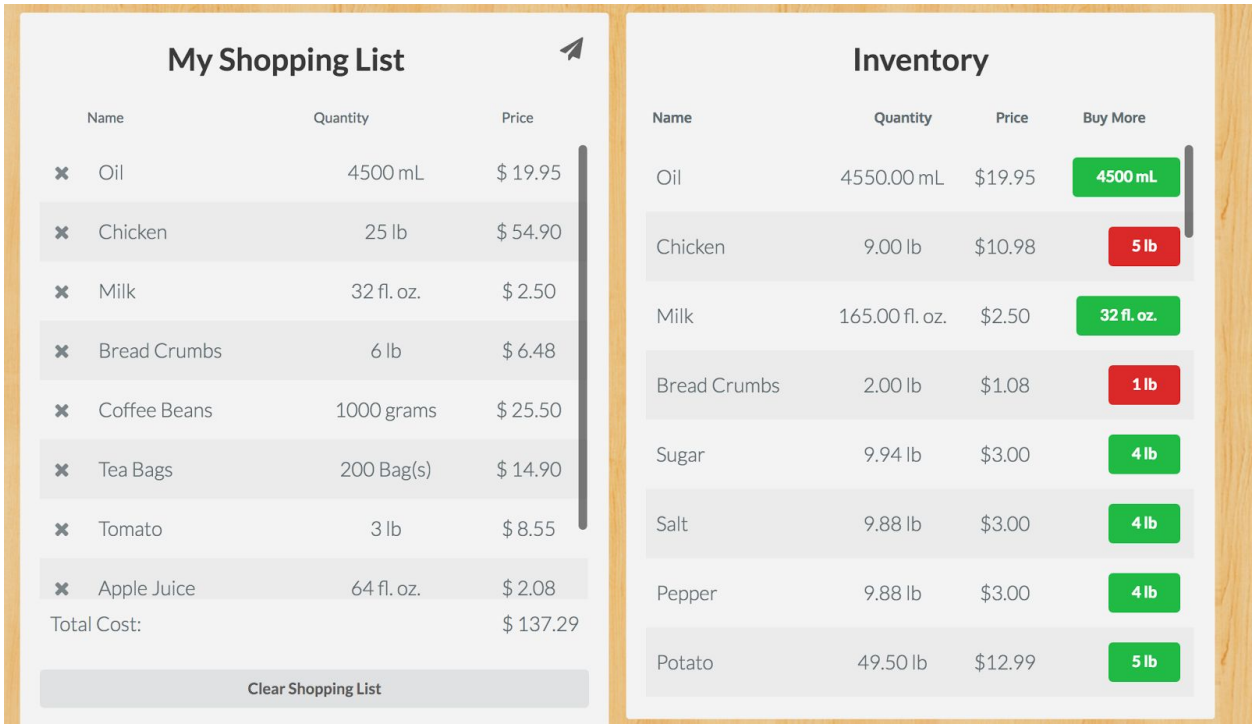


Figure 1.60

The inventory page will be used by the manager to keep track of the inventory. When you arrive at this page, you will be shown two sections, one is a shopping list and the other a list of items in your inventory. In the inventory section, it shows the name, quantity, price and how much of the item you can buy with the given price. Each item has its own quantity and the amount you can buy with the price. The order of items shown in the interface is ordered by the frequency of item

used, so the top item is used the most in the orders. *Buy More* column includes buttons of green and red colors which signify if the item is above or below the predefined threshold level. The items automatically deduct based on the orders that were completed, so manager doesn't have to go in everyday and check inventory and update it manually. If it is red, it means the manager should order the item soon so it doesn't hinder the restaurant's performance. On the other side of the page, the shopping list will initially be empty since you haven't selected anything to buy, but you can click on the button in the *Buy More* column to add it to the shopping list, clicking on it twice will modify the previously added item instead of adding two of them. If the manager accidently adds an item to the shopping list, he can remove it using the ***X*** button located on the left of each item. He can also click on clear shopping list in order to remove all of the items from the list. After the order is finalized, the manager can click on the paper airplane icon on the top right corner of the *My Shopping List* to place to order.

Order Queue

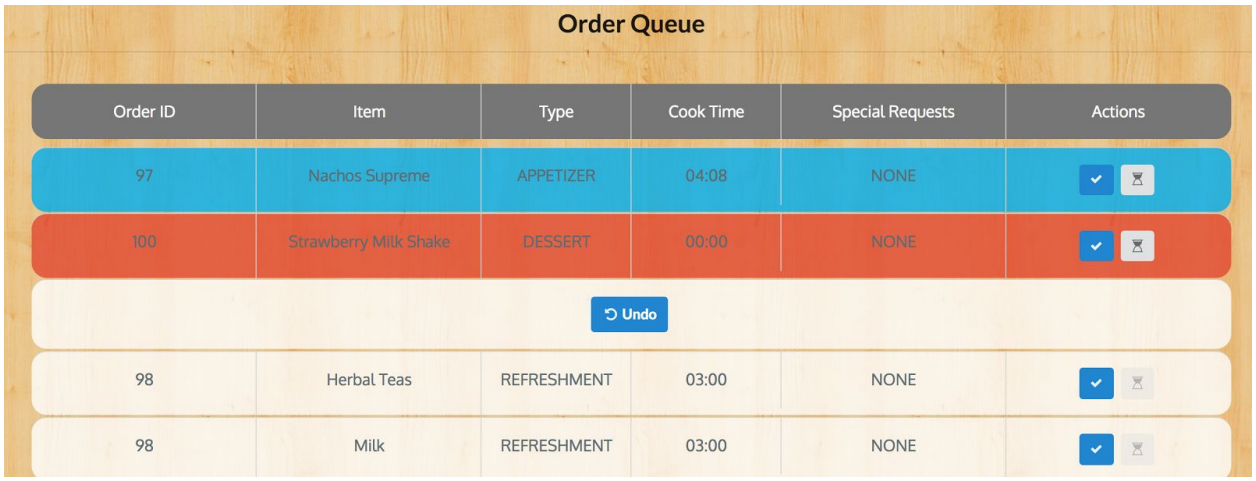


Figure 1.61

The Order Queue interface is used by the Chef to organize and track the incoming orders. The orders are displayed as rows with columns labeled, *Order ID, Item, Type, Cook Time, Special Requests* and *Actions*. The *Order ID* section shows the order ID stored in the database for the respective order. The *Item* is the item name of the item in the order, such as Nachos Supreme. The *Type* of the order specifies which category it falls in, such as appetizer, dessert, entree and refreshment. The *Cook Time* shows the calculated cook time using the algorithm specified in the algorithms section. *Special Requests* will show any special requests such as Less Oil, Low Sugar, etc. The *Actions* column has two icon buttons, a check and an hourglass. The check completed the order and displayed an empty row with an undo button, in case the chef accidentally clicks on the check icon button. The timer button is disabled unless the order is selected, which is indicated by the blue colored row. Clicking on the order row starts the timer for for that order

and turns the row blue. If the time runs out, the row turns red indicating the order is behind schedule and the chef failed to make it in the predicted cook time. While the row is still blue, the chef can click on the timer button to add 30 seconds to current time.

Reservation Page & Confirmation Email

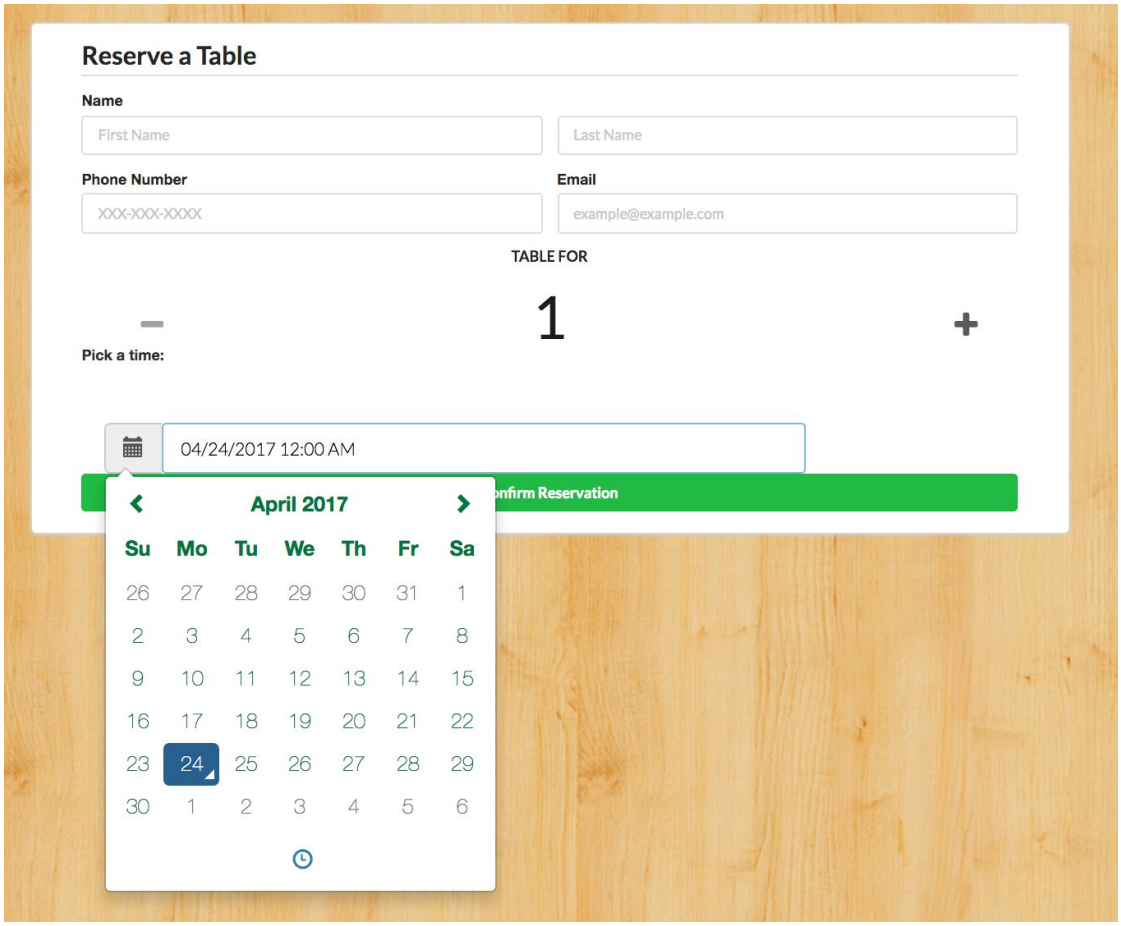


Figure 1.62

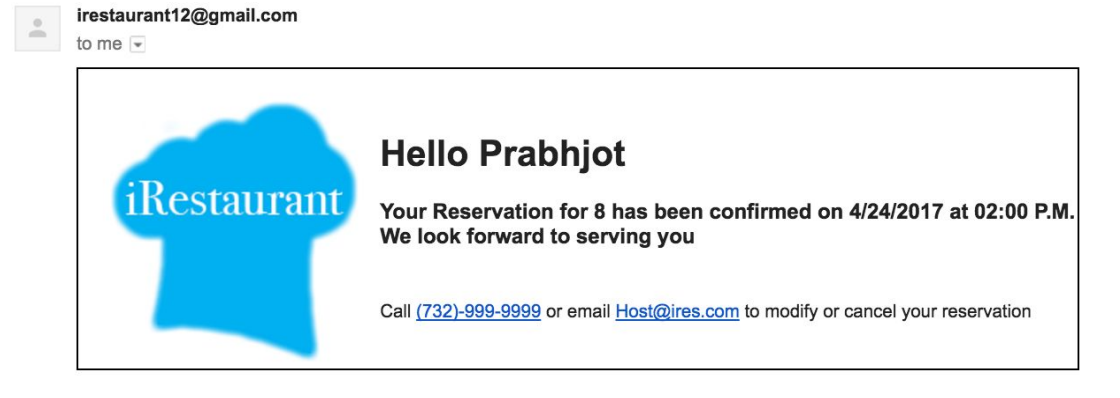


Figure 1.63

The Reservation Page and Email Confirmation shown above are used by the customers of the restaurant to make a reservation and get confirmation. When the customer arrives on the reservation page, they are presented with a form to fill out for the reservation, all these fields are required. The customer enters their first name, last name, phone, and email. After inputting that information, the customer can use the + and - buttons to change the number of people the reservation is being requested for. After selecting the number of people, the customer can click on the calendar icon and navigate through to select a date. After the form is complete, the customer clicks on the green confirm reservation button to send the confirmation.
After the email is received on the user end, it shows up as the figure above detailing the size of the party along with the date and time of the reservation.

Floorplan Interface

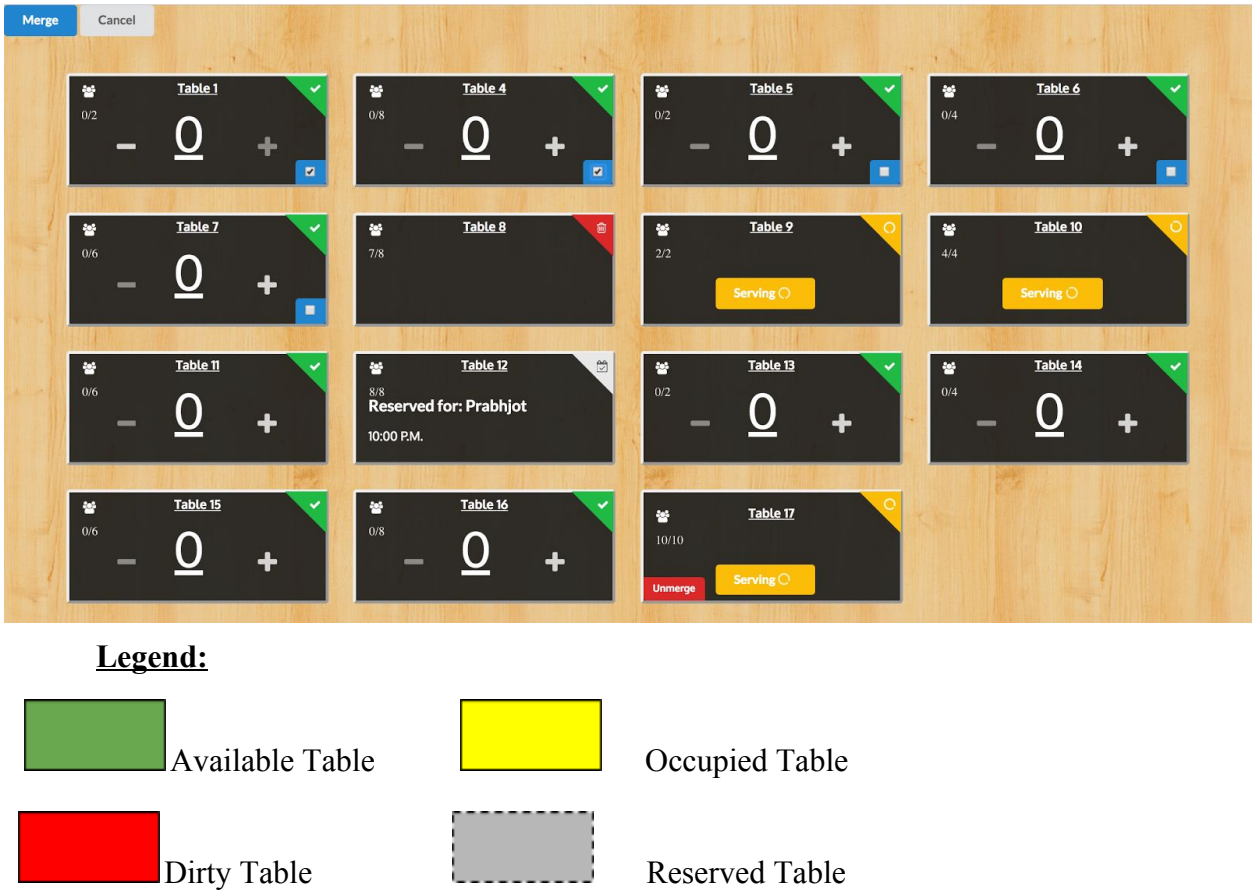


Figure 1.64

The floor status user interface is used by the Host, the Waiter and the BusBoy. It is used to display the status of each table in the restaurant, in this case we have 16 total tables. Each table's

segment has information about the table. In the top right corner it displays a label with the status of each table, the green checked arrow means it's a clean table and ready to have a customer take it. Clicking on the greek label assigns the table to the specified customers using the + and - buttons. The yellow animated circle means the table is taken and the customer is being served by the waiter. After the customer pays their bill, the red label with a trash icon shows that the table is dirty and needs to be cleaned before it can be assigned to another customer. In addition to that, in the top left corner, the table displays number of people occupying it (0,2,4,6,8, etc.) over the size of the table (2, 4, 6, 8, etc.). When a customer walks in, the host can use the + / - buttons to specify the number of customers to be seated. If there are groups of customers that exceed the maximum table size of 8, the host can merge two tables together on the interface to combine them. This will create a new table with a table number higher than the maximum number of 16. In the figure above, table 17 was merged by combining a table of 4 and 6 to create a table of 10, it could also be merged using any combination of tables adding up to 10. In the bottom left corner of the merged table, there is a button to unmerge the table, returning them to the original values and positions. The host can merge the table by clicking on the top left corner of the page, selecting the tables using checkboxes and clicking merge. The host can also click on the reserved table gray corner label to seat the group that reserved the table when they arrive.

Menu Suggestions

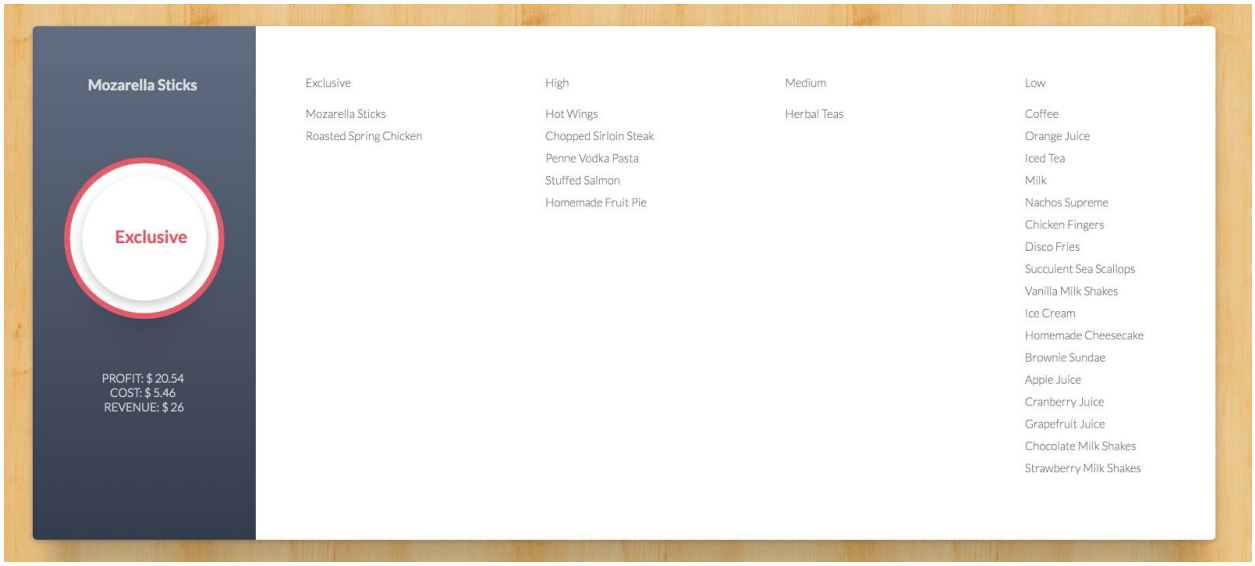


Figure 1.65

The menu suggestions page is for the manager and it displays all the items in the menu currently after they are processed and sorted through the algorithm described in ____. It separates the items into categories of *Exclusive, High, Medium* and *Low.* Clicking on any of the items brings up the statistics in the navy blue left sidebar. It showcases the menu item name, rank (exclusive,

high, medium or low), profit, cost and revenue based on the prices of ingredients used in each item and price of the item.

Statistics and Archive

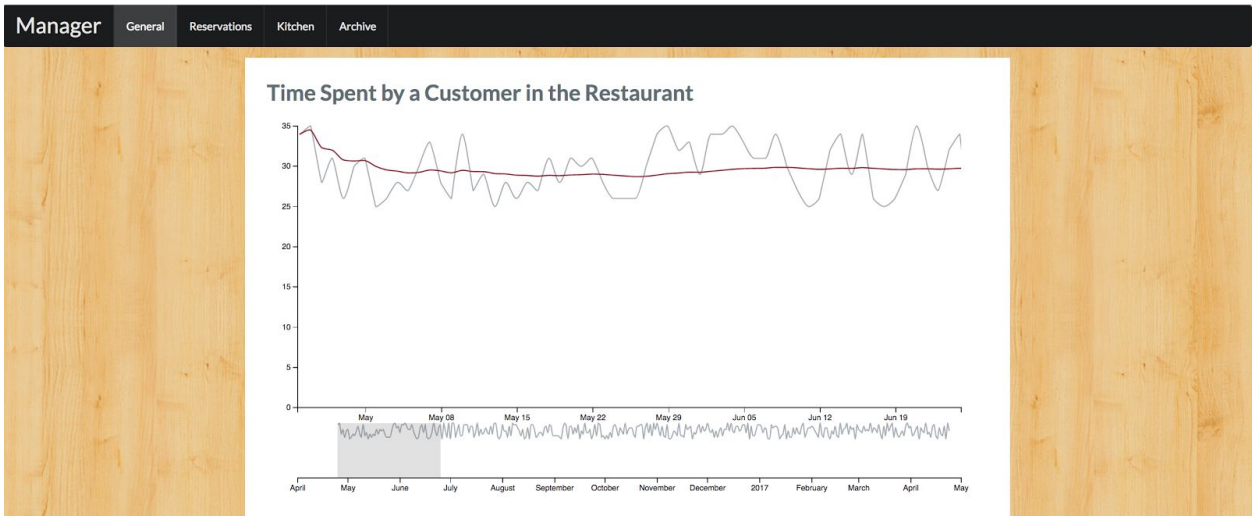


Figure 1.66

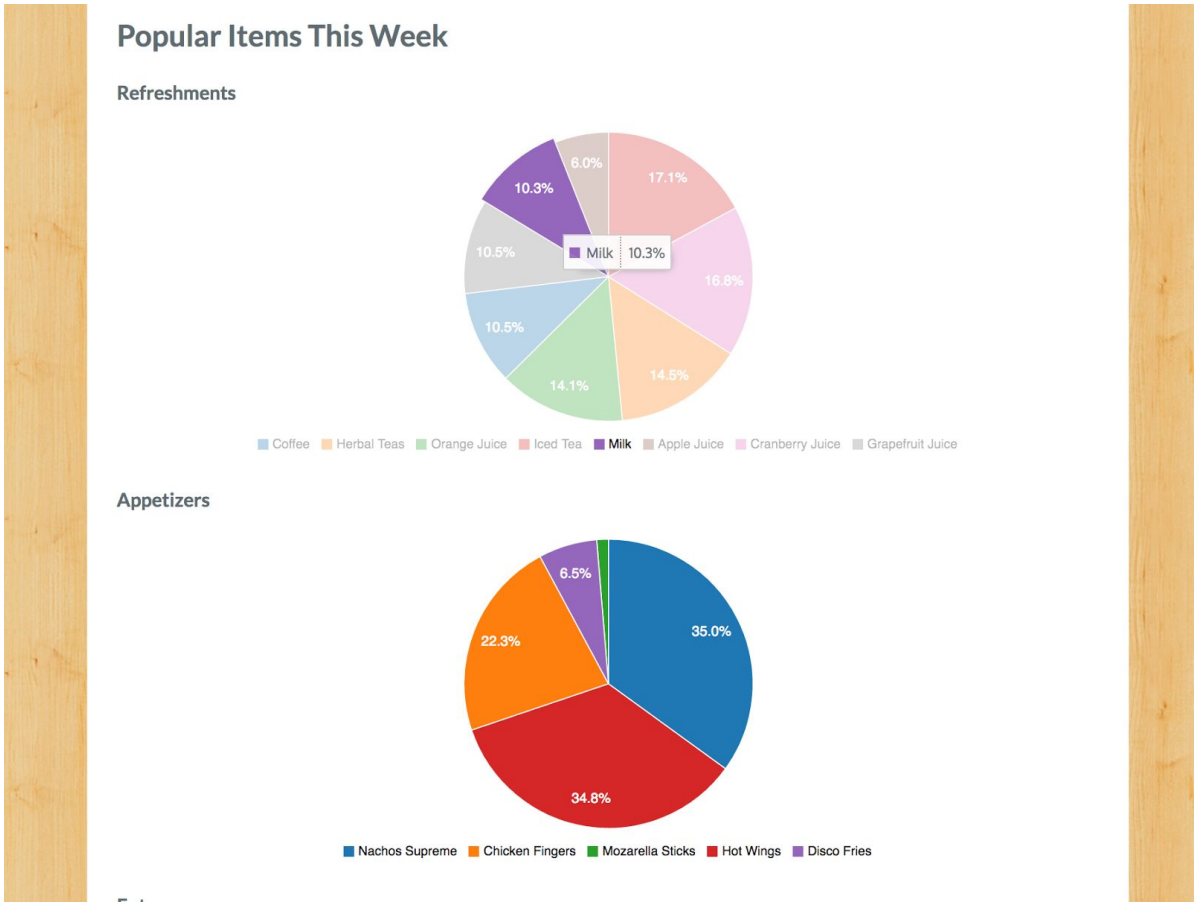


Figure 1.67

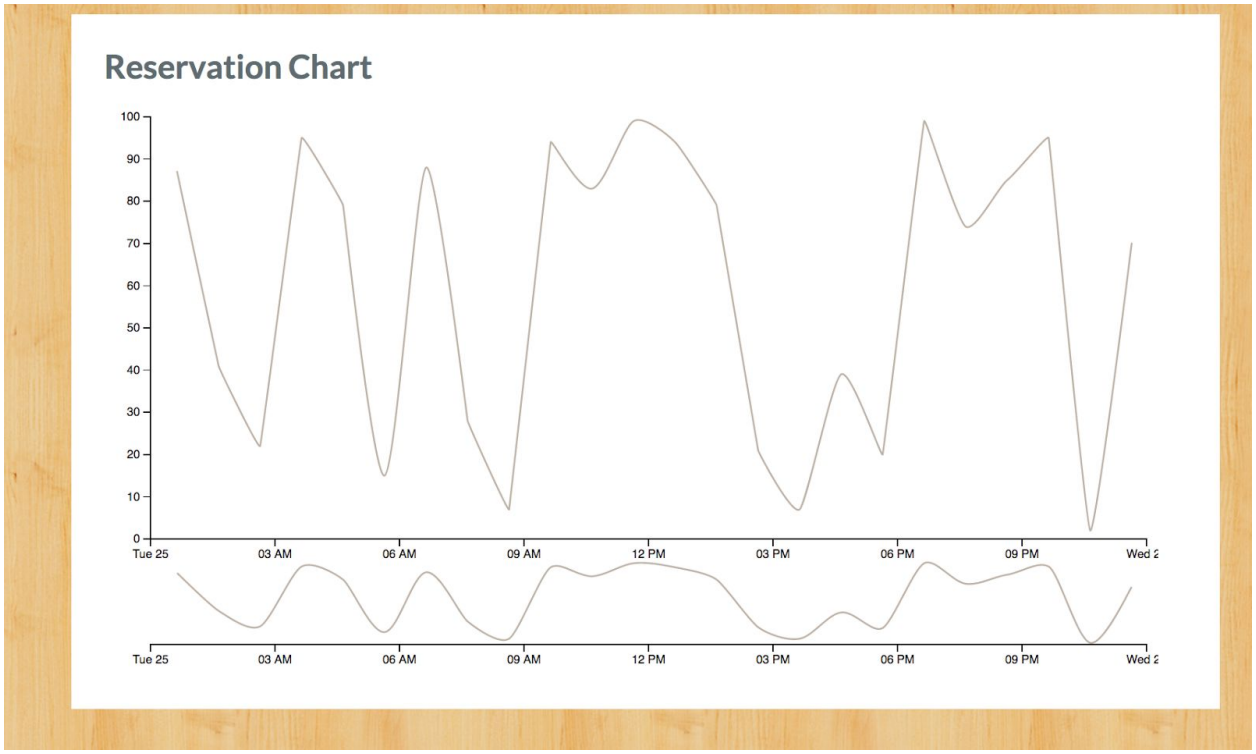


Figure 1.68

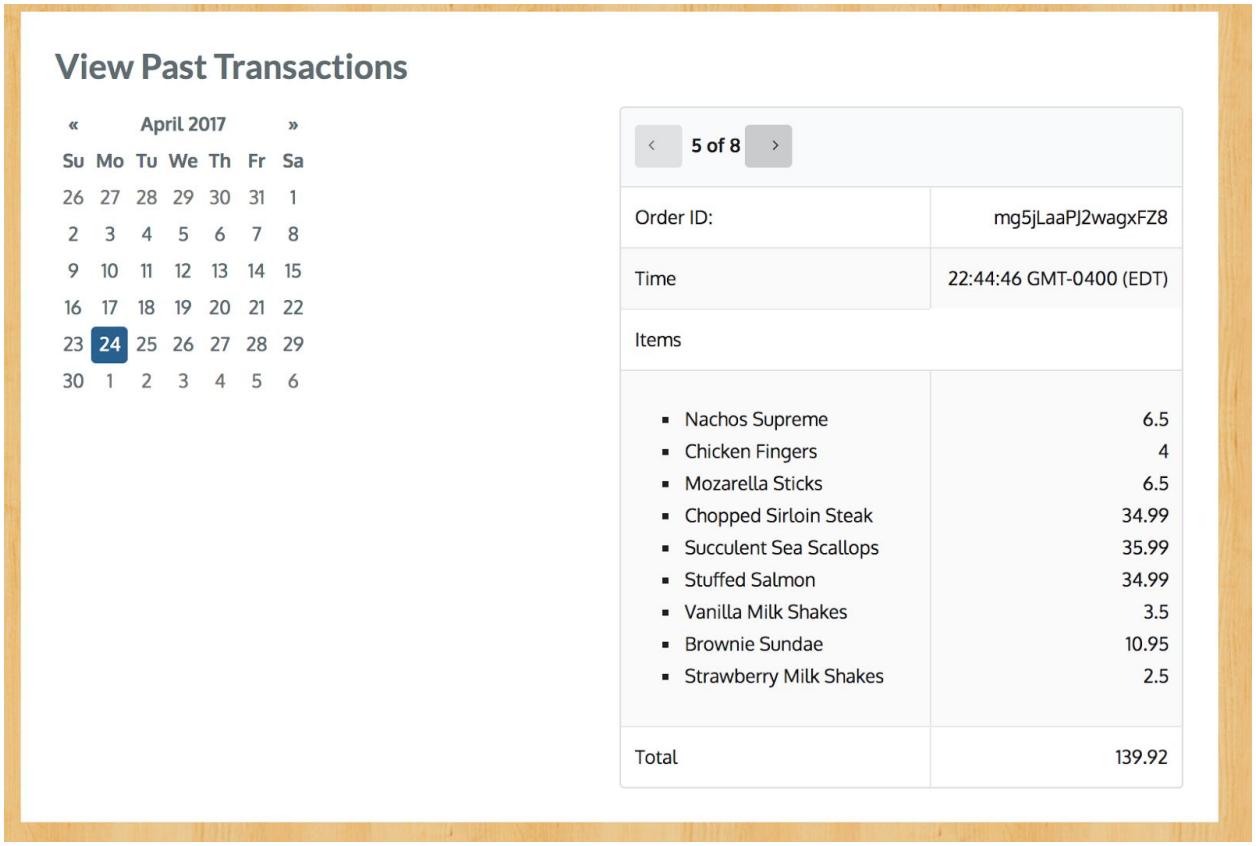


Figure 1.69

The above screenshots show multiple parts of the manager interface. It includes the following tabs: *General, Reservations, Kitchen,* and *Archive*. The *General* tab includes graphs and charts showing the *Time Spent by a Customer in the Restaurant* and a line for the average time spent. As a user you can choose into a certain time period for the graph by clicking and dragging to select a part of the section below the graph. The *General* tab also includes a pie chart showcasing the popular items of the week in each of the food type categories. Figure 1.67 shows that hovering over an item selects it and shows the name of the item and the percentage in the pie chart. This interactive interface displays all the information that the manager needs in a user friendly way.

The *Reservation* tab shows a graph for the number of reservations over the selected time. Similarly to the *Time Spent by a Customer in the Restaurant* graph, you can select the time period you need in the small graph underneath. This allows for easy and quick access to all the information the manager would need.

The *Kitchen* tab has 3 interactive graphs (not pictured) similar to the previous graph described. The graphs include *Cook Time of an Item*, *Usage Of An Ingredient, and Number of Orders.* The first two graphs include a drop down menu showing all the items that are applicable to the graph, so it has all the menu items in the cook time drop down menu and the ingredients in the usage of an ingredient dropdown.

The final tab, *Archive* has a calendar that allows you to click on each date and see the transactions for that date. In the Figure 1.69 above, an example transaction is shown. The archive collects all the items sold along with their prices, time the bill was generated and the order ID of the order. This allows for bookkeeping in the restaurant and the easy to use interface makes it much easier to accomplish the task rather than using pen and paper.

Waiter Interface

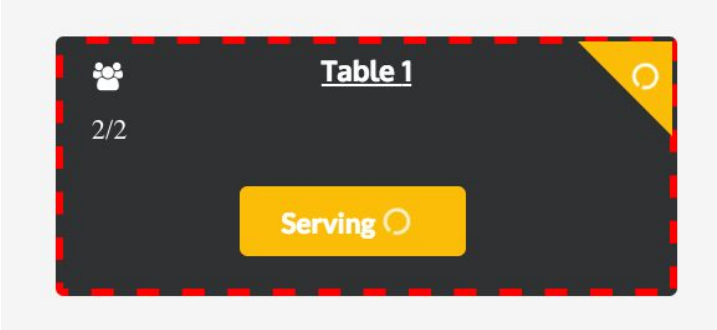


Figure 1.70

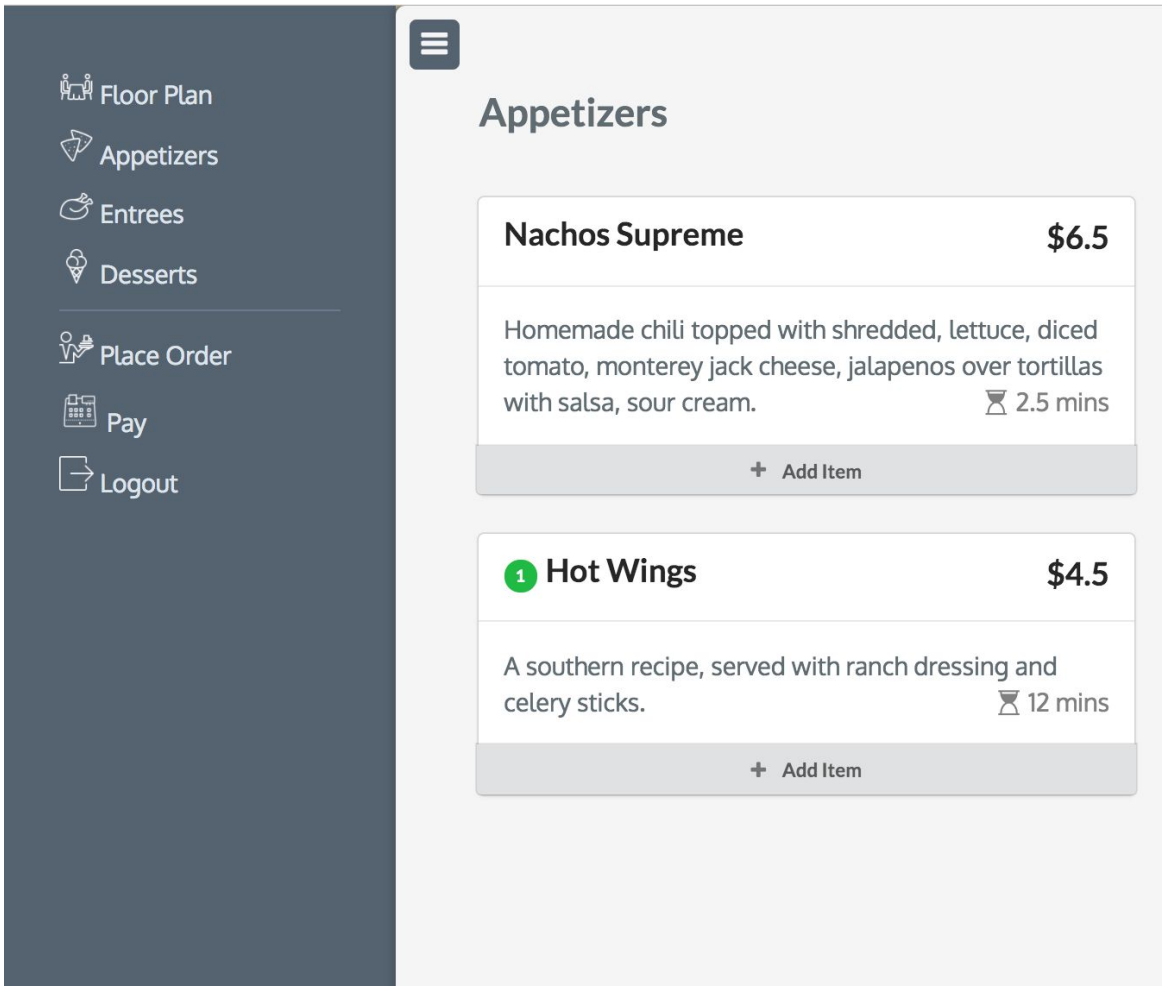


Figure 1.71

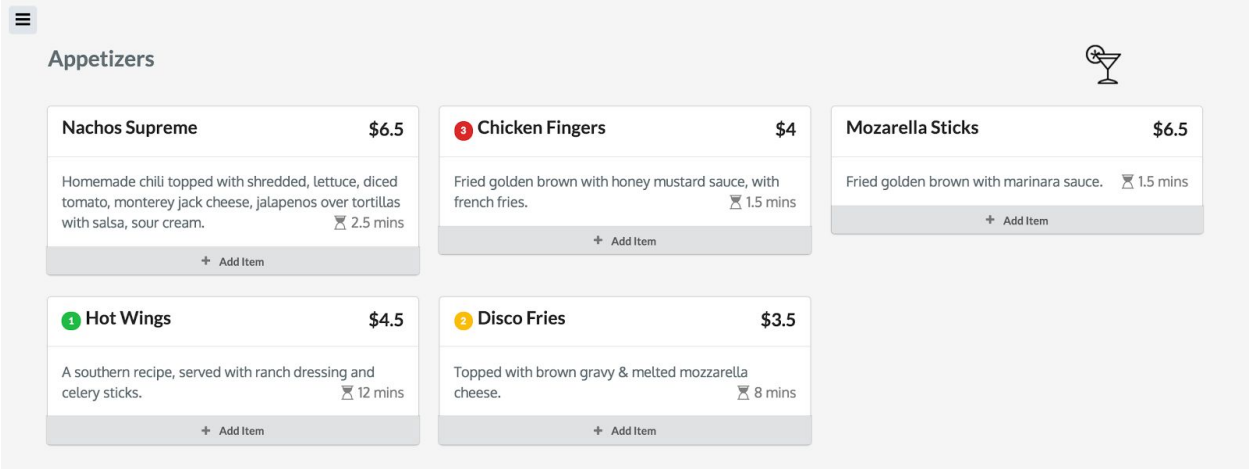


Figure 1.72

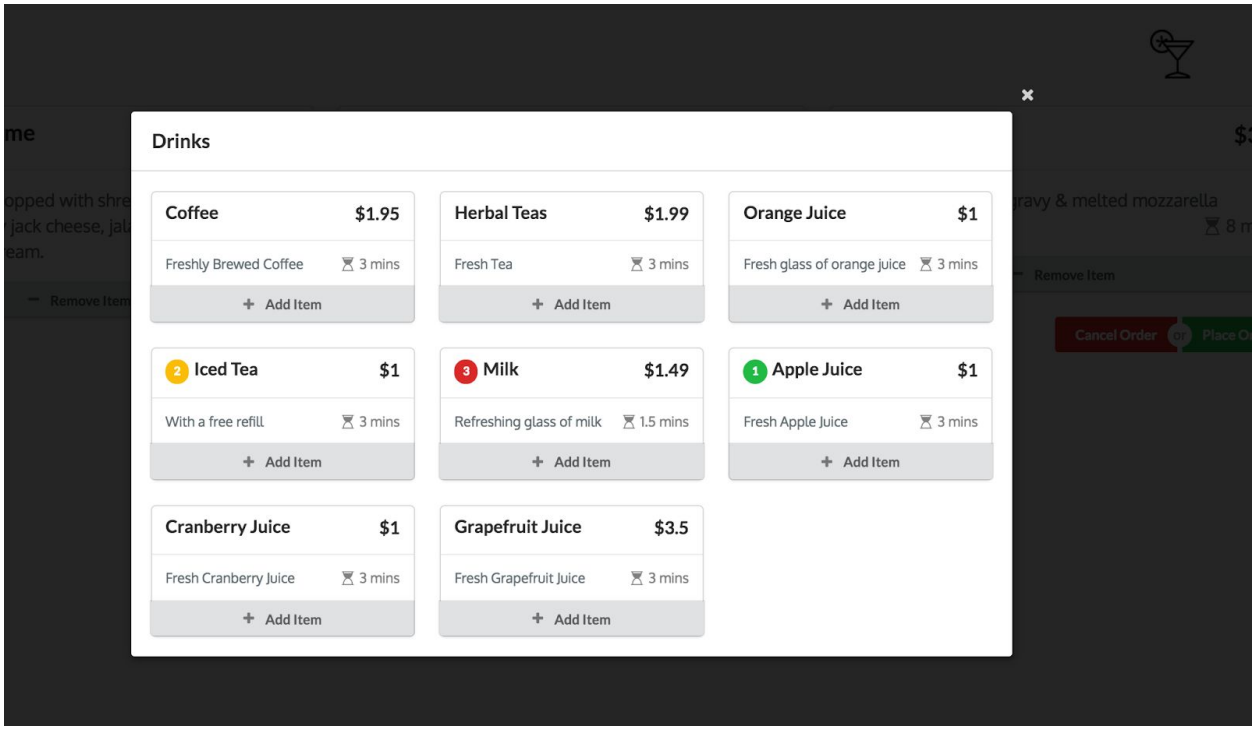


Figure 1.73

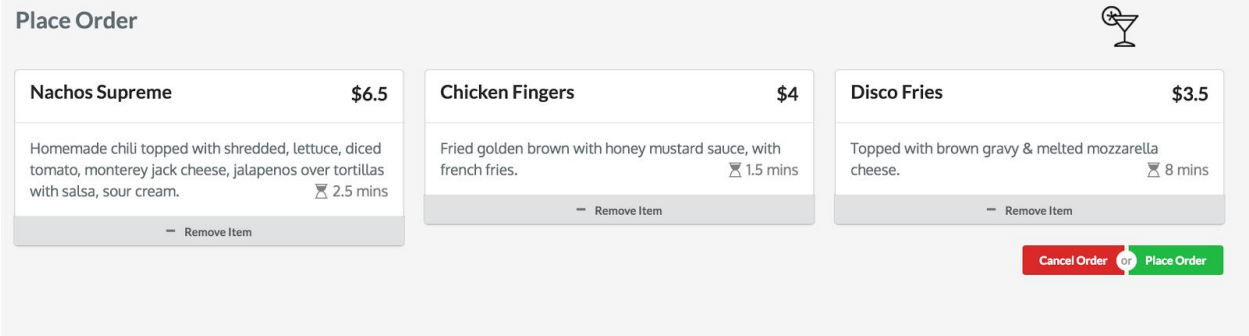


Figure 1.74

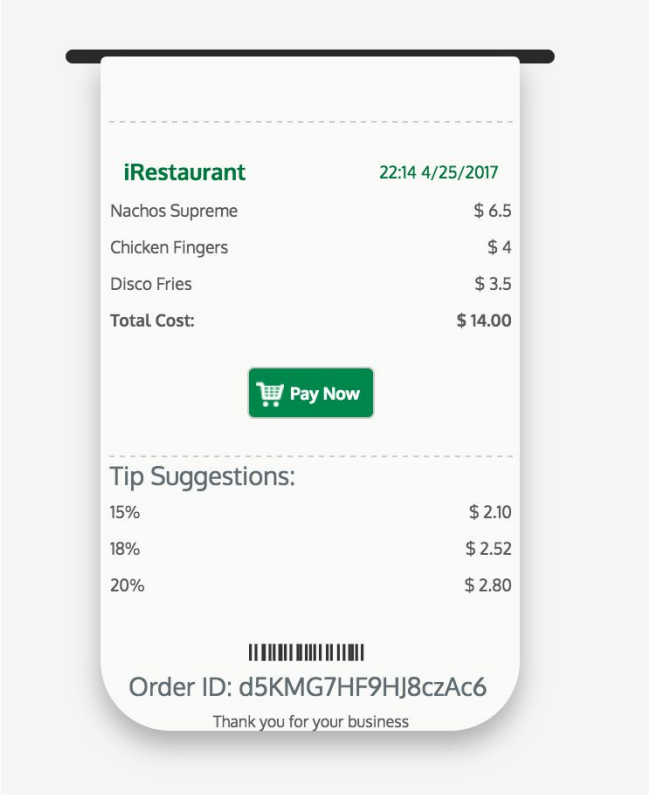


Figure 1.75

When the waiter logs into the application, he is greeted with the floorplan, same as the one shown in Figure 1.64 (floorplan). Here the waiter is able to click on the table name and highlight the table that he is currently serving. This allows an easy way for the waiter to see the tables he is serving. The table gets a dotted red line border, as shown in Figure 1.70.

The waiter can click on the three horizontal bars to open the sidebar, which allows for easy navigation on the tablet that the waiter will be using. The sidebar includes several options as shown in Figure 1.71.

Clicking on *Appetizers* brings up the appetizers menu. On this page, the waiter sees all the appetizer items being sold in the restaurant. The 1, 2, and 3 circular labels represent the most frequently ordered items in each category so it is available at the waiter's fingertips if the customer asks for the popular items at the restaurant. After the customer has decided what to eat, the waiter can pick the items by clicking on the add item button which is made large so it is easy to click on a tablet where the waiter will not be using a precise pointer such as a mouse. The waiter can also click on the drink icon in the top right icon. Clicking on that icon triggers a *modal* that displays the drinks, this is available in all three categories. This also shows the top 3 items color coded. The *modal* is shown in Figure 1.73.

Figure 1.74 shows all the items the waiter added to the order. Now he can place the order to send it to the order queue. Waiter is also able to remove the order if the customer wants to remove the item from the order. After the customer is done with their order, waiter can go and print a bill for them just from his tablet. A sample bill is shown in Figure 1.75, it shows the time the bill was made, the items in the bill along with their prices and total. We also display suggested tips so the customer doesn't waste time figuring out the tip price. A clean user friendly design allows for increased satisfaction.

Login and Register Interface

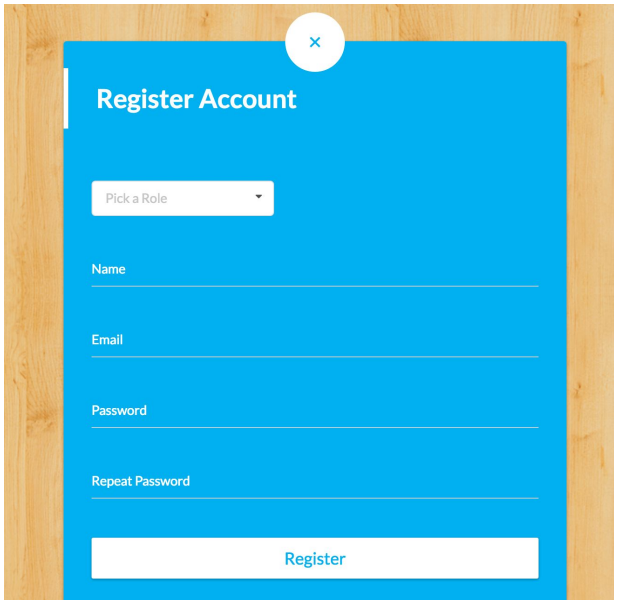


Figure 1.76

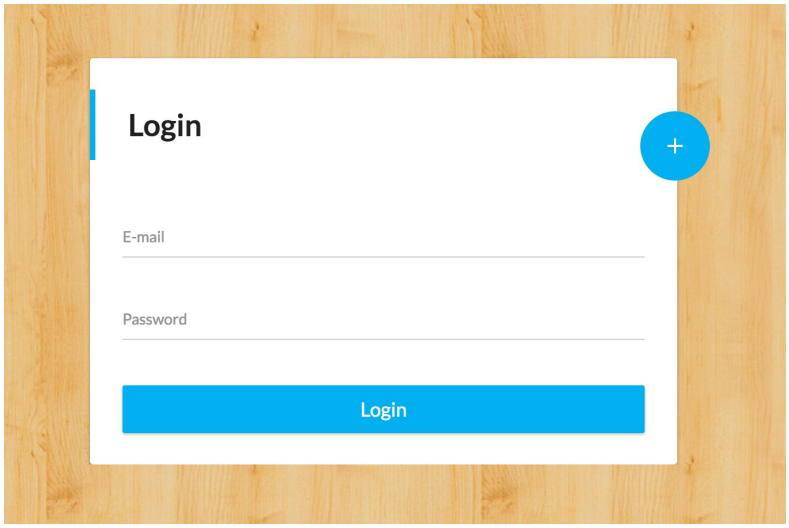


Figure 1.77

The Login and Register information allows the users of our applications to create accounts and login with them so they arrive to their respective pages. To register an account, the users have to select their role from the dropdown menu, it includes the roles such as *Busboy, Waiter, Chef, Manager, and Host*. After selecting a role they input their name, email and password they want to select. After creating an account, they can use that to log in and go to the respective page.

Design of Tests

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.

Table 1.61

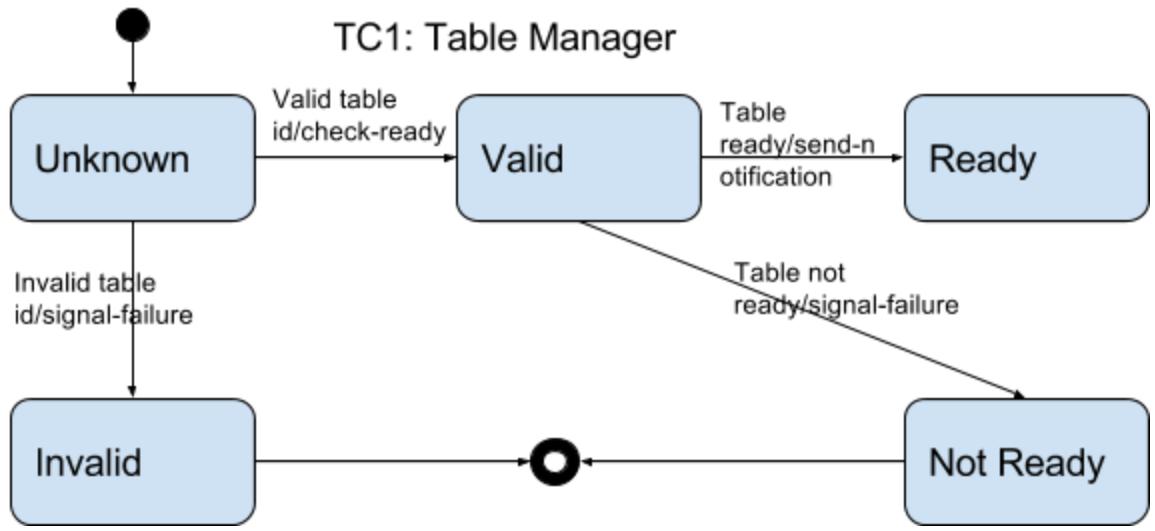


Figure 1.78

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

Table 1.62

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

Table 1.63

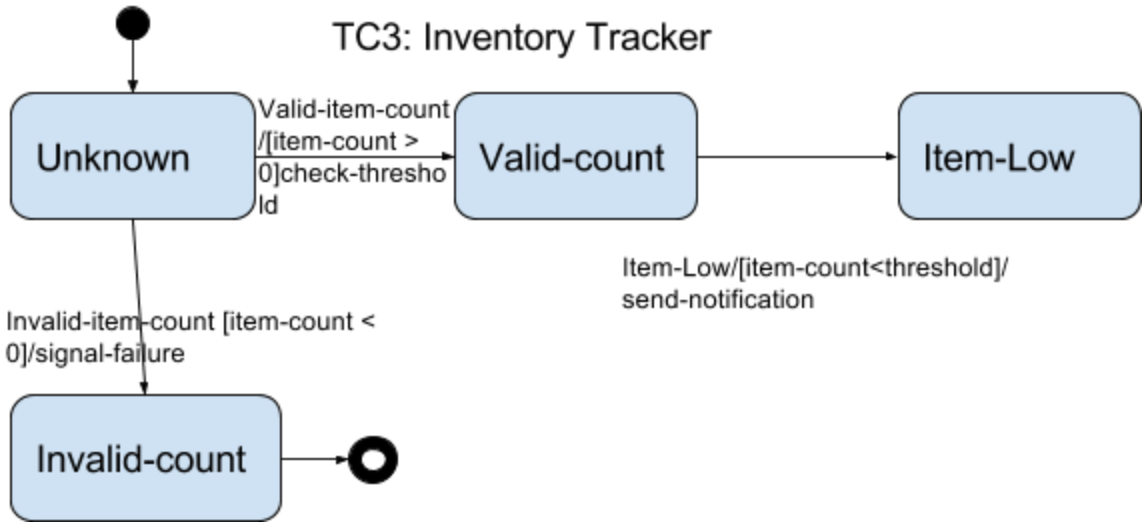


Figure 1.79

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.

Table 1.64

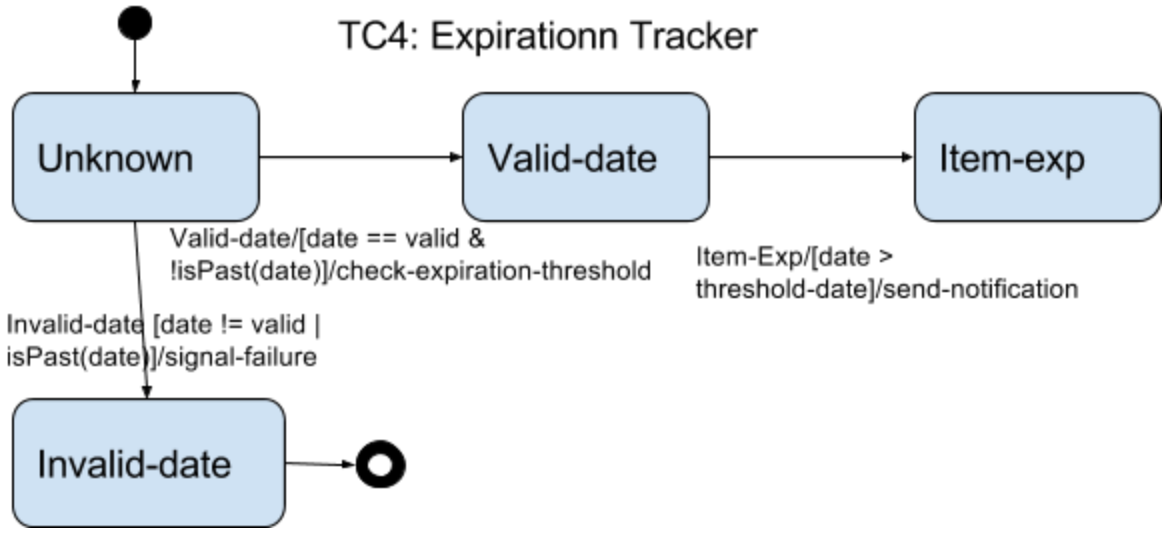


Figure 1.79

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.

Table 1.65

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.

Table 1.66

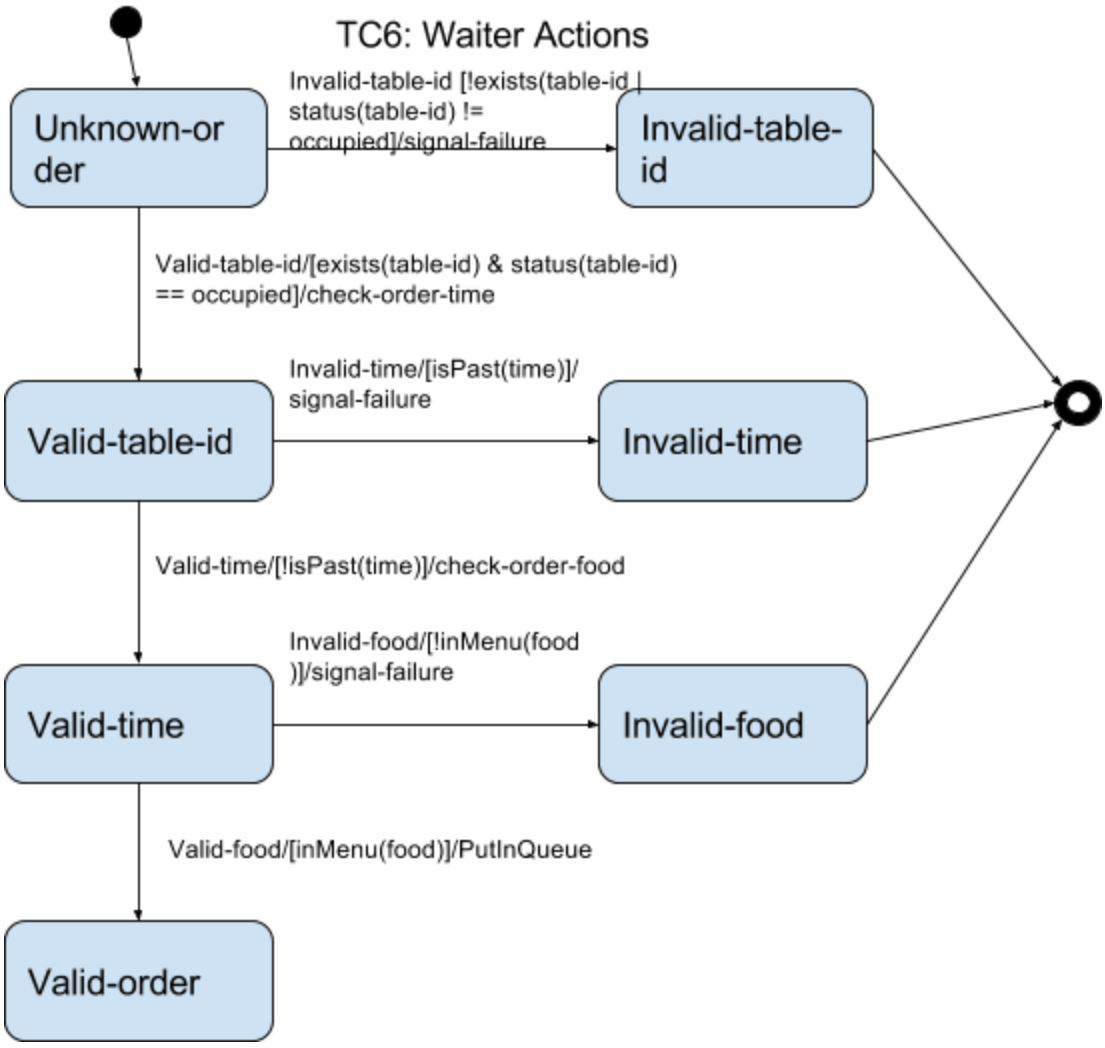


Figure 1.80

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.

Table 1.67

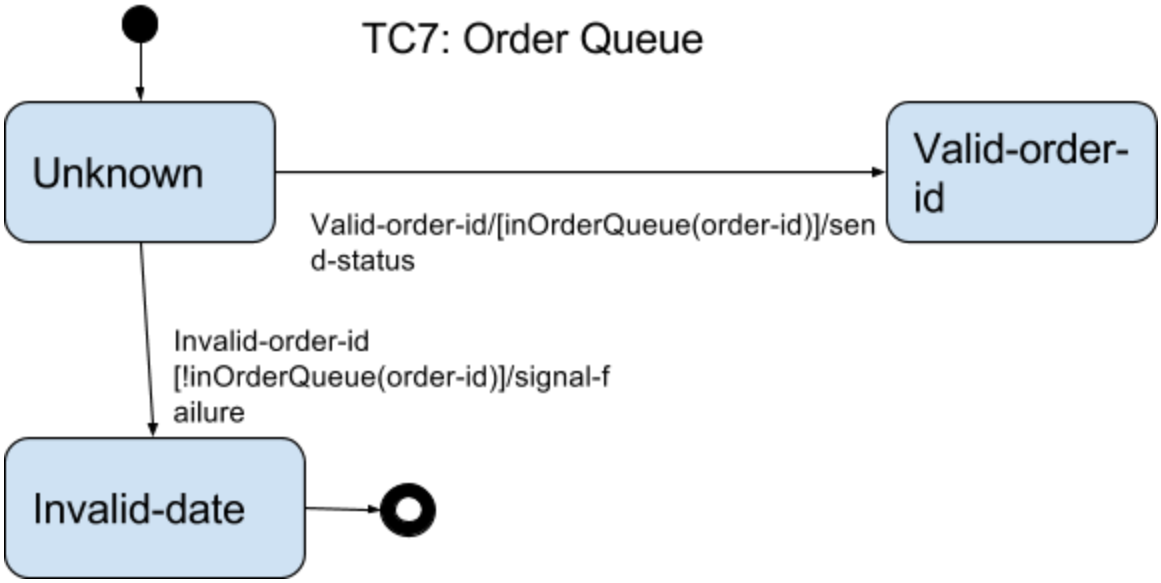


Figure 1.81

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.

Table 1.68

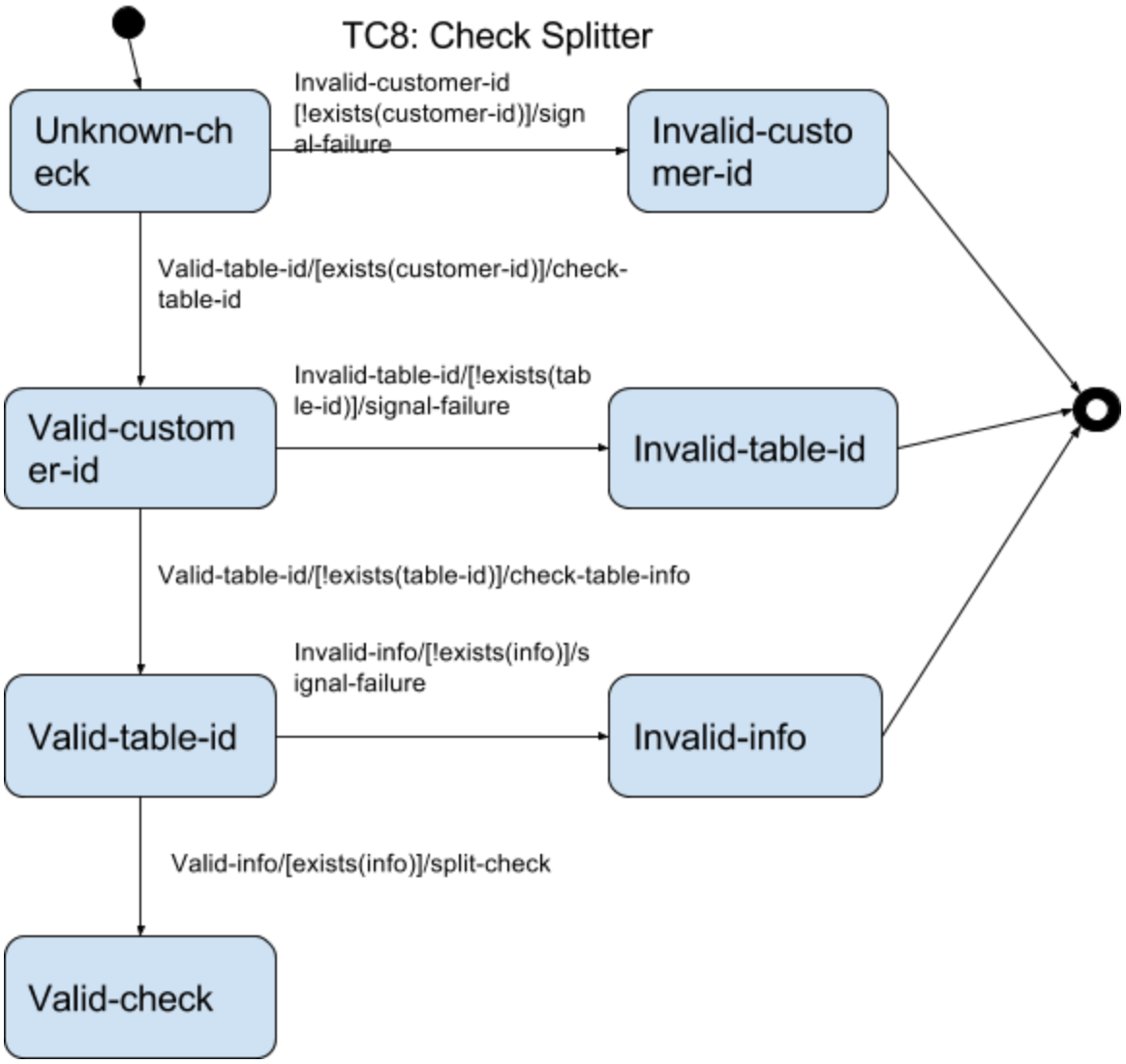


Figure 1.82

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.

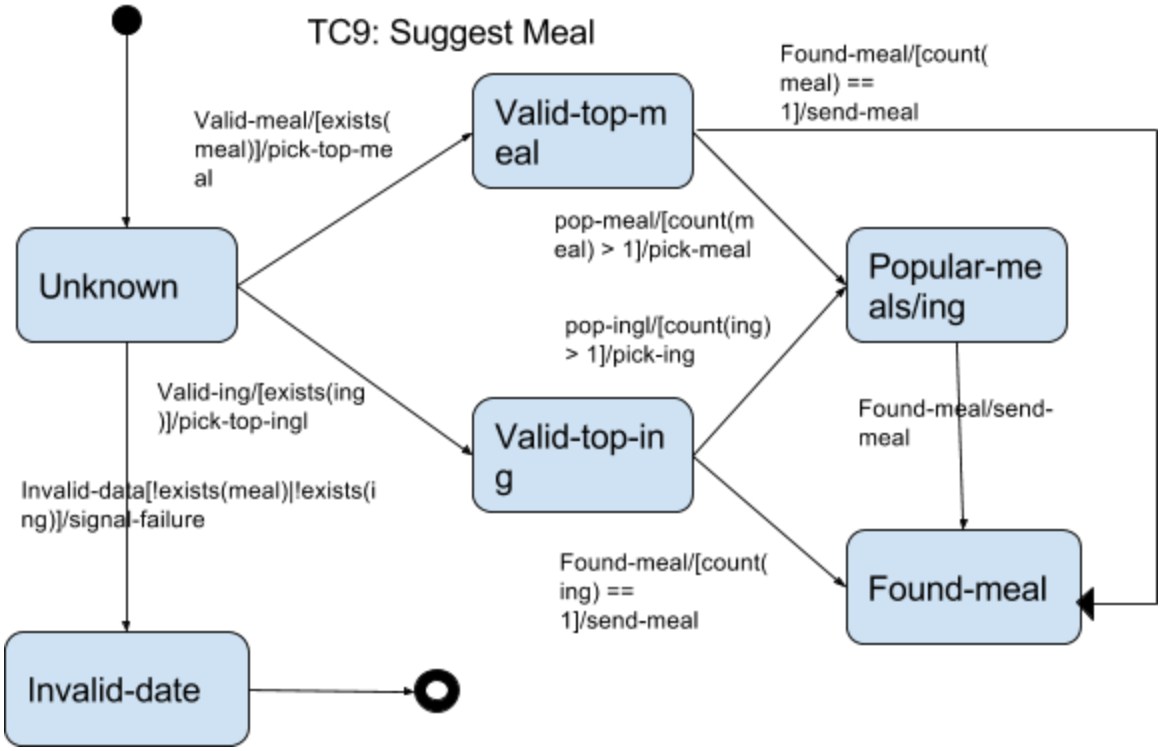


Figure 1.83

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.

Table 1.69

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.

Changes from past reports

Based off of previous reports there have not been major changes in test cases since all the test cases were thought off beforehand so they basically stayed the same.

History of Work, Current Status, and Future 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 different hectic project deadlines 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

*The original plan we had in mind to carry out the numerous tasks involved with this project.
Figure 1.23

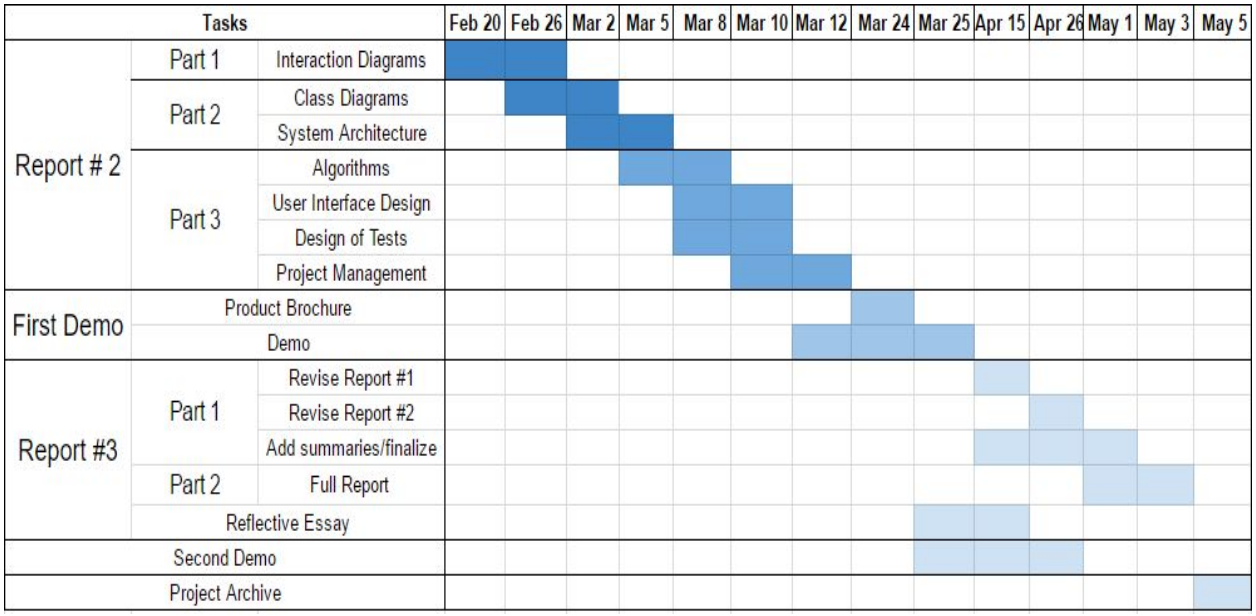


Figure 1.84
Note: The different colors indicate that there is a deadline for submission is between them.

History of Work:

January 23 - January 29:

We first browsed through past projects to determine which ones we found most interesting. Then, we discussed everyone's insight on past projects of interest. We chose the restaurant automation project because we felt we had innovative solutions for this problem domain.

January 30 - February 5

Next, we started work on Report #1, integrating any feedback we received from the proposal. We developed the Customer Statement of Requirements which helped us develop the necessary User Stories to satisfy them.

February 6 - February 12:

During this week, we spent our time analyzing the User Stories and coming up with the corresponding Use Cases. We then divided up into sub groups that would each focus on a separate Use Case; this allowed work to be done in parallel. For each of the fully dressed use

case descriptions, we came up with their corresponding system sequence diagrams and explained in detail how those sequence diagrams behaved. We also started some of the preliminary UI designs to get an idea of what we expect it to look like when we actually implement the project.

February 13 - February 19:

During this week we got closer to finishing up report #1. During this week we did domain analysis on the use cases that we came up with previously and determined their domain concepts, their associations and their respective relationships. We also worked on a traceability matrix that mapped our use cases to their respective domain concepts. We then worked on a few of the mathematical models we would be using for some of our complex solutions. By the end of the week we also determined our plan of work for the rest of the semester.

February 20 - March 28:

This duration of time consisted of starting some of Report #2 and also determining what we would be using to develop our web application. Everyone in the group had different suggestions so we listed all of them down and determined the pros and cons of each suggestion. We decided to use MeteorJS as our full stack web development framework. Unlike most frameworks, MeteorJS does not make distinction between client-side and server-side development. The database is fully accessible from the client and server. This allows for simultaneous development of both the server and client. In addition, this follows the Agile development process because we can divide the project into problems instead of software components, such as client-side and server-side development.

March 29 - April 23:

After completing a successful demo, our next task was to see what other features we could implement in time for Demo #2. While doing all this we took into account the feedback we got for both Report #2 as well as the demo and made sure we kept the corresponding changes in mind for Report #3 and the final demo. This duration of time consisted of further developing our existing features as well as adding more. During this time we also spent time revising our reports for Report # 3 - part 1.

Most of the deadlines evolved as we expected them to. In some situations when we thought that we were behind schedule we would make an effort to put in extra work that week. Doing so would ensure that things were on schedule for the hard deadlines of the deliverables.

Current Status:

The week leading up to the final demo our main task has been to finalize our report 3 - part 1 as well as the main components of our final demo. Our current tasks includes merging all components of the project in time for the demo and also planning the presentation for Demo # 2.

Future Work:

After we finish our demo # 2 we plan on working on our final report (report #3). This will involve modifying much of what we have in report #1 and report #2 to make sure it's consistent with the new changes that we have since the initial planning of the project.

Breakdown of Responsibilities

To make sure that our team is productive and efficient we divided up into three groups of two. Since we have completed an initial design of our system together, our next few weeks would be to build a 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.
Mit Patel and **Raj Patel** are responsible for the Ordering subsystem.
Nill Patel and **Prabhjot Singh** are responsible for Seating and Inventory Management subsystems.

Optimization

Referring to figure 1.65, one of the main benefits of having menu suggestions for the manager is profit. Having menu items in different rankings allows the manager to clearly see which items are doing well and which items are not. Items that are in exclusive lets the manager know that those items are doing the best and should keep them menu. The decision to keep the items on the menu is optimal since it already gives the most profits. Items that are in the low category represent items that are not doing so well. In this case the manager should decide to modify or replace these items. The decision to replace the item with another gives the manager a chance to make more profit with new items. The process can be repeated until the manager is able to get the best menu item that generates more overall gain.

In figure 1.66 and figure 1.67, we showed the general statistics of popular items of the week and the time customers spent in the restaurant. Having this information available can be very valuable to the restaurant manager. For example, with the statistics of time spent by a customer in the restaurant, the manager can view during what time of the day, week, or month customers are staying longer in the restaurant. If the manager sees that customers spend less time during certain hours of the day, he can inform his staff to delay the time food arrives. This way the restaurant would seem more “busy” which would potentially bring in more customers. With the charts of popular items of the week, the manager would be able to know which ingredients they should keep in stock more and which ingredients they wouldn’t need as much.

Moving on to reservation statistics (figure 1.68), the manager is able to see which hours of the day brings in the most reservations. With this information, the manager can allow more tables to be reserved during peak hours of reservation. This would potentially guarantee that customers will come to the restaurant since there are more reservations available during certain times of the day.

Overall, having all of these data trends available to the restaurant manager is very beneficial to him/her. They would be able to optimize the flow of their restaurant and increase profits, leading towards a successful business.

Improvements from Past Projects

In comparison to previous projects, the main intellectual contribution from our team was collecting data to determine trends that would increase the overall efficiency and profitability of the restaurant. We wanted to build an application that would not only automate a restaurant's operations, but would also advise on the financial decisions made by the restaurant operator. For instance, the menu suggestion tool acts as a financial advisor to the manager.[Figure 1.65]

Another example of financial advisory is the statistics page. The calculated statistics for number of reservations over time allows the manager to determine accurate number of tables to allot for reservations instead of guessing the number of tables that may be required. On the other hand, a statistic for the amount of time spent in the restaurant by the customer could be used by the manager to determine the busy hours of the restaurant. Therefore, the manager can make sure that the restaurant is well stocked and staffed. Past projects covered and implemented many great ideas to solve the problem of *restaurant automation*. Essentially, all of these statistics and data analyzation are their own step beyond restaurant automation, that is financial optimization of the restaurant.

All team members managed the project equally!

References

1. "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>.

2. 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>.

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

4. StarUML
   -Program used to create Diagrams

5. Axure
   -Program used to create wireframes for the UI

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

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

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

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